pychoacoustics Documentation

Release 0.5.17

Samuele Carcagno

Mar 24, 2023

Contents

Author Samuele Carcagno <sam.carcagno@gmail.com>

pychoacoustics version 0.5.17, last updated Mar 24, 2023

Disclaimer: This document comes with NO WARRANTY whatsoever of being correct in any of its parts. This document is work in progress.

Contents:

CHAPTER 1

What is pychoacoustics?

pychoacoustics is a software for programming and running experiments in auditory psychophysics (psychoacoustics). The software contains a set of predefined experiments that can be immediately run after installation. Importantly, pychoacoustics is designed to be extensible so that users can add new custom experiments with relative ease. Custom experiments are written in Python, a programming language renowned for its clarity and ease of use. The application is divided in two graphical windows a) the "response box", shown in Figure *The pychoacoustics response box*, with which listeners interact during the experiment, and b) the control window, shown in Figure *The pychoacoustics control window*, that contains a series of widgets (choosers, text fields and buttons) that are used by the experimenter to set all of the relevant experimental parameters which can also be stored and later reloaded into the application.

Some of the main features of pychoacoustics are that:

- pychoacoustics lets you create complex auditory experiments with relative ease
- experimental variables can be easily manipulated, stored, and retrieved using a graphical user interface
- pychoacoustics takes care of stimulus presentation, including setting up interval lights and response buttons
- pychoacoustics takes care of collecting responses, computing summary measures such as threshold estimates, and *d'*, and storing them as CSV files so that they can be readily visualized and processed using statistics programs (R, SPSS, etc…), or spreadsheet applications (Excel, Libreoffice Calc, etc…)
- pychoacoustics supports the most commonly used stimulus presentation procedures, such as the transformed up-down, same-different, and ABX procedures. It also supports less known and cutting-edge procedures such as the update maximum likelihood and the PSI+ and PSI-marginal procedures
- for many procedures pychoacoustics provides graphical summaries of the results
- pychoacoustics stores all important information, including names and values of experimental parameters, timestamps, participant identification codes (if provided), version of the software used to run the

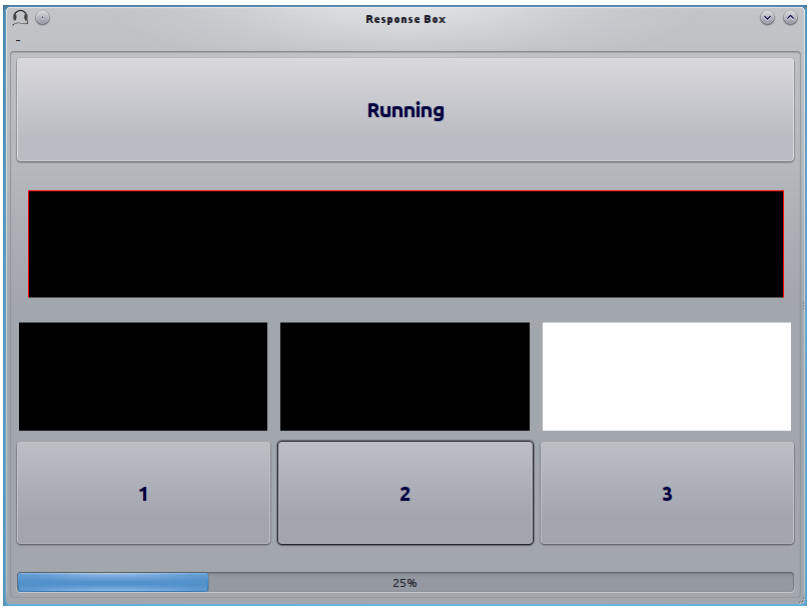

Fig. 1: The pychoacoustics response box

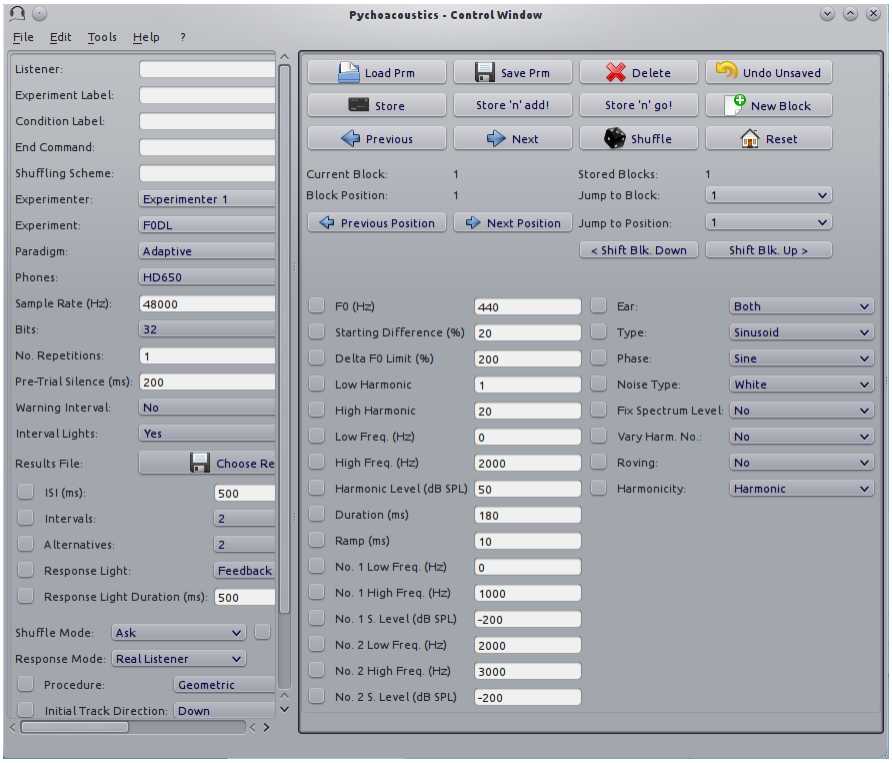

Fig. 2: The pychoacoustics control window

experiment, etc. . . , so that no important information is accidentally lost

- pychoacoustics allows you to present standardized written instructions to participants at the beginning of the experiment, or the beginning of specific blocks of trials. Participants can also get an idea of how much of an experiment they have completed through optional progress bars.
- pychoacoustics includes *sndlib.py* a library that lets you easily generate experimental stimuli commonly used in psychoacoustics experiments (pure tones, complex tones, noise, etc. . .)

I started writing `pychoacoustics` for fun and for the sake of learning around 2008 while doing my PhD under the supervision of Professor Chris Plack at Lancaster University. At that time we were using in the lab a MATLAB program called the "Earlab" written by Professor Plack. `pychoacoustics` has been greatly influenced and inspired by the "Earlab". For this reason, as well as for the time he dedicated to teach me audio programming, I am greatly indebted to Professor Plack.

CHAPTER 2

Installation

pychoacoustics has been successfully installed and used on Linux, Windows, and Mac platforms. Installation instructions for each operating system are provided below.

2.1 Installation on Linux

For Debian and Ubuntu LTS releases there are apt repositories that can be used to install and update pychoacoustics. For other Linux distributions pychoacoustics has to be installed from source (see Section *Installation from source*).

2.1.1 Installation on Debian

Binary packages for the Debian amd64 architecture are hosted on bintray. To install pychoacoustics first install the apt-transport-https package if it is not already installed:

```
sudo apt-get install apt-transport-https
```

then add one of the following lines to /etc/apt/sources.list depending on your Debian version: For Stretch (stable):

```
deb https://dl.bintray.com/sam81/hearinglab stretch main
```

For Jessie (oldstable):

```
deb https://dl.bintray.com/sam81/hearinglab jessie main
```

Download the key with which the repository is signed and add it to the apt keyring:

```
wget -qO - https://bintray.com/user/downloadSubjectPublicKey?username=bintray␣
↪| sudo apt-key add -
```

Refresh the package database and install the package:

```
sudo apt-get update
sudo apt-get install pychoacoustics
```

2.1.2 Installation on Ubuntu LTS Releases

Binary packages for Ubuntu Long Term Support (LTS) releases are hosted on Launchpad. To install pychoacoustics run the following commands:

```
sudo add-apt-repository ppa:samuele-carcagno/hearinglab
sudo apt-get update
sudo apt-get install pychoacoustics
```

2.1.3 Installation on Windows and MacOS

There are experimental binary installers for Windows and MacOS on the downloads page:

http://samcarcagno.altervista.org/pychoacoustics/pychoacoustics.html

these are OK if you just want to try out pychoacoustics to see its look and feel. However, if you want to use pychoacoustics in research it's recommended that for Windows and MacOS you install from source (see below). Producing the binaries for these OSs is time consuming, so the binary installers may be updated less often with the latest bug fixes.

2.1.4 Installation from source

`pychoacoustics` depends on Python and a handful of Python modules. There are two ways to obtain these dependencies. One is to install Python and all the dependencies "manually" (that is one by one). The other (and easier) way is to install a Python distribution that comes with a bundle of pre-installed modules. These include:

- Anaconda: **'https://www.continuum.io/downloads https://www.continuum.io/downloads>'_**
- Pyzo: http://www.pyzo.org/
- WinPython (Windows only): **'http://winpython.github.io/ http://winpython.github.io/>'_**

Step by step instructions to install `pychoacoustics` on Windows with WinPython are provided below. Although these instructions are specific to Windows the installation steps are similar on Mac OS X and Linux systems. If you get stuck at some point don't hesitate to get in touch <sam.carcagno@gmail.com>.

Install with WinPython on Windows

Download the latest version on [WinPython](https://winpython.github.io/) and unpack it to a folder of your choice. Download the Windows source package of *pychoacoustics*. Open the DOS command prompt and change directory to the folder where you unpacked `WinPython`, which should contain the *python.exe* executable. For example:

```
cd C:\Users\audiolab\Desktop\WinPython
```

once you're in the WinPython folder, you can instruct the Pyzo Python interpreter to run *pychoacoustics* by calling python.exe followed by the path where the *pychoacoustics* main file is located. For example:

```
python.exe "C:\Users\audiolab\Desktop\pychoacoustics-0.4.8\pychoacoustics.pyw"
```

Currently there is not an application launcher. There is, however, a file called `pychoacoustics-launcher.bat` inside the *scripts* folder of the source distribution of *pychoacoustics* that after some modifications can be used as a launcher. The content of the file is the following:

```
C:\Python32\python "C:\Python32\site-packages\pychoacoustics.pyw"
%1 %2 %3 %4 %5 %6 %7 %8
```

The first statement `C:\Python32\python` is the path to the Python executable. The second statement is the path to the main file of the `pychoacoustics` app. You simply need to replace those two statements to reflect the Python installation on your system. Following the example above, you would change the contents of the file to:

```
"C:\Users\audiolab\Desktop\WinPython\python.exe"
↪"C:\Users\audiolab\Desktop\pychoacoustics-pyside-0.4.8\pychoacoustics.pyw"
%1 %2 %3 %4 %5 %6 %7 %8
```

You can place the `.bat` launcher wherever you want, for example on your `Desktop` folder. Simply double click on it, and `pychoacoustics` should start.

2.1.5 "Manual" Installation from Source

`pychoacoustics` depends on the installation of a handful of other Python modules:

- Python (version 3) http://www.python.org/
- numpy http://sourceforge.net/projects/numpy/files/
- scipy http://sourceforge.net/projects/scipy/files/

additionally it is necessary to install one of the modules providing Python bindings to the Qt widgets toolkit. There are three parallel versions of `pychoacoustics` that support the major modules providing Python bindings to Qt (PyQt5, PyQt4, and PySide). You need to install only one of these modules, and use the corresponding version of `pychoacoustics`

- PyQt5 https://riverbankcomputing.com/software/pyqt/download5

these programs need to be installed manually. Once these programs are installed you can proceed with the installtion of `pychoacoustics`:

```
python3 setup.py install
```

you can then invoke `pychoacoustics` from a terminal by typing the command

```
pychoacoustics.pyw
```

There are two additional optional dependencies:

- matplotlib http://matplotlib.org/
- pandas http://pandas.pydata.org/

if matplotlib and pandas are installed pychoacoustics can generate graphical summaries of the results of an experimental session.

Install Python and the Dependencies manually on Windows

Please, note that you will need Python version 3 or above to run *pychoacoustics*.

To install the dependencies, download them from their respective websites. Make sure that you pick versions compatible with your architecture (64 or 32 bits), and compatible with you Python version.

After installing the dependencies, it is recommended to add the directory where the Python executable resides to the system `PATH`. In this way you can call `python` from a `DOS` shell by simply typing its name, rather than typing the full path to the Python executable.

By default `python` is installed in `C:`. The name of the Python directory depends on its version number, for example, if you installed Python version 3.2, the python directory will be `C:\Python32`. To add this directory to the system path go to `My Computer` and click `Properties`, then click `Advanced System Settings`. In the `System Properties` window click `Environment Variables`. There you will find an entry called `Path`. Select it and click `Edit`. Be careful not to remove any of the entries that are already written there because it could corrupt your system. Simply append the name of the full path of the folder where Python is installed, at the end of the other entries.

To run `pychoacoustics`, unpack the `pychoacoustics .zip` file containing the source code. Open a `DOS` shell, `cd` to the directory where you unzipped pychoacoustics and launch it with the following command:

```
python pychoacoustics.pyw
```

Currently there is not an application launcher. There is, however, a file called `pychoacoustics-launcher.bat` inside the *scripts* folder of the source distribution of *pychoacoustics* that after some modifications can be used as a launcher. The content of the file is the following:

```
C:\Python32\python "C:\Python32\site-packages\pychoacoustics.pyw"
%1 %2 %3 %4 %5 %6 %7 %8
```

The first statement `C:\Python32\python` is the path to the Python executable. The second statement is the path to the main file of the `pychoacoustics` app. You simply need to replace those two statements to reflect the Python installation on your system. You can place the `.bat` launcher wherever you want, for example on your `Desktop` folder. Simply double click on it, and `pychoacoustics` should start.

CHAPTER 3

Graphical User Interface

The user interface is divided into two windows: the "Control Window" and the "Response Box". The "Control Window" is used to set the experimental parameters, while the "Response Box" is the interface with which listeners interact.

3.1 Quickstart

When `pychoacoustics` is launched, the "Control Window" displays the default parameters for the "Audiogram" experiment. You can select another experiment using the "Experiment" drop-down menu, and edit any of the parameter fields you want to modify. Once you're satisfied with the parameters, you can store them by pressing the "Store" button. This stores one experimental block with the chosen parameters. At this point you can either start running the experiment by pressing the "Start" button on the "Response Box", or you can add more experimental blocks by clicking on the "New Block" button.

To save the parameters to a file click on the "Save Prm" button. Parameter files that have been saved in this way can be later loaded into the program by using the "Load Prm" button.

To save the results of your experiment to a file, click on the "Save Results" button. If you have forgotten to specify a results file in this way, `pychoacoustics` will save the results in a file called `test.txt` in the working directory.

3.2 The Control Window

The control window contains a set of widgets to manage the setup of the experiments, running the experiments, processing results files and managing application preferences. Some of the widgets are general, and some of them are specific either to a given paradigm (e.g. adaptive vs constant stimuli paradigm) or to a given experiment.

In the next section the function of these widgets will be explained, starting with the widgets that are general to all experiments and paradigms.

3.2.1 General Widgets (left panel)

- **Listener** This is simply a label that you can use to identify the listener who is being tested. This label will be written in the header of the results file.
- **Experiment Label**. This is a label to identify the experiment you are running. This label will be written in the header of the results file.
- **Session** This is a label to identify the experimental session, it can be a number or a string. This label will be written in the header of the results file.
- **Condition Label** This is a label to identify the experimental condition of the current block of trials. It is optional, but it may be useful when sorting the experimental results (see *Tabular Results Files*).
- **Task Label** This label will be shown in the response box to tell the listener which task s/he's doing. Useful in case different tasks are mixed within a session.
- **Instructions** This box allows to give task instructions to the listener. If the block of trials occurs at a block position in which task instructions are set to be shown (see "Show Instructions At BP" field below), the text written in this box will will be shown to listeners at the beginning of the block of trials (see *Displaying Task Instructions* for more info).
- **Show Instructions At BP** Indicate the block positions (see *Block Presentation Position* for a definition of block positions) at which the instructions should be shown to the listener. The block positions have to be indicated by a list of numbers separated by commas (see *Displaying Task Instructions* for more info).
- **End Command** Here you can write an operating system command (e.g. a bash command on Unix systems or a DOS command on Windows systems) to be performed at the end of the experimental session. This could be used to run a custom script to analyse the result files, make a backup of the results files or other purposes. There are some variables (such as the name of the results file) that can be accessed with a special string. These are listed in Section *OS Commands* Table tab-pycho_variables Please refer to that section for further info on how to use them.
- **Shuffling Scheme** By default when you click the "Shuffle" button, `pychoacoustics` randomly shuffles all blocks, here you can specify different shuffling schemes (e.g. shuffle the first four blocks among themselves and the last four blocks among themselves). Please refer to Section *Block Presentation Position* for more details.
- **Proc. Res.** Process the "block summary" file at the end of the experimental session in order to obtain a "session summary" file (see *Result Files*).
- **Proc. Res. Table** Process the "table block summary" file at the end of the experimental session in order to obtain a "table session summary" file (see *Result Files*).
- **Plot** Plot the results at the end of the experimental session. This function is available only if both matplotlib and pandas are installed. Plots are available only for some experimental paradigms.

- **PDF Plot** Create a PDF file plotting the results at the end of the experimental session. This function is available only if both matplotlib and pandas are installed. Plots are available only for some experimental paradigms.
- **Experimenter** Here you can select one of the experimenters listed in the experimenter database. Please refer to Section *Edit Experimenters Dialog* for further info on the experimenter database and how it can be used.
- **Experiment** Selects the experiment for the current block.
- **Paradigm** Selects the paradigm (e.g. transformed up-down, constant, etc...) for the current block. The list of paradigms available depends on the experiment that is selected.
- **Phones** Choose from one of the phone models stored in the phones database. Please, refer to Section *Edit Phones Dialog* for further info on how to enter phones and calibration values in the database.
- **Sample Rate (Hz)** Set the sampling rate of the sounds to be played. Any value can be entered in the text fields. However, you should enter a value that is supported by your soundcard. A value that is not supported by your souncard may lead to issues, although it's more likely that your computer will perform an automatic sample rate conversion to a supported sample rate.
- **Bits** Set the bit depth that `pychoacoustics` uses to store sounds to a wav file or play them. Currently values of 16 and 32 bits are supported. A value of 32 bits can be used for 24-bit soundcards. Notice that achieving 24-bit output requires both a 24-bit souncard and a play command that can output 24-bit sounds. Therefore selecting a value of 32 bits here does not guarantee 24-bit playback even if you have a 24-bit souncard. Please, refere to Section *Sound Output* for further information on this issue.
- **No. Repetitions** Set the number of times the sequence of blocks stored in memory should be repeated. If the "Shuffle Mode" (see below) is set to "auto", each time a new repetition starts the block positions will be shuffled. If the "Shuffle Mode" is set to "Ask", each time a new repetition starts the user will be asked if s/he wants to shuffle the block positions. The "Reset" button resets the number of repetitions completed by the listener to zero.
- **Pre-Trial Silence (ms)** Set a silent time interval before the start of each trial. Useful to avoid that a new trial starts immediately after the listener has given his/her response.
- **Warning Interval** Choose whether to present a warning light at the beginning of each trial.
- **Warning Interval Duration (ms)** Sets the duration of the warning interval light. This widget is shown only if the warning interval chooser is set to "Yes".
- **Warning Interval ISI (ms)** Sets the duration of the silent interval between the end of warning interval and the start of the first observation interval. This widget is shown only if the warning interval chooser is set to "Yes".
- **Response Light** Set the kind of feedback to give to participants at the end of each trial. "Feedback" will give feedback (e.g. flash a green, for a correct response, or red, for an incorrect response light. "Neutral" will acknowledge that a responses has been given, but will not give feedback as to whether the response was correct (e.g. flash a white light). "None" will not give any feedback or acknowledgment that a response has been given. (e.g. no light will be flashed, there will nonetheless be a silent interval equal to the response light duration, see below).

- **Response Light Type** Determines the mode in which feedback or acknoledgment of listener responses is given. If "Light", a colored light will be flashed (e.g. a green light to indicate a correct response, and a red light to indicate an incorrect response). If "Text", a string will be presented (e.g. "Correct!" for a correct response, and "Incorrect!" for an incorrect response. If "Smiley", a smiley will be painted in the response light box. Combinations of these three basic feedback presentation modes are also possible.
- **Response Light Duration (ms)** Set the duration of the response light.
- **Results File** Select a file for saving the results. Selecting an existing file will never overwrite its content, it will simply append the new results to its content. If no file is selected, the results will be saved in a file called `test.txt` in the current working directory. You can select a file to save the results even after you have started a block of trials, the results get written to the file only at the end of the block.
- **Shuffle Mode** If the "Shuffle Mode" is "auto", the block presentation positions will be automatically shuffled at the beginning of a series of blocks. If the "Shuffle Mode" is "Ask", at the beginning of a series of blocks the user will be asked if the block presentation positions should be shuffled or not. If the "Shuffle Mode" is "No", the block presentation positions will not be automatically shuffled at the beginning of a series of blocks. See Section *Block Presentation Position* for further information on shuffling the block presentation positions.
- **Response Mode** When "Real Listener" is selected, `pychoacoustics` waits for responses from a human listener. When "Automatic" is selected the program will give responses by itself with a certain percentage correct, that can be specified in the "Percent Correct (%)" text field. This mode is mostly useful for debugging purposes, however it can also be used for experiments in which the participants are passively listening to the stimuli (e.g. some neuroimaging experiments that record cerebral responses rather than behavioural responses). In "Simulated Listener" mode `pychoacoustics` will give responses on the bases of an auditory model. This model needs to be specified in the experiment file, the "Simulated Listener" mode provides just a hook to redirect the control flow to your model. When the "Psychometric" listener mode is selected responses are given automatically according to the shape of a psychometric function (see boxes below for specifying the psychometric function shape). The "Psychometric" listener mode works only for adaptive paradigms (e.g. tranformed up-down, weighted up-down, PEST, UML, PSI). Please, refer to Section *Response Mode* for more information.
- **Psychometric Listener Function** The function family for the psychometric listener. Currently supported functions are "Logistic", "Gaussian" (normal), "Gumbel", and "Weibull".
- **Psychometric Listener Function Fit** Whether the psychometric function is fitted on "Linear" or "Logarithmic" coordinates. With the transformed up-down, PEST, and weighted up-down paradigms you should choose "Logarithmic" if you're using a geometric adaptive procedure. With the PSI and UML paradigms you should choose "Logarithmic" if your the stimulus scaling is set to "Logarithmic".
- **Psychometric Listener Midpoint** The midpoint of the psychometric function, that is the middle point between chance performance and maximum performance. For a two-alternative forced choice task this will correspond to the point at which the listener achieves a 75% correct performance.
- **Psychometric Listener Slope** The slope of the psychometric function. Please note that slopes measured with different psychometric function families (e.g. "Logistic" and "Gaussian") are not directly comparable.
- **Psychometric Listener Lapse** The lapse rate of the psychometric listener.

- **Save psychometric listener data** Save the psychometric listener data to a text file. The first column of the saved file corresponds to the probability of a correct response. The second column corresponds to the stimulus value at which the psychometric listener achieves that probability of a correct response.
- **Plot psychometric listener function** Plot the psychometric function defined for the psychometric listener.

3.2.2 Additional Widgets (left panel)

The following widgets are present only in some experiments:

- **ISI (ms)** Inter-stimulus silent interval, in ms.
- **Intervals** Set the number of observation intervals.
- **Alternatives** Set the number of response alternatives.
- **Alternated (AB) Reps.** This setting makes it possible to present stimuli with the ABAB AAAA paradigm (see [KingEtAl2013]). If the value is set to zero, then on each interval only one stimulus will be presented, either the standard (A), or the comparison (B) stimulus. If the value is set to one, then the correct interval will contain the an alternation of the standard and comparison stimuli (AB), while the incorrect interval will contain two standards (AA). If the value is set to two, then the correct interval will contain two alternations of the standard and comparison stimuli (ABAB) while the incorrect interval will contain four repetitions of the standard and so on.
- **Alternated (AB) Reps. ISI (ms)** Set silent interval between stimuli presented within each AAAA or ABAB interval.
- **Pre-Trial Interval** Choose whether to present the pre-trial interval.
- **Pre-Trial Interval ISI (ms)** Sets the duration of the silent interval between the end of pre-trial interval and the start of the next interval. This widget is shown only if the pre-trial interval chooser is set to "Yes".
- **Precursor Interval** Choose whether to present the precursor interval.
- **Precursor Interval ISI (ms)** Sets the duration of the silent interval between the end of precursor interval and the start of the next interval. This widget is shown only if the precursor interval chooser is set to "Yes".
- **Postcursor Interval** Choose whether to present the postcursor interval.
- **Postcursor Interval ISI (ms)** Sets the duration of the silent interval between the end of postcursor interval and the start of the next interval. This widget is shown only if the postcursor interval chooser is set to "Yes".

3.2.3 General Widgets (right panel)

- **Load Prm** Load in memory experimental parameters stored in a `.prm` file. See Section *Parameters Files* for more info.
- **Save Prm** Save experimental parameters stored in memory in a `.prm` file. See Section *Parameters Files* for more info.

- **Delete** Delete the current block from the list of blocks stored in memory.
- **Undo Unsaved** Reset the parameters in the current block to the parameters that were last saved.
- **Store** Store the parameters changes in memory.
- **Store 'n' add** Store the parameter changes in memory and add a new parameters block.
- **Store 'n' go** Store the parameter changes in memory and move to the next block storage point.
- **New Block** Create a new parameters block (the parameters of the current block will be copied in the new one).
- **Previous** Move to the previous block storage point.
- **Next** Move to the next block storage point.
- **Shuffle** Shuffle the block presentation positions (see *Block Presentation Position*).
- **Reset** Reset the block presentation positions and move to the first block position (see *Block Presentation Position*).
- **Jump to Block** Jump to a given block storage point.
- **Previous Position** Move to the previous block presentation position (see *Block Presentation Position*).
- **Next Position** Move to the next block presentation position (see *Block Presentation Position*).
- **Jump to Position** Jump to the given block presentation position (see *Block Presentation Position*).
- **Shift Blk. Down** Shift the current block to a lower storage point.
- **Shift Blk. Up** Shift the current block to a higher storage point.
- **Experiment Doc** Show the available documentation for the current experiment.

3.2.4 Paradigm Widgets

3.2.5 1-Pair Same/Different Paradigm Widgets

- **No. Trials** Set the number of trials to be presented in the current block.
- **No. Practice Trials** Set the number of practice trials to be presented in the current block. Practice trials are presented at the beginning of the block; the responses to these trials are not included in the statistics.

3.2.6 Constant 1-Interval 2-Alternatives Paradigm Widgets

- **No. Trials** Set the number of trials to be presented in the current block.
- **No. Practice Trials** Set the number of practice trials to be presented in the current block. Practice trials are presented at the beginning of the block; the responses to these trials are not included in the statistics.

3.2.7 Constant m-Intervals n-Alternatives Paradigm Widgets

- **No. Trials** Set the number of trials to be presented in the current block.
- **No. Practice Trials** Set the number of practice trials to be presented in the current block. Practice trials are presented at the beginning of the block; the responses to these trials are not included in the statistics.

3.2.8 Multiple Constants ABX Paradigm Widgets

- **No. Trials** Set the number of trials to be presented in the current block.
- **No. Practice Trials** Set the number of practice trials to be presented in the current block. Practice trials are presented at the beginning of the block; the responses to these trials are not included in the statistics.
- **No. Differences** Set the number of comparisons to perform.

3.2.9 Multiple Constants 1-Interval 2-Alternatives Paradigm Widgets

- **No. Trials** Set the number of trials to be presented in the current block for each condition.
- **No. Practice Trials** Set the number of practice trials to be presented in the current block for each condition. The responses to these trials are not included in the statistics.
- **No. Differences** Set the number of conditions to be used in the current block.

3.2.10 Multiple Constants m-Intervals n-Alternatives Paradigm Widgets

- **No. Trials** Set the number of trials to be presented in the current block for each condition.
- **No. Practice Trials** Set the number of practice trials to be presented in the current block for each condition. The responses to these trials are not included in the statistics.
- **No. Differences** Set the number of conditions to be used in the current block.

3.2.11 Odd One Out Paradigm Widgets

- **No. Trials** Set the number of trials to be presented in the current block.
- **No. Practice Trials** Set the number of practice trials to be presented in the current block. Practice trials are presented at the beginning of the block; the responses to these trials are not included in the statistics.
- **No. Differences** Set the number of comparisons to perform.

3.2.12 PEST Paradigm Widgets

WARNING PEST support is experimental and has received very little testing!

- **Procedure** If "Arithmetic" the quantity defined by the step size will be added or subtracted to the parameter that is adaptively changing. If "Geometric" the parameter that is adaptively changing will be multiplied or divided by the quantity defined by the step size.
- **Corr. Resp. Move Track** This determines whether correct responses move the adaptive track down, or up. Choose down if you want the adaptive parameter to *decrease* as a consequence of correct responses. Choose up if you want the adaptive parameter to *increase* as a consequence of correct responses. For example, in a signal detection task in which the signal level is varied you should choose `Down` (signal level decreases as a consequence of correct responses). On the other hand, in a signal detection task in which the noise level is varied you should choose `Up` (noise level increases as a consequence of correct responses).
- **Percent Correct Tracked** Set the percentage correct point on the psychometric function to be tracked by the adaptive procedure.
- **Initial Step Size** Set the initial step size.
- **Minimum Step Size** Set the minimum step size. When the minimum step size is reached the block is terminated.
- **Maximum Step Size** Set the maximum allowed step size.
- **W** Deviation limit of the sequential test (see [TaylorAndCreelman1967]).

3.2.13 PSI Paradigm Widgets

3.2.14 Transformed Up-Down Paradigm Widgets

- **Procedure** If `Arithmetic` the step size will be added or subtracted to the parameter that is adaptively varied. If `Geometric` the parameter that is adaptively varied will be multiplied or divided by the step size.
- **Corr. Resp. Move Track** This determines whether correct responses move the adaptive track down, or up. Choose down if you want the adaptive parameter to *decrease* as a consequence of correct responses. Choose up if you want the adaptive parameter to *increase* as a consequence of correct responses. For example, in a signal detection task in which the signal level is varied you should choose `Down` (signal level decreases as a consequence of correct responses). On the other hand, in a signal detection task in which the noise level is varied you should choose `Up` (noise level increases as a consequence of correct responses).
- **Rule Down** Set the number of consecutive correct, or incorrect responses, depending on which type of responses move the track down, needed to make a step down.
- **Rule Up** Set the number of consecutive correct, or incorrect responses, depending on which type of responses move the track up, needed to make a step up.

- **Initial Turnpoints** Set the number of initial turnpoints. The initial turnpoints serve to bring quickly the adaptive track towards the listener's threshold. These turnpoints are not included in the threshold estimate.
- **Total Turnpoints** Set the number of total turnpoints. The number of total turnpoints is equal to the number of initial turnpoints that are not included in the threshold estimate plus the number of turnpoints that you want to use for the threshold estimate.
- **Step Size 1** Set the step size for the initial turnpoints.
- **Step Size 2** Set the step size to be used after the number of initial turnpoints has been reached.

3.2.15 Transformed Up-Down Interleaved Paradigm Widgets

- **Procedure** If "Arithmetic" the quantity defined by the step size will be added or subtracted to the parameter that is adaptively changing. If "Geometric" the parameter that is adaptively changing will be multiplied or divided by the quantity defined by the step size.
- **No. Tracks** Set the number of adaptive tracks.
- **Max. Consecutive Trials x Track** Set the maximum number of consecutive trials per track.
- **Turnpoints to Average** Since track selection is pseudo-random, it may happen that for a track the number of total turnpoints collected is greater than the number of total turnpoints requested for that track. If "All final step size (even)" is selected, the threshold will be estimated using all the turnpoints collected after the initial turnpoints, unless the number of these turnpoints is odd, in which case the first of these turnpoints will be discarded. If "First N final step size" is selected the threshold will be estimated using only the number of requested turnpoints collected after the initial turnpoints. If "Last N final step size" is selected the threshold will be estimated using only the last N turnpoints, where N equals the number of requested turnpoints.
- **Corr. Resp. Move Track X** This determines whether correct responses move the adaptive track down, or up. Choose `Down` if you want the adaptive parameter to *decrease* as a consequence of correct responses. Choose `Up` if you want the adaptive parameter to *increase* as a consequence of correct responses. For example, in a signal detection task in which the signal level is varied you should choose `Down` (signal level decreases as a consequence of correct responses). On the other hand, in a signal detection task in which the noise level is varied you should choose `Up` (noise level increases as a consequence of correct responses).
- **Rule Down Track X** Set the number of consecutive correct responses needed to subtract the current step size from the adaptive parameter (for arithmetic procedures) or divide the adaptive parameter by the current step size (for geometric procedures) for track number X.
- **Rule Up Track X** Set the number of consecutive correct, or incorrect responses, depending on which type of responses move track *X* down, needed to make a step down for track *X*.
- **Initial Turnpoints Track X** Set the number of consecutive correct, or incorrect responses, depending on which type of responses move track *X* up, needed to make a step up for track *X*.
- **Total Turnpoints Track X** Set the number of total turnpoints for track number X. The number of total turnpoints is equal to the number of initial turnpoints that are not included in the threshold estimate plus the number of turnpoints that you want to use for the threshold estimate.

- **Step Size 1 Track X** Set the step size for the initial turnpoints for track number X.
- **Step Size 2 Track X** Set the step size to be used after the number of initial turnpoints has been reached for track number X.

3.2.16 UML Paradigm Widgets

- **Psychometric Function** The shape of the psychometric function used to fit the responses of the listener.
- **Posterior Summary** Choose whether to use the mean or the mode for the estimation

of parameter values from the Bayesian posterior distribution of parameter values.

