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

Deliverable 6.1.3

Final report and evaluated software for analysis of dynamic auditory scenes

WP6 *

November 30, 2016

* 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 packages:	WP6
Document number:	D6.1.3
Document title:	Final report and evaluated software for analysis of dynamic auditory scenes
Version:	1

Delivery date:	30th November 2016
Actual publication date:	30th November 2016
Dissemination level:	Public
Nature:	Report

Editors:	Dorothea Kolossa and Guy Brown
Author(s):	Sylvain Argentieri, Jens Blauert, Jonas Braasch, Guy Brown, Benjamin Cohen-L'hyver, Patrick Danès, Torsten Dau, Rémi Decorsière, Thomas Forgue, Bruno Gas, Youssef Kashef, Chungeun Kim, Armin Kohlrausch, Dorothea Kolossa, Ning Ma, Tobias May, Johannes Mohr, Antonyo Musabini, Klaus Obermayer, Ariel Podlubne, Alexander Raake, Christopher Schymura, Sascha Spors, Jalil Taghia, Ivo Trowitzsch, Thomas Walther, Hagen Wierstorf, Fiete Winter
Reviewer(s):	Bruno Gas

Contents

1 Executive summary

In the Two!Ears project, we have developed an intelligent, active computational model of auditory perception and experience, which is capable of operating in a multi-modal context. The resulting Two!Ears system is described in this report, which has three core components.

Firstly, this deliverable provides an overview of the entire software architecture, and it gives specifications of the pertinent knowledge sources that have been developed as part of the Two!Ears software. The emphasis is on abstract specifications of the knowledge sources, rather than implementation details or numerical evaluation results. The reader is referred to Deliverable D3.5 for details on the implementation and evaluation of specific knowledge sources.

Secondly, it gives a number of application examples for its use within the proof-of-concept application of the search-and-rescue scenario.

Finally, this document contains three appendices, covering the relevant parts of the online documentation of the system:

- the software specifications that are pertinent to the auditory front end
- the specification of the core blackboard architecture
- and usage guides for a wide range of applications of the system.

2 Introduction

2.1 Overview

The Two!Ears software has been developed over the entire course of the Two!Ears project. It is available online at `https://github.com/TWOEARS/` in the form of a public github repository. At its core, it is a probabilistic blackboard system, designed to process incoming acoustic, visual and proprioceptive signals, make sense of its surroundings by creating a probabilistic representation of its environment at multiple levels of abstraction, and to plan its next action on the basis of its current understanding of the environment. In this decision, it is additionally guided by a set of rules that help it in understanding its current task—assessing the quality-of-experience of an auditory scene or provide assistance to a search-and-rescue operation in an emergency situation.

The system is available in two versions. A development system can be used within a simulated environment, without needing access to a robotic platform, and a deployment system is capable of real-time operation within the final robotic architecture. Both versions of the system share a common principal architecture to allow for easy deployability of new algorithms designed within the development system, and they are hence described jointly within this report, making distinctions only where specifically necessary.

In addition, the use of the system for a range of tasks has been considered in detail within a scenario-based approach. In order to facilitate the application of the system, we will hence exemplarily describe some of the use cases that have been developed in the course of the Two!Ears project.

2.2 Structure of this report

This document first describes the software specification of the Two!Ears system. This description is composed of two main parts – the specification of the overall blackboard architecture in Section 3.1, and of the knowledge sources in Section 3.4. The evaluation approach is described in Chapter 4, with examples of evaluated applications of the system to tasks that are pertinent within the search-and-rescue scenario.

The document is concluded by a discussion of the software release and an outlook on future work that it enables in Chapter 5.

3 Specification of software framework

This chapter describes the software framework, with a focus on abstract specifications, rather than implementation details. The implementation of the knowledge sources is described in Deliverables D3.4 and D3.5 and an extensive evaluation is provided as well in Deliverables D3.5 and D 4.3.

3.1 Blackboard architecture

The blackboard system is based on the architectural considerations that were presented in Deliverable D3.2. It has been designed to support a great variety of applications, by integrating a rich set of modules which can work either independently or in collaboration, and which can be called sequentially to realize both bottom-up and top-down processing. The system is also easily modifiable, through the exchange and/or extension of modules.

