FP7-ICT-2013-C TWO!EARS Project 618075

**Supplement to Deliverable 2.3
(Extension of the binaural model and integration of monaural and binaural models in software package)**

The auditory front-end framework
User manual

WP2 *

November 30, 2015

* The Two!Ears project (`http://www.twoears.eu`) has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 618075.

Project acronym:	Two!Ears
Project full title:	Reading the world with Two!Ears

Work package:	WP2
Document number:	Supplement to D2.3 (Extension of the binaural model and integration of monaural and binaural models in software package)
Document title:	The auditory front-end framework - User manual
Version:	1

Delivery date:	30th November 2015
Actual publication date:	30th November 2015
Dissemination level:	Restricted
Nature:	Demo

Editor(s)/lead beneficiary:	Tobias May
Author(s):	Tobias May, Remi Decorsière, Chungeun Kim, Armin Kohlrausch
Reviewer(s):	Bruno Gas

Contents

1 Executive summary

The goal of the Two!Ears project is to develop an intelligent, active computational model of auditory perception and experience in a multi-modal context. The auditory front-end (AFE) represents the first stage of the system architecture and concerns bottom-up auditory signal processing, which transforms binaural signals into multi-dimensional auditory representations. The output provided by this AFE consists of several transformed versions of ear signals enriched by perception-based descriptors which form the input to the higher model stages. Specific emphasis is given on the modularity of the software framework, making this AFE more than just a collection of models documented in the literature. Bottom-up signal processing is implemented as a collection of processor modules, which are instantiated and routed by a manager object. A variety of processor modules is provided to compute auditory cues such as ratemaps, interaural time and level differences, interaural coherence, onsets and offsets. An object-oriented approach is used throughout, giving benefits of reusability, encapsulation and extensibility. This affords great flexibility, and allows modification of bottom-up processing in response to feedback from higher levels of the system during run time. Such top-down feedback could, for instance, lead to on-the-fly changes in parameter values of peripheral modules, like the filter bandwidths of the basilar-membrane filters. In addition, the object-oriented AFE framework allows direct switching between alternative peripheral filter modules, while keeping all other components unchanged, allowing for a systematic comparison of alternative processors. Finally, the AFE framework supports online processing of the two-channel ear signals.

In the description of work (DOW) for the Two!Ears project, deliverable 2.3, documenting the extended AFE software package, has the nature of a (public) demonstration. This deliverable has been realized by making the updated AFE software package publicly available at https://github.com/TWOEARS/auditory-front-end. This supplementary document is an updated manual to describe the fundamentals of the current AFE framework and to help others to make use of its range of capabilities. Chapter 2 gives an overview about the framework functionality, including the newly incorporated features to realise feedbacks. A more technical description of the framework is presented in chapter 3. Chapter 4 gives a detailed overview about the auditory representations that are supported by the framework. In addition, the update now includes in Chapter 5 detailed step-by-step instructions for the users to develop his/her own AFE processors, followed by a closing overall conclusion in Chapter 6. To support the concept of “reproducible research”, appendix A comprises a list of all demo files used in the description of the individual modules in Chapter 4. These can be freely accessed and allow all users a replication of the examples in their own software environment.

2 The auditory front-end framework

2.1 Framework functionality

The purpose of the Two!Ears auditory front-end (AFE) is to extract a *subset* of common auditory representations from a binaural recording or from a *stream* of binaural audio data. These representations are to be used later by higher modeling or decision stages. This short description of the role of the AFE highlights its three fundamental properties:

- The framework operates on a request-based mechanism and extracts the *subset* of all available representations which has been requested by the user. Most of the available representations are computed from other representations, i.e., they *depend* on other representations. Because different representations can have a common dependency, the available representations are organized following a "dependency tree". The framework is built such as to respect this structure and limit redundancy. For example, if a user requests A and B, both depending on a representation C, the software will not compute C twice but will instead reuse it. As will be presented later, to achieve this, the processing is shared among processors. Each processor is responsible for one individual step in the extraction of a given representation. The framework then instantiates only the necessary processors at a given time.

- It can operate on a *stream* of input data. In other words, the framework can operate on consecutive chunks of input signal, each of arbitrary length, while returning the same output(s) as if the whole signal (i.e., the concatenated chunks) was used as input.

- The user request can be modified at *run time*, i.e., during the execution of the framework. New representations can be requested, or the parameters of existing representations can be changed in between two blocks of input signal. This mechanism is particularly designed to allow higher stages of the whole Two!Ears framework to provide feedback, requesting adjustments to the computation of auditory representations. In connection to the first point above, when the user requests such a change, the framework will identify where in the dependency tree the requested change starts affecting the processing and will only compute the steps affected.

2.2 Getting started

The Two!Ears AFE framework was developed entirely using Matlab version 8.3.0.532 (R2014a). It was tested for backward compatibility down to Matlab version 8.0.0.783 (R2012b). The source code, test and demo scripts are all available from the public repository at https://github.com/TWOEARS/auditory-front-end.

The files for the AFE are divided in three folders, `/doc`, `/src` and `/test` containing respectively the documentation of the framework, the source code, and various test scripts. Once Matlab opened, the source code (and if needed the other folders) should be added to the Matlab path. This can be done by executing the script `startAuditoryFrontEnd` in the main folder:

```
>> startAuditoryFrontEnd
```

As will be seen in the following subsection, the framework is request-based: the user places one or more requests, and then informs the framework that it should perform the processing. Each request corresponds to a given auditory representation, which is associated with a short *nametag*. The command `requestList` can be used to get a summary of all supported auditory representations:

```
>> requestList

  Request name       Label                             Processor
  ------------       -----                             -------------------
  adaptation         Adaptation loop output            adaptationProc
  amsFeatures        Amplitude modulation spectrogram  modulationProc
  autocorrelation    Autocorrelation computation       autocorrelationProc
  crosscorrelation   Crosscorrelation computation      crosscorrelationProc
  filterbank         DRNL output                       drnlProc
  filterbank         Gammatone filterbank output       gammatoneProc
  gabor              Gabor features extraction         gaborProc
  ic                 Inter-aural coherence             icProc
  ild                Inter-aural level difference      ildProc
  innerhaircell      Inner hair-cell envelope          ihcProc
  itd                Inter-aural time difference       itdProc
  moc                Medial Olivo-Cochlear feedback    mocProc
  myNewRequest       A description of my new request   templateProc
  offsetMap          Offset map                        offsetMapProc
  offsetStrength     Offset strength                   offsetProc
  onsetMap           Onset map                         onsetMapProc
  onsetStrength      Onset strength                    onsetProc
  pitch              Pitch estimation                  pitchProc
  precedence         Precedence effect                 precedenceProc
  ratemap            Ratemap extraction                ratemapProc
  spectralFeatures   Spectral features                 spectralFeaturesProc
  time               Time domain signal                preProc
```

A detailed description of the individual processors used to obtain these auditory representations will be given in chapter 4.

The implementation of the AFE is object-oriented, and two objects are needed to extract any representation:

- A *data* object, in which the input signal, the requested representation, and also the dependent representations that were computed in the process are all stored.

- A *manager* object which takes care of creating the necessary processors as well as managing the processing.

In the following sections, examples of increasing complexity are given to demonstrate how to create these two objects, and which functionalities they offer.

2.3 Computation of an auditory representation

The following sections describe how the AFE framework can be used to compute an auditory representation with default parameters of a given input signal. We will start with a simple example, and gradually explain how the user can gain more control over the respective parameters. It is assumed that the entire input signal - for which the auditory representation should be computed - is available. Therefore, this operation is referred to as *batch processing*. As stated before, the framework is also compatible with *chunk-based processing* (i.e., when the input signal is acquired continuously over time, but the auditory representation is computed for smaller signal chunks). The chunk-based processing will be explained in Sec. 2.4.

2.3.1 Using default parameters

As an example, extracting the interaural level difference `'ild'` for a stereo signal `sIn` (e.g., obtained from a '`.wav`' file through MATLAB'S `wavread`) sampled at a frequency `fsHz` (in Hz) can be done in the following steps:

```
1 % Instantiation of data and manager objects
2 dataObj = dataObject(sIn,fsHz);
3 managerObj = manager(dataObj);
4
5 % Request the computation of ILDs
6 sOut = managerObj.addProcessor('ild');
7
8 % Request the processing
9 managerObj.processSignal;
```

Line 2 and 3 show the instantiation of the two fundamental objects: the data object and the manager. Note that the data object is always instantiated first, as the manager needs a data object instance as input argument to be constructed. The manager instance in line 3 is however an "empty" instance of the `manager` class, in the sense that it will not perform any processing. Hence a processing needs to be requested, as done in line 6. This particular example will request the computation of the inter-aural level difference `'ild'`. This step is configuring the manager instance `managerObj` to perform that type of processing, but the processing itself is performed at line 9 by calling the `processSignal` method of the manager class.

The request of an auditory representation via the `addProcessor` method of the manager class on line 6 returns as an output argument a handle to the requested signal, here named `sOut`. In the AFE framework, signals are also objects. For example, for the output signal just generated:

```
>> sOut

ans =

  TimeFrequencySignal with properties:

          cfHz: [1x31 double]
         Label: 'Interaural level difference'
          Name: 'ild'
    Dimensions: 'nSamples x nFilters'
          FsHz: 100
       Channel: 'mono'
          Data: [267x31 circVBufArrayInterface]
```

This shows the various properties of the signal object `sOut`. These properties will be described in detail in chapter 3. To access the computed representation, e.g., for further processing, one can create a copy of the data contained in the signal into a variable, say `myILDs`:

```
>> myILDs = sOut .Data(:);
```

Note

The use of the column operator `(:)`. That is because the property `.Data` of signal objects is not a conventional MATLAB array and one needs this syntax to access all the values it stores.

The nature of the `.Data` property is further described in Sec. 3.2.1.

2.3.2 Input/output signals dimensionality

The input signal `sIn`, for which a given auditory representation needs to be computed, is a simple array. Its first dimension (lines) should span time. Its first column should correspond to the left channel (or mono channel, if it is not a stereo signal) and the second column to the right channel. This is typically the format returned by MATLAB'S embedded functions `audioread` and `wavread`.

The input signal can be either mono or stereo/binaural. The framework can operate on both. However, some representations, such as the interaural level difference (ILD) as requested in the previous example, are based on a comparison between the left and the right ear signals. If a mono signal was provided instead of a binaural signal, the request of computing the ILD representation would produce the following warning and the request would not be computed:

```
Warning: Cannot instantiate a binaural processor with a mono input signal!
> In manager>manager.addProcessor at 1127
```

The dimensions of the output signal from the `addProcessor` method will depend on the representation requested. In the previous example, the `'ild'` request returns a single output for a stereo input. However, when the request is based on a single channel and the input is stereo, the processing will be performed for left and right channel, and both left and right outputs are returned. In such cases, the output from the method `addProcessor` will be a cell array of dimensions `1 x 2` containing output signals for the left channel (first column) and right channel (second column). For example, the returned `sOut` could take the form:

```
>> sOut

sOut =

    [1x1 TimeFrequencySignal]    [1x1 TimeFrequencySignal]
```

The left-channel output can be accessed using `sOut{1}`, and similarly, `sOut{2}` for the right-channel output.

2.3.3 Changing parameters used for computation

For the requested representation

Each individual processor that is supported by the AFE can be controlled by a set of parameters. Each parameter can be accessed by a unique *nametag* and has a default value. A summary of all parameter names and default values for the individual processors can be listed by the command `parameterHelper`:

```
>> parameterHelper

Parameter handling in the TWO!EARS Auditory Front-End
-------------------------------------------------
The extraction of various auditory representations performed by the TWO!
   EARS Auditory Front-End software involves many parameters. Each
   parameter is given a unique name and a default value. When placing a
```

```
    request for TWO!EARS auditory front-end processing that uses one or
    more non-default parameters, a specific structure of non-default
    parameters needs to be provided as input. Such structure can be
    generated from genParStruct, using pairs of parameter name and chosen
    value as inputs.

Parameters names for each processor are listed below:
        Amplitude modulation
        Auto-correlation
        Cross-correlation
        DRNL filterbank
        Gabor features extractor
        Gammatone filterbank
        IC Extractor
        ILD Extractor
        ITD Extractor
        Medial Olivo-Cochlear feedback processor
        Inner hair-cell envelope extraction
        Neural adaptation model
        Offset detection
        Offset mapping
        Onset detection
        Onset mapping
        Pitch
        Pre-processing stage
        Precedence effect
        Ratemap
        Spectral features
        Plotting parameters
```

Each element in the list is a hyperlink, which will reveal the list of parameters for a given element, e.g.,

```
Interaural Level Difference parameters:

  Name             Default   Description
  ----             -------   -----------
  ild_wname        'hann'    Window name
  ild_wSizeSec     0.02      Window duration (s)
  ild_hSizeSec     0.01      Window step size (s)
```

It can be seen that the ILD processor can be controlled by three parameters, namely `ild_wname`, `ild_wSizeSec` and `ild_hSizeSec`. A particular parameter can be changed by creating a parameter structure which contains the parameter name (*nametags*) and the corresponding value. The function `genParStruct` can be used to create such a parameter structure. For instance:

```
>> parameters = genParStruct('ild_wSizeSec',0.04,'ild_hSizeSec',0.02);
```

```
parameters =
    ild_wSizeSec: 0.0400
    ild_hSizeSec: 0.0200
```

will generate a suitable parameter structure `parameters` to request the computation of ILD with a window duration of 40 ms and a step size of 20 ms. This parameter structure is then passed as a second input argument in the `addProcessor` method of a manager object. The previous example can be rewritten considering the change in parameter values as follows:

```
% Instantiation of data and manager objects
dataObj = dataObject(sIn,fsHz);
managerObj = manager(dataObj);

% Non-default parameter values
parameters = genParStruct('ild_wSizeSec',0.04,'ild_hSizeSec',0.02);

% Place a request for the computation of ILDs
sOut = managerObj.addProcessor('ild',parameters);

% Perform processing
managerObj.processSignal;
```

For a dependency of the request

The previous example showed that the processor extracting ILDs was accepting three parameters. However, the representation it returns, the ILDs, will depend on more than these three parameters. For instance, it includes a certain number of frequency channels, but there is no parameter to control these in the ILD processor. That is because such parameters are from other processors that were used in intermediate steps to obtain the ILD. Controlling these parameters therefore requires knowledge of the individual steps in the processing.

Most auditory representations will depend on another representation, itself being derived from yet another one. Thus, there is a chain of *dependencies* between different representations, and multiple processing stages will be required to compute a particular output. The list of dependencies for a given request can be visualized using the function `Processor.getDependencyList('processorName')`, e.g.,

```
>> Processor.getDependencyList('ildProc')

ans =

    'innerhaircell'    'filterbank'    'time'
```

shows that the ILD depends on the inner hair-cell representation ('innerhaircell'), which itself is obtained from the output of a gammatone filterbank ('filterbank'). The filterbank is derived from the time-domain signal, which itself has no further dependency as it is directly derived from the input signal.

When placing a request to the manager, the user can also request a change in parameters of any of the request's dependencies. For example, the number of frequency channels in the ILD representation is a property of the filterbank, controlled by the parameter 'fb_nChannels' (which name can be found using parameterHelper.m). This parameter can also be requested to have a non-default value, although it is not a parameter of the processor in charge of computing the ILD. This is done in the same way as previously shown:

```
% Non-default parameter values
parameters = genParStruct('fb_nChannels',16);

% Place a request for the computation of ILDs
sOut = managerObj.addProcessor('ild',parameters);

% Perform processing
managerObj.processSignal;
```

The resulting ILD representation stored in sOut will be based on 16 channels, instead of 31.

2.3.4 Compute multiple auditory representations

Place multiple requests

Multiple requests are supported in the framework, and can be carried out by consecutive calls to the addProcessor method of an instance of the manager with a single request argument. It is also possible to have a single call to the addProcessor method with a cell array of requests, e.g.:

```
% Place a request for the computation of ILDs AND autocorrelation
[sOut1 sOut2] = managerObj.addProcessor({'ild','autocorrelation'})
```

This way, the manager set up in the previous example will extract ILD and an auto-correlation representation, and provide handles to the three signals, in sOut1{1} for the ILD (it is a mono representation), sOut2{1} and sOut2{2} for the autocorrelations of respectively left and right channels.

To use non-default parameter values, three syntax are possible:

- If there are several requests, but all use the same set of parameter values p:

```
managerObj.addProcessor({'name1', .. ,'nameN'},p)
```

- If there is only one request (name), but with different sets of parameter values (p1,...,pN), e.g., for investigating the influence of a given parameter:

```
managerObj.addProcessor('name',{p1, .. ,pN})
```

- If there are several requests and some, or all, of them use a different set of parameter values, then it is necessary to have a set of parameter (p1,...,pN) for each request (possibly by duplicating the common ones) and place them in a cell array as follows:

```
managerObj.addProcessor({'name1', .. ,'nameN'},{p1, .. ,pN})
```

Note that in the two examples above, no output is specified for the addProcessor method, but the representations will be computed nonetheless. The output of addProcessor is there for convenience and the following subsection will explain how to get a hang on the computed signals without an explicit handle from addProcessor.

Requests can also be placed directly as optional arguments in the manager constructor, e.g., to reproduce the previous script example:

```
% Instantiation of data and manager objects
dataObj = dataObject(sIn,fsHz);
managerObj = manager(dataObj,{'ild','autocorrelation'});
```

The three possibilities described above can also be used in this syntax form.

Computing the signals

This is done in the exact same way as for a single request, by calling the processSignal method of the manager:

```
% Perform processing
managerObj.processSignal;
```

Access internal signals

The optional output of the addProcessor method is provided for convenience. It is actually a pointer (or handle, in Matlabs terms) to the actual signal object which is hosted by the data object on which the manager is based. Once the processing is carried out, the properties of the data object can be inspected:

```
>> dataObj

dataObj =

  dataObject with properties:

   bufferSize_s: 10
       isStereo: 1
      gammatone: {[1x1 TimeFrequencySignal]  [1x1 TimeFrequencySignal]}
        ratemap: {[1x1 TimeFrequencySignal]  [1x1 TimeFrequencySignal]}
autocorrelation: {[1x1 CorrelationSignal]  [1x1 CorrelationSignal]}
            ild: {[1x1 TimeFrequencySignal]}
          input: {[1x1 TimeDomainSignal]  [1x1 TimeDomainSignal]}
           time: {[1x1 TimeDomainSignal]  [1x1 TimeDomainSignal]}
  innerhaircell: {[1x1 TimeFrequencySignal]  [1x1 TimeFrequencySignal]}
```

Apart from the properties `bufferSize_s` and `isStereo` which are inherent properties of the data object (and discussed later in chapter 3), the remaining properties each correspond to one of the representations computed to achieve the user's request(s). They are each arranged in cell arrays, with first column being the left, or mono channel, and the second column the right channel. For instance, to get a handle `sGammaR` to the right channel of the gammatone filterbank output, type:

```
>> sGammaR = dataObj.filterbank{2}

sGammaR =

   TimeFrequencySignal with properties:

          cfHz: [1x31 double]
         Label: 'Gammatone filterbank output'
          Name: 'filterbank'
    Dimensions: 'nSamples x nFilters'
          FsHz: 44100
       Channel: 'right'
          Data: [118299x31 circVBufArrayInterface]
```

2.3.5 How to plot the result

Plotting auditory representations is made very easy in the AFE framework. As explained before, each representation that was computed during a session is stored as a signal object, which each are individual properties of the data object. Signal objects of each type have a `plot` method. Called without any input arguments, `signal.plot` will adequately plot the representation stored in `signal` in a new figure, and returns as output a handle to said figure. The plotting method for all signals can accept at least one optional argument, which is a handle to an already existing figure or subplot in a figure. This way the representation can be included in an existing plot. A second optional argument is a structure of non-default plot parameters. The `parameterHelper` script also lists plotting options, and they can be modified in the same way as processor parameters, via the script `genParStruct`. These concepts can be summed up in the following example lines, that follows right after the demo code from the previous subsection:

```
11 % Request the processing
12 managerObj.processSignal;
13 
14 % Plot the ILDs in a separate figure
15 sOut{1}.plot;
16 
17 % Create an empty figure with subplots
18 figure;
19 h1 = subplot(2,2,1);
20 h2 = subplot(2,2,2);
21 h3 = subplot(2,2,3);
22 h4 = subplot(2,2,4);
23 
24 % Change plotting options to remove colorbar and reduce title size
25 p = genParStruct('bColorbar',0,'fsize_title',12);
26 
27 % Plot additional representations
28 dataObj.innerhaircell{1}.plot(h1,p);
29 dataObj.innerhaircell{2}.plot(h2,p);
30 dataObj.filterbank{1}.plot(h3,p);
31 dataObj.filterbank{2}.plot(h4,p);
```

This script will produce the two figure windows displayed in Fig. 2.1. Line 22 of the script creates the window “Figure 1”, while lines 35 to 38 populate the window “Figure 2” which was created earlier (in lines 25 to 29).

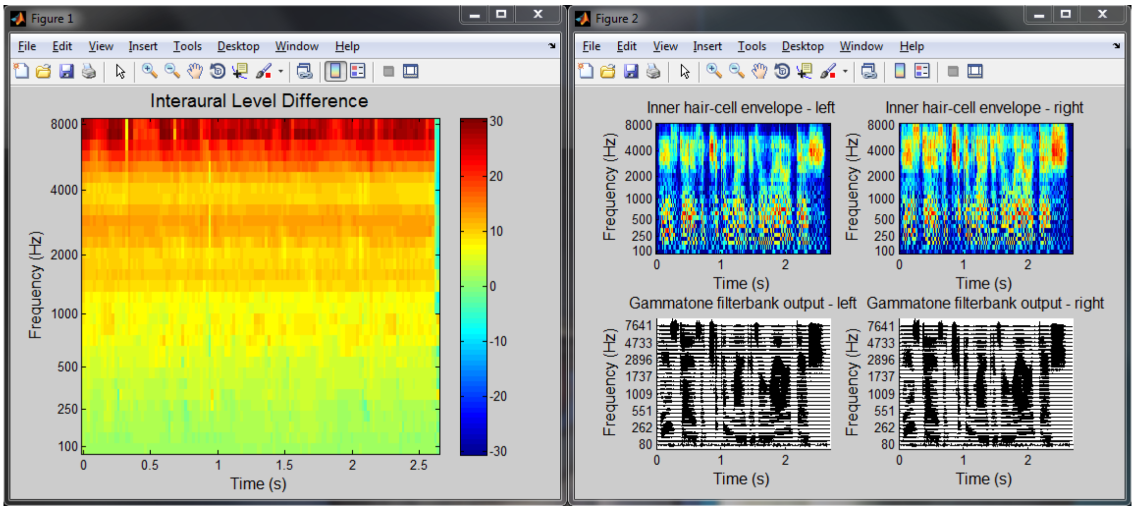
Figure 1
File Edit View Insert Tools Desktop Window Help
Interaural Level Difference
Frequency (Hz)
Time (s)
Figure 2
Inner hair-cell envelope - left
Inner hair-cell envelope - right
Gammatone filterbank output - left
Gammatone filterbank output - right

Figure 2.1: The two example figures generated by the demo script.

2.4 Chunk-based processing

As mentioned in the previous section, the framework is designed to be compatible with chunk-based processing. As opposed to “batch processing”, where the entire input signal is known *a priori*, this means working with consecutive chunks of input signals of arbitrary size. In practice the chunk size will often be the same from one chunk to another. However, this is not a requirement here, and the framework can accept input chunks of varying size.

The main constraint behind working with an input that is segmented into chunks is that the returned output should be exactly the same as if the whole input signal (i.e., the concatenated chunks) was used as input. In other terms, the transition from one chunk to the next needs to be taken into account in the processing. For example, concatenating the outputs obtained from a simple filter applied *separately* to two consecutive chunks will not provide the same output as if the concatenated chunks were used as input. To obtain the same output, one should for example use methods such as overlap-add or overlap-save. This is not trivial, particularly in the context of the AFE where more complex operations than simple filtering are involved. A general description of the method used to ensure chunk-based processing is given in Sec. 3.3.6.

Handling segmented input in practice is done mostly the same way as for a whole input signal. The available demo script `DEMO_ChunkBased.m` provides an example of chunk-based processing by simulating a chunk-based acquisition of the input signal with variable chunk size and computing the corresponding ILDs.

In this script, one can note the two differences in using the AFE in a chunk-based scenario, in comparison to a batch scenario:

```
18 % Instantiation of data and manager objects
19 dataObj = dataObject([],fsHz,10,2);
20 managerObj = manager(dataObj);
```

Because the signal is not known before the processing is carried out, the data object cannot be initialized from the input signal. Hence, as is seen on line 19, one needs to instantiate an empty data object, by leaving the first input argument blank. The sampling frequency is still necessary however. The third argument (here set to `10`) is a global signal buffer size in seconds. Because in an online scenario, the framework could be operating over a long period of time, internal representations cannot be stored over the whole duration and are instead kept for the duration mentioned there. The last argument (`2`) indicates the number of channel that the framework should expect from the input (a mono input would have been indicated by `1`). Again, it is necessary to know the number of channels in the input signal, to instantiate the necessary objects in the data object and the manager.

```
40 % Request the processing of the chunk
41 managerObj.processChunk(sIn(chunkStart:chunkStop,:),1);
```

The processing is carried out on line 40 by calling the `processChunk` method of the manager. This method takes as input argument the new chunk of input signal. The additional argument, `1`, indicates that the results should be appended to the internal representations already computed. This can be set to `0` in cases where keeping track of the output for the previous chunks is unnecessary, for instance if the output of the current chunk is used by a higher-level function. The difference with the `processSignal` method is important. Although `processSignal` actually calls internally `processChunk`, it also resets internal states of the framework (what ensures continuity between chunks) before processing.

The script `DEMO_ChunkBased.m` will also compute the offline result and will plot the difference in output for the two computations. This plot is shown in Fig. 2.2. Note the magnitude on the order of 10^{-15}, which is in the range of Matlab numerical precision, suggesting that the representations computed online or offline are the same up to some round-off errors.