- **Plot UML Par. Space** Generate a graphical summary of the parameter space used to

  initialize the UML procedure.
- **No. Trials** Set the number of trials to be presented in the current block.
- **Swpt. Rule** Choose whether to use an up-down or a random sweetpoint selection rule.
- **Rule Down** The number of consecutive correct responses necessary to move to the lower sweetpoint.
- **Stim. Min** Set the minimum value of the stimulus dimension that is being varied adaptively (e.g. signal level, or frequency difference).
- **Stim. Max** Set the maximum value of the stimulus dimension that is being varied adaptively (e.g. signal level, or frequency difference).
- **Stim. Scaling** Indicate whether the stimulus dimension that is being varied adaptively should be scaled linearly or logarithmically. The
- **Suggested Lapse Swpt.** The suggested stimulus value for the lapse rate sweetpoint. This value is used as the lapse rate sweetpoint unless the current estimate of the psychometric function at the probability value `Pr. Corr. at Est. Lapse Swpt` (see below) is larger. In the latter case the current estimate of the psychometric function at the probability value `Pr. Corr. at Est. Lapse Swpt` is used as the lapse rate sweetpoint, as long as it is smaller than `Stim. Max`, in which case `Stim. Max` will be used as the lapse rate sweet point.
- **Pr. Corr. at Est. Lapse Swpt.** The proportion correct at the estimated lapse sweet point. If the estimated lapse sweetpoint exceeds the will be used the suggested lapse sweetpoint, the estimated lapse sweetpoint will be used as the lapse rate sweetpoint.
- **Mid Point Min** The minimum possible value of the midpoint of the psychometric function.
- **Mid Point Max** The maximum possible value of the midpoint of the psychometric function.
- **Mid Point Step** The size of the step between successive points in the grid defining the parameter space for the midpoint of the psychometric function.
- **Mid Point Prior** The shape of the prior distribution for the midpoint of the psychometric function.
- **Mid Point mu** The mean of the prior distribution for the midpoint of the psychometric function.

- **Mid Point STD** The standard deviation of the prior distribution for the midpoint of the psychometric function.
- **Slope Min** The minimum possible value of the slope of the psychometric function.
- **Slope Max** The maximum possible value of the slope of the psychometric function.
- **Slope Step** The size of the step between successive points in the grid defining the parameter space for the slope of the psychometric function.
- **Slope Spacing** Indicate whether the spacing between successive points in the grid defining the parameter space for the slope of the psychometric function should be linear or logarithmic.
- **Slope Prior** The shape of the prior distribution for the slope of the psychometric function.
- **Slope mu** The mean of the prior distribution for the slope of the psychometric function.
- **Slope STD** The standard deviation of the prior distribution for the slope of the psychometric function.
- **Lapse Min** The minimum possible value of the lapse rate of the psychometric function.
- **Lapse Max** The maximum possible value of the lapse rate of the psychometric function.
- **Lapse Step** The size of the step between successive points in the grid defining the parameter space for the lapse rate of the psychometric function.
- **Lapse Spacing** Indicate whether the spacing between successive points in the grid defining the parameter space for the lapse rate of the psychometric function should be linear or logarithmic.
- **Lapse Prior** The shape of the prior distribution for the lapse rate of the psychometric function.
- **Lapse mu** The mean of the prior distribution for the lapse rate of the psychometric function.
- **Lapse STD** The standard deviation of the prior distribution for the lapse rate of the psychometric function.
- **Load UML state from prev. blocks** If `Yes`, at the end of each block the state of the UML posterior parameter distribution will be saved in a file named after the condition label of the block. When a subsequent block with the same condition label is encountered this file will be used to set the initial UML posterior parameter distribution for the block instead of the priors specified in the control window.

3.2.17 Weighted Up-Down Paradigm Widgets

- **Procedure** If "Arithmetic" the quantity defined by the step size will be added or subtracted to the parameter that is adaptively changing. If "Geometric" the parameter that is adaptively changing will be multiplied or divided by the quantity defined by the step size.
- **Corr. Resp. Move Track** This determines whether correct responses move the adaptive track down, or up. Choose down if you want the adaptive parameter to *decrease* as a consequence of correct responses. Choose up if you want the adaptive parameter to *increase* as a consequence of correct responses. For example, in a signal detection task in which the signal level is varied you should choose `Down` (signal level decreases as a consequence of correct responses). On the other hand, in a signal detection task in which the noise level is varied you should choose `Up` (noise level increases as a consequence of correct responses).

- **Percent Correct Tracked** Set the percentage correct point on the psychometric function to be tracked by the adaptive procedure. The ratio of the "Up" and "Down" steps is automatically adjusted by the software to satisfy this criterion.
- **Initial Turnpoints** Set the number of initial turnpoints. The initial turnpoints serve to bring quickly the adaptive track towards the listener's threshold. These turnpoints are not included in the threshold estimate.
- **Total Turnpoints** Set the number of total turnpoints. The number of total turnpoints is equal to the number of initial turnpoints that are not included in the threshold estimate plus the number of turnpoints that you want to use for the threshold estimate.
- **Step Size 1** Set the "Down" step size for the initial turnpoints. The "Up" step size is automatically calculated to satisfy the "Percent Correct Tracked" criterion.
- **Step Size 2** Set the "Down" step size to be used after the number of initial turnpoints has been reached. The "Up" step size is automatically calculated to satisfy the "Percent Correct Tracked" criterion.

3.2.18 Weighted Up-Down Interleaved Paradigm Widgets

- **Procedure** If "Arithmetic" the quantity defined by the step size will be added or subtracted to the parameter that is adaptively changing. If "Geometric" the parameter that is adaptively changing will be multiplied or divided by the quantity defined by the step size.
- **No. Tracks** Set the number of adaptive tracks.
- **Max. Consecutive Trials x Track** Set the maximum number of consecutive trials per track.
- **Turnpoints to Average** Since track selection is pseudo-random, it may happen that for a track the number of total turnpoints collected is greater than the number of total turnpoints requested for that track. If "All final step size (even)" is selected, the threshold will be estimated using all the turnpoints collected after the initial turnpoints, unless the number of these turnpoints is odd, in which case the first of these turnpoints will be discarded. If "First N final step size" is selected the threshold will be estimated using only the number of requested turnpoints collected after the initial turnpoints. If "Last N final step size" is selected the threshold will be estimated using only the last N turnpoints, where N equals the number of requested turnpoints.
- **Corr. Resp. Move Track X** This determines whether correct responses move the adaptive track number *X* down, or up. Choose `Down` if you want the adaptive parameter to *decrease* as a consequence of correct responses. Choose `Up` if you want the adaptive parameter to *increase* as a consequence of correct responses. For example, in a signal detection task in which the signal level is varied you should choose `Down` (signal level decreases as a consequence of correct responses). On the other hand, in a signal detection task in which the noise level is varied you should choose `Up` (noise level increases as a consequence of correct responses).
- **Percent Correct Tracked** Set the percentage correct point on the psychometric function to be tracked by the adaptive procedure for track number X. The ratio of the "Up" and "Down" steps is automatically adjusted by the software to satisfy this criterion.
- **Initial Turnpoints Track X** Set the number of initial turnpoints for track number X. The initial turnpoints serve to bring quickly the adaptive track towards the listener's threshold. These turnpoints are not included in the threshold estimate.

- **Total Turnpoints Track X** Set the number of total turnpoints for track number X. The number of total turnpoints is equal to the number of initial turnpoints that are not included in the threshold estimate plus the number of turnpoints that you want to use for the threshold estimate.
- **Step Size 1 Track X** Set the "Down" step size for the initial turnpoints for track number X. The "Up" step size is automatically calculated to satisfy the "Percent Correct Tracked" criterion.
- **Step Size 2 Track X** Set the "Down" step size to be used after the number of initial turnpoints has been reached for track number X. The "Up" step size is automatically calculated to satisfy the "Percent Correct Tracked" criterion.

3.2.19 The Menu Bar

A screenshot of the menu bar is shown in Figure *The menu bar*. This bar is located in the upper left corner of the "Control Window". Each menu will be described below.

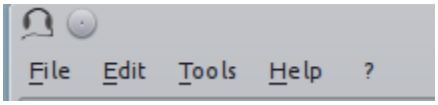

Fig. 1: The menu bar

3.2.20 The File Menu

- **Process Results (Plain Text)** Process block summary results files to obtain session summary results files. For more info see Section *Process Results Dialog*.
- **Process Results Table** Process block summary results table files to obtain session summary table results files. For more info see Section *Process Results Dialog*.
- **Open Results File** Open the file where `pychoacoustics` is currently saving data with the default text editor.
- **Exit** Close `pychoacoustics`.

3.2.21 The Edit Menu

- **Edit Preferences** Edit application preferences. See Section *Edit Preferences Dialog* for further info.
- **Edit Phones** Edit the phones database, and set the calibration levels for your phones. See Section *Edit Phones Dialog* for further info.
- **Edit Experimenters** Edit the experimenters database. See Section *Edit Experimenters Dialog* for further info.

3.2.22 The Tools Menu

- **Swap Blocks** Swap the storage position of two parameter blocks.

3.2.23 The Help Menu

- **Manual (pdf)** Open a pdf copy of the manual.
- **Manual (html)** Open a html copy of the manual.
- **Fortunes** Show psychoacoustics fortunes. I'm always collecting new ones, so if you happen to know any interesting ones, please, e-mail them to me <sam.carcagno@gmail.com> so that I can add them to the collection.
- **About pychoacoustics** Show information about the licence, the version of the software and the version of the libraries it depends on.

3.2.24 The "what's this?" Button.

If you click on this button, and then click on a widget, you can get some information about the widget (this is not implemented for all widgets).

3.3 Process Results Dialog

Figure *The process results dialog* show a screenshot of the process results dialog. The dialog is the same for all procedures, except that for procedures in which *d'* is computed, there is an additional checkbox asking whether to apply a correction to hit/false alarm rates of zero or one. For information on the format of the result files, please see Section *Result Files*. For tabular results files, if both matplotlib and pandas are installed there are additional checkboxes allowing to plot the results in a window or on a pdf file. Not all experimental paradigms support plotting.

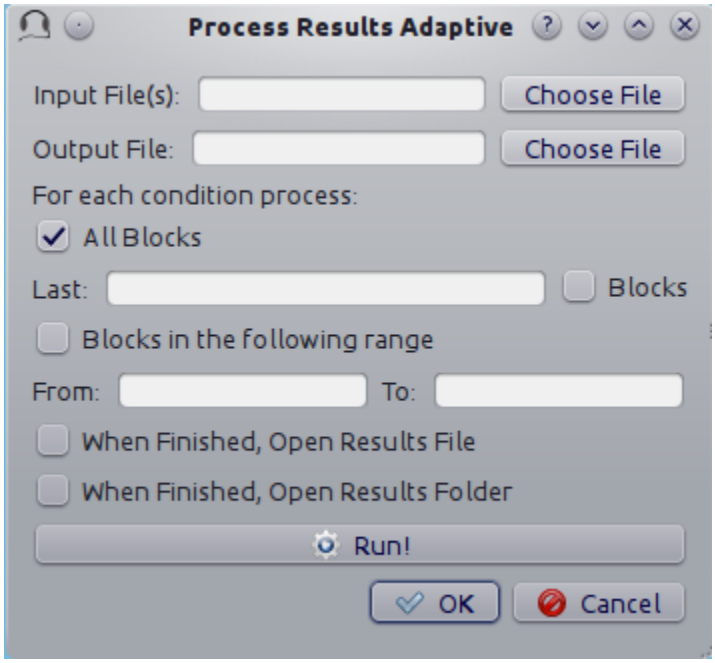

Fig. 2: The process results dialog

- **Input File(s)** Give the filepath of one or more files to be processed. The "Choose File" button can be used to select the file(s). Multiple filepaths should be separated by a semicolon ";".
- **Output File** Give the filename of the output file.
- **For each condition process:**
  - **All Blocks** If checked, all blocks in the result file(s) will be processd.
  - **Last X Blocks** If checked, only the last X blocks will be processed.
  - **Blocks in the following range** If checked, only blocks in the specified range will be processed (indexing starts from 1).
- **d-prime correction** If checked, convert hit rates of 0 and 1 to $1/2N$ and $1 - 1/(2N)$ respectively, where N is the number of trials, to avoid infinite values of *d'* (see [MacmillanAndCreelman2005] p. 8). This checkbox is available only for some paradigms.
- **When finished, open results file** If checked, the output file will be opened in the default text editor when processing has finished.
- **When finished, open results folder** If checked, the folder containing the output file will be opened when processing has finished.
- **Run!** Click this button to process the result files.

3.4 Edit Preferences Dialog

The preferences dialog is divided into several tabs. These are described in turn below.

3.4.1 General

- **Language (requires restart)** Choose the application language. At the moment and for the foreseeable future only English is supported.
- **Country (requires restart)** Set the country locale to be used for the application. Some things (for example the way dates are written in result files) depend on this setting.
- **Response Box Language (requires restart)** Choose the language to be used for the "Response Box". This sets the language to be used for the button labels and other GUI elements that the experimental listener is presented with.
- **Response Box Country (requires restart)** Set the country locale for the response box.
- **csv separator** Choose the separator field to be used when writing the csv tabular result files.
- **Warn if listener name missing** If checked, pop up a warning message if the listener name is missing at the beginning of a session.
- **Warning if session label missing** If checked, pop up a warning message if the session label is missing at the beginning of a session.
- **Process results when finished** If checked, process automatically the block summary file to generate the session summary file at the end of the experiment.

- **d-prime correction** If checked, when automatically processing result files, convert hit rates of 0 and 1 to $1/2N$ and $1 - 1/(2N)$ respectively, where N is the number of trials, to avoid infinite values of *d'* (see [MacmillanAndCreelman2005] p. 8).
- **Max Recursion Depth (requires restart)** Set the maximum recursion depth of the Python interpreter stack. This setting should be changed only if you intend to run `pychoacoustics` in automatic or simulated listener response mode (see *Response Mode*). Beware, setting a max recursion depth value smaller than the default value may cause `pychoacoustics` to crash or not even start. In case `pychoacoustics` does not start because of this, delete your preferences settings file to restore the default max recursion depth value.
- **Execute command at startup** Executes an OS command at startup. May be useful to initialize a soundcard in certain situations.

3.4.2 Sound

- **Play Command** Set an internal or external command to play sounds.
- **Device** Set the soundcard to be used to play sounds. This chooser is available only for certain internal play commands (currently alsaaudio and pyaudio).
- **Buffer Size (samples)** Set the buffer size in number of samples to be used to output sounds. This chooser is available only for certain internal play commands (currently alsaaudio and pyaudio).
- **Default Sampling Rate** Set the default sampling rate.
- **Default Bits** Set the default bit depth.
- **Wav manager (requires restart)** Choose the wav manager.
- **Write wav file** Write wav files with the sounds played on each trial in the current `pychoacoustics` working directory.
- **Write sound sequence segment wavs** For sound sequences, write a wav file for each segment of the sequence in the current `pychoacoustics` working directory.
- **Append silence to each sound (ms)** Append a silence of the given duration at the end of each sound. This is useful on some versions of the Windows operating system that may cut the sound buffer before it has ended resulting in audible clicks.

3.4.3 Response Box

- **Response Box Button Font** Choose the font for the response box button.
- **Correct Light Color** Choose the color of the feedback light after a correct response.
- **Incorrect Light Color** Choose the color of the feedback light after an incorrect response.
- **Neutral Light Color** Choose the color of the feedback light when specific feedback as to the correctness of the response is not given. A light is instead simply flashed to acknowledge that the response has been recorded.

- **Off Light Color** Choose the color of the response light when the response light is off (that is when feedback of any kind is not being given.
- **Response Light Font** Choose the font of used to present text in the response light area when feedback is textual.
- **Correct Response Text Feedback** Choose the feedback text to show in case of a correct response. If left to `(Default)`, a default message will be shown in the language chosen for the response box (if available). Applies only if feedback is textual.
- **Incorrect Response Text Feedback** Choose the feedback text to show in case of an incorrect response. If left to `(Default)`, a default message will be shown in the language chosen for response box (if available). Applies only if feedback is textual.
- **Neutral Response Text Feedback** Choose the feedback text to show when specific feedback as to the correctness of the response is not given. If left to `(Default)`, a default message will be shown in the language chosen for response box (if available). Applies only if feedback is textual.
- **Correct Text Color** Choose the color of the feedback text to show in case of a correct response. Applies only if feedback is textual.
- **Incorrect Response Text Feedback** Choose the color of the feedback text to show in case of an incorrect response. Applies only if feedback is textual.
- **Neutral Response Text Feedback** Choose the color of the feedback text to show when specific feedback as to the correctness of the response is not given. Applies only if feedback is textual.

3.4.4 Notifications

- **Play End Message** If checked, play a wav file at the end of the experiment. This could be short message to let the listeners know they have finished and thank them for their participation in the experiment. One or more wav files need to be set through the "Choose wav" button for this work.
- **Choose wav** Choose the wav file to be played as the end message. Clicking on this button brings up another dialog where you can select the wav files to be played and their output RMS. Only one of the wav files listed here and with the "Use" flag set to will be randomly chosen and played.
- **blocks before end of experiment** Set how many blocks before the end of the experiment the two actions listed below (send notification e-mail and execute custom command) should be performed.
- **Send notification e-mail** If checked, send a notification e-mail to the experimenter to notify her that the experiment is about to finish.
- **Execute custom command** If checked, execute an operating system command before the end of the experiment. This command could be used to automatically send an sms for example.
- **Send data via e-mail** At the end of the experiment, send the results file to the experimenter .
- **Execute custom command** At the end of the experiment, execute an operating system command.
- **Outgoing Server (SMTP)** Set the name of the SMTP server to be used by `pychoacoustics` to send e-mails.
- **Port** Set the port number for the SMTP server.

- **Security** Set the security protocol for network exchanges with the SMTP server.
- **Server requires identification** Check this if the SMTP server requires identification.
- **Username** Set the username for the SMTP server.
- **Password** Set the password for the SMTP server.
- **Send test e-mail** Send a test e-mail to check that the server settings are OK.

3.4.5 EEG

- **ON Trigger** The ON trigger value (decimal).
- **OFF Trigger** The OFF trigger value (decimal).
- **Trigger Duration (ms)** The duration of the trigger in milliseconds.

3.5 Edit Phones Dialog

A screenshot of the "Edit Phones" dialog is shown in Figure *Edit Phones Dialog*.

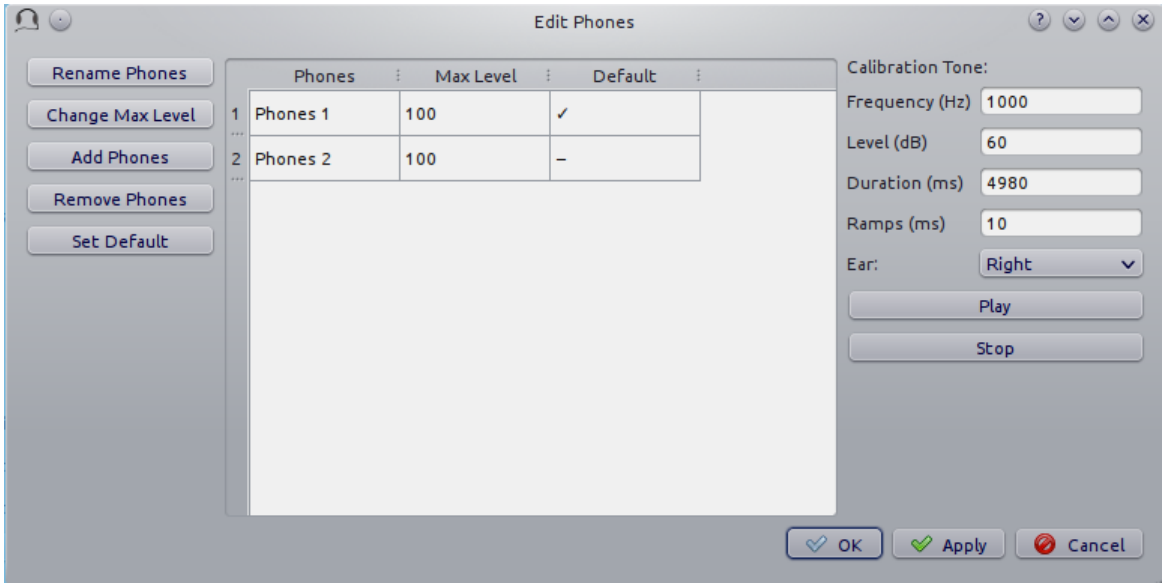

Fig. 3: Edit Phones Dialog

Most of the fields should be pretty much self-explanatory. Using this dialog you can add headphones/earphones models to the phones database. The phone with the "Default" flag set will be selected by default when `pychoacoustics` is started. In the "Max Level" field you should enter the level in dB SPL that is output by the phone for a full amplitude sinusoid (a sinusoid with a peak amplitude of 1). This value will be used by `pychoacoustics` to output sounds at specific levels in dB SPL. On the rightmost panel of the dialog you have facilities to play a sinusoid with a specified level. You can use these facilities

to check with a SPL meter (or a voltmeter depending on how you're doing it) that the actual output level corresponds to the desired output level. Using these facilities you can also play a full amplitude sinusoid: you need to set the level of the sinuoid to the "Max Level" of the phone in the dialog (whatever it is). Be careful because it can be very loud! More detailed instructions on the calibration procedure are provided below.

3.5.1 Calibrating with an SPL meter

Open the "Edit Phones" dialog. Select the phone for which you want to calibrate and note its `MaxLevel` (by default this is set to 100 dB SPL). Use the rightmost panel to play a 1-kHz sinusoid at the `MaxLevel` (e.g. 100 dB), and read the measurement on the SPL meter. Change the `MaxLevel` for the phone to the measurement you just read on the SPL meter.

You don't actually need to play the sinusoid at the `MaxLevel` (and it may be better not to do so because you may get distortions at very high levels). Instead, you could for example play it at a level equal to `MaxLevel` - 20. The reading that you would obtain from the SPL meter would then be 20 dB below the `MaxLevel`. You would then simply add 20 to the SPL meter reading and set `MaxLevel` to this value.

3.5.2 Calibrating with a voltmeter

Open the "Edit Phones" dialog. Select the phone for which you want to calibrate and note its `MaxLevel` (by default this is set to 100 dB SPL). Use the rightmost panel to play a 1-kHz sinusoid at the `MaxLevel` (e.g. 100 dB), and note the RMS voltage reading from a voltmeter connected to a cable receiving input from the soundcard. Manufacturers of professional phones usually provide datasheets indicating what is the dB SPL level output by the phone when it is driven by a 1-volt $_{\text{RMS}}$ sinusoid at 1 kHz. You can use this figure to calculate what the dB SPL output is for the 1-kHz sinusoid. Suppose that the dB SPL output for a 1-volt $_{\text{RMS}}$ sinusoid at 1 kHz is L_r, and the voltage output for the sinusoid played at `MaxLevel` is V_x, the dB SPL output for the sinusoid (L_x) will be:

$$L_x = L_r + 20log10(V_x)$$

if the reference RMS voltage in the datasheet is not 1 but some other value V_r, L_x can be calculated as:

$$L_x = L_r + 20log10(V_x/V_r)$$

Finally, set the `MaxLevel` for the phone you're calibrating to L_x. As for the SPL meter calibration you do not actually need to play the sinusoid at the `MaxLevel` (and it may be better not to do so because you may get distortions at very high levels). Instead, you could for example play it at a level equal to `MaxLevel` - 20. You would then add back the 20 dBs in the equation to compute L_x:

$$L_x = L_r + 20log10(V_x) + 20$$

3.6 Edit Experimenters Dialog

A screenshot of the "Edit Experimenters" dialog is shown in Figure *Edit Experimenters Dialog*.

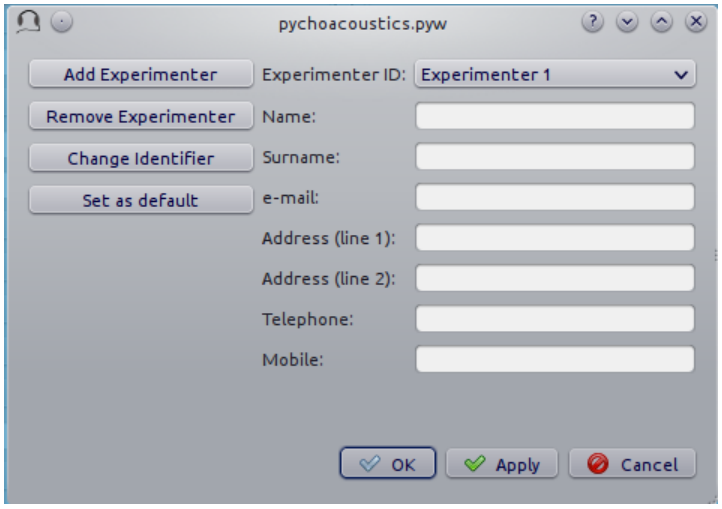

Fig. 4: Edit Experimenters Dialog

Most of the fields should be pretty much self-explanatory. Here you can add the details of the experimenters that work in your lab in the experimenter database. The main functions of this database at the moment are a) writing the experimenter name in the results file; b) using the experimenter e-mail for sending notifications and/or results files (see Section sec-edit_pref_dia_notifications).

3.7 The Response Box

The "response box" consists of a large button (the "status button") that is used to start a block of trials, a feedback light to display trial by trial feedback, interval lights to mark observation intervals, and response buttons. The responses can be given either by means of mouse clicks, or using the numeric keypad (key "1" for the first button, key "2" for the second button etc. . .). Responses given before all observation intervals have been presented are not accepted.

The status button can be activated by pressing the `Ctrl+R` shortcut. At the start of each block the label of the "Status Button" is set to "Start". Once the listener starts a block of trials the label of the status button changes to "Running". When a whole series of blocks is finished the label of the status button changes to "Finish". If no blocks are stored in memory the label of the status button is set to "Wait".

On the top left corner of the response box there is a semi-hidden menu signalled by a little hyphen ("-"). If you click on it you have access to two functions. The "Show/Hide Control Window" function can be used to hide the control window while the experiment is running. This is useful because it prevents the listener from accidentally changing your experimental parameters or accidentally closing `pychoacoustics` (the response box itself has no "close" button, so it is not possible to close that). The "Show/Hide progress Bar" function can be used to display a progress bar at the bottom of the response box. The progress bar estimates what percentage of the experiment has been completed. This estimate depends on the procedure used (for constant procedures it is based on the number of trials done, while for adaptive procedures it is based on the number of turnpoints reached) and on the specific parameters of a given experiment (trial duration, number of trials, or number or turnpoints, all of which can differ between blocks), so in some cases the estimate can be off the mark. The "Show/Hide block progress Bar" can be used to show the position of the current block and the total number of blocks.

CHAPTER 4

Command Line User Interface

In order to automate certain tasks, or perform some advanced operations, `pychoacoustics` can be called from the command line with a number of command line options. The list of possible command line options is shown below:

- `-h, --help` Show help message.
- `-f, --file FILE` Load parameters file `FILE`.
- `-r, --results FILE` Save the results to file `FILE`.
- `-l, --listener LISTENER` Set listener label to `LISTENER`.
- `-s, --session SESSION` Set session label to `SESSION`.
- `-c, --conceal` Hide Control and Parameters Windows.
- `-p, --progbar` Show the progress bar.
- `-b, --blockprogbar` Show the progress bar.
- `-q, --quit` Quit after finished.
- `-a, --autostart` Automatically start the first stored block.
- `-k, --reset` Reset block positions.
- `-z, --seed` Set random seed.
- `-x, --recursion-depth` Set the maximum recursion depth (this overrides the maximum recursion depth set in the preferences window).
- `-g, --graphicssystem` sets the backend to be used for on-screen widgets and QPixmaps. Available options are raster and opengl.
- `-d, --display` This option is only valid for X11 and sets the X display (default is $DISPLAY).

each command line option has a short (single dash, one letter) and long (double dash, one word) form, for example to show the help message, you can use either of the two following commands:

```
$ pychoacoustics -h
$ pychoacoustics --help
```

CHAPTER 5

Paradigms

Todo: Give better description of the available paradigms.

5.1 Available Paradigms

5.1.1 Transformed Up-Down

This paradigm implements the transformed up-down adaptive procedures described by [Levitt1971]. It can be used with n-intervals, n-alternatives forced choice tasks, in which $n - 1$ "standard" stimuli and a single "comparison" stimulus are presented, each in a different temporal interval. The order of the intervals is randomized from trial to trial. The "comparison" stimulus usually differs from the "standard" stimuli for a single characteristic (e.g. pitch or loudness), and the listener has to tell in which temporal interval it was presented. A classical example is the 2-intervals 2-alternatives forced-choice task. Tasks that present a reference stimulus in the first interval, and therefore have n intervals and $n-1$ alternatives are also supported (see [GrimaultEtAl2002] for an example of such tasks)

5.1.2 Transformed Up-Down Interleaved

This paradigm implements the interleaved transformed up-down procedure described by [Jesteadt1980] .

5.1.3 Weighted Up-Down

This paradigm implements the weighted up-down adaptive procedure described by [Kaernbach1991].

5.1.4 Weighted Up-Down Interleaved

This paradigm combines the interleaved procedure described by [Jesteadt1980] with the weighted up-down method described by [Kaernbach1991].

5.1.5 Constant m-Intervals n-Alternatives

This paradigm implements a constant difference method for forced choice tasks with m-intervals and n-alternatives. For example, it can be used for running a 2-intervals, 2-alternatives forced-choice frequency-discrimination task with a constant difference between the stimuli in the standard and comparison intervals.

5.1.6 Constant 1-Interval 2-Alternatives

This paradigm implements a constant difference method for tasks with a single observation interval and two response alternatives, such as the "Yes/No" signal detection task.

5.1.7 Constant 1-Pair Same/Different

This paradigm implements a constant difference method for "same/different" tasks with a single pair of stimuli to compare.

5.1.8 Multiple Constants 1-Pair Same/Different

This paradigm implements a constant difference method for "same/different" tasks with multiple pairs of stimuli to compare.

5.1.9 Multiple Constants ABX

This paradigm implements a constant difference method for "ABX" tasks with multiple pairs of stimuli to compare.

5.1.10 Odd One Out

This paradigm implements a three-alternatives oddity procedure (see [VersfeldEtAl1996]).

5.1.11 PEST

This paradigm implements the PEST adaptive procedure described by [TaylorAndCreelman1967]. However, beware that support for this procedure in `pychoacoustics` is very experimental. Its implementation has received very little testing.

5.1.12 PSI

This paradigm implements the PSI+ and PSI-marginal adaptive procedures described by [Prins2013].

5.1.13 UML

This paradigm implements the updated maximum likelihood (UML) adaptive procedure described by [ShenAndRichards2012].

CHAPTER 6

Result Files

pychoacoustics outputs several types of result files, these are listed in Table *List of result files produced by pychoacoustics*

Table 1: List of result files produced by pychoacoustics

Type	Example	Formatting	Suffix
Block summary	myres.txt	Plain	".txt"
Trial summary	myres_trial.txt	Plain	"_trial.txt"
Session Summary	myres_sess.txt	Plain	"_sess.txt"
Tabular Block Summary	myres_table.csv	Tabular	"_table.csv"
Tabular Trial Summary	myres_table_trial.csv	Tabular	"_table_trial.csv"
Tabular Session Summary	myres_table_sess.csv	Tabular	"_table_sess.csv"

there are both "plain text" and "tabular" versions of result files. The plain text version stores along with the results each parameter that was used during the experiment. The tabular result files on the other hand store a smaller number of parameters, although additional parameters can be stored if the experimenter wishes to do so (see *Tabular Results Files*). An important advantage of tabular result files is that they are easy to import in other software (e.g. R, Libreoffice Calc) for data analysis.

The plain-text "block-summary" and tabular "block-summary" result files contain summaries for each experimental block that was run. The plain-text "trial-summary" and tabular "trial-summary" result files instead contain information on each single trial. The "block-summary" result files (either in plain or tabular format) can be usually processed to obtain "session-summary" files. The "session-summary" files contain summaries for an entire experimental session. In these files the results are averaged across different blocks that have exactly the same stored parameters.

In order to obtain the session-summary files you need to use the appropriate functions that can be accessed from the pychoacoustics "File" menu. Alternatively, you can check the "Proc. Res." and "Proc. Res. Table" checkboxes in the control window (see *General Widgets (left panel)*) to let pychoacoustics

automatically process these files at the end of an experimental session. If processing the result files manually, choose "Process Results (Plain Text)" from the "File" menu, to convert a block-summary file into a session-summary file. Choose "Process Results Table" to convert a tabular block-summary file into a tabular session summary file. You can choose to process all blocks present in the file (default action), the last n blocks (of each condition), or a range of blocks (for each condition). Once you have selected the file to process and specified the blocks to process you can click "Run!" to perform the processing. The functions that process the block-summary files also allow you to plot the results. Please, note that both the ability to process the block-summary files and plot the results are not available for all paradigms. A list of the result files processing and plotting facilities available for each paradigm is given in Table *Process results and plot facilities for various paradigms*

Table 2: Process results and plot facilities for various paradigms

Procedure	Proc. Res.	Proc. Res. Table	Plot
Constant 1-Interval 2-Alternatives	Yes	Yes	Yes
Constant 1-Pair Same/Different	Yes	Yes	Yes
Constant m-Intervals n-Alternatives	Yes	Yes	Yes
Multiple Constants ABX	Yes	Yes	Yes
Multiple Constants 1-Interval 2-Alternatives	Yes	Yes	Yes
Multiple Constants 1-Pair Same/Different	Yes	Yes	Yes
Multiple Constants m-Intervals n-Alternatives	Yes	Yes	Yes
Multiple Constants Odd One Out	No	Yes	No
Multiple Constants Sound Comparison	No	No	No
PEST	Yes	Yes	Yes
PSI	No	No	No
Transformed Up-Down	Yes	Yes	Yes
Transformed Up-Down Interleaved	Yes	Yes	Yes
UML	No	No	No
Weighted Up-Down	Yes	Yes	Yes
Weighted Up-Down Interleaved	Yes	Yes	Yes

6.1 Tabular Results Files

The tabular results files are comma separated value (CSV) text files that can be opened in a text file editor or a spreadsheet application. The separator used by default is the semicolon ";", but another separator can be specified in the `pychoacoustics` preferences window. When processing block-summary table files, make sure that the csv separator in the "Process Results Table" window matches the separator used in the file.

