brian2hears Documentation

Brian authors

Jan 26, 2021

Contents

Brian hears (for Brian 2) is an auditory modelling library for Python. It is built as an extension to the neural network simulator package Brian 2, but can also be used on its own.

Note that this is a direct update of the original Brian hears package for Brian 1. Some features available in Brian 2 will not work with this package, see *Update for Brian 2*.

CHAPTER 1

Contents

1.1 Introduction

1.1.1 Download and installation

If you do not already have a recent version of Brian2, please install it following the installation instructions in its documentation.

To download and install Brian2Hears, use pip:

```
pip install brian2hears
```

1.1.2 Getting started

Brian hears is primarily designed for generating and manipulating sounds, and applying large banks of filters. We import the package by writing:

```
from brian2 import *
from brian2hears import *
```

Then, for example, to generate a tone or a whitenoise we would write:

```
sound1 = tone(1*kHz, .1*second)
sound2 = whitenoise(.1*second)
```

These sounds can then be manipulated in various ways, for example:

```
sound = sound1+sound2
sound = sound.ramp()
```

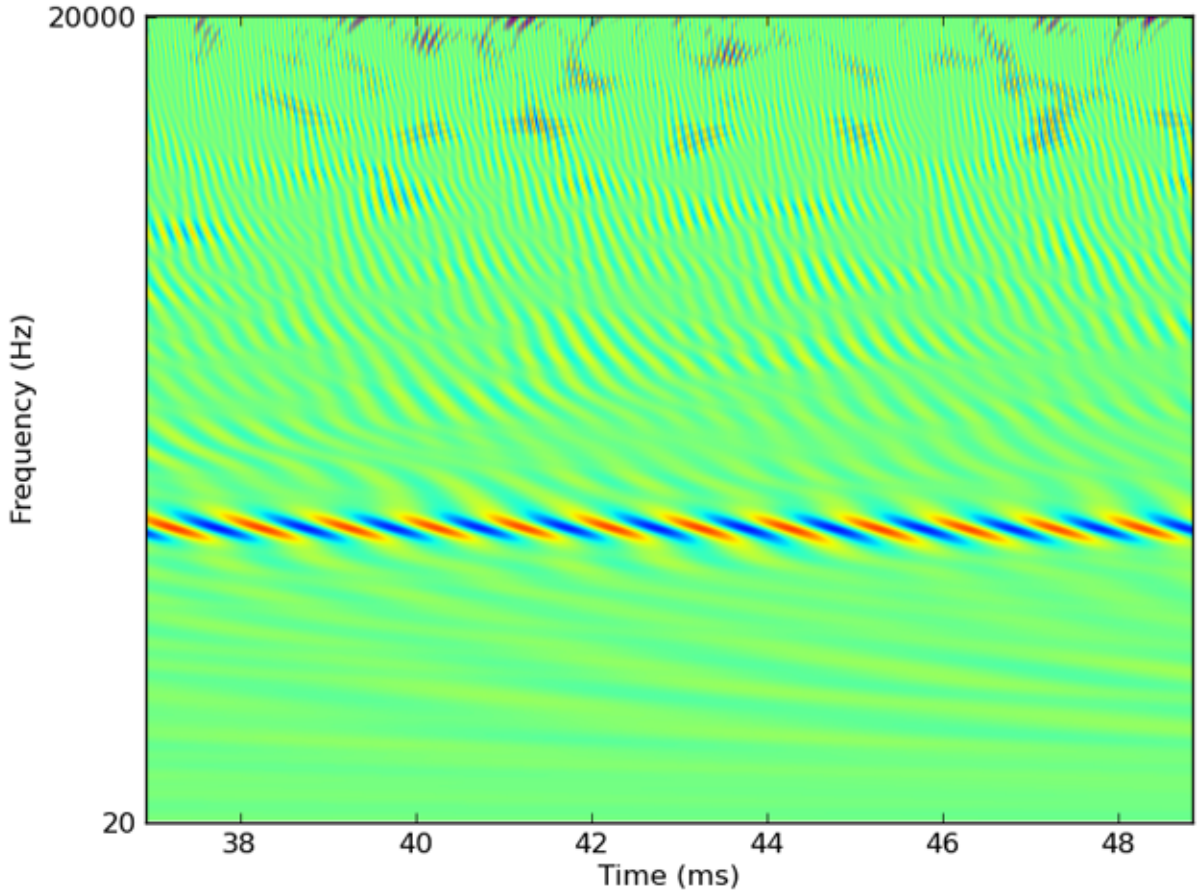
If you have the pygame package installed, you can also play these sounds:

```
sound.play()
```

We can filter these sounds through a bank of 3000 gammatone filters covering the human auditory range as follows:

```
cf = erbspace(20*Hz, 20*kHz, 3000)
fb = Gammatone(sound, cf)
output = fb.process()
```

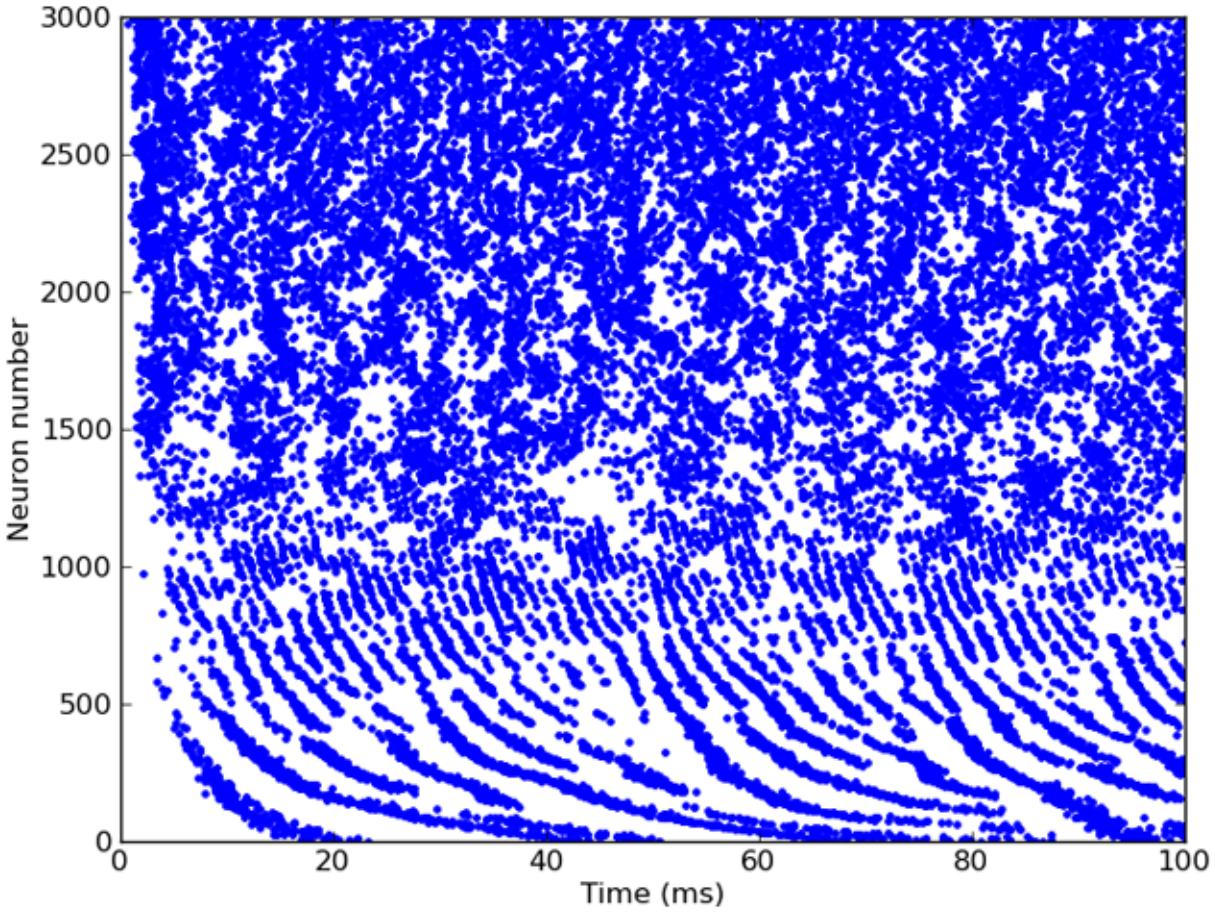
The output of this would look something like this (zoomed into one region):



Alternatively, if we're interested in modelling auditory nerve fibres, we could feed the output of this filterbank directly into a group of neurons defined with Brian:

```
# Half-wave rectification and compression [x]^(1/3)
ihc = FunctionFilterbank(fb, lambda x: 3*clip(x, 0, Inf)**(1.0/3.0))
# Leaky integrate-and-fire model with noise and refractoriness
eqs = '''
dv/dt = (I-v)/(1*ms)+0.2*xi*(2/(1*ms))**.5 : 1 (unless refractory)
I : 1
'''
anf = FilterbankGroup(ihc, 'I', eqs, reset='v=0', threshold='v>1', refractory=5*ms)
```

This model would give output something like this:



The human cochlea applies the equivalent of 3000 auditory filters, which causes a technical problem for modellers which this package is designed to address. At a typical sample rate, the output of 3000 filters would saturate the computer's RAM in a few seconds. To deal with this, we use online computation, that is we only ever keep in memory the output of the filters for a relatively short duration (say, the most recent 20ms), do our modelling with these values, and then discard them. Although this requires that some models be rewritten for online rather than offline computation, it allows us to easily handle models with very large numbers of channels. 3000 or 6000 for human monaural or binaural processing is straightforward, and even much larger banks of filters can be used (for example, around 30,000 in Goodman DFM, Brette R (2010). Spike-timing-based computation in sound localization. PLoS Comput. Biol. 6(11): e1000993. doi:10.1371/journal.pcbi.1000993). Techniques for online computation are discussed below in the section *Online computation*.

Brian hears consists of classes and functions for defining *Sounds*, *Filter chains*, cochlear models, neuron models and *Head-related transfer functions*. These classes are designed to be modular and easily extendable. Typically, a model will consist of a chain starting with a sound which is plugged into a chain of filter banks, which are then plugged into a neuron model.

The two main classes in Brian hears are `Sound` and `Filterbank`, which function very similarly. Each consists of multiple channels (typically just 1 or 2 in the case of sounds, and many in the case of filterbanks, but in principle any number of channels is possible for either). The difference is that a filterbank has an input source, which can be either a sound or another filterbank.

All scripts using Brian hears should start by importing the Brian and Brian hears packages as follows:

```
from brian2 import *
from brian2hears import *
```

See also:

Reference documentation for Brian2Hears, which covers everything in this overview in detail, and more. List of *examples of using Brian hears*.

1.1.3 Update for Brian 2

For users of Brian hears for Brian 1, note that the following no longer works in Python 2 (although it will work in Python 3):

```
sound = whitenoise(100*ms)
sound[:10*ms] # to get the first 10 ms of a sound
```

This is because of a change in the way units are handled between Brian 1 and Brian 2. To get the same effect, you can write:

```
sound = whitenoise(100*ms)
sound[slice(0*ms, 10*ms)] # to get the first 10 ms of a sound
```

This will work in both Python 2 and 3.

For users of Brian 2, note that the following will not work with *FilterbankGroup*:

- `store()` and `restore()` will not work unless you are calling `store()` at time t=0.
- The standalone mode of Brian 2 will not work.

1.2 Sounds

Sounds can be loaded from a WAV or AIFF file with the *loadsound()* function (and saved with the *savesound()* function or *Sound.save()* method), or by initialising with a filename:

```
sound = loadsound('test.wav')
sound = Sound('test.aif')
sound.save('test.wav')
```

Various standard types of sounds can also be constructed, e.g. pure tones, white noise, clicks and silence:

```
sound = tone(1*kHz, 1*second)
sound = whitenoise(1*second)
sound = click(1*ms)
sound = silence(1*second)
```

You can pass a function of time or an array to initialise a sound:

```
# Equivalent to Sound.tone
sound = Sound(lambda t:sin(50*Hz*2*pi*t), duration=1*second)

# Equivalent to Sound.whitenoise
sound = Sound(randn(int(1*second*44.1*kHz)), samplerate=44.1*kHz)
```

Multiple channel sounds can be passed as a list or tuple of filenames, arrays or *Sound* objects:

```
sound = Sound(('left.wav', 'right.wav'))
sound = Sound((randn(44100), randn(44100)), samplerate=44.1*kHz)
sound = Sound((Sound.tone(1*kHz, 1*second),
               Sound.tone(2*kHz, 1*second)))
```

A multi-channel sound is also a numpy array of shape `(nsamples, nchannels)`, and can be initialised as this (or converted to a standard numpy array):

```
sound = Sound(randn(44100, 2), samplerate=44.1*kHz)
arr = array(sound)
```

Sounds can be added and multiplied:

```
sound = Sound.tone(1*kHz, 1*second)+0.1*Sound.whitenoise(1*second)
```

For more details on combining and operating on sounds, including shifting them in time, repeating them, resampling them, ramping them, finding and setting intensities, plotting spectrograms, etc., see *Sound*.

Sounds can be played using the *play()* function or *Sound.play()* method:

```
play(sound)
sound.play()
```

Sequences of sounds can be played as:

```
play(sound1, sound2, sound3)
```

The number of channels in a sound can be found using the `nchannels` attribute, and individual channels can be extracted using the *Sound.channel()* method, or using the `left` and `right` attributes in the case of stereo sounds:

```
print sound.nchannels
print amax(abs(sound.left-sound.channel(0)))
```

As an example of using this, the following swaps the channels in a stereo sound:

```
sound = Sound('test_stereo.wav')
swappedsound = Sound((sound.right, sound.left))
swappedsound.play()
```

The level of the sound can be computed and changed with the `sound.level` attribute. Levels are returned in dB which is a special unit in Brian hears. For example, `10*dB+10` will raise an error because `10` does not have units of dB. The multiplicative gain of a value in dB can be computed with the function `gain(level)`. All dB values are measured as RMS dB SPL assuming that the values of the sound object are measured in Pascals. Some examples:

```
sound = whitenoise(100*ms)
print sound.level
sound.level = 60*dB
sound.level += 10*dB
sound *= gain(-10*dB)
```

1.3 Filter chains

The standard way to set up a model based on filterbanks is to start with a sound and then construct a chain of filterbanks that modify it, for example a common model of cochlear filtering is to apply a bank of gammatone filters, and then

half wave rectify and compress it (for example, with a 1/3 power law). This can be achieved in Brian hears as follows (for 3000 channels in the human hearing range from 20 Hz to 20 kHz):

```
cfmin, cfmax, cfN = 20*Hz, 20*kHz, 3000
cf = erbspace(cfmin, cfmax, cfN)
sound = Sound('test.wav')
gfb = GammatoneFilterbank(sound, cf)
ihc = FunctionFilterbank(gfb, lambda x: clip(x, 0, Inf)**(1.0/3.0))
```

The *erbspace()* function constructs an array of centre frequencies on the ERB scale. The `GammatoneFilterbank(source, cf)` class creates a bank of gammatone filters with inputs coming from `source` and the centre frequencies in the array `cf`. The `FunctionFilterbank(source, func)` creates a bank of filters that applies the given function `func` to the inputs in `source`.

Filterbanks can be added and multiplied, for example for creating a linear and nonlinear path, e.g.:

```
sum_path_fb = 0.1*linear_path_fb+0.2*nonlinear_path_fb
```

A filterbank must have an input with either a single channel or an equal number of channels. In the former case, the single channel is duplicated for each of the output channels. However, you might want to apply gammatone filters to a stereo sound, for example, but in this case it's not clear how to duplicate the channels and you have to specify it explicitly. You can do this using the *Repeat*, *Tile*, *Join* and *Interleave* filterbanks. For example, if the input is a stereo sound with channels LR then you can get an output with channels LLLRRR or LRLRLR by writing (respectively):

```
fb = Repeat(sound, 3)
fb = Tile(sound, 3)
```

To combine multiple filterbanks into one, you can either join them in series or interleave them, as follows:

```
fb = Join(source1, source2)
fb = Interleave(source1, source2)
```

For a more general (but more complicated) approach, see *RestructureFilterbank*.

Two of the most important generic filterbanks (upon which many of the others are based) are *LinearFilterbank* and *FIRFilterbank*. The former is a generic digital filter for FIR and IIR filters. The latter is specifically for FIR filters. These can be implemented with the former, but the implementation is optimised using FFTs with the latter (which can often be hundreds of times faster, particularly for long impulse responses). IIR filter banks can be designed using *IIRFilterbank* which is based on the syntax of the `iirdesign` scipy function.

You can change the input source to a *Filterbank* by modifying its `source` attribute, e.g. to change the input sound of a filterbank `fb` you might do:

```
fb.source = newsound
```

Note that the new source should have the same number of channels.