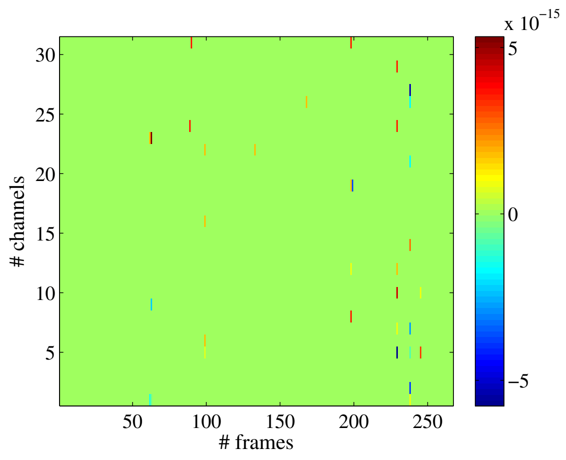

Figure 2.2: Difference in ILD obtained in online and offline processing

2.5 Feedback inclusion

A key concept of the AFE is its ability to respond to feedback from the user or from external, higher stage models. Conceptually, feedback at the stage of auditory feature extraction is realised by allowing changes in parameters and/or changes in which features are extracted at run time, i.e., in between two chunks of input signal in a chunk-based processing scenario (2.4).

In practice, three types of feedback can be identified: - A new request is placed - One or more parameters of an existing request is changed - A processor has become obsolete and is deleted.

2.5.1 Placing a new request

Placing a new request at run time, i.e., online, is done exactly as it is done offline (2.3), by calling the `addProcessor` method of an existing manager instance.

2.5.2 Modifying a processor parameter

Warning

Some parameters are blacklisted for modifications as they would imply a change in dimension in the output signal of the processor. If you need to perform this change anyway, consider placing a new request instead of modifying an existing one.

Modifying a processor parameter can be done by calling the `modifyParameter` method of that processor in between two calls to the `processChunk` of the manager instance.

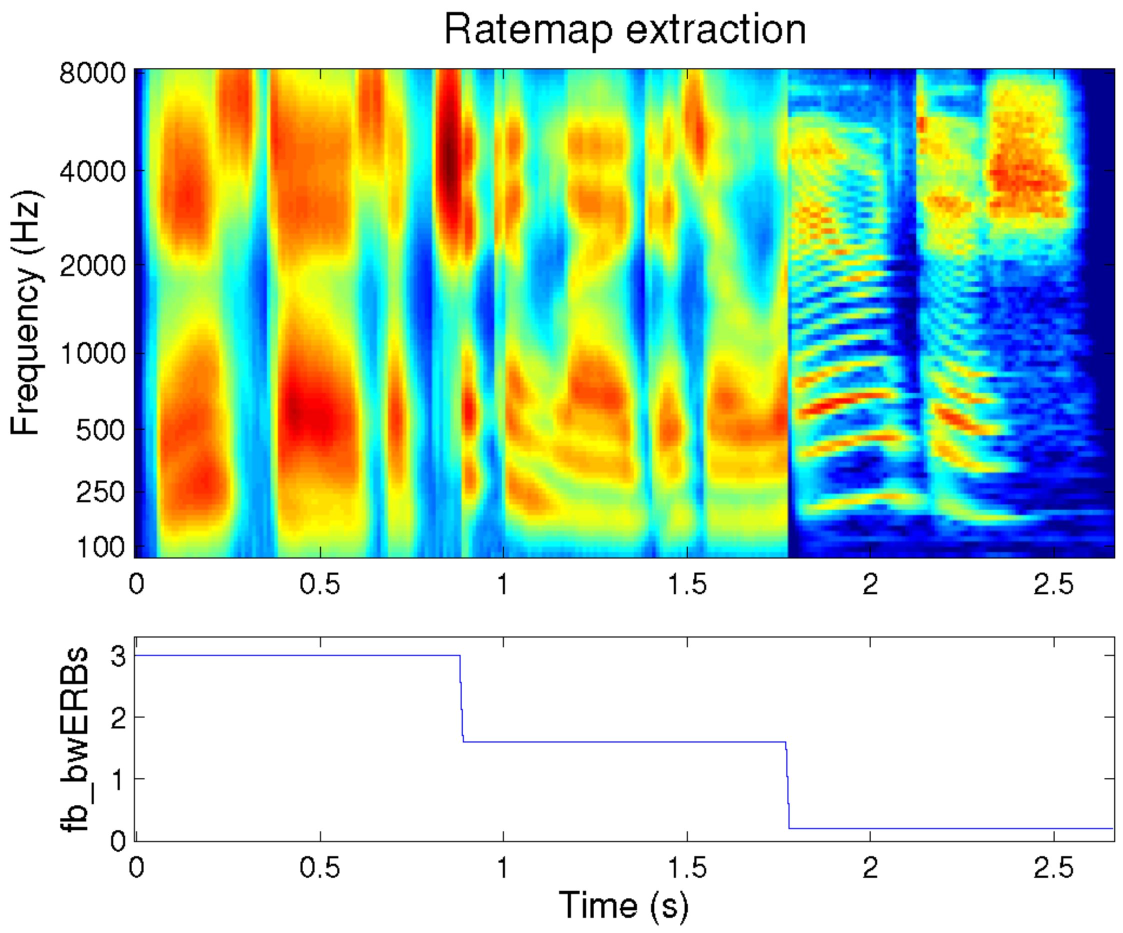

Figure 2.3: Sharpening the frequency selectivity of the ear by means of feedback

Figure 2.3 illustrates feedback capability of the AFE. This is a rate-map representation of a speech signal that is extracted online. The bandwidth of auditory filters, controlled by the parameter `fb_bwERBs` in the original request was set to 3 `ERBs`, an abnormally large value in comparison to a normal-hearing frequency selectivity. Throughout the processing, the bandwidth is reduced to 1.5 `ERBs` by calling:

```
mObj.Processors{2}.modifyParameter('fb_bwERBs',1.5);
```

in between two calls to the “processChunk“ method of the manager “mObj“, at around 0.9s. Here, `mObj.Processors{2}` points to the auditory filterbank processor, an instance of a gammatone processor. The bandwidth is later (at 1.75s) reduced even further (to about 0.25). Figure 2.3 illustrates how the narrower auditory filters will reveal the harmonic structure of speech.

Note

If a processor is modified in response to feedback, subsequent processors need to reset themselves, in order not to carry on incorrect internal states. This is done automatically inside the framework. For example, in the figure above, internal filters of the inner hair-cell envelope extraction and the ratemap computation are reset accordingly when the bandwidth parameter is changed

2.5.3 Deleting a processor

Deleting a processor is simply done by calling its `remove` method. As for parameter modifications, this affects subsequent processors, as they will also become obsolete. Hence they will also be automatically deleted.

Deleting processors will leave empty entries in the `manager.Processors` cell array. To clean up the list of processor, call the `cleanup()` method of your manager instance.

2.6 List of commands

This section sums up the commands that could be relevant to a standard user of the AFE framework. It does not describe each action extensively, nor does it give a full list of corresponding parameters. A more detailed description can be obtained through calling the help script of a given method from MATLAB’S command window. Note that one can get help on a specific method of a given class. For example

```
>> help manager.processChunk
```

will return help related to the `processChunk` method of the manager class. The following aims at being concise, hence optional inputs are marked as "`...`" and can be reviewed from the specific method help.

Signal objects `sObj`

`sObj.Data(:)`	Returns all the data in the signal
`sObj.Data(n1:n2)`	Returns the data in the time interval `[n1,n2]` (samples)
`sObj.findProcessor(mObj)`	Finds processor that computed the signal
`sObj.getParameters(mObj)`	Parameter summary for that signal
`sObj.getSignalBlock(T,...)`	Returns last `T` seconds of the signal
`sObj.play`	Plays back the signal (time-domain signals only)
`sObj.plot(...)`	Plots the signal

Data objects `dObj`

`dataObject(s,fs,bufSize,nChannels)`	Constructor
`dObj.addSignal(sObj)`	Adds a signal object
`dObj.clearData`	Clears all signals in `dObj`
`dObj.getParameterSummary(mObj)`	Lists parameter used for each signal
`dObj.play`	Plays back the containing audio signal

Processors `pObj`

`pObj.LowerDependencies`	List of processors `pObj` depends on
`pObj.UpperDependencies`	List of processors depending on `pObj`
`pObj.getCurrentParameters`	Parameter summary for that processor
`pObj.getDependentParameter(parName)`	Value of a parameter from `pObj` or its dependencies
`pObj.hasParameters(parStruct)`	True if `pObj` used the exact values in `parStruct`
`pObj.Input`	Handle to input signal object
`pObj.Output`	Handle to output signal object
`pObj.modifyParameter`	Change a parameter value
`pObj.remove`	Removes a processor (and its subsequent processors)

Manager `mObj`

`manager(dObj)`	Constructor
`manager(dObj,name,param)`	Constructor with initial request
`mObj.addProcessor(name,param)`	Adds a processor (including eventual dependencies)
`mObj.Data`	Handle to the associated data object
`mObj.processChunk(input,...)`	Process a new chunk
`mObj.Processors`	Lists instantiated processors
`mObj.processSignal`	Process a signal offline
`mObj.reset`	Resets all processors
`mObj.cleanup`	Cleans up the list of processors

2.7 Acknowledgment

The AFE framework includes the following contributions from publicly available MATLAB toolboxes or classes:

- AMToolbox
- LTFAT
- Voicebox
- circVBuf

3 Technical description

3.1 Overview

Many different auditory models are available that can transform an input signal into an auditory representation. The actual design challenges behind the AFE framework arise from the multiplicity of supported representations, the requirement to process continuous signal in a chunk-based manner, and the ability to change what is being computed at run-time, which will allow the incorporation of *feedback* from higher processing stages. In addition to these three constraints, the framework will be subject to frequent updates in the future of the Two!Ears project (e.g., adding new processors), so the expandability and maintainability of its implementation should be optimal. For these reasons, the framework is implemented using a modular object-oriented approach.

This chapter exposes the architecture and interactions of all the objects involved in the AFE and how the main constraints were tackled conceptually. In an effort to respect encapsulation and the hierarchical organization of the objects, the sections are arranged in a "bottom-up" way: from the most fundamental objects to the more global processing.

All classes involved in the AFE implementation are inheriting the Matlab `handle` master class. This allows every created object to be of the `handle` type, and simulates a "call-by-reference" when manipulating the objects. Given an object `obj` inheriting the handle class, doing `obj2 = obj` will not copy the object, but only obtain a pointer to it. If `obj` is modified, then so is `obj2`. This avoids unnecessary copies of objects, limiting memory use, as well as providing user friendly handles to objects included under many levels of class hierarchy. The user can manipulate a simple short-named handle instead of tediously accessing the object.

3.2 Data handling

3.2.1 Circular buffer

Memory pre-allocation of large arrays in Matlab is well known to be a critical operation for optimizing computation time. The AFE, particularly in an online scenario, will be confronted with this problem. For each new chunk of the input signal, chunks of output are computed for each internal representation and are appended to the already existing output. Computation time will be strongly affected if the arrays containing the data are not initialized appropriately (i.e., the memory it occupies is pre-allocated) to fit the input signal duration.

The issue in a real-time scenario is that the signal duration is unknown. To overcome this problem, data for each signal is stored in a buffer of fixed duration which is itself pre-allocated. Buffers are updated following a first in, first out (FIFO) rule: once the buffer is full, the oldest samples in the buffer are overwritten by the new signal samples.

The `circVBuf` class

A conceptual way of implementing a FIFO rule is to use circular (or ring) buffers. The inconvenience of a traditional, linear buffer is that once it is full and new input overwrites old samples (i.e., it is in its "steady-state"), reading the data from it implies reaching the end of the buffer and continuing reading from its beginning. The data read will be in two fragments, because of the linear buffer having a physical beginning and end which do not match to the oldest and newest data samples. This is eliminated in circular buffers which do not have a beginning or end, and a contiguous segment is always obtained upon reading. Circular buffers were implemented for the AFE framework based on the third-party class provided by Göbbert (2014), which has been slightly modified to account for multi-dimensional data (instead of vector-only).

Circular buffer interface

The `circVBuf` class provides a buffer that is conceptually circular, in the sense that it allows continuous reading of the data. However in practice it still stores data in a linear array in Matlab (the size of which is, however, twice the size of the actual data). Accessing stored data requires knowledge about this class and can be tedious to a naive user. To eliminate confusion and make the buffer transparent to the user, the interface `circVBuffArrayInterface` was implemented, with the aim of allowing the buffer to use most basic array operations.

Given a circular buffer `circBuffer`, the interface is obtained by

```
buffer = circVBufArrayInterface(circBuffer)
```

It will allow the following operations:

- `buffer(n1:n2)` returns stored data between positions `n1` and `n2`, where position `1` is the oldest sample in the buffer (but not necessarily the first one in the actual array storing data, due to circularity). For multiple dimensions, these indices always refer to the first dimension. To return stored data up to the most recent sample, use `buffer(n1:end)`.
- `buffer(:)` returns all data stored in the buffer (ignoring "empty" sections of the buffer, if said buffer was never filled).
- `buffer('new')` returns the latest chunk of data that was added to the buffer.
- `length(buffer)` returns the effective (i.e., ignoring empty sections) buffer length across its first dimension.
- `size(buffer)` returns the effective size of the buffer (including other dimensions).
- `numel(buffer)` returns the total number of elements stored (calculated as product of the effective dimensions).
- `isempty(buffer)` returns `true` when no data is stored, `false` otherwise.

This provides an array behavior to the buffers, simplifying greatly their use.

Note

Note that the only limitation is the need of the column operator `:` to access all data, as in `buffer(:)`. Without it, `buffer` will return a handle to the `circVBufArrayInterface` object.

3.2.2 Signal objects

Signals are implemented as objects in the AFE. To avoid code repetition and make better use of object-oriented concepts, signals are grouped according to their dimensionality, as they then share the same properties. The following classes are implemented:

- `TimeDomainSignal` for one-dimensional (time) signals.
- `TimeFrequencySignal` which stores two-dimensional signals where the first dimension relates to time (but can be, e.g., a frame index) and the second to the frequency channel. These signals include as an additional property a vector of channel center frequencies `cfHz`. Signals of such form are obtained from requesting, for example, `'filterbank'`, `'innerhaircell'`,`'ild'`,...
- `CorrelationSignal` for three-dimensional signals where the third dimension is a lag position. These include also the `cfHz` property as well as a vector of lags (`lags`).
- `ModulationSignal` for three-dimensional signals where the third dimension is a modulation frequency. These include `cfHz` and `modCfHz` (vector of center modulation frequencies) as properties.
- `FeatureSignal` used to store a collection of time-domain signals, each associated to a specific name. Each feature is a single vector, and all of them are arranged as columns of a same matrix. Hence they include an ordered list of features names `fList` that labels each column.

All these classes inherit the parent `Signal` class. Hence they all share the following common "read-only" properties:

- `Label`, which is a "formal" description of the signal, e.g., `'Inner hair-cell envelope'`, used for example when plotting the signal.
- `Name`, which is a nametag unique to each signal type, e.g., `'innerhaircell'`. This name corresponds to the name used for a request to the manager.
- `Dimensions`, which describes in a short string how dimensions are arranged in the signal, e.g., `'nSamples x nFilters'`
- `FsHz`, the sampling frequency of this specific signal. If the signal is framed or downsampled (e.g., like a ratemap or an ILD) this value will be different from the input signal's sampling frequency.
- `Channel`, which states `'left'`, `'right'` or `'mono'`, depending on which channel from the input signal this signal was derived.
- `Data`, an interface object (`circVBufArrayInterface` described earlier) to the circular buffer containing all data. The actual buffer, `Buf` is a `circVBuf` object and a protected property of the signal (not visible to the user).

The `Signal` class defines the following methods that are then shared among children objects:

- A super constructor, which sets up the internal buffer according to the signal dimensions. Each children signal class is calling this super constructor before populating its other properties.
- An `appendChunk` method used to fill the internal buffer.
- A `setData` method used for initializing the internal buffer given some data.
- A `clearData` method for re-initialization.
- The `getSignalBlock` method returning a segment of data of chosen duration, starting from the newest elements.
- The `findProcessor` method which, given a handle to a manager object, will retrieve which processor has computed this specific signal (by comparing it with the `Output` property of each processor, described in Sec. 3.3.1).
- A `getParameters` method which, given a handle to a manager object, will retrieve the list of parameters used in the processing to obtain that signal.

In addition, the `Signal` class defines an abstract `plot` method, which each children should implement. This cannot be defined in the parent class as the plotting routines will be drastically different depending on children signal dimensionality. Children classes therefore only implement their own constructor (which still calls the super-constructor) and their respective plotting routines.

3.2.3 Data objects

Description

Many signal objects are instantiated by the AFE (one per representation involved and per channel). To organize and keep track of them, they are collected in a `dataObject` class. This class inherits the `dynamicprops` MATLAB class (itself inheriting the `handle`) class. This allows to dynamically define properties of the class.

This way, each signal involved in a given session of the AFE framework will be grouped according to its class in a distinct property of the `dataObject`, with name given by the signal `signal.Name` unique nametag. Extra properties of the data object include:

- `bufferSize_s` which is the common duration of all `circVBuf` objects in the signals.
- A flag `isStereo`, which if true will indicate to the data object that all signals come as pairs of left/right channels.

Data objects are constructed by providing an input signal (which can be empty in online scenarios), a mandatory sampling frequency in Hz, a global buffer size (10s by default), and the number of channels of the input (1 or 2). This number of channel is not necessary if an input signal is used as argument in the constructor but needs to be provided otherwise.

The `dataObject` definition includes the following, self-explanatory methods:

- `addSignal(signalToAdd)`
- `clearData`
- `getParameterSummary` returning a list of all parameters used for the computation of all included signal (given a handle to the corresponding manager).
- `play`, provided for user convenience.

Signal organization

As mentioned before, data objects store signal objects. Each class of signal occupies a property in the data object named after the signal `.Name` property. Multiple signals of the same class will be stored as a cell array in that property. In the cell array, the first column is always for the left channel (or mono signal), and the second column for the right channel. If multiple signals of the same type are present (e.g., if the user requested the same representation twice but with a change of parameters), then the corresponding signals are stored in different lines of the array. For instance, for a session where the user requested the inner hair-cell envelope twice, with the second request changing only the way of extracting the envelope (i.e., the parameter `'ihc_method'`), the following data object is created:

```
>> dataObj

dataObj =

  dataObject with properties:

     bufferSize_s: 10
         isStereo: 1
             time: {[1x1 TimeDomainSignal]  [1x1 TimeDomainSignal]}
            input: {[1x1 TimeDomainSignal]  [1x1 TimeDomainSignal]}
        gammatone: {[1x1 TimeFrequencySignal]  [1x1 TimeFrequencySignal]}
    innerhaircell: {2x2 cell}
```

Each signal-related field except `innerhaircell` is a cell array of a single line (one signal), and two columns (for left and right channel). Because the second request from the user included only a change in parameter for the inner hair-cell computation, the same initial `gammatone` signal is used for both, but there are two output `innerhaircell` signals (hence a cell array of two lines) for each channel (hence two columns).

In that case, to distinguish between the two signals and know which one was computed with which set of parameter, one can call the signal's `getParameters` method. Given a handle to the manager object, it will return a list of all parameters used to obtain that signal (including parameters used in intermediate processing steps).

3.3 Processors

Processors are at the core of the AFE. Each processor is responsible for an individual step in the processing, i.e., going from representation A to representation B. They are adapted from existing models documented in the literature such as to allow for block-based (online) processing. This is made possible by keeping track of the information necessary to transition adequately between two chunks of input. The nature of this "information" varies depending on the processor, and we use in the following the term "internal state" of the processor to refer to it. Internal states and online processing compatibility are then assessed in Sec. 3.3.6.

A detailed overview of all processors, with a list of all parameters they accept, is given in chapter 4. Hence this section will focus on the properties and methods shared among every processors, as well as the techniques employed to make processing compatible with chunk-based inputs.

3.3.1 General considerations

As for signal objects, processors make use of inheritance, with a parent `Processor` class. The parent class defines shared properties of the processor, abstract classes that each children must implement, and a couple of methods shared among children.

The motivation behind the implementation of these methods is probably not clear at this stage, but should appear in the following sections. Many of these methods are used in the manager object described later for organising and routing the processing such as to always perform as few operations as needed.

3.3.2 Properties

Each processor shares the properties:

- `Type` - describes formaly the processing performed
- `Input` - handle to input signal object
- `Output` - handle to output signal object
- `isBinaural` - Flag indicating the need of left and right channel as input
- `FsHzIn` - Input signal sampling frequency
- `FsHzOut` - Output signal sampling frequency
- `UpperDependencies` - List of processors that directly depend on this processor
- `LowerDependencies` - List of processors this processor directly depends on
- `Channel` - Audio channel this processor operates on
- `parameters` - Parameter object instance that contains parameter values for this processor

In addition, three private properties are implemented:

- `bHidden` - A flag indicating that the processor should be hidden from the framework. This is used for example for "sub-processors" such as `downSamplerProc`
- `listenToModify` - An event listener for modifications in any lower dependent processor
- `listenToDelete` - An event listener for deletion of any lower dependent processor

3.3.3 Feedback handling

To these two listeners mentioned above correspond two events, `hasChanged` and `isDeleted`. These events are used in connection to feedback as a mean to communicate between processors. When parameters of a processor are modified, it will broadcast a message that will be picked up by its upper dependencies which will then "know" they have to react accordingly (usually by resetting). Connecting events and listeners is done automatically when instantiating a "processing tree". Modifying a parameter is done via the `modifyParameter` method which will broadcast the `hasChanged` message to upper dependencies.

3.3.4 Abstract and shared methods

The parent `Processor` class defines the following abstract methods. Because these methods are children dependent, each processor sub-class `pObj` should then implement them:

- `out = pObj.processChunk(in)`, the core processing method. Returns an output `out` given the input `in`. It will, if necessary, use the internal states of the processor (derived from previous chunk(s) of input) to calculate the output. These internal states should be accordingly updated in this method after the processing was performed. Next sub-section provides more details regarding these internal states.
- `pObj.reset`, that clears the internal states of the processor. To be used e.g., in an offline scenario in between two different input signals.

Some methods are then identical across all processors and are therefore implemented in the parent `Processor` class:

- `getDependentParameter` and `getDependentProperty` recursively recovers the value of a specific parameter (or property) used by `pObj` or by one of its dependencies
- `hasParameters` checks that the processor uses a specific set of parameter values
- `getCurrentParameters` returns a structure of the parameter values currently used by the processor.

3.3.5 Potentially overridden methods

Most processors behave in similar ways with regard to how many inputs and outputs they have, as well as how they connect with their dependencies. However, there can always be exceptions. To provide sufficient code modularity to easily handle these exceptions without changing existing code, heavy use of methods overriding was made. This means that general behaviour for a given method is implemented in the `Processor` super-class, and any children which needs to handle things differently will override this specific method. These methods susceptible to being overridden are the following, in order in which they are called:

- `prepareForProcessing`: Finalise processor initialisation or re-initialise after receiving feedback
- `addInput`: Populate the `Input` property

- `addOutput`: Populate the `Output` property
- `instantiateOutput`: Instantiate an output signal and add it to the data object
- `initiateProcessing`: Calls the processing method, appropriately routing inputs and output signals to the input and output arguments of the `processChunk` method.

Any of these method are then overridden in children that do not behave "normally" (e.g., processors with multiple input or outputs)

3.3.6 `processChunk` method and chunk-based compatibility

General approach

As briefly exposed above, exact computation performed by each processors are taken from published models, and are described individually in chapter 4. However, most of the available implementations are for batch processing, i.e., using one whole input signal at once. To be included in the AFE, these implementations need to be adapted to account for chunk-based processing, i.e., when the input signal is fed to the system in non-overlapping contiguous blocks, or chunks.

Some processors rely on the input only at time t to generate the output at time t. These processors are then compatible as such with chunk-based processing. This is the case for instance for the `itdProc` which given cross-correlation deduces the interaural time differences (ITDs). That is because the processor, at time t, is provided a cross-correlation value as input (which is a function of frequency and lag), and only locates for each frequency the lag value for which the cross-correlation is maximal. There is no influence of past (or future) inputs to provide the output at time t. This is unfortunately not the case for most processors, which output at a given time will be influenced, to different extent, by older input. However, so far, all the processing involved in the AFE is causal, i.e., might depend on past input, but will not depend on future input.

Adapting offline implementations to online is of course case-dependent, and how it was done for each individual processors will not be described here. However the same concept is used for each, and can be related to the *overlap-save* method traditionally used for filtering long signals (or a stream of input signal) with a finite impulse response (FIR) filter. This concept revolves around using an internal buffer to store the input samples of a given chunk that will influence the processing of the next chunk. Because of the causality, these samples will always be at the end of the present chunk. Considering a processor which is in “steady-state” (i.e., has a populated internal buffer) and a new incoming chunk of input signal, the following steps are performed:

1. The buffer is appended in the beginning of the new input chunk. Conceptually, this provides also a chunk of the input signal, but a longer one that starts at an earlier point in time.
2. The input extended in this way is processed following the computations described in literature. If the input is required to have specific dimensions in time (e.g., when windowing is performed), then it is virtually truncated to these dimensions (i.e., input samples falling outside the required dimensions are discarded). The goal is for the output to be as long as possible while still being "valid", i.e., not being influenced by the boundary with the next input chunks. If additional output was generated due to the appended buffer, it is discarded.
3. The buffer is updated to prepare for the next input chunk. This step can vary between processors but the idea is to store in the buffer the end of the current chunk which did not generate output, or which will influence the output of next chunk.

An example: ratemap

A practical example to better illustrate the concepts described above is given in the following. The ratemap is conceptually a "framed" version of an inner hair-cell (IHC) multichannel envelope. The IHC envelope is a two-dimensional representation (time versus frequency), and the ratemap extraction is the same procedure repeated for every frequency channel. Hence the following is described for a single channel. To extract the ratemap, the envelope is windowed by a set of overlapping windows, and its magnitude averaged in each window. This process is adapted to online processing as illustrated in Fig. 3.1

The three above-mentioned steps are followed:

1. The internal buffer (which can be empty, e.g., if first chunk) is appended to the input chunk.
2. This "extended" input is then processed. In that case, it is windowed and the average is taken in each window.
3. The "valid" outputs form the output chunk. Note that the right-most window (dashed line) is not fully covering the signal. Hence the output it would provide is not "valid", since it would also partly depend on the content of the next input chunk. Therefore the section of the signal corresponding to this incomplete window forms the new buffer.

Note that the output chunk could in theory be empty. If the duration of the "extended" input in step 1 is shorter than the duration of the window, then no valid output is produced for this chunk, and the whole extended input will be transferred to the internal buffer.