The tabular result files contain three sets of columns:

- paradigm-specific columns (e.g. threshold estimate, for the transformed up-down procedure, or *d'* for the constant 1-pair same/different procedure). The columns that are specific to each paradigm will be described in Section *Result Files by Paradigm*
- fixed columns that are common to all paradigms (e.g. date and time at which a block of trials started). Among these columns there is a "condition" column, where the "condition label" is written (see

General Widgets (left panel)). It is a good practice to assign a condition label as it makes it easy to sort the results as a function of the experimental condition.

- additional user-defined columns specific to each experiment

The way in which these additional user-defined columns are stored is as follows: Several text fields and choosers in `pychoacoustics` have what we will call `inSummary` check boxes. Some of these are shown marked by ellipses in Figure *inSummary check boxes*.

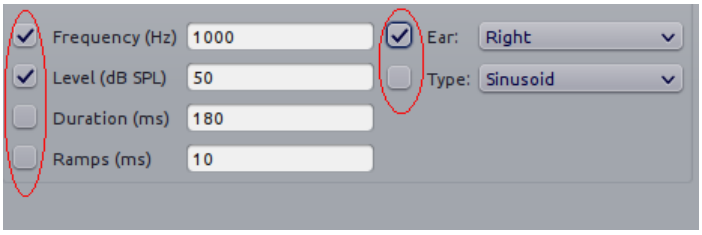

Fig. 1: `inSummary` check boxes

In the example shown in Figure *inSummary check boxes* the frequency, level and ear parameters will be stored, each in a separate column, in the tabular block-summary file, while the parameters corresponding to the unchecked boxes (duration, ramps and type) will be not. This is useful if you are running an experiment in which you are systematically varying only a few parameters across different blocks, and want to keep track of only those parameters. The `inSummary` check boxes also provide visual landmarks for quickly spotting the widgets with your parameters of interest in `pychoacoustics`.

Notice that the "Process Results Table" function, as mentioned in the previous section, will average the results for blocks with the same parameters stored in the tabular block-summary file. This means that if you are varying a certain parameter (e.g., level) across blocks, but you don't check the corresponding `inSummary` check box (for each block), the value of the parameter will not be stored in the tabular block-summary file, and as a consequence the "Process Results Table" function will not be able to sort the blocks according to the "level" parameter, and will average the results across all blocks. Not all is lost because the "level" parameter will be nonetheless stored in the "block-summary" plain-text file, but you will need more work before you can process your results with a statistical software package.

Figure *Transformed up-down table block-summary result file* shows a table block-summary result file from a transformed up-down procedure opened in Libreoffice Calc.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	threshold_geometric	SD	condition	listener	session	experimentLabel	date	time	duration	block	experiment	paradigm	F0 (Hz)
2	0.114	1.719	PT_0.6kHz_40dB	SC			14/01/2016	17:04	238.847	1	F0DL	Transformed Up-Down	600
3	3.534	1.703	CT_40	SC			14/01/2016	17:07	194.272	2	F0DL	Transformed Up-Down	100
4	0.131	1.755	PT_0.6kHz_80dB	SC			14/01/2016	17:12	287.45	3	F0DL	Transformed Up-Down	600
5	1.324	1.994	CT_80	SC			14/01/2016	17:16	247.025	4	F0DL	Transformed Up-Down	100
6	0.064	1.754	PT_2kHz_40dB	SC			14/01/2016	17:20	255.538	5	F0DL	Transformed Up-Down	2000
7	0.059	1.833	PT_2kHz_80dB	SC			14/01/2016	17:26	310.153	6	F0DL	Transformed Up-Down	2000
8													

Fig. 2: Transformed up-down table block-summary result file

the first two columns ("threshold geometric", and "SD") are specific to the transformed up-down procedure. The set of fixed columns that are common to all paradigms is described below:

- **condition** the the "condition label" for the block (see *General Widgets (left panel)*)
- **listener** the listener identifier (see *General Widgets (left panel)*)

- **session** the session identifier (see *General Widgets (left panel)*)
- **experimentLabel** the label assigned to the current experiment (see *General Widgets (left panel)*)
- **date** the date (DD/MM/YYYY) at which the block started
- **time** the time at which the block started
- **duration** how long it took for the listener to complete the block, in seconds
- **block** the block presentation position
- **experiment** the name of the experiment that was run
- **paradigm** the paradigm with which the experiment was run

The tabular trial-summary result files contain information on each single trial. For example for the transformed up-down paradigm they record the response (1 for correct, 0 for incorrect), and the value of the adaptive difference (the variable that is being varied adaptively to find its threshold). This trial by trial information can be used for various purposes, for example, it can be used to fit psychometric functions from the results of adaptive procedures.

The tabular result files contain four sets of columns:

- paradigm-specific columns (e.g. threshold estimate, for the transformed up-down procedure, or *d'* for the constant 1-pair same/different procedure). The columns that are specific to each paradigm will be described in Section *Result Files by Paradigm*
- experiment-specific columns, for example a frequency discrimination task with roving frequency of the standard may store the value of the standard frequency on each trial
- fixed columns that are common to all paradigms (e.g. date and time at which a block of trials started). Among these columns there is a "condition" column, where the "condition label" is written (see *General Widgets (left panel)*). It is a good practice to assign a condition label as it makes it easy to sort the results as a function of the experimental condition.
- additional user-defined columns specific to each experiment

---

Todo: For the experiment-specific column in tabular trial-summary files, make reference to prm['additional_parameters_to_write'] when it will be explained in the writing your own experiments section

---

6.2 Plain-Text Result Files

The "block-summary" result and the "trial-summary" result files have a header for each experimental block. The start of the header is marked by a line of 54 asterixes, an example is given below:

```
******************************************************
pychoacoustics version: 0.2.73; build date: 01-Mar-2014 09:45
Experiment version: pychoacoustics.default_experiments.audiogram 0.2.73 01-
↪Mar-2014 09:45
```

(continues on next page)

(continued from previous page)

```
Block Number: 1
Block Position: 1
Start: 01/03/2014 14:07
```

the header gives info on the software version, the experiment version (if available), the block storage point (Block Number), the block presentation position (Block Position), and has a timestamp marking the date and time at which the block was started.

After the header, there is a "parameters section" listing the experimental parameters. The beginning and the end of this section are marked by a line of 54 plus signs, a snippet of the parameters section is shown below:

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Experiment Label:
Session Label:
Condition Label:
Experiment:    Audiogram
Listener: L3
[ ... ]
Response Light Duration (ms): 500
ISI:           500

Ear: Right
Signal Type: Sinusoid
Frequency (Hz):  1000
Level (dB SPL):  50
Duration (ms):  180
Ramps (ms):  10
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

After the parameters section there is a "results section". The specific structure of this section depends on the paradigm (e.g. transformed up-down, or constant 1-interval 2-alternatives) used. The specific structure of the result section for each type of procedure will be illustrated in Section *Result Files by Paradigm*. The results section of a block-summary result file will contain summary statistics for a whole block of trials, while the results section of a trial-summary result file will contain trial-by-trial information. Besides having paradigm-specific information, "trial-summary" result files may also have experiment specific information. For example for a frequency discrimination task with roving frequency of the standard, the trial-summary result file may store the value of the standard frequency on each trial. For both "block-summary", and "trial-summary" result files the result section ends invariably with a timestamp marking the date and time at which the experimental block was completed, and a further line indicating how much time the listener took to complete the block of trials.

Todo: For the experiment-specific column in plain-text trial-summary result files, make reference to prm['additional_parameters_to_write'] when it will be explained in the writing your own experiments section

---

The "session-summary" result files have a section listing the parameters used for each experimental condition. After this section, a summary statistic for each block of the given experimental condition is presented, followed by a summary statistic for all the blocks. The specific structure of this result section for each type

of procedure will be illustrated in Section *Result Files by Paradigm*.

6.3 Result Files by Paradigm

In this section the fields of result files that are specific to each paradigm will be described.

6.3.1 Transformed Up-Down and Weighted Up-Down

Tabular Block-Summary Result Files (Transformed Up-Down and Weighted Up-Down)

The transformed up-down and weighted up-down tabular block-summary result files have two paradigm-specific columns:

- **threshold_arithmetic** or **threshold_geometric** the estimate of the threshold derived by averaging the final turnpoints of the adaptive track. Whether the column is named "threshold_arithmetic", or "threshold_geometric" depends on whether the adaptive track followed a geometric, or an arithmetic procedure. For "threshold_arithmetic" the threshold estimate is the arithmetic mean of the turnpoints, for "threshold_geometric" the threshold estimate is the geometric mean of the turnpoints.
- **SD** the standard deviation of the final turnpoints of the adaptive track. If the procedure is geometric the geometric standard deviation is calculated, otherwise the arithmetic standard deviation is calculated.

Tabular Trial-Summary Result Files (Transformed Up-Down and Weighted Up-Down)

The transformed up-down and weighted up-down tabular trial-summary result files have two paradigm-specific columns:

- **adaptive_difference** the value of the adaptive difference, that is the variable that is being varied adaptively to find its threshold
- **response** 1 if the response was correct, 0 otherwise

Tabular Session-Summary Result Files (Transformed Up-Down and Weighted Up-Down)

The transformed up-down and weighted up-down tabular session-summary result files have two paradigm-specific columns:

- **threshold_arithmetic** or **threshold_geometric** the arithmetic, or geometric average of the threshold estimates obtained in each block
- **SE** the arithmetic or geometric standard error of the threshold estimates obtained in each block

Plain-Text Block-Summary Result Files (Transformed Up-Down and Weighted Up-Down)

The results section of a transformed up-down procedure are shown below (weighted up-down result files have the same structure):

```
42.00 62.00 58.00 66.00 | 60.00 64.00 58.00 62.00 54.00 56.00 50.00 52.00 |

turnpointMean = 57.00, s.d. =  4.90
B1 = 30, B2 = 22
```

the first line lists the turnpoints; the first | sign separates the initial turnpoints, which are not included in the threshold estimate, from the final turnpoints. The second line shows the threshold estimate (`turnpointMean`) which is obtained by averaging the final turnpoints, and the standard deviation of the turnpoints. The final line lists the number of times each button was pressed by the listener. In the above case the listener pressed button one 30 times and button two 22 times. This may be useful to detect any biases in the choice of interval. The results above were collected using an arithmetic procedure. When the results are obtained with a geometric procedure the second line of the results section labels the threshold estimate as `geometric turnpointMean`, as shown in the example below:

```
0.08  5.00  1.25 80.00 | 10.00 40.00 10.00 200.00 25.00 200.00  6.25 25.00 |

geometric turnpointMean = 29.82, s.d. =  3.75
B1 = 22, B2 = 40
```

and the threshold and standard deviation values are computed as geometric mean, and geometric standard deviation, respectively.

Plain-Text Trial-Summary Result Files (Transformed Up-Down and Weighted Up-Down)

A snippet from a transformed up-down trial-summary result file is shown below:

```
50.0; 1;
50.0; 1;
46.0; 1;
46.0; 1;
42.0; 1;
42.0; 0;
46.0; 0;
50.0; 1;
```

each row represents a trial, the first colum shows the value of the adaptive difference for that trial (e.g. the level of the signal in a signal detection task), while the second column indicates whether the response was correct (`1`), or incorrect (`0`). Note that depending on the experiment, additional variables may be stored in a `trial-summary` result file. For example, in the `F0DL` experiment, which has an option to use either a fixed, or a roving F0, the F0 for the trial is listed in the third column of the `trial-summary` result file, as shown below:

```
20.0; 1; 408.58891957189206 ;
20.0; 1; 409.72312872085564 ;
5.0; 1; 474.15423804320403 ;
5.0; 1; 404.43567907073964 ;
1.25; 1; 456.6493420827598 ;
1.25; 1; 406.34270314673716 ;
```

Plain-Text Session-Summary Files (Transformed Up-Down and Weighted Up-Down)

The result section of a session-summary result file for a transformed up-down procedure is shown below:

```
57.00
44.00

Mean = 50.50
SE =  6.50
```

the session included two blocks of trials, and the first two lines list the threshold estimate for each of these blocks. The following lines present the mean and the standard error of these threshold estimates. If the procedure is arithmetic, the mean and the standard error are calculated as the arithmetic meand and the arithmetic standard error. If the procedure is geometric, the mean and the standard error are calculated as the geometric meand and the geometric standard error.

6.3.2 Transformed Up-Down and Weighted Up-Down Interleaved Result Files

Tabular Block-Summary Result Files (Transformed Up-Down and Weighted Up-Down Interleaved)

For each adaptive track, the transformed up-down and weighted up-down interleaved tabular block-summary result files have two paradigm-specific columns:

- **threshold_arithmetic_trackX** or **threshold_geometric_trackX** the estimate of the threshold derived by averaging the final turnpoints of the adaptive track number *X*. Whether the column is named "threshold_arithmetic", or "threshold_geometric" depends on whether the adaptive track followed a geometric, or an arithmetic procedure. For "threshold_arithmetic" the threshold estimate is the arithmetic mean of the turnpoints, for "threshold_geometric" the threshold estimate is the geometric mean of the turnpoints.
- **SD_trackX** the standard deviation of the final turnpoints of the adaptive track number *X*. If the procedure is geometric the geometric standard deviation is calculated, otherwise the arithmetic standard deviation is calculated.

Tabular Trial-Summary Result Files (Transformed Up-Down and Weighted Up-Down Interleaved)

Not currently implemented.

Tabular Session-Summary Result Files (Transformed Up-Down and Weighted Up-Down Interleaved)

For each adaptive track, the transformed up-down and weighted up-down interleaved tabular session-summary result files have two paradigm-specific columns:

- **threshold_arithmetic_trackX** or **threshold_geometric_trackX** the arithmetic, or geometric average of the threshold estimates obtained in each block for the adaptive track number *X*

- **SE_trackX** the arithmetic or geometric standard error of the threshold estimates obtained in each block for the track number *X*

Plain-Text Block-Summary Result Files (Transformed Up-Down and Weighted Up-Down Interleaved)

The result section of a plain-text block-summary file with a transformed up-down interleaved paradigm is shown below:

```
TRACK 1:
-212.00 -208.00 -212.00 -200.00 | -204.00 -200.00 -204.00 -202.00 -204.00 -
↪202.00 -208.00 -206.00 -208.00 -202.00 -206.00 -202.00 -208.00 -206.00 -208.
↪00 -204.00 -208.00 -204.00 -210.00 -206.00 -210.00 -204.00 -206.00 -204.00 |

turnpointMean = -205.25, s.d. =  2.69
B1 = 44, B2 = 47

TRACK 2:
-208.00 -200.00 -208.00 -204.00 | -214.00 -212.00 -228.00 -224.00 -226.00 -
↪224.00 -232.00 -230.00 -232.00 -230.00 -238.00 -232.00 |

turnpointMean = -226.83, s.d. =  7.55
B1 = 29, B2 = 42
```

for each track, after the track label ("TRACK 1", "TRACK 2", etc…), the first line lists the turnpoints; the first | sign separates the initial turnpoints, which are not included in the threshold estimate, from the final turnpoints. The second line after the track label shows the threshold estimate (`turnpointMean`) which is obtained by averaging the final turnpoints, and the standard deviation of the turnpoints. The final line lists the number of times each button was pressed by the listener. The results above were collected using an arithmetic procedure. When the results are obtained with a geometric procedure the second line of the results section labels the threshold estimate as `geometric turnpointMean`, and the threshold and standard deviation values are computed as geometric mean, and geometric standard deviation, respectively.

Plain-Text Trial-Summary Result Files (Transformed Up-Down and Weighted Up-Down Interleaved)

A snippet from the he results section of a plain-text trial-summary file for a transformed up-down interleaved paradigm is shown below:

```
-200.0; TRACK 1; 1;
-200.0; TRACK 2; 1;
-200.0; TRACK 1; 1;
-200.0; TRACK 2; 1;
```

for each trial, the first column shows the value of the adaptive difference (e.g. the level of the signal in a signal detection task), the second column shows the track number, and the third column indicates whether the response was correct (`1`), or incorrect (`0`)

Plain-Text Session-Summary Files (Transformed Up-Down and Weighted Up-Down Interleaved)

The results section of a plain-text session-summary file for a transformed up-down interleaved paradigm is shown below:

```
----------
TRACK 1:
-205.25
-228.33

Mean = -216.79
SE = 11.54


----------
TRACK 2:
-226.83
-214.14

Mean = -220.49
SE =  6.35
```

for each track, first a list of the threshold estimates obtained in each block is printed. Then the geometric or arithmetic (depending on the procedure) mean and standard deviation are shown.

6.3.3 UML and PSI Result Files

Tabular Block-Summary Result Files (UML and PSI)

The UML and PSI tabular block-summary result files have three paradigm-specific columns:

- **threshold** the estimate of the threshold, or the midpoint of the psychometric function
- **slope** the estimate of the slope of the psychometric function
- **lapse** the estimate of the lapse rate, which determines the upper asymptote of the psychometric function

Tabular Trial-Summary Result Files (UML and PSI)

The UML and PSI tabular block-summary result files have two paradigm-specific columns:

- **adaptive_difference** the value at each trial of the parameter that is adaptively varied to find the psychometric function
- **response** the response of the listener, 1 if s/he chose the correct interval, 0 otherwise

Tabular Session-Summary Result Files (UML and PSI)

Not currently implemented. Probably this will be never implemented because it makes more sense to obtain session estimates by fitting psychometric functions to the responses across all the session than to average the estimates from the tabular block-summary result files.

Plain-Text Block-Summary Result Files (UML and PSI)

The results section of a UML procedure is shown below (the structure for the PSI procedure is the same):

```
Midpoint = 0.046
Slope = 1.299
Lapse = 0.061

B1 = 51, B2 = 49
```

the first line shows the estimated midpoint of the psychometric function (the threshold), the second line shows the estimated slope of the psychometric function, and the third line shows the estimated lapse rate which determines the upper asymptote of the psychometric function.

Plain-Text Trial-Summary Result Files (UML and PSI)

The result section of the UML and PSI tabular trial-summary files has two paradigm-specific columns. The first column shows the value of the adaptive difference (the parameter that is adaptively varied to find the psychometric function) for each trial. The second column shows the response (1 for correct, 0 otherwise) given by the listener on each trial.

Plain-Text Session-Summary Result Files (UML and PSI)

Not currently implemented. Probably this will be never implemented because it makes more sense to obtain session estimates by fitting psychometric functions to the responses across all the session than to average the estimates from the plain-text block-summary result files.

6.3.4 PEST Result Files

Tabular Block-Summary Result Files (PEST)

The PEST tabular block-summary result files have a single paradigm specific column:

- **threshold_arithmetic** or **threshold_geometric** the threshold estimate. Whether the column is labelled "threshold_arithmetic", or "threshold_geometric" depends on whether an arithmetic, or a geometric procedure was used to vary the adaptive difference (the variable that is being varied adaptively to find its threshold).

Tabular Trial-Summary Result Files (PEST)

Not currently implemented.

Tabular Session-Summary Result Files (PEST)

The PEST tabular session-summary result files have two paradigm specific columns:

- **threshold_arithmetic** or **threshold_geometric** the threshold estimated by averaging across trial blocks. Whether the column is labelled "threshold_arithmetic", or "threshold_geometric" depends on whether an arithmetic, or a geometric procedure was used to vary the adaptive difference (the variable that is being varied adaptively to find its threshold). For "threshold_arithmetic" the threshold estimate is obtained by the arithmetic mean of the threshold estimates in each block. For "threshold_geometric" the threshold estimate is obtained by the geometric mean of the threshold estimates in each block.
- **SE** the standard error of the mean threshold obtained by averaging across blocks. For "threshold_arithmetic" the standard error is obtained by the arithmetic standard error of the threshold estimates in each block. For "threshold_geometric" the standard error is obtained by the geometric standard error of the threshold estimates in each block.

Plain-Text Block-Summary Result Files (PEST)

The result section of a plain-text block-summary result file obtained with the PEST paradigm is shown below:

```
Threshold =  0.62
B1 = 179, B2 = 160
```

the first line shows the threshold estimate. The second line shows how many times the listener pressed each button. This may be useful to detect any biases in the choice of interval.

Plain-Text Trial-Summary Result Files (PEST)

A snippet from a plain-text trial-summary result file obtained with the PEST paradigm is shown below:

```
50.0; 1;
50.0; 1;
50.0; 1;
50.0; 1;
50.0; 1;
50.0; 1;
50.0; 1;
45.0; 1;
40.0; 1;
30.0; 1;
30.0; 1;
30.0; 1;
```

the first column shows the value of the adaptive difference that was tested in each trial. The second column indicates whether the listener's response was correct or not (1 for correct, 0 otherwise).

Plain-Text Session-Summary Files (PEST)

The result section of a plain-text session-summary result file obtained with the PEST paradigm is shown below:

```
0.62
-0.62

Mean =  0.00
SE =  0.62
```

the section starts with a listing of the threshold estimates obtained in each block. After this listing the mean and standard error (arithmetic, or geometric, depending on the procedure used) of these threshold estimates are shown.

6.3.5 Constant m-Intervals n-Alternatives Result Files

Tabular Block-Summary Result Files (Constant m-Intervals n-Alternatives)

The constant m-intervals n-alternatives tabular block-summary result files have four paradigm-specific columns:

- **dprime** the *d'* value
- **perc_corr** the percentage of correct response
- **n_corr** the number of correct responses
- **n_trials** the total number of trials

Tabular Trial-Summary Result Files (Constant m-Intervals n-Alternatives)

Not currently available

Tabular Session-Summary Result Files (Constant m-Intervals n-Alternatives)

The constant m-intervals n-alternatives tabular session-summary result files have four paradigm-specific columns:

- **dprime** the *d'* value
- **perc_corr** the percentage of correct response
- **n_corr** the number of correct responses
- **n_trials** the total number of trials

Plain-Text Block-Summary Result Files (Constant m-Intervals n-Alternatives)

The result section of a plain-text block-summary result file obtained with a constant m-intervals n-alternatives procedure is shown below:

```
No. Correct = 37
No. Total = 50
Percent Correct =  0.74
d-prime = 0.910
```

the first row shows the number of correct responses, the second row shows the total number of trials, the third row shows the percentage of correct responses, while the last row shows the *d'* value.

Plain-Text Trial-Summary Result Files (Constant m-Intervals n-Alternatives)

A snippet from a plain-text trial-summary result file obtained with a constant m-intervals n-alternatives procedure is shown below:

```
1;
1;
0;
0;
1;
```

the first and only column shows the response of the listenr (1 for correct, 0 otherwise).

Plain-Text Session-Summary Files (Constant m-Intervals n-Alternatives)

```
d-prime Block 1 = 0.910
d-prime Block 2 = 0.742

No. Correct = 72
No. Total = 100
Percent Correct = 72.00
d-prime = 0.824
```

6.3.6 Multiple Constants m-Intervals n-Alternatives Result Files

Tabular Block-Summary Result Files (Multiple Constants m-Intervals n-Alternatives)

The multiple constants m-intervals n-alternatives tabular block-summary result files have four paradigm-specific columns:

- **dprime_subcX** the *d'* value for sub-condition *X*
- **perc_corr_subcX** the percentage of correct response for sub-condition *X*
- **n_corr_subcX** the number of correct responses for sub-condition *X*

- **n_trials_subcX** the total number of trials for sub-condition *X*

Tabular Trial-Summary Result Files (Multiple Constants m-Intervals n-Alternatives)

Not currently implemented.

Tabular Session-Summary Result Files (Multiple Constants m-Intervals n-Alternatives)

The multiple constants m-intervals n-alternatives tabular session-summary result files have the following paradigm-specific columns:

- **dprime_ALL** the *d'* value across sub-conditions
- **perc_corr_ALL** the percentage of correct response across sub-conditions
- **n_corr_ALL** the number of correct responses across sub-conditions
- **n_trials_ALL** the total number of trials across sub-conditions

then for each sub-condition:

- **dprime_subcX** the *d'* value for sub-condition *X*
- **perc_corr_subcX** the percentage of correct response for sub-condition *X*
- **n_corr_subcX** the number of correct responses for sub-condition *X*
- **n_trials_subcX** the total number of trials for sub-condition *X*

Plain-Text Block-Summary Result Files (Multiple Constants m-Intervals n-Alternatives)

The result section of a plain-text block-summary result file obtained with a multiple constants m-intervals n-alternatives procedure is shown below:

```
CONDITION, 1; 1000.0
No. Correct = 17
No. Total = 25
Percent Correct = 68.00
d-prime = 0.661

CONDITION, 2; 2000.0
No. Correct = 19
No. Total = 25
Percent Correct = 76.00
d-prime = 0.999

CONDITION, ALL
No. Correct = 36
No. Total = 50
Percent Correct = 72.00
d-prime = 0.824
```

first, for each condition, after a line with the sub-condition number and condition label, the number of correct responses, the number of total trials, the percent of correct responses, and *d'* are show in successive lines. Then the same information is shown for the data pooled across sub-conditions.

Plain-Text Trial-Summary Result Files (Multiple Constants m-Intervals n-Alternatives)

A snippet from a plain-text trial-summary result file obtained with a multiple constants m-intervals n-alternatives procedure is shown below:

```
1000.0; 1;
1000.0; 1;
2000.0; 1;
1000.0; 0;
1000.0; 1;
2000.0; 1;
```

the first column shows the sub-condition label for each trial, the second column shows the response of the listener (1 for correct, 0 otherwise).

Plain-Text Session-Summary Files (Multiple Constants m-Intervals n-Alternatives)

The result section of a plain-text session-summary result file obtained with a multiple constants m-intervals n-alternatives procedure is shown below:

```
CONDITION 1; 1000.0
Percent Correct Block 1 = 68.00
Percent Correct Block 2 = 64.00

No. Correct = 33
No. Total = 50
Percent Correct = 66.00
d-prime = 0.583

CONDITION 2; 2000.0
Percent Correct Block 1 = 76.00
Percent Correct Block 2 = 72.00

No. Correct = 37
No. Total = 50
Percent Correct = 74.00
d-prime = 0.910

CONDITION ALL
Percent Correct Block 1 = 72.00
Percent Correct Block 2 = 68.00

No. Correct = 70
No. Total = 100
Percent Correct = 70.00
d-prime = 0.742
```

first, for each condition, a line with the sub-condition number and sub-condition label is shown, followed by a list of the percentage of correct responses for that condition in each block. After these lines the number of correct responses, the number of total trials, the percentage of correct responses, and *d'* are show in successive lines.

After these summaries for each sub-condition, the same summaries are shown for the data pooled across sub-conditions ("CONDITION ALL").

6.3.7 Constant 1-Intervals 2-Alternatives Result Files

Tabular Block-Summary Result Files (Constant 1-Intervals 2-Alternatives)

The constant 1-interval 2-alternatives tabular block-summary result files have six paradigm-specific columns:

- **dprime** the *d'* value
- **nTotal** the total number of trials
- **nCorrectA** the number of correct responses for *A* (signal present) trials
- **nTotalA** the total number of *A* (signal present) trials
- **nCorrectB** the number of correct responses for *B* (signal absent) trials
- **nTotalB** the total number of *B* (signal absent) trials

Tabular Trial-Summary Result Files (Constant 1-Intervals 2-Alternatives)

Not currently implemented.

Tabular Session-Summary Result Files (Constant 1-Intervals 2-Alternatives)

The constant 1-interval 2-alternatives tabular session-summary result files have six paradigm-specific columns:

- **dprime** the *d'* value
- **nTotal** the total number of trials
- **nCorrectA** the number of correct responses for *A* (signal present) trials
- **nTotalA** the total number of *A* (signal present) trials
- **nCorrectB** the number of correct responses for *B* (signal absent) trials
- **nTotalB** the total number of *B* (signal absent) trials

Plain-Text Block-Summary Result Files (Constant 1-Intervals 2-Alternatives)

The result section of a plain-text block-summary result file obtained with a constant 1-interval 2-alternatives procedure is shown below:

```
No. Correct = 16
No. Total = 25
Percent Correct = 64.00
d-prime = 0.785

No. Correct Condition Yes = 8
No. Total Condition Yes = 11
Percent Correct Condition Yes = 72.73
No. Correct Condition No = 8
No. Total Condition No = 14
Percent Correct Condition No = 57.14
```

the first part shows the number of correct responses, number of total trials, the percentage of correct responses, and the d' value. The second part shows the number of correct responses, the number of total trials, and the percentage of correct responses separately for signal present (in this case "Yes"), and signal absent (in this case "No") trials. Please, note that "Yes", and "No" are the names of the condition of the experiment that was ran. In other experiments the names of the conditions will differ.

Plain-Text Trial-Summary Result Files (Constant 1-Intervals 2-Alternatives)

A snippet from a plain-text trial-summary result file obtained with a constant 1-interval 2-alternatives procedure is shown below:

```
Yes; 1;
Yes; 1;
Yes; 0;
Yes; 1;
Yes; 1;
No; 0;
No; 0;
No; 0;
No; 0;
No; 1;
```

the first column shows the name of the condition (in this case "Yes" for signal present, and "No" for signal absent). The second column indicates whether the response of the listener was correct or not (1 for correct, 0 otherwise).

Plain-Text Session-Summary Files (Constant 1-Intervals 2-Alternatives)

The result section of a plain-text session-summary result file obtained with a constant 1-interval 2-alternatives procedure is shown below:

```
d-prime Block 1 = 0.785
d-prime Block 2 = 0.097

No. Correct = 29
No. Total = 50
Percent Correct = 58.00
d-prime = 0.416

No. Correct A = 14
No. Total A = 23
Percent Correct A = 60.87
No. Correct B = 15
No. Total B = 27
Percent Correct B = 55.56
```

the section starts with a list of the *d'* values obtained on each block of trials. The second paragraph shows the number of correct responses, the total number of trials, the percentage of correct responses, and the *d'* value. The last paragraph shows the number of correct responses, the total number of trials, and the percentage of correct responses separately for "A" (signal present), and "B" (signal absent) trials.

6.3.8 Multiple Constants 1-Intervals 2-Alternatives Result Files

Tabular Block-Summary Result Files (Multiple Constants 1-Intervals 2-Alternatives)

The multiple constants 1-interval 2-alternatives tabular block-summary result files have the following paradigm-specific columns:

- **dprime_ALL** the *d'* value across all sub-conditions
- **nTotal_ALL** the total number of trials across all sub-conditions
- **nCorrectA_ALL** the number of correct responses for *A* (signal present) trials across all sub-conditions
- **nTotalA_ALL** the total number of *A* (signal present) trials across all sub-conditions
- **nCorrectB_ALL** the number of correct responses for *B* (signal absent) trials across all sub-conditions
- **nTotalB_ALL** the total number of *B* (signal absent) trials across all sub-conditions

then for each sub-condition:

- **dprime_subcX** the *d'* value for sub-condition *X*
- **nTotal_subcX** the total number of trials for sub-condition *X*
- **nCorrectA_subcX** the number of correct responses for *A* (signal present) trials for sub-condition *X*
- **nTotalA_subcX** the total number of *A* (signal present) trials for sub-condition *X*
- **nCorrectB_subcX** the number of correct responses for *B* (signal absent) trials for sub-condition *X*
- **nTotalB_subcX** the total number of *B* (signal absent) trials for sub-condition *X*

Tabular Trial-Summary Result Files (Multiple Constants 1-Intervals 2-Alternatives)

Not currently implemented.

Tabular Session-Summary Result Files (Multiple Constants 1-Intervals 2-Alternatives)

The multiple constants 1-interval 2-alternatives tabular session-summary result files have the following paradigm-specific columns:

- **dprime_ALL** the *d'* value across all sub-conditions
- **nTotal_ALL** the total number of trials across all sub-conditions
- **nCorrectA_ALL** the number of correct responses for *A* (signal present) trials across all sub-conditions
- **nTotalA_ALL** the total number of *A* (signal present) trials across all sub-conditions
- **nCorrectB_ALL** the number of correct responses for *B* (signal absent) trials across all sub-conditions
- **nTotalB_ALL** the total number of *B* (signal absent) trials across all sub-conditions

then for each sub-condition:

- **dprime_subcX** the *d'* value for sub-condition *X*
- **nTotal_subcX** the total number of trials for sub-condition *X*
- **nCorrectA_subcX** the number of correct responses for *A* (signal present) trials for sub-condition *X*
- **nTotalA_subcX** the total number of *A* (signal present) trials for sub-condition *X*
- **nCorrectB_subcX** the number of correct responses for *B* (signal absent) trials for sub-condition *X*
- **nTotalB_subcX** the total number of *B* (signal absent) trials for sub-condition *X*

Plain-Text Block-Summary Result Files (Multiple Constants 1-Intervals 2-Alternatives)

The result section of a plain-text block-summary result file obtained with a multiple constants 1-interval 2-alternatives procedure is shown below:

```
CONDITION: 1; Center Frequency, 1000.0
No. Correct = 22
No. Total = 25
Percent Correct = 88.00
d-prime = 2.480

No. Correct Subcondition present = 13
No. Total Subcondition present = 16
Percent Correct Subcondition present = 81.25
No. Correct Subcondition absent = 9
No. Total Subcondition absent = 9
Percent Correct Subcondition absent = 100.00
```

(continues on next page)

(continued from previous page)

```
CONDITION: 2; Center Frequency, 1001.0
No. Correct = 20
No. Total = 25
Percent Correct = 80.00
d-prime = 1.695

No. Correct Subcondition present = 9
No. Total Subcondition present = 12
Percent Correct Subcondition present = 75.00
No. Correct Subcondition absent = 11
No. Total Subcondition absent = 13
Percent Correct Subcondition absent = 84.62


CONDITION: ALL
No. Correct = 42
No Total = 50
Percent Correct = 84.00
d-prime = 2.127

No. Correct Subcondition present = 22
No. Total Subcondition present = 28
Percent Correct Subcondition present = 78.57
No. Correct Subcondition absent = 20
No. Total Subcondition absent = 22
Percent Correct Subcondition absent = 90.91
```

there are three parts, one containing summaries for each sub-condition, and one containing summaries for the data pooled across all sub-conditions ("CONDITION: ALL"). The parts containing summaries for each sub-condition start with the sub-condition number, and sub-condition label (this will vary from experiment to experiment). Following this, the total number of correct responses, the total number of trials, the percentage of correct responses, and the *d'* value for the subcondition are shown. Then the number of correct responses, the number of total responses, and the percentage of correct responses are shown for each sub-sub-condition within a sub-condition. This same information is then shown for the data pooled across all sub-condition.