You can implement control paths (using the output of one filter chain path to modify the parameters of another filter chain path) using *ControlFilterbank* (see reference documentation for more details). For examples of this in action, see the following:

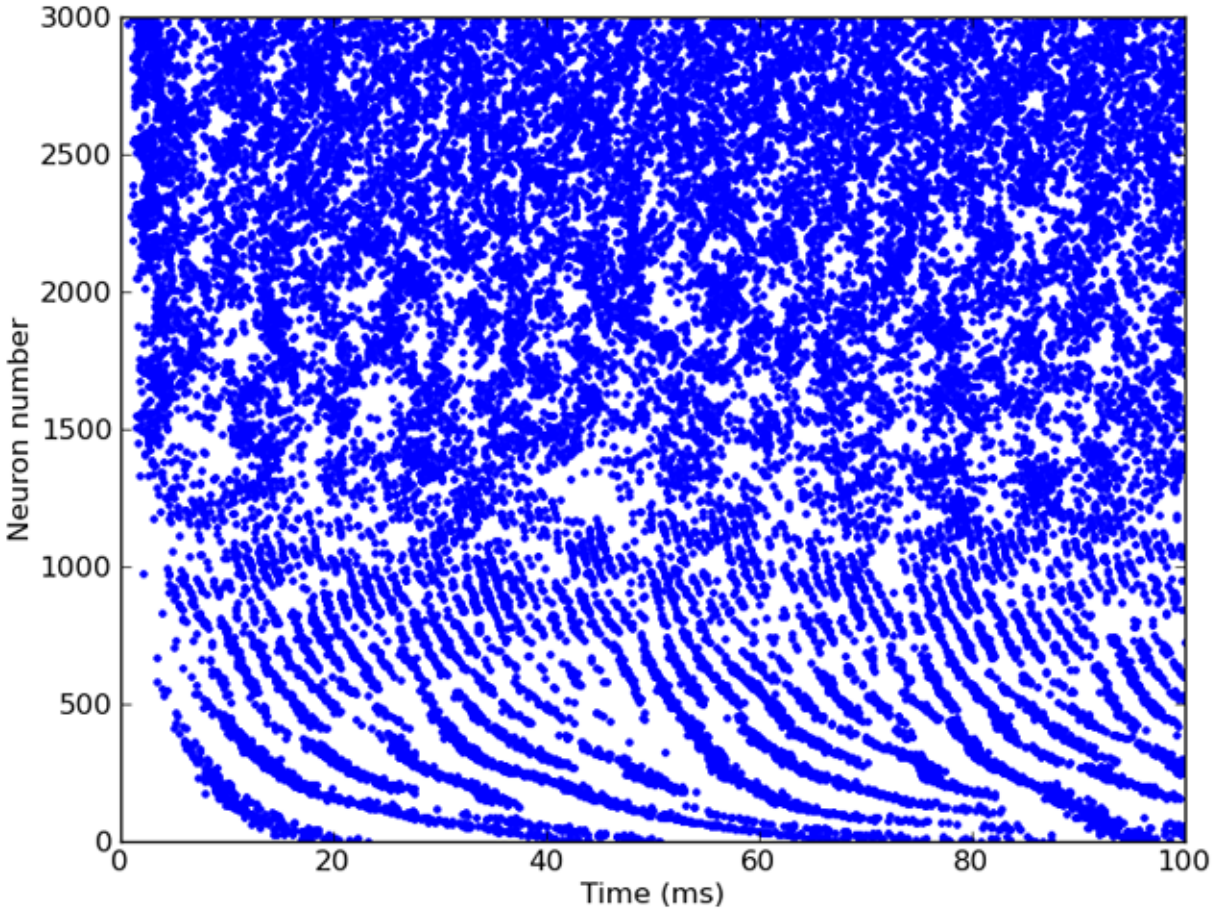
- *Time varying filter (1)*.
- *Time varying filter (2)*.
- *Compressive Gammachirp filter (DCGC)*.

1.4 Connecting with Brian

To create spiking neuron models based on filter chains, you use the *FilterbankGroup* class. This acts exactly like a standard Brian NeuronGroup except that you give a source filterbank and choose a state variable in the target equations for the output of the filterbank. A simple auditory nerve fibre model would take the inner hair cell model from earlier, and feed it into a noisy leaky integrate-and-fire model as follows:

```
# Inner hair cell model as before
cfmin, cfmax, cfN = 20*Hz, 20*kHz, 3000
cf = erbspace(cfmin, cfmax, cfN)
sound = Sound.whitenoise(100*ms)
gfb = Gammatone(sound, cf)
ihc = FunctionFilterbank(gfb, lambda x: 3*clip(x, 0, Inf)**(1.0/3.0))
# Leaky integrate-and-fire model with noise and refractoriness
eqs = '''
dv/dt = (I-v)/(1*ms)+0.2*xi*(2/(1*ms))**.5 : 1 (unless refractory)
I : 1
'''
G = FilterbankGroup(ihc, 'I', eqs, reset='v=0', threshold='v>1', refractory=5*ms)
# Run, and raster plot of the spikes
M = SpikeMonitor(G)
run(sound.duration)
plot(M.t/ms, M.i, '.')
show()
```

And here's the output:



1.5 Plotting

Often, you want to use log-scaled axes for frequency in plots, but the built-in matplotlib axis labelling for log-scaled axes doesn't work well for frequencies. We provided two functions (*log_frequency_xaxis_labels()* and *log_frequency_yaxis_labels()*) to automatically set useful axis labels. For example:

```
cf = erbspace(100*Hz, 10*kHz)
...
semilogx(cf, response)
axis('tight')
log_frequency_xaxis_labels()
```

1.6 Online computation

Typically in auditory modelling, we precompute the entire output of each channel of the filterbank ("offline computation"), and then work with that. This is straightforward, but puts a severe limit on the number of channels we can use or the length of time we can work with (otherwise the RAM would be quickly exhausted). Brian hears allows us to use a very large number of channels in filterbanks, but at the cost of only storing the output of the filterbanks for a relatively short period of time ("online computation"). This requires a slight change in the way we use the output of the filterbanks, but is actually not too difficult. For example, suppose we wanted to compute the vector of RMS values for

each channel of the output of the filterbank. Traditionally, or if we just use the syntax `output = fb.process()` in Brian hears, we have an array `output` of shape `(nsamples, nchannels)`. We could compute the vector of RMS values as:

```
rms = sqrt(mean(output**2, axis=0))
```

To do the same thing with online computation, we simply store a vector of the running sum of squares, and update it for each buffered segment as it is computed. At the end of the processing, we divide the sum of squares by the number of samples and take the square root.

The *`Filterbank.process()`* method allows us to pass an optional function `f(output, running)` of two arguments. In this case, *`process()`* will first call `running = f(output, 0)` for the first buffered segment `output`. It will then call `running = f(output, running)` for each subsequent segment. In other words, it will "accumulate" the output of `f`, passing the output of each call to the subsequent call. To compute the vector of RMS values then, we simply do:

```
def sum_of_squares(input, running):
    return running+sum(input**2, axis=0)

rms = sqrt(fb.process(sum_of_squares)/nsamples)
```

If the computation you wish to perform is more complicated than can be achieved with the *`process()`* method, you can derive a class from *`Filterbank`* (see that class' reference documentation for more details on this).

1.7 Buffering interface

The *`Sound`* and *`Filterbank`* classes (and all classes derived from them) all implement the same buffering mechanism. The purpose of this is to allow for efficient processing of multiple channels in buffers. Rather than precomputing the application of filters to all channels (which for large numbers of channels or long sounds would not fit in memory), we process small chunks at a time. The entire design of these classes is based on the idea of buffering, as defined by the base class *`Bufferable`* (see section *Class diagram*). Each class has two methods, `buffer_init()` to initialise the buffer, and `buffer_fetch(start, end)` to fetch the portion of the buffer from samples with indices from `start` to `end` (not including `end` as standard for Python). The `buffer_fetch(start, end)` method should return a 2D array of shape `(end-start, nchannels)` with the buffered values.

From the user point of view, all you need to do, having set up a chain of *`Sound`* and *`Filterbank`* objects, is to call `buffer_fetch(start, end)` repeatedly. If the output of a *`Filterbank`* is being plugged into a *`FilterbankGroup`* object, everything is handled automatically. For cases where the number of channels is small or the length of the input source is short, you can use the *`Filterbank.process()`* method to automatically handle the initialisation and repeated application of `buffer_fetch`.

To extend *`Filterbank`*, it is often sufficient just to implement the `buffer_apply(input)` method. See the documentation for *`Filterbank`* for more details.

1.8 Library

Brian hears comes with a package of predefined filter classes to be used as basic blocks by the user. All of them are implemented as filterbanks.

First, a series of standard filters widely used in audio processing are available:

Class	Descripition	Example
IIRFilterbank	Bank of low, high, bandpass or bandstop filter of type Chebyshef, Elliptic, etc...	*IIR filterbank*
Butterworth	Bank of low, high, bandpass or bandstop Butterworth filters	*Butterworth filters*
LowPass	Bank of lowpass filters of order 1	*Cochleagram*

Second, the library provides linear auditory filters developed to model the middle ear transfer function and the frequency analysis of the cochlea:

Class	Description	Example
MiddleEar	Linear bandpass filter, based on middle-ear frequency response properties	*Spiking output of the Tan&Carney model*
Gammatone	Bank of IIR gammatone filters (based on Slaney implementation)	*Gammatone filters*
ApproximateGammatone	Bank of IIR gammatone filters (based on Hohmann implementation)	*Approximate Gammatone filters*
LogGammachirp	Bank of IIR gammachirp filters with logarithmic sweep (based on Irino implementation)	*Logarithmic Gammachirp filters*
LinearGammachirp	Bank of FIR chirp filters with linear sweep and gamma envelope	*Linear Gammachirp filters*
LinearGaborchirp	Bank of FIR chirp filters with linear sweep and gaussian envelope	

Finally, Brian hears comes with a series of complex nonlinear cochlear models developed to model nonlinear effects such as filter bandwith level dependency, two-tones suppression, peak position level dependency, etc.

Class	Description	Example
DRNL	Dual resonance nonlinear filter as described in Lopez-Paveda and Meddis, JASA 2001	*Dual resonance nonlinear filter (DRNL)*
DCGC	Compressive gammachirp auditory filter as described in Irino and Patterson, JASA 2001	*Compressive Gammachirp filter (DCGC)*
TanCarney	Auditory phenomenological model as described in Tan and Carney, JASA 2003	*Spiking output of the Tan&Carney model*
ZhangSynapse	Model of an inner hair cell – auditory nerve synapse (Zhang et al., JASA 2001)	*Spiking output of the Tan&Carney model*

1.9 Head-related transfer functions

You can work with head-related transfer functions (HRTFs) using the three classes HRTF (a single pair of left/right ear HRTFs), HRTFSet (a set of HRTFs, typically for a single individual), and HRTFDatabase (for working with databases of individuals). At the moment, we have included only one HRTF database, the IRCAM_LISTEN public HRTF database. There is also one artificial HRTF database, HeadlessDatabase used for generating HRTFs of artifically introduced ITDs.

An example of loading the IRCAM database, selecting a subject and plotting the pair of impulse responses for a particular direction:

```
hrtfdb = IRCAM_LISTEN()
hrtfset = hrtfdb.load_subject(1002)
hrtf = hrtfset(azim=30, elev=15)
plot(hrtf.left)
plot(hrtf.right)
show()
```

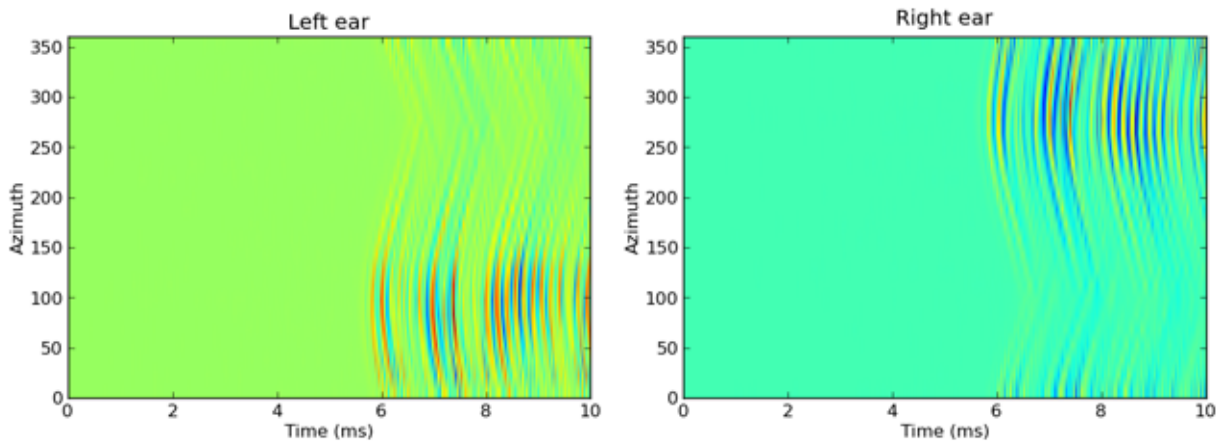
HRTFSet has a set of coordinates, which can be accessed via the `coordinates` attribute, e.g.:

```
print hrtfset.coordinates['azim']
print hrtfset.coordinates['elev']
```

You can also generated filterbanks associated either to an *HRTF* or an entire *HRTFSet*. Here is an example of doing this with the IRCAM database, and applying this filterbank to some white noise and plotting the response as an image:

```
# Load database
hrtfdb = IRCAM_LISTEN()
hrtfset = hrtfdb.load_subject(1002)
# Select only the horizontal plane
hrtfset = hrtfset.subset(lambda elev: elev==0)
# Set up a filterbank
sound = whitenoise(10*ms)
fb = hrtfset.filterbank(sound)
# Extract the filtered response and plot
img = fb.process().T
img_left = img[:img.shape[0]/2, :]
img_right = img[img.shape[0]/2:, :]
subplot(121)
imshow(img_left, origin='lower', aspect='auto',
       extent=(0, sound.duration/ms, 0, 360))
xlabel('Time (ms)')
ylabel('Azimuth')
title('Left ear')
subplot(122)
imshow(img_right, origin='lower', aspect='auto',
       extent=(0, sound.duration/ms, 0, 360))
xlabel('Time (ms)')
ylabel('Azimuth')
title('Right ear')
show()
```

This generates the following output:



For more details, see the reference documentation for *HRTF*, *HRTFSet*, *HRTFDatabase*, *IRCAM_LISTEN* and *HeadlessDatabase*.

1.10 Reference

brian2hears.**set_default_samplerate**(*samplerate*)
Sets the default samplerate for Brian hears objects, by default 44.1 kHz.

1.10.1 Sounds

class brian2hears.**Sound**
Class for working with sounds, including loading/saving, manipulating and playing.

For an overview, see *Sounds*.

Initialisation

The following arguments are used to initialise a sound object

data Can be a filename, an array, a function or a sequence (list or tuple). If its a filename, the sound file (WAV or AIFF) will be loaded. If its an array, it should have shape `(nsamples, nchannels)`. If its a function, it should be a function f(t). If its a sequence, the items in the sequence can be filenames, functions, arrays or Sound objects. The output will be a multi-channel sound with channels the corresponding sound for each element of the sequence.

samplerate=None The samplerate, if necessary, will use the default (for an array or function) or the samplerate of the data (for a filename).

duration=None The duration of the sound, if initialising with a function.

Loading, saving and playing

static load(*filename*)
Load the file given by filename and returns a Sound object. Sound file can be either a .wav or a .aif file.

save(*filename*, *normalise=False*, *samplewidth=2*)
Save the sound as a WAV.

If the normalise keyword is set to True, the amplitude of the sound will be normalised to 1. The samplewidth keyword can be 1 or 2 to save the data as 8 or 16 bit samples.

play (*normalise=False*, *sleep=False*)
Plays the sound (normalised to avoid clipping if required). If sleep=True then the function will wait until the sound has finished playing before returning.

Properties

duration
The length of the sound in seconds.

nsamples
The number of samples in the sound.

nchannels
The number of channels in the sound.

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

left
The left channel for a stereo sound.

right
The right channel for a stereo sound.

channel (*n*)
Returns the nth channel of the sound.

Generating sounds

All sound generating methods can be used with durations arguments in samples (int) or units (e.g. 500*ms). One can also set the number of channels by setting the keyword argument nchannels to the desired value. Notice that for noise the channels will be generated independantly.

static tone (*frequency*, *duration*, *phase=0*, *samplerate=None*, *nchannels=1*)
Returns a pure tone at frequency for duration, using the default samplerate or the given one. The `frequency` and `phase` parameters can be single values, in which case multiple channels can be specified with the `nchannels` argument, or they can be sequences (lists/tuples/arrays) in which case there is one frequency or phase for each channel.

static whitenoise (*duration*, *samplerate=None*, *nchannels=1*)
Returns a white noise. If the samplerate is not specified, the global default value will be used.

static powerlawnoise (*duration*, *alpha*, *samplerate=None*, *nchannels=1*, *normalise=False*)
Returns a power-law noise for the given duration. Spectral density per unit of bandwidth scales as 1/(f**alpha).

Sample usage:

```
noise = powerlawnoise(200*ms, 1, samplerate=44100*Hz)
```

Arguments:

duration Duration of the desired output.

alpha Power law exponent.

samplerate Desired output samplerate

static brownnoise (*duration*, *samplerate=None*, *nchannels=1*, *normalise=False*)
Returns brown noise, i.e *powerlawnoise()* with alpha=2

static pinknoise (*duration*, *samplerate=None*, *nchannels=1*, *normalise=False*)
Returns pink noise, i.e *powerlawnoise()* with alpha=1

static silence(*duration*, *samplerate=None*, *nchannels=1*)
Returns a silent, zero sound for the given duration. Set nchannels to set the number of channels.

static click(*duration=1*, *peak=None*, *samplerate=None*, *nchannels=1*)
Returns a click of the given duration (in time or samples)

If `peak` is not specified, the amplitude will be 1, otherwise `peak` refers to the peak dB SPL of the click, according to the formula `28e-6*10**(peak/20.)`.

static clicks(*duration*, *n*, *interval*, *peak=None*, *samplerate=None*, *nchannels=1*)
Returns a series of n clicks (see *click()*) separated by interval.

static harmoniccomplex(*f0*, *duration*, *amplitude=1*, *phase=0*, *samplerate=None*, *nchannels=1*)
Returns a harmonic complex composed of pure tones at integer multiples of the fundamental frequency `f0`. The `amplitude` and `phase` keywords can be set to either a single value or an array of values. In the former case the value is set for all harmonics, and harmonics up to the sampling frequency are generated. In the latter each harmonic parameter is set separately, and the number of harmonics generated corresponds to the length of the array.

static vowel(*vowel=None*, *formants=None*, *pitch=100.0*, *duration=1.0*, *samplerate=None*, *nchannels=1*)
Returns an artifically created spoken vowel sound (following the source-filter model of speech production) with a given `pitch`.

The vowel can be specified by either providing `vowel` as a string ('a', 'i' or 'u') or by setting `formants` to a sequence of formant frequencies.

The returned sound is normalized to a maximum amplitude of 1.