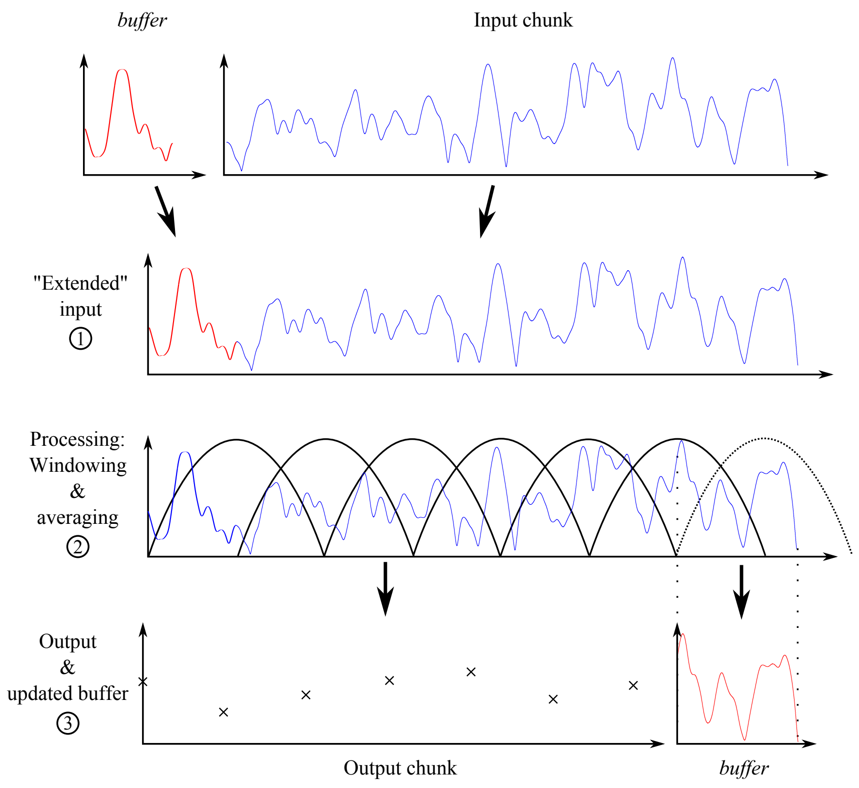
buffer
Input chunk
"Extended" input ①
Processing: Windowing & averaging ②
Output & updated buffer ③
Output chunk
buffer

Figure 3.1: Three steps for simple online windowing, given a chunk of input and an internal buffer.

This is unlikely to happen in practice however.

Particular case for filters

The processing performed by the AFE often involves filtering (e.g., in auditory filterbank processing, inner haircell envelope detection, or amplitude modulation detection). While filtering by FIR filters could in principle be made compatible with chunk-based processing using the principle described above, it will be impractical for filters with long impulse response, and in theory impossible for infinite impulse response (IIR) filters.

For this reason, chunk-based compatibility is managed differently for filtering. In MATLAB's `filter` function, the user can specify initial conditions and can get as optional output the final conditions of the filter delays. These take the form of a vector, of dimension equal to the filter order.

In the AFE, filters are implemented as objects, and encapsulate a private `states` property.

This property simply contains the final conditions of the filter delays, i.e., its internal states after the last processing it performed. If applied to a new input chunk, these states are used as initial condition and are updated after the processing. This will provide a continuous output given a fragmented input.

3.4 Manager

The `manager` class is fundamental in the AFE framework. It is responsible for, from a user request, instantiating the correct processors and signal objects, and linking these signals as inputs/outputs of each processor. In a standard session of the AFE, only a single instance of this class is created. It is with this object that the user interacts.

3.4.1 Processors and signals instantiation

Single request

A standard call to the manager constructor, i.e., with no other argument than a handle to an already created data object `dataObj` will produce an "empty" manager:

```
>> mObj = manager(dataObj)

mObj =

manager with properties:

    Processors: []
     InputList: []
    OutputList: []
           Map: []
          Data: [1x1 dataObject]
```

Empty properties include a list of processors, of input signals, output signals, and a mapping vector that provides a processing order. The `Data` property is simply a handle to the `dataObj` object provided for convenience.

Populating these properties is made via the `addProcessor` method already described in Sec. 2.3. From a given request and an empty manager, instantiating the adequate processors and signals is done following these steps:

1. Get the list of signals needed to compute the user request, using the `getDependencies` function.

2. Flip this list around such as to have the list starting with `'time'`, and ending up with the requested signal. The list then provides the needed signals in the order they should be computed.

3. Loop over the elements of the list. For each signal on the list:

   a) Instantiate a corresponding processor (two if stereo signal)

   b) Instantiate the signal that will contain the output of the processor (two if stereo)

   c) Add the signal(s) to `dataObj`

   d) A handle to the output signal of the previous processor on the list is stored as the current processor's input (in `mObj.InputList` as well as in the processor's `Input` property). If it is the first element of the list, this will link to the original time domain signal.

   e) A handle to the newly instantiated signal is stored similarly as output. This handle is stored further for the next element in the loop.

   f) A handle to the previously instantiated processor is stored in the current processor's `Dependencies` property (possibly empty if first element of the list).

4. Generate a linear mapping (vector of indexes of the processors ordered in increasing processing order).

5. Return a handle to the requested signal to the user.

Once `addProcessor` is called, the properties of the manager will have been populated, e.g.:

```
>> mObj

mObj =

  manager with properties:

    Processors: {3x2 cell}
     InputList: {3x2 cell}
    OutputList: {3x2 cell}
           Map: [1 2 3]
          Data: [1x1 dataObject]
```

Processors are arranged with the same convention as for signals in a data objects: they are stored in a cell array, where the first column is for left (or mono) channel, and second column for right channel. Different lines are for different processors, e.g.:

```
>> mObj.Processors
```

```
ans =

    [1x1 preProc       ]     [1x1 preProc       ]
    [1x1 gammatoneProc]     [1x1 gammatoneProc]
    [1x1 ihcProc       ]     [1x1 ihcProc       ]
```

InputList and OutputList are cell arrays of handles to signal objects. An element in one of them will correspond to the input/output of the processor at the same position in the cell array.

Handling of multiple requests

The above-described process gets more complicated when a request is placed in a non-empty manager (i.e., when multiple requests have been placed). The same steps could be used, and would result in a functioning result. However, this would likely be sub-optimal in terms of computations. If the new request has common elements with representations that are already computed, one need not recompute them.

If correctly implemented, a manager should be able to "branch" the processing, such that only new representations, or representations where a parameter has been changed, are recomputed. Achieving this relies on the findInitProc method of the manager, which is described in more details in the next subsection. This method is passed the same arguments as the addProcessor method, i.e., a request name and a structure of parameters. It will return a handle to an already existing processor in the manager that is exactly computing one of the steps needed for that request. It will return the "highest" already existing step. In other terms, it finds the point in the already existing ordered list of processors where the processing should "branch out" to obtain the newly requested feature. Knowing the processor to start from and updating accordingly the list of signals/processors that need to be instantiated, the same procedure as before can then be used in the addProcessor method.

The findInitProc method

To find an initial processor suitable in a request, this method calls the hasProcessor method of the manager and the hasParameters method of each processor. From a given request, it can obtain a list of necessary processing steps from getDependencies and run the list backwards. For each element of the list, findInitProc "asks" the manager if it has such a processor via its hasProcessor method. If yes, it calls this processor hasParameters method to verify that what the processor computes corresponds to the request. If yes, then it found a suitable initial step. If no, it moves on to the next element in the list and repeats.

3.4.2 Carrying out the processing

As of the current AFE implementation, the processing is linear and the `processChunk` methods of each individual processor are called one after the other when asking the manager to start processing (via its `initiateProcessing` method). The order in which the processors are called is important, as some will take as input what was other's output. This order is stored in the property `Map` of the manager. `Map` is a vector of indexes corresponding to the lines in the `Processors` cell array property of the manager. It is constructed at instantiation of the processors. Conceptually, if there are `N` instantiated processors, the `processChunk` method of the manager `mObj` will call the `initiateProcessing` method of each processor following this loop:

```
for ii = 1:n_proc
        % Get index of current processor
        jj = mObj.Map(ii);

        % Perform the processing by calling initiateProcessing
        mObj.Processors{jj,1}.initiateProcessing;

        if size(mObj.Processors,2) == 2 && ~isempty(mObj.Processors{jj,2})
        mObj.Processors{jj,2}.initiateProcessing;
        end
end
```

Note

Note the difference between indexes `ii` which relate to the processing order (processing first `ii=1` and last `ii=n_proc`) and `jj = mObj.Map(ii)` which relate the processing order with the actual position of the processors in the cell array `mObj.Processors`.

4 Available processors

This chapter presents a detailed description of all processors that are currently supported by the AFE framework. Each processor can be controlled by a set of parameters, which will be explained and all default settings will be listed. Finally, a demonstration will be given, showing the functionality of each processor. The corresponding MATLAB files are contained in the AFE folder `/test` and can be used to reproduce the individual plots. A summary of all demo scripts can be found in appendix A. A full list of available processors can be displayed by using the command `requestList`. An overview of the commands for instantiating processors is given in Sec. 2.3.

4.1 Pre-processing (`preProc.m`)

Prior to computing any of the supported auditory representations, the input signal stored in the data object can be pre-processed with one of the following elements:

1. Direct current (DC) bias removal
2. Pre-emphasis
3. Root mean square (RMS) normalization using an automatic gain control (AGC)
4. Level scaling to a pre-defined sound pressure level (SPL) reference
5. Middle ear filtering

The order of processing is fixed. However, individual stages can be activated or deactivated, depending on the requirement of the user. The output is a time domain signal representation that is used as input to the next processors. Moreover, a list of adjustable parameters is listed in Tab. 4.1.

The influence of each individual pre-processing stage except for the level scaling is illustrated in Fig. 4.1, which can be reproduced by running the script `DEMO_PreProcessing.m`. Panel 1 shows the left and the right ears signals of two sentences at two different levels. The ear signals are then mixed with a sinusoid at 0.5 Hz to simulate an interfering humming noise. This humming can be effectively removed by the DC removal filter, as shown in

Table 4.1: List of parameters related to the auditory representation `'time'`.

Parameter	Default	Description
`pp_bRemoveDC`	`false`	Activate DC removal filter
`pp_cutoffHzDC`	`20`	Cut-off frequency in Hz of the high-pass filter
`pp_bPreEmphasis`	`false`	Activate pre-emphasis filter
`pp_coefPreEmphasis`	`0.97`	Coefficient of first-order high-pass filter
`pp_bNormalizeRMS`	`false`	Activate RMS normalization
`pp_intTimeSecRMS`	`2`	Time constant in s used for RMS estimation
`pp_bBinauralRMS`	`true`	Link RMS normalization across both ear signals
`pp_bLevelScaling`	`false`	Apply level scaling to the given reference
`pp_refSPLdB`	`100`	Reference dB SPL to correspond to the input RMS of 1
`pp_bMiddleEarFiltering`	`false`	Apply middle ear filtering
`pp_middleEarModel`	`'jepsen'`	Middle ear filter model

panel 3. Panel 4 shows the influence of the pre-emphasis stage. The AGC can be used to equalize the long-term RMS level difference between the two sentences. However, if the level difference between both ear signals should be preserved, it is important to synchronize the AGC across both channels, as illustrated in panel 5 and 6. Panel 7 shows the influence of the level scaling when using a reference value of 100 dB SPL. Panel 8 shows the signals after middle ear filtering, as the stapes motion velocity. Each individual pre-processing stage is described in the following subsections.

4.1.1 DC removal filter

To remove low-frequency humming, a DC removal filter can be activated by using the flag `pp_bRemoveDC = true`. The DC removal filter is based on a fourth-order IIR butterworth filter with a cut-off frequency of 20 Hz, as specified by the parameter `pp_cutoffHzDC = 20`.

4.1.2 Pre-emphasis

A common pre-processing stage in the context of automatic speech recognition (ASR) includes a signal whitening. The goal of this pre-processing stage is to roughly compensate for the decreased energy at higher frequencies (e.g. due to lip radiation). Therefore, a first-order FIR high-pass filter is employed, where the filter coefficient `pp_coefPreEmphasis` determines the amount of pre-emphasis and is typically selected from the range between 0.9

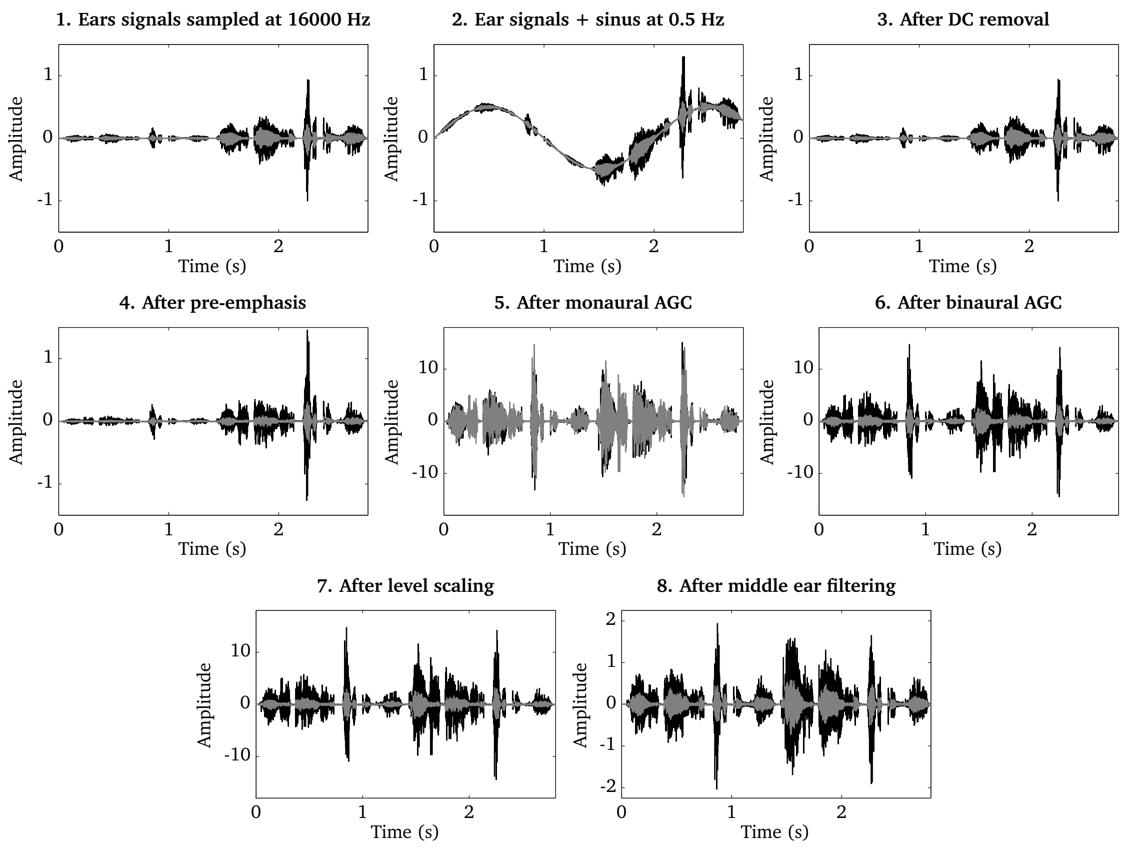

Figure 4.1: Illustration of the individual pre-processing steps. 1) Ear signals consisting of two sentences recorded at different levels, 2) ear signals mixed with a 0.5 Hz humming 3), ear signals after DC removal filter, 4) influence of pre-emphasis filter, 5) monaural RMS normalization, 6) binaural RMS normalization, 7) level scaling and 8) middle ear filtering.

and 1. Here, we set the coefficient to `pp_coefPreEmphasis = 0.97` by default according to (Young *et al.*, 2006). This pre-emphasis filter can be activated by setting the flag `pp_bPreEmphasis = true`.

4.1.3 RMS normalization

A signal level normalization stage is available which can be used to equalize long-term level differences (e.g. when recording two speakers at two different distances). For some applications, such as ASR and speaker identification systems, it can be advantageous to maintain a constant signal power, such that the features extracted by subsequent processors are invariant to the overall signal level. To achieve this, the input signal is normalized by its RMS value that has been estimated by a first-order low-pass filter with a time constant of `pp_intTimeSecRMS = 2`. Such a normalization stage has also been suggested in the context of amplitude modulation spectrogram (AMS) feature extraction (Tchorz and Kollmeier, 2003), which are described in Sec. 4.13. The choice of the time constant is a balance between maintaining the level fluctuations across individual words and allowing the normalization stage to follow sudden level changes.

The normalization can be either applied independently for the left and the right ear signal by setting the parameter `pp_bBinauralRMS = false`, or the processing can be linked across ear signals by setting `pp_bBinauralRMS = true`. When being used in the binaural mode, the larger RMS value of both ear signals is used for normalization, which will preserve the binaural cues (e.g. ITDs and ILDs) that are encoded in the signal. The RMS normalizaion can be activated by the parameter `pp_bNormalizeRMS = true`.

4.1.4 Level reference and scaling

This stage is designed to implement the effect of calibration, in which the amplitude of the incoming digital signal is matched to sound pressure in the physical domain. This operation is necessary when any of the AFE models requires the input to be represented in physical units (such as pascals, see the middle ear filtering stage below). Within the current AFE framework, the dual-resonance non-linear (DRNL) filterbank model requires this signal representation (see Sec. 4.2.2). The request for this is given by setting `pp_bApplyLevelScaling = true`, with a reference value `pp_refSPLdB` in dB SPL which should correspond to the input RMS of 1. Then the input signal is scaled accordingly, if it had been calibrated to a different reference. The default value of `pp_refSPLdB` is 100, which corresponds to the convention used in the work of Jepsen *et al.* (2008). The implementation is adopted from the AMToolbox (Søndergaard and Majdak, 2013).

4.1.5 Middle ear filtering

This stage corresponds to the operation of the middle ear where the vibration from the eardrum is transformed into the stapes motion. The filter model is based on the findings from the measurement of human stapes displacement by Goode *et al.* (1994). Its implementation is adopted from the AMToolbox (Søndergaard and Majdak, 2013), which derives the stapes velocity as the output (Lopez-Poveda and Meddis, 2001, Jepsen *et al.*, 2008). The input is assumed to be the eardrum pressure represented in pascals which in turn assumes prior calibration. This input-output representation in physical units is required particularly when the DRNL filterbank model is used for the basilar membrane (BM) operation, because of its level-dependent nonlinearity, designed based on that representation (see Sec. 4.2.2). When including the middle-ear filtering in combination with the linear gammatone filter, only the simple band-pass characteristic of this model is needed without the need for input calibration or consideration of the input/output units. The middle ear filtering can be applied by setting `pp_bMiddleEarFiltering = true`. The filter data from Lopez-Poveda and Meddis (2001) or from Jepsen *et al.* (2008) can be used for the processing, by specifying the model `pp_middleEarModel = 'lopezpoveda'` or `pp_middleEarModel = 'jepsen'` respectively.

4.2 Auditory filterbank

One central processing element of the AFE is the separation of incoming acoustic signals into different spectral bands, as it happens in the human inner ear. In psychoacoustic modeling, two different approaches have been followed over the years. One is the simulation of this stage by a *linear* filterbank composed of gammatone filters. This linear gammatone filterbank can be considered a standard element for auditory models and has therefore been included in the Two!Ears framework. A computationally more challenging, but at the same time physiologically more plausible simulation of this process can be realized by a *nonlinear* BM model, and we have implemented the DRNL model, as developed by Meddis *et al.* (2001). The filterbank representation is requested by using the nametag `'filterbank'`. The filterbank type can be controlled by the parameter `fb_type`. To select a gammatone filterbank, `fb_type` should be set to `'gammatone'` (which is the default), whereas the DRNL filterbank is used when setting `fb_type = 'drnl'`. Some of the parameters are common to the two filterbank, while some are specific, in which case their value is disregarded if the other type of filterbank was requested. Table 4.2 summarizes all parameters corresponding to the `'filterbank'` request. Parameters specific to a filterbank type are separated by a horizontal line. The two filterbank implementations are described in detail in the following two subsections, along with their corresponding parameters.

4.2.1 Gammatone (`gammatoneProc.m`)

The time domain signal can be processed by a bank of gammatone filters that simulates the frequency selective properties of the human BM. The corresponding Matlab function is adopted from the AMToolbox (Søndergaard and Majdak, 2013). The gammatone filters cover a frequency range between `fb_lowFreqHz` and `fb_highFreqHz` and are linearly spaced on the ERB scale (Glasberg and Moore, 1990). In addition, the distance between adjacent filter center frequencies on the ERB scale can be specified by `fb_nERBs`, which effectively controls the frequency resolution of the gammatone filterbank. There are three different ways to control the center frequencies of the individual gammatone filters:

1. Define a vector with center frequencies, e.g. `fb_cfHz = [100 200 500 ...]`. In this

Table 4.2: List of parameters related to the auditory representation `'filterbank'`.

Parameter	Default	Description
`fb_type`	`'gammatone'`	Filterbank type, `'gammatone'` or `'drnl'`
`fb_lowFreqHz`	`80`	Lowest characteristic frequency in Hz
`fb_highFreqHz`	`8000`	Highest characteristic frequency in Hz
`fb_nERBs`	`1`	Distance between adjacent filters in ERB
`fb_nChannels`	`[]`	Number of frequency channels
`fb_cfHz`	`[]`	Vector of characteristic frequencies in Hz
`fb_nGamma`	`4`	Filter order, `'gammatone'`-only
`fb_bwERBs`	`1.01859`	Filter bandwidth in ERB, `'gammatone'`-only
`fb_lowFreqHz`	`80`	Lowest characteristic frequency in Hz, `'gammatone'`-only
`fb_mocIpsi`	`1`	Ipsilateral MOC factor (0 to 1). Given as a scalar (across all frequency channels) or a vector (individual per frequency channel), `'drnl'`-only
`fb_mocContra`	`1`	Contralateral MOC factor (0 to 1). Same format as `fb_mocIpsi`, `'drnl'`-only
`fb_model`	`'CASP'`	DRNL model (reserved for future extension), `'drnl'`-only

case, the parameters `fb_lowFreqHz`, `fb_highFreqHz`, `fb_nERBs` and `fb_nChannels` are ignored.

2. Specify `fb_lowFreqHz`, `fb_highFreqHz` and `fb_nChannels`. The requested number of filters `fb_nChannels` will be spaced between `fb_lowFreqHz` and `fb_highFreqHz`. The center frequencies of the first and the last filter will match with `fb_lowFreqHz` and `fb_highFreqHz`, respectively. To accommodate an arbitrary number of filters, the spacing between adjacent filters `fb_nERBs` will be automatically adjusted. Note that this changes the overlap between neighboring filters.

3. It is also possible to specify `fb_lowFreqHz`, `fb_highFreqHz` and `fb_nERBs`. Starting at `fb_lowFreqHz`, the center frequencies will be spaced at a distance of `fb_nERBs` on the ERB scale until the specified frequency range is covered. The center frequency of the last filter will not necessarily match with `fb_highFreqHz`.

The filter order, which determines the slope of the filter skirts, is set to `fb_nGamma = 4` by default. The bandwidths of the gammatone filters depend on the filter order and the center frequency, and the default scaling factor for a forth-order filter is approximately `fb_bwERBs = 1.01859`. When adjusting the parameter `fb_bwERBs`, it should be noted that the resulting filter shape will deviate from the original gammatone filter as measured by Glasberg and Moore (1990). For instance, increasing `fb_bwERBs` leads to a broader filter shape. A full list of parameters is shown in Tab. 4.2.

The gammatone filterbank is illustrated in Fig. 4.2, which has been produced by the script `DEMO_Gammatone.m`. The speech signal shown in the left panel is passed through a bank of 16 gammatone filters spaced between 80 and 8000 Hz. The output of each individual filter is shown in the right panel.

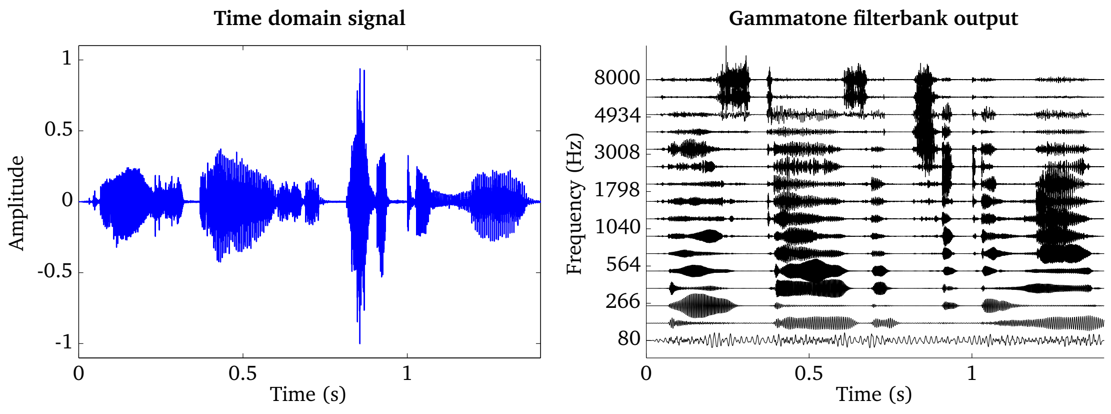

Figure 4.2: Time domain signal (left panel) and the corresponding output of the gammatone processor consisting of 16 auditory filters spaced between 80 and 8000 Hz (right panel).

4.2.2 Dual-resonance non-linear filterbank (`drnlProc.m`)

The DRNL filterbank models the nonlinear operation of the cochlear, in addition to the frequency selective feature of the BM. The DRNL processor was motivated by attempts to better represent the nonlinear operation of the BM in the modelling, and allows for testing the performance of peripheral models with the BM nonlinearity and medial olivo-cochlear (MOC) feedback in comparison to that with the conventional linear BM model. All the internal representations that depend on the BM output can be extracted using the DRNL processor in the dependency chain in place of the gammatone filterbank. This can reveal the implication of the BM nonlinearity and MOC feedback for activities such as speech perception in noise (see Brown *et al.* (2010) for example) or source localisation. It is expected that the use of a nonlinear model, together with the adaptation loops (see Sec. 4.4), will reduce the influence of overall level on the internal representations and extracted features. In this sense, the use of the DRNL model is a physiologically motivated alternative for a linear BM model where the influence of level is typically removed by the use of a level normalization stage (see AGC in Sec. 4.1 for example).

The structure of DRNL filterbank is based on the work of Meddis *et al.* (2001). The frequencies corresponding to the places along the BM, over which the responses are to be derived and observed, are specified as a list of characteristic frequencies `fb_cfHz`. For each characteristic frequency channel, the time domain input signal is passed through linear and nonlinear paths, as seen in Fig. 4.3. Currently the implementation follows the model defined as computational auditory signal-processing and perception (CASP) by Jepsen *et al.* (2008), in terms of the detailed structure and operation, which is specified by the default argument 'CASP' for `fb_model`.