Plain-Text Trial-Summary Result Files (Multiple Constants 1-Intervals 2-Alternatives)

A snippet from a plain-text trial-summary result file obtained with a multiple constants 1-interval 2-alternatives procedure is shown below:

```
Center Frequency, 1001.0; present; 0;
Center Frequency, 1000.0; absent; 1;
Center Frequency, 1001.0; absent; 1;
Center Frequency, 1000.0; present; 1;
```

the first column shows the subcondition label. The second column shows the sub-sub-condition label (trial type). The third column indicates whether the listener's response was correct or not (1 for a correct response, 0 otherwise).

Plain-Text Session-Summary Files (Multiple Constants 1-Intervals 2-Alternatives)

The result section of a plain-text session-summary result file obtained with a multiple constants 1-interval 2-alternatives procedure is shown below:

```
CONDITION: 1; Center Frequency, 1000.0
d-prime Block 1 = 2.480
d-prime Block 2 = 2.108

No. Correct = 43
No. Total = 50
Percent Correct = 86.00
d-prime = 2.480

No. Correct A = 26
No. Total A = 32
Percent Correct A = 81.25
No. Correct B = 17
No. Total B = 18
Percent Correct B = 94.44

----------
CONDITION: 2; Center Frequency, 1001.0
d-prime Block 1 = 1.695
d-prime Block 2 = 1.177

No. Correct = 38
No. Total = 50
Percent Correct = 76.00
d-prime = 1.411

No. Correct A = 18
No. Total A = 24
Percent Correct A = 75.00
No. Correct B = 20
No. Total B = 26
Percent Correct B = 76.92

----------
CONDITION: ALL
d-prime Block 1 = 2.127
d-prime Block 2 = 1.539

No. Correct = 81
No. Total = 100
Percent Correct = 81.00
d-prime = 1.790

No. Correct A = 44
No. Total A = 56
Percent Correct A = 78.57
No. Correct B = 37
No. Total B = 44
```

(continues on next page)

(continued from previous page)

```
Percent Correct B = 84.09
```

there are three parts, one containing summaries for each sub-condition, and one containing summaries for the data pooled across all sub-conditions. The parts containing summaries for each sub-condition start with a line showing the sub-condition number, and sub-condition label (this will vary from experiment to experiment). Following this there is a listing of *d'* values obtain in each block for that subcondition. The next lines show the total number of correct responses, the total number of trials, the percentage of correct responses, and the *d'* value for the given sub-condition. Then, the number of correct responses, the number of trials, and the percentage of correct responses are shown for each trial type ("A" for signal present, "B" for signal absent) within a sub-condition. This same information is then shown for the data pooled across all sub-condition.

6.3.9 Constant 1-Pair Same/Different Result Files

Tabular Block-Summary Result Files (Constant 1-Pair Same/Different)

The constant 1-pair same/different tabular block-summary result files have seven paradigm-specific columns:

- **dprime_IO** the estimated *d'* for a listener using the independent observations strategy
- **dprime_diff** the estimated *d'* for a listener using the differencing strategy
- **nTotal** the total number of trials
- **nCorrect_same** the number of correct response for "same" trials
- **nTotal_same** the total number of "same" trials
- **nCorrect_different** the number of correct response for "different" trials
- **nTotal_different** the total number of "different" trials

Tabular Session-Summary Result Files (Constant 1-Pair Same/Different)

The constant 1-pair same/different tabular session-summary result files have seven paradigm-specific columns:

- **dprime_IO** the estimated *d'* for a listener using the independent observations strategy
- **dprime_diff** the estimated *d'* for a listener using the differencing strategy
- **nTotal** the total number of trials
- **nCorrect_same** the number of correct response for "same" trials
- **nTotal_same** the total number of "same" trials
- **nCorrect_different** the number of correct response for "different" trials
- **nTotal_different** the total number of "different" trials

Plain-Text Block-Summary Result Files (Constant 1-Pair Same/Different)

The results section for a constant 1-pair same/different is shown below:

```
No. Correct = 7
No. Total = 10
Percent Correct = 70.00
d-prime IO = 1.860
d-prime diff = 2.223

No. Correct Condition same = 4
No. Total Condition same = 6
Percent Correct Condition same= 66.67
No. Correct Condition different = 3
No. Total Condition different = 4
Percent Correct Condition different= 75.00
```

the first line shows the total number of correct responses. The second line shows the total number of trials. The third line shows the percentage of correct responses. The fourth line shows the estimated *d'* for a listener using the independent observations strategy. The fifth line shows the estimated *d'* for a listener using the differencing strategy. The following lines show the number of correct responses, the total number of trials, and the percentage of correct responses, separately for "same", and "different" trials.

Plain-Text Trial-Summary Result Files (Constant 1-Pair Same/Different)

A snippet from the result section of a 1-pair same/different file is shown below:

```
same; 0;
same; 0;
different; 0;
same; 1;
different; 1;
```

the first column indicates whether the trial was a "same", or "different" trial. The second column shows the response (1 for correct, 0 otherwise) given by the listener on each trial.

Plain-Text Session-Summary Files (Constant 1-Pair Same/Different)

The result section for a 1-pair same/different paradigm session is shown below:

```
d-prime IO Block 1 = 2.430
d-prime diff Block 1 = 2.923
d-prime IO Block 2 = 1.955
d-prime diff Block 2 = 2.406

No. Correct = 46
No. Total = 60
Percent Correct = 76.67
d-prime IO = 2.250
```

(continues on next page)

(continued from previous page)

```
d-prime diff = 2.726

No. Correct A = 24
No. Total A = 33
Percent Correct A = 72.73
No. Correct B = 22
No. Total B = 27
Percent Correct B = 81.48
```

the first paragraph is a listing of the *d'* values calculated according to the independent observations and differencing strategy for each block of trials in the session. The secon paragraph lists the number of correct responses, total number of trials, percent correct, and *d'* values (for both independent observation and differencing strategy) across all the blocks of trials in the session. The last paragraph shows summary statistics for "same", and "different" trials separately ("A" refers to "same" trials, and "B" refers to "different" trials.

6.3.10 Multiple Constants 1-Pair Same-Different Result Files

Tabular Block-Summary Result Files (Multiple Constants 1-Pair Same-Different)

The multiple constants 1-pair same/different tabular block-summary result files have seven paradigm-specific columns for each pair of stimuli that are tested:

- **dprime_IO_pairX** the estimated *d'* for a listener using the independent observations strategy for the stimulus pair number *X*
- **dprime_diff_pairX** the estimated *d'* for a listener using the differencing strategy for the stimulus pair number *X*
- **nTotal_pairX** the total number of trials for the stimulus pair number *X*
- **nCorrect_same_pairX** the number of correct response for "same" trials for the stimulus pair number *X*
- **nTotal_same_pairX** the total number of "same" trials for the stimulus pair number *X*
- **nCorrect_different_pairX** the number of correct response for "different" trials for the stimulus pair number *X*
- **nTotal_different_pairX** the total number of "different" trials for the stimulus pair number *X*

Tabular Trial-Summary Result Files (Multiple Constants 1-Pair Same-Different)

The multiple constants 1-pair same/different tabular trial-summary result files have six paradigm-specific columns:

- **pair** the stimulus pair tested in the given trial
- **stim1** the label of the stimulus that was presented in the first interval
- **stim2 the label of the stimulus that was presented in the second** interval

- **case** whether the trial was a "same" or a "different" trial
- **response** 1 for a correct response, 0 otherwise

Tabular Session-Summary Result Files (Multiple Constants 1-Pair Same-Different)

The multiple constants 1-pair same/different tabular session-summary result files have seven paradigm-specific columns for each pair of stimuli that are tested:

- **dprime_IO_pairX** the estimated *d'* for a listener using the independent observations strategy for the stimulus pair number *X*
- **dprime_diff_pairX** the estimated *d'* for a listener using the differencing strategy for the stimulus pair number *X*
- **nTotal_pairX** the total number of trials for the stimulus pair number *X*
- **nCorrect_same_pairX** the number of correct response for "same" trials for the stimulus pair number *X*
- **nTotal_same_pairX** the total number of "same" trials for the stimulus pair number *X*
- **nCorrect_different_pairX** the number of correct response for "different" trials for the stimulus pair number *X*
- **nTotal_different_pairX** the total number of "different" trials for the stimulus pair number *X*

Plain-Text Block-Summary Result Files (Multiple Constants 1-Pair Same-Different)

The result section of a plain-text block-summary file obtained with the multiple constants 1-pair same/different paradigm is shown below:

```
DIFFERENCE: Pair1
No. Correct = 21
No. Total = 25
Percent Correct = 84.00
d-prime IO = 2.698
d-prime diff = 3.397

No. Correct Condition same = 12
No. Total Condition same = 14
Percent Correct Condition same = 85.71
No. Correct Condition different = 9
No. Total Condition different = 11
Percent Correct Condition different = 81.82


DIFFERENCE: Pair2
No. Correct = 19
No. Total = 25
Percent Correct = 76.00
d-prime IO = 2.216
```

(continues on next page)

(continued from previous page)

```
d-prime diff = 2.756

No. Correct Condition same = 8
No. Total Condition same = 10
Percent Correct Condition same = 80.00
No. Correct Condition different = 11
No. Total Condition different = 15
Percent Correct Condition different = 73.33
```

the result section is composed of two parts for each pair of stimuli tested. The first part lists first the pair number, and then gives summary statistics for that pair (number of correct responses, total number of trials, percent correct, *d'* for the independent observations strategy, *d'* for the differencing strategy). The second part gives summary statistics separately for "same" and "different" trials.

Plain-Text Trial-Summary Result Files (Multiple Constants 1-Pair Same-Different)

A snippet from a plain-text trial-summary result file for the multiple constants 1-pair same/different paradigm is shown below:

```
Pair2_WAV2-WAV1_different; 0;
Pair1_WAV2-WAV1_different; 1;
Pair2_WAV1-WAV2_different; 1;
Pair1_WAV2-WAV2_same; 0;
Pair2_WAV1-WAV2_different; 1;
Pair2_WAV2-WAV1_different; 1;
Pair1_WAV2-WAV2_same; 1;
```

the first column tells the stimulus pair that was tested in each trial, as well as the sequence of stimuli that was played, and whether the trial was a "same", or a "different" trial. The second column shows the response (1 for correct, 0 otherwise).

Plain-Text Session-Summary Files (Multiple Constants 1-Pair Same-Different)

Not currently implemented.

6.3.11 Multiple Constants ABX Result Files

Tabular Block-Summary Result Files (Multiple Constants ABX)

The multiple constants ABX tabular block-summary result files have seven paradigm-specific columns for each pair of stimuli that are tested:

- **dprime_IO_pairZ** the estimated *d'* for a listener using the independent observations strategy for the stimulus pair number *Z*
- **dprime_diff_pairZ** the estimated *d'* for a listener using the differencing strategy for the stimulus pair number *Z*

- **nTotal_pairZ** the total number of trials for the stimulus pair number *Z*
- **nCorrect_A_pairZ** the number of correct response for "A" trials for the stimulus pair number *Z*
- **nTotal_A_pairZ** the total number of "A" trials for the stimulus pair number *Z*
- **nCorrect_B_pairZ** the number of correct response for "B" trials for the stimulus pair number *Z*
- **nTotal_B_pairZ** the total number of "B" trials for the stimulus pair number *Z*

Tabular Trial-Summary Result Files (Multiple Constants ABX)

The multiple constants ABX tabular trial-summary result files have six paradigm-specific columns:

- **pair** the stimulus pair tested in the given trial
- **A** the label of the stimulus that was presented in interval *A*
- **B** the label of the stimulus that was presented in interval *B*
- **X** the label of the stimulus that was presented in interval *X*
- **case** whether the *X* stimulus is the same as the one presented in interval *A* or *B*
- **response** 1 for a correct response, 0 otherwise

Tabular Session-Summary Result Files (Multiple Constants ABX)

The multiple constants ABX tabular session-summary result files have seven paradigm-specific columns for each pair of stimuli that are tested:

- **dprime_IO_pairZ** the estimated *d'* for a listener using the independent observations strategy for the stimulus pair number *Z*
- **dprime_diff_pairZ** the estimated *d'* for a listener using the differencing strategy for the stimulus pair number *Z*
- **nTotal_pairZ** the total number of trials for the stimulus pair number *Z*
- **nCorrect_A_pairZ** the number of correct response for "A" trials for the stimulus pair number *Z*
- **nTotal_A_pairZ** the total number of "A" trials for the stimulus pair number *Z*
- **nCorrect_B_pairZ** the number of correct response for "B" trials for the stimulus pair number *Z*
- **nTotal_B_pairZ** the total number of "B" trials for the stimulus pair number *Z*

Plain-Text Block-Summary Result Files (Multiple Constants ABX)

The result section of a plain-text block-summary file for the multiple constants ABX paradigm is shown below:

```
DIFFERENCE: Pair1
No. Correct = 17
No. Total = 25
Percent Correct = 68.00
d-prime IO = 1.313
d-prime diff = 1.468

No. Correct Condition A = 12
No. Total Condition A = 16
Percent Correct Condition A = 75.00
No. Correct Condition B = 5
No. Total Condition B = 9
Percent Correct Condition B = 55.56
```

the result section is composed of two parts for each pair of stimuli tested. The first part lists first the pair number, and then gives summary statistics for that pair (number of correct responses, total number of trials, percent correct, *d'* for the independent observations strategy, *d'* for the differencing strategy). The second part gives summary statistics separately for trials in which stimulus *X* was presented in interval *A* and for trials in which it was presented in interval *B*.

Plain-Text Trial-Summary Result Files (Multiple Constants ABX)

A snippet from a plain-text trial-summary result file for the multiple constants ABX paradigm is shown below:

```
Pair2_WAV1-WAV2_WAV1_A; 1;
Pair2_WAV1-WAV2_WAV1_A; 0;
Pair2_WAV2-WAV1_WAV2_A; 1;
Pair2_WAV1-WAV2_WAV2_B; 0;
Pair2_WAV1-WAV2_WAV2_B; 1;
Pair1_WAV2-WAV1_WAV2_A; 1;
Pair2_WAV1-WAV2_WAV2_B; 1;
```

the first column tells the stimulus pair that was tested in each trial, as well as the sequence of stimuli that was played, and whether stimulus *X* was presented in interval *A* or in interval *B*. The second column shows the response (1 for correct, 0 otherwise).

Plain-Text Session-Summary Files (Multiple Constants ABX)

Not currently implemented.

6.3.12 Multiple Constants Odd One Out Result Files

Tabular Block-Summary Result Files (Multiple Constants Odd One Out)

The multiple constants odd one out tabular block-summary result files have five paradigm-specific columns for each subcondition tested:

- **nCorr_subcndX** the number of correct response in subcondition *X*
- **nTrials_subcndX** the number of trials in subcondition *X*
- **percCorr_subcndX** the percentage of correct responses in subcondition *X*
- **dprime_IO_subcndX** the estimated *d'* in subcondition *X* for a listener using the independent observations strategy
- **dprime_diff_subcndX** the estimated *d'* in subcondition *X* for a listener using the differencing strategy

Tabular Trial-Summary Result Files (Multiple Constants Odd One Out)

The multiple constants odd one out tabular trial-summary result files have two paradigm-specific columns:

- **subcondition** the subcondition tested in each trial
- **response** 1 for a correct response, 0 otherwise

Tabular Session-Summary Result Files (Multiple Constants Odd One Out)

The multiple constants odd one out tabular session-summary result files have five paradigm-specific columns for each subcondition tested:

- **nCorr_subcndX** the number of correct response in subcondition *X*
- **nTrials_subcndX** the number of trials in subcondition *X*
- **percCorr_subcndX** the percentage of correct responses in subcondition *X*
- **dprime_IO_subcndX** the estimated *d'* in subcondition *X* for a listener using the independent observations strategy
- **dprime_diff_subcndX** the estimated *d'* in subcondition *X* for a listener using the differencing strategy

Plain-Text Block-Summary Result Files (Multiple Constants Odd One Out)

The result section of a plain-text block-summary file for the multiple constants odd one out paradigm is shown below:

```
Condition Comparison1

No. Correct = 9
No. Trials = 10
Percent Correct = 90.000
d-prime IO = 3.324
d-prime diff = 4.028


Condition Comparison2

No. Correct = 7
```

(continues on next page)

(continued from previous page)

```
No. Trials = 10
Percent Correct = 70.000
d-prime IO = 2.101
d-prime diff = 2.504


B1 = 7, B2 = 7, B3 = 6
```

for each subcondition, after the subcondition label, the following values are listed on successive lines: the number of correct trials, the number of trials, the percentage of correct responses, the *d'* value for a listener using the independent observations strategy, and the *d'* value for a listener using the differencing strategy.

The final line shows the number of times each button was pressed by the listener.

Plain-Text Trial-Summary Result Files (Multiple Constants Odd One Out)

A snippet from a plain-text trial-summary result file for the multiple constants odd one out paradigm is shown below:

```
Comparison2; 0;
Comparison2; 1;
Comparison1; 1;
Comparison1; 1;
Comparison1; 1;
Comparison1; 0;
```

the first column shows the label of the subcondition tested on each trial, the second column shows the response (1 for correct, 0 otherwise).

Plain-Text Session-Summary Files (Multiple Constants Odd One Out)

Not currently implemented.

6.3.13 Multiple Constants Sound Comparison Result Files

Tabular Block-Summary Result Files (Multiple Constants Sound Comparison)

The multiple constants odd one out tabular block-summary result files have the following paradigm-specific columns:

- **nTrials** the total number of trials per subcondition

then, for each condition:

- **stim1_count_subcndX** the number of times stimulus 1 was chosen as the odd one in subcondition *X*
- **stim1_percent_subcndX** the percent of times stimulus 1 was chosen as the odd one in subcondition *X*

- **stim2_count_subcndX** the number of times stimulus 3 was chosen as the odd one in subcondition *X*
- **stim2_percent_subcndX** the percent of times stimulus 2 was chosen as the odd one in subcondition *X*
- **stim3_count_subcndX** the number of times stimulus 3 was chosen as the odd one in subcondition *X*
- **stim3_percent_subcndX** the percent of times stimulus 3 was chosen as the odd one in subcondition *X*

Tabular Trial-Summary Result Files (Multiple Constants Sound Comparison)

Not currently implemented.

Tabular Session-Summary Result Files (Multiple Constants Sound Comparison)

Not currently implemented.

Plain-Text Block-Summary Result Files (Multiple Constants Sound Comparison)

The result section of an odd-one-out plain-text block-summary result file is shown below:

```
Condition Comparison1
Stimulus 1 = 8/25; Percent = 32.00
Stimulus 2 = 7/25; Percent = 28.00
Stimulus 3 = 10/25; Percent = 40.00


Condition Comparison2
Stimulus 1 = 10/25; Percent = 40.00
Stimulus 2 = 6/25; Percent = 24.00
Stimulus 3 = 9/25; Percent = 36.00


B1 = 5, B2 = 40, B3 = 5
```

for each condition tested in a block of trials the result section lists the name of the condition, and for each stimulus, the number of times it was chosen as the odd one out of the total number of trials, and the percentage of times it was chosen as the odd one.

The last line in the code snippet above shows the number of times each button was pressed.

Plain-Text Trial-Summary Result Files (Multiple Constants Sound Comparison)

A snippet from the result section of an odd-one-out plain-text trial-summary result file is shown below:

```
Comparison2; 1;
Comparison1; 2;
Comparison1; 3;
```

the first column shows the condition tested on each trial. The second column shows the alternative chosen.

Plain-Text Session-Summary Files (Multiple Constants Sound Comparison)

Not currently implemented.

6.4 Log Results Files

`pychoacoustics` automatically saves backup copies of the "block summary" and "trial-summary" files in a backup folder. On Linux systems this folder is located in

```
~/.local/share/data/pychoacoustics/data_backup
```

on Windows systems it is located in

```
C:\\Users\username\.local\share\data\pychoacoustics\data_backup
```

where `username` is your account login name. A separate file is saved for each block of trials that is run. These files are named according to the date and time at which the blocks were started (the naming follows the YY-MM-DD-HH-MM-SS scheme). Unlike other results files, that are written only once a block of trials has been completed, these log results files get written as soon as information is available (e.g., a new line in the "trial-summary" results file is written at the end of each trial).

CHAPTER 7

Default Experiments

7.1 Audiogram

7.2 Demo Audiogram Multiple Frequencies

7.3 Demo Frequency Discrimination

7.4 Demo Signal Detection

Measure d' for the detection of a pure tone in a Yes/No task.

The available fields are:

- **Frequency (Hz) :** The frequency of the pure tone signal
- **Duration (ms) :** Tone duration (excluding ramps), in ms
- **Ramps (ms) :** Duration of each ramp, in ms
- **Level (dB SPL):** Level of the signal in dB SPL.

The available choosers are:

- **Ear: [`Right`, `Left`, `Both`]** The ear to which the signal will be presented

7.5 Dummy Adaptive

7.6 F0DL

7.7 Level Discrimination

7.8 WAV ABX

7.9 WAV Comparison

7.10 WAV Same/Different

CHAPTER 8

The pychoacoustics Engine

8.1 Sound Output

8.1.1 Sound Output on Linux

On Linux systems pychoacoustics can either output sound (numpy arrays) directly to the soundcard, or write a WAV file for each sound and call an external command to play it. Currently, sending sounds directly to the soundcard is possible only through the alsaaudio, or through the pyaudio Python modules. These modules are optional, and you need to install them yourself to be able to use them. Note that I've experienced issues (occasional pops and crackles) with pyaudio on the hardware that I have tested. Sound output with alsaaudio, on the other hand, has been working very well. Once the modules are installed, they will be detected automatically and you will be able to select one of them as the "Play Command" in the sound preferences dialog. When you select alsaaudio as the play command, if you have multiple soundcards, you can select the device to which the sound will be sent. There will be also an option to set the size of the buffer that alsaaudio uses to play sounds. If the buffer is not filled completely by a sound (buffer size greater than number of samples in the sound), it will be zero padded. This may lead to some latency between the offset of a sound and the onset of the following one. If you set a value smaller than one the buffer size will be automatically set to the number of samples in the sound that is being played.

Using an external command to play sounds generally works very well and is fast on modern hardware. pychoacoustics tries to detect available play commands on your system each time it starts up. On Linux systems, the recommended play command is aplay, which is installed by default on most Linux distributions. aplay supports 24-bit output on 24-bit soundcards with appropriate Linux drivers. Other possible play commands are play, which is provided by sox and sndfile-play, which is provided by the libsndfile tools. You can call another program by choosing "custom" in the "Play Command" drop-down menu and spelling out the name of the command in the box below.

8.1.2 Sound Output on Windows

The command that pychoacoustics uses by default on Windows is winsound. This command supports only 16-bit output. pychoacoustics can also use pyaudio to output sound on Windows. pyaudio, however, needs to be manually installed. pyaudio can use several Windows sound APIs, including MME, ASIO, and WASAPI. The pyaudio binaries available on the official project website support only the MME API, which is limited to 16-bit output. ASIO and WASAPI on the other hand, can play sounds with full 24-bit resolution. In order to have pyaudio built with ASIO and/or WASAPI support you need to either build it from source, enabling these APIs (not for the faint of heart), or dowload the unoffical binaries made available by Christoph Gohlke on his website.

Other possible play commands on Windows are play, which is provided by sox and sndfile-play, which is provided by the libsndfile tools. These programs need to be installed by the user. If they are in the system path, pychoacoustics will detect them automatically. Note that external media players with a graphical user interface (like foobar2000) may not work well with pychoacoustics.

8.1.3 Sound Output on macOS

By default pychoacoustics uses the afplay command to output sound on macOS. If pyaudio is properly installed and configured for the Python distribution used to run pychoacoustics it can also be used by pychoacoustics to play sounds on macOS.

8.1.4 Sound Output on FreeBSD

The default command used by pychoacoustics to play sound on FreeBSD is wavplay. Several other commands can be used to play sound on FreeBSD systems, see here.

8.2 Parameters Files

Parameters files are plain text files, that can be modified through pychoacoustics or through a text editor. They contain a header with information that applies to all the experimental blocks stored in a parameters file, and sections corresponding to the parameters that are specific to each experimental block store in a parameters file. The header contains the following fields:

- Phones
- Shuffle Mode
- Response Mode
- Auto Resp. Mode Perc. Corr.
- Sample Rate
- Bits
- Trigger On/Off
- Experiment Label

- End Command
- Shuffling Scheme
- No. Repetitions
- Proc. Res.
- Proc. Res. Table
- Plot
- PDF Plot

You can refer to Section *General Widgets (left panel)* to know what each of these fields represents.

The sections that contain the parameters for each experimental block are subdivided into fields that are separated by one or more dots. You should not change this formatting when modifying parameters files.

A fragment from a parameters file is shown below:

```
Paradigm: Adaptive
Intervals: 2 :False
Alternatives: 2 :False
```

each entry here has two or three elements separated by colons. The first element represents the variable of interest, the second element its value, and the third element is a boolean value that determines whether the `inSummary` checkbox will be checked or not (see Section *Result Files* for more info on this). You can have one or more spaces between each element and the colon separator. Each entry has to be written on a single line.

8.3 Block Presentation Position

We will define the serial position at which a block is presented during an experimental session as its "presentation position", and the serial position at which a block is stored in a parameters file as its "storage point".

Clicking the "Shuffle" button randomises the presentation positions of the blocks, but leaves the order in which the blocks are stored in a parameters file untouched. The "Previous" and "Next" buttons, as well as the "Jump to Block" chooser let you navigate across the blocks storage points, while the "Previous Position", and the "Next Position" buttons, as well as the "Jump to Position" chooser let you navigate across the blocks presentation positions.

The block presentation positions are recorded in the parameters files. This is useful in case you have to interrupt an experimental session whose block presentation positions had been randomized, before it is finished, and continue it at a later date. In this case you can save the parameters file, reload it next time, and let the listener complete the experimental blocks that s/he had not run because of the interruption. Notice that each time you load a parameters file `pychoacoustics` will automatically move to the first block presentation position. Therefore, you will have to note down what was the last block that your listener had run in the interrupted session (or find out by looking at the results file) and move to the presentation position of the following block yourself.

By default clicking on the "Shuffle" button performs a simple full randomization of the block presentation positions. However, you can specify more complex shuffling schemes in the "Shuffling Scheme" text field. Let's say you want to present two tasks in your experiment, a frequency discrimination and an intensity discrimination task. Each task has four subconditions, (e.g. four different base frequencies for the frequency discrimination task and four different base intensities for the intensity discrimination task). Your parameters file will contain eight blocks in total, blocks one to four are for the frequency discrimination task and blocks five to eight are for the intensity discrimination task. During the experiment you want your participants to run first the four frequency discrimination conditions in random order, and afterwards the four intensity discrimination conditions in random order. To achieve this you can enter the following shuffling scheme:

```
([1,2,3,4], [5,6,7,8])
```

basically you specify sequences (which can be nested) with your experimental blocks, sequences within round parentheses `()` are not shuffled, while sequences within square brackets `[]` are shuffled. Following the previous example, if you want to present first the four blocks of one of the tasks (either frequency or intensity) in random order, and then the four blocks of the other task in random order, you would specify your shuffling scheme as follows:

```
[[1,2,3,4], [5,6,7,8]]
```

on the other hand, if you want to present first the four blocks of one of the tasks (either frequency or intensity) in sequential order and then the four blocks of the other task in sequential order, you would specify your shuffling scheme as follows:

```
[(1,2,3,4), (5,6,7,8)]
```

you can have any variation you like on the theme, and the lists can be nested ad libitum, so for example you could have:

```
[(1,2,[3,4]), (5,6,7,8)]
```

this would instruct `pychoacoustics` to present first either the four frequency conditions or the four intensity conditions. The first two frequency conditions are presented sequentially, while the last two are shuffled. To save typing you can give ranges rather than listing all blocks individually. For example:

```
([1-4], [5-8])
```

is equivalent to:

```
([1,2,3,4], [5,6,7,8])
```

8.4 Displaying Task Instructions

Although it is common to simply give task instructions verbally for psychophysics experiments, sometimes it is useful to present task instructions on the computer screen while the listener is running a test. For example, there may be cases in which you want to your participants to perform two different tasks within the same session. You may want your participants to perform a frequency discrimination task with a pure tone for the first two blocks of trials, and then run two blocks of an intensity discrimination task with the

same stimulus. In these cases it is necessary to present visually the task instructions on the computer screen either at the beginning of each block, or only at the blocks where the task changes. *pychoacoustics* allows you to store task instructions for each block of trials in the "Instructions" box on the left side of the control window. The "Show Instructions At BP" box below allows you to set the block positions at which the instructions will be shown. In the example above, in which the listener has to complete two blocks of the frequency discrimination task first, and then complete two blocks of the intensity discrimination task, you could input'1,2,3,4' in the "Show Instructions At BP" box to show task instructions at the beginning of each block. Alternatively, you could input'1,3' in the "Show Instructions At BP" box to show task instructions only when a new task is starting. You should keep in mind that the "Show Instructions At BP" box sets the block positions at which the instructions will be shown. Depending on the shuffling scheme that you're using these may be different from the block storage points (see *Block Presentation Position* above for more info).

8.5 OS Commands

`pychoacoustics` can be instructed to run operating system (OS) commands at the end of an experiment. This may be useful to run custom scripts that may analyse the result files, backup result files or perform other operations.

In the control window, you can enter commands that you want to be executed at the end of a specific experiment in the "End Command" box. This command will be saved in the parameters file of the experiment.

In the "Preferences Dialog", under the "Notifications" tab you can instead set a command that will be executed at the end of each experiment you run, or n blocks before the end of each experiment you run. These commands should be entered in the "Execute custom command" boxes.

The commands that you can execute are OS commands, therefore they are different on Linux and Windows platforms. On Linux, for example, assuming that you store all your experimental results in the directory "/home/foo/exp/", you could automatically make a backup of these files in the directory "/home/foo/backup/exp/" by using the command

```
$ rsync -r -t -v --progress -s /home/foo/exp/ /home/foo/backup/exp/
```

To make things more interesting, you can use some special strings to pass `pychoacoustics` internal variables to your commands. For example, if you want to copy the results file of the current experiment to the directory "/home/foo/res/", you can use the command

```
$ cp [resFile] /home/foo/backup/exp/
```

here the special string `[resFile]` will be converted to the name of the file where `pychoacoustics` has saved the data. To make sure that the command executes without errors even if the name of the result file contains white spaces you should put the variable referring to the filename between quotes:

```
$ cp "[resFile]" /home/foo/backup/exp/
```

A full listing of the internal `pychoacoustics` variables that can be called by special strings in your commands is given in Table tab-pycho_variables

8.6 Preferences Settings

All the settings that can be manipulated in the "Preferences" dialog, as well as the "Phones" and "Experimenters" dialogs are stored in a file in the user home directory. On Linux this file is located in:

```
~/.config/pychoacoustics/preferences.py
```

On Windows, assuming the root drive is "C" it is located in:

```
C:\\Users\username\.config/pychoacoustics\preferences.py
```

where `username` is your Windows login username. Although I strive to avoid this, the way in which the preferences settings are stored may change in newer versions of pychoacoustics. This means that when pychoacoustics is upgraded to a newer version it may sometimes not start or throw out errors. To address these issues, please, try removing the old preferences file. Of course this means that you're going to lose all the settings that you had previously saved. To avoid loosing any precious information, such as the calibration values of your headphones, write down all important info before removing the preferences file.

8.7 Response Mode

`pychoacoustics` was designed to run interactive experiments in which a listener hears some stimuli and gives a response through a button or key press. This is the default mode, called "Real Listener" mode. `pychoacoustics` provides two additional response modes, "Automatic" and "Simulated Listener". These modes can be set through the control window.

In "Automatic" response mode, rather than waiting for the listener to give a response, `pychoacoustics` gives itself a response and proceeds to the next trial. The probability that this automatic response is correct can also be set through the control window. The "Automatic" response mode has two main functions. The first is testing and debugging an experiment. Rather than running the experiment yourself, you can launch `pychoacoustics` in "Automatic" response mode and check that everything runs smoothly, the program doesn't crash, and the result files are saved correctly. The second function of the automatic response mode is to allow passive presentation of the stimuli. Some neuroimaging experiments (e.g. electroencephalographic or functional magnetic resonance recordings) are performed with listeners passively listening to the stimuli. These experiments usually also require that the program presenting the stimuli sends triggers to the recording equipment to flag the start of a trial. Potentially this can also be done in `pychoacoustics` (and we've done it in our lab for electroencephalographic recordings), but at the moment this functionality is not implemented in a general way in the program.

The "Simulated Listener" mode is simply a hook that allows you to redirect the control flow of the program to some code that simulates a listener and provides a response. Notice that `pychoacoustics` does not provide any simulation code in itself, the simulation code has to be written by you for a specific experiment. If no simulation code is written in the experiment file, `pychoacoustics` will do nothing in simulated listenr mode. Further details on how to use the "Simulated Listener" mode are provided in Section *Simulations*.