The implementation is based on the MakeVowel function written by Richard O. Duda, part of the Auditory Toolbox for Matlab by Malcolm Slaney: https://engineering.purdue.edu/~malcolm/interval/1998-010/

Timing and sequencing

static sequence(**sounds*, *samplerate=None*)
Returns the sequence of sounds in the list sounds joined together

repeat(*n*)
Repeats the sound n times

extended(*duration*)
Returns the Sound with length extended by the given duration, which can be the number of samples or a length of time in seconds.

shifted(*duration*, *fractional=False*, *filter_length=2048*)
Returns the sound delayed by duration, which can be the number of samples or a length of time in seconds. Normally, only integer numbers of samples will be used, but if `fractional=True` then the filtering method from http://www.labbookpages.co.uk/audio/beamforming/fractionalDelay.html will be used (introducing some small numerical errors). With this method, you can specify the `filter_length`, larger values are slower but more accurate, especially at higher frequencies. The large default value of 2048 samples provides good accuracy for sounds with frequencies above 20 Hz, but not for lower frequency sounds. If you are restricted to high frequency sounds, a smaller value will be more efficient. Note that if `fractional=True` then `duration` is assumed to be a time not a number of samples.

resized(*L*)
Returns the Sound with length extended (or contracted) to have L samples.

Slicing

One can slice sound objects in various ways, for example `sound[100*ms:200*ms]` returns the part of the sound between 100 ms and 200 ms (not including the right hand end point). If the sound is less than 200 ms long it will be zero padded. You can also set values using slicing, e.g. `sound[:50*ms] = 0` will silence the

first 50 ms of the sound. The syntax is the same as usual for Python slicing. In addition, you can select a subset of the channels by doing, for example, `sound[:, -5:]` would be the last 5 channels. For time indices, either times or samples can be given, e.g. `sound[:100]` gives the first 100 samples. In addition, steps can be used for example to reverse a sound as `sound[::-1]`.

Note that slicing with units of time rather than samples will only work in Python 3. In Python 2, you can get the same effect by writing, for example, `sound[slice(0*ms, 10*ms)]`. This is a change from the original version of `brian.hears`.

Arithmetic operations

Standard arithemetical operations and numpy functions work as you would expect with sounds, e.g. `sound1+sound2`, `3*sound` or `abs(sound)`.

Level

`level`
Can be used to get or set the level of a sound, which should be in dB. For single channel sounds a value in dB is used, for multiple channel sounds a value in dB can be used for setting the level (all channels will be set to the same level), or a list/tuple/array of levels. It is assumed that the unit of the sound is Pascals.

`atlevel`(*level*)
Returns the sound at the given level in dB SPL (RMS) assuming array is in Pascals. `level` should be a value in dB, or a tuple of levels, one for each channel.

`maxlevel`
Can be used to set or get the maximum level of a sound. For mono sounds, this is the same as the level, but for multichannel sounds it is the maximum level across the channels. Relative level differences will be preserved. The specified level should be a value in dB, and it is assumed that the unit of the sound is Pascals.

`atmaxlevel`(*level*)
Returns the sound with the maximum level across channels set to the given level. Relative level differences will be preserved. The specified level should be a value in dB and it is assumed that the unit of the sound is Pascals.

Ramping

`ramp`(*when='onset'*, *duration=0.01*, *envelope=None*, *inplace=True*)
Adds a ramp on/off to the sound

`when='onset'` Can take values 'onset', 'offset' or 'both'

`duration=10*ms` The time over which the ramping happens

`envelope` A ramping function, if not specified uses `sin(pi*t/2)**2`. The function should be a function of one variable `t` ranging from 0 to 1, and should increase from `f(0)=0` to `f(0)=1`. The reverse is applied for the offset ramp.

`inplace` Whether to apply ramping to current sound or return a new array.

`ramped`(*when='onset'*, *duration=0.01*, *envelope=None*)
Returns a ramped version of the sound (see *`Sound.ramp()`*).

Plotting

`spectrogram`(*low=None*, *high=None*, *log_power=True*, *other=None*, ***kwds*)
Plots a spectrogram of the sound

Arguments:

`low=None, high=None` If these are left unspecified, it shows the full spectrogram, otherwise it shows only between `low` and `high` in Hz.

log_power=True If True the colour represents the log of the power.

****kwds** Are passed to Pylab's `specgram` command.

Returns the values returned by pylab's `specgram`, namely `(pxx, freqs, bins, im)` where `pxx` is a 2D array of powers, `freqs` is the corresponding frequencies, `bins` are the time bins, and `im` is the image axis.

spectrum(*low=None*, *high=None*, *log_power=True*, *display=False*)
Returns the spectrum of the sound and optionally plots it.

Arguments:

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

log_power=True If True it returns the log of the power.

display=False Whether to plot the output.

Returns `(Z, freqs, phase)` where `Z` is a 1D array of powers, `freqs` is the corresponding frequencies, `phase` is the unwrapped phase of spectrum.

brian2hears.**savesound**(*sound*, *filename*, *normalise=False*, *samplewidth=2*)
Save the sound as a WAV.

If the normalise keyword is set to True, the amplitude of the sound will be normalised to 1. The samplewidth keyword can be 1 or 2 to save the data as 8 or 16 bit samples.

brian2hears.**loadsound**(*filename*)
Load the file given by filename and returns a Sound object. Sound file can be either a .wav or a .aif file.

brian2hears.**play**(**sounds*, *normalise=False*, *sleep=False*)
Plays the sound (normalised to avoid clipping if required). If sleep=True then the function will wait until the sound has finished playing before returning.

brian2hears.**whitenoise**(*duration*, *samplerate=None*, *nchannels=1*)
Returns a white noise. If the samplerate is not specified, the global default value will be used.

brian2hears.**powerlawnoise**(*duration*, *alpha*, *samplerate=None*, *nchannels=1*, *normalise=False*)
Returns a power-law noise for the given duration. Spectral density per unit of bandwidth scales as 1/(f**alpha).

Sample usage:

```
noise = powerlawnoise(200*ms, 1, samplerate=44100*Hz)
```

Arguments:

duration Duration of the desired output.

alpha Power law exponent.

samplerate Desired output samplerate

brian2hears.**brownnoise**(*duration*, *samplerate=None*, *nchannels=1*, *normalise=False*)
Returns brown noise, i.e *powerlawnoise()* with alpha=2

brian2hears.**pinknoise**(*duration*, *samplerate=None*, *nchannels=1*, *normalise=False*)
Returns pink noise, i.e *powerlawnoise()* with alpha=1

brian2hears.**irns**(*delay*, *gain*, *niter*, *duration*, *samplerate=None*, *nchannels=1*)
Returns an IRN_S noise. The iterated ripple noise is obtained trough a cascade of gain and delay filtering. For more details: see Yost 1996 or chapter 15 in Hartman Sound Signal Sensation.

brian2hears.**irno**(*delay*, *gain*, *niter*, *duration*, *samplerate=None*, *nchannels=1*)
Returns an IRN_O noise. The iterated ripple noise is obtained many attenuated and delayed version of the original broadband noise. For more details: see Yost 1996 or chapter 15 in Hartman Sound Signal Sensation.

brian2hears.**tone**(*frequency*, *duration*, *phase=0*, *samplerate=None*, *nchannels=1*)
Returns a pure tone at frequency for duration, using the default samplerate or the given one. The `frequency` and `phase` parameters can be single values, in which case multiple channels can be specified with the `nchannels` argument, or they can be sequences (lists/tuples/arrays) in which case there is one frequency or phase for each channel.

brian2hears.**click**(*duration=1*, *peak=None*, *samplerate=None*, *nchannels=1*)
Returns a click of the given duration (in time or samples)

If `peak` is not specified, the amplitude will be 1, otherwise `peak` refers to the peak dB SPL of the click, according to the formula `28e-6*10**(peak/20.)`.

brian2hears.**clicks**(*duration*, *n*, *interval*, *peak=None*, *samplerate=None*, *nchannels=1*)
Returns a series of n clicks (see *click()*) separated by interval.

brian2hears.**harmoniccomplex**(*f0*, *duration*, *amplitude=1*, *phase=0*, *samplerate=None*, *nchannels=1*)
Returns a harmonic complex composed of pure tones at integer multiples of the fundamental frequency `f0`. The `amplitude` and `phase` keywords can be set to either a single value or an array of values. In the former case the value is set for all harmonics, and harmonics up to the sampling frequency are generated. In the latter each harmonic parameter is set separately, and the number of harmonics generated corresponds to the length of the array.

brian2hears.**silence**(*duration*, *samplerate=None*, *nchannels=1*)
Returns a silent, zero sound for the given duration. Set nchannels to set the number of channels.

brian2hears.**sequence**(**sounds*, *samplerate=None*)
Returns the sequence of sounds in the list sounds joined together

dB

class brian2hears.**dB_type**
The type of values in dB.

dB values are assumed to be RMS dB SPL assuming that the sound source is measured in Pascals.

class brian2hears.**dB_error**
Error raised when values in dB are used inconsistently with other units.

1.10.2 Filterbanks

class brian2hears.**LinearFilterbank**(*source*, *b*, *a*)
Generalised linear filterbank

Initialisation arguments:

source The input to the filterbank, must have the same number of channels or just a single channel. In the latter case, the channels will be replicated.

b, a The coeffs b, a must be of shape `(nchannels, m)` or `(nchannels, m, p)`. Here `m` is the order of the filters, and `p` is the number of filters in a chain (first you apply `[:, :, 0]`, then `[:, :, 1]`, etc.).

The filter parameters are stored in the modifiable attributes `filt_b`, `filt_a` and `filt_state` (the variable `z` in the section below).

Has one method:

decascade(*ncascade=1*)
Reduces cascades of low order filters into smaller cascades of high order filters.

`ncascade` is the number of cascaded filters to use, which should be a divisor of the original number.

Note that higher order filters are often numerically unstable.

Notes

These notes adapted from scipy's `lfilter()` function.

The filterbank is implemented as a direct II transposed structure. This means that for a single channel and element of the filter cascade, the output y for an input x is defined by:

```
a[0]*y[m] = b[0]*x[m] + b[1]*x[m-1] + ... + b[m]*x[0]
                      - a[1]*y[m-1] - ... - a[m]*y[0]
```

using the following difference equations:

```
y[i] = b[0]*x[i] + z[0,i-1]
z[0,i] = b[1]*x[i] + z[1,i-1] - a[1]*y[i]
...
z[m-3,i] = b[m-2]*x[i] + z[m-2,i-1] - a[m-2]*y[i]
z[m-2,i] = b[m-1]*x[i] - a[m-1]*y[i]
```

where i is the output sample number.

The rational transfer function describing this filter in the z-transform domain is:

```
                    -1              -nb
        b[0] + b[1]z  + ... + b[m] z
Y(z) = --------------------------------- X(z)
                    -1              -na
        a[0] + a[1]z  + ... + a[m] z
```

class brian2hears.**FIRFilterbank**(*source*, *impulse_response*, *use_linearfilterbank=False*, *minimum_buffer_size=None*)
Finite impulse response filterbank

Initialisation parameters:

source Source sound or filterbank.

impulse_response Either a 1D array providing a single impulse response applied to every input channel, or a 2D array of shape `(nchannels, ir_length)` for `ir_length` the number of samples in the impulse response. Note that if you are using a multichannel sound `x` as a set of impulse responses, the array should be `impulse_response=array(x.T)`.

minimum_buffer_size=None If specified, gives a minimum size to the buffer. By default, for the FFT convolution based implementation of `FIRFilterbank`, the minimum buffer size will be `3*ir_length`. For maximum efficiency with FFTs, `buffer_size+ir_length` should be a power of 2 (otherwise there will be some zero padding), and `buffer_size` should be as large as possible.

class brian2hears.**RestructureFilterbank**(*source*, *numrepeat=1*, *type='serial'*, *numtile=1*, *indexmapping=None*)
Filterbank used to restructure channels, including repeating and interleaving.

Standard forms of usage:

Repeat mono source N times:

```
RestructureFilterbank(source, N)
```

For a stereo source, N copies of the left channel followed by N copies of the right channel:

```
RestructureFilterbank(source, N)
```

For a stereo source, N copies of the channels tiled as LRLRLR...LR:

```
RestructureFilterbank(source, numtile=N)
```

For two stereo sources AB and CD, join them together in serial to form the output channels in order ABCD:

```
RestructureFilterbank((AB, CD))
```

For two stereo sources AB and CD, join them together interleaved to form the output channels in order ACBD:

```
RestructureFilterbank((AB, CD), type='interleave')
```

These arguments can also be combined together, for example to AB and CD into output channels AABBCCDDAABBCCDDAABBCCDD:

```
RestructureFilterbank((AB, CD), 2, 'serial', 3)
```

The three arguments are the number of repeats before joining, the joining type ('serial' or 'interleave') and the number of tilings after joining. See below for details.

Initialise arguments:

source Input source or list of sources.

numrepeat=1 Number of times each channel in each of the input sources is repeated before mixing the source channels. For example, with repeat=2 an input source with channels `AB` will be repeated to form `AABB`

type='serial' The method for joining the source channels, the options are `'serial'` to join the channels in series, or `'interleave'` to interleave them. In the case of `'interleave'`, each source must have the same number of channels. An example of serial, if the input sources are `abc` and `def` the output would be `abcdef`. For interleave, the output would be `adbecf`.

numtile=1 The number of times the joined channels are tiled, so if the joined channels are `ABC` and `numtile=3` the output will be `ABCABCABC`.

indexmapping=None Instead of specifying the restructuring via `numrepeat, type, numtile` you can directly give the mapping of input indices to output indices. So for a single stereo source input, `indexmapping=[1,0]` would reverse left and right. Similarly, with two mono sources, `indexmapping=[1,0]` would have channel 0 of the output correspond to source 1 and channel 1 of the output corresponding to source 0. This is because the indices are counted in order of channels starting from the first source and continuing to the last. For example, suppose you had two sources, each consisting of a stereo sound, say source 0 was `AB` and source 1 was `CD` then `indexmapping=[1, 0, 3, 2]` would swap the left and right of each source, but leave the order of the sources the same, i.e. the output would be `BADC`.

class brian2hears.**Join**(**sources*)
Filterbank that joins the channels of its inputs in series, e.g. with two input sources with channels AB and CD respectively, the output would have channels ABCD. You can initialise with multiple sources separated by commas, or by passing a list of sources.

class brian2hears.**Interleave**(**sources*)
Filterbank that interleaves the channels of its inputs, e.g. with two input sources with channels AB and CD

respectively, the output would have channels ACBD. You can initialise with multiple sources separated by commas, or by passing a list of sources.

class brian2hears.**Repeat**(*source*, *numrepeat*)
Filterbank that repeats each channel from its input, e.g. with 3 repeats channels ABC would map to AAABBBCCC.

class brian2hears.**Tile**(*source*, *numtile*)
Filterbank that tiles the channels from its input, e.g. with 3 tiles channels ABC would map to ABCABCABC.

class brian2hears.**FunctionFilterbank**(*source*, *func*, *nchannels=None*, ***params*)
Filterbank that just applies a given function. The function should take as many arguments as there are sources.

For example, to half-wave rectify inputs:

```
FunctionFilterbank(source, lambda x: clip(x, 0, Inf))
```

The syntax `lambda x: clip(x, 0, Inf)` defines a function object that takes a single argument `x` and returns `clip(x, 0, Inf)`. The numpy function `clip(x, low, high)` returns the values of `x` clipped between `low` and `high` (so if `x<low` it returns `low`, if `x>high` it returns `high`, otherwise it returns `x`). The symbol `Inf` means infinity, i.e. no clipping of positive values.

Technical details

Note that functions should operate on arrays, in particular on 2D buffered segments, which are arrays of shape `(bufsize, nchannels)`. Typically, most standard functions from numpy will work element-wise.

If you want a filterbank that changes the shape of the input (e.g. changes the number of channels), set the `nchannels` keyword argument to the number of output channels.

class brian2hears.**SumFilterbank**(*source*, *weights=None*)
Sum filterbanks together with given weight vectors.

For example, to take the sum of two filterbanks:

```
SumFilterbank((fb1, fb2))
```

To take the difference:

```
SumFilterbank((fb1, fb2), (1, -1))
```

class brian2hears.**DoNothingFilterbank**(*source*)
Filterbank that does nothing to its input.

Useful for removing a set of filters without having to rewrite your code. Can also be used for simply writing compound derived classes. For example, if you want a compound Filterbank that does AFilterbank and then BFilterbank, but you want to encapsulate that into a single class, you could do:

```
class ABFilterbank(DoNothingFilterbank):
    def __init__(self, source):
        a = AFilterbank(source)
        b = BFilterbank(a)
        DoNothingFilterbank.__init__(self, b)
```

However, a more general way of writing compound filterbanks is to use *CombinedFilterbank*.

class brian2hears.**ControlFilterbank**(*source*, *inputs*, *targets*, *updater*, *max_interval=None*)
Filterbank that can be used for controlling behaviour at runtime

Typically, this class is used to implement a control path in an auditory model, modifying some filterbank parameters based on the output of other filterbanks (or the same ones).

The controller has a set of input filterbanks whose output values are used to modify a set of output filterbanks. The update is done by a user specified function or class which is passed these output values. The controller should be inserted as the last bank in a chain.

Initialisation arguments:

source The source filterbank, the values from this are used unmodified as the output of this filterbank.

inputs Either a single filterbank, or sequence of filterbanks which are used as inputs to the `updater`.

targets The filterbank or sequence of filterbanks that are modified by the updater.

updater The function or class which does the updating, see below.

max_interval If specified, ensures that the updater is called at least as often as this interval (but it may be called more often). Can be specified as a time or a number of samples.

The updater

The `updater` argument can be either a function or class instance. If it is a function, it should have a form like:

```
# A single input
def updater(input):
    ...

# Two inputs
def updater(input1, input2):
    ...

# Arbitrary number of inputs
def updater(*inputs):
    ...
```

Each argument `input` to the function is a numpy array of shape `(numsamples, numchannels)` where `numsamples` is the number of samples just computed, and `numchannels` is the number of channels in the corresponding filterbank. The function is not restricted in what it can do with these inputs.

Functions can be used to implement relatively simple controllers, but for more complicated situations you may want to maintain some state variables for example, and in this case you can use a class. The object `updater` should be an instance of a class that defines the `__call__` method (with the same syntax as above for functions). In addition, you can define a reinitialisation method `reinit()` which will be called when the `buffer_init()` method is called on the filterbank, although this is entirely optional.