In principle, the system contains four fundamental building blocks, as detailed in D6.1.2:

Peripheral processing The incoming acoustic and visual signals are preprocessed, with visual processing carried out by one module, whereas acoustic preprocessing is achieved in a physiologically inspired multistage approach.

Blackboard The blackboard is the central data repository of the platform, which also keeps track of the history of this data in order to enable working on time series data. An associated blackboard monitor provides a view of the blackboard's state of information to the scheduler.

Knowledge Sources (KSs) are modules that define their own functionality, to be executed in the blackboard system. They define for themselves, which data they need for execution and which data they produce, but they do not need to know, how or where the data is stored. The blackboard system, in contrast, provides the tools for requesting and storing this data, but it does not care about its actual content.

Scheduler The scheduler executes the KSs in the appropriate, dynamic order. The order in which KSs get executed is initially computed (or *scheduled*) in a task-specific manner. It is then re-scheduled after every execution of a KS, since the conditions determining the order may have changed, or new, more urgent, KSs may be waiting for execution.

In the deployment system, the ***robot interface*** constitutes another significant component.

An overview of the Two!Ears software architecture including the connection of the blackboard system to all other software modules is shown in Fig. 3.1. The blackboard system has been released as part of the Two!Ears system with the corresponding documentation[1] of all its software components.

In the following, we will specify the components of the system, beginning with a brief review of the available documentation of the preprocessing modules in Section 3.2, followed by the scheduler in Section 3.3, specifying the knowledge sources (Sec. 3.4), and concluding with the specification of the robot interface in Section 3.5.