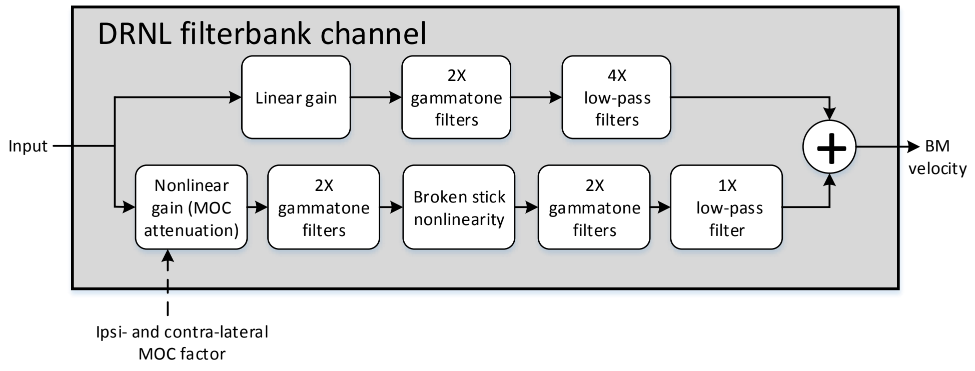

Figure 4.3: DRNL filterbank channel structure, following the CASP model specification (Jepsen *et al.*, 2008) as default, with an additional nonlinear gain stage to receive feedback.

In the CASP model, the linear path consists of a gain stage, two cascaded gammatone filters, and four cascaded low-pass filters; the nonlinear path consists of a gain (attenuation) stage, two cascaded gammatone filters, a 'broken stick' nonlinearity stage, two more cascaded gammatone filters, and a low-pass filter. The outputs at the two paths are then summed as the BM output motion. These sub-modules and their individual parameters (e.g., gammatone filter center frequencies) are specific to the model and hidden to the users. Details regarding the original idea behind the parameter derivation can be found in (Lopez-Poveda and Meddis, 2001), which the CASP model slightly modified to provide a better fit of the output to physiological findings from human cochlear research works.

The MOC feedback is implemented in an open-loop structure within the DRNL filterbank model as the gain factor to be applied to the nonlinear path. This approach is used by Ferry and Meddis (2007), where the attenuation caused by the MOC feedback at each of the filterbank channels is controlled externally by the user. Two additional input arguments are introduced for this feature: `fb_mocIpsi` and `fb_mocContra`. These represent the amount of reflexive feedback through the ipsilateral and contralateral paths, in the form of a factor from 0 to 1 that the nonlinear path input signal is multiplied by in conjunction. Conceptually, `fb_mocIpsi = 1` and `fb_mocContra = 1` would mean that no attenuation is applied to the nonlinear path input, and `fb_mocIpsi = 0` and `fb_mocContra = 0` would mean that the nonlinear path is totally eliminated. Tab. 4.2 summarizes the parameters for the DRNL processor that can be controlled by the user. Note that `fb_cfHz` corresponds to the *characteristic* frequencies and not the *centre* frequencies as used in the gammatone filterbank, although they can have the same values for comparison. Otherwise, the characteristic frequencies can be generated in the same way as the center frequencies for the gammatone filterbank.

Figure 4.4 shows the BM stage output at 1 kHz characteristic frequency using the DRNL processor (on the right hand side), compared to that using the gammatone filterbank (left hand side), based on the right ear input signal shown in panel 1 of Fig. 4.1 (speech excerpt repeated twice with a level difference), scaled down by 10 dB. The plots can be generated by running the script `DEMO_DRNL.m`. It should be noted that the CASP model of DRNL filterbank expects the input signal to be transformed into the middle ear *stapes velocity* before processing. Therefore, for direct comparison of the outputs in this example, the same pre-processing was applied for the gammatone filterbank (stapes velocity was used as the input, through the level scaling and middle ear filtering). It is seen that the level difference between the initial speech component and its repetition is reduced with the nonlinearity incorporated, compared to the gammatone filterbank output, which shows the compressive nature of the nonlinear model responding to input level changes as described earlier.

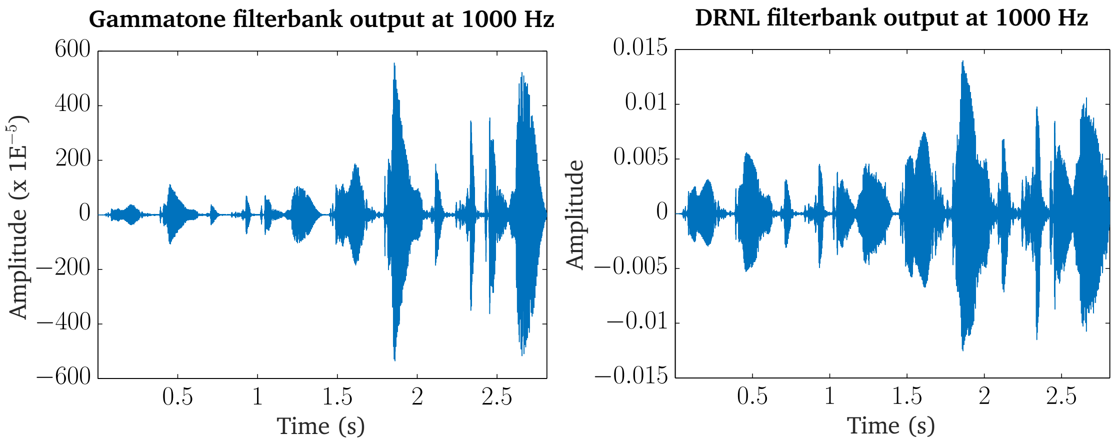

Figure 4.4: The gammatone processor output (left panel) compared to the output of the DRNL processor (right panel), based on the right ear signal shown in panel 1 of Fig. 4.1 scaled down by 10 dB, at 1 kHz center or characteristic frequency. Note that the input signal is converted to the stapes velocity before entering both processors for direct comparison. The level difference between the two speech excerpts is reduced in the DRNL response, showing its compressive nature to input level variations.

4.3 Inner hair-cell (`ihcProc.m`)

The IHC functionality is simulated by extracting the envelope of the output of individual gammatone filters. The corresponding IHC function is adopted from the AMToolbox (Søndergaard and Majdak, 2013). Typically, the envelope is extracted by combining half-wave rectification and low-pass filtering. The low-pass filter is motivated by the loss of phase-locking in the auditory nerve at higher frequencies (Bernstein and Trahiotis, 1996, Bernstein *et al.*, 1999). Depending on the cut-off frequency of the IHC models, it is possible to control the amount of fine-structure information that is present in higher frequency channels. The cut-off frequency and the order of the corresponding low-pass filter vary across methods and a complete overview of supported IHC models is given in Tab. 4.3. A particular model can be selected by using the parameter `ihc_method`.

Table 4.3: List of supported IHC models related to the auditory representation `'innerhaircell'`.

`ihc_method`	Description
`'hilbert'`	Hilbert transform
`'halfwave'`	Half-wave rectification
`'fullwave'`	Full-wave rectification
`'square'`	Squared
`'dau'`	Half-wave rectification and low-pass filtering at 1000 Hz (Dau *et al.*, 1996)
`'joergensen'`	Hilbert transform and low-pass filtering at 150 Hz (Jørgensen and Dau, 2011)
`'breebart'`	Half-wave rectification and low-pass filtering at 770 Hz (Breebaart *et al.*, 2001)
`'bernstein'`	Half-wave rectification, compression and low-pass filtering at 425 Hz (Bernstein *et al.*, 1999)

The effect of the IHC processor is demonstrated in Fig. 4.5, where the output of the gammatone filterbank is compared with the output of an IHC model by running the script `DEMO_IHC.m`. Whereas individual peaks are resolved in the lowest channel of the IHC output, only the envelope is retained at higher frequencies.

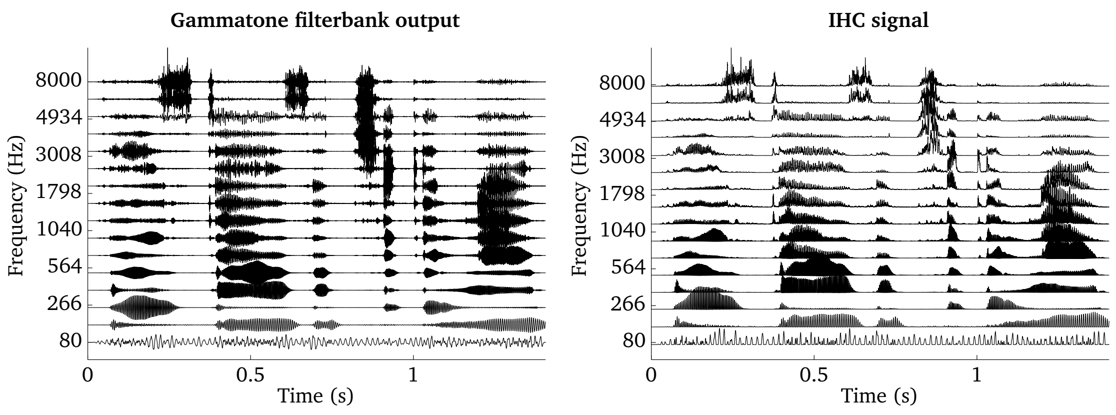

Figure 4.5: Illustration of the envelope extraction processor. BM output (left panel) and the corresponding IHC model output using `ihc_method = 'dau'` (right panel).

4.4 Adaptation (`adaptationProc.m`)

This processor corresponds to the adaptive response of the auditory nerve fibers, in which abrupt changes in the input result in emphasised overshoots followed by gradual decay to compressed steady-state level (Smith, 1977, Smith *et al.*, 1983). The function is adopted from the AMToolbox (Søndergaard and Majdak, 2013). The adaptation stage is modelled as a chain of five feedback loops in series. Each of the loops consists of a low-pass filter with its own time constant, and a division operator (Püschel, 1988, Dau *et al.*, 1996, 1997a). At each stage, the input is divided by its low-pass filtered version. The time constant affects the charging / releasing state of the filter output at a given moment, and thus affects the amount of attenuation caused by the division. This implementation realises the characteristics of the process that input variations which are rapid compared to the time constants are linearly transformed, whereas stationary input signals go through logarithmic compression.

The adaptation processor uses three parameters to generate the output from the IHC representation: `adpt_lim` determines the maximum ratio of the onset response amplitude against the steady-state response, which sets a limit to the overshoot caused by the loops. `adpt_mindB` sets the lowest audible threshold of the input signal. `adpt_tau` are the time constants of the loops. Though the default model uses five loops and thus five time constants, variable number of elements of `adpt_tau` is supported which can vary the number of loops. Some specific sets of these parameters, as used in related studies, are also supported optionally with the `adpt_model` parameter. This can be given instead of the other three parameters, which will set them as used by the respective researchers. Tab. 4.4 lists the parameters and their default values, and Tab. 4.5 lists the supported models. The output signal is expressed in model units (MU) which deviates the input-output relation from a perfect logarithmic transform, such that the input level increment at low level range results in a smaller output level increment than the input increment at higher level range. This corresponds to a smaller just-noticeable level change at high levels than at low levels (Dau *et al.*, 1996). Jepsen *et al.* (2008), with the use of DRNL model for the BM stage, introduces an additional squaring expansion process between the IHC output and the adaptation stage, which transforms the input that comes through the DRNL - IHC processors into an intensity-like representation to be compatible with the adaptation implementation originally designed based on the use of gammatone filterbank. The adaptation processor recognises whether DRNL or gammatone processor is used in the chain and adjusts the input signal accordingly.

The effect of the adaptation processor - the exaggeration of rapid variations - is demonstrated in Fig. 4.6, where the output of the IHC model from the same input as used in the example of Sec. 4.3 (the right panel of Fig. 4.5) is compared to the adaptation output by running the script `DEMO_Adaptation.m`.

Table 4.4: List of parameters related to the auditory representation `'adaptation'`.

Parameter	Default	Description
`adpt_lim`	`10`	Overshoot limiting ratio
`adpt_mindB`	`0`	Lowest audible threshold of the signal (in dB SPL)
`adpt_tau`	`[0.005 0.050 0.129 0.253 0.500]`	Time constants of feedback loops
`adpt_model`	`'' (empty)`	Implementation model (`'adt_dau'`, `'adt_puschel'`, or `'adt_breebart'`): can be used instead of the above three parameters (See Tab. 4.5)

Table 4.5: List of supported models related to the auditory representation `'adaptation'`.

`adpt_model`	Description
`'adt_dau'`	Choose the parameters as in the models of Dau *et al.* (1996, 1997a). This consists of 5 adaptation loops with an overshoot limit of 10 and a minimum level of 0 dB. This is a correction in regard to the model described in Dau *et al.* (1996), which did not use overshoot limiting. The adaptation loops have an exponentially spaced time constants (`adpt_tau=[0.005 0.050 0.129 0.253 0.500]`)
`'adt_puschel'`	Choose the parameters as in the original model (Püschel, 1988). This consists of 5 adaptation loops without overshoot limiting (`adpt_lim=0`). The adapation loops have a linearly spaced time constants (`adpt_tau=[0.0050 0.1288 0.2525 0.3762 0.5000]`).
`'adt_breebaart'`	As `'adt_puschel'`, but with overshoot limiting

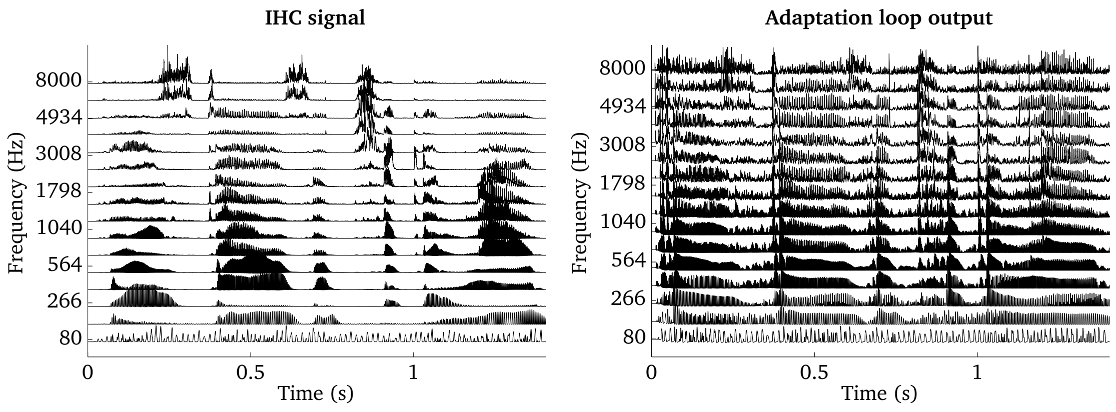

Figure 4.6: Illustration of the adaptation processor. IHC output (left panel) as the input to the adaptation processor and the corresponding output using `adpt_model='adt_dau'` (right panel).

4.5 Auto-correlation (`autocorrelationProc.m`)

Auto-correlation is an important computational concept that has been extensively studied in the context of predicting human pitch perception (Licklider, 1951, Meddis and Hewitt, 1991). To measure the amount of periodicity that is present in individual frequency channels, the auto-correlation function (ACF) is computed in the fast Fourier transform (FFT) domain for short time frames based on the IHC representation. The *unbiased* ACF scaling is used to account for the fact that fewer terms contribute to the ACF at longer time lags. The resulting ACF is normalized by the ACF at lag zero to ensure values between minus one and one. The window size `ac_wSizeSec` determines how well low-frequency pitch signals can be reliably estimated and common choices are within the range of 10 ms –

30 ms.

For the purpose of pitch estimation, it has been suggested to modify the signal prior to correlation analysis in order to reduce the influence of the formant structure on the resulting ACF (Rabiner, 1977). This pre-processing can be activated by the flag `ac_bCenterClip` and the following nonlinear operations can be selected for `ac_ccMethod`: center clip and compress `'clc'`, center clip `'cc'`, and combined center and peak clip `'sgn'`. The percentage of center clipping is controlled by the flag `ac_ccAlpha`, which sets the clipping level to a fixed percentage of the frame-based maximum signal level.

A generalized ACF has been suggested by Tolonen and Karjalainen (2000), where the exponent `ac_K` can be used to control the amount of compression that is applied to the ACF. The conventional ACF function is computed using a value of `ac_K` $= 2$, whereas the function is compressed when a smaller value than 2 is used. The choice of this parameter is a trade-off between sharpening the peaks in the resulting ACF function and amplifying the noise floor. A value of `ac_K` $= 2/3$ has been suggested as a good compromise (Tolonen and Karjalainen, 2000). A list of all ACF-related parameters is given in Tab. 4.6. Note that these parameters will influence the pitch processor, which is described in Sec. 4.11.

Table 4.6: List of parameters related to the auditory representation `'autocorrelation'`.

Parameter	Default	Description
`ac_wname`	`'hann'`	Window type
`ac_wSizeSec`	`0.02`	Window duration (s)
`ac_hSizeSec`	`0.01`	Window step size (s)
`ac_bCenterClip`	`false`	Activate center clipping
`ac_clipMethod`	`'clp'`	Center clipping method (`'clc'`, `'clp'`, or `'sgn'`)
`ac_clipAlpha`	`0.6`	Center clipping threshold within $[0, 1]$
`ac_K`	`2`	Exponent in ACF

A demonstration of the ACF processor is shown in Fig. 4.7, which has been produced by the scrip `DEMO_ACF.m`. It shows the IHC output in response to a 20 ms speech signal for 16 frequency channels (left panel). The corresponding ACF is presented in the upper right panel, whereas the summary auto-correlation function (SACF) is shown in the bottom right panel. Prominent peaks in the SACF indicate lag periods which correspond to integer multiples of the fundamental frequency of the analyzed speech signal. This relationship is exploited by the pitch processor, which is described in Sec. 4.11.

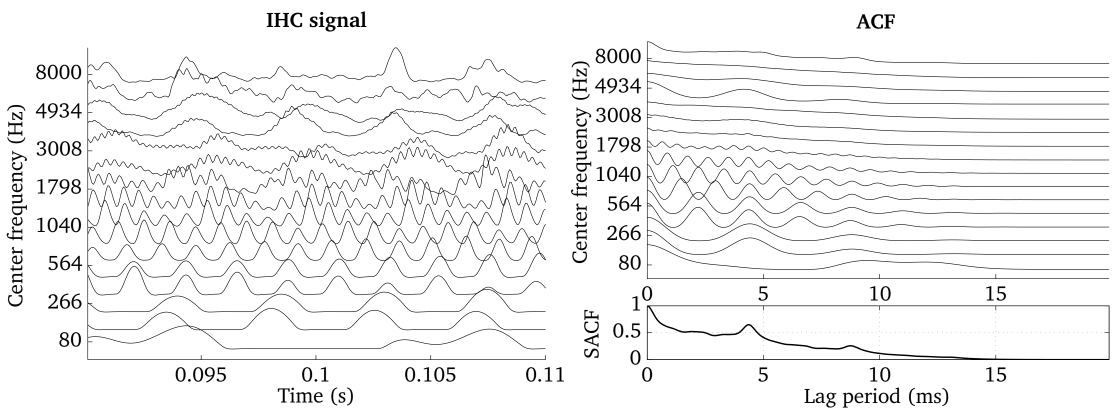

Figure 4.7: IHC representation of a speech signal shown for one time frame of 20 ms duration (left panel) and the corresponding ACF (right panel). The SACF summarizes the ACF across all frequency channels (bottom right panel).

4.6 Ratemap (`ratemapProc.m`)

The ratemap represents a map of auditory nerve firing rates (Brown and Cooke, 1994) and is frequently employed as a spectral feature in computational auditory scene analysis (CASA) systems (Wang and Brown, 2006), ASR (Cooke *et al.*, 2001) and speaker identification systems (May *et al.*, 2012). The ratemap is computed for individual frequency channels by smoothing the IHC signal representation with a leaky integrator that has a time constant of typically `rm_decaySec` = 8 ms. Then, the smoothed IHC signal is averaged across all samples within a time frame and thus the ratemap can be interpreted as an auditory spectrogram. Depending on whether the ratemap scaling `rm_scaling` has been set to `'magnitude'` or `'power'`, either the magnitude or the squared samples are averaged within each time frame. The temporal resolution can be adjusted by the window size `rm_wSizeSec` and the step size `rm_hSizeSec`. Moreover, it is possible to control the shape of the window function `rm_wname`, which is used to weight the individual samples within a frame prior to averaging. The default ratemap parameters are listed in Tab. 4.7.

Table 4.7: List of parameters related to the auditory representation `'ratemap'`.

Parameter	Default	Description
`rm_wname`	`'hann'`	Window type
`rm_wSizeSec`	`0.02`	Window duration (s)
`rm_hSizeSec`	`0.01`	Window step size (s)
`rm_scaling`	`'power'`	Ratemap scaling (`'magnitude'` or `'power'`)
`rm_decaySec`	`0.008`	Leaky integrator time constant (s)

The ratemap is demonstrated by the script `DEMO_Ratemap` and the corresponding plots are presented in Fig. 4.8. The IHC representation of a speech signal is shown in the left panel, using a bank of 64 gammatone filters spaced between 80 and 8000 Hz. The corresponding ratemap representation scaled in dB is presented in the right panel.

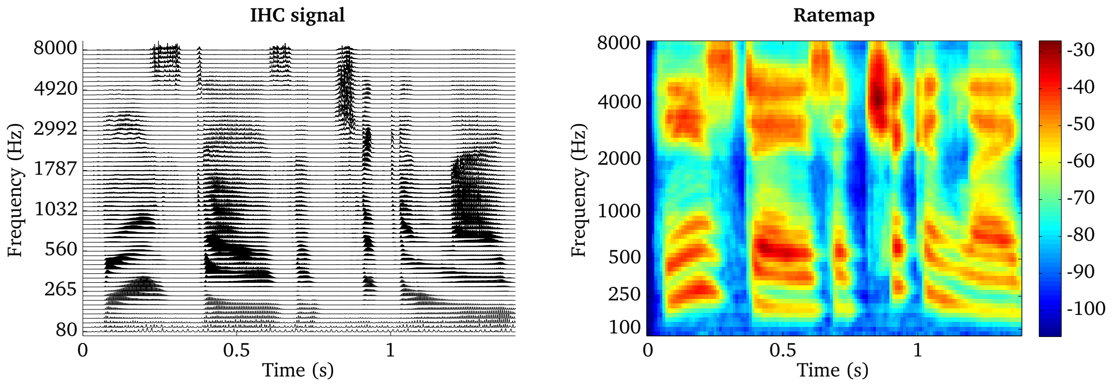

Figure 4.8: IHC representation of s speech signal using 64 auditory filters (left panel) and the corresponding ratemap representation (right panel).

4.7 Spectral features (`spectralFeaturesProc.m`)

In order to characterize the spectral content of the ear signals, a set of spectral features is available that can serve as a physical correlate to perceptual attributes, such as timbre and coloration (Peeters *et al.*, 2011). All spectral features summarize the spectral content of the ratemap representation across auditory filters and are computed for individual time frames. The following 14 spectral features are available:

1. `'centroid'` : The spectral centroid represents the center of gravity of the ratemap and is one of the most frequently-used timbre parameters (Tzanetakis and Cook, 2002, Jensen and Andersen, 2004, Peeters *et al.*, 2011). The centroid is normalized by the highest ratemap center frequency to reduce the influence of the gammatone parameters.

2. `'spread'` : The spectral spread describes the average deviation of the ratemap around its centroid, which is commonly associated with the bandwidth of the signal. Noise-like signals have usually a large spectral spread, while individual tonal sounds with isolated peaks will result in a low spectral spread. Similar to the centroid, the spectral spread is normalized by the highest ratemap center frequency, such that the feature value ranges between zero and one.

3. `'brightness'` : The brightness reflects the amount of high frequency information and is measured by relating the energy above a pre-defined cutoff frequency to the

total energy. This cutoff frequency is set to `sf_br_cf = 1500` Hz by default (Jensen and Andersen, 2004, Peeters *et al.*, 2011). This feature might be used to quantify the sensation of sharpness.

4. `'high-frequency content'` : The high-frequency content is another metric that measures the energy associated with high frequencies. It is derived by weighting each channel in the ratemap by its squared center frequency and integrating this representation across all frequency channels (Jensen and Andersen, 2004). To reduce the sensitivity of this feature to the overall signal level, the high-frequency content feature is normalized by the ratemap integrated across-frequency.

5. `'crest'` : The spectral crest measure (SCM) is defined as the ratio between the maximum value and the arithmetic mean and can be used to characterize the peakiness of the ratemap. The feature value is low for signals with a flat spectrum and high for a ratemap with a distinct spectral peak (Peeters *et al.*, 2011, Lerch, 2012).

6. `'decrease'` : The spectral decrease describes the average spectral slope of the ratemap representation, putting a stronger emphasis on the low frequencies. (Peeters *et al.*, 2011)

7. `'entropy'` : The entropy can be used to capture the peakiness of the spectral representation (Misra *et al.*, 2004). The resulting feature is low for a ratemap with many distinct spectral peaks and high for a flat ratemap spectrum.

8. `'flatness'` : The spectral flattness measure (SFM) is defined as the ratio of the geometric mean to the arithmetic mean and can be used to distinguish between harmonic (SFM is close to zero) and a noisy signals (SFM is close to one) (Peeters *et al.*, 2011).

9. `'irregularity'` : The spectral irregularity quantifies the variations of the logarithmically-scaled ratemap across frequencies (Jensen and Andersen, 2004).

10. `'kurtosis'` : The excess kurtosis measures whether the spectrum can be characterized by a Gaussian distribution (Lerch, 2012). This feature will be zero for a Gaussian distribution.

11. `'skewness'` : The spectral skewness measures the symmetry of the spectrum around its arithmetic mean (Lerch, 2012). The feature will be zero for silent segments and high for voiced speech where substantial energy is present around the fundamental frequency.

12. `'roll-off'` : Determines the frequency in Hz below which a pre-defined percentage `sf_ro_perc` of the total spectral energy is concentrated. Common values for this threshold are between `sf_ro_perc` $= 0.85$ (Tzanetakis and Cook, 2002) and

sf_ro_perc = 0.95 (Scheirer and Slaney, 1997, Peeters *et al.*, 2011). The roll-off feature is normalized by the highest ratemap center frequency and ranges between zero and one. This feature can be useful to distinguish voiced from unvoiced signals.

13. `'flux'` : The spectral flux evaluates the temporal variation of the logarithmically-scaled ratemap across adjacent frames (Lerch, 2012). It has been suggested to be useful for the distinction of music and speech signals, since music has a higher rate of change (Scheirer and Slaney, 1997).