Example

The following will do a simple form of gain control, where the gain parameter will drift exponentially towards target_rms/rms with a given time constant:

```
# This class implements the gain (see Filterbank for details)
class GainFilterbank(Filterbank):
    def __init__(self, source, gain=1.0):
        Filterbank.__init__(self, source)
        self.gain = gain
    def buffer_apply(self, input):
        return self.gain*input

# This is the class for the updater object
class GainController(object):
    def __init__(self, target, target_rms, time_constant):
        self.target = target
        self.target_rms = target_rms
```

(continues on next page)

(continued from previous page)

```
        self.time_constant = time_constant
    def reinit(self):
        self.sumsquare = 0
        self.numsamples = 0
    def __call__(self, input):
        T = input.shape[0]/self.target.samplerate
        self.sumsquare += sum(input**2)
        self.numsamples += input.size
        rms = sqrt(self.sumsquare/self.numsamples)
        g = self.target.gain
        g_tgt = self.target_rms/rms
        tau = self.time_constant
        self.target.gain = g_tgt+exp(-T/tau)*(g-g_tgt)
```

And an example of using this with an input `source`, a target RMS of 0.2 and a time constant of 50 ms, updating every 10 ms:

```
gain_fb = GainFilterbank(source)
updater = GainController(gain_fb, 0.2, 50*ms)
control = ControlFilterbank(gain_fb, source, gain_fb, updater, 10*ms)
```

class brian2hears.**CombinedFilterbank**(*source*)

Filterbank that encapsulates a chain of filterbanks internally.

This class should mostly be used by people writing extensions to Brian hears rather than by users directly. The purpose is to take an existing chain of filterbanks and wrap them up so they appear to the user as a single filterbank which can be used exactly as any other filterbank.

In order to do this, derive from this class and in your initialisation follow this pattern:

```
class RectifiedGammatone(CombinedFilterbank):
    def __init__(self, source, cf):
        CombinedFilterbank.__init__(self, source)
        source = self.get_modified_source()
        # At this point, insert your chain of filterbanks acting on
        # the modified source object
        gfb = Gammatone(source, cf)
        rectified = FunctionFilterbank(gfb,
                        lambda input: clip(input, 0, Inf))
        # Finally, set the output filterbank to be the last in your chain
        self.set_output(fb)
```

This combination of a *Gammatone* and a rectification via a *FunctionFilterbank* can now be used as a single filterbank, for example:

```
x = whitenoise(100*ms)
fb = RectifiedGammatone(x, [1*kHz, 1.5*kHz])
y = fb.process()
```

Details

The reason for the `get_modified_source()` call is that the source attribute of a filterbank can be changed after creation. The modified source provides a buffer (in fact, a *DoNothingFilterbank*) so that the input to the chain of filters defined by the derived class doesn't need to be changed.

class brian2hears.**FractionalDelay**(*source*, *delays*, *filter_length=None*, ***args*)

Filterbank for applying delays which are fractional multiples of the timestep

Initialised with arguments:

source Source sound or filterbank.

delays A list or array of delays to apply (the number of channels in the filterbank will be equal to the length of this).

filter_length=None Use this to explicitly set the length of the impulse response, should be odd. If not specified, it will be automatically determined from the delays. See notes below.

****args** Arguments to pass to *FIRFilterbank* (from which this class is derived).

Attributes

delay_offset
The global delay offset. If the specified delay in a given channel is `delay` the actual delay will be `delay_offset+delay`. It is equal to `(filter_length/2)/source.samplerate`.

filter_length
The length of the filter to use. This is automatically determined from the delays. Note that `delay_offset` should be larger than the maximum positive or negative delay. The minimum filter length is by default 2048 samples, which allows for good accuracy for signals with power above 20 Hz. For low frequency analysis, longer filters will be necessary. For high frequency analysis, a shorter filter length could be used for a more efficient computation.

Notes

Inducing a delay for a sound that is an integer multiple of the timestep (1/samplerate) can be done simply by offsetting the samples, e.g. `sound[3:]` is `sound` delayed by `3/sound.samplerate`. However, for fractional multiples of the timestep, the sound needs to be filtered. The theory and code for this was adapted from http://www.labbookpages.co.uk/audio/beamforming/fractionalDelay.html.

The filters induce a delay of `delay_offset+delay` where `delay_offset` is a positive value larger than the maximum positive or negative delay. This value is available as the attribute `delay_offset`.

1.10.3 Filterbank library

class brian2hears.**Gammatone**(*source*, *cf*, *b=1.019*, *erb_order=1*, *ear_Q=9.26449*, *min_bw=24.7*)
Bank of gammatone filters.

They are implemented as cascades of four 2nd-order IIR filters (this 8th-order digital filter corresponds to a 4th-order gammatone filter).

The approximated impulse response IR is defined as follow $\mathrm{IR}(t) = t^3 \exp(-2\pi b \mathrm{ERB}(f)t) \cos(2\pi f t)$ where $\mathrm{ERB}(f) = 24.7 + 0.108f$ [Hz] is the equivalent rectangular bandwidth of the filter centered at f.

It comes from Slaney's exact gammatone implementation (Slaney, M., 1993, "An Efficient Implementation of the Patterson-Holdsworth Auditory Filter Bank". Apple Computer Technical Report #35). The code is based on Slaney's Matlab implementation.

Initialised with arguments:

source Source of the filterbank.

cf List or array of center frequencies.

b=1.019 parameter which determines the bandwidth of the filters (and reciprocally the duration of its impulse response). In particular, the bandwidth = b.ERB(cf), where ERB(cf) is the equivalent bandwidth at frequency `cf`. The default value of `b` to a best fit (Patterson et al., 1992). `b` can either be a scalar and will be the same for every channel or an array of the same length as `cf`.

erb_order=1, ear_Q=9.26449, min_bw=24.7 Parameters used to compute the ERB bandwidth. $\text{ERB} = ((\text{cf}/\text{ear\_Q})^{\text{erb\_order}} + \text{min\_bw}^{\text{erb\_order}})^{(1/\text{erb\_order})}$. Their default values are the ones recommended in Glasberg and Moore, 1990.

cascade=None Specify 1 or 2 to use a cascade of 1 or 2 order 8 or 4 filters instead of 4 2nd order filters. Note that this is more efficient but may induce numerical stability issues.

class brian2hears.**ApproximateGammatone**(*source*, *cf*, *bandwidth*, *order=4*)

Bank of approximate gammatone filters implemented as a cascade of `order` IIR gammatone filters.

The filter is derived from the sampled version of the complex analog gammatone impulse response $g_\gamma(t) = t^{\gamma-1}(\lambda e^{i\eta t})^\gamma$ where γ corresponds to `order`, η defines the oscillation frequency `cf`, and λ defines the bandwidth parameter.

The design is based on the Hohmann implementation as described in Hohmann, V., 2002, "Frequency analysis and synthesis using a Gammatone filterbank", Acta Acustica United with Acustica. The code is based on the Matlab gammatone implementation from Meddis' toolbox.

Initialised with arguments:

source Source of the filterbank.

cf List or array of center frequencies.

bandwidth List or array of filters bandwidth corresponding, one for each cf.

order=4 The number of 1st-order gammatone filters put in cascade, and therefore the order the resulting gammatone filters.

class brian2hears.**LogGammachirp**(*source*, *f*, *b=1.019*, *c=1*, *ncascades=4*)

Bank of gammachirp filters with a logarithmic frequency sweep.

The approximated impulse response IR is defined as follows: $\text{IR}(t) = t^3 e^{-2\pi b \text{ERB}(f)t}\cos(2\pi(ft + c\cdot\ln(t))$ where $\text{ERB}(f) = 24.7 + 0.108f$ [Hz] is the equivalent rectangular bandwidth of the filter centered at f.

The implementation is a cascade of 4 2nd-order IIR gammatone filters followed by a cascade of ncascades 2nd-order asymmetric compensation filters as introduced in Unoki et al. 2001, "Improvement of an IIR asymmetric compensation gammachirp filter".

Initialisation parameters:

source Source sound or filterbank.

f List or array of the sweep ending frequencies ($f_{\text{instantaneous}} = f + c/t$).

b=1.019 Parameters which determine the duration of the impulse response. `b` can either be a scalar and will be the same for every channel or an array with the same length as `f`.

c=1 The glide slope (or sweep rate) given in Hz/second. The trajectory of the instantaneous frequency towards f is an upchirp when c<0 and a downchirp when c>0. `c` can either be a scalar and will be the same for every channel or an array with the same length as `f`.

ncascades=4 Number of times the asymmetric compensation filter is put in cascade. The default value comes from Unoki et al. 2001.

class brian2hears.**LinearGammachirp**(*source*, *f*, *time_constant*, *c=1*, *phase=0*)

Bank of gammachirp filters with linear frequency sweeps and gamma envelope as described in Wagner et al. 2009, "Auditory responses in the barn owl's nucleus laminaris to clicks: impulse response and signal analysis of neurophonic potential", J. Neurophysiol.

The impulse response IR is defined as follow $\text{IR}(t) = t^3 e^{-t/\sigma}\cos(2\pi(ft + c/2t^2) + \phi)$ where σ corresponds to `time_constant` and ϕ to `phase` (see definition of parameters).

Those filters are implemented as FIR filters using truncated time representations of gammachirp functions as the impulse response. The impulse responses, which need to have the same length for every channel, have a duration of 15 times the biggest time constant. The length of the impulse response is therefore `15*max(time_constant)*sampling_rate`. The impulse responses are normalized with respect to the transmitted power, i.e. the rms of the filter taps is 1.

Initialisation parameters:

source Source sound or filterbank.

f List or array of the sweep starting frequencies ($f_{\text{instantaneous}} = f + ct$).

time_constant Determines the duration of the envelope and consequently the length of the impulse response.

c=1 The glide slope (or sweep rate) given in Hz/second. The time-dependent instantaneous frequency is `f+c*t` and is therefore going upward when c>0 and downward when c<0. `c` can either be a scalar and will be the same for every channel or an array with the same length as `f`.

phase=0 Phase shift of the carrier.

Has attributes:

length_impulse_response Number of samples in the impulse responses.

impulse_response Array of shape `(nchannels, length_impulse_response)` with each row being an impulse response for the corresponding channel.

class brian2hears.**LinearGaborchirp**(*source*, *f*, *time_constant*, *c=1*, *phase=0*)

Bank of gammachirp filters with linear frequency sweeps and gaussian envelope as described in Wagner et al. 2009, "Auditory responses in the barn owl's nucleus laminaris to clicks: impulse response and signal analysis of neurophonic potential", J. Neurophysiol.

The impulse response IR is defined as follows: $\text{IR}(t) = e^{-t/2\sigma^2}\cos(2\pi(ft + c/2t^2) + \phi)$, where σ corresponds to `time_constant` and ϕ to `phase` (see definition of parameters).

These filters are implemented as FIR filters using truncated time representations of gammachirp functions as the impulse response. The impulse responses, which need to have the same length for every channel, have a duration of 12 times the biggest time constant. The length of the impulse response is therefore `12*max(time_constant)*sampling_rate`. The envelope is a gaussian function (Gabor filter). The impulse responses are normalized with respect to the transmitted power, i.e. the rms of the filter taps is 1.

Initialisation parameters:

source Source sound or filterbank.

f List or array of the sweep starting frequencies ($f_{\text{instantaneous}} = f + c * t$).

time_constant Determines the duration of the envelope and consequently the length of the impluse response.

c=1 The glide slope (or sweep rate) given ins Hz/second. The time-dependent instantaneous frequency is `f+c*t` and is therefore going upward when c>0 and downward when c<0. `c` can either be a scalar and will be the same for every channel or an array with the same length as `f`.

phase=0 Phase shift of the carrier.

Has attributes:

length_impulse_response Number of sample in the impulse responses.

impulse_response Array of shape `(nchannels, length_impulse_response)` with each row being an impulse response for the corresponding channel.

class brian2hears.**IIRFilterbank**(*source*, *nchannels*, *passband*, *stopband*, *gpass*, *gstop*, *btype*, *ftype*)

Filterbank of IIR filters. The filters can be low, high, bandstop or bandpass and be of type Elliptic, Butterworth, Chebyshev etc. The `passband` and `stopband` can be scalars (for low or high pass) or pairs of parameters (for stopband and passband) yielding similar filters for every channel. They can also be arrays of shape `(1, nchannels)` for low and high pass or `(2, nchannels)` for stopband and passband yielding different filters along channels. This class uses the scipy iirdesign function to generate filter coefficients for every channel.

See the documentation for scipy.signal.iirdesign for more details.

Initialisation parameters:

samplerate The sample rate in Hz.

nchannels The number of channels in the bank

passband, stopband The edges of the pass and stop bands in Hz. For lowpass and highpass filters, in the case of similar filters for each channel, they are scalars and passband<stopband for low pass or stopband>passband for a highpass. For a bandpass or bandstop filter, in the case of similar filters for each channel, make passband and stopband a list with two elements, e.g. for a bandpass have `passband=[200*Hz, 500*Hz]` and `stopband=[100*Hz, 600*Hz]`. `passband` and `stopband` can also be arrays of shape `(1, nchannels)` for low and high pass or `(2, nchannels)` for stopband and passband yielding different filters along channels.

gpass The maximum loss in the passband in dB. Can be a scalar or an array of length `nchannels`.

gstop The minimum attenuation in the stopband in dB. Can be a scalar or an array of length `nchannels`.

btype One of 'low', 'high', 'bandpass' or 'bandstop'.

ftype The type of IIR filter to design: 'ellip' (elliptic), 'butter' (Butterworth), 'cheby1' (Chebyshev I), 'cheby2' (Chebyshev II), 'bessel' (Bessel).

class brian2hears.**Butterworth**(*source*, *nchannels*, *order*, *fc*, *btype='low'*)

Filterbank of low, high, bandstop or bandpass Butterworth filters. The cut-off frequencies or the band frequencies can either be the same for each channel or different along channels.

Initialisation parameters:

samplerate Sample rate.

nchannels Number of filters in the bank.

order Order of the filters.

fc Cutoff parameter(s) in Hz. For the case of a lowpass or highpass filterbank, `fc` is either a scalar (thus the same value for all of the channels) or an array of length `nchannels`. For the case of a bandpass or bandstop, `fc` is either a pair of scalar defining the bandpass or bandstop (thus the same values for all of the channels) or an array of shape `(2, nchannels)` to define a pair for every channel.

btype One of 'low', 'high', 'bandpass' or 'bandstop'.

class brian2hears.**Cascade**(*source*, *filterbank*, *n*)

Cascade of `n` times a linear filterbank.

Initialised with arguments:

source Source of the new filterbank.

filterbank Filterbank object to be put in cascade

n Number of cascades

class brian2hears.**LowPass**(*source*, *fc*)

Bank of 1st-order lowpass filters

The code is based on the code found in the Meddis toolbox. It was implemented here to be used in the DRNL cochlear model implementation.

Initialised with arguments:

source Source of the filterbank.

fc Value, list or array (with length = number of channels) of cutoff frequencies.

class brian2hears.**AsymmetricCompensation**(*source*, *f*, *b=1.019*, *c=1*, *ncascades=4*)
Bank of asymmetric compensation filters.

Those filters are meant to be used in cascade with gammatone filters to approximate gammachirp filters (Unoki et al., 2001, Improvement of an IIR asymmetric compensation gammachirp filter, Acoust. Sci. & Tech.). They are implemented a a cascade of low order filters. The code is based on the implementation found in the AIM-MAT toolbox.

Initialised with arguments:

source Source of the filterbank.

f List or array of the cut off frequencies.

b=1.019 Determines the duration of the impulse response. Can either be a scalar and will be the same for every channel or an array with the same length as `cf`.

c=1 The glide slope when this filter is used to implement a gammachirp. Can either be a scalar and will be the same for every channel or an array with the same length as `cf`.

ncascades=4 The number of time the basic filter is put in cascade.

1.10.4 Auditory model library

class brian2hears.**DRNL**(*source*, *cf*, *type='human'*, *param={}*)
Implementation of the dual resonance nonlinear (DRNL) filter as described in Lopez-Paveda, E. and Meddis, R., "A human nonlinear cochlear filterbank", JASA 2001.

The entire pathway consists of the sum of a linear and a nonlinear pathway.

The linear path consists of a bank of bandpass filters (second order gammatone), a low pass function, and a gain/attenuation factor, g, in a cascade.

The nonlinear path is a cascade consisting of a bank of gammatone filters, a compression function, a second bank of gammatone filters, and a low pass function, in that order.

Initialised with arguments:

source Source of the cochlear model.

cf List or array of center frequencies.

type defines the parameters set corresponding to a certain fit. It can be either:

type='human' The parameters come from Lopez-Paveda, E. and Meddis, R.., "A human nonlinear cochlear filterbank", JASA 2001.

type ='guinea pig' The parameters come from Summer et al., "A nonlinear filter-bank model of the guinea-pig cochlear nerve: Rate responses", JASA 2003.

param Dictionary used to overwrite the default parameters given in the original papers.

The possible parameters to change and their default values for humans (see Lopez-Paveda, E. and Meddis, R.,"A human nonlinear cochlear filterbank", JASA 2001. for notation) are:

```
param['stape_scale']=0.00014
param['order_linear']=3
param['order_nonlinear']=3
```

from there on the parameters are given in the form $x = 10^{\mathrm{p0}+m\log_{10}(\mathrm{cf})}$ where `cf` is the center frequency:

```
param['cf_lin_p0']=-0.067
param['cf_lin_m']=1.016
param['bw_lin_p0']=0.037
param['bw_lin_m']=0.785
param['cf_nl_p0']=-0.052
param['cf_nl_m']=1.016
param['bw_nl_p0']=-0.031
param['bw_nl_m']=0.774
param['a_p0']=1.402
param['a_m']=0.819
param['b_p0']=1.619
param['b_m']=-0.818
param['c_p0']=-0.602
param['c_m']=0
param['g_p0']=4.2
param['g_m']=0.48
param['lp_lin_cutoff_p0']=-0.067
param['lp_lin_cutoff_m']=1.016
param['lp_nl_cutoff_p0']=-0.052
param['lp_nl_cutoff_m']=1.016
```

class brian2hears.**DCGC**(*source*, *cf*, *update_interval=1*, *param={}*)

The compressive gammachirp auditory filter as described in Irino, T. and Patterson R., "A compressive gammachirp auditory filter for both physiological and psychophysical data", JASA 2001.