Both the "Automatic" and the "Simulated Listener" make recursive function calls. In Python the number of recursive function calls that you can make is limited. If your experiment passes this limit `pychoacoustics` will crash. The limit can be raised, up to a certain extent (which is dependent on

your operating system, see the documentation for the setrecursionlimit function in the Python `sys` module) through the "Max Recursion Depth" setting that you can find in the preferences window, or set through a command line option when running `pychoacoustics` from the command line. Notice that the total number of recursive calls that your program will make to complete an experiments will be higher than the number of trials in the experiment, so you should set the "Max Recursion Depth" to a value higher than the number of trials you're planning to perform (how much higher I don't know, you should find out by trial and error, a few hundred points higher is usually sufficient). If you're planning to run a very high number of trials in "Automatic" or "Simulated Listener" mode, rather than raising the max recursion depth, it may be better to split the experiment in several parts. You can always write a script that automatically launches `pychoacoustics` from the command line instructing it to load a given parameters file. On UNIX machines you could write a shell script to do that, but an easier way is perhaps to use python itself to write the script. For example, the `python` script could be:

```
#! /usr/bin/env python
for i in range(5):
   cmd = "pychoacoustics --file prms.prm -l L1 -s s1 -q -a \
         --recursion-depth 3000"
```

here we're telling `pychoacoustics` to load the parameters file `prms.prm`, set the listener identifier to "L1" and the session label to s1. The `-q` option instructs the program to exit at the end of the experiment. This way the recursion depth count is effectively restarted each time `pychoacoustics` is closed and launched again from the script. When the `--recursion-depth` option is passed as a command line argument, as in the example above, it overrides the max recursion depth value set in the preferences window. If the `-a` option is passed, as in the examples above, `pychoacoustics` will start automatically at the beginning of each of the five series . This is useful for debugging or simulations, so that you can start the script and leave the program complete unattended (you need to make sure that the "Shuffling Mode" is not set to "Ask" and that you pass listener and session labels if you want the program to run completely unattended).

CHAPTER 9

Writing your own Experiments

9.1 First Steps

pychoacoustics can be easily extended with new experiments written by users. User-written experiments need to reside in a Python package called labexp, and this package needs to be in your Python path. No worries if you're not familiar with packaging Python software, we'll go through the process of adding a new experiment step by step.

First of all, you need to create a directory called pychoacoustics_exp inside your home directory, and a sub-directory called labexp inside the pychoacoustics_exp directory. If you don't know where your home directory is located you can find out from a Python shell with the following commands:

```
import os
os.path.expanduser('~')
```

You can create the pychoacoustics_exp and labexp directories from a Python shell as shown below:

```
import os
dirPath = os.path.expanduser('~/pychoacoustics_exp/labexp/')
os.makedirs(dirPath)
```

Each user experiment will be written in a single file contained in the labexp directory. Let's imagine we want to create an experiment for a frequency discrimination task. We create a file named freq.py in the labexp directory. In addition to the experiment file we need an additional file that lists all the experiments contained in the labexp directory. This file must be named __init__.py, and in our case it will have the following content:

```
__all__ = ["freq"]
```

here the variable __all__ is simply a Python list with the name of the experiment files. So, if one day

we decide to write a new experiment on, let's say, level discrimination, in a file called `lev.py` we would simply add it to the list in `__init__.py`:

```
__all__ = ["freq",
           "lev"]
```

For people familiar with packaging Python modules it should be clear by now that the `labexp` folder is a Python package containing various modules (the experiment files). If at some point we want to remove an experiment from `pychoacoustics`, for example because it contains a bug that does not allow the program to start, we can simply remove it from the list in `__init__.py`. Let's go back to the `freq.py` file. Here we need to define three functions. For our example the names of these functions would be:

```
initialize_freq()
select_default_parameters_freq()
doTrial_freq()
```

basically the function names consist of a fixed prefix, followed by the name of the experiment file. So, in the case of the level experiment example, written in the file `lev.py`, the three functions would be called:

```
initialize_lev()
select_default_parameters_lev()
doTrial_lev()
```

we'll look at each function in detail in the next section. Briefly, the `initialize_` function is used to set some general parameters and options for our experiment; the `select_default_parameters_` function lists all the widgets (text fields and choosers) of our experiment and their default values; finally, the `doTrial_` function contains the code that generates the sounds and plays them during the experiment.

9.1.1 Anatomy of a `pychoacoustics` experiment file

The `initialize_` function

The `initialize_` function of our frequency discrimination experiment is shown below:

```
1  def initialize_freq(prm):
2    exp_name = "Frequency Discrimination Demo"
3    prm["experimentsChoices"].append(exp_name)
4    prm[exp_name] = {}
5    prm[exp_name]["paradigmChoices"] = ["Transformed Up-Down",
6                                        "Weighted Up-Down",
7                                        "UML",
8                                        "PSI"]
9
10   prm[exp_name]["opts"] = ["hasISIBox", "hasAlternativesChooser",
11                            "hasFeedback"]
12
13   prm[exp_name]['defaultAdaptiveType'] = "Geometric"
14   prm[exp_name]['defaultNIntervals'] = 2
15   prm[exp_name]['defaultNAlternatives'] = 2
16   prm[exp_name]["execString"] = "freq"
```

(continues on next page)

(continued from previous page)

```
17      prm[exp_name]["version"] = "1"
18
19      return prm
```

When the function is called, it is passed a dictionary containing various parameters through the `prm` argument. The function modifies this dictionary by adding the parameters of the experiment, and returns the dictionary back to the main routine.

Let's analyze the function for our experiment. On line 2 we give a label to the experiment, this can be anything we want, except the label of an experiment already existing. On line 3 we add this experiment label to the list of "experimentsChoices". On line 4 we create a new sub-dictionary that has as a key the experiment label. Next we list the paradigms that our experiment supports by creating a `paradigmChoices` key and giving the names of the supported paradigms as a list. The paradigms listed here must be within the set of paradigms supported by `pychoacoustics` (see Section *Available Paradigms* for a description of the paradigms currently supported). In the next line we set an `opts` key containing a list of options. The full list of options that can be set here is described in details in Section *The Experiment "opts"*. In brief, for our experiment we want to have a widget to set the silent interval (ISI) between presentation intervals (`hasISIBox`), a widget to choose the number of response alternatives (`hasAlternativesChooser`), and a widget to set the feedback on or off for a given block of trials (`hasFeedback`).

In the next line we specify `defaultAdaptiveType`, the default type of adaptive track that will be selected when the experiment is loaded, this could be either "Geometric", or "Arithmetic". Specifying a "defaultAdaptiveType" is optional. The type of the adaptive procedure can in any case be changed later by the experimenter in the control window. In the next two lines we specify the default number of intervals, and the default number of alternatives that will be used when the experiment is loaded. Since we have inserted the "hasAlternativesChooser" option, the number of intervals and alternatives can be later changed by the experimenter using the appropriate choosers in the control window. The next line of the `initialize_` function sets the `execString` of our experiment. This must be the name of our experiment file, so in our case `freq`. Finally, we give our experiment a version label. This is optional, but it can be very useful as this version label will be stored in the result files when the experiment is run. This makes it possible to track which version of the experiment was used in a given session.

Before we proceed, a note on the use of a function called `QApplication.translate` is necessary. You may occasionally see this function in `pychoacoustics` experiment files and in this manual. This function serves to translate strings from one language to another. For the moment it doesn't really do much in `pychoacoustics` because string translation is not currently functional for the control window, it is only functional for the response box. This function takes three string arguments, and the text to be translated is the middle argument. For example, in the `initialize_` function above, we could have written `QApplication.translate("", "Transformed Up-Down", "")` instead of `Transformed Up-Down`. You don't need to use this function in your experiments. If you do, you need to import the `QApplication`. How to do this depends on which version of `PyQt` you're using, as shown below:

```
from PyQt4.QtGui import QApplication #if you're using PyQt4
from PySide.QtGui import QApplication #if you're using PySide
from PyQt5.QtWidgets import QApplication #if you're using PyQt5
```

The `select_default_parameters_` function

All the widgets (text fields and choosers) needed for an experiment are defined in the `select_default_parameters_` function. For our frequency discrimination experiment, the function looks as follows:

```
1  def select_default_parameters_freq(parent, paradigm, par):
2
3    field = []
4    fieldLabel = []
5    chooser = []
6    chooserLabel = []
7    chooserOptions = []
8
9    fieldLabel.append("Frequency (Hz)")
10   field.append(1000)
11
12   fieldLabel.append("Difference (%)")
13   field.append(20)
14
15   fieldLabel.append("Level (dB SPL)")
16   field.append(50)
17
18   fieldLabel.append("Duration (ms)")
19   field.append(180)
20
21   fieldLabel.append("Ramps (ms)")
22   field.append(10)
23
24
25   chooserOptions.append(["Right",
26                          "Left",
27                          "Both"])
28   chooserLabel.append("Ear:")
29   chooser.append("Right")
30
31   prm = {}
32   prm['field'] = field
33   prm['fieldLabel'] = fieldLabel
34   prm['chooser'] = chooser
35   prm['chooserLabel'] = chooserLabel
36   prm['chooserOptions'] =  chooserOptions
37
38   return prm
```

The `select_default_parameters_` function accepts three arguments, "parent" is simply a reference to the pychoacoustics application, "paradigm" is the paradigm with which the function has been called, while "par" is a variable that can hold some special values for initializing the function. The use of the "par" argument will be discussed later on when procedures with interleaved tracks will be described. For the time being you should just know that the `select_default_parameters_` should always have this argument. From line three to line seven, we create a series of empty lists. The `field` and `fieldLabel` lists will hold the default values of our text field widgets, and their labels, respectively. The `chooser` and

`chooserLabel` lists will likewise hold the default values of our chooser widgets, and their labels, while the `chooserOptions` list will hold the possible values that our choosers can take. On lines 9 to 29 we populate these lists for our frequency discrimination experiment. From line 31 to line 36 we insert in a dictionary the `field`, `fieldLabel`, `chooser`, `chooserLabel` and `chooserOptions` lists that we previously creaetd and populated. Finally, on line 38, the function returns this dictionary.

The `doTrial_` function

The `doTrial_` function is called each time a trial is started, and is responsible for generating the sounds and presenting them to the listener. The `doTrial_` function for our frequency discrimination experiment is shown below:

```
1  def doTrial_freq(parent):
2
3     currBlock = 'b'+ str(parent.prm['currentBlock'])
4      if parent.prm['startOfBlock'] == True:
5          parent.prm['adaptiveParam'] = \
6            parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index(
   ↪"Difference (%)")]
7          parent.writeResultsHeader('log')
8
9      frequency = \
10       parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index(
   ↪"Frequency (Hz)")]
11     level = \
12       parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index("Level␣
   ↪(dB SPL)")]
13     duration = \
14       parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index("Duration␣
   ↪(ms)")]
15     ramps = \
16       parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index("Ramps␣
   ↪(ms)")]
17     channel = \
18       parent.prm[currBlock]['chooser'][parent.prm['chooserLabel'].index("Ear:
   ↪")]
19     phase = 0
20
21     correctFrequency = frequency + (frequency*parent.prm['adaptiveParam'])/100
22     stimulusCorrect = pureTone(correctFrequency, phase, level, duration,
23                                ramps, channel, parent.prm['sampRate'],
24                                parent.prm['maxLevel'])
25
26     stimulusIncorrect = []
27     for i in range((parent.prm['nIntervals']-1)):
28         thisSnd = pureTone(frequency, phase, level, duration, ramps, channel,
29                            parent.prm['sampRate'], parent.prm['maxLevel'])
30         stimulusIncorrect.append(thisSnd)
31
32     parent.playRandomisedIntervals(stimulusCorrect, stimulusIncorrect)
```

As you can see on the first line, the `doTrial_` function is passed as an argument its `parent`. This is impor-

tant because the parent contains a dictionary with the parameters for the current experiment (`parent.prm`). The parameters for each stored block of the experiment are stored in the `parent.prm` dictionary with keys starting with `b` followed by the block number. For example `parent.prm['b3']` contains the parameters for the third stored block. The current block number is stored in `parent.prm['currentBlock']`, and on line 3 we retrieve the dictionary key for the current block. On line 4 we start an `if` block that is executed only at the first trial of each block. In this block we retrieve the % frequency difference between the standard and the comparison stimuli for the first trial, and we store it in the `parent.prm['adaptiveParam']` variable. Since we're using an adaptive procedure, this variable will be automatically increased or decreased by `pychoacoustics` on successive trials on the bases of the responses given by the listener. On line 7 we tell `pychoacoustics` to write the header of the 'log' result files (see *Log Results Files*).

On lines 9-16 we read off the values of the text field widgets for the current block of trials. The values of these field widgets are stored in the list `parent.prm[currBlock]['field']`, and we exploit the label of each text field widget to retrieve its index in the list. For example `parent.prm['fieldLabel'].index("Frequency (Hz)")` retrieves the index of the text widget that stores the frequency of the standard tone for the current block of trials. On line 18 we read off the value of the only chooser widget for the current block of trials. The values of chooser widgets are stored in the list `parent.prm[currBlock]['chooser']`, and we exploit the label of each chooser widget to retrieve its index in the list as we did for text field widgets.

Our next step will be to generate the stimuli for the trial. In a *X*-Intervals task we have to generate *X* stimuli. In our case, the standard stimuli will have always the same frequency, we retrieved its value on lines 9-10 of our `doTrial_` function. If a listener presses the button corresponding to one of the the standard stimuli his response will be incorrect. For this reason we will store the standard stimuli in a list called `stimulusIncorrect = []`. The comparison stimulus will be instead stored in a variable called `stimulusCorrect`. The frequency of the comparison stimulus, which can vary from trial to trial, depending on the current value of `parent.prm['adaptiveParam']` is computed on line 21. On lines 22-24 we generate the stimulus using the `pureTone` function that is available in the `sndlib` module. Note that in order to access this function you need to import it by adding the following line at the top of the `freq.py` file where the experiment is stored:

```
from pychoacoustics.sndlib import pureTone
```

Note also that we need to pass the current samplig rate and the current maximum output level of our headphones (see *Edit Phones Dialog*) to the `pureTone` function. Their values are stored respectively in the `parent.prm['sampRate']` and `parent.prm['maxLevel']` variables. On lines 26-30 we generate and store the standard stimuli in the `stimulusIncorrect` list. The number of standard stimuli to generate will be equal to the number of intervals minus one. The number of intervals is stored in the `parent.prm['nIntervals']` variable. Finally on line 32 we call the `parent.playRandomisedIntervals` function to play the stimuli. This function requires two arguments, the correct stimulus, and a list containing the incorrect stimuli. That's it, our frequency discrimination experiment is ready and we can test it on `pychoacoustics`.

Adding support for the Constant Paradigm

So far our frequency discrimination experiment supports only adaptive paradigms.

Adding support for the constant paradigm, in which the frequency difference between the standard and comparison stimuli is fixed across a block of trials is easy. All we need to do is add "Constant m-Intervals

n-Alternatives" to the list of paradigms supported paradims in the `initialize_` function:

```
prm[exp_name]["paradigmChoices"] = ["Transformed Up-Down",
                                    "Weighted Up-Down",
                                    "UML",
                                    "PSI"
                                    "Constant m-Intervals n-Alternatives"]
```

Now our frequency discrimination task supports also the constant paradigm.

Showing/Hiding Widgets Dynamically

Often you may want to write a single experiment that can handle a number of different experimental conditions. This usually leads to a growing number of widgets in the control window that can be distracting. To address this issue, in `pychoacoustics` it is possible to dinamically show or hide widgets depending on the value taken by chooser widgets. To do this, you need to write a function called `get_fields_to_hide_` that specifies the conditions upon which certain widgets are shown or hidden.

For a practical example, let's extend the frequency discrimination experiment described in the sections above so that it can handle not only conditions in which the standard frequency is fixed, but also conditions in which the standard frequency is roved from trial to trial within a specified frequency range. In the `select_default_parameters_` function of our frequency discrimination experiment we had a text field for setting the standard frequency:

```
fieldLabel.append("Frequency (Hz)")
field.append(1000)
```

now we'll add two additional text fields to set the frequency range for the roved-frequency case:

```
fieldLabel.append("Frequency (Hz)")
field.append(1000)

fieldLabel.append("Min. Frequency (Hz)")
field.append(250)

fieldLabel.append("Max. Frequency (Hz)")
field.append(4000)
```

we also add a chooser to control whether for the current block the standard frequency should be fixed or roved:

```
chooserOptions.append(["Fixed",
                       "Roved"])
chooserLabel.append("Standard Frequency:")
chooser.append("Fixed")
```

The `get_fields_to_hide_` for this experiment is shown below:

```
1 def get_fields_to_hide_freq(parent):
2     if parent.chooser[parent.prm['chooserLabel'].index("Standard Frequency:")].
  ↪currentText() == "Fixed":
```

(continues on next page)

(continued from previous page)

```
3       parent.fieldsToHide = [parent.prm['fieldLabel'].index("Min. Frequency
↪(Hz)"),
4                              parent.prm['fieldLabel'].index("Max. Frequency
↪(Hz)")]
5       parent.fieldsToShow = [parent.prm['fieldLabel'].index("Frequency (Hz)")]
6    elif parent.chooser[parent.prm['chooserLabel'].index("Standard Frequency:
↪")].currentText() == "Roved":
7       parent.fieldsToHide = [parent.prm['fieldLabel'].index("Frequency (Hz)")]
8       parent.fieldsToShow = [parent.prm['fieldLabel'].index("Min. Frequency
↪(Hz)"),
9                              parent.prm['fieldLabel'].index("Max. Frequency
↪(Hz)")]
```

As for the other experiment functions that we have discussed before, the actual name is the concatenation of a prefix, in this case `get_fields_to_hide_`, and the name of the experiment file, in this case `freq`. As you can see on line 1, this function takes as an argument `parent`, which contains the lists of widgets for the current experiment. We need to tell the `get_fields_to_hide_` function that if the standard frequency is fixed, it should show only the `Frequency (Hz)` text field, and hide the `Min. Frequency (Hz)` and `Max. Frequency (Hz)` text fields. Vice-versa, if the standard frequency is roved, it should show only the `Min. Frequency (Hz)` and `Max. Frequency (Hz)` text fields, and hide the `Frequency (Hz)` text field. On line 2 we start an `if` block which will be executed if the value of the `Standard Frequency` chooser (retrieved by the `currentText` attribute), is set to `Fixed`. Note how we exploit once again the `chooserLabel` to find the index of the chooser we want with `parent.prm['chooserLabel'].index("Standard Frequency:")`. Next, we define two lists, one containing the indexes of the fields to hide `parent.fieldsToHide`, and one containing the indexes of the fields to show `parent.fieldsToShow`. Once more we exploit the `fieldLabel` to retrieve the indexes of the fields we want to get (e.g. `parent.prm['fieldLabel'].index("Min. Frequency (Hz)")`). From line 6 to line 9 we handle the case in which the standard frequency is roved. The logic of the code is the same as for the fixed standard frequency case.

To complete the experiment we need to add a couple of lines to the `doTrial_` function to handle the case in which the standard frequency is roved. The new function is shown below:

```
1  def doTrial_freq2(parent):
2     currBlock = 'b'+ str(parent.prm['currentBlock'])
3     if parent.prm['startOfBlock'] == True:
4        parent.prm['adaptiveParam'] = \
5          parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index(
↪"Difference (%)")]
6        parent.writeResultsHeader('log')
7
8     frequency = \
9       parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index("Frequency
↪(Hz)")]
10    minFrequency = \
11      parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index("Min.
↪Frequency (Hz)")]
12    maxFrequency = \
13      parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index("Max.
↪Frequency (Hz)")]
```

(continues on next page)

(continued from previous page)

```
14    level = \
15      parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index("Level (dB␣
↪SPL)")]
16    duration = \
17      parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index("Duration␣
↪(ms)")]
18    ramps = \
19      parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index("Ramps (ms)
↪")]
20    phase = 0
21    channel = \
22      parent.prm[currBlock]['chooser'][parent.prm['chooserLabel'].index("Ear:
↪")]
23    stdFreq = \
24       parent.prm[currBlock]['chooser'][parent.prm['chooserLabel'].index(
↪"Standard Frequency:")]
25
26    if stdFreq == "Roved":
27       frequency = random.uniform(minFrequency, maxFrequency)
28    correctFrequency = frequency + (frequency*parent.prm['adaptiveParam'])/100
29    stimulusCorrect = pureTone(correctFrequency, phase, level, duration,
30                               ramps, channel, parent.prm['sampRate'],
31                               parent.prm['maxLevel'])
32
33    stimulusIncorrect = []
34    for i in range((parent.prm['nIntervals']-1)):
35       thisSnd = pureTone(frequency, phase, level, duration, ramps, channel,
36                          parent.prm['sampRate'], parent.prm['maxLevel'])
37       stimulusIncorrect.append(thisSnd)
38    parent.playRandomisedIntervals(stimulusCorrect, stimulusIncorrect)
```

On lines 10-13 we read off the minimum and maximum frequency values for the roved-standard case. On line 23-24 we retrieve the value of the `Standard Frequency:` chooser. On lines 26-27 we state that if the value of the standard frequency chooser is equal to `Roved`, then the standard frequency for that trial should be drawn from a uniform distribution ranging from `minFrequency` to `maxFrequency`. The rest of the function is unchanged. Note that we're using the a Python module called `random` on line 27, so we need to add `import random` at the top of our `freq.py` file.

It is also possible to show/hide choosers. Let's extend the frequency-discrimination experiment by allowing for the possibility that the standard frequency is roved on a log scale (which in fact would be a better choice given that frequency scaling in the auditory system is approximately logarithmic). To do this, we first add a new chooser to set the roving scale:

```
chooserOptions.append(["Linear",
                       "Log"])
chooserLabel.append("Roving Scale:")
chooser.append("Linear")
```

Because this chooser is useful only when the standard frequency is roved, we'll tell the `get_fields_to_hide_` function to show/hide it depending on the value of the `Standard Frequency` chooser. The new `get_fields_to_hide_` function is shown below:

```
1  def get_fields_to_hide_freq(parent):
2     if parent.chooser[parent.prm['chooserLabel'].index("Standard Frequency:")].
   ↪currentText() == "Fixed":
3        parent.fieldsToHide = [parent.prm['fieldLabel'].index("Min. Frequency␣
   ↪(Hz)"),
4                               parent.prm['fieldLabel'].index("Max. Frequency␣
   ↪(Hz)")]
5        parent.fieldsToShow = [parent.prm['fieldLabel'].index("Frequency (Hz)")]
6        parent.choosersToHide = [parent.prm['chooserLabel'].index("Roving Scale:
   ↪")]
7     elif parent.chooser[parent.prm['chooserLabel'].index("Standard Frequency:
   ↪")].currentText() == "Roved":
8        parent.fieldsToHide = [parent.prm['fieldLabel'].index("Frequency (Hz)")]
9        parent.fieldsToShow = [parent.prm['fieldLabel'].index("Min. Frequency␣
   ↪(Hz)"),
10                              parent.prm['fieldLabel'].index("Max. Frequency␣
   ↪(Hz)")]
11       parent.choosersToShow = [parent.prm['chooserLabel'].index("Roving Scale:
   ↪")]
```

We've just added two lines. Line 6 gets executed if the `Standard Frequency` chooser is set to `Fixed`, and adds the `Roving Scale` chooser to the `parent.choosersToHide` list. Line 11 gets executed if the `Standard Frequency` chooser is set to `Roved`, and adds the `Roving Scale` chooser to the `parent.choosersToShow` list.

Finally, we need to add/modify a couple of lines of the `doTrial_` function. First of all we need to read off the value of the new `Roving Scale` chooser:

```
rovingScale = \
  parent.prm[currBlock]['chooser'][parent.prm['chooserLabel'].index("Roving␣
↪Scale:")]
```

second, we need to set the standard frequency depending on whether it is drawn from a linear or a logarithmic distribution:

```
if stdFreq == "Roved":
   if rovingScale == "Linear":
      frequency = random.uniform(minFrequency, maxFrequency)
   elif rovingScale == "Log":
      frequency = 10**(random.uniform(log10(minFrequency),␣
↪log10(maxFrequency)))
```

Note that we're using the `log10` function from numpy here, so we need to add `from numpy import log10` at the top of our `freq.py` file.

9.2 Writing a "Constant 1-Interval 2-Alternatives" Paradigm Experiment

In the next paragraphs we'll see an example of an experiment using the "Constant 1-Interval 2-Alternatives" paradigm. The experiment a is simple "Yes/No" signal detection task. On each trial the listener is presented

with a single interval which may or may not contain a pure tone, and s/he has to tell if the tone was present or not.

The `initialize_` function for the signal detection experiment is shown below, since the general framework for writing an experiment is the same as for the adaptive paradigm, only the differences from an adaptive-paradigm experiment will be highlited.

```
1  def initialize_sig_detect(prm):
2     exp_name = "Signal Detection Demo"
3     prm["experimentsChoices"].append(exp_name)
4     prm[exp_name] = {}
5     prm[exp_name]["paradigmChoices"] = ["Constant 1-Interval 2-Alternatives"]
6     prm[exp_name]["opts"] = ["hasFeedback"]
7     prm[exp_name]["buttonLabels"] = ["Yes", "No"]
8     prm[exp_name]['defaultNIntervals'] = 1
9     prm[exp_name]['defaultNAlternatives'] = 2
10
11    prm[exp_name]["execString"] = "sig_detect"
12    return prm
```

On line 5 we list the available paradigms for the experiment, in this case the only paradigm possible is `Constant 1-Interval 2-Alternatives`. On line 7 we insert `hasFeedback` to the list of experiment options, so that feedback can be provided at the end of each trial. Since we'll have a single observation interval we don't add the `hasISIBox` option, because we don't need to have a silent inteval between observation intervals. On line 7, we set the labels for the buttons, which represent the two response alternatives: "Yes" or "No". On line 8 and line 9 we set the number of intervals and the number of response alternatives.

The `select_default_parameters_` function for the signal detection experiment is shown below:

```
1  def select_default_parameters_sig_detect(parent, par):
2
3     field = []
4     fieldLabel = []
5     chooser = []
6     chooserLabel = []
7     chooserOptions = []
8
9     fieldLabel.append(parent.tr("Frequency (Hz)"))
10    field.append(1000)
11
12    fieldLabel.append(parent.tr("Duration (ms)"))
13    field.append(2)
14
15    fieldLabel.append(parent.tr("Ramps (ms)"))
16    field.append(4)
17
18    fieldLabel.append(parent.tr("Level (dB SPL)"))
19    field.append(30)
20
21    chooserOptions.append([parent.tr("Right"), parent.tr("Left"), parent.tr(
↪"Both")])
```

(continues on next page)

(continued from previous page)

```
    chooserLabel.append(parent.tr("Channel:"))
    chooser.append(parent.tr("Both"))

    prm = {}
    prm['field'] = field
    prm['fieldLabel'] = fieldLabel
    prm['chooser'] = chooser
    prm['chooserLabel'] = chooserLabel
    prm['chooserOptions'] =  chooserOptions

    return prm
```

there is nothing really new here compared to experiments with adaptive paradigms that we have seen before. We initialize the text fields that we need in order to set the frequency duration and level of the signal. We also initialize a chooser to set the channels on which the signal should be presented.

The `doTrial_` function for the signal detection task is shown below:

```
def doTrial_sig_detect(parent):

    currBlock = 'b'+ str(parent.prm['currentBlock'])
    if parent.prm['startOfBlock'] == True:
        parent.writeResultsHeader('log')
        parent.prm['conditions'] = ["Yes","No"]

    parent.currentCondition = random.choice(parent.prm['conditions'])
    if parent.currentCondition == 'Yes':
        parent.correctButton = 1
    elif parent.currentCondition == 'No':
        parent.correctButton = 2

    freq    = parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index(
↪"Frequency (Hz)")]
    dur     = parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index(
↪"Duration (ms)")]
    ramps   = parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index(
↪"Ramps (ms)")]
    lev     = parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index(
↪"Level (dB SPL)")]
    phase   = 0
    channel = parent.prm[currBlock]['chooser'][parent.prm['chooserLabel'].
↪index(parent.tr("Channel:"))]

    if parent.currentCondition == 'No':
        lev = -200
    sig = pureTone(freq, phase, lev, dur, ramps, channel, parent.prm['sampRate
↪'], parent.prm['maxLevel'])


    parent.playSequentialIntervals([sig])
```

For experiments using the "Constant 1-Interval 2-Alternatives" paradigm it is necessary to list the experi-

mental conditions in the `doTrial_` function. We do this on line 6. On line 8, we bind the response buttons to the correct response. Since the button number 1 is the "Yes" button, we say that in the case of a signal trial (`parent.currentCondition == "Yes"`) the correct button to press is the button number 1, otherwise the correct button to press is the button number 2.

On lines 14-23 we read off the values of the text fields and generate the sound to play (signal or silence) according to the experimental condition. Finally, on line 25 we use the `parent.playSequentialIntervals` function to present the sound to the listener. This function accepts as an argument a list of sounds to play sequentially. In our case we have only a single sound to insert in the list. More details on the `playSequentialIntervals` function are provided in Section *The Play Sound Functions*.

9.3 Writing an experiment for the Transformed Up-Down Interleaved, Weighted Up-Down Interleaved, and Multiple Constants m-Intervals n-Alternatives Paradigms

This section will walk you through an example of an experiment that can be used with the transformed up-down interleaved and weighted up-down interleaved paradigms. These paradigms are simple extensions of the transformed up-down and weighted up-down paradigms in which multiple independent adaptive tracks are run simultaneously and are randomly interleaved in a single block of trials.

Because experiments that support the transformed up-down interleaved and weighted up-down interleaved paradigms can be easily modified to support also the multiple constants m-intervals n-alternatives paradigm, this paradigm will be also added in our example experiment. This paradigm is a simple extension of the constant m-intervals n-alternatives paradigm, in which rather than having a single constant difference between the standard and comparison tones, multiple constant differences are tested in a single block of trials.

The example experiment that we'll look at is a simple signal detection in quiet experiment, that could be used to measure an audiogram. For this reason it is called "Demo Audiogram Multiple Frequencies" (it can be found in the file `audiogram_mf.py` in the `default_experiments` folder). The experiment can be used to setup a virtually unlimited number of adaptive tracks, and each track can be used to track the signal-detection threshold for a specific frequency.

As for the multiple constants procedure, the experiment could be similarly used to measure percent correct performance for tones of different frequencies presented at the same level. However, a more interesting possibility is to use the experiment to measure percent correct performance for the same frequency at different fixed levels. This could then be used to derive a psychometric function relating percent correct performance to signal level.

The `initialize_` function of the experiment is shown below:

```
1 def initialize_audiogram_mf(prm):
2     exp_name = QApplication.translate("","Demo Audiogram Multiple Frequencies",
  ↪"")
3     prm["experimentsChoices"].append(exp_name)
4     prm[exp_name] = {}
5     prm[exp_name]["paradigmChoices"] = [QApplication.translate("","Transformed␣
  ↪Up-Down Interleaved",""),
```

(continues on next page)

(continued from previous page)

```
                                    QApplication.translate("","Weighted Up-Down Interleaved",""),
                                    QApplication.translate("","Multiple Constants m-Intervals n-Alternatives","")]


    prm[exp_name]["opts"] = ["hasISIBox", "hasAlternativesChooser", "hasFeedback",
                             "hasNTracksChooser"]
    prm[exp_name]['defaultAdaptiveType'] = QApplication.translate("","Arithmetic","")
    prm[exp_name]['defaultNIntervals'] = 2
    prm[exp_name]['defaultNAlternatives'] = 2
    prm[exp_name]['defaultNTracks'] = 4

    prm[exp_name]["execString"] = "audiogram_mf"
    prm[exp_name]["version"] = "1"

 return prm
```

the first part of the function doesn't need much explanation if you've follwed the previous examples. The experiments `opts` has a new item `hasNTracksChooser`. This option allows the user to dynamically change the number of adaptive tracks to be used (or the number of constant differences to measure for the multiple constants paradigm). Besides this, the only new thing compared to previous examples is that we also specify the default number of tracks with `prm[exp_name]['defaultNTracks'] = 4`.

The `select_default_parameters_` for the "Demo Audiogram Multiple Frequencies" experiment is shown below:

```
def select_default_parameters_audiogram_mf(parent, par):

    nDifferences = par['nDifferences']

    field = []
    fieldLabel = []
    chooser = []
    chooserLabel = []
    chooserOptions = []

    for i in range(nDifferences):
        fieldLabel.append(parent.tr("Frequency (Hz) " + str(i+1)))
        field.append(1000+1000*i)
        fieldLabel.append(QApplication.translate("","Level (dB SPL) " + str(i+1),""))
        field.append(50)

    fieldLabel.append(QApplication.translate("","Bandwidth (Hz)",""))
    field.append(10)

    fieldLabel.append(QApplication.translate("","Duration (ms)",""))
    field.append(180)
```

(continues on next page)

(continued from previous page)

```

   fieldLabel.append(QApplication.translate("","Ramps (ms)",""))
   field.append(10)


   chooserOptions.append([QApplication.translate("","Right",""),
                          QApplication.translate("","Left",""),
                          QApplication.translate("","Both","")])
   chooserLabel.append(QApplication.translate("","Ear:",""))
   chooser.append(QApplication.translate("","Right",""))
   chooserOptions.append([QApplication.translate("","Sinusoid",""),
                          QApplication.translate("","Narrowband Noise","")])
   chooserLabel.append(QApplication.translate("","Type:",""))
   chooser.append(QApplication.translate("","Sinusoid",""))

   prm = {}
   prm['field'] = field
   prm['fieldLabel'] = fieldLabel
   prm['chooser'] = chooser
   prm['chooserLabel'] = chooserLabel
   prm['chooserOptions'] =  chooserOptions

   return prm
```

The transformed/weighted up-down interleaved paradigms can be run with any number of adaptive tracks. Similarly, the multiple constants m-intervals n-alternatives procedure can be run with any number of constant differences between the standard and comparison intervals. All the user has to do is select the desired number of adaptive tracks, or constant differences from the appropriate chooser in the `pychoacoustics` control window. `select_default_parameters_` function, however, needs to know how many tracks or how many constant differences are being run in order to set up the necessary fields storing the experimental variables. The `par` argument that is always passed to the `select_default_parameters_` function has the purpose of passing additional parameters to dinamycally modify the behavior of the function in cases like this.