14. `'variation'` : The spectral variation is defined as one minus the normalized correlation between two adjacent time frames of the ratemap (Peeters *et al.*, 2011).

A list of all parameters is presented in Tab. 4.8.

Table 4.8: List of parameters related to the auditory representation `'spectral_features'`.

Parameter	Default	Description
`sf_requests`	`'all'`	List of requested spectral features (e.g. `'flux'`). Type `help spectralFeaturesProc` in the MATLAB command window to display the full list of supported spectral features.)
`sf_br_cf`	`1500`	Cut-off frequency in Hz for brightness feature
`sf_ro_perc`	`0.85`	Threshold (re. 1) for spectral roll-off feature

The extraction of spectral features is demonstrated by the script `Demo_SpectralFeatures.m`, which produces the plots shown in Fig. 4.9. The complete set of 14 spectral features is computed for the speech signal shown in the top left panel. Whenever the unit of the spectral feature was given in frequency, the feature is shown in black in combination with the corresponding ratemap representation.

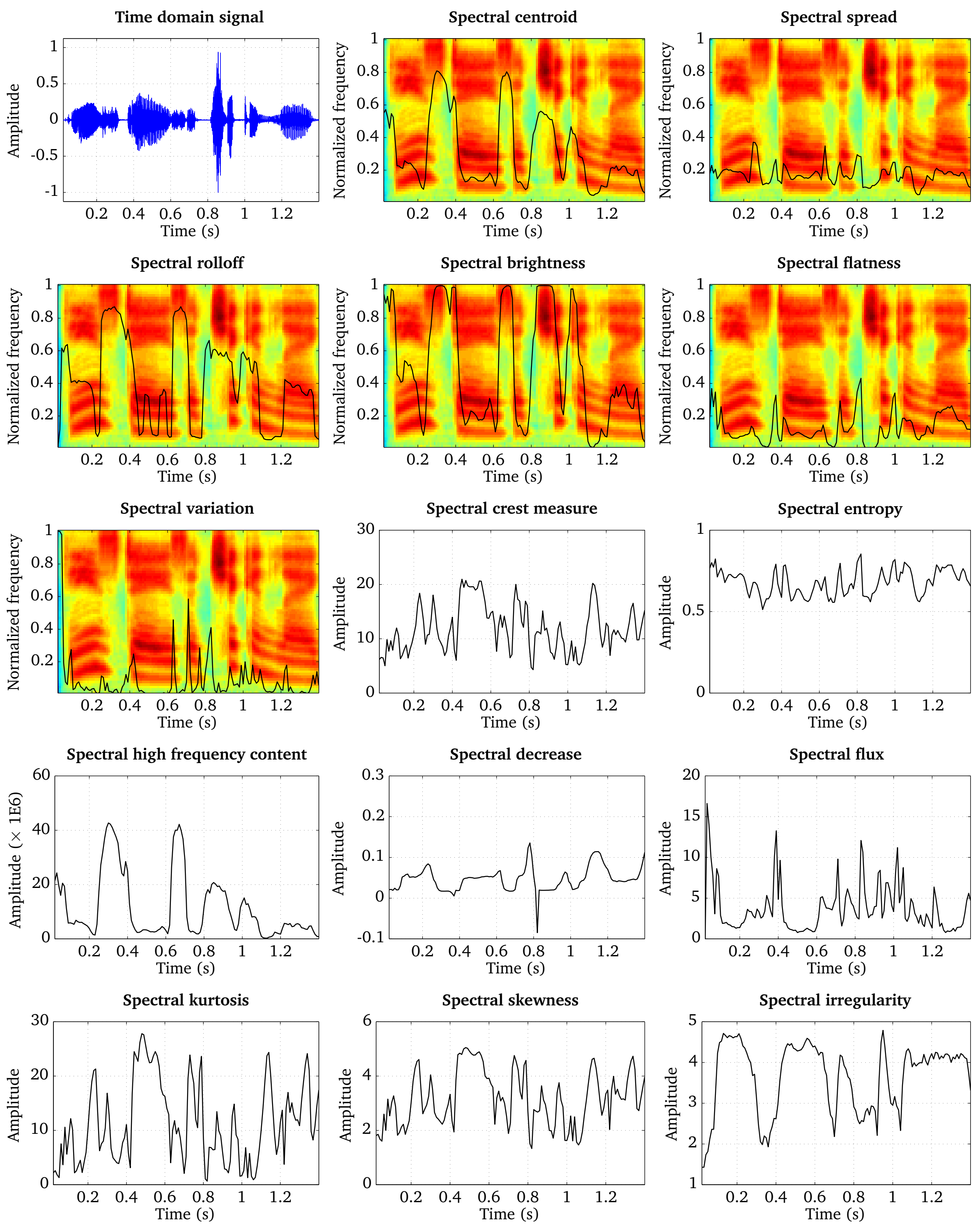

Figure 4.9: Speech signal and 14 spectral features that were extracted based on the ratemap representation.

4.8 Onset strength (`onsetProc.m`)

According to Bregman (1990), common onsets and offsets across frequency are important grouping cues that are utilized by the human auditory system to organize and integrate sounds originating from the same source. The onset processor is based on the ratemap representation, and therefore, the choice of the ratemap parameters, as listed in Tab. 4.7, will influence the output of the onset processor. The temporal resolution is controlled by the window size `rm_wSizeSec` and the step size `rm_hSizeSec`, respectively. The amount of temporal smoothing can be adjusted by the leaky integrator time constant `rm_decaySec`, which reduces the amount of temporal fluctuations in the ratemap. Onset are detected by measuring the frame-based increase in energy of the ratemap representation. This detection is performed based on the logarithmically-scaled energy, as suggested by Klapuri (1999). It is possible to limit the strength of individual onsets to an upper limit, which is by default set to `ons_maxOnsetdB = 30`. A list of all parameters is presented in Tab. 4.9.

Table 4.9: List of parameters related to the auditory representation `'onset_strength'`.

Parameter	Default	Description
`ons_maxOnsetdB`	30	Upper limit for onset strength in dB

The resulting onset strength expressed in dB, which is a function of time frame and frequency channel, is shown in Fig. 4.10. The two figures can be replicated by running the script `DEMO_OnsetStrength.m`. When considering speech as an input signal, it can be seen that onsets appear simultaneously across a broad frequency range and typically mark the beginning of an auditory event.

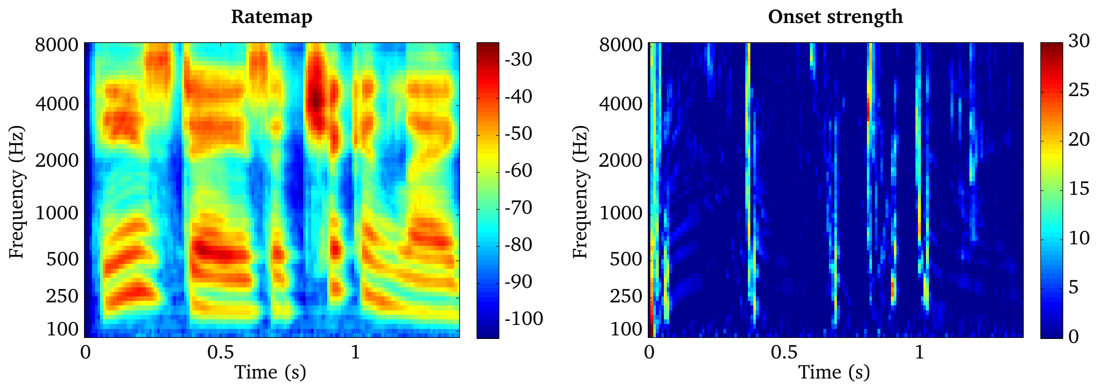

Figure 4.10: Ratemap representation (left panel) of speech and the corresponding onset strength in dB (right panel).

4.9 Offset strength (`offsetProc.m`)

Similarly to onsets, the strength of offsets can be estimated by measuring the frame-based decrease in logarithmically-scaled energy. As discussed in the previous section, the selected ratemap parameters as listed in Tab. 4.7 will influence the offset processor. Similar to the onset strength, the offset strength can be constrained to a maximum value of `ons_maxOffsetdB = 30`. A list of all parameters is presented in Tab. 4.10.

Table 4.10: List of parameters related to the auditory representation `'offset_strength'`.

Parameter	Default	Description
`ofs_maxOffsetdB`	`30`	Upper limit for offset strength in dB

The offset strength is demonstrated by the script `DEMO_OffsetStrength.m` and the corresponding figures are depicted in Fig. 4.11. It can be seen that the overall magnitude of the offset strength is lower compared to the onset strength. Moreover, the detected offsets are less synchronized across frequency.

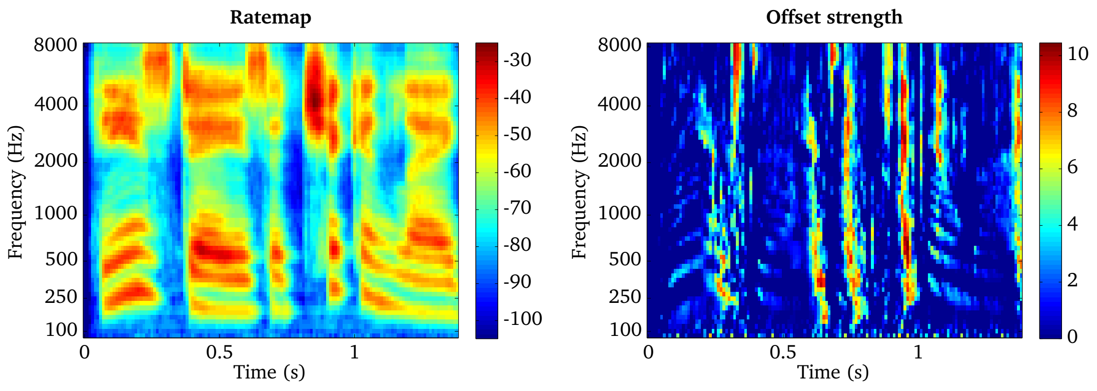

Figure 4.11: Ratemap representation (left panel) of speech and the corresponding offset strength in dB (right panel).

4.10 Binary onset and offset maps (`transientMapProc.m`)

The information about sudden intensity changes, as represented by onsets or offsets, can be combined in order to organize and group the acoustic input according to individual auditory events. The required processing is similar for both onsets and offsets, and is summarized by the term *transient detection*. To apply this transient detection based on the onset strength or offset strength, the user should use the request name `'onset_map'` or `'offset_map'`, respectively. Based on the transient strength which is derived from the corresponding onset strength and offset strength processor (described in Sec. 4.8 and 4.9), a binary decision about transient activity is formed, where only the most salient information is retained. To achieve this, temporal and across-frequency constraints are imposed for the transient information. Motivated by the observation that two sounds are perceived as separated auditory events when the difference in terms of their onset time is in the range of 20 ms – 40 ms (Turgeon *et al.*, 2002), transients are fused if they appear within a pre-defined *time context*. If two transients appear within this time context, only the stronger one will be considered. This time context can be adjusted by `trm_fuseWithinSec`. Moreover, the minimum across-frequency context can be controlled by the parameters `trm_minSpread`. To allow for this selection, individual transients which are connected across multiple time-frequency (T-F) units are extracted using MATLAB'S image labeling tool `bwlabel` . The binary transient map will only retain those transients which consists of at least `trm_minSpread` connected T-F units. The salience of the cue can be specified by the detection thresholds `trm_minStrengthdB`. Whereas this thresholds control the required relative change, a global threshold excludes transient activity if the corresponding ratemap level is below a pre-defined threshold, as determined by `trm_minValuedB`. A summary of all parameters is given in Tab. 4.11.

Table 4.11: List of parameters related to the auditory representation `'onset_map'` and `'offset_map'`.

Parameter	Default	Description
`trm_fuseWithinSec`	`30E-3`	Time constant below which transients are fused
`trm_minSpread`	`5`	Minimum number of connected T-F units
`trm_minStrengthdB`	`3`	Minimum onset strength in dB
`trm_minValuedB`	`-80`	Minimum ratemap level in dB

To illustrate the benefit of selecting onset and offset information, a ratemap representation is shown in Fig. 4.12 (left panel), where the corresponding onsets and offsets detected by the `transientMapProc`, through two individual requests `'onset_map'` and `'offset_map'`, and without applying any temporal or across-frequency constraints are overlaid (respectively in black and white). It can be seen that the onset and offset information is quite noisy. When only retaining the most salient onsets and offsets by applying temporal and across-frequency constraints (right panel), the remaining onsets and offsets can be used as temporal markers, which clearly mark the beginning and the end of individual auditory events.

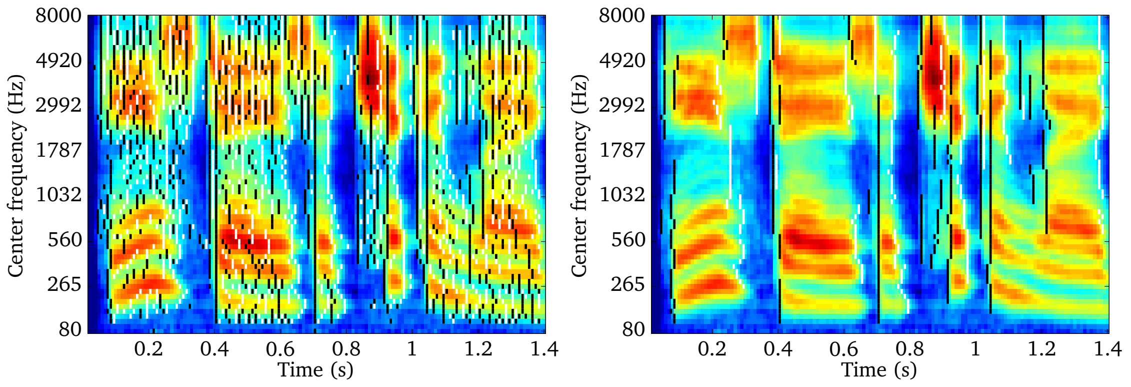

Figure 4.12: Detected onsets and offsets indicated by the black and white vertical bars. The left panels shows all onset and offset events, whereas the right panel applies temporal and across-frequency constraints in order to retain the most salient onset and offset events.

4.11 Pitch (pitchProc.m)

Following Slaney and Lyon (1990), Meddis *et al.* (2001), Meddis and O'Mard (1997), the subband periodicity analysis obtained by the ACF can be integrated across frequency by giving equal weight to each frequency channel. The resulting SACF reflects the strength of periodicity as a function of the lag period for a given time frame, as illustrated in Fig. 4.7. Based on the SACF representation, the most salient peak within the plausible pitch frequency range `p_pitchRangeHz` is detected for each frame in order to obtain an estimation of the fundamental frequency. In addition to the peak position, the corresponding amplitude of the SACF is used to reflect the confidence of the underlying pitch estimation. More specifically, if the SACF magnitude drops below a pre-defined percentage `p_confThresPerc` of its global maximum, the corresponding pitch estimate is considered unreliable and set to zero. The estimated pitch contour is smoothed across time frames by a median filter of order `p_orderMedFilt`, which aims at reducing the amount of octave errors. A list of all parameters is presented in Tab. 4.12. In the context of pitch estimation, it will be useful to experiment with the settings related to the non-linear pre-processing of the ACF, as described in Sec. 4.5.

Table 4.12: List of parameters related to the auditory representation `'pitch'`.

Parameter	Default	Description
`p_pitchRangeHz`	`[80 400]`	Plausible pitch frequency range in Hz
`p_confThresPerc`	`0.7`	Confidence threshold related to the SACF magnitude
`p_orderMedFilt`	`3`	Order of the median filter

The task of pitch estimation is demonstrated by the script `DEMO_Pitch` and the corresponding plots are presented in Fig. 4.13. The pitch is estimated for an anechoic speech signal (top left panel). The corresponding SACF is presented in the top right panel, where each black cross represents the most salient lag period per time frame. The plausible pitch range is indicated by the two white dashed lines. The confidence measure of each individual pitch estimates is shown in the bottom left panel, which is used to set the estimated pitch to zero if the magnitude of the SACF is below the threshold. The final pitch contour is post-processed with a median filter and shown in the bottom right panel. Unvoiced frames, where no pitch frequency was detected, are indicated by `NaN`'s.

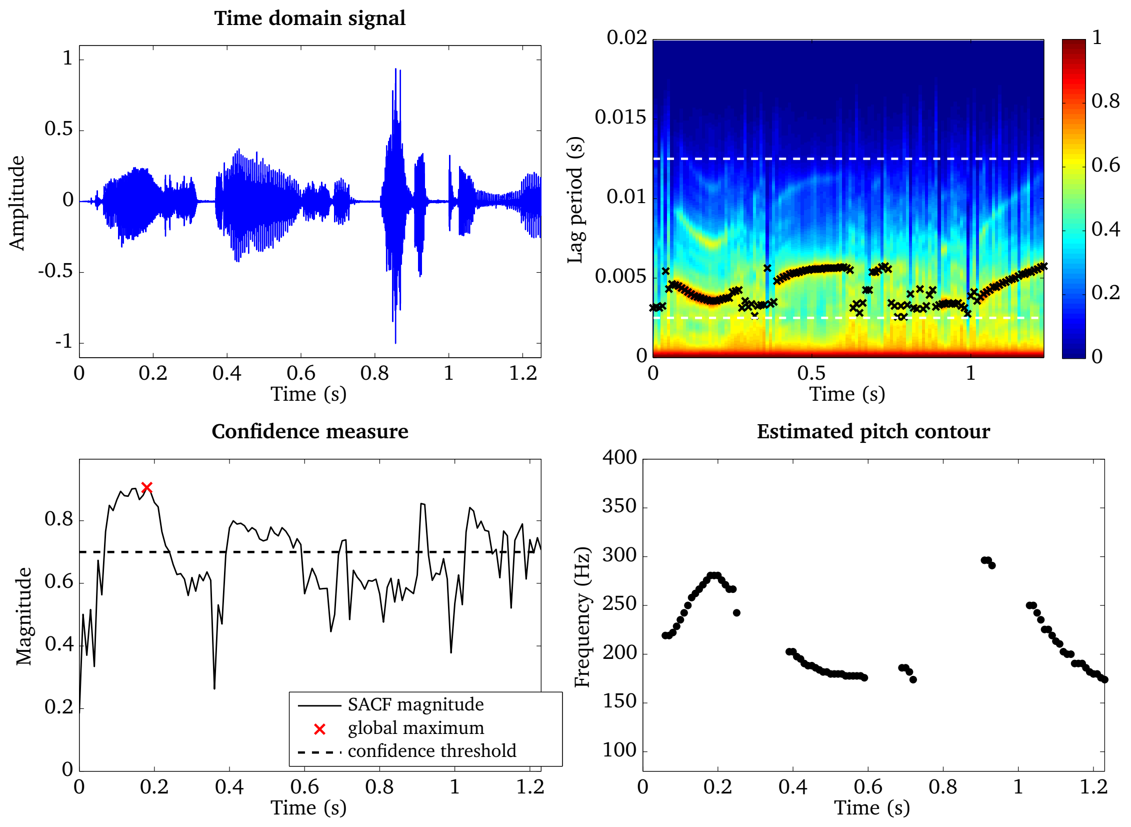

Figure 4.13: Time domain signal (top left panel) and the corresponding SACF (top right panel). The confidence measure based on the SACF magnitude is used to select reliable pitch estimates (bottom left panel). The final pitch estimate is post-processed by a median filter (bottom right panel).

4.12 Medial Olivo-Cochlear (MOC) feedback (`mocProc.m`)

It has now been a well known fact that in the auditory system, an efferent pathway of fibers exists, originating from the auditory neurons in the olivary complex to the outer hair cells (Guinan, 2006). This operates as a top-down feedback path, as opposed to the bottom-up peripheral signal transmission towards the brain, affecting the movement of the basilar membrane in response to the input stimulus. The MOC processor mimics this feedback, particularly originating from the medial part of the olivary complex. In AFE, this feedback is realised by monitoring the output from the ratemap processor which corresponds to the auditory neurons' firing rate, and by controlling accordingly the nonlinear path gain of the DRNL processor which corresponds to the basilar membrane's nonlinear operation. This approach is based on the work of Clark *et al.* (2012), except that the auditory nerve processing model is simplified as the ratemap processor in AFE.

The input to the MOC processor is the time frame-frequency representation from the ratemap processor. This is then converted into an attenuation factor per each frequency channel. The constants for this rate-to-attenuation conversion are internal parameters of the processor, which can be set in accordance with various physiological findings such as those of Liberman (1988). The amplitude relationship was adopted from the work of Clark *et al.* (2012). The time course and delay of the feedback activity, such as in the work of Backus and Guinan (2006), can be approximated by adjusting the leaky integrator time constant `rm_decaySec` and the window step size `rm_hSizeSec` of the ratemap processor.

In addition to this so-called reflexive feedback, realised as a closed-loop operation, the reflective feedback is realised by means of additional control parameters that can be modified externally in an open-loop manner. The two parameters `moc_mocIpsi` and `moc_mocContra` are included for this purpose. Depending on applications, these two can be accessed and adjusted via the Blackboard system, and applied jointly with the reflexive feedback to the DRNL nonlinear path as the final multiplicative gain factor. Tab. 4.13 lists the parameters for the MOC processor, including the above-mentioned two. The other two parameters `moc_mocThresholdRatedB` and `moc_mocMaxAttenuationdB` are specified such that the input level-MOC attenuation relationship is fitted best to the data of Liberman (1988) which is scaled within a range of 0 to 40 dB by Clark *et al.* (2012).

Fig. 4.14 shows, firstly on the left panel, the input-output characteristics of the MOC processor, using on-frequency stimulation from tones at 520Hz and 3980Hz, same as in the work of Liberman (1988). As mentioned above, the relationship between the input level and the MOC attenuation activity through the ratemap representation was derived through curve fitting to the available data set of Liberman (1988), which is also shown on the plot. An example of input signal-DRNL output pair at 40dB input level is shown on

Table 4.13: List of parameters related to the auditory representation `'moc'`.

Parameter	Default	Description
`moc_mocIpsi`	`1`	Ipsilateral MOC feedback factor (0 to 1)
`moc_mocContra`	`1`	Contralateral MOC feedback factor (0 to 1)
`moc_mocThresholdRatedB`	`-180`	Threshold ratemap value for MOC activation in dB
`moc_mocMaxAttenuationdB`	`40`	Maximum possible MOC attenuation in dB

the right panel. The feedback applies an attenuation at the later part of the tone. These plots can be generated by running the script `DEMO_MOC.m`.

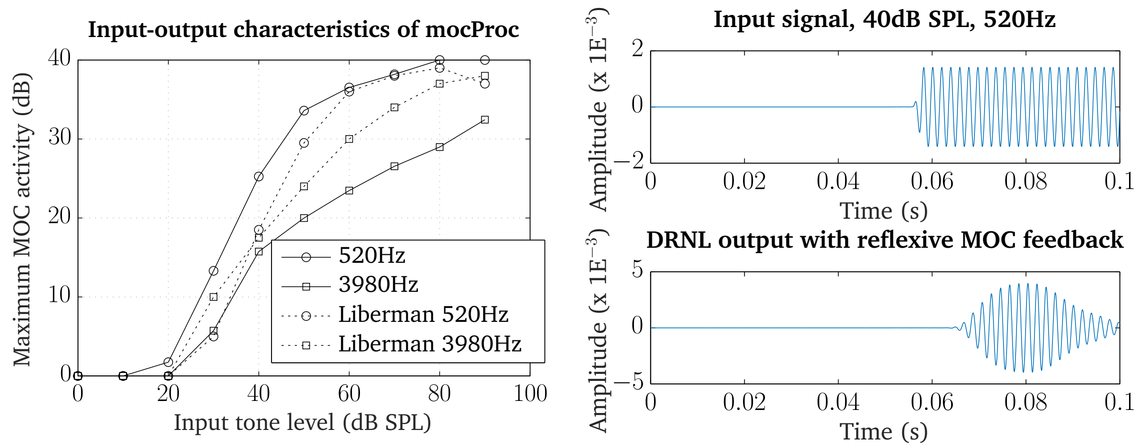

Figure 4.14: Left panel: input-output characteristics of the MOC processor for on-frequency tone stimulus at 520 and 3980Hz. The data set of Liberman (1988), scaled as in the work of Clark *et al.* (2012), is also shown for comparison. Right panel: DRNL processor output (bottom) from a 50-ms tone at 40dB SPL and 520Hz (top), with the reflexive MOC feedback operating.

4.13 Amplitude modulation spectrogram (`modulationProc.m`)

The detection of envelope fluctuations is a very fundamental ability of the human auditory system which plays a major role in speech perception. Consequently, computational models have tried to exploit speech- and noise specific characteristics of amplitude modulations by extracting so-called amplitude modulation spectrogram (AMS) features with linearly-scaled modulation filters (Kollmeier and Koch, 1994, Tchorz and Kollmeier, 2003, Kim *et al.*, 2009, May and Dau, 2013, May and Gerkmann, 2014, May and Dau, 2014a, May *et al.*, 2015). The use of linearly-scaled modulation filters is, however, not consistent with psychoacoustic data on modulation detection and masking in humans (Bacon and Grantham, 1989, Houtgast, 1989, Dau *et al.*, 1997a,b, Ewert and Dau, 2000). As demonstrated by Ewert and Dau

(2000), the processing of envelope fluctuations can be described effectively by a second-order band-pass filterbank with logarithmically-spaced center frequencies. Moreover, it has been shown that an AMS feature representation based on an auditory-inspired modulation filterbank with logarithmically-scaled modulation filters substantially improved the performance of computational speech segregation in the presence of stationary and fluctuating interferers (May and Dau, 2014b). In addition, such a processing based on auditory-inspired modulation filters has recently also been successful in speech intelligibility prediction studies (Jørgensen and Dau, 2011, Jørgensen *et al.*, 2013). To investigate the contribution of both AMS feature representations, the amplitude modulation processor can be used to extract linearly- and logarithmically-scaled AMS features. Therefore, each frequency channel of the IHC representation is analyzed by a bank of modulation filters. The type of modulation filters can be controlled by setting the parameter `ams_fbType` to either `'lin'` or `'log'`. To illustrate the difference between linear linearly-scaled and logarithmically-scaled modulation filters, the corresponding filterbank responses are shown in Fig. 4.15. The linear modulation filterbank is implemented in the frequency domain, whereas the logarithmically-scaled filterbank is realized by a band of second-order IIR butterworth filters with a constant-Q factor of 1. The modulation filter with the lowest center frequency is always implemented as a low-pass filter, as illustrated in the right panel of Fig. 4.15.