Technical implementation details and notation can be found in Irino, T. and Patterson R., "A Dynamic Compressive Gammachirp Auditory Filterbank", IEEE Trans Audio Speech Lang Processing.

The model consists of a control pathway and a signal pathway in parallel.

The control pathway consists of a bank of bandpass filters followed by a bank of highpass filters (this chain yields a bank of gammachirp filters).

The signal pathway consist of a bank of fix bandpass filters followed by a bank of highpass filters with variable cutoff frequencies (this chain yields a bank of gammachirp filters with a level-dependent bandwidth). The highpass filters of the signal pathway are controlled by the output levels of the two stages of the control pathway.

Initialised with arguments:

source Source of the cochlear model.

cf List or array of center frequencies.

update_interval Interval in samples controlling how often the band pass filter of the signal pathway is updated. Smaller values are more accurate, but give longer computation times.

param Dictionary used to overwrite the default parameters given in the original paper.

The possible parameters to change and their default values (see Irino, T. and Patterson R., "A Dynamic Compressive Gammachirp Auditory Filterbank", IEEE Trans Audio Speech Lang Processing) are:

```
param['b1'] = 1.81
param['c1'] = -2.96
param['b2'] = 2.17
```

(continues on next page)

(continued from previous page)

```
param['c2'] = 2.2
param['decay_tcst'] = .5*ms
param['lev_weight'] = .5
param['level_ref'] = 50.
param['level_pwr1'] = 1.5
param['level_pwr2'] = .5
param['RMStoSPL'] = 30.
param['frat0'] = .2330
param['frat1'] = .005
param['lct_ERB'] = 1.5  #value of the shift in ERB frequencies
param['frat_control'] = 1.08
param['order_gc']=4
param['ERBrate']= 21.4*log10(4.37*cf/1000+1) # cf is the center frequency
param['ERBwidth']= 24.7*(4.37*cf/1000 + 1)
```

class brian2hears.**MiddleEar**(*source*, *gain=1*, ***kwds*)

Implements the middle ear model from Tan & Carney (2003) (linear filter with two pole pairs and one double zero). The gain is normalized for the response of the analog filter at 1000Hz as in the model of Tan & Carney (their actual C code does however result in a slightly different normalization, the difference in overall level is about 0.33dB (to get exactly the same output as in their model, set the `gain` parameter to 0.962512703689).

Tan, Q., and L. H. Carney. "A Phenomenological Model for the Responses of Auditory-nerve Fibers. II. Nonlinear Tuning with a Frequency Glide". The Journal of the Acoustical Society of America 114 (2003): 2007.

class brian2hears.**TanCarney**(*source*, *cf*, *update_interval=1*, *param=None*)

Class implementing the nonlinear auditory filterbank model as described in Tan, G. and Carney, L., "A phenomenological model for the responses of auditory-nerve fibers. II. Nonlinear tuning with a frequency glide", JASA 2003.

The model consists of a control path and a signal path. The control path controls both its own bandwidth via a feedback loop and also the bandwidth of the signal path.

Initialised with arguments:

source Source of the cochlear model.

cf List or array of center frequencies.

update_interval Interval in samples controlling how often the band pass filter of the signal pathway is updated. Smaller values are more accurate but increase the computation time.

param Dictionary used to overwrite the default parameters given in the original paper.

class brian2hears.**ZhangSynapse**(*source*, *CF*, *n_per_channel=1*, *params=None*)

A *FilterbankGroup* that represents an IHC-AN synapse according to the Zhang et al. (2001) model. The `source` should be a filterbank, producing V_ihc (e.g. *TanCarney*). `CF` specifies the characteristic frequencies of the AN fibers. `params` overwrites any parameters values given in the publication.

The group emits spikes according to a time-varying Poisson process with absolute and relative refractoriness (probability of spiking is given by state variable `R`). The continuous probability of spiking without refractoriness is available in the state variable `s`.

The `n_per_channel` argument can be used to generate multiple spike trains for every channel.

If all you need is the state variable `s`, you can use the class *ZhangSynapseRate* instead which does not simulate the spike-generating Poisson process.

For details see: Zhang, X., M. G. Heinz, I. C. Bruce, and L. H. Carney. "A Phenomenological Model for the Responses of Auditory-nerve Fibers: I. Nonlinear Tuning with Compression and Suppression". The Journal of

the Acoustical Society of America 109 (2001): 648.

class brian2hears.**ZhangSynapseRate**(*source*, *CF*, *params=None*)
A *FilterbankGroup* that represents an IHC-AN synapse according to the Zhang et al. (2001) model, see *ZhangSynapse* for details. This class does not actually generate any spikes, it only simulates the time-varying firing rate (not taking refractory effects into account) `s`.

class brian2hears.**ZhangSynapseSpikes**(*source*, *n_per_channel=1*, *params=None*)
The spike-generating Poisson process (with absolute and relative refractoriness) of an IHC-AN synapse according to the Zhang et al. (2001) model. The `source` has to have a state variable `s`, representing the firing rate (e.g. the class *ZhangSynapseRate*).

The `n_per_channel` argument can be used to generate multiple spike trains for every channel of the source group.

1.10.5 Filterbank group

class brian2hears.**FilterbankGroup**(*filterbank*, *targetvar*, **args*, ***kwds*)
Allows a Filterbank object to be used as a NeuronGroup

Initialised as a standard `NeuronGroup` object, but with two additional arguments at the beginning, and no `N` (number of neurons) argument. The number of neurons in the group will be the number of channels in the filterbank.

filterbank The Filterbank object to be used by the group. In fact, any *Bufferable* object can be used.

targetvar The target variable to put the filterbank output into.

One additional keyword is available beyond that of `NeuronGroup`:

buffersize=32 The size of the buffered segments to fetch each time. The efficiency depends on this in an unpredictable way, larger values mean more time spent in optimised code, but are worse for the cache. In many cases, the default value is a good tradeoff. Values can be given as a number of samples, or a length of time in seconds.

Note that if you specify your own `Clock`, it should have 1/dt=samplerate.

1.10.6 Functions

brian2hears.**erbspace**(*low*, *high*, *N*, *earQ=9.26449*, *minBW=24.7*, *order=1*)
Returns the centre frequencies on an ERB scale.

low, high Lower and upper frequencies

N Number of channels

earQ=9.26449, minBW=24.7, order=1 Default Glasberg and Moore parameters.

brian2hears.**asymmetric_compensation_coeffs**(*samplerate*, *fr*, *filt_b*, *filt_a*, *b*, *c*, *p0*, *p1*, *p2*, *p3*, *p4*)
This function is used to generated the coefficient of the asymmetric compensation filter used for the gammachirp implementation.

1.10.7 Plotting

brian2hears.**log_frequency_xaxis_labels**(*ax=None*, *freqs=None*)
Sets tick positions for log-scale frequency x-axis at sensible locations.

Also uses scalar representation rather than exponential (i.e. 100 rather than 10^2).

ax=None The axis to set, or uses `gca()` if `None`.

freqs=None Override the default frequency locations with your preferred tick locations.

See also: *log_frequency_yaxis_labels()*.

Note: with log scaled axes, it can be useful to call `axis('tight')` before setting the ticks.

brian2hears.**log_frequency_yaxis_labels**(*ax=None*, *freqs=None*)
Sets tick positions for log-scale frequency x-axis at sensible locations.

Also uses scalar representation rather than exponential (i.e. 100 rather than 10^2).

ax=None The axis to set, or uses `gca()` if `None`.

freqs=None Override the default frequency locations with your preferred tick locations.

See also: *log_frequency_yaxis_labels()*.

Note: with log scaled axes, it can be useful to call `axis('tight')` before setting the ticks.

1.10.8 HRTFs

class brian2hears.**HRTF**(*hrir_l*, *hrir_r=None*)
Head related transfer function.

Attributes

impulse_response The pair of impulse responses (as stereo *Sound* objects)

fir The impulse responses in a format suitable for using with *FIRFilterbank* (the transpose of `impulse_response`).

left, right The two HRTFs (mono *Sound* objects)

samplerate The sample rate of the HRTFs.

Methods

apply(*sound*)
Returns a stereo *Sound* object formed by applying the pair of HRTFs to the mono `sound` input. Equivalently, you can write `hrtf(sound)` for `hrtf` an *HRTF* object.

filterbank(*source*, ***kwds*)
Returns an *FIRFilterbank* object that can be used to apply the HRTF as part of a chain of filterbanks.

You can get the number of samples in the impulse response with `len(hrtf)`.

class brian2hears.**HRTFSet**(*data*, *samplerate*, *coordinates*)
A collection of HRTFs, typically for a single individual.

Normally this object is created automatically by an *HRTFDatabase*.

Attributes

hrtf A list of `HRTF` objects for each index.

num_indices The number of HRTF locations. You can also use `len(hrtfset)`.

num_samples The sample length of each HRTF.

fir_serial, fir_interleaved The impulse responses in a format suitable for using with *FIRFilterbank*, in serial (LLLLL...RRRRR....) or interleaved (LRLRLR...).

Methods

subset (*condition*)
Generates the subset of the set of HRTFs whose coordinates satisfy the `condition`. This should be one of: a boolean array of length the number of HRTFs in the set, with values of True/False to indicate if the corresponding HRTF should be included or not; an integer array with the indices of the HRTFs to keep; or a function whose argument names are names of the parameters of the coordinate system, e.g. `condition=lambda azim:azim<pi/2`.

filterbank (*source*, *interleaved=False*, ***kwds*)
Returns an *FIRFilterbank* object which applies all of the HRTFs in the set. If `interleaved=False` then the channels are arranged in the order LLLL...RRRR..., otherwise they are arranged in the order LRLRLR....

get_index (***kwds*)
Return the index of the HRTF with the coords specified by keyword.

You can access an HRTF by index via `hrtfset[index]`, or by its coordinates via `hrtfset(coord1=val1, coord2=val2)`.

Initialisation

data An array of shape (2, num_indices, num_samples) where data[0,:,:] is the left ear and data[1,:,:] is the right ear, num_indices is the number of HRTFs for each ear, and num_samples is the length of the HRTF.

samplerate The sample rate for the HRTFs (should have units of Hz).

coordinates A record array of length `num_indices` giving the coordinates of each HRTF. You can use *make_coordinates()* to help with this.

class brian2hears.**HRTFDatabase** (*samplerate=None*)
Base class for databases of HRTFs

Should have an attribute 'subjects' giving a list of available subjects, and a method `load_subject(subject)` which returns an `HRTFSet` for that subject.

The initialiser should take (optional) keywords:

samplerate The intended samplerate (resampling will be used if it is wrong). If left unset, the natural samplerate of the data set will be used.

brian2hears.**make_coordinates** (***kwds*)
Creates a numpy record array from the keywords passed to the function. Each keyword/value pair should be the name of the coordinate the array of values of that coordinate for each location. Returns a numpy record array. For example:

```
coords = make_coordinates(azimuth=[0, 30, 60, 0, 30, 60],
                          elevation=[0, 0, 0, 30, 30, 30])
print coords['azimuth']
```

class brian2hears.**IRCAM_LISTEN** (*basedir=None*, *compensated=False*, *samplerate=None*)
HRTFDatabase for the IRCAM LISTEN public HRTF database.

For details on the database, see the website.

The database object can be initialised with the following arguments:

basedir=None The directory where the database has been downloaded and extracted, e.g. `r'D:\HRTF\IRCAM'`. Multiple directories in a list can be provided as well (e.g IRCAM and IRCAM New). Note that if you set this to None, it will use the environment variable IRCAM_LISTEN if that has been set.

compensated=False Whether to use the raw or compensated impulse responses.

samplerate=None If specified, you can resample the impulse responses to a different samplerate, otherwise uses the default 44.1 kHz.

The coordinates are pairs `(azim, elev)` where `azim` ranges from 0 to 345 degrees in steps of 15 degrees, and elev ranges from -45 to 90 in steps of 15 degrees. After loading the database, the attribute 'subjects' gives all the subjects number that were detected as installed.

Obtaining the database

The database can be downloaded here. Each subject archive should be extracted to a folder (e.g. IRCAM) with the names of the subject, e.g. IRCAM/IRC_1002, etc.

class brian2hears.**HeadlessDatabase**(*n=None*, *azim_max=1.5707963267948966*, *diameter=0.22308*, *itd=None*, *samplerate=None*, *fractional_itds=False*)

Database for creating HRTFSet with artificial interaural time-differences

Initialisation keywords:

n, azim_max, diameter Specify the ITDs for two ears separated by distance `diameter` with no head. ITDs corresponding to `n` angles equally spaced between `-azim_max` and `azim_max` are used. The default diameter is that which gives the maximum ITD as 650 microseconds. The ITDs are computed with the formula `diameter*sin(azim)/speed_of_sound_in_air`. In this case, the generated *HRTFSet* will have coordinates of `azim` and `itd`.

itd Instead of specifying the keywords above, just give the ITDs directly. In this case, the generated *HRTFSet* will have coordinates of `itd` only.

fractional_itds=False Set this to `True` to allow ITDs with a fractional multiple of the timestep `1/samplerate`. Note that the filters used to do this are not perfect and so this will introduce a small amount of numerical error, and so shouldn't be used unless this level of timing precision is required. See *FractionalDelay* for more details.

To get the HRTFSet, the simplest thing to do is just:

```
hrtfset = HeadlessDatabase(13).load_subject()
```

The generated ITDs can be returned using the `itd` attribute of the *HeadlessDatabase* object.

If `fractional_itds=False` then Note that the delays induced in the left and right channels are not symmetric as making them so wastes half the samplerate (if the delay to the left channel is itd/2 and the delay to the right channel is -itd/2). Instead, for each channel either the left channel delay is 0 and the right channel delay is -itd (if itd<0) or the left channel delay is itd and the right channel delay is 0 (if itd>0).

If `fractional_itds=True` then delays in the left and right channels will be symmetric around a global offset of `delay_offset`.

1.10.9 Base classes

Useful for understanding more about the internals.

class brian2hears.**Bufferable**

Base class for brian2hears classes

Defines a buffering interface of two methods:

buffer_init() Initialise the buffer, should set the time pointer to zero and do any other initialisation that the object needs.

buffer_fetch(start, end) Fetch the next samples `start:end` from the buffer. Value returned should be an array of shape `(end-start, nchannels)`. Can throw an `IndexError` exception if it is outside the possible range.

In addition, bufferable objects should define attributes:

nchannels The number of channels in the buffer.

samplerate The sample rate in Hz.

By default, the class will define a default buffering mechanism which can easily be extended. To extend the default buffering mechanism, simply implement the method:

buffer_fetch_next(samples) Returns the next `samples` from the buffer.

The default methods for `buffer_init()` and `buffer_fetch()` will define a buffer cache which will get larger if it needs to to accommodate a `buffer_fetch(start, end)` where `end-start` is larger than the current cache. If the filterbank has a `minimum_buffer_size` attribute, the internal cache will always have at least this size, and the `buffer_fetch_next(samples)` method will always get called with `samples>=minimum_buffer_size`. This can be useful to ensure that the buffering is done efficiently internally, even if the user request buffered chunks that are too small. If the filterbank has a `maximum_buffer_size` attribute then `buffer_fetch_next(samples)` will always be called with `samples<=maximum_buffer_size` - this can be useful for either memory consumption reasons or for implementing time varying filters that need to update on a shorter time window than the overall buffer size.

The following attributes will automatically be maintained:

self.cached_buffer_start, self.cached_buffer_end The start and end of the cached segment of the buffer

self.cached_buffer_output An array of shape `((cached_buffer_end-cached_buffer_start, nchannels)` with the current cached segment of the buffer. Note that this array can change size.

class brian2hears.**Filterbank**(*source*)
Generalised filterbank object

Documentation common to all filterbanks

Filterbanks all share a few basic attributes:

source
The source of the filterbank, a *Bufferable* object, e.g. another *Filterbank* or a *Sound*. It can also be a tuple of sources. Can be changed after the object is created, although note that for some filterbanks this may cause problems if they do make assumptions about the input based on the first source object they were passed. If this is causing problems, you can insert a dummy filterbank (*DoNothingFilterbank*) which is guaranteed to work if you change the source.

nchannels
The number of channels.

samplerate
The sample rate.

duration
The duration of the filterbank. If it is not specified by the user, it is computed by finding the maximum of its source durations. If these are not specified a `KeyError` will be raised.

To process the output of a filterbank, the following method can be used:

process(*func=None*, *duration=None*, *buffersize=32*)
Returns the output of the filterbank for the given duration.

func If a function is specified, it should be a function of one or two arguments that will be called on each filtered buffered segment (of shape `(buffersize, nchannels)` in order. If the function has one argument, the argument should be buffered segment. If it has two arguments, the second argument is the value returned by the previous application of the function (or 0 for the first application). In this case, the method will return the final value returned by the function. See example below.

duration=None The length of time (in seconds) or number of samples to process. If no `func` is specified, the method will return an array of shape `(duration, nchannels)` with the filtered outputs. Note that in many cases, this will be too large to fit in memory, in which you will want to process the filtered outputs online, by providing a function `func` (see example below). If no duration is specified, the maximum duration of the inputs to the filterbank will be used, or an error raised if they do not have durations.

buffersize=32 The size of the buffered segments to fetch, as a length of time or number of samples. 32 samples typically gives reasonably good performance.

For example, to compute the RMS of each channel in a filterbank, you would do:

```
def sum_of_squares(input, running_sum_of_squares):
    return running_sum_of_squares+sum(input**2, axis=0)
rms = sqrt(fb.process(sum_of_squares)/nsamples)
```

Alternatively, the buffer interface can be used, which is described in more detail below.

Filterbank also defines arithmetical operations for +, -, *, / where the other operand can be a filterbank or scalar.

Details on the class

This class is a base class not designed to be instantiated. A Filterbank object should define the interface of *Bufferable*, as well as defining a `source` attribute. This is normally a *Bufferable* object, but could be an iterable of sources (for example, for filterbanks that mix or add multiple inputs).

The `buffer_fetch_next(samples)` method has a default implementation that fetches the next input, and calls the `buffer_apply(input)` method on it, which can be overridden by a derived class. This is typically the easiest way to implement a new filterbank. Filterbanks with multiple sources will need to override this default implementation.