In the case of paradigms with interleaved tracks, or multiple constant differences the `par` argument has a key called `nDifferences` that specifies the number of tracks or constant differences. For the current experiment we retieve this value on line 3. Then, on lines 11-15 we set up a for loop in which we add a field to store the frequency and level of the tones for each adaptive track. The rest of the function is similar to previous examples, so it will not be discussed further.

The `get_fields_to_hide_` function for the "Demo Audiogram Multiple Frequencies" experiment is shown in the code block below. Again, nothing new here.

```
def get_fields_to_hide_audiogram_mf(parent):
   if parent.chooser[parent.prm['chooserLabel'].index(QApplication.translate("
↪","Type:",""))].currentText() == QApplication.translate("","Sinusoid",""):
      parent.fieldsToHide = [parent.prm['fieldLabel'].index(QApplication.
↪translate("","Bandwidth (Hz)",""))]
   else:
      parent.fieldsToShow = [parent.prm['fieldLabel'].index(QApplication.
↪translate("","Bandwidth (Hz)",""))]
```

The doTrial_ function for the "Demo Audiogram Multiple Frequencies" experiment is shown below:

```
def doTrial_audiogram_mf(parent):
   currBlock = 'b'+ str(parent.prm['currentBlock'])
   nDifferences = parent.prm['nDifferences']
   if parent.prm['startOfBlock'] == True:
      parent.prm['additional_parameters_to_write'] = {}
      parent.prm['conditions'] = []
      parent.prm['adaptiveParam'] = []
      for i in range(nDifferences):
         parent.prm['conditions'].append(str(parent.prm[currBlock]['field
↪'][parent.prm['fieldLabel'].index(QApplication.translate("","Frequency (Hz)
↪" + str(i+1),""))]))
         parent.prm['adaptiveParam'].append(parent.prm[currBlock]['field
↪'][parent.prm['fieldLabel'].index(QApplication.translate("","Level (dB SPL)
↪" + str(i+1),""))])
      parent.writeResultsHeader('log')

   frequency = []
   for i in range(nDifferences):
      frequency.append(parent.prm[currBlock]['field'][parent.prm['fieldLabel
↪'].index(QApplication.translate("","Frequency (Hz) " + str(i+1),""))])

   parent.currentCondition = parent.prm['conditions'][parent.prm[
↪'currentDifference']] #this is necessary for counting correct/total trials
   correctLevel = parent.prm['adaptiveParam'][parent.prm['currentDifference']]

   currentFrequency = frequency[parent.prm['currentDifference']]
   bandwidth = parent.prm[currBlock]['field'][parent.prm['fieldLabel'].
↪index(QApplication.translate("","Bandwidth (Hz)",""))]
   phase = 0

   incorrectLevel = -200
   duration = parent.prm[currBlock]['field'][parent.prm['fieldLabel'].
↪index(QApplication.translate("","Duration (ms)",""))]
   ramps = parent.prm[currBlock]['field'][parent.prm['fieldLabel'].
↪index(QApplication.translate("","Ramps (ms)",""))]
   channel = parent.prm[currBlock]['chooser'][parent.prm['chooserLabel'].
↪index(QApplication.translate("","Ear:",""))]
   sndType = parent.prm[currBlock]['chooser'][parent.prm['chooserLabel'].
↪index(QApplication.translate("","Type:",""))]

   if sndType == QApplication.translate("","Narrowband Noise",""):
      if bandwidth > 0:
         parent.stimulusCorrect = steepNoise(currentFrequency-(bandwidth/2),␣
↪currentFrequency+(bandwidth/2), correctLevel - (10*log10(bandwidth)),
                                             duration, ramps, channel, parent.
↪prm['sampRate'], parent.prm['maxLevel'])
      else:
         parent.stimulusCorrect = pureTone(currentFrequency, phase,␣
↪correctLevel, duration, ramps, channel, parent.prm['sampRate'], parent.prm[
↪'maxLevel'])
   elif sndType == QApplication.translate("","Sinusoid",""):
```

(continues on next page)

(continued from previous page)

```
37      parent.stimulusCorrect = pureTone(currentFrequency, phase, correctLevel,
↪ duration, ramps, channel, parent.prm['sampRate'], parent.prm['maxLevel'])
38
39
40    parent.stimulusIncorrect = []
41    for i in range((parent.prm['nIntervals']-1)):
42       thisSnd = pureTone(currentFrequency, phase, incorrectLevel, duration,␣
↪ramps, channel, parent.prm['sampRate'], parent.prm['maxLevel'])
43       parent.stimulusIncorrect.append(thisSnd)
44    parent.playRandomisedIntervals(parent.stimulusCorrect, parent.
↪stimulusIncorrect)
```

note that on line 3 we retrieve the number of adaptive tracks (for adaptive interleaved paradigms), or the number of constant differences (for multiple constant paradigms) that we're currently running. This parameter is stored in the `parent.prm` dictionary.

At the start of a block of trials (cfr. line 4) we set up a number of parameters. Among these there are two in particular that need some explanation. The ‘‘parent.prm[‘adaptiveParam’] on line 7 is a list that is populated in the for loop starting on line 9 with the initial values of the parameter that is adaptively varying for each track. The ‘‘parent.prm[‘conditions’] on the other hand is a parameter that is used only when the experiment is run with the multiple constants m-intervals n-alternatives paradigm. It's a list of labels for each "condition" that is being run in the experiment, that is for each constant difference that is being tested.

On lines 13-15 we retrieve the frequencies of the tones used for each track or constant difference.

On line 17 we retrieve the label of the current condition and store it in the `parent.currentCondition` variable. Thisvariable will be used by `pychoacoustics` for keeping count of the correct and total number of trials for each constant difference when using the multiple constants paradigm. Note how the `parent.prm['currentDifference']` variable is used for this purpose. This variable is the index to the current track or current cosnatnt difference that is being currently tested. This variable is set outside of the `doTrial_` function, (a random track or constant difference is chosen for each trial) but we can retrieve its value through the `parent` handle.

On line 18 we make use of the `parent.prm['currentDifference']` variable again, this time to retrieve the level of the comparison stimulus for the track or constant difference that is run on the current trial. The rest of the function is not different from the `doTrial_` functions used in transformed/weighted up-down paradigms with non-interleaved tracks, and should be easy to follow if you've followed the previous examples.

9.3.1 Writing a matching experiment using interleaved adaptive tracks

The transformed up-down and weighted up-down interleaved procedures can be used to write matching experiments. As described by [Jesteadt1980], two interleaved adaptive tracks can be used to target points on the psychometric function that are symmetric around the 50% point (e.g. 71% and 29%), and then average the threshold in each track in order to estimate the point of subjective equality. For example, in a level-matching experiment one track could target the point at which the listener judges the comparison tone to be louder than the standard tone 71% of the time, while the other track targets the point at which the listener judges the comparison tone to be louder than the standard 29% of the time (or equivalently, softer than the standard 71% of the time).

In this section we'll show how to write in `pychoacoustics` a level-matching experiment similar to the one described by [Jesteadt1980]. This experiment is one of the default experiments available in `pychoacoustics`, and is called `Demo Level Matching`.

The `initialize_` function of the experiment is shown in the code block below.

```
def initialize_lev_match(prm):
   exp_name = "Demo Level Matching"
   prm["experimentsChoices"].append(exp_name)
   prm[exp_name] = {}
   prm[exp_name]["paradigmChoices"] = ["Transformed Up-Down Interleaved",
                                       "Weighted Up-Down Interleaved"]

   prm[exp_name]["opts"] = ["hasISIBox", "hasAlternativesChooser"]
   prm[exp_name]['defaultAdaptiveType'] = QApplication.translate("",
↪"Arithmetic","")
   prm[exp_name]['defaultNIntervals'] = 2
   prm[exp_name]['defaultNAlternatives'] = 2
   prm[exp_name]['defaultNTracks'] = 2
   prm[exp_name]["execString"] = "lev_match"
```

among the `paradigmChoices` we include the "Transformed Up-Down Interleaved", and the "Weighted Up-Down Interleaved". The experiment has just two experiment `opts`: one to add an ISI box, the other one to add an alternatives chooser (we'll probably want to run this experiment only with two intervals, and two alternatives, so in principle we could do without the alternative chooser, but currently, for technical reasons the `hasAlternativesChooser` option has to be added with the "Transformed Up-Down Interleaved", and the "Weighted Up-Down Interleaved" paradigms). Besides specifying the default number of intervals and alternatives, we also specify the default number of interleaved tracks using the `defaultNTracks` key. Because we have not added a `hasNTracksChooser` in the experiment the default number of tracks specified here will be the default and only possible number of tracks in the experiment.

The `select_default_parameters_` function is shown below:

```
def select_default_parameters_lev_match(parent, par):

   field = []
   fieldLabel = []
   chooser = []
   chooserLabel = []
   chooserOptions = []

   fieldLabel.append("Starting Level Track 1 (dB SPL)")
   field.append(75)

   fieldLabel.append("Starting Level Track 2 (dB SPL)")
   field.append(55)

   fieldLabel.append(parent.tr("Frequency Standard Tone (Hz)"))
   field.append(1000)

   fieldLabel.append(parent.tr("Frequency Comparison Tone (Hz)"))
   field.append(250)
```

(continues on next page)

(continued from previous page)

```
    fieldLabel.append(parent.tr("Level Standard Tone (dB SPL)"))
    field.append(65)

    fieldLabel.append(parent.tr("Duration (ms)"))
    field.append(180)

    fieldLabel.append(parent.tr("Ramps (ms)"))
    field.append(10)

    chooserOptions.append(["Right", "Left", "Both"])
    chooserLabel.append(QApplication.translate("","Ear:",""))
    chooser.append(QApplication.translate("","Both",""))


    prm = {}
    prm['field'] = field
    prm['fieldLabel'] = fieldLabel
    prm['chooser'] = chooser
    prm['chooserLabel'] = chooserLabel
    prm['chooserOptions'] =  chooserOptions

    return prm
```

the first two fields will be used to set the starting level of the comparison tone in each track. The next two fields will be used to set the frequencies of the standard and comparison tone. The next field will be used to set the level of the standard tone which will be fixed throughout a block of trials. The last two fields will be used to set the duration of the tone (excluding the ramps), and the duration of its onset and offset ramps. The only chooser will be used to set the ear to which the tones will be presented.

The `doTrial_` function for the level matching experiment is shown below:

```
def doTrial_lev_match(parent):
    currBlock = 'b'+ str(parent.prm['currentBlock'])
    if parent.prm['startOfBlock'] == True:
        parent.prm['adaptiveParam'] = []
        parent.prm['adaptiveParam'].append(parent.prm[currBlock]['field
↪'][parent.prm['fieldLabel'].index("Starting Level Track 1 (dB SPL)")])
        parent.prm['adaptiveParam'].append(parent.prm[currBlock]['field
↪'][parent.prm['fieldLabel'].index("Starting Level Track 2 (dB SPL)")])
        parent.writeResultsHeader('log')



    standardFrequency = parent.prm[currBlock]['field'][parent.prm['fieldLabel'].
↪index("Frequency Standard Tone (Hz)")]
    comparisonFrequency = parent.prm[currBlock]['field'][parent.prm['fieldLabel
↪'].index("Frequency Comparison Tone (Hz)")]
    standardLevel = parent.prm[currBlock]['field'][parent.prm['fieldLabel'].
↪index("Level Standard Tone (dB SPL)")]
    duration = parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index(
↪"Duration (ms)")]
```

(continues on next page)

(continued from previous page)

```
15   ramps = parent.prm[currBlock]['field'][parent.prm['fieldLabel'].index(
↪"Ramps (ms)")]
16   phase = 0
17   channel = parent.prm[currBlock]['chooser'][parent.prm['chooserLabel'].index(
↪"Ear:")]
18
19   comparisonLevel = parent.prm['adaptiveParam'][parent.prm['currentDifference
↪']]
20
21   comparisonTone = pureTone(comparisonFrequency, phase, comparisonLevel,␣
↪duration, ramps,
22                             channel, parent.prm['sampRate'], parent.prm[
↪'maxLevel'])
23
24   standardToneList = []
25   for i in range((parent.prm['nIntervals']-1)):
26      thisSnd = pureTone(standardFrequency, phase, standardLevel, duration,␣
↪ramps, channel,
27                         parent.prm['sampRate'], parent.prm['maxLevel'])
28      standardToneList.append(thisSnd)
29   parent.playRandomisedIntervals(comparisonTone, standardToneList)
```

The adaptive parameter for an experiment with interleaved tracks is not a single number, but a list containing the values of the adaptive parameter for each track. Therefore, on line 4 we create the list, and on lines 5 and 6 we populate this list with the initial values of each of the adaptive tracks.

From lines 11 to 17 we retrieve the values of all the fields and choosers. Nothing new here. On line 19 we retrieve the value of the adaptive parameter (which in this case is the level of the comparison tone) for the current trial. To do this, we refer to a key in the `parent.prm` dictionary called `currentDifference`. This key holds the index of the track which has been randomly selected by `pychoacoustics` for the current trial.

From line 21 to 28 we prepare the stimuli to be presented in the standard and comparison intervals. We then pass these stimuli as arguments to the `playRandomisedIntervals` functions. This experiment is ready to be run.

The up-down rules of the two adaptive tracks need to be set up appropriately to run the matching experiment. Let's, take as an example the experiment described in [Jesteadt1980] in which we wish to determine the intensity of a 250-Hz tone required to match the loudness of a 1000-Hz tone presented at 40 dB SPL. In the `pychoacoustics` control window, after having selected the `Demo Level Matching` experiment, we set the frequency of the standard tone to 1000 Hz, and the frequency of the comparison tone to 250 Hz. We also set the level of the standard tone to 40 dB SPL. We then set the upper, and lower tracks to 60 and 30 dB SPL, two values that should bracket the point of subjective equality.

The task for the listener is an objective one: s/he will have to tell on each trial which tone was louder. For track 1, we set the rule down to 2, and the rule up to 1. For track 2 instead, we set the rule down to 1, and the rule up to 2. In this way, track 1 will target the point in the psychometric function at which the listener judges the comparison tone to be louder than the standard 70.7% of the time. Track 2 will target instead the point in the psychometric function at which the listener judges the comparison tone to be louder than the standard 29.3% of the time. For track 1, when the listener chooses the *comparison* interval twice in a row the level of the 250-Hz tone (the comparison tone) is decreased, while each time s/he chooses the standard

interval the level of the 250-Hz tone is increased. For track 2, when the listener chooses the *standard* interval twice in a row the level of the 250-Hz tone is increased, while each time the listener chooses the level of the 250-Hz tone is decreased. For both tracks "correct" responses move the track down. There are no correct or incorrect responses in a subjective task like this. The `Corr. Resp. Move Track X` (down or up) choosers are not named appropriately for this task. They should be named something like "when the comparison interval is chosen track X moves" down or up. However, since the underlying code for adaptive interleaved paradigms is the same for objective and subjective tasks, for simplicity and ease of maintenance of the underlying code they are called `Corr. Resp. Move Track X` (down or up). .

9.4 Writing a "Constant 1-Pair Same/Different" Paradigm Experiment

Todo: Describe of to write experiments for the "Constant 1-Pair Same/Different" paradigm.

9.5 Writing an "Odd One Out" Paradigm Experiment

Todo: Describe of to write experiments for the "Odd One Out" paradigm.

9.6 The Experiment "opts"

- **''hasAlternativesChooser''** This option adds two chooser widgets, one to dynamically change the number of observation intervals (labelled "Intervals"), and one to dinamically change the number of response alternatives (labelled "Alternatives). This option is generally used in adaptive paradigms ("Transformed Up-Down", "Weighted Up-Down", as well as their interleaved versions). The number of response alternatives that can be choosen from the widget can be either equal to the number of observation intervals, or to the number of observation intervals minus one. In the latter case the standard stimulus is presented in the first interval, as a reference, with no corresponding response alternative, see [GrimaultEtAl2002] for an example of this n-intervals, $n - 1$ alternatives presentation mode. The selected number of intervals and alternatives can be accessed in the experiment file through the `parent.prm['nIntervals']`, and `parent.prm['nAlternatives']` variables respectively.
- **''hasAltReps''** This option can be used to change the way in which the stimuli are presented in the "Transformed Up-Down" paradigm or other adaptive paradigms. In these paradigms, normally there is an observation interval containing the target stimulus (comparison interval), and one or more other intervals containing the non-target stimuli (standard intervals). An alternative way to present the stimuli is to have an alternation of the target and non-target stimuli (e.g. ABAB) in the comparison interval, and a repetition of the non-target stimulus in the standard interval (AAAA) [KingEtAl2013]. If the `hasAltReps` option is enabled, there will be two additional text boxes, `Alternated (AB) Reps.` and `Alternated (AB) Reps. ISI (ms)`. The first text box controls the number of times the alternated target and non-target stimuli should be repeated, a value of zero corresponds to

no alternation, that is only a single stimulus (either the target, or the non target) is presented in each interval. If the value is one, a single alternation will occur (AB), if the value is two, two alternations occur (ABAB), and so on. The second text box controls the ISI between the stimuli presented within an interval. The selected number of alternated repetitions, and the ISI between alternating stimuli can be accessed in the experiment file through the `parent.prm['altReps']`, and `parent.prm['altRepsISI']` variables respectively. The setup of the alternated repetitions must be done within each experiment file.

- **''hasFeedback''** This option controls whether the "Response Light" chooser has a "Feedback" option or not. You may want to enable this option for all "objective" experiments that have a clear "correct" response. You may want to disable this option for "subjective" experiments, such as matching experiments, in which there is no "correct" response.
- **''hasISIBox''** If this option is enabled, a box labelled `ISI (ms)` is added. This is generally used to set the silent period between observation intervals in the "Transformed Up-Down" and similar adaptive procedures. Its value can be accessed in the experiment file through the `parent.prm['isi']` variable. However, normally this should not be necessary because the `playRandomisedIntervals` function automatically uses this value to set the silent period between observation intervals.
- **''hasNDifferencesChooser''** This option is useful in the "Multiple Constants 1-Interval 2-Alternatives Paradigm" to dinamically change the number of experimental conditions. For example, if you have a signal detection experiment in which a fixed number of signals (with a constant amplitude) can occur, this option allows to choose the number of conditions dinamically. If this option is enabled, a chooser labelled `No. Alternatives` is added. The value selected can be accessed through the `par['nDifferences']` variable in the `select_default_parameters_` function, and through the `parent.prm['nDifferences']` variable in the `doTrial` function.
- **''hasNTracksChooser''** This option can be used to dinamically change the number of tracks in interleaved adaptive paradigms (e.g. "Transformed Up-Down Interleaved). If enabled, a `No. Tracks` chooser is added. The value selected can be accessed through the `par['nDifferences']` variable in the `select_default_parameters_` function, and through the `parent.prm['nDifferences']` variable in the `doTrial` function.
- **''hasPrecursorInterval''** If this option is enabled, a chooser controlling whether a precursor interval should be presented or not is added. This chooser is labelled `Precursor Interval`. If this option is enabled, and the chooser is set to "Yes", then a `precursorStim` sound needs to be passed to the `playRandomisedIntervals` function. This sound will be presented before each observation interval.
- **''hasPostcursorInterval''** If this option is enabled, a chooser controlling whether a postcursor interval should be presented or not is added. This chooser is labelled `Postcursor Interval`. If this option is enabled, and the chooser is set to "Yes", then a `postcursorStim` sound needs to be passed to the `playRandomisedIntervals` function. This sound will be presented after each observation interval.
- **''hasPreTrialInterval''** If this option is enabled, a chooser controlling whether a pre-trial interval should be presented or not is added. This chooser is labelled `Pre-Trial Interval`. If this option is enabled, and the chooser is set to "Yes", then a `preTrialStim` sound needs to be passed to the `playRandomisedIntervals` function. This sound will be presented at the beginning of each trial.

9.7 The Play Sound Functions

Todo: Illustrate the functions to play sounds

9.8 Simulations

`pychoacoustics` is not designed to run simulations in itself, however it provides a hook to redirect the control flow to an auditory model that you need to specify yourself in the experiment file. You can retrieve the current response mode from the experiment file with:

```
parent.prm['allBlocks']['responseMode']
```

so, in the experiment file, after the creation of the stimuli for the trial you can redirect the control flow of the program depending on the response mode:

```
if parent.prm['allBlocks']['responseMode'] != "Simulated Listener":
    #we are not in simulation mode, play the stimuli for the listener
    parent.playSoundSequence(sndSeq, ISIs)
if parent.prm['allBlocks']['responseMode'] == "Simulated Listener":
    #we are in simulation mode
    #pass the stimuli to an auditory model and decision device
    #---
    #Here you specify your model, pychoacoustics doesn't do it for you!
    # at the end your simulated listener arrives to a response that is
    # either correct or incorrect
    #---
    parent.prm['trialRunning'] = False
    #this is needed for technical reasons (if the 'trialRunning'
    #flag were set to 'True' pychoacoustics would not process
    #the response.
    #
    #let's suppose that at the end of the simulation you store the
    #response in a variable called 'resp', that can take as values
    #either the string 'Correct' or the string 'Incorrect'.
    #You can then proceed to let pychoacoustics process the response:
    #
    if resp == 'Correct':
        parent.sortResponse(parent.correctButton)
    elif resp == 'Incorrect':
        #list all the possible 'incorrect' buttons
        inc_buttons = numpy.delete(numpy.arange(
                                    self.prm['nAlternatives'])+1,
                                    self.correctButton-1))
        #choose one of the incorrect buttons
        parent.sortResponse(random.choice(inc_buttons))
```

CHAPTER 10

Troubleshooting

10.1 The computer crashed in the middle of an experimental session

`pychoacoustics` saves the results at the end of each block, therefore only the results from the last uncompleted block will be lost, the results of completed blocks will not be lost. If you have an experiment with many different blocks presented in random order it may be difficult to see which blocks the listener had already completed and set `pychoacoustics` to run only the blocks that were not run. To address this issue `pychoacoustics` keeps a copy of the parameters, including the block presentation order after shuffling, in a file called `.tmp_prm.prm` (this is a hidden file on Linux systems). Therefore, after the crash you can simply load this parameters file and move to the block position that the listener was running when the computer crashed to resume the experiment.

A second function of the `.tmp_prm.prm` file is to keep a copy of parameters that were stored in memory, but not saved to a file. If your computer crashed while you were setting up a parameters for an experiment that were not yet saved (or were only partially saved) to a file, you can retrieve them after the crash by loading the `.tmp_prm.prm` file. One important thing to keep in mind is that the `.tmp_prm.prm` will be overwritten as soon as new parameters are stored in memory by a `pychoacoustics` instance opened in the same directory. Therefore it is advisable to make a copy of the `.tmp_prm.prm` file renaming it to avoid accidentally loosing its contents after the crash.

sndlib	A module for generating sounds in python.

CHAPTER 11

sndlib

A module for generating sounds in python.

Functions

AMTone([frequency, AMFreq, AMDepth, phase, ...])	Generate an amplitude modulated tone.
AMToneIPD([frequency, AMFreq, AMDepth, ...])	Generate an amplitude modulated tone with an interaural phase difference (IPD) in the carrier and/or modulation phase.
AMToneVarLev([frequency, AMFreq, AMDepth, ...])	Generate an amplitude modulated (AM) tone.
ERBDistance(f1, f2)	Compute the distance in equivalent rectangular bandwiths (ERBs) between f1 and f2.
FMTone([fc, fm, mi, phase, level, duration, ...])	Generate a frequency modulated tone.
ITDShift(sig, f1, f2, ITD, channel, fs)	Set the ITD of a sound within the frequency region bounded by 'f1' and 'f2'
addSounds(snd1, snd2, delay, fs)	Add or concatenate two sounds.
binauralPureTone([frequency, phase, level, ...])	Generate a pure tone with an optional interaural time or level difference.
broadbandNoise([spectrumLevel, duration, ...])	Synthetise a broadband noise.
camSinFMComplex([F0, lowHarm, highHarm, ...])	Generate a complex tone frequency modulated with an exponential sinusoid.
camSinFMTone([fc, fm, deltaCams, fmPhase, ...])	Generate a tone frequency modulated with an exponential sinusoid.

Continued on next page

Table 1 – continued from previous page

chirp([freqStart, ftype, rate, level, ...])	Synthetize a chirp, that is a tone with frequency changing linearly or exponentially over time with a give rate.
complexTone([F0, harmPhase, lowHarm, ...])	Synthetise a complex tone.
complexToneIPD([F0, harmPhase, lowHarm, ...])	Synthetise a complex tone with an interaural phase difference (IPD).
complexToneParallel([F0, harmPhase, ...])	Synthetise a complex tone.
delayAdd(sig, delay, gain, iterations, ...)	Delay and add algorithm for the generation of iterated rippled noise.
dichoticNoiseFromSin([F0, lowHarm, ...])	Generate Huggins pitch or narrow-band noise from random-phase sinusoids.
expAMNoise([fc, fm, deltaCents, fmPhase, ...])	Generate a sinusoidally amplitude-modulated noise with an exponentially modulated AM frequency.
expSinFMComplex([F0, lowHarm, highHarm, ...])	Generate a frequency-modulated complex tone with an exponential sinusoid.
expSinFMTone([fc, fm, deltaCents, fmPhase, ...])	Generate a frequency-modulated tone with an exponential sinusoid.
fir2Filt(f1, f2, f3, f4, snd, fs)	Filter signal with a fir2 filter.
fm_complex1([midF0, harmPhase, lowHarm, ...])	Synthetise a complex tone with an embedded FM starting and stopping at a chosen time after the tone onset.
fm_complex2([midF0, harmPhase, lowHarm, ...])	Synthetise a complex tone with an embedded FM starting and stopping at a chosen time after the tone onset.
freqFromERBInterval(f1, deltaERB)	Compute the frequency, in Hz, corresponding to a distance, in equivalent rectangular bandwidths (ERBs), of 'deltaERB' from f1.
gate(ramps, sig, fs)	Impose onset and offset ramps to a sound.
getRMS(sig[, channel])	Compute the root mean square (RMS) value of the signal.
glide([freqStart, ftype, excursion, level, ...])	Synthetize a rising or falling tone glide with frequency changing linearly or exponentially.
harmComplFromNarrowbandNoise([F0, lowHarm, ...])	Generate an harmonic complex tone from narrow noise bands.
imposeLevelGlide(sig, deltaL, startTime, ...)	Impose a glide in level to a sound.
intNCyclesFreq(freq, duration)	Compute the frequency closest to 'freq' that has an integer number of cycles for the given sound duration.
itdtoipd(itd, freq)	Convert an interaural time difference to an equivalent interaural phase difference for a given frequency.
joinSndISI(sndList, ISIList, fs)	Join a list of sounds with given interstimulus intervals

Continued on next page

Table 1 – continued from previous page

makeAsynchChord(freqs, levels, phases, ...)	Generate an asynchronous chord.
makeBlueRef(sig, fs, refHz)	Convert a white noise into a blue noise.
makeHugginsPitch([F0, lowHarm, highHarm, ...])	Synthetise a complex Huggings Pitch.
makeIRN([delay, gain, iterations, ...])	Synthetise a iterated rippled noise
makePink(sig, fs)	Convert a white noise into a pink noise.
makePinkRef(sig, fs, refHz)	Convert a white noise into a pink noise.
makeRedRef(sig, fs, refHz)	Convert a white noise into a red noise.
makeSilence([duration, fs])	Generate a silence.
makeVioletRef(sig, fs, refHz)	Convert a white noise into a violet noise.
nextpow2(x)	Next power of two.
phaseShift(sig, f1, f2, phaseShift, ...)	Shift the interaural phases of a sound within a given frequency region.
pinkNoiseFromSin([compLevel, lowCmp, ...])	Generate a pink noise by adding sinusoids spaced by a fixed interval in cents.
pinkNoiseFromSin2([compLevel, lowCmp, ...])	Generate a pink noise by adding sinusoids spaced by a fixed interval in cents.
pureTone([frequency, phase, level, ...])	Synthetise a pure tone.
scale(level, sig)	Increase or decrease the amplitude of a sound signal.
setLevel_(level, snd, maxLevel[, channel])	Set the RMS level of a sound signal to a given value.
spectralModNoise([spectrumLevel, duration, ...])	Generate a broadband noise with a modulated spectral envelope.
steepNoise([frequency1, frequency2, level, ...])	Synthetise band-limited noise from the addition of random-phase sinusoids.

CHAPTER 12

sndlib – Sound Synthesis Library

A module for generating sounds in python.

sndlib.**AMTone**(*frequency=1000*, *AMFreq=20*, *AMDepth=1*, *phase=0*, *AMPhase=0*, *level=60*, *duration=980*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Generate an amplitude modulated tone.

Parameters

frequency [float] Carrier frequency in hertz.

AMFreq [float] Amplitude modulation frequency in Hz.

AMDepth [float] Amplitude modulation depth (a value of 1 corresponds to 100% modulation).

phase [float] Starting phase in radians.

AMPhase [float] Starting AM phase in radians.

level [float] Average tone level in dB SPL. See notes.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel [string ('Right', 'Left' or 'Both')] Channel in which the tone will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats]

Notes

For a fixed base amplitude, the average power of an AM tone (as defined in this function) increases proportionally with AM depth by a factor of 1+AMDepth^2/2 (Viemeister, 1979, Yost et al., 1989, Hartmann, 2004). This function compensates for this average increase in power. You can use the *AMToneVarLev* function if you want to generate AM tones varying in average power with AM depth.

References

[H], [V79], [YSO]

Examples

```
>>> snd = AMTone(frequency=1000, AMFreq=20, AMDepth=1, phase=0,
...     AMPhase=1.5*pi, level=65, duration=180, ramp=10, channel='Both',
...     fs=48000, maxLevel=100)
```

sndlib.**AMToneIPD**(*frequency=1000*, *AMFreq=20*, *AMDepth=1*, *phase=0*, *AMPhase=0*, *phaseIPD=0*, *AMPhaseIPD=0*, *level=60*, *duration=980*, *ramp=10*, *channel=u'Right'*, *fs=48000*, *maxLevel=101*)

Generate an amplitude modulated tone with an interaural phase difference (IPD) in the carrier and/or modulation phase.

Parameters

frequency [float] Carrier frequency in hertz.

AMFreq [float] Amplitude modulation frequency in Hz.

AMDepth [float] Amplitude modulation depth (a value of 1 corresponds to 100% modulation).

phase [float] Starting phase in radians.

AMPhase [float] Starting AM phase in radians.

phaseIPD [float] IPD to apply to the carrier phase.

AMPhaseIPD [float] IPD to apply to the modulation phase.

level [float] Average tone level in dB SPL. See notes.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel [string ('Right', 'Left')] Channel in which the phase will be shifted.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats]

Notes

For a fixed base amplitude, the average power of an AM tone (as defined in this function) increases proportionally with AM depth by a factor of 1+AMDepth^2/2 (Viemeister, 1979, Yost et al., 1989, Hartmann, 2004). This function does not compensate for this average increase in power. You can use the *AMTone* function if you want to generate AM tones matched in average power irrespective of AM depth.

References

[H], [V79], [YSO]

Examples

```
>>> snd = AMToneIPD(frequency=1000, AMFreq=20, AMDepth=1, phase=0,␣
↪AMPhase=1.5*pi,
...     phaseIPD=0, AMPhaseIPD=pi, level=65,
...     duration=180, ramp=10, channel='Right', fs=48000, maxLevel=100)
```

sndlib.**AMToneVarLev**(*frequency=1000*, *AMFreq=20*, *AMDepth=1*, *phase=0*, *AMPhase=0*, *level=60*, *duration=980*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Generate an amplitude modulated (AM) tone.

Parameters

frequency [float] Carrier frequency in hertz.

AMFreq [float] Amplitude modulation frequency in Hz.

AMDepth [float] Amplitude modulation depth (a value of 1 corresponds to 100% modulation).

phase [float] Starting phase in radians.

AMPhase [float] Starting AM phase in radians.

level [float] Average level of the tone in dB SPL when the *AMDepth* is zero. The level of the tone will be higher when *AMDepth* is > zero. See notes.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel [string ('Right', 'Left' or 'Both')] Channel in which the tone will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats]

Notes

For a fixed base amplitude, the average power of an AM tone (as defined in this function) increases proportionally with AM depth by a factor of 1+AMDepth^2/2 (Viemeister, 1979, Yost et al., 1989, Hartmann, 2004). This function does not compensate for this average increase in power. You can use the *AMTone* function if you want to generate AM tones matched in average power irrespective of AM depth.

References

[H], [R47c1c0644075-V79], [YSO]

Examples

```
>>> snd = AMToneVarLev(frequency=1000, AMFreq=20, AMDepth=1, phase=0,
...       AMPhase=1.5*pi, level=65, duration=180, ramp=10, channel='Both
↪',
...       fs=48000, maxLevel=100)
```

sndlib.**ERBDistance**(*f1*, *f2*)

Compute the distance in equivalent rectangular bandwiths (ERBs) between f1 and f2.

Parameters

f1 [float] frequency 1 in Hz

f2 [float] frequency 2 in Hz

Returns

deltaERB [float] distance between f1 and f2 in ERBs

References

[GM]

Examples

```
>>> ERBDistance(1000, 1200)
```

sndlib.**FMTone**(*fc=1000*, *fm=40*, *mi=1*, *phase=0*, *level=60*, *duration=180*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Generate a frequency modulated tone.

Parameters

fc [float] Carrier frequency in hertz. This is the frequency of the tone at fm zero crossing.

fm [float] Modulation frequency in Hz.

mi [float] Modulation index, also called beta and is equal to deltaF/fm, where deltaF is the maximum deviation of the instantaneous frequency from the carrier frequency.

phase [float] Starting phase in radians.

level [float] Tone level in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel ['Right', 'Left' or 'Both'] Channel in which the tone will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats]

Examples

```
>>> snd = FMTone(fc=1000, fm=40, mi=1, phase=0, level=55, duration=180,
...     ramp=10, channel='Both', fs=48000, maxLevel=100)
```

sndlib.**ITDShift**(*sig*, *f1*, *f2*, *ITD*, *channel*, *fs*)

Set the ITD of a sound within the frequency region bounded by 'f1' and 'f2'

Parameters

sig [array of floats] Input signal.

f1 [float] The start point of the frequency region to be phase-shifted in hertz.

f2 [float] The end point of the frequency region to be phase-shifted in hertz.