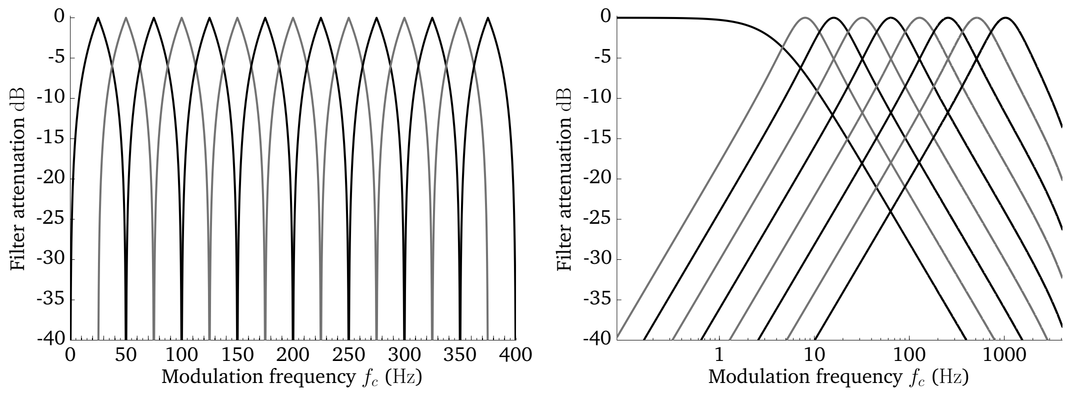

Figure 4.15: Transfer functions of 15 linearly-scaled (left panel) and 9 logarithmically-scaled (right panel) modulation filters.

Similarly to the gammatone processor described in Sect 4.2.1, there are different ways to control the center frequencies of the individual modulation filters, which depend on the type of modulation filters

- `ams_fbType = 'lin'`

  1. Specify `ams_lowFreqHz`, `ams_highFreqHz` and `ams_nFilter`. The requested number of filters `ams_nFilter` will be linearly-spaced between `ams_lowFreqHz`

and `ams_highFreqHz`. If `ams_nFilter` is omitted, the number of filters will be set to 15 by default.

- `ams_fbType = 'log'`
  1. Directly define a vector of center frequencies, e.g. `ams_cfHz = [4 8 16 ...]`. In this case, the parameters `ams_lowFreqHz`, `ams_highFreqHz`, and `ams_nFilter` are ignored.
  2. Specify `ams_lowFreqHz` and `ams_highFreqHz`. Starting at `ams_lowFreqHz`, the center frequencies will be logarithmically-spaced at integer powers of two, e.g. $2^2, 2^3, 2^4 \ldots$ until the higher frequency limit `ams_highFreqHz` is reached.
  3. Specify `ams_lowFreqHz`, `ams_highFreqHz` and `ams_nFilter`. The requested number of filters `ams_nFilter` will be spaced logarithmically as power of two between `ams_lowFreqHz` and `ams_highFreqHz`.

The temporal resolution at which the AMS features are computed is specified by the window size `ams_wSizeSec` and the step size `ams_hSizeSec`. The window size is an important parameter, because it determines how many periods of the lowest modulation frequencies can be resolved within one individual time frame. Moreover, the window shape can be adjusted by `ams_wname`. Finally, the IHC representation can be downsampled prior to modulation analysis by selecting a downsampling ratio `ams_dsRatio` larger than 1. A full list of AMS feature parameters is shown in Tab. 4.14.

Table 4.14: List of parameters related to the auditory representation `'ams_features'`.

Parameter	Default	Description
`ams_fbType`	`'log'`	Filterbank type (`'lin'` or `'log'`)
`ams_nFilter`	`[]`	Number of modulation filters (integer)
`ams_lowFreqHz`	`4`	Lowest modulation filter center frequency in Hz
`ams_highFreqHz`	`1024`	Highest modulation filter center frequency in Hz
`ams_cfHz`	`[]`	Vector of modulation filter center frequencies in Hz
`ams_dsRatio`	`4`	Downsampling ratio of the IHC representation
`ams_wSizeSec`	`32E-3`	Window duration in s
`ams_hSizeSec`	`16E-3`	Window step size in s
`ams_wname`	`'rectwin'`	Window name

The functionality of the AMS feature processor is demonstrated by the script `DEMO_AMS` and the corresponding four plots are presented in Fig. 4.16. The time domain speech signal (top left panel) is transformed into a IHC representation (top right panel) using 23 frequency channels spaced between 80 and 8000 Hz. The linear and the logarithmic AMS feature representations are shown in the bottom panels. The response of the modulation filters are stacked on top of each other for each IHC frequency channel, such that the AMS feature representations can be read like spectrograms. It can be seen that the linear AMS feature representation is more noisy in comparison to the logarithmically-scaled AMS features. Moreover, the logarithmically-scaled modulation pattern shows a much higher correlation with the activity reflected in the IHC representation.

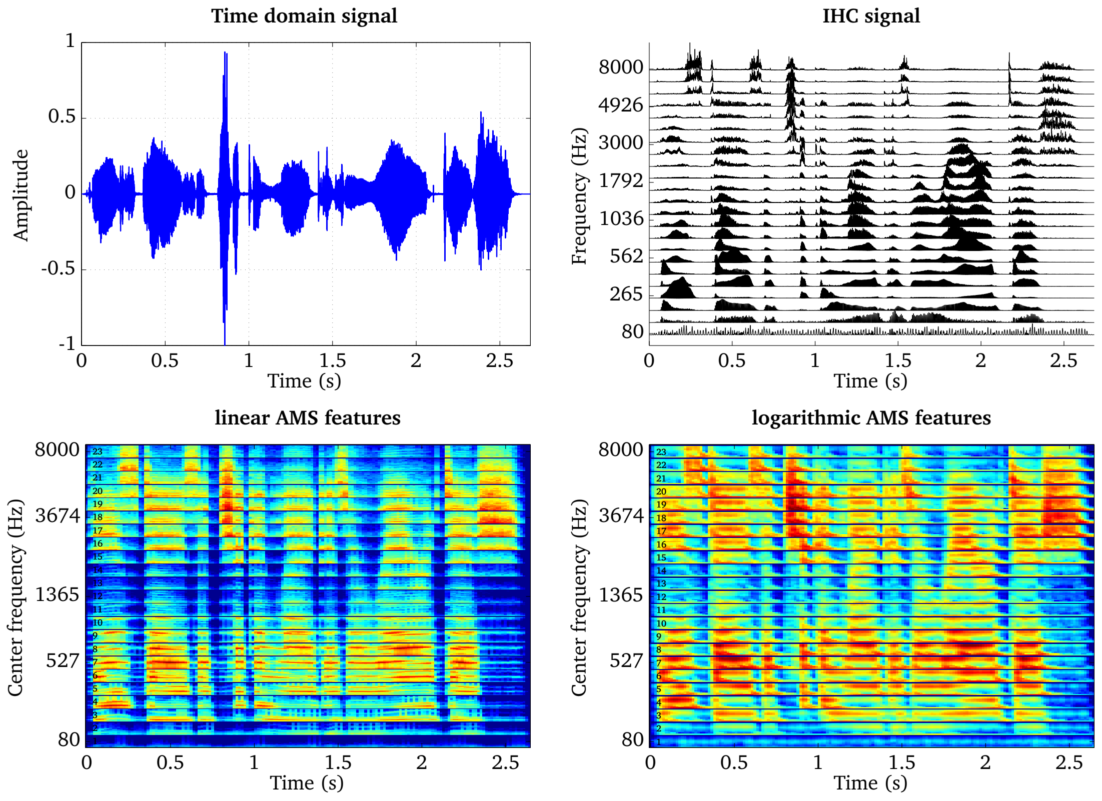

Figure 4.16: Speech signal (top left panel) and the corresponding IHC representation (top right panel) using 23 frequency channels spaced between 80 and 8000 Hz. Linear AMS features (bottom left panel) and logarithmic AMS features (bottom right panel). The response of the modulation filters are stacked on top of each other for each IHC frequency channel, and each frequency channel is visually separated by a horizontal black line. The individual frequency channels, ranging from 1 to 23, are labels at the left hand side.

4.14 Spectro-temporal modulation spectrogram

Neuro-physiological studies suggest that the response of neurons in the primary auditory cortex of mammals are tuned to specific spectro-temporal patterns (Theunissen *et al.*, 2001, Qiu *et al.*, 2003). This response characteristic of neurons can be described by the so-called spectro-temporal receptive field (STRF). As suggested by Qiu *et al.* (2003), the STRF can be effectively modeled by two-dimensional (2D) Gabor functions. Based on these findings, a spectro-temporal filterbank consisting of 41 Gabor filters has been designed by Schädler *et al.* (2012). This filterbank has been optimized for the task of ASR, and the respective real parts of the 41 Gabor filters is shown in Fig. 4.17.

The input is a log-compressed ratemap with a required resolution of 100 Hz, which corresponds to a step size of 10 ms. To reduce the correlation between individual Gabor features and to limit the dimensionality of the resulting Gabor feature space, a selection of representative ratemap frequency channels will be automatically performed for each Gabor filter (Schädler *et al.*, 2012). For instance, the reference implementation based on 23 frequency channels produces a 311 dimensional Gabor feature space.

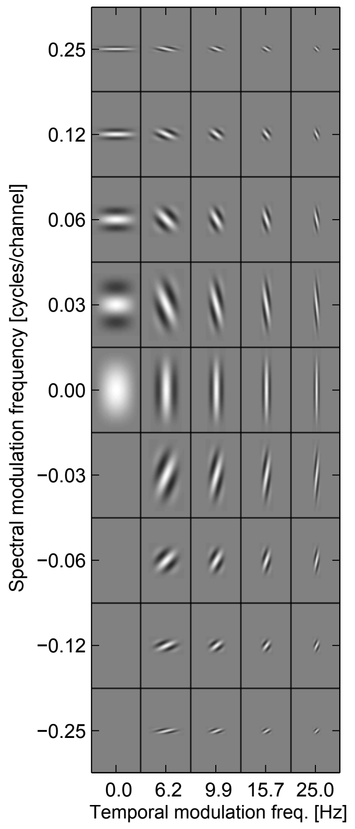

Figure 4.17: Real part of 41 spectro-temporal Gabor filters.

The Gabor feature processor is demonstrated by the script `DEMO_GaborFeatures.m`, which produces the two plots shown in Fig. 4.18. A log-compressed ratemap with 25 ms time frames and 23 frequency channels spaced between 124 and 3657 Hz is shown in the left panel for a speech signal. These ratemap parameters have been adjusted to meet the specifications as recommended in the European telecommunications standards institute (ETSI) standard (ETSI ES 201 108 v1.1.3, 2003). The corresponding Gabor feature space with 311 dimension is presented in the right panel, where vowel transition (e.g. at time frames around 0.2 s) are well captured. This aspect might be particularly relevant for the task of ASR.

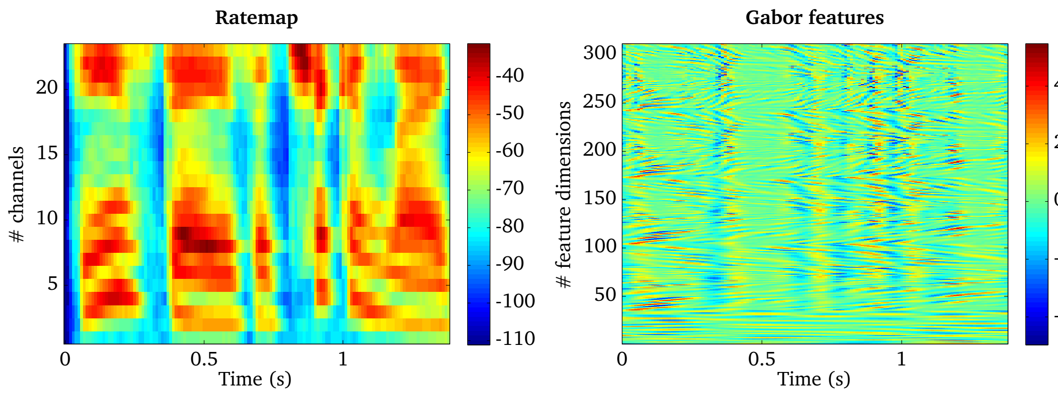

Figure 4.18: Ratemap representation of a speech signal (left panel) and the corresponding output of the Gabor feature processor (right panel).

4.15 Cross-correlation (`crosscorrelationProc.m`)

The IHC representations of the left and the right ear signals is used to compute the normalized cross-correlation function (CCF) in the FFT domain for short time frames of `cc_wSizeSec` duration with a step size of `cc_hSizeSec`. The CCF is normalized by the auto-correlation sequence at lag zero. This normalized CCF is then evaluated for time lags within `cc_maxDelaySec` (e.g., $[-1\,\text{ms}, 1\,\text{ms}]$) and is thus a three-dimensional function of time frame, frequency channel and lag time. An overview of all CCF parameters is given in Tab. 4.15. Note that the choice of these parameters will influence the computation of the ITD and the interaural coherence (IC) processors, which are described in Sec. 4.16 and Sec. 4.18, respectively.

Table 4.15: List of parameters related to the auditory representation `'crosscorrelation'`.

Parameter	Default	Description
`cc_wname`	`'hann'`	Window type
`cc_wSizeSec`	`0.02`	Window duration (s)
`cc_hSizeSec`	`0.01`	Window step size (s)
`cc_maxDelaySec`	`0.0011`	Maximum delay (s) considered in CCF computation

The script `DEMO_Crosscorrelation.m` demonstrates the functionality of the CCF function and the resulting plots are shown in Fig. 4.19. The left panel shows the ear signals for a speech source that is located closer to the right ear. As result, the left ear signal is smaller in amplitude and is delayed in comparison to the right ear signal. The corresponding CCF is shown in the right panel for 32 auditory channels, where peaks are centered around positive time lags, indicating that the source is closer to the right ear. This is even more evident by looking at the summary cross-correlation function (SCCF), as shown in the bottom right panel.

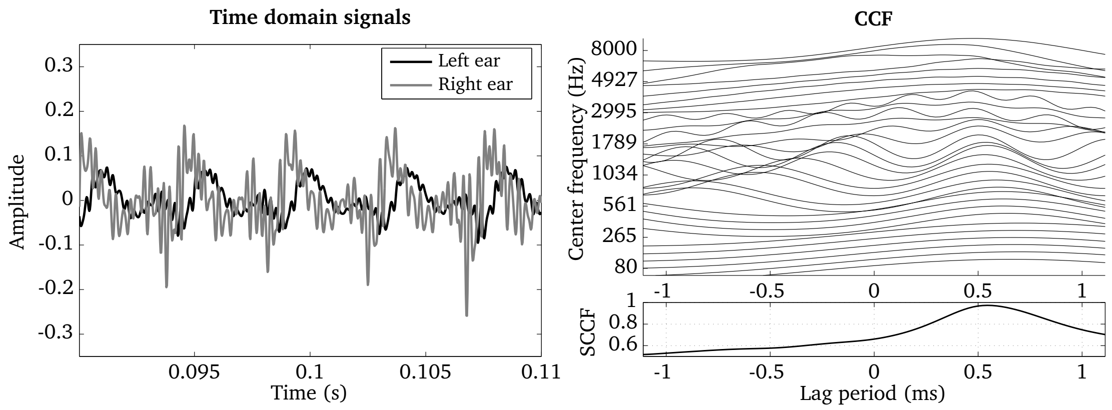

Figure 4.19: Left and right ear signals shown for one time frame of 20 ms duration (left panel) and the corresponding CCF (right panel). The SCCF summarizes the CCF across all auditory channels (bottom right panel).

4.16 Interaural time differences (`itdProc.m`)

The ITD between the left and the right ear signal is estimated for individual frequency channels and time frames by locating the time lag that corresponds to the most prominent peak in the normalized CCF. This estimation is further refined by a parabolic interpolation stage (May *et al.*, 2011, 2013). The ITD processor does not have any adjustable parameters, but it relies on the CCF described in Sec. 4.15 and its corresponding parameters (see Tab. 4.15). The ITD representation is computed by using the request entry `'itd'`.

The ITD processor is demonstrated by the script `DEMO_ITD.m`, which produces two plots as shown in Fig. 4.20. The ear signals for a speech source that is located closer to the right ear are shown in the left panel. The corresponding ITD estimation is presented for each individual T-F unit (right panel). Apart from a few estimation errors, the estimated ITD between both ears is in the range of 0.5 ms for the majority of T-F units.

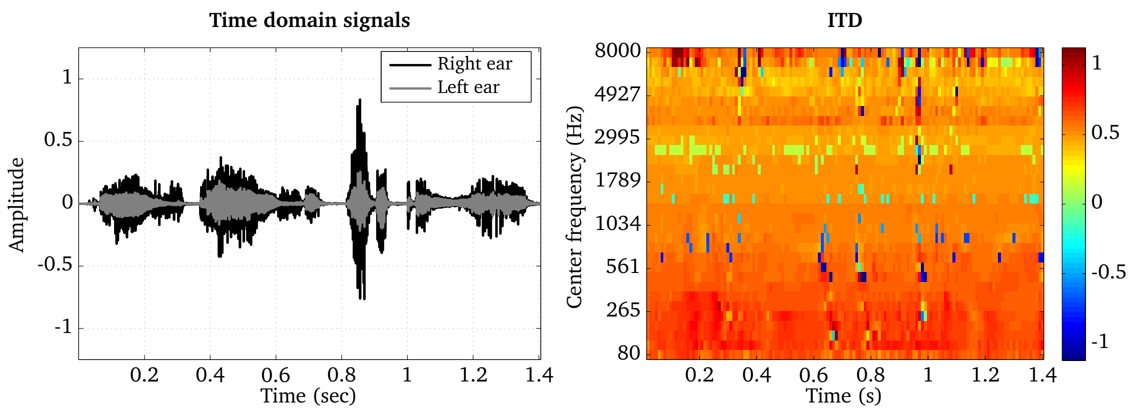

Figure 4.20: Binaural speech signal (left panel) and the estimated ITD in ms shown as a function of time frames and frequency channels.

4.17 Interaural level differences (ildProc.m)

The ILD is estimated for individual frequency channels by comparing the frame-based energy of the left and the right-ear IHC representations. The temporal resolution can be controlled by the frame size `ild_wSizeSec` and the step size `ild_hSizeSec`. Moreover, the window shape can be adjusted by the parameter `ild_wname`. The resulting ILD is expressed in dB and negative values indicate a sound source positioned at the left-hand side, whereas a positive ILD corresponds to a source located at the right-hand side. A full list of parameters is shown in Tab. 4.16.

Table 4.16: List of parameters related to the auditory representation `'ild'`.

Parameter	Default	Description
`ild_wSizeSec`	`'20E-3'`	Window duration in s
`ild_hSizeSec`	`10E-3`	Window step size in s
`ild_wname`	`'hann'`	Window name

The ILD processor is demonstrated by the script `DEMO_ILD.m` and the resulting plots are presented in Fig. 4.21. The ear signals are shown for a speech source that is more closely located to the right ear (left panel). The corresponding ILD estimates are presented for individual T-F units. It is apparent that the ILDs change considerably as a function of the center frequency. Whereas hardly any ILDs are observed for low frequencies, a strong influence can be seen at higher frequencies where ILDs can be as high as 30 dB.

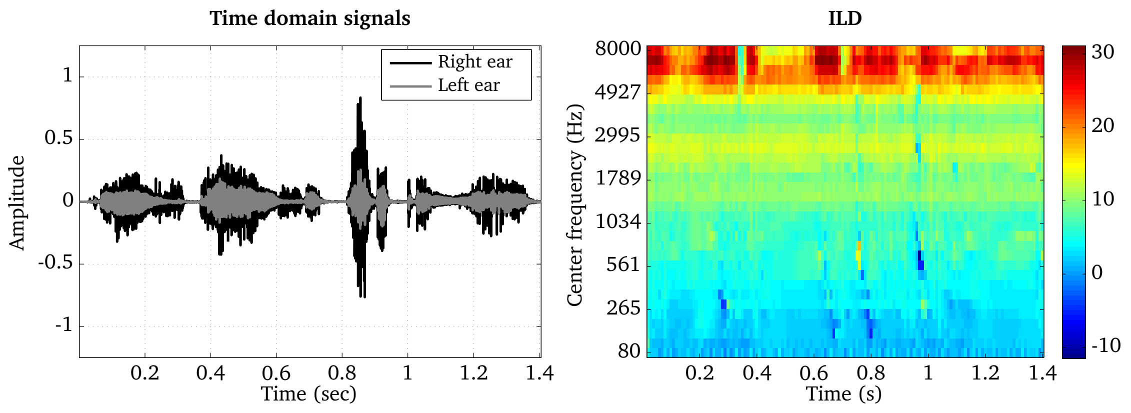

Figure 4.21: Binaural speech signal (left panel) and the estimated ILD in dB shown as a function of time frames and frequency channels.

4.18 Interaural coherence (`icProc.m`)

The IC is estimated by determining the maximum value of the normalized CCF. It has been suggested that the IC can be used to select T-F units where the binaural cues (ITDs and ILDs) are dominated by the direct sound of an individual sound source, and thus, are likely to reflect the true location of one of the active sources (Faller and Merimaa, 2004). The IC processor does not have any controllable parameters itself, but it depends on the settings of the CCF processor, which is described in Sec. 4.15. The IC representation is computed by using the request entry `'ic'`.

The application of the IC processor is demonstrated by the script `DEMO_IC`, which produces the following four plots shown in Fig. 4.22. The top left and bottom left panels show the anechoic and reverberant speech signal, respectively. It can be seen that the time domain signal is smeared due to the influence of the reverberation. The IC for the anechoic signal is close to one for most of the individual T-F units, which indicates that the corresponding binaural cues are reliable. In contrast, the IC for the reverberant signal is substantially lower for many T-F units, suggesting that the corresponding binaural cues might be unreliable due to the impact of the reverberation.

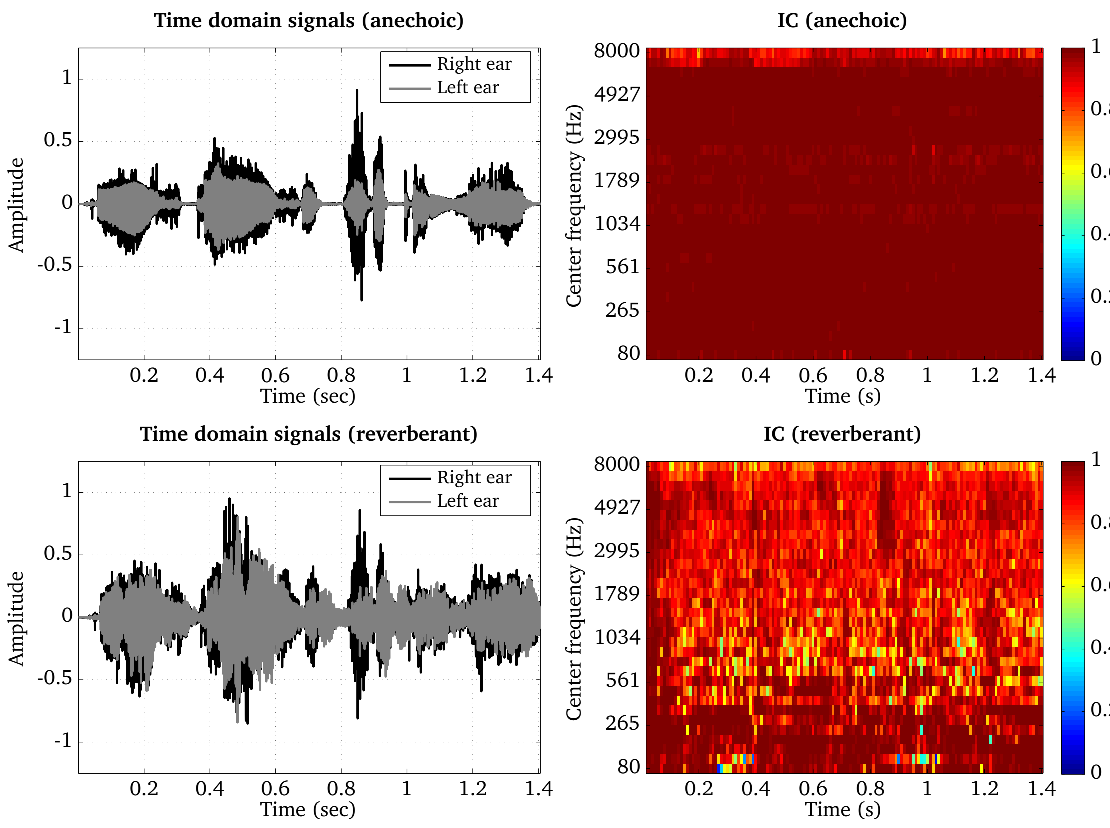

Figure 4.22: Time domain signals and the corresponding interaural coherence as a function of time frames and frequency channels estimated for a speech signal in anechoic and reverberant conditions. Anechoic speech (top left panel) and the corresponding IC (top right panel). Reverberant speech (bottom left panel) and the corresponding IC (bottom right panel).

4.19 Precedence effect (`precedenceProc.m`)

The precedence effect describes the ability of humans to fuse and localize the sound based on the first-arriving parts, in the presence of its successive version with a time delay below an echo-generating threshold (Wallach *et al.*, 1949). The effect of the later-arriving sound is suppressed by the first part in the localization process. The precedence effect processor in AFE models this, with the strategy based on the work of Braasch (2013). The processor detects and removes the lag from a binaural input signal with a delayed repetition, by means of an autocorrelation mechanism and deconvolution. Then it derives the ITD and ILD based on these lag-removed signals.

The input to the precedence effect processor is a binaural time-frequency signal chunk from the gammatone filterbank. Then for each chunk a pair of ITD and ILD values is calculated as the output, by integrating the ITDs and ILDs across the frequency channels according to the weighted-image model (Stern *et al.*, 1988), and through amplitude-weighted summation. Since these ITD/ILD calculation methods of the precedence effect processor are different from what are used for the AFE ITD and ILD processors, the AFE ITD and ILD processors are not connected to the precedence effect processor. Instead the steps for the correlation analyses and the ITD/ ILD calculation are coded inside the processor as its own specific techniques. Tab. 4.17 lists the parameters needed to operate the precedence effect processor.

Table 4.17: List of parameters related to the auditory representation `'precedence'`.