There is a default `__init__` method that can be called by a derived class that sets the `source`, `nchannels` and `samplerate` from that of the `source` object. For multiple sources, the default implementation will check that each source has the same number of channels and samplerate and will raise an error if not.

There is a default `buffer_init()` method that calls `buffer_init()` on the `source` (or list of sources).

Example of deriving a class

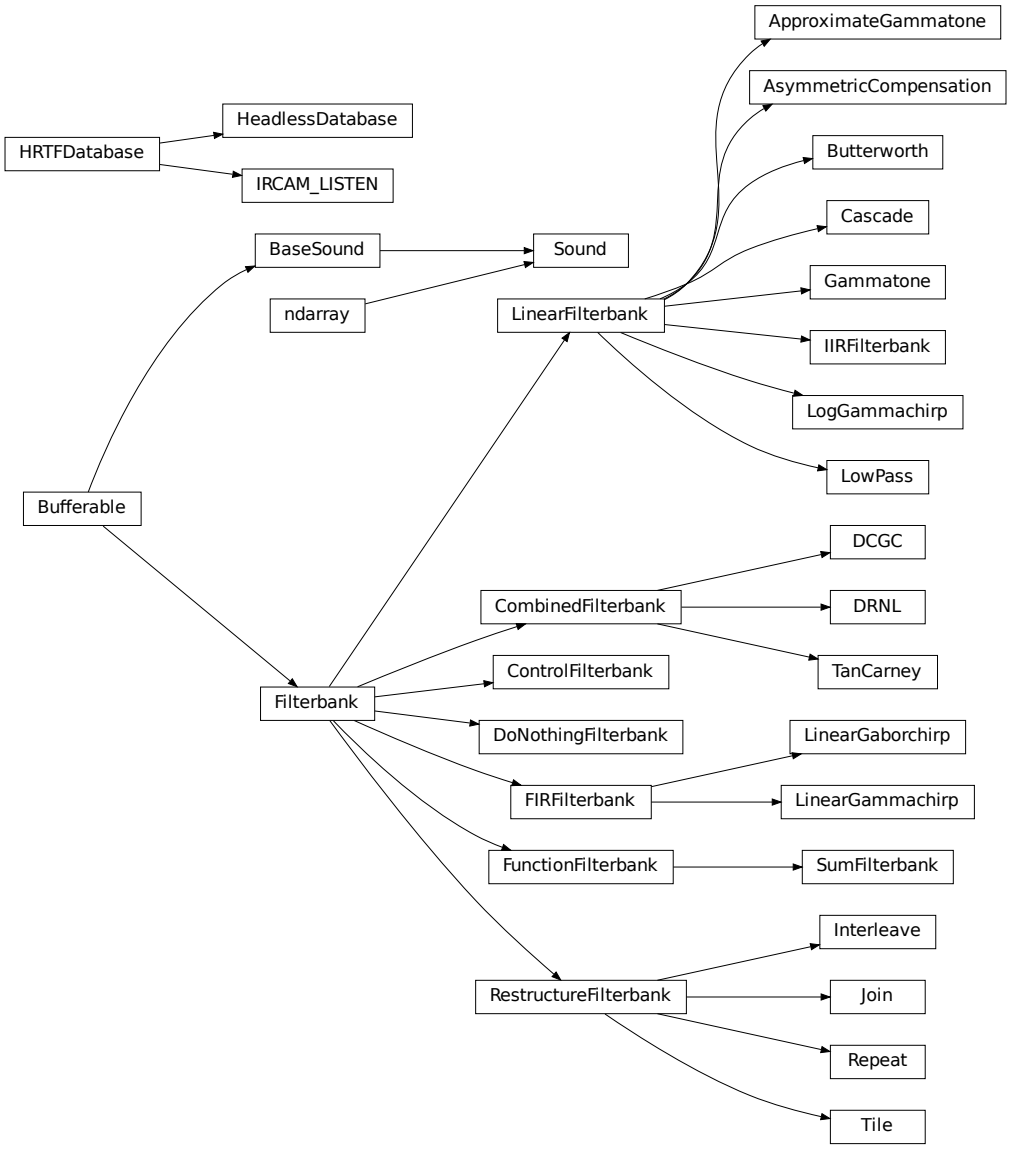
The following class takes N input channels and sums them to a single output channel:

```
class AccumulateFilterbank(Filterbank):
    def __init__(self, source):
        Filterbank.__init__(self, source)
        self.nchannels = 1
    def buffer_apply(self, input):
        return reshape(sum(input, axis=1), (input.shape[0], 1))
```

Note that the default `Filterbank.__init__` will set the number of channels equal to the number of source channels, but we want to change it to have a single output channel. We use the `buffer_apply` method which automatically handles the efficient cacheing of the buffer for us. The method receives the array `input` which has shape `(bufsize, nchannels)` and sums over the channels (`axis=1`). It's important to reshape the output so that it has shape `(bufsize, outputnchannels)` so that it can be used as the input to subsequent filterbanks.

class brian2hears.**BaseSound**
Base class for Sound and OnlineSound

1.10.10 Class diagram



1.11 Examples

1.11.1 Sounds

Example of basic use and manipulation of sounds with Brian hears.

```
from brian2 import *
from brian2hears import *

sound1 = tone(1*kHz, 1*second)
sound2 = whitenoise(1*second)

sound = sound1+sound2
sound = sound.ramp()

# Comment this line out if you don't have pygame installed
sound.play()

# The first 20ms of the sound
startsound = sound[slice(0*ms, 20*ms)]

subplot(121)
plot(startsound.times, startsound)
subplot(122)
sound.spectrogram()
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.2 Cochleagram

Example of basic filtering of a sound with Brian hears. This example implements a cochleagram based on a gammatone filterbank followed by halfwave rectification, cube root compression and 10 Hz low pass filtering.

```
from brian2 import *
from brian2hears import *

sound1 = tone(1*kHz, .1*second)
sound2 = whitenoise(.1*second)

sound = sound1+sound2
sound = sound.ramp()

cf = erbspace(20*Hz, 20*kHz, 3000)
gammatone = Gammatone(sound, cf)
cochlea = FunctionFilterbank(gammatone, lambda x: clip(x, 0, Inf)**(1.0/3.0))
lowpass = LowPass(cochlea, 10*Hz)
output = lowpass.process()

imshow(output.T, origin='lower', aspect='auto', vmin=0)
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.3 Online computation

Example of online computation using *process()*. Plots the RMS value of each channel output by a gammatone filterbank.

```
from brian2 import *
from brian2hears import *

sound1 = tone(1*kHz, .1*second)
sound2 = whitenoise(.1*second)

sound = sound1+sound2
sound = sound.ramp()

sound.level = 60*dB

cf = erbspace(20*Hz, 20*kHz, 3000)
fb = Gammatone(sound, cf)

def sum_of_squares(input, running):
    return running+sum(input**2, axis=0)

rms = sqrt(fb.process(sum_of_squares)/sound.nsamples)

sound_rms = sqrt(mean(sound**2))

axhline(sound_rms, ls='--')
plot(cf, rms)
xlabel('Frequency (Hz)')
ylabel('RMS')
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.4 Logarithmic Gammachirp filters

Example of the use of the class *LogGammachirp* available in the library. It implements a filterbank of IIR gammachirp filters as Unoki et al. 2001, "Improvement of an IIR asymmetric compensation gammachirp filter". In this example, a white noise is filtered by a linear gammachirp filterbank and the resulting cochleogram is plotted. The different impulse responses are also plotted.

```
from brian2 import *
from brian2hears import *

sound = whitenoise(100*ms).ramp()
sound.level = 50*dB

nbr_center_frequencies = 50  #number of frequency channels in the filterbank

c1 = -2.96 #glide slope
b1 = 1.81  #factor determining the time constant of the filters

#center frequencies with a spacing following an ERB scale
cf = erbspace(100*Hz, 1000*Hz, nbr_center_frequencies)

gamma_chirp = LogGammachirp(sound, cf, c=c1, b=b1)
```

(continues on next page)

(continued from previous page)

```
gamma_chirp_mon = gamma_chirp.process()

figure()
imshow(flipud(gamma_chirp_mon.T), aspect='auto')
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.5 Linear Gammachirp filters

Example of the use of the class *LinearGammachirp* available in the library. It implements a filterbank of FIR gammatone filters with linear frequency sweeps as described in Wagner et al. 2009, "Auditory responses in the barn owl's nucleus laminaris to clicks: impulse response and signal analysis of neurophonic potential", J. Neurophysiol. In this example, a white noise is filtered by a gammachirp filterbank and the resulting cochleogram is plotted. The different impulse responses are also plotted.

```
from brian2 import *
from brian2hears import *

sound = whitenoise(100*ms).ramp()
sound.level = 50*dB

nbr_center_frequencies = 10  #number of frequency channels in the filterbank
#center frequencies with a spacing following an ERB scale
center_frequencies = erbspace(100*Hz, 1000*Hz, nbr_center_frequencies)

c = 0.0 #glide slope
time_constant = linspace(3, 0.3, nbr_center_frequencies)*ms

gamma_chirp = LinearGammachirp(sound, center_frequencies, time_constant, c)

gamma_chirp_mon = gamma_chirp.process()

figure()

imshow(gamma_chirp_mon.T, aspect='auto')
figure()
plot(gamma_chirp.impulse_response.T)
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.6 Auditory nerve fibre model

Example of a simple auditory nerve fibre model with Brian hears.

```
from brian2 import *
from brian2hears import *

sound1 = tone(1*kHz, .1*second)
sound2 = whitenoise(.1*second)

sound = sound1+sound2
```

(continues on next page)

(continued from previous page)

```
sound = sound.ramp()

cf = erbspace(20*Hz, 20*kHz, 3000)
cochlea = Gammatone(sound, cf)

# Half-wave rectification and compression [x]^(1/3)
ihc = FunctionFilterbank(cochlea, lambda x: 3*clip(x, 0, Inf)**(1.0/3.0))

# Leaky integrate-and-fire model with noise and refractoriness
eqs = '''
dv/dt = (I-v)/(1*ms)+0.2*xi*(2/(1*ms))**.5 : 1 (unless refractory)
I : 1
'''
anf = FilterbankGroup(ihc, 'I', eqs, reset='v=0', threshold='v>1', refractory=5*ms,␣
↪method='euler')

M = SpikeMonitor(anf)
run(sound.duration)
plot(M.t/ms, M.i, ',k')
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.7 Gammatone filters

Example of the use of the class *Gammatone* available in the library. It implements a fitlerbank of IIR gammatone filters as described in Slaney, M., 1993, "An Efficient Implementation of the Patterson-Holdsworth Auditory Filter Bank". Apple Computer Technical Report #35. In this example, a white noise is filtered by a gammatone filterbank and the resulting cochleogram is plotted.

```
from brian2 import *
from brian2hears import *
from matplotlib import pyplot

sound = whitenoise(100*ms).ramp()
sound.level = 50*dB

nbr_center_frequencies = 50
b1 = 1.019  #factor determining the time constant of the filters
#center frequencies with a spacing following an ERB scale
center_frequencies = erbspace(100*Hz, 1000*Hz, nbr_center_frequencies)
gammatone = Gammatone(sound, center_frequencies, b=b1)

gt_mon = gammatone.process()

figure()
imshow(gt_mon.T, aspect='auto', origin='lower',
       extent=(0, sound.duration/ms,
               center_frequencies[0]/Hz, center_frequencies[-1]/Hz))
pyplot.yscale('log')
title('Cochleogram')
ylabel('Frequency (Hz)')
xlabel('Time (ms)')

show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.8 HRTFs

Example showing the use of HRTFs in Brian hears. Note that you will need to download the *IRCAM_LISTEN* database and set the IRCAM_LISTEN environment variable to point to the location where you saved it.

```
from brian2 import *
from brian2hears import *
# Load database
hrtfdb = IRCAM_LISTEN()
hrtfset = hrtfdb.load_subject(hrtfdb.subjects[0])
# Select only the horizontal plane
hrtfset = hrtfset.subset(lambda elev: elev==0)
# Set up a filterbank
sound = whitenoise(10*ms)
fb = hrtfset.filterbank(sound)
# Extract the filtered response and plot
img = fb.process().T
img_left = img[:img.shape[0]//2, :]
img_right = img[img.shape[0]//2:, :]
subplot(121)
imshow(img_left, origin='lower', aspect='auto',
       extent=(0, sound.duration/ms, 0, 360))
xlabel('Time (ms)')
ylabel('Azimuth')
title('Left ear')
subplot(122)
imshow(img_right, origin='lower', aspect='auto',
       extent=(0, sound.duration/ms, 0, 360))
xlabel('Time (ms)')
ylabel('Azimuth')
title('Right ear')
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.9 Approximate Gammatone filters

Example of the use of the class *ApproximateGammatone* available in the library. It implements a filterbank of approximate gammatone filters as described in Hohmann, V., 2002, "Frequency analysis and synthesis using a Gammatone filterbank", Acta Acustica United with Acustica. In this example, a white noise is filtered by a gammatone filterbank and the resulting cochleogram is plotted.

```
from brian2 import *
from brian2hears import whitenoise, erbspace, dB
from brian2hears.filtering.filterbanklibrary import ApproximateGammatone

level=50*dB # level of the input sound in rms dB SPL
sound = whitenoise(100*ms).ramp() # generation of a white noise
sound = sound.atlevel(level) # set the sound to a certain dB level

nbr_center_frequencies = 50  # number of frequency channels in the filterbank
# center frequencies with a spacing following an ERB scale
center_frequencies = erbspace(100*Hz, 1000*Hz, nbr_center_frequencies)
```

(continues on next page)

(continued from previous page)

```
# bandwidth of the filters (different in each channel)
bw = 10**(0.037+0.785*log10(center_frequencies/Hz))

gammatone = ApproximateGammatone(sound, center_frequencies, bw, order=3)

gt_mon = gammatone.process()

figure()
imshow(flipud(gt_mon.T), aspect='auto')
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.10 Cochlear models

Example of the use of the cochlear models (*DRNL*, *DCGC* and *TanCarney*) available in the library.

```
from brian2 import *
from brian2hears import *

simulation_duration = 50*ms
set_default_samplerate(50*kHz)
sound = whitenoise(simulation_duration)
sound = sound.atlevel(50*dB) # level in rms dB SPL
cf = erbspace(100*Hz, 1000*Hz, 50) # centre frequencies

param_drnl = {}
param_drnl['lp_nl_cutoff_m'] = 1.1

param_dcgc = {}
param_dcgc['c1'] = -2.96

figure(figsize=(10, 4))
for i, (model, param) in enumerate([(DRNL, param_drnl),
                                    (DCGC, param_dcgc),
                                    (TanCarney, None)]):
    fb = model(sound, cf, param=param)
    out = fb.process()
    subplot(1, 3, i+1)
    title(model.__name__)
    imshow(flipud(out.T), aspect='auto', extent=(0, simulation_duration/ms, 0,␣
↪len(cf)-1))
    xlabel('Time (ms)')
    if i==0:
        ylabel('CF (kHz)')
        yticks([0, len(cf)-1], [cf[0]/kHz, cf[-1]/kHz])
    else:
        yticks([])

tight_layout()
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.11 Butterworth filters

Example of the use of the class *Butterworth* available in the library. In this example, a white noise is filtered by a bank of butterworth bandpass filters and lowpass filters which are different for every channels. The centre or cutoff frequency of the filters are linearly taken between 100kHz and 1000kHz and its bandwidth frequency increases linearly with frequency.

```
from brian2 import *
from brian2hears import *

level = 50*dB  # level of the input sound in rms dB SPL
sound = whitenoise(100*ms).ramp()
sound = sound.atlevel(level)
order = 2 #order of the filters

#### example of a bank of bandpass filter ################
nchannels = 50
center_frequencies = linspace(100*Hz, 1000*Hz, nchannels)
bw = linspace(50*Hz, 300*Hz, nchannels) # bandwidth of the filters
#arrays of shape (2 x nchannels) defining the passband frequencies (Hz)
fc = vstack((center_frequencies-bw/2, center_frequencies+bw/2))

filterbank = Butterworth(sound, nchannels, order, fc, 'bandpass')

filterbank_mon = filterbank.process()

figure()
subplot(211)
imshow(flipud(filterbank_mon.T), aspect='auto')

### example of a bank of lowpass filter ################
nchannels = 50
cutoff_frequencies = linspace(200*Hz, 1000*Hz, nchannels)

filterbank = Butterworth(sound, nchannels, order, cutoff_frequencies, 'low')

filterbank_mon = filterbank.process()

subplot(212)
imshow(flipud(filterbank_mon.T), aspect='auto')
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.12 Artificial Vowels

This example implements the artificial vowels from Culling, J. F. and Summerfield, Q. (1995a). “Perceptual segregation of concurrent speech sounds: absence of across-frequency grouping by common interaural delay” J. Acoust. Soc. Am. 98, 785-797.