1 `http://docs.twoears.eu/en/latest/blackboard/`

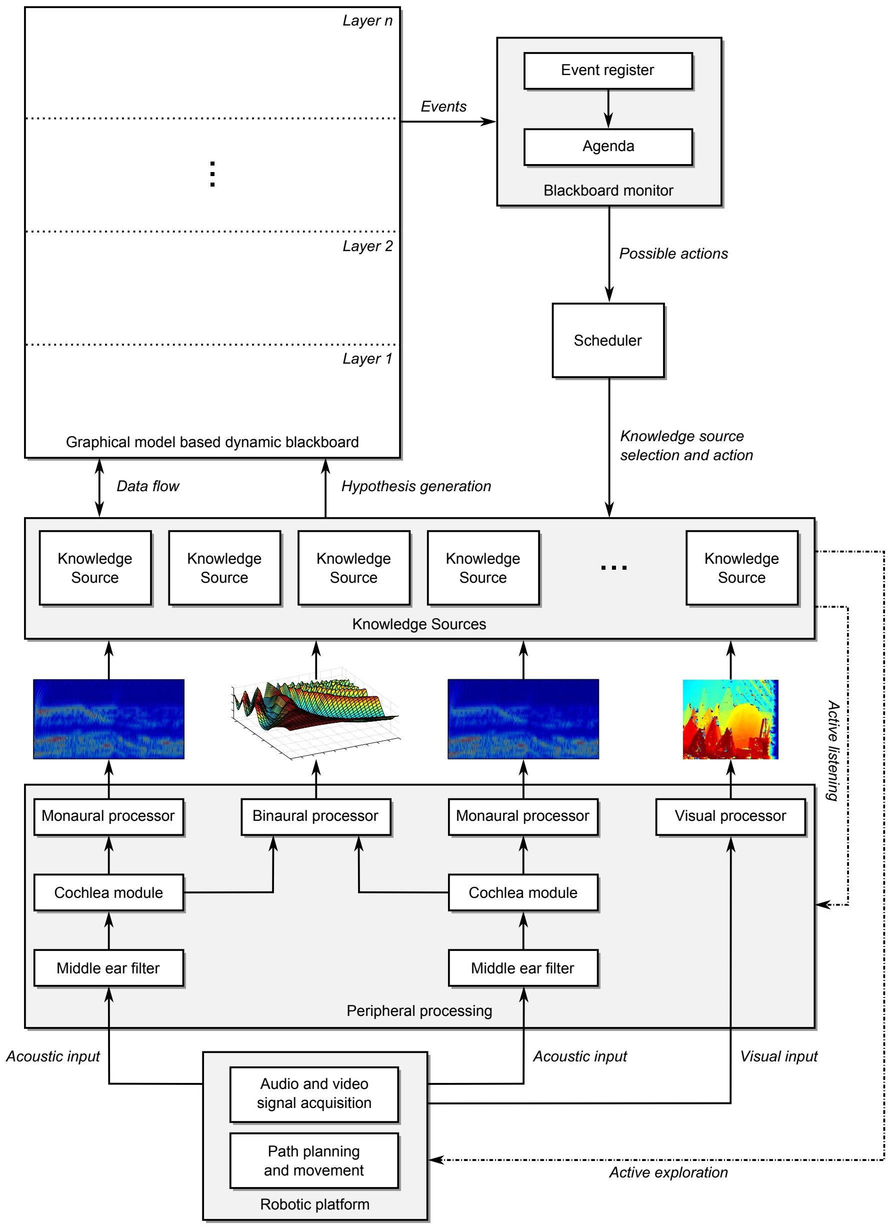


Figure 3.1: Overview of the general Two!Ears software architecture.

3.2 Peripheral processing

The peripheral processing block contains many alternative processing modules and processing paths. It is only mentioned here for the sake of giving a complete overview of the system, but it is not within the focus of the present deliverable. Instead, it has been introduced in detail in the WP2 deliverables and it is described under `http://docs.twoears.eu/en/latest/afe/`. Hence, we attach the complete documentation in Appendix A in order to make this deliverable comprehensive, but we do not introduce the components here.

Instead, we assume in the following that the features derived by peripheral processing block, shown at the bottom of Fig. 3.1, are provided as input values to the blackboard system.

3.3 Scheduler

The scheduler is the component of the blackboard system that actually executes the knowledge sources – but first, it schedules them, that is, it decides the order in which knowledge sources waiting in the agenda get executed. This order is rescheduled after every execution of a knowledge source, since the conditions determining the order may have changed, or new knowledge sources may be present in the agenda that are more urgent.

The implementation of the scheduler within the Two!Ears framework comprises a dynamic scheduling scheme, where the order of knowledge source execution can either be fixed or depend on a dynamically exchangeable priority value. This allows for the design of flexible processing schedules, which can be adapted to specific requirements during run-time. Furthermore, designated knowledge sources can be declared as periodically called instances, which are not scheduled according to priority values, but are executed after specified time intervals. This is especially helpful for repeated tasks like localisation or source identification, which need to be frequently updated, rather than be called on demand. In contrast, knowledge sources that deal with decision making or actuator control of the robotic platform are required to be executed after specific hypotheses on the blackboard have emerged. This behavior can be handled by the current scheduler implementation through its dynamic nature. A detailed overview of the implementation details and application programming interface of the scheduler can be found under `http://docs.twoears.eu/en/latest/blackboard/architecture/scheduler/` and in Appendix B.

3.4 Knowledge sources

3.4.1 Localisation

A number of knowledge sources (KSs) are developed to work together for estimation of source location.

Sound localisation from binaural cues

We describe KSs related to source localisation when the robot is assumed to be stationary. However, robot head movements can be triggered in case of front-back confusions.

DnnLocationKS

- **Description**:
  Computes posterior probabilities of source azimuths for a chunk of signals using deep neural networks (DNNs). The probabilities are computed independently for each signal chunk.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - AuditoryFrontEndKSs → 'KsFiredEvent'
  - FactorialSourceModelKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' → LocalisationDecisionKS
- **Writes**:
  - 'sourcesAzimuthsDistributionHypotheses'
- **Reads**:
  - 'interauralCrossCorrelation'
  - 'interauralLevelDifferences'
  - 'sourceSegregationHypothesis'