Parameter	Default	Description
`prec_wSizeSec`	`20E-3`	Window duration in s
`prec_hSizeSec`	`10E-3`	Window step size in s
`prec_maxDelaySec`	`10E-3`	Maximum delay in s for autocorrelation computation

Fig. 4.23 shows the output from a demonstration script `DEMO_precedence.m`. The input signal is a 800-Hz wide bandpass noise of 400 ms length, centered at 500 Hz, mixed with a reflection that has a 2-ms delay, and made binaural with an ITD of 0.4 ms and a 0-dB ILD. During the processing, windowed chunks are used as the input, with the length of 20 ms. It can be seen that after some initial confusion, the processor estimates the intended ITD and ILD values as more chunks are analyzed.

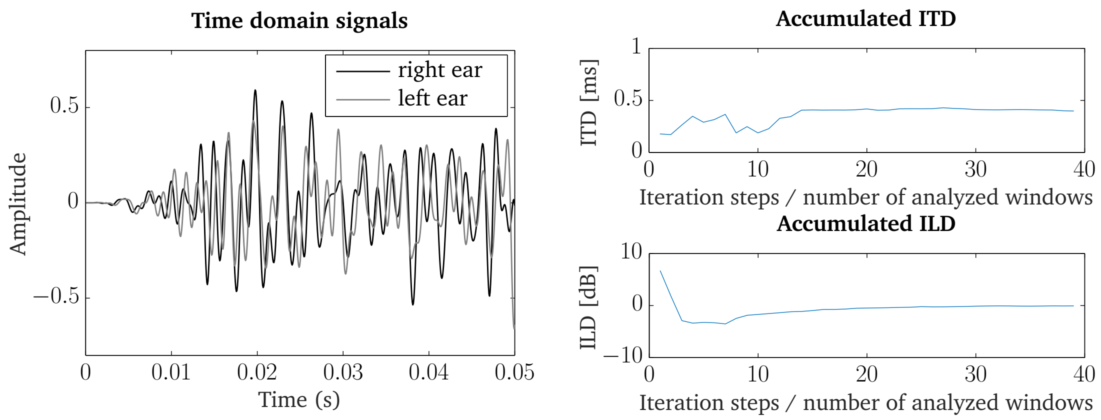

Figure 4.23: Left panel: band-pass input noise signal, 400 ms long (only the first 50 ms is shown), 800 Hz wide, centered at 500 Hz, mixed with a reflection of a 2-ms delay, and made binaural with an ITD of 0.4 ms and ILD of 0 dB. Right panel: estimated ITD ILD shown as a function of time frames.

5 Add your own processors

The auditory front-end (AFE) framework has been designed in such a way that it can be easily upgraded. To add a new processor, write its class definition in a new `.m` file and add it to the `/src/Processors` folder. If correctly written, the processor should be automatically detected by the framework and be ready to use. This section documents in details how to correctly write the class definition of a new processor. It is highly recommended to look into the definition of existing processors to get a grasp of how classes are defined and written in Matlab. In the following, we will sometimes refer to a particular existing processor to illustrate some aspects of the implementation.

Note

- The following descriptions are exhaustive, and adding a processor to the framework is actually easier than the length of this page suggests!
- This tutorial is written assuming limited knowledge about object-oriented programming using Matlab. Hence most object-orientated programming (OOP) concepts involved are briefly explained.
- You can base your implementation on the available `templateProc.m` file which contains a pre-populated list of properties and methods. Simply copy the file, rename it to your processor name, and follow the instructions.

5.1 Check-list for adding a new processor

To write the class definition for a new processor such that it will be recognised and properly integrated, one has to follow these steps:

1. Set up the specific properties of the processor class
2. Implement the processor's static methods
3. Implementing parameters "getter" methods

4. Implement the processor constructor
5. Take a break and test your implementation
6. Implement the core processing method
7. Override parent methods (optional)
8. Allowing alternative processing options (optional)
9. Add a new type of signal (optional)
10. Final testing

5.2 Getting started and setting up processor properties

The properties of an object are a way to store data used by the object. There are two types of properties for processors, those which:

- store all the parameters needed to integrate the processor into the framework (e.g., the sampling frequency on which it operates, the number of inputs/outputs, ...)
- store parameter values which are used in the actual processing

When writing the class definition for a new processor, it is only necessary to implement the latter: parameters which are needed in the computation. All parameters needed for the integration of the processor in the framework are already defined in the parent `Processor` class. Your new processor should inherit this parent class in order to automatically have access to the properties and methods of the parent class. Inheritance in Matlab is indicated by the command `< nameOfParentClass` following the name of your new class in the first line of its definition.

The new processor class definition should be saved in a `.m` file that has the same name as the defined class. In the example below, that would be `myNewProcessor.m`.

There are usually two categories of properties to be implemented for a new processor: external (user-controlled) parameters and internal parameters necessary for the processor but which do not need to be known to the "outside world".

Note

Only the two types of properties below have been used so far in every processor implementation. However, it is fine to add more if needed for your new processor.

5.2.1 External parameters controllable by the user

External parameters are directly related to the parameters the user can access and change. The actual values for these are stored in a specific object accessible via the `.parameters` property of the processor. Defining them as individual properties seems redundant, and is therefore optional. However it can be very convenient in order to simplify the access to the parameter value and to make your code more readable.

Instead of storing an actual value, the corresponding processor property should only point to a value in the `.parameters` object. This will avoid having two different values for the same parameter. To do this, external parameters should be defined as a set of dependent properties. This is indicated by the `Dependent = true` property attribute. If a property is set to `Dependent`, then a corresponding "getter" method has to be implemented for it. This will be developed in a following section. For example, if your new processor has two parameters, `parA` and `parB`, you can define these as properties as follow:

```
classdef myNewProcessor < Processor

    properties (Dependent = true)
            parA;
            parB;
    end

    %...

end
```

This will allow easier access to these values in your code. For example, `myNewProcessor.parA` will always give the same output as

`myNewProcessor.parameters.map('xx_nameTagOfParameterA')`,

even if the parameter value changes due to feedback. This simplifies greatly the code, particularly when many parameters are involved.

5.2.2 Internal parameters

Internal parameters are sometimes (not always) needed for the functioning of the processor. They are typically used to store internal states of the processor (e.g., to allow continuity in block-based processing), filter instances (if your processor involves filtering), or just intermediate parameter values used to make code more readable.

Because they are "internal" to the processor, these parameters are usually stored as a set of private properties by using the `GetAccess = private` property attributes. This will virtually make the property invisible and inaccessible to all other objects.

5.3 Implementing static methods

Static methods are methods that can be called without an existing instance of an object. In the implementation of processors, they are used to store all the hard-coded information. This can be for example the processor name, the type of signal it accepts as input, or the names and default values of its external parameters. A static method is implemented by defining it in a method block with the `(Static)` method attribute:

```
classdef myNewProcessor < Processor

    % ... Properties and other methods definition

    methods (Static)

            function out = myStaticMethod_1(in)
                    %...
            end

            function out = myStaticMethod_2(in)
                    %...
            end

    end

end
```

Static methods share the same structure and names across processors, so they can easily be copy/pasted from an existing processor and then modified to reflect your new processor. The following three methods have to be implemented.

- `.getDependency()`: Returns the type of input signal by its user request name
- `.getParameterInfo()`: Returns names, default values, and descriptions of external parameters
- `.getProcessorInfo()`: Returns information about the processor as a Matlab structure

As they are used to hard-code and return information, none of these methods accept input arguments.

5.3.1 `getDependency`

This method returns the type of input signal your processor should accept:

```
function name = getDependency()
    name = 'requestNameOfInputSignal';
end
```

where `'requestNameOfInputSignal'` is the request name of the signal that should be used as input. “Request name” corresponds to the request a user would place in order to obtain a particular signal. For example, the inner hair-cell envelope processor requires as input the output of e.g., a gammatone filterbank. The request name for this signal is `'filterbank'` which should therefore be the output of the static method `ihcProc.getDependency()`. You can also check the list of currently valid request names by typing `requestList` in Matlab's command window.

If you are unsure about which name should be used, consider which processor would come directly before your new processor in a processing chain (i.e., the processor your new processor depends on). Say it is named `dependentProc`. Then typing:

```
dependentProc.getProcessorInfo.requestName
```

in Matlab's command window will return the corresponding request name you should output in your `getDependency` method.

5.3.2 `getParameterInfo`

This method hard-codes all information regarding the (external) parameters used by your processor, i.e., lists of their names, default values, and description. These are used to populate the output of the helper script `parameterHelper` and to give a default value to parameters when your processor is instantiated.

The lists are returned as cell arrays of strings (or any other type for the default parameter values). They should follow the same order, such that the n-th member of each of the three lists relate to the same parameter.

Parameter names need not be the same as the parameter property name you defined earlier. This will become apparent in the next section. In fact, names should be changed to at least include a two or three letters prefix that is unique to your new processor. You can make sure it is not already in use by browsing through the output of the `parameterHelper` script.

The method should look like this:

```matlab
function [names,defValues,description] = getParameterInfo()

    names = {'xx_par1','xx_par2','xx_par3'};

    defValues = {0.5, ...
                 [1 2 3 4], ...
                 'someStringValue'};

    description = {'Tuning factor of dummy example (s)',...
                   'Vector of unused frequencies (Hz)',...
                   'Model name (''someStringValue'' or ''anotherValue'')'}

end
```

This dummy example illustrates the following important points:

- Use a unique prefix in the name of the parameters (`xx_` above) that abbreviates the name or task of the processor.
- Find a short, but self-explanatory parameter name (*not* like `parX` above). If it makes sense, you can re-use the same name as a parameter involved in another processor. The prefix will make the name unique.
- Default values can be of any type (e.g., float number, array, strings,...)
- Descriptions should be as short as possible while still explanatory. Mention if applicable the units or the different alternatives.

5.3.3 `getProcessorInfo`

This method stores the properties of the processor that are needed to integrate it in the framework. It outputs a structure with the following fields:

- `.name`: A short, self-explanatory name for the processor
- `.label`: A name for the processor that is used as a label. It can be the same as .name if that is sufficient, or a bit longer if needed.
- `.requestName`: The name tag of the request that a user should input when calling the `.addProcessor` method of the manager. This has to be a valid Matlab name (e.g., it *cannot* include spaces).
- `.requestLabel`: A longer name for the signal this processor produces, used e.g., as plot labels.

- `outputType`: The type of signal object (name of the class) this processor produces. If none of the existing signals in the framework are suitable, you will need to implement a new one.
- `isBinaural`: Set to 0 if your processor operates on a single channel (e.g., an auditory filterbank) or to 1 if it *needs* a binaural input (e.g., the inter-aural level differences processor). If your processor can operate on both mono and stereo signals (such as the pre-processor `preProc.m`), set it to 2.

Your method should initialise the structure that will be returned as output and give a value to all of the above-mentioned fields:

```
%...

function pInfo = getProcessorInfo

  pInfo = struct;

  pInfo.name = 'MyProcessor';
  pInfo.label = 'Processor doing things';
  % etc...

end
```

5.4 Implementing parameters "getter" methods

As described in an earlier section, external parameters of the processor, i.e., those that can be modified by the user, are implemented as `Dependent` properties of your processor class. For your implementation to be valid, a "getter" method needs to be implemented for each of these parameters. If not, Matlab will generate an error when trying to access that parameter value. If a property is set as Dependent, then its getter method will be called whenever the program tries to access that property. In general, this can be useful for a property that *depends* on others and that need to be recomputed whenever accessed. In the present case, we will set the getter method to read the corresponding parameter value in the parameter object associated with your processor. If the value of the parameter has changed throughout the processing (e.g., in response to feedback), then we are sure to always get the updated value.

"Getter" methods for parameters are implemented without any method attribute and always follow the same structure. Hence they can easily be copy/pasted and adjusted:

```
methods

    function value = get.parName(pObj)
```

```
            value = pObj.parameters.map('xx_parNameTag')
    end

    % ... implement one get. method for each parameter

end
```

In the above example, `parName` is the name of the parameter as a dependent property of your processor class, and `xx_parNameTag` is the name of the parameter defined in the static `.getParameterInfo` method. `pObj` represents an instance of your processor class, it does not need to be changed across methods.

5.5 Implement the processor constructor

For any possible application, every class should implement a very specific method: a class constructor. A class constructor is a function that has the exact same name as your class. It can take any combination of input arguments but can return only a single output: an “instance” of your class.

In the AFE architecture however, the input arguments to the constructor of all processors have been standardised, such that all processor constructors can be called using the exact same arguments. The input arguments should be (in this order) the sampling frequency of the input signal to the processor and an instance of a parameter object returned e.g. by the script `genParStruct.m`. The constructor's role is then to create an object of the class, and often to initialise all its properties. Most of this initialisation step is the same across all processors (e.g., setting input/output sampling frequencies, indicating the type of processor, ...). Hence all processor constructors rely heavily on the constructor of their parent class (or super-constructor), `Processor(...)` which defines these across-processors operations. This allows to have all this code in one place which reduces the code you have to write for your processor, as well as reducing chances for bugs and increasing maintainability. This concept of “inheritance” will be discussed in a further section.

In practice, this means that the constructor for your processor will be very short:

```
function pObj = myNewProcessor(fs,parObj)
    %myNewProcessor   ... Provide some help here ...

    if nargin<2||isempty(parObj); parObj = Parameters; end
    if nargin<1; fs = []; end

    % Call super-constructor
    pObj = pObj@Processor(fs, fsOut,'myNewProcessor',parObj);

    % Additional code depending on your processor
```

```
11      % ...
12
13  end
```

Note

The constructor method should be placed in a "method" block with no method attributes.

Let us break down the constructor structure line by line:

- Line 1: As stated earlier, all processor constructors take two input and return a single output, your processor instance `pObj`. Matlab restricts all constructors to return a single output. If for any reason you need additional outputs, you would have to place them in a property of your processor instead of a regular output. Input arguments are the **input** sampling frequency, i.e., the sampling frequency of the signal at the input of the processor, and a parameter object parObj.
- Line 2: This is where you will place help regarding how to call this constructor. Because they have a generic form across all processors, you can easily copy/paste it from another processor.
- Lines 4 and 5: An important aspect in this implementation is that the constructor should be called with no input argument and still return a valid instance of the processor, without any error. Hence these two lines define default values for inputs if none were specified.
- Line 8: This line generates a processor instance by calling the class super-constructor. The super-constructor takes four inputs:
  - the **input** sampling frequency `fs`
  - the **output** sampling frequency. If your processor does not modify the sampling rate, then you can replace `fsOut` with `fs`. If the output sampling rate of your processor if **fixed**, i.e., not depending on external parameters, then you can specify it here, in place of `fsOut`. Lastly, if the output sampling rate depends on some external parameters (i.e., susceptible to change via feedback from the user), then you should leave the `fsOut` field empty: `[]`. The output sampling rate will be defined in another method that is called every time feedback is involved.
  - the name of the children processor, here `myNewProcessor`.
  - the parameter object `parObj` already provided as input.

- Line 11: Your processor might need additional initialisation. All extra code should go there. To ensure that no error is generated when calling the constructor with no arguments (which Matlab sometimes does implicitly), the code should be embedded in a `if nargin > 0 ... end` block. Here you can for example initialise buffers or internal properties.

Warning

The initialisation of anything that depends on external parameters (e.g., filters, framing windows, ...) is not performed here on line 11. When parameters change due to feedback, these properties need to be re-initialised. Hence their initialisation is performed in another method that will be described in a following section.

5.6 Preliminary testing

At this stage of the implementation, your processor should be correctly instantiated and recognised by the framework. In some cases (e.g., your processor is a simple single input / single output processor), it might even be correctly integrated and routed to other processors. In any case, now is a good time to take a break from writing code and do some preliminary testing. We will go through a few example tests you can run, describe which problems could arise and suggest how to solve them. Try to run these tests in the order they are listed below, as this will help troubleshooting. They should run as expected before you go further in your implementation.

Note

You will not be able to instantiate your processor before you have written a concrete implementation to **Processor** abstract methods. To carry out the tests below, just write empty **processChunk** and **reset** methods. In this way, Matlab will not complain about trying to instantiate a class that contains abstract methods. The actual implementation of these methods will be described in later sections.

5.6.1 Default instantiation

As mentioned when implementing the constructor, you should be able to get a valid instance of your processor by calling its constructor without any input arguments:

```
>> p = myNewProcessor
```

If this line returns an error, then you have to revise your implementation of the constructor. The error message should indicate where the problem is located, so that you can easily correct it. If your processor cannot be instantiated with no arguments, then it will not be listed as a valid processor.

If on the other hand this line executed without error, then there are two things you should control:

1. The line above (if not ended by a semicolon) should display the visible, public properties of the processor. Check that this list corresponds to the properties you defined in your implementation. The property values should be the default values you have defined in your `getParameterInfo` static method. If a property is missing, then you forgot to list it in the beginning of your class definition (or you defined it as `Hidden` or `Private`). If a value is incorrect, or empty, then it is a mistake in your `getParameterInfo` method. In addition, the `Type` property should refer to the `name` field returned by `getProcessorInfo` static method.
2. Inspect the external parameters of the processor by typing `p.parameters`. This should return a list of all external parameters. Control that all parameters are there and that their default value is correct.

To test that your external properties are indeed dependent, you can change the value of one or more of them directly in your `parameter` processor property and see if that change is reflected in the dependent property. For example if you type:

```
p.parameters.map('xx_par1') = someRandomValue
```

then this should be reflected in the property associated with that parameter.

Note

The input and output frequency properties of your processor, `FsHzIn` and `FsHzOut` are probably incorrect, but that is normal as you did not specify the sampling frequency when calling the constructor with no arguments.

5.6.2 Is it a valid processor?

To test whether your processor is recognised as a valid processor, run the `requestList` script. The signal request name corresponding to your processor should appear in the list (i.e., the name defined in `getProcessorInfo.requestName`). If not (and the previous test did work), then maybe your class definition file is not located in the correct folder. Move it to the `src/Processors` folder. Another possibility is that you made your processor hidden (which should not happen if you followed these instructions). Setting explicitly the `bHidden` property of your processor to `1` will hide it from the framework. This is used in order to allow "sub-processors" in the framework, but it is probably not the case for you here so you should not enable this option.

5.6.3 Are parameters correctly described?

If your processor is properly recognised, then you can call the `parameterHelper` script from the command window. There you should see a new category corresponding to your processor. Clicking on it will display a list of user-controllable parameters for your processor, as well as their descriptions. Feel free to adjust your `getParameterInfo` static method to have a more suitable description.

5.7 Implementing the core processing method

At this stage, and if the previous tests were successfully passed, your processor should be correctly detected by the AFE framework. However, there is still some work to do. In particular, the core of your processor has to be implemented, which performs the processing of the input signal and returns a corresponding output.

This section will provide guidelines as to how to implement that method. However, this task is very dependent on the functionality of a particular processor. You can get insights as to how to perform the signal processing task by looking at the code of the `.processChunk` methods of existing processors.

Note

Some of the challenges in implementing the processing method were already presented in a section of the technical description. It is recommended at that stage to go back and read that section again.

5.7.1 Input and output arguments

The processing method should be called `processChunk` and be placed in a block of methods with no attributes (e.g., following the class constructor). The function takes a single effective input argument, a chunk of input signal and returns a single output argument, the corresponding chunk of output signal. Because it is a non-static method of the processor, an instance of the processor is passed as first input argument. Hence the method definition looks something like this for a monaural single-output processor:

```
function out = processChunk(pObj,in)

  % The signal processing to obtain out from in is written here
  %
  % ...

end
```

Or, for a binaural single-output processor (such as `ildProc`):

```
function out = processChunk(pObj,in_left,in_right)

  % The signal processing to obtain out from in is written here
  %
  % ...

end
```

If your processor is not of one of the two kinds described above, then you are free to use a different signature for your `processChunk` method (i.e., different number of input or output arguments). However, you will then have to override the `initiateProcessing` method.

Given an instance of your processor, say `p`, this allows you to call this method (and in general all methods taking an object instance as first argument) in two different ways:

- `processChunk(p,in)`
- `p.processChunk(in)`

The two calls will of course return the same output.

Note

Having an instance of the processor as an argument means that you can access all of its properties to carry out the processing. In particular, the external and internal parameter properties you have defined earlier. For example, the processing method of a simple "gain" processor could read as `out = in * p.gain`

The arguments `in` and `out` are arrays containing "pure" data. Although signal-related data is stored as specific signal objects in the Auditory front-end, only the data is passed around when it comes to processing. It is done internally to avoid unnecessary copies. So it is not something that has to be addressed in the implementation of your processing method. Your input is an array whose dimensionality depends on the type of signal. Dimensions are ordered in the same way as in the data-storing buffer of the signal object. For example, the input `in` in the `gammatoneProc.processChunk` is a one-dimensional array indexing time. Similarly, the output should be arranged in the same way than in its corresponding output signal object. For example, the output `out` of `modulationProc.processChunk` is a three-dimensional array where the first dimension indexes time, the second refers to audio frequency and the third corresponds to modulation frequency. Just like the way data is stored in the `modulationSignal.Data` buffer.

Note

The first dimension for all signals used in the AFE is always indexing time.

5.7.2 Chunk-based and signal-based processing

As the name of the method `processChunk` suggests, you should implement the processing method such that it can process consecutive chunks of input signal, as opposed to the entire signal at once. This enables "online" processing, and eventually "real-time" processing once the software has been sufficiently optimised. This has two fundamental consequences on your implementation:

1. The input data to the processing method can be of arbitrary duration.
2. The processing method needs to maintain continuity between input chunks. In other words, when concatenating the outputs obtained by processing individual consecutive chunks of input, one need to obtain the same output as if all the consecutive input were concatenated and processed at once.

Point 1. above implies that depending on the type of processing you are carrying out, it might be necessary to buffer the input signal. For example, processors involving framing of the signal, such as `ratemapProc` or `ildProc`, need to put the segment of the input signal that went out of bound of the framing operation in a buffer. This buffer is then appended to the beginning of the next input chunk. This is illustrated in a section of the technical description of the framework. This also means that for some processor (those which lower the sampling rate in general), an input that is too short in time might produce an empty output. But this input will still be considered in the next chunk.

Point 2. is the most challenging one because it very much depends on the processing carried out by the processor. Hence there are no general guidelines. However, the AFE comes with some building blocks to help with this task. It features for instance filter objects that can be used for processing. All filters manage their internal states themselves, such that output continuity is ensured. For an example on how to use filters, see e.g. `gammatoneProc.processChunk`. Sometimes however, one need more than simple filtering operations. One can often find a workaround by using some sort of “overlap-save” method using smart buffering of the input or output as described in the technical description. A good example of using buffering for output continuity can be found in e.g., `ildProc.processChunk`.

5.7.3 Reset method

To ensure continuity between output chunks, your new processor might include “internal states” (e.g., built-in filter objects or internal buffers). Normally, incoming chunks of input are assumed to be consecutive segments of a same signal. However, the user can decide to process an entirely new signal as input at any time. In this case, your processor should be able to reset its internal states.

This is performed by the `reset` method. This method should be implemented in a method block with no method attributes, just like the constructor. It should simply reset the filters (if any) by calling all the filters `reset` methods, and/or empty all internal buffers.

If your processor does not need any internal state storage, then the `reset` method should still be implemented (as it is an abstract method of the parent class) but can be left empty (see, e.g., `itdProc.reset`).

5.8 Override parent methods

The AFE framework was developed to maximise code reusing. Many of the existing processors, although they carry out different processing tasks, have common attributes in terms of e.g., number of inputs, number of outputs, how to call their processing methods, ... Hence all aspects of initialisation (and re-initialisation following a response to feedback) and input/output routing have been implemented for common-cases as methods of the parent `Processor` class. If your processor does not behave similarly to others in one of these regards, then this approach allows you to redefine the specific method in your new children processor class definition. In the object oriented jargon, this procedure is called method overriding.

In the following, we list the methods that might need overriding and how to do so. Subsections for each methods will start with a description of what the method does and a note explaining in which cases the method needs to be overridden, such that you can quickly identify if this is necessary for your processor. Some examples of existing processors that override a given method will also be given so they can be used as examples. Note that all non- static methods from the parent `Processor` class can be overridden if necessary. The following list only concerns methods that were written with overriding in mind to deal with particular cases.

Note

Overridden methods need to have the same method attribute(s) as the parent method they are overriding.

5.8.1 Initialisation methods

`verifyParameters`

This method is called at the end of the `Processor` super-constructor. It ensures that user-provided parameters are valid. The current implementation of the AFE relies on the user being responsible and aware of which type or values are suitable for a given parameter. Therefore, we do not perform a systematic check of all parameters. Sometimes though, you might want to verify that user-provided parameters are correct in order to avoid Matlab returning an error at a later stage. For example, `ihcProc.verifyParameters` will check that the inner hair-cell model name entered by the user is part of the list of valid names.

Another use for the `verifyParameters` method is to solve conflicts between parameters. For example, the auditory filterbank in `gammatoneProc` can be instantiated in three different ways (e.g., by providing a range of frequency and a number of channels, or directly a vector of centre frequencies). The user- provided parameters for this processor are therefore potentially “over-determining” the position of centre frequencies. To make sure that there is no conflict, some priority rules are defined in `gammatoneProc.verifyParameters` to ensure that a unique and non-ambiguous vector of centre frequencies is generated.

Note

This method does nothing by default. Override it if you need to perform specific checks on external parameters (i.e., the user-provided parameters extended by the default values) before instantiating your processor.

To override this method, place it in a methods block with the `Access=protected` attribute. The method takes only an instance of the processor object (say, `pObj`) as input argument, and does not return any output.

If you are checking that parameters have valid values, replace those which are invalid with their default value in `pObj.parameters.map` (see e.g., `ihcProc.verifyParameters`). It is a good practice here to inform the user by returning a warning, so that he/she knows that the default value is used instead.

If you are solving conflicts between parameters, set up a priority rule and only retain user-provided parameters that have higher priority according to this rule (see e.g., `gammatoneProc.verifyParameters`). Mention explicitly this rule in the help line of your processor constructor.

`prepareForProcessing`

This method performs the remaining initialisation steps that we purposely did not include in the constructor as they initialise properties that are susceptible to change when receiving feedback. It also includes initialisation steps that can be performed only once processors have been linked together in a "processing tree". For example, `ildProc` needs to know the original sampling frequency of the signal before its cross-correlation was computed to provide lag values in seconds. But to access the cross-correlation processor and request that value, the two processors need to be linked together already, which does not happen at the level of instantiation but later. Hence this method will be called for each processors once they all have been inter-linked, but also whenever feedback is received.

Note

Override this method if your processor has properties or internal parameters that can be changed via user feedback or that comes directly from preceding processors in the processing tree.