```
from brian2 import *
from brian2hears import *

duration = 409.6*ms
width = 150*Hz/2
samplerate = 10*kHz
```

(continues on next page)

(continued from previous page)

```
set_default_samplerate(samplerate)

centres = [225*Hz, 625*Hz, 975*Hz, 1925*Hz]
vowels = {
    'ee':[centres[0], centres[3]],
    'ar':[centres[1], centres[2]],
    'oo':[centres[0], centres[2]],
    'er':[centres[1], centres[3]]
    }

def generate_vowel(vowel):
    vowel = vowels[vowel]
    x = whitenoise(duration)
    y = fft(asarray(x).flatten())
    f = fftfreq(len(x), 1/samplerate)
    I = zeros(len(f), dtype=bool)
    for cf in vowel:
        I = I|((abs(f)<cf+width)&(abs(f)>cf-width))
    I = ~I
    y[I] = 0
    x = ifft(y)
    return Sound(x.real)

v1 = generate_vowel('ee').ramp()
v2 = generate_vowel('ar').ramp()
v3 = generate_vowel('oo').ramp()
v4 = generate_vowel('er').ramp()

for s in [v1, v2, v3, v4]:
    s.play(normalise=True, sleep=True)

s1 = Sound((v1, v2))
#s1.play(normalise=True, sleep=True)

s2 = Sound((v3, v4))
#s2.play(normalise=True, sleep=True)

v1.save('mono_sound.wav')
s1.save('stereo_sound.wav')

subplot(211)
plot(v1.times, v1)
subplot(212)
v1.spectrogram()
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.13 IIR filterbank

Example of the use of the class *IIRFilterbank* available in the library. In this example, a white noise is filtered by a bank of chebyshev bandpass filters and lowpass filters which are different for every channels. The centre frequencies of the filters are linearly taken between 100kHz and 1000kHz and its bandwidth or cutoff frequency increases linearly with frequency.

```
from brian2 import *
from brian2hears import *

sound = whitenoise(100*ms).ramp()
sound.level = 50*dB

### example of a bank of bandpass filter ################
nchannels = 50
center_frequencies = linspace(200*Hz, 1000*Hz, nchannels)  #center frequencies
bw = linspace(50*Hz, 300*Hz, nchannels)  #bandwidth of the filters
# The maximum loss in the passband in dB. Can be a scalar or an array of length
# nchannels
gpass = 1.*dB
# The minimum attenuation in the stopband in dB. Can be a scalar or an array
# of length nchannels
gstop = 10.*dB
#arrays of shape (2 x nchannels) defining the passband frequencies (Hz)
passband = vstack((center_frequencies-bw/2, center_frequencies+bw/2))
#arrays of shape (2 x nchannels) defining the stopband frequencies (Hz)
stopband = vstack((center_frequencies-1.1*bw, center_frequencies+1.1*bw))

filterbank = IIRFilterbank(sound, nchannels, passband, stopband, gpass, gstop,
                           'bandstop', 'cheby1')
filterbank_mon = filterbank.process()

figure()
subplot(211)
imshow(flipud(filterbank_mon.T), aspect='auto')

#### example of a bank of lowpass filter ################
nchannels = 50
cutoff_frequencies = linspace(100*Hz, 1000*Hz, nchannels)
#bandwidth of the transition region between the en of the pass band and the
#begin of the stop band
width_transition = linspace(50*Hz, 300*Hz, nchannels)
# The maximum loss in the passband in dB. Can be a scalar or an array of length
# nchannels
gpass = 1*dB
# The minimum attenuation in the stopband in dB. Can be a scalar or an array of
# length nchannels
gstop = 10*dB
passband = cutoff_frequencies-width_transition/2
stopband = cutoff_frequencies+width_transition/2

filterbank = IIRFilterbank(sound, nchannels, passband, stopband, gpass, gstop,
                           'low','cheby1')
filterbank_mon=filterbank.process()

subplot(212)
imshow(flipud(filterbank_mon.T), aspect='auto')
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.14 Dual resonance nonlinear filter (DRNL)

Implementation example of the dual resonance nonlinear (DRNL) filter with parameters fitted for human as described in Lopez-Paveda, E. and Meddis, R., A human nonlinear cochlear filterbank, JASA 2001.

A class called *DRNL* implementing this model is available in the library.

The entire pathway consists of the sum of a linear and a nonlinear pathway.

The linear path consists of a bank of bandpass filters (second order gammatone), a low pass function, and a gain/attenuation factor, g, in a cascade.

The nonlinear path is a cascade consisting of a bank of gammatone filters, a compression function, a second bank of gammatone filters, and a low pass function, in that order.

The parameters are given in the form `10**(p0+mlog10(cf))`.

```
from brian2 import *
from brian2hears import *

simulation_duration = 50*ms
samplerate = 50*kHz
level = 50*dB  # level of the input sound in rms dB SPL
sound = whitenoise(simulation_duration, samplerate).ramp()
sound.level = level

nbr_cf = 50  #number of centre frequencies
#center frequencies with a spacing following an ERB scale
center_frequencies = erbspace(100*Hz,1000*Hz, nbr_cf)
center_frequencies = asarray(center_frequencies) # avoid units issues

#conversion to stape velocity (which are the units needed by the following centres)
sound = sound*0.00014

#### Linear Pathway ####

#bandpass filter (second order gammatone filter)
center_frequencies_linear = 10**(-0.067+1.016*log10(center_frequencies))
bandwidth_linear = 10**(0.037+0.785*log10(center_frequencies))
order_linear = 3
gammatone = ApproximateGammatone(sound, center_frequencies_linear,
                                 bandwidth_linear, order=order_linear)

#linear gain
g = 10**(4.2-0.48*log10(center_frequencies))
func_gain = lambda x:g*x
gain = FunctionFilterbank(gammatone, func_gain)

#low pass filter(cascade of 4 second order lowpass butterworth filters)
cutoff_frequencies_linear = center_frequencies_linear
order_lowpass_linear = 2
lp_l = LowPass(gain, cutoff_frequencies_linear)
lowpass_linear = Cascade(gain, lp_l, 4)

#### Nonlinear Pathway ####

#bandpass filter (third order gammatone filters)
center_frequencies_nonlinear = center_frequencies
bandwidth_nonlinear = 10**(-0.031+0.774*log10(center_frequencies))
```

(continues on next page)

(continued from previous page)

```
order_nonlinear = 3
bandpass_nonlinear1 = ApproximateGammatone(sound, center_frequencies_nonlinear,
                                           bandwidth_nonlinear,
                                           order=order_nonlinear)

#compression (linear at low level, compress at high level)
a = 10**(1.402+0.819*log10(center_frequencies))  #linear gain
b = 10**(1.619-0.818*log10(center_frequencies))
v = .2 #compression exponent
func_compression = lambda x: sign(x)*minimum(a*abs(x), b*abs(x)**v)
compression = FunctionFilterbank(bandpass_nonlinear1, func_compression)

#bandpass filter (third order gammatone filters)
bandpass_nonlinear2 = ApproximateGammatone(compression,
                                           center_frequencies_nonlinear,
                                           bandwidth_nonlinear,
                                           order=order_nonlinear)

#low pass filter
cutoff_frequencies_nonlinear = center_frequencies_nonlinear
order_lowpass_nonlinear = 2
lp_nl = LowPass(bandpass_nonlinear2, cutoff_frequencies_nonlinear)
lowpass_nonlinear = Cascade(bandpass_nonlinear2, lp_nl, 3)

#adding the two pathways
dnrl_filter = lowpass_linear+lowpass_nonlinear

dnrl = dnrl_filter.process()

figure()
imshow(flipud(dnrl.T), aspect='auto')
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.15 Time varying filter (1)

This example implements a band pass filter whose center frequency is modulated by an Ornstein-Uhlenbeck. The white noise term used for this process is output by a FunctionFilterbank. The bandpass filter coefficients update is an example of how to use a *ControlFilterbank*. The bandpass filter is a basic biquadratic filter for which the Q factor and the center frequency must be given. The input is a white noise.

```
from brian2 import *
from brian2hears import *

samplerate = 20*kHz
SoundDuration = 300*ms
sound = whitenoise(SoundDuration, samplerate).ramp()

#number of frequency channel (here it must be one as a spectrogram of the
#output is plotted)
nchannels = 1

fc_init = 5000*Hz   #initial center frequency of the band pass filter
Q = 5               #quality factor of the band pass filter
```

(continues on next page)

(continued from previous page)

```
update_interval = 4 # the filter coefficients are updated every 4 samples

#parameters of the Ornstein-Uhlenbeck process
s_i = 1200*Hz
tau_i = 100*ms
mu_i = fc_init/tau_i
sigma_i = sqrt(2)*s_i/sqrt(tau_i)
deltaT = defaultclock.dt

#this function  is used in a FunctionFilterbank. It outputs a noise term that
#will be later used by the controler to update the center frequency
noise = lambda x: mu_i*deltaT+sigma_i*randn(1)*sqrt(deltaT)
noise_generator = FunctionFilterbank(sound, noise)

#this class will take as input the output of the noise generator and as target
#the bandpass filter center frequency
class CoeffController(object):
    def __init__(self, target):
        self.target = target
        self.deltaT = 1./samplerate
        self.BW = 2*arcsinh(1./2/Q)*1.44269
        self.fc = fc_init

    def __call__(self, input):
        #the control variables are taken as the last of the buffer
        noise_term = input[-1,:]
        #update the center frequency by updateing the OU process
        self.fc = asarray(self.fc-self.fc/tau_i*self.deltaT)+noise_term

        w0 = 2*pi*self.fc/float(samplerate)
        #update the coefficient of the biquadratic filterbank
        alpha = sin(w0)*sinh(log(2)/2*self.BW*w0/sin(w0))
        self.target.filt_b[:, 0, 0] = sin(w0)/2
        self.target.filt_b[:, 1, 0] = 0
        self.target.filt_b[:, 2, 0] = -sin(w0)/2

        self.target.filt_a[:, 0, 0] = 1+alpha
        self.target.filt_a[:, 1, 0] = -2*cos(w0)
        self.target.filt_a[:, 2, 0] = 1-alpha

# In the present example the time varying filter is a LinearFilterbank therefore
#we must initialise the filter coefficients; the one used for the first buffer␣
↪computation
w0 = 2*pi*fc_init/samplerate
BW = 2*arcsinh(1./2/Q)*1.44269
alpha = sin(w0)*sinh(log(2)/2*BW*w0/sin(w0))

filt_b = zeros((nchannels, 3, 1))
filt_a = zeros((nchannels, 3, 1))
filt_b[:, 0, 0] = sin(w0)/2
filt_b[:, 1, 0] = 0
filt_b[:, 2, 0] = -sin(w0)/2
filt_a[:, 0, 0] = 1+alpha
filt_a[:, 1, 0] = -2*cos(w0)
filt_a[:, 2, 0] = 1-alpha

#the filter which will have time varying coefficients
```

(continues on next page)

(continued from previous page)

```
bandpass_filter = LinearFilterbank(sound, filt_b, filt_a)
#the updater
updater = CoeffController(bandpass_filter)

#the controller. Remember it must be the last of the chain
control = ControlFilterbank(bandpass_filter, noise_generator, bandpass_filter,
                            updater, update_interval)

time_varying_filter_mon = control.process()

figure(1)
pxx, freqs, bins, im = specgram(squeeze(time_varying_filter_mon),
                                NFFT=256, Fs=float(samplerate), noverlap=240)
imshow(flipud(pxx), aspect='auto')

show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.16 Time varying filter (2)

This example implements a band pass filter whose center frequency is modulated by a sinusoid function. This modulator is implemented as a *FunctionFilterbank*. One state variable (here time) must be kept; it is therefore implemented with a class. The bandpass filter coefficients update is an example of how to use a *ControlFilterbank*. The bandpass filter is a basic biquadratic filter for which the Q factor and the center frequency must be given. The input is a white noise.

```
from brian2 import *
from brian2hears import *


samplerate = 20*kHz
SoundDuration = 300*ms
sound = whitenoise(SoundDuration, samplerate).ramp()

#number of frequency channel (here it must be one as a spectrogram of the
#output is plotted)
nchannels = 1

fc_init = 5000*Hz   #initial center frequency of the band pass filter
Q = 5               #quality factor of the band pass filter
update_interval = 1 # the filter coefficients are updated every sample

mean_center_freq = 4*kHz #mean frequency around which the CF will oscillate
amplitude = 1500*Hz      #amplitude of the oscillation
frequency = 10*Hz        #frequency of the oscillation

#this class is used in a FunctionFilterbank (via its __call__). It outputs the
#center frequency of the band pass filter. Its output is thus later passed as
#input to the controler.
class CenterFrequencyGenerator(object):
    def __init__(self):
        self.t=0*second

    def __call__(self, input):
```

(continues on next page)

(continued from previous page)

```
        #update of the center frequency
        fc = mean_center_freq+amplitude*sin(2*pi*frequency*self.t)
        #update of the state variable
        self.t = self.t+1./samplerate
        return fc

center_frequency = CenterFrequencyGenerator()

fc_generator = FunctionFilterbank(sound, center_frequency)

#the updater of the controller generates new filter coefficient of the band pass
#filter based on the center frequency it receives from the fc_generator
#(its input)
class CoeffController(object):
    def __init__(self, target):
        self.BW = 2*arcsinh(1./2/Q)*1.44269
        self.target=target

    def __call__(self, input):
        fc = input[-1,:] #the control variables are taken as the last of the buffer
        w0 = 2*pi*fc/array(samplerate)
        alpha = sin(w0)*sinh(log(2)/2*self.BW*w0/sin(w0))

        self.target.filt_b[:, 0, 0] = sin(w0)/2
        self.target.filt_b[:, 1, 0] = 0
        self.target.filt_b[:, 2, 0] = -sin(w0)/2

        self.target.filt_a[:, 0, 0] = 1+alpha
        self.target.filt_a[:, 1, 0] = -2*cos(w0)
        self.target.filt_a[:, 2, 0] = 1-alpha

# In the present example the time varying filter is a LinearFilterbank therefore
#we must initialise the filter coefficients; the one used for the first buffer␣
↪computation
w0 = 2*pi*fc_init/samplerate
BW = 2*arcsinh(1./2/Q)*1.44269
alpha = sin(w0)*sinh(log(2)/2*BW*w0/sin(w0))

filt_b = zeros((nchannels, 3, 1))
filt_a = zeros((nchannels, 3, 1))

filt_b[:, 0, 0] = sin(w0)/2
filt_b[:, 1, 0] = 0
filt_b[:, 2, 0] = -sin(w0)/2

filt_a[:, 0, 0] = 1+alpha
filt_a[:, 1, 0] = -2*cos(w0)
filt_a[:, 2, 0] = 1-alpha

#the filter which will have time varying coefficients
bandpass_filter = LinearFilterbank(sound, filt_b, filt_a)
#the updater
updater = CoeffController(bandpass_filter)

#the controller. Remember it must be the last of the chain
control = ControlFilterbank(bandpass_filter, fc_generator, bandpass_filter,
                            updater, update_interval)
```

(continues on next page)

(continued from previous page)

```
time_varying_filter_mon = control.process()

figure(1)
pxx, freqs, bins, im = specgram(squeeze(time_varying_filter_mon),
                                NFFT=256, Fs=float(samplerate), noverlap=240)
imshow(flipud(pxx), aspect='auto')

show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.17 Sound localisation model

Example demonstrating the use of many features of Brian hears, including HRTFs, restructuring filters and integration with Brian. Implements a simplified version of the "ideal" sound localisation model from Goodman and Brette (2010).

The sound is played at a particular spatial location (indicated on the final plot by a red +). Each location has a corresponding assembly of neurons, whose summed firing rates give the sizes of the blue circles in the plot. The most strongly responding assembly is indicated by the green x, which is the estimate of the location by the model.

Note that you will need to download the *IRCAM_LISTEN* database and set the `IRCAM_LISTEN` environment variable to point to the location where you saved it.

Reference:

Goodman DFM, Brette R (2010). Spike-timing-based computation in sound localization. PLoS Comput. Biol. 6(11).

```
from brian2 import *
from brian2hears import *

# Download the IRCAM database, and replace this filename with the location
# you downloaded it to
hrtfdb = IRCAM_LISTEN()
subject = hrtfdb.subjects[0]
hrtfset = hrtfdb.load_subject(subject)
# This gives the number of spatial locations in the set of HRTFs
num_indices = hrtfset.num_indices
# Choose a random location for the sound to come from
index = randint(hrtfset.num_indices)
# A sound to test the model with
sound = Sound.whitenoise(500*ms)
# This is the specific HRTF for the chosen location
hrtf = hrtfset.hrtf[index]
# We apply the chosen HRTF to the sound, the output has 2 channels
hrtf_fb = hrtf.filterbank(sound)
# We swap these channels (equivalent to swapping the channels in the
# subsequent filters, but simpler to do it with the inputs)
swapped_channels = RestructureFilterbank(hrtf_fb, indexmapping=[1, 0])
# Now we apply all of the possible pairs of HRTFs in the set to these
# swapped channels, which means repeating them num_indices times first
hrtfset_fb = hrtfset.filterbank(Repeat(swapped_channels, num_indices))
# Now we apply cochlear filtering (logically, this comes before the HRTF
# filtering, but since convolution is commutative it is more efficient to
# do the cochlear filtering afterwards
cfmin, cfmax, cfN = 150*Hz, 5*kHz, 40
```

(continues on next page)

(continued from previous page)

```
cf = erbspace(cfmin, cfmax, cfN)
# We repeat each of the HRTFSet filterbank channels cfN times, so that
# for each location we will apply each possible cochlear frequency
gfb = Gammatone(Repeat(hrtfset_fb, cfN),
                tile(cf, hrtfset_fb.nchannels))
# Half wave rectification and compression
cochlea = FunctionFilterbank(gfb, lambda x:15*clip(x, 0, Inf)**(1.0/3.0))
# Leaky integrate and fire neuron model
eqs = '''
dV/dt = (I-V)/(1*ms)+0.1*xi/(0.5*ms)**.5 : 1 (unless refractory)
I : 1
'''
G = FilterbankGroup(cochlea, 'I', eqs, reset='V=0', threshold='V>1', refractory=5*ms,␣
↪method='Euler')
# The coincidence detector (cd) neurons
cd = NeuronGroup(num_indices*cfN, eqs, reset='V=0', threshold='V>1', refractory=1*ms,␣
↪method='Euler', dt=G.dt[:])
# Each CD neuron receives precisely two inputs, one from the left ear and
# one from the right, for each location and each cochlear frequency
C = Synapses(G, cd, on_pre='V += 0.5', dt=G.dt[:])
C.connect(j='i', skip_if_invalid=True)
C.connect(j='i-num_indices*cfN', skip_if_invalid=True)
# We want to just count the number of CD spikes
counter = SpikeMonitor(cd, record=False)
# Run the simulation, giving a report on how long it will take as we run
run(sound.duration, report='stderr')
# We take the array of counts, and reshape them into a 2D array which we sum
# across frequencies to get the spike count of each location-specific assembly
count = counter.count[:].copy()
count.shape = (num_indices, cfN)
count = sum(count, axis=1)
count = array(count, dtype=float)/amax(count)
# Our guess of the location is the index of the strongest firing assembly
index_guess = argmax(count)
# Now we plot the output, using the coordinates of the HRTFSet
coords = hrtfset.coordinates
azim, elev = coords['azim'], coords['elev']
scatter(azim, elev, 100*count)
plot([azim[index]], [elev[index]], '+r', ms=15, mew=2)
plot([azim[index_guess]], [elev[index_guess]], 'xg', ms=15, mew=2)
xlabel('Azimuth (deg)')
ylabel('Elevation (deg)')
xlim(-5, 350)
ylim(-50, 95)
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.18 Compressive Gammachirp filter (DCGC)

Implementation example of the compressive gammachirp auditory filter as described in Irino, T. and Patterson R., "A compressive gammachirp auditory filter for both physiological and psychophysical data", JASA 2001.

A class called *DCGC* implementing this model is available in the library.

Technical implementation details and notation can be found in Irino, T. and Patterson R., "A Dynamic Compressive Gammachirp Auditory Filterbank", IEEE Trans Audio Speech Lang Processing.