ITD [float] The amount of ITD shift in microseconds

channel [string ('Right' or 'Left')] The channel in which to apply the shift.

fs [float] The sampling frequency of the sound.

Returns

out [2-dimensional array of floats]

Examples

```
>>> noise = broadbandNoise(spectrumLevel=40, duration=180, ramp=10,
...     channel='Both', fs=48000, maxLevel=100)
>>> hp = ITDShift(sig=noise, f1=500, f2=600, ITD=5,
        channel='Left', fs=48000) #this generates a Dichotic Pitch
```

sndlib.**addSounds**(*snd1*, *snd2*, *delay*, *fs*)

Add or concatenate two sounds.

Parameters

snd1 [array of floats] First sound.

snd2 [array of floats] Second sound.

delay [float] Delay in milliseconds between the onset of 'snd1' and the onset of 'snd2'

fs [float] Sampling frequency in hertz of the two sounds.

Returns

snd [2-dimensional array of floats]

Examples

```
>>> snd1 = pureTone(frequency=440, phase=0, level=65, duration=180,
...     ramp=10, channel='Right', fs=48000, maxLevel=100)
>>> snd2 = pureTone(frequency=880, phase=0, level=65, duration=180,
...     ramp=10, channel='Right', fs=48000, maxLevel=100)
>>> snd = addSounds(snd1=snd1, snd2=snd2, delay=100, fs=48000)
```

sndlib.**binauralPureTone**(*frequency=1000*, *phase=0*, *level=60*, *duration=980*, *ramp=10*, *channel=u'Both'*, *itd=0*, *itdRef=u'Right'*, *ild=10*, *ildRef=u'Right'*, *fs=48000*, *maxLevel=101*)

Generate a pure tone with an optional interaural time or level difference.

Parameters

frequency [float] Tone frequency in hertz.

phase [float] Starting phase in radians.

level [float] Tone level in dB SPL. If 'ild' is different than zero, this will be the level of the tone in the reference channel.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel [string ('Right', 'Left' or 'Both')] Channel in which the tone will be generated.

itd [float] Interaural time difference, in microseconds.

itdRef ['Right', 'Left' or None] The reference channel for the 'itd'. The interaural time difference will be applied to the other channel with respect to the reference channel.

ild [float] Interaural level difference in dB SPL.

ildRef ['Right', 'Left' or None] The reference channel for the 'ild'. The level of the other channel will be icreased of attenuated by 'ild' dB SPL with respect to the reference channel.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> itdTone = binauralPureTone(frequency=440, phase=0, level=65,␣
↪duration=180,
...      ramp=10, channel='Both', itd=480, itdRef='Right', ild=0,␣
↪ildRef=None,
...      fs=48000, maxLevel=100)
>>> ildTone = binauralPureTone(frequency=440, phase=0, level=65,␣
↪duration=180,
...      ramp=10, channel='Both', itd=0, itdRef=None, ild=-20, ildRef=
↪'Right',
...      fs=48000, maxLevel=100)
```

sndlib.**broadbandNoise**(*spectrumLevel=25*, *duration=980*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Synthetise a broadband noise.

Parameters

spectrumLevel [float] Intensity spectrum level of the noise in dB SPL.

duration [float] Noise duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel [string ("Right", "Left", "Both", or "Dichotic")] Channel in which the noise will be generated. If 'Both' the same noise will be generated in both channels. If 'Dichotic' the noise will be independent at the two ears.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> noise = broadbandNoise(spectrumLevel=40, duration=180, ramp=10,
...     channel="Both", fs=48000, maxLevel=100)
```

sndlib.**camSinFMComplex**(*F0=150*, *lowHarm=1*, *highHarm=10*, *harmPhase=u'Sine'*, *fm=5*, *deltaCams=1*, *fmPhase=3.141592653589793*, *level=60*, *duration=180*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Generate a complex tone frequency modulated with an exponential sinusoid.

Parameters

F0 [float] Fundamental aarrier frequency in hertz.

lowHarm: int Lowest harmonic number.

highHarm: int Highest harmonic number.

harmPhase: string Harmonic phase relationship. One of 'Sine', 'Cosine', or 'Alternating'.

fm [float] Modulation frequency in Hz.

deltaCams [float] Frequency excursion in cam units (ERBn number scale).

fmPhase [float] Starting fmPhase in radians.

level [float] Tone level in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel ['Right', 'Left' or 'Both'] Channel in which the tone will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats]

Examples

```
>>> snd_peak = camSinFMComplex(F0=150, lowHarm=1, highHarm=10, harmPhase=
↪"Sine", fm=5, deltaCams=1, fmPhase=pi, level=60,
...     duration=180, ramp=10, channel="Both", fs=48000, maxLevel=100)
>>> snd_trough = camSinFMComplex(F0=150, lowHarm=1, highHarm=10,␣
↪harmPhase="Sine", fm=5, deltaCams=1, fmPhase=0, level=60,
...     duration=180, ramp=10, channel="Both", fs=48000, maxLevel=100)
```

sndlib.**camSinFMTone**(*fc=450*, *fm=5*, *deltaCams=1*, *fmPhase=3.141592653589793*, *startPhase=0*, *level=60*, *duration=180*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=100*)

Generate a tone frequency modulated with an exponential sinusoid.

Parameters

fc [float] Carrier frequency in hertz.

fm [float] Modulation frequency in Hz.

deltaCams [float] Frequency excursion in cam units (ERBn number scale).

fmPhase [float] Starting fmPhase in radians.

level [float] Tone level in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel ['Right', 'Left' or 'Both'] Channel in which the tone will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats]

Examples

```
>>> tone_peak = camSinFMTone(fc=450, fm=5, deltaCams=1, fmPhase=pi,␣
↪startPhase=0, level=60,
...     duration=180, ramp=10, channel="Both", fs=48000, maxLevel=100)
>>> tone_trough = camSinFMTone(fc=450, fm=5, deltaCams=1, fmPhase=0,␣
↪startPhase=0, level=60,
...     duration=180, ramp=10, channel="Both", fs=48000, maxLevel=100)
```

sndlib.**chirp**(*freqStart=440*, *ftype=u'linear'*, *rate=500*, *level=60*, *duration=980*, *phase=0*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Synthetize a chirp, that is a tone with frequency changing linearly or exponentially over time with a give rate.

Parameters

freqStart [float] Starting frequency in hertz.

ftype [string] If 'linear', the frequency will change linearly on a Hz scale. If 'exponential', the frequency will change exponentially on a cents scale.

rate [float] Rate of frequency change, Hz/s if ftype is 'linear', and cents/s if ftype is 'exponential'.

level [float] Level of the tone in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel [string ('Right', 'Left' or 'Both')] Channel in which the tone will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> gl = chirp(freqStart=440, ftype='linear', rate=500, level=55,
        duration=980, phase=0, ramp=10, channel='Both',
        fs=48000, maxLevel=100)
```

sndlib.**complexTone**(*F0=220*, *harmPhase=u'Sine'*, *lowHarm=1*, *highHarm=10*, *stretch=0*, *level=60*, *duration=980*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Synthetise a complex tone.

Parameters

F0 [float] Tone fundamental frequency in hertz.

harmPhase [one of 'Sine', 'Cosine', 'Alternating', 'Random', 'Schroeder-', 'Schroeder+'] Phase relationship between the partials of the complex tone.

lowHarm [int] Lowest harmonic component number.

highHarm [int] Highest harmonic component number.

stretch [float] Harmonic stretch in %F0. Increase each harmonic frequency by a fixed value that is equal to (F0*stretch)/100. If 'stretch' is different than zero, an inhanmonic complex tone will be generated.

level [float] The level of each partial in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel ['Right', 'Left', 'Both', 'Odd Right' or 'Odd Left'] Channel in which the tone will be generated. If 'channel' if 'Odd Right', odd numbered harmonics will be presented to the right channel and even number harmonics to the left channel. The opposite is true if 'channel' is 'Odd Left'.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> ct = complexTone(F0=440, harmPhase='Sine', lowHarm=3, highHarm=10,
...      stretch=0, level=55, duration=180, ramp=10, channel='Both',
...      fs=48000, maxLevel=100)
```

sndlib.**complexToneIPD**(*F0=220*, *harmPhase=u'Sine'*, *lowHarm=1*, *highHarm=10*, *stretch=0*, *level=60*, *duration=980*, *ramp=10*, *IPD=3.14*, *targetEar=u'Right'*, *fs=48000*, *maxLevel=101*)

Synthetise a complex tone with an interaural phase difference (IPD).

Parameters

F0 [float] Tone fundamental frequency in hertz.

harmPhase [one of 'Sine', 'Cosine', 'Alternating', 'Random', 'Schroeder-', 'Schroeder+'] Phase relationship between the partials of the complex tone.

lowHarm [int] Lowest harmonic component number.

highHarm [int] Highest harmonic component number.

stretch [float] Harmonic stretch in %F0. Increase each harmonic frequency by a fixed value that is equal to (F0*stretch)/100. If 'stretch' is different than zero, an inhanmonic complex tone will be generated.

level [float] The level of each partial in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

IPD [float] Interaural phase difference, in radians.

targetEar [string] The ear in which the phase will be shifted.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> ct = complexToneIPD(F0=440, harmPhase='Sine', lowHarm=3, highHarm=10,
...      stretch=0, level=55, duration=180, ramp=10, IPD=3.14, targetEar=
↪"Right",
...      fs=48000, maxLevel=100)
```

sndlib.**complexToneParallel**(*F0=220*, *harmPhase=u'Sine'*, *lowHarm=1*, *highHarm=10*, *stretch=0*, *level=0*, *duration=980*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Synthetise a complex tone.

This function produces the same results of complexTone. The only difference is that it uses the multiprocessing Python module to exploit multicore processors and compute the partials in a parallel fashion. Notice that there is a substantial overhead in setting up the parallel computations. This means that for relatively short sounds (in the order of seconds), this function will actually be *slower* than complexTone.

Parameters

F0 [float] Tone fundamental frequency in hertz.

harmPhase [one of 'Sine', 'Cosine', 'Alternating', 'Random', 'Schroeder-', 'Schroeder+'] Phase relationship between the partials of the complex tone.

lowHarm [int] Lowest harmonic component number.

highHarm [int] Highest harmonic component number.

stretch [float] Harmonic stretch in %F0. Increase each harmonic frequency by a fixed value that is equal to (F0*stretch)/100. If 'stretch' is different than zero, an inhanmonic complex tone will be generated.

level [float] The level of each partial in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel ['Right', 'Left', 'Both', 'Odd Right' or 'Odd Left'] Channel in which the tone will be generated. If 'channel' if 'Odd Right', odd numbered harmonics will be presented to the right channel and even number harmonics to the left channel. The opposite is true if 'channel' is 'Odd Left'.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> ct = complexToneParallel(F0=440, harmPhase='Sine', lowHarm=3,␣
↪highHarm=10,
...      stretch=0, level=55, duration=180, ramp=10, channel='Both',
...      fs=48000, maxLevel=100)
```

sndlib.**delayAdd**(*sig*, *delay*, *gain*, *iterations*, *configuration*, *fs*)

Delay and add algorithm for the generation of iterated rippled noise.

Parameters

sig [array of floats] The signal to manipulate

delay [float] delay in seconds

gain [float] The gain to apply to the delayed signal

iterations [int] The number of iterations of the delay-add cycle

configuration [string] If 'Add Same', the output of iteration N-1 is added to delayed signal of the current iteration. If 'Add Original', the original signal is added to delayed signal of the current iteration.

fs [int] Sampling frequency in Hz.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

References

[YPS1996]

Examples

```
>>> noise = broadbandNoise(spectrumLevel=40, duration=180, ramp=10,
...     channel='Both', fs=48000, maxLevel=100)
>>> irn = delayAdd(sig=noise, delay=1/440, gain=1, iterations=6,␣
↪configuration='Add Same', fs=48000)
```

sndlib.**dichoticNoiseFromSin**(*F0=300*, *lowHarm=1*, *highHarm=3*, *compLevel=30*, *narrowBandCompLevel=30*, *lowFreq=40*, *highFreq=2000*, *compSpacing=10*, *sigBandwidth=100*, *distanceUnit=u'Cent'*, *phaseRelationship=u'NoSpi'*, *dichoticDifference=u'IPD Stepped'*, *dichoticDifferenceValue=3.141592653589793*, *duration=380*, *ramp=10*, *fs=48000*, *maxLevel=101*)

Generate Huggins pitch or narrow-band noise from random-phase sinusoids.

This function generates first noise by adding closely spaced sinusoids in a wide frequency range. Then, it can apply an interaural time difference (ITD), an interaural phase difference (IPD) or a level increase to harmonically related narrow frequency bands within the noise. In the first two cases (ITD and IPD) the result is a dichotic pitch. In the last case the pitch can also be heard monaurally; adjusting the level increase, its salience can be closely matched to that of a dichotic pitch.

Parameters

F0 [float] Centre frequency of the fundamental in hertz.

lowHarm [int] Lowest harmonic component number.

highHarm [int] Highest harmonic component number.

compLevel [float] Level of each sinusoidal frequency component of the noise.

lowFreq [float] Lowest frequency in hertz of the noise.

highFreq [float] Highest frequency in hertz of the noise.

compSpacing [float] Spacing between the sinusoidal components used to generate the noise.

sigBandwidth [float] Width of each harmonically related frequency band.

distanceUnit [string (one of 'Hz', 'Cent', 'ERB')] The unit of measure used for 'compSpacing' and 'sigBandwidth'

phaseRelationship [string ('NoSpi' or 'NpiSo')] If NoSpi, the phase of the regions within each frequency band will be shifted. If NpiSo, the phase of the regions between each frequency band will be shifted.

dichoticDifference [string ('IPD Stepped', 'IPD Random', 'ITD', 'ILD Right', 'ILD Left')] Selects whether the decorrelation in the target regions will be achieved by applying a costant interaural phase shift (IPD), a random IPD shift, a costant interaural time difference (ITD), or a constant interaural level difference achieved by a level change in the right ('ILD Right') or the left ('ILD Left') ear.

dichoticDifferenceValue [float] For 'IPD Stepped', this is the phase offset, in radians, between the correlated and the uncorrelated regions. For 'ITD' this is the ITD in the transition region, in micro seconds. For 'Random Phase', the range

of phase shift randomization in the uncorrelated regions. For 'ILD Left' or 'ILD Right' this is the level difference between the left and right ear in the uncorrelated regions.

narrowBandCompLevel [float] Level of the sinusoidal components in the frequency bands. If the 'narrowBandCompLevel' is greater than the level of the background noise ('compLevel'), a complex tone consisting of narrowband noises in noise will be generated.

duration [float] Sound duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> s1 = dichoticNoiseFromSin(F0=300, lowHarm=1, highHarm=3,
    compLevel=30, narrowBandCompLevel=30,
    lowFreq=40, highFreq=2000, compSpacing=10,
    sigBandwidth=100, distanceUnit='Cent',
    phaseRelationship='NoSpi', dichoticDifference='IPD Stepped',
    dichoticDifferenceValue=pi, duration=380, ramp=10,
    fs=48000, maxLevel=101)
```

sndlib.**expAMNoise**(*fc=150*, *fm=2.5*, *deltaCents=1200*, *fmPhase=3.141592653589793*, *AMDepth=1*, *spectrumLevel=24*, *duration=480*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Generate a sinusoidally amplitude-modulated noise with an exponentially modulated AM frequency.

Parameters

fc [float] Carrier AM frequency in hertz.

fm [float] Modulation of the AM frequency in Hz.

deltaCents [float] AM frequency excursion in cents. The instataneous AM frequency of the noise will vary from fc**(-deltaCents/1200) to fc**(deltaCents/1200).

fmPhase [float] Starting phase of the AM modulation in radians.

AMDepth [float] Amplitude modulation depth.

spectrumLevel [float] Noise spectrum level in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel ['Right', 'Left' or 'Both'] Channel in which the tone will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats]

Examples

```
>>> snd = expAMNoise(fc=150, fm=2.5, deltaCents=1200, fmPhase=3.14,
    AMDepth = 1, spectrumLevel=24, duration=480, ramp=10, channel='Both',
↪
    fs=48000, maxLevel=100)
```

sndlib.**expSinFMComplex**(*F0=150*, *lowHarm=1*, *highHarm=10*, *harmPhase=u'Sine'*, *fm=40*, *deltaCents=1200*, *fmPhase=0*, *level=60*, *duration=180*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Generate a frequency-modulated complex tone with an exponential sinusoid.

Parameters

fc [float] Carrier frequency in hertz.

fm [float] Modulation frequency in Hz.

deltaCents [float]

Frequency excursion in cents. The instataneous frequency of the tone will vary from fc**(-deltaCents/1200) to fc**(+deltaCents/1200).

fmPhase [float] Starting fmPhase in radians.

level [float] Tone level in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel ['Right', 'Left' or 'Both'] Channel in which the tone will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats]

Examples

```
>>> tone_peak = expSinFMComplex(F0=150, lowHarm=1, highHarm=10,␣
↪harmPhase="Sine", fm=5, deltaCents=300, fmPhase=pi, level=60,
...      duration=180, ramp=10, channel="Both", fs=48000, maxLevel=101)
>>> tone_trough = expSinFMComplex(F0=150, lowHarm=1, highHarm=10,␣
↪harmPhase="Sine", fm=5, deltaCents=300, fmPhase=0, level=60,
...      duration=180, ramp=10, channel="Both", fs=48000, maxLevel=101)
```

sndlib.**expSinFMTone**(*fc=450*, *fm=5*, *deltaCents=300*, *fmPhase=3.141592653589793*, *startPhase=0*, *level=60*, *duration=180*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Generate a frequency-modulated tone with an exponential sinusoid.

Parameters

fc [float] Carrier frequency in hertz.

fm [float] Modulation frequency in Hz.

deltaCents [float]

Frequency excursion in cents. The instataneous frequency of the tone will vary from fc**(-deltaCents/1200) to fc**(+deltaCents/1200).

fmPhase [float] Starting fmPhase in radians.

level [float] Tone level in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel ['Right', 'Left' or 'Both'] Channel in which the tone will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats]

Examples

```
>>> tone_peak = expSinFMTone(fc=450, fm=5, deltaCents=300, fmPhase=pi,␣
↪level=60,
...      duration=180, ramp=10, channel="Both", fs=48000, maxLevel=101)
>>> tone_trough = expSinFMTone(fc=450, fm=5, deltaCents=300, fmPhase=0,␣
↪level=60,
...      duration=180, ramp=10, channel="Both", fs=48000, maxLevel=101)
```

sndlib.**fir2Filt**(*f1*, *f2*, *f3*, *f4*, *snd*, *fs*)

Filter signal with a fir2 filter.

This function designs and applies a fir2 filter to a sound. The frequency response of the ideal filter will transition from 0 to 1 between 'f1' and 'f2', and from 1 to zero between 'f3' and 'f4'. The frequencies must be given in increasing order.

Parameters

f1 [float] Frequency in hertz of the point at which the transition for the low-frequency cutoff ends.

f2 [float] Frequency in hertz of the point at which the transition for the low-frequency cutoff starts.

f3 [float] Frequency in hertz of the point at which the transition for the high-frequency cutoff starts.

f4 [float] Frequency in hertz of the point at which the transition for the high-frequency cutoff ends.

snd [array of floats] The sound to be filtered.

fs [int] Sampling frequency of 'snd'.

Returns

snd [2-dimensional array of floats]

Notes

If 'f1' and 'f2' are zero the filter will be lowpass. If 'f3' and 'f4' are equal to or greater than the nyquist frequency (fs/2) the filter will be highpass. In the other cases the filter will be bandpass.

The order of the filter (number of taps) is fixed at 256. This function uses internally 'scipy.signal.firwin2'.

Examples

```
>>> noise = broadbandNoise(spectrumLevel=40, duration=180, ramp=10,
...     channel='Both', fs=48000, maxLevel=100)
>>> lpNoise = fir2Filt(f1=0, f2=0, f3=1000, f4=1200,
...     snd=noise, fs=48000) #lowpass filter
>>> hpNoise = fir2Filt(f1=5000, f2=6000, f3=24000, f4=26000,
...     snd=noise, fs=48000) #highpass filter
>>> bpNoise = fir2Filt(f1=400, f2=600, f3=4000, f4=4400,
...     snd=noise, fs=48000) #bandpass filter
```

sndlib.**fm_complex1**(*midF0=140*, *harmPhase=u'Sine'*, *lowHarm=1*, *highHarm=10*, *level=60*, *duration=430*, *ramp=10*, *fmFreq=1.25*, *fmDepth=40*, *fmStartPhase=4.71238898038469*, *fmStartTime=25*, *fmDuration=400*, *levelAdj=True*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Synthetise a complex tone with an embedded FM starting and stopping at a chosen time after the tone

onset.

Parameters

midF0 [float] F0 at the FM zero crossing

harmPhase [one of 'Sine', 'Cosine', 'Alternating', 'Random', 'Schroeder-', or 'Schroeder+'] Phase relationship between the partials of the complex tone.

lowHarm [int] Lowest harmonic component number.

highHarm [int] Highest harmonic component number.

level [float] The level of each partial in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

fmFreq [float] FM frequency in Hz.

fmDepth [float] FM depth in %

fmStartPhase [float] Starting phase of FM

fmStartTime [float] Start of FM in ms after start of tone

fmDuration [float] Duration of FM, in ms

levelAdj [logical] If *True*, scale the harmonic level so that for a complex tone within a bandpass filter the overall level does not change with F0 modulations.

channel: 'Right', 'Left', 'Both', 'Odd Right' or 'Odd Left' Channel in which the tone will be generated. If 'channel' if 'Odd Right', odd numbered harmonics will be presented to the right channel and even number harmonics to the left channel. The opposite is true if 'channel' is 'Odd Left'.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Examples

```
>>> tone_up = fm_complex1(midF0=140, harmPhase="Sine", lowHarm=1,␣
↪highHarm=10, level=60, duration=430, ramp=10, fmFreq=1.25, fmDepth=40,␣
↪fmStartPhase=1.5*pi, fmStartTime=25, fmDuration=400, levelAdj=True,␣
↪channel="Both", fs=48000, maxLevel=101)
>>> tone_down = fm_complex1(midF0=140, harmPhase="Sine", lowHarm=1,␣
↪highHarm=10, level=60, duration=430, ramp=10, fmFreq=1.25, fmDepth=40,␣
↪fmStartPhase=0.5*pi, fmStartTime=25, fmDuration=400, levelAdj=True,␣
↪channel="Both", fs=48000, maxLevel=101)
```

sndlib.**fm_complex2**(*midF0=140*, *harmPhase=u'Sine'*, *lowHarm=1*, *highHarm=10*, *level=60*, *duration=430*, *ramp=10*, *fmFreq=1.25*, *fmDepth=40*, *fmStartPhase=4.71238898038469*, *fmStartTime=25*, *fmDuration=400*, *levelAdj=True*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Synthetise a complex tone with an embedded FM starting and stopping at a chosen time after the tone onset.

Parameters

midF0 [float] F0 at the FM zero crossing

harmPhase [one of 'Sine', 'Cosine', 'Alternating', 'Random', 'Schroeder-', 'Schroeder+'] Phase relationship between the partials of the complex tone.

lowHarm [int] Lowest harmonic component number.

highHarm [int] Highest harmonic component number.

level [float] The level of each partial in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

fmFreq [float] FM frequency in Hz.

fmDepth [float] FM depth in %

fmStartPhase [float] Starting phase of FM

fmStartTime [float] Start of FM in ms after start of tone

fmDuration [float] Duration of FM, in ms

levelAdj [logical] If *True*, scale the harmonic level so that for a complex tone within a bandpass filter the overall level does not change with F0 modulations.

channel: 'Right', 'Left', 'Both', 'Odd Right' or 'Odd Left' Channel in which the tone will be generated. If 'channel' if 'Odd Right', odd numbered harmonics will be presented to the right channel and even number harmonics to the left channel. The opposite is true if 'channel' is 'Odd Left'.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Examples

```
>>> tone_up = fm_complex2(midF0=140, harmPhase="Sine", lowHarm=1,␣
↪highHarm=10, level=60, duration=430, ramp=10, fmFreq=1.25, fmDepth=40,␣
↪fmStartPhase=1.5*pi, fmStartTime=25, fmDuration=400, levelAdj=True,␣
↪channel="Both", fs=48000, maxLevel=101)
>>> tone_down = fm_complex2(midF0=140, harmPhase="Sine", lowHarm=1,␣
↪highHarm=10, level=60, duration=430, ramp=10, fmFreq=1.25, fmDepth=40,␣
↪fmStartPhase=0.5*pi, fmStartTime=25, fmDuration=400, levelAdj=True,␣
↪channel="Both", fs=48000, maxLevel=101)
```

(continues on next page)

(continued from previous page)

sndlib.**freqFromERBInterval**(*f1*, *deltaERB*)

Compute the frequency, in Hz, corresponding to a distance, in equivalent rectangular bandwidths (ERBs), of 'deltaERB' from f1.

Parameters

f1 [float] frequency at one end of the interval in Hz

deltaERB [float] distance in ERBs

Returns

f2 [float] frequency at the other end of the interval in Hz

References

[GM]

Examples

```
>>> freqFromERBInterval(100, 1.5)
>>> freqFromERBInterval(100, -1.5)
>>> #for several frequencies
>>> freqFromERBInterval([100, 200, 300], 1.5)
>>> # for several distances
>>> freqFromERBInterval(100, [1, 1.5, 2])
```

sndlib.**gate**(*ramps*, *sig*, *fs*)

Impose onset and offset ramps to a sound.

Parameters

ramps [float] The duration of the ramps.

sig [array of floats] The signal on which the ramps should be imposed.

fs [int] The sampling frequency os 'sig'

Returns

sig [array of floats] The ramped signal.

Examples

```
>>> noise = broadbandNoise(spectrumLevel=40, duration=200, ramp=0,
...     channel='Both', fs=48000, maxLevel=100)
>>> gate(ramps=10, sig=noise, fs=48000)
```

sndlib.**getRMS**(*sig*, *channel=u'each'*)

Compute the root mean square (RMS) value of the signal.

Parameters

sig [array of floats] The signal for which the RMS needs to be computed.

channel [string or int] Either an integer indicating the channel number, 'each' for a list of the RMS values in each channel, or 'all' for the RMS across all channels.

Returns

rms [float] The RMS of 'sig'.

Examples

```
>>> pt = pureTone(frequency=440, phase=0, level=65, duration=180,
...     ramp=10, channel="Right", fs=48000, maxLevel=100)
>>> getRMS(pt, channel="each")
```

sndlib.**glide**(*freqStart=440*, *ftype=u'exponential'*, *excursion=500*, *level=60*, *duration=180*, *phase=0*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Synthetize a rising or falling tone glide with frequency changing linearly or exponentially.

Parameters

freqStart [float] Starting frequency in hertz.

ftype [string] If 'linear', the frequency will change linearly on a Hz scale. If 'exponential', the frequency will change exponentially on a cents scale.

excursion [float] If ftype is 'linear', excursion is the total frequency change in Hz. The final frequency will be freqStart + excursion. If ftype is 'exponential', excursion is the total frequency change in cents. The final frequency in Hz will be freqStart*2**(excursion/1200).

level [float] Level of the tone in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel [string ('Right', 'Left' or 'Both')] Channel in which the tone will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> gl = glide(freqStart=440, ftype='exponential', excursion=500,
        level=55, duration=180, phase=0, ramp=10, channel='Both',
        fs=48000, maxLevel=100)
```

sndlib.**harmComplFromNarrowbandNoise**(*F0=440*, *lowHarm=1*, *highHarm=8*, *level=40*, *bandwidth=80*, *bandwidthUnit=u'Hz'*, *stretch=0*, *duration=180*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Generate an harmonic complex tone from narrow noise bands.

Parameters

F0 [float] Fundamental frequency of the complex.

lowHarm [int] Lowest harmonic component number. The first component is #1.

highHarm [int] Highest harmonic component number.

level [float] The spectrum level of the noise bands in dB SPL.

bandwidth [float] The width of each noise band.

bandwidthUnit [string ('Hz', 'Cent', 'ERB')] Defines whether the bandwith of the noise bands is expressed in hertz (Hz), cents (Cent), or equivalent rectangular bandwidths (ERB).

stretch [float] Harmonic stretch in %F0. Increase each harmonic frequency by a fixed value that is equal to (F0*stretch)/100. If 'stretch' is different than zero, an inhanmonic complex tone will be generated.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel ['Right', 'Left', 'Both', 'Odd Right' or 'Odd Left'] Channel in which the tone will be generated. If 'channel' if 'Odd Right', odd numbered harmonics will be presented to the right channel and even number harmonics to the left channel. The opposite is true if 'channel' is 'Odd Left'.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [array of floats]

Examples

```
>>> c1 = harmComplFromNarrowbandNoise(F0=440, lowHarm=3, highHarm=8,
    level=40, bandwidth=80, bandwidthUnit="Hz", stretch=0, duration=180,
↪ ramp=10, channel='Both',
    fs=48000, maxLevel=100)
```

sndlib.**imposeLevelGlide**(*sig*, *deltaL*, *startTime*, *endTime*, *channel*, *fs*)

Impose a glide in level to a sound.

This function changes the level of a sound with a smooth transition (an amplitude ramp) between 'startTime' and 'endTime'. If the signal input to the function has a level L, the signal output by the function will have a level L between time 0 and 'startTime', and a level L+deltaL between endTime and the end of the sound.

Parameters

sig [float] Sound on which to impose the level change.

deltaL [float] Magnitude of the level change in dB SPL.

startTime [float] Start of the level transition in milliseconds.

endTime [float] End of the level transition in milliseconds.

channel [string ('Right', 'Left' or 'Both')] Channel to which apply the level transition.

fs [int] Samplig frequency of the sound in Hz.

Returns

snd [array of floats]

Examples

```
>>> pt = pureTone(frequency=440, phase=0, level=65, duration=180,
...     ramp=10, channel='Right', fs=48000, maxLevel=100)
>>> pt2 = imposeLevelGlide(sig=pt, deltaL=10, startTime=100,
        endTime=120, channel='Both', fs=48000)
```

sndlib.**intNCyclesFreq**(*freq*, *duration*)

Compute the frequency closest to 'freq' that has an integer number of cycles for the given sound duration.

Parameters

frequency [float] Frequency in hertz.

duration [float] Duration of the sound, in milliseconds.

Returns

adjFreq [float]

Examples

```
>>> intNCyclesFreq(freq=2.1, duration=1000)
2.0
>>> intNCyclesFreq(freq=2, duration=1000)
2.0
```

sndlib.**itdtoipd**(*itd*, *freq*)

Convert an interaural time difference to an equivalent interaural phase difference for a given frequency.

Parameters

itd [float] Interaural time difference in seconds.

freq [float] Frequency in hertz.

Returns

ipd [float]

Examples

```
>>> itd = 300 #microseconds
>>> itd = 300/1000000 #convert to seconds
>>> itdtoipd(itd=itd, freq=1000)
```

sndlib.**joinSndISI**(*sndList*, *ISIList*, *fs*)

Join a list of sounds with given interstimulus intervals

Parameters

sndList [list of arrays] The sounds to be joined.

ISIList [list of floats] The interstimulus intervals between the sounds in milliseconds. This list should have one element less than the sndList.

fs [int] Sampling frequency of the sounds in Hz.

Returns

snd [array of floats]

Examples

```
>>> pt1 = pureTone(frequency=440, phase=0, level=65, duration=180,
...       ramp=10, channel='Right', fs=48000, maxLevel=100)
>>> pt2 = pureTone(frequency=440, phase=0, level=65, duration=180,
...       ramp=10, channel='Right', fs=48000, maxLevel=100)
>>> tone_seq = joinSndISI([pt1, pt2], [500], 48000)
```

sndlib.**makeAsynchChord**(*freqs*, *levels*, *phases*, *tonesDuration*, *tonesRamps*, *tonesChannel*, *SOA*, *fs*, *maxLevel*)

Generate an asynchronous chord.

This function will add a set of pure tones with a given stimulus onset asynchrony (SOA). The temporal order of the successive tones is random.

Parameters

freqs [array or list of floats.] Frequencies of the chord components in hertz.

levels [array or list of floats.] Level of each chord component in dB SPL.

phases [array or list of floats.] Starting phase of each chord component.

tonesDuration [float] Duration of the tones composing the chord in milliseconds. All tones have the same duration.

tonesRamps [float] Duration of the onset and offset ramps in milliseconds. The total duration of the tones will be tonesDuration+ramp*2.

tonesChannel [string ('Right', 'Left' or 'Both')] Channel in which the tones will be generated.

SOA [float] Onset asynchrony between the chord components.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats]

Examples

```
>>> freqs = [250, 500, 1000]
>>> levels = [50, 50, 50]
>>> phases = [0, 0, 0]
>>> c1 = makeAsynchChord(freqs=freqs, levels=levels, phases=phases,
        tonesDuration=180, tonesRamps=10, tonesChannel='Both',
        SOA=60, fs=48000, maxLevel=100)
```

sndlib.**makeBlueRef**(*sig*, *fs*, *refHz*)

Convert a white noise into a blue noise.

The spectrum level of the blue noise at the frequency 'refHz' will be equal to the spectrum level of the white noise input to the function.