This method should have the `Hidden=true` method attribute. Hidden methods are sometimes used in the AFE when we need public access to it (i.e., other objects than the processor itself should be able to call the method) but when it is not deemed necessary to have the user call it. The user can still call the method by explicitly writing its name, but the method will not appear in the list of methods returned by Matlab script `methods(.)` nor by Matlab's automatic completion.

The method only takes an instance of the processor as input argument and does not return outputs. In the method, you should initialise all internal parameters that are susceptible to changes from user feedback. Note that this includes the processor's output sampling frequency `FsHzOut` if this frequency depends on the processor parameters. A good example is `ratemapProc.prepareForProcessing`, which initialises internal parameters (framing windows), the output sampling frequency and some filters.

`instantiateOutput`

This method is called just after a processor has been instantiated to create a signal object that will contain the output of this new processor and add the signal to the data object.

Note

Override this method if your output signal object constructor needs additional input arguments (e.g., for a `FeatureSignal`), if your processor generates more than one type of output, or if your processor can generate either mono or stereo output (e.g., the current `preProc`). There is no processor in the current implementation that generates two different outputs. However, the pre- processor can generate either mono or stereo outputs depending on the number of channels in the input signal (see `preProc.instantiateOutput` for an example).

This method should have the `Hidden=true` method attribute. It takes as input an instance of your processor and a instance of a data object to add the signal to. It returns the output signal object(s) as a cell array with the usual convention that first column is left channel (or mono) and right column is right channel. Different lines are for different types of signals.

Warning

Because there is no such processor at the moment, creating a new processor that returns two different types of output (and not just left/right channels) might involve additional changes. This is left to the developers responsibility to test and adjust existing code.

5.8.2 Input/output routing methods

When the manager creates a processing "tree", it also populates the `Input` and `Output` properties of each processors with handles to their respective input and output signal objects. The methods defined in the parent `Processor` should cover most cases already, and it is unlikely that you will have to override them for your own processor. For these two methods, it is important to remember the internal convention when storing multiple signals in a cell array: columns are for audio channels (first column is left or mono and second column is right). Different lines are for different types of signals.

The way `Input` and `Output` properties are routed should be in accordance with how they are used in the `initiateProcessing` method, which will be described in the next subsection.

`addInput`

This method takes an instance of the processor and a cell array of handles to dependent processors (i.e., processors one level below in the processing tree) and does not return any arguments. Instead, it will populate the `Input` property of your processor with a cell array of handles to the signals that are outputs to the dependent processors. The current implementation of `Processor.addInput` works for three cases, which overall cover all currently existing processors in the AFE:

- There is a single dependent processor which has a single output.
- There are two dependent processors each with single output corresponding to the left and right channels of a same input signal.
- There is a single dependent processor which produces two outputs: a left and a right channel (such as preProc for stereo signals).

Note

Override this method if your processor input signals are related to its dependent processors in a different way than the three scenarios listed above.

This method should have the `Hidden=true` attribute. You should just route the output of your dependent processors to the input of your new processor adequately. Again, it was not necessary thus far to override this method, hence no examples can be provided here. Additionally, this functionality has not been tested, so it might imply some minor reworking of other code components.

`addOutput`

This method adds a signal object (or a cell array of signals) to the `Output` property of your processor.

Note

Override this method if your processor has multiple outputs of different types. If your processor returns two outputs as the left and right channel of a same representation, it is not necessary to override this method.

This method should have the `Hidden=true` method attribute. It takes as input an instance of the processor and a single or a cell array of signal objects.

5.8.3 Processing method

`initiateProcessing`

This method is closely linked to the `addInput`, `addOutput` and `processChunk` methods. It is a wrapper to the actual processing method that routes elements of the cell arrays `Input` and `Output` to actual inputs and outputs of the `processChunk` method and call that method. It also appends the new chunk(s) of output to the corresponding output signal(s).

The parent implementation considers two cases: monaural and binaural (i.e., a “left” and a “right” inputs) which produce single outputs.

Note

Override this method if your processor is not part of the two cases above or if your implementation of the `processChunk` has a different signature than the standard.

A good example of an overridden `initiateProcessing` method can be found inside the pre-processor code: `preProc.initiateProcessing`, as the processing method of the pre-processor does not have a standard signature as it returns two outputs (left and right channels).

5.9 Allowing alternative processing options

Sometimes, two different processors (implemented as two different classes) can perform the same operation. The choice between such alternative processors is made depending on a given user-provided (or default) request parameter value. This is the case for example for the auditory filterbank, which can be performed by either a Gammatone filterbank (`gammatoneProc.m`) or a dual-resonance non- linear filterbank (`drnlProc.m`).

As can be seen when browsing `parameterHelper`, the two processors should be listed under the same request name, and one of the parameters (`'fb_type'` in the example above) should allow to switch between the two (or more) alternatives. When the manager instantiates the processors and notices that a given representation has alternative ways of being computed, it will call the methods `isSuitableForRequest` of each alternatives to know which one should be used.

Therefore, if your processor represents an alternative way of carrying out a given operation, you should implement its `isSuitableForRequest` method, as well as for its alternative, if it was not already existing.

This method takes as unique input an instance of a processor and will look into its `parameters` property to determine if it is the suitable alternative. It will return a boolean indicating if it is suitable (`true`) or not (`false`). Note that this method is called internally, not from an actual processor instance that would be used afterwards, but from a dummy, empty processor generated using the user-provided request and parameters.

See `gammatoneProc.isSuitableForRequest` and `drnlProc.isSuitableForRequest` for examples.

5.10 Implement a new signal type

The AFE supports already a wide range of signal types:

- `TimeDomainSignal`: used for single-dimensional signal
- `TimeFrequencySignal`: used for two-dimension signals (time and frequency)
- `CorrelationSignal`: used for three-dimension signals (time, frequency and lags)
- `ModulationSignal`: used for three-dimension signals (time, audio frequency and modulation frequency).
- `FeatureSignal`: used for a labelled collection of time-domain signals
- `BinaryMask`: used for two-dimensional (time and frequency) binary signals (0 or 1).

If your new processor generates a new type of signal that is not currently supported, you might have to add your own implementation of a new signal. This tutorial will not go in details on how to implement new signal types. However, the following aspects should be considered:

- Your signal class should inherit the parent `Signal` class.

- It should implement the abstract `plot` method. If there is no practical way of plotting your signal, this method could be left empty.
- Its constructor should take as argument a handle to your new processor (that generates this signal as output), a buffer size in seconds, and a vector of size across the other dimensions (`[size_dim2, size_dim3,...]`). If more arguments are needed (as is the case for `FeatureSignal`), then this signature can be changed, but the `instantiateOutput` of your processor should also be overridden.

5.11 Recommendations for final testing

Now the implementation of your new processor should be finalised, and it is important to test it thoroughly. Below are some recommendations with regard to testing:

- Make sure that all aspects of your implementation work. Test for mono as well as stereo input signals, vary your processor parameters and check that the change is reflected accordingly in the output.
- If you have based your implementation on another existing implementation (even better, one that is documented in the literature), then compare your new implementation with the reference implementation and control that both provide the same output up to a reasonable error. A reasonable error, for a processor that does not involve stochastic processes should be around quantisation error, assuming that your new implementation is exactly as the reference.
- Test the online capability of your processor (i.e., maintaining the continuity of its output) by processing a whole signal and the same signal cut into chunks. Both runs should provide the same output (up to a “reasonable error”). You can use the test script `test_onlineVSoffline` to perform that task.

6 Conclusions

This supplementary document to Deliverable D2.3 has been created as an extensive instruction manual of the up-to-date version of AFE framework, such that the users can understand the fundamentals of its architecture and operation, and can make use of its range of capabilities.

The overall objective of the AFE is to transform the listeners' ear signals into multi-dimensional auditory representations. The software architecture is based on an object-oriented approach, resulting in a highly modular and flexible framework. This enables the AFE to be usable not only as a stand-alone software toolkit, but also in connection with other software/hardware platforms, such as the blackboard system of WP3 (see description in D3.2), and furthermore the robotics platform of WP5 (ROS/GenoM3, via a dedicated Matlab bridge - see description in D5.2).

One of the key functionalities of the AFE is its ability to adjust the bottom-up signal processing via feedback mechanisms. The AFE framework has been re-factored to allow on-the-fly parameter changes of individual processors. Consequently, peripheral properties can be modified during runtime and additional auditory feature representations can be requested, allowing the AFE to dynamically respond to changes in the acoustic environment.

A List of DEMO files

For the sake of reproducible research, source code to reproduce all the plots in the present document is provided as individual *demo* scripts. Table A.1 lists all the individual demo scripts, with a reference to the corresponding section in the report and a short description. These scripts can be found in the `/test` folder of the AFE software package.

Table A.1: List of demo scripts reproducing all the Matlab plots in this document.

Filename	Section	Description
`DEMO_Adaptation.m`	4.4	Adaptation loops
`DEMO_AMS.m`	4.13	Amplitude modulation spectrogram
`DEMO_Autocorrelation.m`	4.5	Autocorrelation
`DEMO_ChunkBased.m`	2.4	Chunk-based use of the AFE
`DEMO_Crosscorrelation.m`	4.15	Cross-correlation
`DEMO_DRNL.m`	4.2.2	Dual-resonance non-linear auditory filterbank
`DEMO_GaborFeatures.m`	4.14	Spectro-temporal modulation spectrogram
`DEMO_Gammatone.m`	4.2.1	Gammatone auditory filterbank
`DEMO_IC.m`	4.18	Interaural coherence
`DEMO_IHC.m`	4.3	Inner hair-cell modeling
`DEMO_ILD.m`	4.17	Interaural level difference
`DEMO_ITD.m`	4.16	Interaural time difference
`DEMO_MOC.m`	4.12	Medial Olivo-Cochlear feedback
`DEMO_OffsetStrength.m`	4.9	Offset strength detection
`DEMO_OnsetOffsetMaps.m`	4.10	Onset and offset mapping
`DEMO_OnsetStrength.m`	4.8	Onset strength detection
`DEMO_Pitch.m`	4.11	Pitch estimation
`DEMO_Precedence.m`	4.19	Precedence effect
`DEMO_PreProcessing.m`	4.1	Pre-processing of input signal
`DEMO_Ratemap.m`	4.6	Ratemap extraction
`DEMO_SpectralFeatures.m`	4.7	Spectral features extraction

List of Acronyms

Acronyms

2D two-dimensional

ACF auto-correlation function

AFE auditory front-end

AGC automatic gain control

AMS amplitude modulation spectrogram

ASR automatic speech recognition

BM basilar membrane

CASA computational auditory scene analysis

CASP computational auditory signal-processing and perception

CCF cross-correlation function

DC direct current

DOW description of work

DRNL dual-resonance non-linear

ETSI European telecommunications standards institute

FFT fast Fourier transform

FIFO first in, first out

FIR finite impulse response

IC interaural coherence

IHC inner hair-cell

IIR infinite impulse response

ILD interaural level difference

ITD interaural time difference

MOC medial olivo-cochlear

MU model units

OOP object-orientated programming

RMS root mean square

SACF summary auto-correlation function

SCCF summary cross-correlation function

SCM spectral crest measure

SFM spectral flattness measure

SPL sound pressure level

STRF spectro-temporal receptive field

T-F time-frequency

Bibliography

Backus, B. C. and Guinan, J. J. (**2006**), "Time-course of the human medial olivocochlear reflex," *The Journal of the Acoustical Society of America* **119**(5 Pt 1), pp. 2889–2904. (Cited on page 61)

Bacon, S. P. and Grantham, D. W. (**1989**), "Modulation masking: Effects of modulation frequency, depths, and phase," *Journal of the Acoustical Society of America* **85**(6), pp. 2575–2580. (Cited on page 62)

Bernstein, L. R. and Trahiotis, C. (**1996**), "The normalized correlation: Accounting for binaural detection across center frequency," *Journal of the Acoustical Society of America* **100**(6), pp. 3774–3784. (Cited on page 46)

Bernstein, L. R., van de Par, S., and Trahiotis, C. (**1999**), "The normalized interaural correlation: Accounting for $\mathrm{N_oS_\pi}$ thresholds obtained with Gaussian and "low-noise" masking noise," *Journal of the Acoustical Society of America* **106**(2), pp. 870–876. (Cited on page 46)

Braasch, J. (**2013**), "A precedence effect model to simulate localization dominance using an adaptive, stimulus parameter-based inhibition process." *The Journal of the Acoustical Society of America* **134**(1), pp. 420–35. (Cited on page 71)

Breebaart, J., van de Par, S., and Kohlrausch, A. (**2001**), "Binaural processing model based on contralateral inhibition. I. Model structure," *Journal of the Acoustical Society of America* **110**(2), pp. 1074–1088. (Cited on page 46)

Bregman, A. S. (**1990**), *Auditory scene analysis: The perceptual organization of sound*, The MIT Press, Cambridge, MA, USA. (Cited on page 56)

Brown, G. J. and Cooke, M. P. (**1994**), "Computational auditory scene analysis," *Computer Speech and Language* **8**(4), pp. 297–336. (Cited on page 51)

Brown, G. J., Ferry, R. T., and Meddis, R. (**2010**), "A computer model of auditory efferent suppression: implications for the recognition of speech in noise." *The Journal of the Acoustical Society of America* **127**(2), pp. 943–54. (Cited on page 43)

Clark, N. R., Brown, G. J., Jürgens, T., and Meddis, R. (**2012**), "A frequency-selective feedback model of auditory efferent suppression and its implications for the recognition of speech in noise." *Journal of the Acoustical Society of America* **132**(3), pp. 1535–1541. (Cited on pages 61 and 62)

Cooke, M., Green, P., Josifovski, L., and Vizinho, A. (**2001**), "Robust automatic speech recognition with missing and unreliable acoustic data," *Speech Communication* **34**(3), pp. 267–285. (Cited on page 51)

Dau, T., Püschel, D., and Kohlrausch, A. (**1996**), "A quantitative model of the "effective" signal processing in the auditory system. I. Model structure," *Journal of the Acoustical Society of America* **99**(6), pp. 3615–3622. (Cited on pages 46, 47, 48, and 49)

Dau, T., Püschel, D., and Kohlrausch, A. (**1997**a), "Modeling auditory processing of amplitude modulation. I. Detection and masking with narrow-band carriers," *Journal of the Acoustical Society of America* **102**(5), pp. 2892–2905. (Cited on pages 47, 49, and 62)

Dau, T., Püschel, D., and Kohlrausch, A. (**1997**b), "Modeling auditory processing of amplitude modulation. II. Spectral and temporal integration," *Journal of the Acoustical Society of America* **102**(5), pp. 2906–2919. (Cited on page 62)

ETSI ES 201 108 v1.1.3 (**2003**), "Speech processing, transmission and quality aspects (STQ); distributed speech recognition; front-end feature extraction algorithm; compression algorithms," URL `www.etsi.org`. (Cited on page 67)

Ewert, S. D. and Dau, T. (**2000**), "Characterizing frequency selectivity for envelope fluctuations," *Journal of the Acoustical Society of America* **108**(3), pp. 1181–1196. (Cited on page 62)

Faller, C. and Merimaa, J. (**2004**), "Source localization in complex listening situations: Selection of binaural cues based on interaural coherence," *Journal of the Acoustical Society of America* **116**(5), pp. 3075–3089. (Cited on page 70)

Ferry, R. T. and Meddis, R. (**2007**), "A computer model of medial efferent suppression in the mammalian auditory system," *The Journal of the Acoustical Society of America* **122**(6), pp. 3519. (Cited on page 44)

Glasberg, B. R. and Moore, B. C. J. (**1990**), "Derivation of auditory filter shapes from notched-noise data," *Hearing Research* **47**(1-2), pp. 103–138. (Cited on pages 41 and 42)

Göbbert, J. H. (**2014**), "Circular double buffered vector buffer (`circVBuf.m`)," *Matlab file exchange* URL `http://www.mathworks.com/matlabcentral/fileexchange/47025-circvbuf`, accessed: 2014-10-30. (Cited on page 22)

Goode, R. L., Killion, M., Nakamura, K., and Nishihara, S. (**1994**), "New knowledge about the function of the human middle ear: development of an improved analog model." *The*

American journal of otology **15**(2), pp. 145–154. (Cited on page 40)

Guinan, J. J. (**2006**), "Olivocochlear efferents: anatomy, physiology, function, and the measurement of efferent effects in humans." *Ear and hearing* **27**(6), pp. 589–607, URL `http://www.ncbi.nlm.nih.gov/pubmed/17086072`. (Cited on page 61)

Houtgast, T. (**1989**), "Frequency selectivity in amplitude-modulation detection," *Journal of the Acoustical Society of America* **85**(4), pp. 1676–1680. (Cited on page 62)

Jensen, K. and Andersen, T. H. (**2004**), "Real-time beat estimation using feature extraction," in *Computer Music Modeling and Retrieval*, edited by U. K. Wiil, Springer, Berlin–Heidelberg, Lecture Notes in Computer Science, pp. 13–22. (Cited on pages 52 and 53)

Jepsen, M. L., Ewert, S. D., and Dau, T. (**2008**), "A computational model of human auditory signal processing and perception." *Journal of the Acoustical Society of America* **124**(1), pp. 422–438. (Cited on pages 40, 41, 44, and 48)

Jørgensen, S. and Dau, T. (**2011**), "Predicting speech intelligibility based on the signal-to-noise envelope power ratio after modulation-frequency selective processing," *Journal of the Acoustical Society of America* **130**(3), pp. 1475–1487. (Cited on pages 46 and 63)

Jørgensen, S., Ewert, S. D., and Dau, T. (**2013**), "A multi-resolution envelope-power based model for speech intelligibility," *Journal of the Acoustical Society of America* **134**(1), pp. 1–11. (Cited on page 63)

Kim, G., Lu, Y., Hu, Y., and Loizou, P. C. (**2009**), "An algorithm that improves speech intelligibility in noise for normal-hearing listeners," *Journal of the Acoustical Society of America* **126**(3), pp. 1486–1494. (Cited on page 62)

Klapuri, A. (**1999**), "Sound onset detection by applying psychoacoustic knowledge," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3089–3092. (Cited on page 56)

Kollmeier, B. and Koch, R. (**1994**), "Speech enhancement based on physiological and psychoacoustical models of modulation perception and binaural interaction," *Journal of the Acoustical Society of America* **95**(3), pp. 1593–1602. (Cited on page 62)

Lerch, A. (**2012**), *An Introduction to Audio Content Analysis: Applications in Signal Processing and Music Informatics*, John Wiley & Sons, Hoboken, NJ, USA. (Cited on pages 53 and 54)

Liberman, M. C. (**1988**), "Response properties of cochlear efferent neurons: monaural vs. binaural stimulation and the effects of noise," *Journal of Neurophysiology* **60**(5), pp. 1779–1798, URL `http://jn.physiology.org/content/60/5/1779`. (Cited on pages 61 and 62)

Licklider, J. C. R. (**1951**), "A duplex theory of pitch perception," *Experientia* **7**(4), pp. 128–134. (Cited on page 49)

Lopez-Poveda, E. A. and Meddis, R. (**2001**), "A human nonlinear cochlear filterbank," *Journal of the Acoustical Society of America* **110**(6), pp. 3107–3118. (Cited on pages 40, 41, and 44)

May, T., Bentsen, T., and Dau, T. (**2015**), "The role of temporal resolution in modulation-based speech segregation," in *Proceedings of the Annual Conference of the International Speech Communication Association*, pp. 170–174. (Cited on page 62)

May, T. and Dau, T. (**2013**), "Environment-aware ideal binary mask estimation using monaural cues," in *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pp. 1–4. (Cited on page 62)

May, T. and Dau, T. (**2014**a), "Requirements for the evaluation of computational speech segregation systems," *Journal of the Acoustical Society of America* **136**(6), pp. EL398–EL404. (Cited on page 62)

May, T. and Dau, T. (**2014**b), "Computational speech segregation based on an auditory-inspired modulation analysis," *Journal of the Acoustical Society of America* **136**(6), pp. 3350–3359. (Cited on page 63)

May, T. and Gerkmann, T. (**2014**), "Generalization of supervised learning for binary mask estimation," in *International Workshop on Acoustic Signal Enhancement*, Antibes, France. (Cited on page 62)

May, T., van de Par, S., and Kohlrausch, A. (**2011**), "A probabilistic model for robust localization based on a binaural auditory front-end," *IEEE Transactions on Audio, Speech, and Language Processing* **19**(1), pp. 1–13. (Cited on page 68)

May, T., van de Par, S., and Kohlrausch, A. (**2012**), "Noise-robust speaker recognition combining missing data techniques and universal background modeling," *IEEE Transactions on Audio, Speech, and Language Processing* **20**(1), pp. 108–121. (Cited on page 51)

May, T., van de Par, S., and Kohlrausch, A. (**2013**), "Binaural Localization and Detection of Speakers in Complex Acoustic Scenes," in *The technology of binaural listening*, edited by J. Blauert, Springer, Berlin–Heidelberg–New York NY, chap. 15, pp. 397–425. (Cited on page 68)

Meddis, R. and Hewitt, M. J. (**1991**), "Virtual pitch and phase sensitivity of a computer model of the auditory periphery. I: Pitch identification," *Journal of the Acoustical Society of America* **89**(6), pp. 2866–2882. (Cited on page 49)

Meddis, R. and O'Mard, L. (**1997**), "A unitary model of pitch perception," *Journal of the*

Acoustical Society of America **102**(3), pp. 1811–1820. (Cited on page 59)

Meddis, R., O'Mard, L. P., and Lopez-Poveda, E. A. (**2001**), "A computational algorithm for computing nonlinear auditory frequency selectivity," *Journal of the Acoustical Society of America* **109**(6), pp. 2852–2861. (Cited on pages 41, 43, and 59)

Misra, H., Ikbal, S., Bourlard, H., and Hermansky, H. (**2004**), "Spectral entropy based feature for robust ASR," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 193–196. (Cited on page 53)

Peeters, G., Giordano, B. L., Susini, P., Misdariis, N., and McAdams, S. (**2011**), "The timbre toolbox: Extracting audio descriptors from musical signals." *Journal of the Acoustical Society of America* **130**(5), pp. 2902–2916. (Cited on pages 52, 53, and 54)

Püschel, D. (**1988**), "Prinzipien der zeitlichen Analyse beim Hören," Ph.D. thesis, University of Göttingen. (Cited on pages 47 and 49)

Qiu, A., Schreiner, C. E., and Escabì, M. A. (**2003**), "Gabor analysis of auditory midbrain receptive fields: Spectro-temporal and binaural composition." *Journal of Neurophysiology* **90**(1), pp. 456–476. (Cited on pages 65 and 66)

Rabiner, L. R. (**1977**), "On the use of autocorrelation analysis for pitch detection," *IEEE Transactions on Audio, Speech, and Language Processing* **25**(1), pp. 24–33. (Cited on page 50)

Schädler, M. R., Meyer, B. T., and Kollmeier, B. (**2012**), "Spectro-temporal modulation subspace-spanning filter bank features for robust automatic speech recognition," *Journal of the Acoustical Society of America* **131**(5), pp. 4134–4151. (Cited on page 66)

Scheirer, E. and Slaney, M. (**1997**), "Construction and evaluation of a robust multifeature speech/music discriminator," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1331–1334. (Cited on page 54)

Slaney, M. and Lyon, R. F. (**1990**), "A perceptual pitch detector," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 357–360. (Cited on page 59)

Smith, R. L. (**1977**), "Short-term adaptation in single auditory nerve fibers: some post-stimulatory effects," *J Neurophysiol* **40**(5), pp. 1098–1111. (Cited on page 47)

Smith, R. L., Brachman, M. L., and Goodman, D. a. (**1983**), "Adaptation in the Auditory Periphery," *Annals of the New York Academy of Sciences* **405**(1 Cochlear Pros), pp. 79–93. (Cited on page 47)

Søndergaard, P. L. and Majdak, P. (**2013**), "The auditory modeling toolbox," in *The*

Technology of Binaural Listening, edited by J. Blauert, Springer, Heidelberg–New York NY–Dordrecht–London, chap. 2, pp. 33–56. (Cited on pages 40, 41, 46, and 47)

Stern, R. M., Zeiberg, A. S., and Trahiotis, C. (**1988**), "Lateralization of complex binaural stimuli: A weighted-image model," *The Journal of the Acoustical Society of America* **84**(1), pp. 156–165, URL `http://scitation.aip.org/content/asa/journal/jasa/84/1/10.1121/1.396982`. (Cited on page 72)

Tchorz, J. and Kollmeier, B. (**2003**), "SNR estimation based on amplitude modulation analysis with applications to noise suppression," *IEEE Transactions on Audio, Speech, and Language Processing* **11**(3), pp. 184–192. (Cited on pages 40 and 62)

Theunissen, F. E., David, S. V., Singh, N. C., Hsu, A., Vinje, W. E., and Gallant, J. L. (**2001**), "Estimating spatio-temporal receptive fields of auditory and visual neurons from their responses to natural stimuli," *Network: Computation in Neural Systems* **12**, pp. 289–316. (Cited on page 65)

Tolonen, T. and Karjalainen, M. (**2000**), "A computationally efficient multipitch analysis model," *IEEE Transactions on Audio, Speech, and Language Processing* **8**(6), pp. 708–716. (Cited on page 50)

Turgeon, M., Bregman, A. S., and Ahad, P. A. (**2002**), "Rhythmic masking release: Contribution of cues for perceptual organization to the cross-spectral fusion of concurrent narrow-band noises," *Journal of the Acoustical Society of America* **111**(4), pp. 1819–1831. (Cited on page 58)

Tzanetakis, G. and Cook, P. (**2002**), "Musical genre classification of audio signals," *IEEE Transactions on Audio, Speech, and Language Processing* **10**(5), pp. 293–302. (Cited on pages 52 and 53)

Wallach, H., Newman, E. B., and Rosenzweig, M. R. (**1949**), "The Precedence Effect in Sound Localization," *The American Journal of Psychology* **62**(3), pp. 315–336, URL `http://www.jstor.org/stable/1418275`. (Cited on page 71)

Wang, D. L. and Brown, G. J. (Eds.) (**2006**), *Computational Auditory Scene Analysis: Principles, Algorithms and Applications*, Wiley / IEEE Press. (Cited on page 51)

Young, S., Evermann, G., Gales, M., Hain, T., Kershaw, D., Liu, X., Moore, G., Odell, J., Ollason, D., Povey, D., Valtchev, V., and Woodland, P. (**2006**), *The HTK Book (for HTK Version 3.4)*, Cambridge University Engineering Department, URL `http://htk.eng.cam.ac.uk`. (Cited on page 39)