```
from brian2 import *
from brian2hears import *

simulation_duration = 50*ms
samplerate = 50*kHz
level = 50*dB # level of the input sound in rms dB SPL
sound = whitenoise(simulation_duration, samplerate).ramp()
sound = sound.atlevel(level)

nbr_cf = 50 # number of centre frequencies
# center frequencies with a spacing following an ERB scale
cf = erbspace(100*Hz, 1000*Hz, nbr_cf)

c1 = -2.96 #glide slope of the first filterbank
b1 = 1.81  #factor determining the time constant of the first filterbank
c2 = 2.2   #glide slope of the second filterbank
b2 = 2.17  #factor determining the time constant of the second filterbank

order_ERB = 4
ERBrate = 21.4*log10(4.37*(cf/kHz)+1)
ERBwidth = 24.7*(4.37*(cf/kHz) + 1)
ERBspace = mean(diff(ERBrate))

# the filter coefficients are updated every update_interval (here in samples)
update_interval = 1



#bank of passive gammachirp filters. As the control path uses the same passive
#filterbank than the signal path (but shifted in frequency)
#this filterbank is used by both pathway.
pGc = LogGammachirp(sound, cf, b=b1, c=c1)

fp1 = asarray(cf) + c1*ERBwidth*b1/order_ERB #centre frequency of the signal path

#### Control Path ####

#the first filterbank in the control path consists of gammachirp filters
#value of the shift in ERB frequencies of the control path with respect to the signal␣
↪path
lct_ERB = 1.5
n_ch_shift = round(lct_ERB/ERBspace) #value of the shift in channels
#index of the channel of the control path taken from pGc
indch1_control = minimum(maximum(1, arange(1, nbr_cf+1)+n_ch_shift), nbr_cf).
↪astype(int)-1
fp1_control = fp1[indch1_control]
#the control path bank pass filter uses the channels of pGc indexed by indch1_control
pGc_control = RestructureFilterbank(pGc, indexmapping=indch1_control)

#the second filterbank in the control path consists of fixed asymmetric compensation␣
↪filters
frat_control = 1.08
fr2_control = frat_control*fp1_control
asym_comp_control = AsymmetricCompensation(pGc_control, fr2_control, b=b2, c=c2)

#definition of the pole of the asymmetric comensation filters
p0 = 2
```

(continues on next page)

(continued from previous page)

```
p1 = 1.7818*(1-0.0791*b2)*(1-0.1655*abs(c2))
p2 = 0.5689*(1-0.1620*b2)*(1-0.0857*abs(c2))
p3 = 0.2523*(1-0.0244*b2)*(1+0.0574*abs(c2))
p4 = 1.0724

#definition of the parameters used in the control path output levels computation
#(see IEEE paper for details)
decay_tcst = .5*ms
order = 1.
lev_weight = .5
level_ref = 50.
level_pwr1 = 1.5
level_pwr2 = .5
RMStoSPL = 30.
frat0 = .2330
frat1 = .005
exp_deca_val = exp(-1/(decay_tcst*samplerate)*log(2))
level_min = 10**(-RMStoSPL/20)

#definition of the controller class. What is does it take the outputs of the
#first and second fitlerbanks of the control filter as input, compute an overall
#intensity level for each frequency channel. It then uses those level to update
#the filter coefficient of its target, the asymmetric compensation filterbank of
#the signal path.
class CompensensationFilterUpdater(object):
    def __init__(self, target):
        self.target = target
        self.level1_prev = -100
        self.level2_prev = -100

    def __call__(self, *input):
         value1 = input[0][-1,:]
         value2 = input[1][-1,:]
         #the current level value is chosen as the max between the current
         #output and the previous one decreased by a decay
         level1 = maximum(maximum(value1, 0), self.level1_prev*exp_deca_val)
         level2 = maximum(maximum(value2, 0), self.level2_prev*exp_deca_val)

         self.level1_prev = level1 #the value is stored for the next iteration
         self.level2_prev = level2
         #the overall intensity is computed between the two filterbank outputs
         level_total = lev_weight*level_ref*(level1/level_ref)**level_pwr1+\
                   (1-lev_weight)*level_ref*(level2/level_ref)**level_pwr2
         #then it is converted in dB
         level_dB = 20*log10(maximum(level_total, level_min))+RMStoSPL
         #the frequency factor is calculated
         frat = frat0 + frat1*level_dB
         #the centre frequency of the asymmetric compensation filters are updated
         fr2 = fp1*frat
         coeffs = asymmetric_compensation_coeffs(samplerate, fr2,
                        self.target.filt_b, self.target.filt_a, b2, c2,
                        p0, p1, p2, p3, p4)
         self.target.filt_b, self.target.filt_a = coeffs

#### Signal Path ####
#the signal path consists of the passive gammachirp filterbank pGc previously
#defined followed by a asymmetric compensation filterbank
```

(continues on next page)

(continued from previous page)

```
fr1 = fp1*frat0
varyingfilter_signal_path = AsymmetricCompensation(pGc, fr1, b=b2, c=c2)
updater = CompensensationFilterUpdater(varyingfilter_signal_path)
 #the controler which takes the two filterbanks of the control path as inputs
 #and the varying filter of the signal path as target is instantiated
control = ControlFilterbank(varyingfilter_signal_path,
                            [pGc_control, asym_comp_control],
                            varyingfilter_signal_path, updater, update_interval)

#run the simulation
#Remember that the controler are at the end of the chain and the output of the
#whole path comes from them
signal = control.process()

figure()
imshow(flipud(signal.T), aspect='auto')
show()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.11.19 Tan&Carney (2003)

Tan, Q., and L. H. Carney. "A Phenomenological Model for the Responses of Auditory-nerve Fibers. II. Nonlinear Tuning with a Frequency Glide". The Journal of the Acoustical Society of America 114 (2003): 2007.

Response area and phase response in the Tan&Carney model

Response area and phase response of a model fiber with CF=2200Hz in the Tan&Carney model. Reproduces Fig. 11 from:

Tan, Q., and L. H. Carney. "A Phenomenological Model for the Responses of Auditory-nerve Fibers. II. Nonlinear Tuning with a Frequency Glide". The Journal of the Acoustical Society of America 114 (2003): 2007.

```
import matplotlib.pyplot as plt
import numpy as np

from brian2 import *
from brian2hears import *
from six.moves import range as xrange

def product(*args):
    # Simple (and inefficient) variant of itertools.product that works for
    # Python 2.5 (directly returns a list instead of yielding one item at a
    # time)
    pools = map(tuple, args)
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    return result

duration = 50*ms
samplerate = 50*kHz
set_default_samplerate(samplerate)
CF = 2200
```

(continues on next page)

(continued from previous page)

```
freqs = np.arange(250.0, 3501., 50.)*Hz
levels = [10, 30, 50, 70, 90]
cf_level = product(freqs, levels)
tones = Sound([Sound.sequence([tone(freq, duration).atlevel(level*dB).ramp(when='both
↪',
                                                                                    ␣
↪duration=2.5*ms,
                                                                                    ␣
↪inplace=False)])
               for freq, level in cf_level])

ihc = TanCarney(MiddleEar(tones), [CF] * len(cf_level), update_interval=2)
syn = ZhangSynapse(ihc, CF)
s_mon = StateMonitor(syn, 's', record=True, clock=syn.clock)
net = Network(syn, s_mon)
net.run(duration)

reshaped = s_mon.s[:].reshape((len(freqs), len(levels), -1))

# calculate the phase with respect to the stimulus
pi = np.pi
min_freq, max_freq = 1100*Hz, 2900*Hz
freq_subset = freqs[(freqs>=min_freq) & (freqs<=max_freq)]
reshaped_subset = reshaped[(freqs>=min_freq) & (freqs<=max_freq), :, :]
phases = np.zeros((reshaped_subset.shape[0], len(levels)))
for f_idx, freq in enumerate(freq_subset):
    period = 1.0 / freq
    for l_idx in xrange(len(levels)):
        phase_angles = np.arange(reshaped_subset.shape[2])/samplerate % period /␣
↪period * 2*pi
        temp_phases = (np.exp(1j * phase_angles) *
                       reshaped_subset[f_idx, l_idx, :]/Hz)
        phases[f_idx, l_idx] = np.angle(np.sum(temp_phases))

plt.subplot(2, 1, 1)
rate = reshaped.mean(axis=2)
plt.plot(freqs, rate)
plt.ylabel('Spikes/sec')
plt.legend(['%.0f dB' % level for level in levels], loc='best')
plt.xlim(0, 4000)
plt.ylim(0, 250)

plt.subplot(2, 1, 2)
relative_phases = (phases.T - phases[:, -1]).T
relative_phases[relative_phases > pi] = relative_phases[relative_phases > pi] - 2*pi
relative_phases[relative_phases < -pi] = relative_phases[relative_phases < -pi] + 2*pi
plt.plot(freq_subset, relative_phases / pi)
plt.ylabel("Phase Re:90dB (pi radians)")
plt.xlabel('Frequency (Hz)')
plt.legend(['%.0f dB' % level for level in levels], loc='best')
plt.xlim(0, 4000)
plt.ylim(-0.5, 0.75)
plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

Spiking output of the Tan&Carney model

Fig. 1 and 3 (spking output without spiking/refractory period) should reproduce the output of the AN3_test_tone.m and AN3_test_click.m scripts, available in the code accompanying the paper Tan & Carney (2003). This matlab code is available from http://www.urmc.rochester.edu/labs/Carney-Lab/publications/auditory-models.cfm

Tan, Q., and L. H. Carney. "A Phenomenological Model for the Responses of Auditory-nerve Fibers. II. Nonlinear Tuning with a Frequency Glide". The Journal of the Acoustical Society of America 114 (2003): 2007.

```
import numpy as np
import matplotlib.pyplot as plt

from brian2 import *
from brian2hears import (Sound, get_samplerate, set_default_samplerate, tone,
                         click, silence, dB, TanCarney, MiddleEar, ZhangSynapse)

set_default_samplerate(50*kHz)
sample_length = 1 / get_samplerate(None)
cf = 1000 * Hz

print('Testing click response')
duration = 25*ms
levels = [40, 60, 80, 100, 120]
# a click of two samples
tones = Sound([Sound.sequence([click(sample_length*2, peak=level*dB),
                               silence(duration=duration - sample_length)])
           for level in levels])
ihc = TanCarney(MiddleEar(tones), [cf] * len(levels), update_interval=1)
syn = ZhangSynapse(ihc, cf)
mon = StateMonitor(syn, ['s', 'R'], record=True, clock=syn.clock)
spike_mon = SpikeMonitor(syn)
net = Network(syn, mon, spike_mon)
net.run(duration * 1.5)

spiketimes = spike_mon.spike_trains()

for idx, level in enumerate(levels):
    plt.figure(1)
    plt.subplot(len(levels), 1, idx + 1)
    plt.plot(mon.t/ms, mon.s[idx])
    plt.xlim(0, 25)
    plt.xlabel('Time (msec)')
    plt.ylabel('Sp/sec')
    plt.text(15, np.nanmax(mon.s[idx])/2., 'Peak SPL=%s SPL' % str(level*dB));
    ymin, ymax = plt.ylim()
    if idx == 0:
        plt.title('Click responses')

    plt.figure(2)
    plt.subplot(len(levels), 1, idx + 1)
    plt.plot(mon.t/ms, mon.R[idx])
    plt.xlabel('Time (msec)')
    plt.xlabel('Time (msec)')
    plt.text(15, np.nanmax(mon.s[idx])/2., 'Peak SPL=%s SPL' % str(level*dB));
    plt.ylim(ymin, ymax)
    if idx == 0:
        plt.title('Click responses (with spikes and refractoriness)')
    plt.plot(spiketimes[idx]/ms,
```

(continues on next page)

(continued from previous page)

```
        np.ones(len(spiketimes[idx])) * np.nanmax(mon.R[idx]), 'rx')

print('Testing tone response')
duration = 60*ms
levels = [0, 20, 40, 60, 80]
tones = Sound([Sound.sequence([tone(cf, duration).atlevel(level*dB).ramp(when='both',
                                                                          ↪duration=10*ms,
                                                                          ↪inplace=False),
                               silence(duration=duration/2)])
               for level in levels])
ihc = TanCarney(MiddleEar(tones), [cf] * len(levels), update_interval=1)
syn = ZhangSynapse(ihc, cf)
mon = StateMonitor(syn, ['s', 'R'], record=True, clock=syn.clock)
spike_mon = SpikeMonitor(syn)
net = Network(syn, mon, spike_mon)
net.run(duration * 1.5)

spiketimes = spike_mon.spike_trains()

for idx, level in enumerate(levels):
    plt.figure(3)
    plt.subplot(len(levels), 1, idx + 1)
    plt.plot(mon.t/ms, mon.s[idx])
    plt.xlim(0, 120)
    plt.xlabel('Time (msec)')
    plt.ylabel('Sp/sec')
    plt.text(1.25 * duration/ms, np.nanmax(mon.s[idx])/2., '%s SPL' % str(level*dB));
    ymin, ymax = plt.ylim()
    if idx == 0:
        plt.title('CF=%.0f Hz - Response to Tone at CF' % cf)

    plt.figure(4)
    plt.subplot(len(levels), 1, idx + 1)
    plt.plot(mon.t/ms, mon.R[idx])
    plt.xlabel('Time (msec)')
    plt.xlabel('Time (msec)')
    plt.text(1.25 * duration/ms, np.nanmax(mon.R[idx])/2., '%s SPL' % str(level*dB));
    plt.ylim(ymin, ymax)
    if idx == 0:
        plt.title('CF=%.0f Hz - Response to Tone at CF (with spikes and
↪refractoriness)' % cf)
    plt.plot(spiketimes[idx] / ms,
         np.ones(len(spiketimes[idx])) * np.nanmax(mon.R[idx]), 'rx')

plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

CF-dependence of compressive nonlinearity in the Tan&Carney model

Reproduces Fig. 7 from:

Tan, Q., and L. H. Carney. "A Phenomenological Model for the Responses of Auditory-nerve Fibers. II. Nonlinear Tuning with a Frequency Glide". The Journal of the Acoustical Society of America 114 (2003): 2007.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

from brian2 import *
from brian2hears import *
from brian2hears.filtering.tan_carney import TanCarneySignal, MiddleEar

samplerate = 50*kHz
set_default_samplerate(samplerate)
duration = 50*ms

def product(*args):
    # Simple (and inefficient) variant of itertools.product that works for
    # Python 2.5 (directly returns a list instead of yielding one item at a
    # time)
    pools = map(tuple, args)
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    return result

def gen_tone(freq, level):
    '''
    Little helper function to generate a pure tone at frequency `freq` with
    the given `level`. The tone has a duration of 50ms and is ramped with
    two ramps of 2.5ms.
    '''
    freq = float(freq) * Hz
    level = float(level) * dB
    return tone(freq, duration).ramp(when='both',
                                     duration=2.5*ms,
                                     inplace=False).atlevel(level)

freqs = [500, 1100, 2000, 4000]
levels = np.arange(-10, 100.1, 5)
cf_level = product(freqs, levels)

# steady-state
start = int(10*ms*samplerate)
end = int(45*ms*samplerate)

# For Figure 7 we have manually adjusts the gain for different CFs -- otherwise
# the RMS values wouldn't be identical for low CFs. Therefore, try to estimate
# suitable gain values first using the lowest CF as a reference
ref_tone = gen_tone(freqs[0], levels[0])
F_out_reference = TanCarneySignal(MiddleEar(ref_tone, gain=1), freqs[0],
                                  update_interval=1).process().flatten()

ref_rms = np.sqrt(np.mean((F_out_reference[start:end] -
                           np.mean(F_out_reference[start:end]))**2))

gains = np.linspace(0.1, 1, 50) # for higher CFs we need lower gains
cf_gains = product(freqs[1:], gains)
tones = Sound([gen_tone(freq, levels[0]) for freq, _ in cf_gains])
F_out_test = TanCarneySignal(MiddleEar(tones, gain=np.array([g for _, g in cf_
↪gains])),
```

(continues on next page)

(continued from previous page)

```
                                [cf for cf,_  in cf_gains], update_interval=1).process()

reshaped_Fout = F_out_test.T.reshape((len(freqs[1:]), len(gains), -1))
rms = np.sqrt(np.mean((reshaped_Fout[:, :, start:end].T -
                       np.mean(reshaped_Fout[:, :, start:end], axis=2).T).T**2,
                       axis=2))

# get the best gain for each CF using simple linear interpolation
gain_dict = {freqs[0]: 1.} # reference gain
for idx, freq in enumerate(freqs[1:]):
    gain_dict[freq] = interp1d(rms[idx, :], gains)(ref_rms)

# now do the real test: tones at different levels for different CFs
tones = Sound([gen_tone(freq, level) for freq, level in cf_level])
F_out = TanCarneySignal(MiddleEar(tones,
                                  gain=np.array([gain_dict[cf] for cf, _ in cf_
↪level])),
                        [cf for cf, _ in cf_level],
                        update_interval=1).process()

reshaped_Fout = F_out.T.reshape((len(freqs), len(levels), -1))

rms = np.sqrt(np.mean((reshaped_Fout[:, :, start:end].T -
                       np.mean(reshaped_Fout[:, :, start:end], axis=2).T).T**2,
                       axis=2))

# This should more or less reproduce Fig. 7
plt.plot(levels, rms.T)
plt.legend(['%.0f Hz' % cf for cf in freqs], loc='best')
plt.xlim(-20, 100)
plt.ylim(1e-6, 1)
plt.yscale('log')
plt.xlabel('input signal SPL (dB)')
plt.ylabel('rms of AC component of Fout')
plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

Index

H

I

J

L

M

N

O

P

R

S

T

V

W

Z