Parameters

sig [array of floats] The white noise to be turned into a blue noise.

fs [int] Sampling frequency of the sound.

refHz [int] Reference frequency in Hz. The amplitude of the other frequencies will be scaled with respect to the amplitude of this frequency.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> noise = broadbandNoise(spectrumLevel=40, duration=180, ramp=10,
...     channel='Both', fs=48000, maxLevel=100)
>>> noise = makeBlueRef(sig=noise, fs=48000, refHz=1000)
```

sndlib.**makeHugginsPitch**(*F0=300*, *lowHarm=1*, *highHarm=3*, *spectrumLevel=45*, *bandwidth=100*, *bandwidthUnit=u'Hz'*, *dichoticDifference=u'IPD Stepped'*, *dichoticDifferenceValue=3.141592653589793*, *phaseRelationship=u'NoSpi'*, *stretch=0*, *noiseType=u'White'*, *duration=480*, *ramp=10*, *fs=48000*, *maxLevel=101*)

Synthetise a complex Huggings Pitch.

Parameters

F0 [float] The centre frequency of the F0 of the complex in hertz.

lowHarm [int] Lowest harmonic component number.

highHarm [int] Highest harmonic component number.

spectrumLevel [float] The spectrum level of the noise from which the Huggins pitch is derived in dB SPL. If 'noiseType' is 'Pink', the spectrum level will be equal to 'spectrumLevel' at 1 kHz.

bandwidth [float] Bandwidth of the frequency regions in which the phase transitions occurr.

bandwidthUnit [string ('Hz', 'Cent', 'ERB')] Defines whether the bandwith of the decorrelated bands is expressed in hertz (Hz), cents (Cent), or equivalent rectangular bandwidths (ERB).

dichoticDifference [string ('IPD Linear', 'IPD Stepped', 'IPD Random', 'ITD')] Selects whether the decorrelation in the target regions will be achieved by applying a costant interaural phase shift (IPD), an costant interaural time difference (ITD), or a random IPD shift.

dichoticDifferenceValue [float] For 'IPD Linear' this is the phase difference between the start and the end of each transition region, in radians. For 'IPD Stepped', this is the phase offset, in radians, between the correlated and the uncorrelated regions. For 'ITD' this is the ITD in the transition region, in micro seconds. For 'Random Phase', the range of phase shift randomization in the uncorrelated regions.

phaseRelationship [string ('NoSpi' or 'NpiSo')] If NoSpi, the phase of the regions within each frequency band will be shifted. If NpiSo, the phase of the regions between each frequency band will be shifted.

stretch [float] Harmonic stretch in %F0. Increase each harmonic frequency by a fixed value that is equal to (F0*stretch)/100. If 'stretch' is different than zero, an inhanmonic complex tone will be generated.

noiseType [string ('White' or 'Pink')] The type of noise used to derive the Huggins Pitch.

duration [float] Complex duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

References

[CH], [AS], [ZH]

Examples

```
>>> hp = makeHugginsPitch(F0=300, lowHarm=1, highHarm=3,␣
↪spectrumLevel=45,
    bandwidth=100, bandwidthUnit='Hz', dichoticDifference='IPD Stepped',
    dichoticDifferenceValue=pi, phaseRelationship='NoSpi', stretch=0,
    noiseType='White', duration=380, ramp=10, fs=48000, maxLevel=101)
```

sndlib.**makeIRN**(*delay=0.0022727272727272726*, *gain=1*, *iterations=6*, *configuration=u'Add Same'*, *spectrumLevel=25*, *duration=280*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Synthetise a iterated rippled noise

Parameters

delay [float] delay in seconds

gain [float] The gain to apply to the delayed signal

iterations [int] The number of iterations of the delay-add cycle

configuration [string] If 'Add Same', the output of iteration N-1 is added to delayed signal of the current iteration. If 'Add Original', the original signal is added to delayed signal of the current iteration.

spectrumLevel [float] Intensity spectrum level of the noise in dB SPL.

duration [float] Noise duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel [string ('Right', 'Left', 'Both', or 'Dichotic')] Channel in which the noise will be generated.

fs [int] Sampling frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> irn = makeIRN(delay=1/440, gain=1, iterations=6, configuration='Add␣
↪Same',
       spectrumLevel=25, duration=280, ramp=10, channel='Both', fs=48000,
       maxLevel=101)
```

sndlib.**makePink**(*sig*, *fs*)

Convert a white noise into a pink noise.

The spectrum level of the pink noise at 1000 Hz will be equal to the spectrum level of the white noise input to the function.

Parameters

sig [array of floats] The white noise to be turned into a pink noise.

fs [int] Sampling frequency of the sound.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> noise = broadbandNoise(spectrumLevel=40, duration=180, ramp=10,
...     channel='Both', fs=48000, maxLevel=100)
>>> noise = makePink(sig=noise, fs=48000)
```

sndlib.**makePinkRef**(*sig*, *fs*, *refHz*)

Convert a white noise into a pink noise.

The spectrum level of the pink noise at the frequency 'refHz' will be equal to the spectrum level of the white noise input to the function.

Parameters

sig [array of floats] The white noise to be turned into a pink noise.

fs [int] Sampling frequency of the sound.

refHz [int] Reference frequency in Hz. The amplitude of the other frequencies will be scaled with respect to the amplitude of this frequency.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> noise = broadbandNoise(spectrumLevel=40, duration=180, ramp=10,
...     channel='Both', fs=48000, maxLevel=100)
>>> noise = makePinkRef(sig=noise, fs=48000, refHz=1000)
```

sndlib.**makeRedRef**(*sig*, *fs*, *refHz*)

Convert a white noise into a red noise.

The spectrum level of the red noise at the frequency 'refHz' will be equal to the spectrum level of the white noise input to the function.

Parameters

sig [array of floats] The white noise to be turned into a red noise.

fs [int] Sampling frequency of the sound.

refHz [int] Reference frequency in Hz. The amplitude of the other frequencies will be scaled with respect to the amplitude of this frequency.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> noise = broadbandNoise(spectrumLevel=40, duration=180, ramp=10,
...     channel='Both', fs=48000, maxLevel=100)
>>> noise = makeRedRef(sig=noise, fs=48000, refHz=1000)
```

sndlib.**makeSilence**(*duration=1000*, *fs=48000*)

Generate a silence.

This function just fills an array with zeros for the desired duration.

Parameters

duration [float] Duration of the silence in milliseconds.

fs [int] Samplig frequency in Hz.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> sil = makeSilence(duration=200, fs=48000)
```

sndlib.**makeVioletRef**(*sig*, *fs*, *refHz*)

Convert a white noise into a violet noise.

The spectrum level of the violet noise at the frequency 'refHz' will be equal to the spectrum level of the white noise input to the function.

Parameters

sig [array of floats] The white noise to be turned into a violet noise.

fs [int] Sampling frequency of the sound.

refHz [int] Reference frequency in Hz. The amplitude of the other frequencies will be scaled with respect to the amplitude of this frequency.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> noise = broadbandNoise(spectrumLevel=40, duration=180, ramp=10,
...     channel='Both', fs=48000, maxLevel=100)
>>> noise = makeVioletRef(sig=noise, fs=48000, refHz=1000)
```

sndlib.**nextpow2**(*x*)

Next power of two.

Parameters

x [int] Base number.

Returns

out [float] The power to which 2 should be raised.

Examples

```
>>> nextpow2(511)
9
>>> 2**9
512
```

sndlib.**phaseShift**(*sig*, *f1*, *f2*, *phaseShift*, *phaseShiftType*, *channel*, *fs*)

Shift the interaural phases of a sound within a given frequency region.

Parameters

sig [array of floats] Input signal.

f1 [float] The start point of the frequency region to be phase-shifted in hertz.

f2 [float] The end point of the frequency region to be phase-shifted in hertz.

phaseShift [float] The amount of phase shift in radians.

phaseShiftType [string ('Linear', 'Step', 'Random')] If 'Linear' the phase changes progressively on a linear Hz scale from X to X+'phaseShift' from f1 to f2. If 'Stepped' 'phaseShift' is added as a constant to the phases from f1 to f2. If 'Random' a random phase shift from 0 to 'phaseShift' is added to each frequency component from f1 to f2.

channel [string (one of 'Right', 'Left' or 'Both')] The channel in which to apply the phase shift.

fs [float] The sampling frequency of the sound.

Returns

out [2-dimensional array of floats]

Examples

```
>>> noise = broadbandNoise(spectrumLevel=40, duration=180, ramp=10,
...     channel='Both', fs=48000, maxLevel=100)
>>> hp = phaseShift(sig=noise, f1=500, f2=600, phaseShift=3.14,
        phaseShiftType='Linear', channel='Left', fs=48000) #this␣
↪generates a Dichotic Pitch
```

sndlib.**pinkNoiseFromSin**(*compLevel=23*, *lowCmp=100*, *highCmp=1000*, *spacing=20*, *duration=180*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Generate a pink noise by adding sinusoids spaced by a fixed interval in cents.

Parameters

compLevel [float] Level of each sinusoidal component in dB SPL.

lowCmp [float] Frequency of the lowest noise component in hertz.

highCmp [float] Frequency of the highest noise component in hertz.

spacing [float] Spacing between the frequencies of the sinusoidal components in hertz.

duration [float] Noise duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel [string ('Right', 'Left' or 'Both')] Channel in which the noise will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> noise = pinkNoiseFromSin(compLevel=23, lowCmp=100, highCmp=1000,
    spacing=20, duration=180, ramp=10, channel='Both',
    fs=48000, maxLevel=100)
```

sndlib.**pinkNoiseFromSin2**(*compLevel=23*, *lowCmp=100*, *highCmp=1000*, *spacing=20*, *duration=180*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Generate a pink noise by adding sinusoids spaced by a fixed interval in cents.

This function should produce the same output of pinkNoiseFromSin, it simply uses a different algorithm that uses matrix operations instead of a for loop. It doesn't seem to be much faster though.

Parameters

compLevel [float] Level of each sinusoidal component in dB SPL.

lowCmp [float] Frequency of the lowest noise component in hertz.

highCmp [float] Frequency of the highest noise component in hertz.

spacing [float] Spacing between the frequencies of the sinusoidal components in hertz.

duration [float] Noise duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel [string ('Right', 'Left' or 'Both')] Channel in which the noise will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> noise = pinkNoiseFromSin2(compLevel=23, lowCmp=100, highCmp=1000,
    spacing=20, duration=180, ramp=10, channel='Both',
    fs=48000, maxLevel=100)
```

sndlib.**pureTone**(*frequency=1000*, *phase=0*, *level=60*, *duration=980*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Synthetise a pure tone.

Parameters

frequency [float] Tone frequency in hertz.

phase [float] Starting phase in radians.

level [float] Tone level in dB SPL.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel [string ('Right', 'Left' or 'Both')] Channel in which the tone will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> pt = pureTone(frequency=440, phase=0, level=65, duration=180,
...     ramp=10, channel='Right', fs=48000, maxLevel=100)
>>> pt.shape
(9600, 2)
```

sndlib.**scale**(*level*, *sig*)

Increase or decrease the amplitude of a sound signal.

Parameters

level [float] Desired increment or decrement in dB SPL.

signal [array of floats] Signal to scale.

Returns

sig [2-dimensional array of floats]

Examples

```
>>> noise = broadbandNoise(spectrumLevel=40, duration=180, ramp=10,
...     channel='Both', fs=48000, maxLevel=100)
>>> noise = scale(level=-10, sig=noise) #reduce level by 10 dB
```

sndlib.**setLevel_**(*level*, *snd*, *maxLevel*, *channel=u'Both'*)

Set the RMS level of a sound signal to a given value.

Parameters

level [float] The desired RMS level of the signal in dB SPL.

snd [array of floats] Signal whose level is to be set.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

channel [string ('Right', 'Left' or 'Both')] Channel in which the level will be set.

Returns

sig [2-dimensional array of floats]

Examples

```
>>> import copy
>>> pt = pureTone(frequency=1000, phase=0, level=60, duration=100,
...     ramp=0, channel="Both", fs=48000, maxLevel=100)
>>> pt2 = copy.copy(pt) #this function modifies the argument so make a␣
↪copy!
>>> pt2 = setLevel_(level=40, snd=pt2, maxLevel=100, channel="Both")
↪#set spectrum level to 20 dB SPL
>>> levDiff = 20*log10(getRMS(pt)[1]/getRMS(pt2)[1])
```

sndlib.**spectralModNoise**(*spectrumLevel=25*, *duration=980*, *ramp=10*, *modAmp=10*, *modFreq=1*, *phase=u'Random'*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Generate a broadband noise with a modulated spectral envelope. The modulation is sinuisoidal on a log2 frequency, and logarithmic amplitude (dB) scale.

Parameters

spectrumLevel [float] Intensity spectrum level of the noise in dB SPL (before modulation). Note that the shaped noise is scaled after shaping so as to have the same overall RMS level than the noise before shaping.

duration [float] Noise duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

modAmp: float Modulation amplitude (peak-to-trough difference) in dB.

modFreq: float Modulation frequency in cycles-per-octave

modPhase: string Starting modulation phase

channel [string ("Right", "Left", "Both", or "Dichotic")] Channel in which the noise will be generated. If 'Both' the same noise will be generated in both channels. If 'Dichotic' the noise will be independent at the two ears.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

sig [2-dimensional array of floats]

References

Andéol, G., Macpherson, E. A., & Sabin, A. T. (2013). Sound localization in noise and sensitivity to spectral shape. Hearing Research, 304, 20–27. https://doi.org/10.1016/j.heares.2013.06.001

Eddins, D. A., & Bero, E. M. (2007). Spectral modulation detection as a function of modulation frequency, carrier bandwidth, and carrier frequency region. The Journal of the Acoustical Society of America, 121(1), 363–372. https://doi.org/10.1121/1.2382347

Litvak, L. M., Spahr, A. J., Saoji, A. A., & Fridman, G. Y. (2007). Relationship between perception of spectral ripple and speech recognition in cochlear implant and vocoder listeners. The Journal of the Acoustical Society of America, 122(2), 982–991. https://doi.org/10.1121/1.2749413

Examples

```
>>> noise = spectralModNoise(spectrumLevel=40, duration=180, ramp=10,
...     modAmp=10, modFreq=2, phase="Random",
...     channel='Both', fs=48000, maxLevel=100)
```

sndlib.**steepNoise**(*frequency1=440*, *frequency2=660*, *level=60*, *duration=180*, *ramp=10*, *channel=u'Both'*, *fs=48000*, *maxLevel=101*)

Synthetise band-limited noise from the addition of random-phase sinusoids.

Parameters

frequency1 [float] Start frequency of the noise.

frequency2 [float] End frequency of the noise.

level [float] Noise spectrum level.

duration [float] Tone duration (excluding ramps) in milliseconds.

ramp [float] Duration of the onset and offset ramps in milliseconds. The total duration of the sound will be duration+ramp*2.

channel [string ('Right', 'Left' or 'Both')] Channel in which the tone will be generated.

fs [int] Samplig frequency in Hz.

maxLevel [float] Level in dB SPL output by the soundcard for a sinusoid of amplitude 1.

Returns

snd [2-dimensional array of floats] The array has dimensions (nSamples, 2).

Examples

```
>>> nbNoise = steepNoise(frequency1=440, frequency2=660, level=65,
...     duration=180, ramp=10, channel='Right', fs=48000, maxLevel=100)
```

pysdt	A module for computing signal detection theory measures.

CHAPTER 13

pysdt

A module for computing signal detection theory measures. Some of the functions in this module have been ported to python from the 'psyphy' R package of Kenneth Knoblauch http://cran.r-project.org/web/packages/psyphy/index.html

Functions

compute_proportions(nCA, nTA, nIB, nTB, corr)	Compute proportions with optional corrections for extreme proportions.
dprime_ABX(H, FA, meth)	Compute *d'* for ABX task from 'hit' and 'false alarm' rates.
dprime_ABX_from_counts(nCA, nTA, nCB, nTB, ...)	Compute *d'* for ABX task from counts of correct and total responses.
dprime_SD(H, FA, meth)	Compute *d'* for one interval same/different task from 'hit' and 'false alarm' rates.
dprime_SD_from_counts(nCA, nTA, nCB, nTB, ...)	Compute *d'* for one interval same/different task from counts of correct and total responses.
dprime_mAFC(Pc, m)	Compute *d'* corresponding to a certain proportion of correct
dprime_oddity(prCorr[, meth])	Compute *d'* for oddity task from proportion of correct responses.
dprime_yes_no(H, FA)	Compute *d'* for one interval 'yes/no' type tasks from hits and false alarm rates.
dprime_yes_no_from_counts(nCA, nTA, nCB, ...)	Compute *d'* for one interval 'yes/no' type tasks from counts of correct and total responses.

Continued on next page

Table 1 – continued from previous page

gaussianPsy(x, alphax, betax, gammax, lambdax)	Compute the gaussian psychometric function.
gumbelPsy(x, alphax, betax, gammax, lambdax)	Compute the gumbel psychometric function.
invGaussianPsy(p, alphax, betax, gammax, lambdax)	Compute the inverse gaussian psychometric function.
invGumbelPsy(p, alphax, betax, gammax, lambdax)	Compute the inverse gumbel psychometric function.
invLogisticPsy(p, alphax, betax, gammax, lambdax)	Compute the inverse logistic psychometric function.
invWeibullPsy(p, alphax, betax, gammax, lambdax)	Compute the inverse weibull psychometric function.
logisticLikelihood(lev, response, alphax, ...)	
logisticPsy(x, alphax, betax, gammax, lambdax)	Compute the logistic psychometric function.
weibullPsy(x, alphax, betax, gammax, lambdax)	Compute the weibull psychometric function.

CHAPTER 14

pysdt – Signal Detection Theory Measures

A module for computing signal detection theory measures. Some of the functions in this module have been ported to python from the 'psyphy' R package of Kenneth Knoblauch http://cran.r-project.org/web/packages/psyphy/index.html

pysdt.**compute_proportions**(*nCA*, *nTA*, *nIB*, *nTB*, *corr*)

Compute proportions with optional corrections for extreme proportions.

Parameters

nCA [float]

Number of correct 'A' trials

nTA [int] Number of total 'A' trials

nIB [float] Number of incorrect 'B' trials

nTB [int] Number of total 'B' trials

corr [string] The correction to apply, *none* for no correction, 'loglinear' for the log-linear correction, and *2N* for the '2N' correction.

Returns

HR [float]

Hit rate

FA [float] False alarm rate

References

[1], [2]

Examples

```
>>> H,F = compute_proportions(8, 10, 2, 10, "loglinear")
>>> H,F = compute_proportions(10, 10, 2, 10, "loglinear")
>>> H,F = compute_proportions(10, 10, 2, 10, "2N")
```

pysdt.**dprime_ABX**(*H*, *FA*, *meth*)

Compute *d'* for ABX task from 'hit' and 'false alarm' rates.

Parameters

H [float] Hit rate.

FA [float] False alarms rate.

meth [string] 'diff' for differencing strategy or 'IO' for independent observations strategy.

Returns

dprime [float] *d'* value

References

[1]

Examples

```
>>> dp = dprime_ABX(0.7, 0.2, 'IO')
>>> dp = dprime_ABX(0.7, 0.2, 'diff')
```

pysdt.**dprime_ABX_from_counts**(*nCA*, *nTA*, *nCB*, *nTB*, *meth*, *corr*)

Compute *d'* for ABX task from counts of correct and total responses.

Parameters

nCA [int] Number of correct responses in 'same' trials.

nTA [int] Total number of 'same' trials.

nCB [int] Number of correct responses in 'different' trials.

nTB [int] Total number of 'different' trials.

meth [string] 'diff' for differencing strategy or 'IO' for independent observations strategy.

corr [logical] if True, apply the correction to avoid hit and false alarm rates of 0 or one.

Returns

dprime [float] *d'* value

References

[1]

Examples

```
>>> dp = dprime_ABX(0.7, 0.2, 'IO')
```

pysdt.**dprime_SD**(*H*, *FA*, *meth*)

Compute *d'* for one interval same/different task from 'hit' and 'false alarm' rates.

Parameters

H [float] Hit rate.

FA [float] False alarms rate.

meth [string] 'diff' for differencing strategy or 'IO' for independent observations strategy.

Returns

dprime [float] *d'* value

References

[1], [2]

Examples

```
>>> dp = dprime_SD(0.7, 0.2, 'IO')
```

pysdt.**dprime_SD_from_counts**(*nCA*, *nTA*, *nCB*, *nTB*, *meth*, *corr*)

Compute *d'* for one interval same/different task from counts of correct and total responses.

Parameters

nCA [int] Number of correct responses in 'same' trials.

nTA [int] Total number of 'same' trials.

nCB [int] Number of correct responses in 'different' trials.

nTB [int] Total number of 'different' trials.

meth [string] 'diff' for differencing strategy or 'IO' for independent observations strategy.

corr [logical] if True, apply the correction to avoid hit and false alarm rates of 0 or one.

Returns

dprime [float] *d'* value

References

[1], [2]

Examples

```
>>> dp = dprime_SD(0.7, 0.2, 'IO')
```

pysdt.**dprime_mAFC**(*Pc*, *m*)

Compute *d'* corresponding to a certain proportion of correct responses in m-AFC tasks.

Parameters

Pc [float]

Proportion of correct responses.

m [int] Number of alternatives.

Returns

dprime [float] *d'* value

References

[1], [2]

Examples

```
>>> dp = dprime_mAFC(0.7, 3)
```

pysdt.**dprime_oddity**(*prCorr*, *meth=u'diff'*)

Compute *d'* for oddity task from proportion of correct responses. Only valid for the case in which there are three presentation intervals.

Parameters

prCorr [float] Proportion of correct responses.

meth [string] 'diff' for differencing strategy or 'IO' for independent observations strategy.

Returns

dprime [float] *d'* value

References

[1], [2]

Examples

```
>>> dp = dprime_oddity(0.7)
>>> dp = dprime_oddity(0.8)
```

pysdt.**dprime_yes_no**(*H*, *FA*)

Compute *d'* for one interval 'yes/no' type tasks from hits and false alarm rates.

Parameters

H [float] Hit rate.

FA [float] False alarms rate.

Returns

dprime [float] *d'* value

References

[1], [2]

Examples

```
>>> dp = dprime_yes_no(0.7, 0.2)
```

pysdt.**dprime_yes_no_from_counts**(*nCA*, *nTA*, *nCB*, *nTB*, *corr*)

Compute *d'* for one interval 'yes/no' type tasks from counts of correct and total responses.

Parameters

nCA [int] Number of correct responses in 'signal' trials.

nTA [int] Total number of 'signal' trials.

nCB [int] Number of correct responses in 'noise' trials.

nTB [int] Total number of 'noise' trials.

corr [logical] if True, apply the correction to avoid hit and false alarm rates of 0 or one.

Returns

dprime [float] *d'* value

References

[1], [2]

Examples

```
>>> dp = dprime_yes_no_from_counts(nCA=70, nTA=100, nCB=80, nTB=100,␣
↪corr=True)
```

pysdt.**gaussianPsy**(*x*, *alphax*, *betax*, *gammax*, *lambdax*)

Compute the gaussian psychometric function.

Parameters

x : Stimulus level(s).

alphax: Mid-point(s) of the psychometric function.

betax: The slope of the psychometric function.

gammax: Lower limit of the psychometric function (guess rate).

lambdax: The lapse rate.

Returns

pc : Proportion correct at the stimulus level(s) *x*.

References

[1]

pysdt.**gumbelPsy**(*x*, *alphax*, *betax*, *gammax*, *lambdax*)

Compute the gumbel psychometric function.

Parameters

x : Stimulus level(s).

alphax: Mid-point(s) of the psychometric function.

betax: The slope of the psychometric function.

gammax: Lower limit of the psychometric function (guess rate).

lambdax: The lapse rate.

Returns

pc : Proportion correct at the stimulus level(s) *x*.

References

[1]

pysdt.**invGaussianPsy**(*p*, *alphax*, *betax*, *gammax*, *lambdax*)
Compute the inverse gaussian psychometric function.

Parameters

p : Proportion correct on the psychometric function.

alphax: Mid-point(s) of the psychometric function.

betax: The slope of the psychometric function.

gammax: Lower limit of the psychometric function.

lambdax: The lapse rate.

Returns

x : Stimulus level at which proportion correct equals *p* for the listener specified by the function.

References

[1]

pysdt.**invGumbelPsy**(*p*, *alphax*, *betax*, *gammax*, *lambdax*)
Compute the inverse gumbel psychometric function.

Parameters

p : Proportion correct on the psychometric function.

alphax: Mid-point(s) of the psychometric function.

betax: The slope of the psychometric function.

gammax: Lower limit of the psychometric function.

lambdax: The lapse rate.

Returns

x : Stimulus level at which proportion correct equals *p* for the listener specified by the function.

References

[1]

pysdt.**invLogisticPsy**(*p*, *alphax*, *betax*, *gammax*, *lambdax*)

Compute the inverse logistic psychometric function.

Parameters

p : Proportion correct on the psychometric function.

alphax: Mid-point(s) of the psychometric function.

betax: The slope of the psychometric function.

gammax: Lower limit of the psychometric function.

lambdax: The lapse rate.

Returns

x : Stimulus level at which proportion correct equals *p* for the listener specified by the function.

References

[1]

pysdt.**invWeibullPsy**(*p*, *alphax*, *betax*, *gammax*, *lambdax*)

Compute the inverse weibull psychometric function.

Parameters

p : Proportion correct on the psychometric function.

alphax: Mid-point(s) of the psychometric function.

betax: The slope of the psychometric function.

gammax: Lower limit of the psychometric function.

lambdax: The lapse rate.

Returns

x : Stimulus level at which proportion correct equals *p* for the listener specified by the function.

References

[1]

pysdt.**logisticPsy**(*x*, *alphax*, *betax*, *gammax*, *lambdax*)

Compute the logistic psychometric function.

Parameters

x : Stimulus level(s).

alphax: Mid-point(s) of the psychometric function.

betax: The slope of the psychometric function.

gammax: Lower limit of the psychometric function (guess rate).

lambdax: The lapse rate.

Returns

pc : Proportion correct at the stimulus level(s) *x*.

References

[1]

pysdt.**weibullPsy**(*x*, *alphax*, *betax*, *gammax*, *lambdax*)

Compute the weibull psychometric function.

Parameters

x : Stimulus level(s).

alphax: Mid-point(s) of the psychometric function.

betax: The slope of the psychometric function.

gammax: Lower limit of the psychometric function (guess rate).

lambdax: The lapse rate.

Returns

pc : Proportion correct at the stimulus level(s) *x*.

References

[1]

CHAPTER 15

References

CHAPTER 16

GNU Free Documentation License

Version 1.2, November 2002

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The

"Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with…Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

CHAPTER 17

Indices and tables

- genindex
- modindex
- search

Bibliography

[H] Hartmann, W. M. (2004). Signals, Sound, and Sensation. Springer Science & Business Media

[V79] Viemeister, N. F. (1979). Temporal modulation transfer functions based upon modulation thresholds. The Journal of the Acoustical Society of America, 66(5), 1364–1380. https://doi.org/10.1121/1.383531

[YSO] Yost, W., Sheft, S., & Opie, J. (1989). Modulation interference in detection and discrimination of amplitude modulation. The Journal of the Acoustical Society of America, 86(December 1989), 2138–2147. https://doi.org/10.1121/1.398474

[H] Hartmann, W. M. (2004). Signals, Sound, and Sensation. Springer Science & Business Media

[V79] Viemeister, N. F. (1979). Temporal modulation transfer functions based upon modulation thresholds. The Journal of the Acoustical Society of America, 66(5), 1364–1380. https://doi.org/10.1121/1.383531

[YSO] Yost, W., Sheft, S., & Opie, J. (1989). Modulation interference in detection and discrimination of amplitude modulation. The Journal of the Acoustical Society of America, 86(December 1989), 2138–2147. https://doi.org/10.1121/1.398474

[H] Hartmann, W. M. (2004). Signals, Sound, and Sensation. Springer Science & Business Media

[YSO] Yost, W., Sheft, S., & Opie, J. (1989). Modulation interference in detection and discrimination of amplitude modulation. The Journal of the Acoustical Society of America, 86(December 1989), 2138–2147. https://doi.org/10.1121/1.398474

[GM] Glasberg, B. R., & Moore, B. C. J. (1990). Derivation of auditory filter shapes from notched-noise data. Hear. Res., 47(1-2), 103–38.

[YPS1996] Yost, W. A., Patterson, R., & Sheft, S. (1996). A time domain description for the pitch strength of iterated rippled noise. J. Acoust. Soc. Am., 99(2), 1066–78.

[GM] Glasberg, B. R., & Moore, B. C. J. (1990). Derivation of auditory filter shapes from notched-noise data. Hear. Res., 47(1-2), 103–38.

[CH] Cramer, E. M., & Huggins, W. H. (1958). Creation of Pitch through Binaural Interaction. J. Acoust. Soc. Am., 30(5), 413.

[AS] Akeroyd, M. A., & Summerfield, a Q. (2000). The lateralization of simple dichotic pitches. J. Acoust. Soc. Am., 108(1), 316–334.

[ZH] Zhang, P. X., & Hartmann, W. M. (2008). Lateralization of Huggins pitch. J. Acoust. Soc. Am., 124(6), 3873–87.

[1] Hautus, M. J. (1995). Corrections for extreme proportions and their biasing effects on estimated values of *d'*. *Behavior Research Methods, Instruments, & Computers, 27(I)*, 46–51. http://doi.org/10.3758/BF03203619

[2] Macmillan, N. A., & Creelman, C. D. (2004). *Detection Theory: A User's Guide (2nd ed.)*. London: Lawrence Erlbraum Associates.

[1] Macmillan, N. A., & Creelman, C. D. (2004). *Detection Theory: A User's Guide (2nd ed.)*. London: Lawrence Erlbraum Associates.

[1] Macmillan, N. A., & Creelman, C. D. (2004). *Detection Theory: A User's Guide (2nd ed.)*. London: Lawrence Erlbraum Associates.

[1] Macmillan, N. A., & Creelman, C. D. (2004). *Detection Theory: A User's Guide (2nd ed.)*. London: Lawrence Erlbraum Associates.

[2] Kingdom, F. A. A., & Prins, N. (2010). *Psychophysics: A Practical Introduction*. Academic Press.

[1] Macmillan, N. A., & Creelman, C. D. (2004). *Detection Theory: A User's Guide (2nd ed.)*. London: Lawrence Erlbraum Associates.

[2] Kingdom, F. A. A., & Prins, N. (2010). *Psychophysics: A Practical Introduction*. Academic Press.

[1] Green, D. M., & Swets, J. A. (1988). *Signal Detection Theory and Psychophysics*. Los Altos, California: Peninsula Publishing.

[2] Green, D. M., & Dai, H. P. (1991). Probability of being correct with 1 of M orthogonal signals. *Perception & Psychophysics, 49(1)*, 100–101.

[1] Macmillan, N. A., & Creelman, C. D. (2004). *Detection Theory: A User's Guide (2nd ed.)*. London: Lawrence Erlbraum Associates.

[2] Versfeld, N. J., Dai, H., & Green, D. M. (1996). The optimum decision rules for the oddity task. *Perception & Psychophysics, 58(1)*, 10–21.

[1] Green, D. M., & Swets, J. A. (1988). *Signal Detection Theory and Psychophysics*. Los Altos, California: Peninsula Publishing.

[2] Macmillan, N. A., & Creelman, C. D. (2004). *Detection Theory: A User's Guide (2nd ed.)*. London: Lawrence Erlbraum Associates.

[1] Green, D. M., & Swets, J. A. (1988). *Signal Detection Theory and Psychophysics*. Los Altos, California: Peninsula Publishing.

[2] Macmillan, N. A., & Creelman, C. D. (2004). *Detection Theory: A User's Guide (2nd ed.)*. London: Lawrence Erlbraum Associates.

[1] Kingdom, F. A. A., & Prins, N. (2010). *Psychophysics: A Practical Introduction*. Academic Press.

[1] Kingdom, F. A. A., & Prins, N. (2010). *Psychophysics: A Practical Introduction*. Academic Press.

[1] Kingdom, F. A. A., & Prins, N. (2010). *Psychophysics: A Practical Introduction*. Academic Press.

[1] Kingdom, F. A. A., & Prins, N. (2010). *Psychophysics: A Practical Introduction*. Academic Press.

[1] Kingdom, F. A. A., & Prins, N. (2010). *Psychophysics: A Practical Introduction*. Academic Press.

[1] Kingdom, F. A. A., & Prins, N. (2010). *Psychophysics: A Practical Introduction*. Academic Press.

[1] Kingdom, F. A. A., & Prins, N. (2010). *Psychophysics: A Practical Introduction*. Academic Press.

[1] Kingdom, F. A. A., & Prins, N. (2010). *Psychophysics: A Practical Introduction*. Academic Press.

[GrimaultEtAl2002] Grimault, N., Micheyl, C., Carlyon, R. P., & Collet, L. (2002). Evidence for two pitch encoding mechanisms using a selective auditory training paradigm. Percept. Psychophys., 64(2), 189–197.

[Jesteadt1980] Jesteadt,W. (1980). An adaptive procedure for subjective judgments. Percept Psychophys, 28(1), 85–88.

[Kaernbach1991] Kaernbach, C. (1991). Simple adaptive testing with the weighted up-down method. Percept Psychophys, 49(3), 227–229.

[KingEtAl2013] King, A., Hopkins, K., & Plack, C. J. (2013). Differences in short-term training for interaural phase difference discrimination between two different forced-choice paradigms. J. Acoust. Soc. Am., 134(4), 2635. doi:10.1121/1.4819116

[Levitt1971] Levitt, H. (1971). Transformed up-down methods in psychoacoustics. J. Acoust. Soc. Am., 49(2), 467-477.

[MacmillanAndCreelman2005] Macmillan, N. A., & Creelman, C. D. (2005). Detection theory: A user's guide (2dn ed.). Mahwah, NJ: Lawrence Erlbaum Associates.

[Prins2013] Prins, N. (2013). The psi-marginal adaptive method: How to give nuisance parameters the attention they deserve (no more, no less). Journal of Vision, 13, 1–17. doi:10.1167/13.7.3

[ShenAndRichards2012] Shen, Y., & Richards, V. (2012). A maximum-likelihood procedure for estimating psychometric functions: Thresholds, slopes, and lapses of attention. J. Acoust. Soc. Am., 132, 957–967. doi:10.1121/1.4733540

[TaylorAndCreelman1967] Taylor, M., & Creelman, C. (1967). PEST: Efficient estimates on probability functions. J. Acoust. Soc. Am., 41(4A), 782-787. doi:10.1121/1.1910407

[VersfeldEtAl1996] Versfeld, N. J., Dai, H., & Green, D. M. (1996). The optimum decision rules for the oddity task. Percept. Psychophys., 58(1), 10–21.

Python Module Index

p

s

Index