GmmLocationKS

- **Description**:
  Computes posterior probabilities of source azimuths for a chunk of signals using Gaussian mixture models (GMMs). The probabilities are computed independently for each signal chunk. The KS is interchangeable with DnnLocationKS.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - AuditoryFrontEndKSs → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' → LocalisationDecisionKS
- **Writes**:
  - 'sourcesAzimuthsDistributionHypotheses'
- **Reads**:
  - 'interauralTimeCorrelation'
  - 'interauralLevelDifferences'

LocalisationDecisionKS

- **Description**:
  Examines source azimuth hypotheses in order to predict a source location. In the case of a confusion, a head rotation can be triggered. Azimuth hypotheses for each signal chunk are integrated across time with a leaky integrator.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - DnnLocationKS → 'KsFiredEvent'
  - GmmLocationKS → 'KsFiredEvent'
- **Emits**:
  - 'RotateHead' → HeadRotationKS
  - 'KsFiredEvent' → SegmentationKS
- **Writes**:
  - 'locationHypothesis'
- **Reads**:
  - 'sourcesAzimuthsDistributionHypotheses'
  - 'locationHypothesis'

HeadRotationKS

- **Description**:
  Decides how to rotate the robot head and performs head rotation
- **Interfaces**:
  - BlackboardSystem.robotConnect
- **Receives**:
  - LocalisationDecisionKS → ‘RotateHead’
- **Emits**:
- **Writes**:
- **Reads**:
  - ‘locationHypothesis’

Sound source localisation using sensorimotor flow

The below knowledge sources are applicable for dynamic scene exploration with the actual robotic system.

SensorimotorLocalisationKS

- **Description**:
  Computes the most likely azimuths (relative to the binaural head) of 1+ sources on the basis of the binaural signal. Computes a Gaussian mixture representation of the posterior pdf of the position (azimuth and range) of a single active source by incorporating the motion of the binaural head. Note that the above component runs on the robot, not directly in the blackboard system. Hence, it is only available in the deployment system.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - Scheduler → ‘AgendaEmpty’
- **Emits**:
  - ‘KsFiredEvent’
- **Writes**:
  - ‘locationHypothesis’
- **Reads**:
  - binaural signal
  - sensor velocity

MostInformativeLocalMotionKS

- **Description**:
  Computes the direction of the velocity vector of a binaural head which would locally improve the quality of the audiomotor localization of a single source. Note that the above component runs on the robot, not directly in the blackboard system. Hence, it is only available in the deployment system.
- **Interfaces**:
  - BlackboardSystem.robotConnect
- **Receives**:
  - ReactToStimulusKS → ‘AuditoryObjectFormed’
- **Emits**:
  - No emission
- **Writes**:
  - No information written
- **Reads**:
  - ‘locationHypothesis’

Forming audio-visual objects

The formation of audio-visual objects is computed by the *HeadTurningModulationKS*[2] (HTMKS), and in particular through one of its two modules: the *MultimodalFusion& Inference* module. This process of making the robot interpret its environment through the notion of **objects** is a part of the more global computation of the head movements triggered by the HTMKS.

2 see Deliverable **4.3**, Section **c1, c2 & c6**

HeadTurningModulationKS

- **Description**:
  The HTMKS generates the composition of all the audio-visual objects the robot has observed so far in the current environment. This is achieved through the MFImod mainly but relies on many other KSs.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - ObjectDetectionKS → ‘KsFiredEvent’
- **Emits**:
  - ‘KsFiredEvent’ (upon generation of a new hypothesis)
- **Writes**:
  - ‘ObservedObjectsHypotheses’
- **Reads**:
  - identityHypotheses
  - sourcesAzimuthsDistributionHypotheses
  - visualIdentityHypotheses
  - objectDetectionHypotheses
  - audiovisualHypotheses

ObjectDetectionKS

- **Description**:
  The ObjectDetectionKS generates an hypothesis of whether the current audio and/or visual frame belongs to a new object or to an object that has already been observed.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - VisualIdentityKS → ‘KsFiredEvent’
- **Emits**:
  - ‘KsFiredEvent’ (upon generation of a new hypothesis)
- **Writes**:
  - ‘objectDetectionHypotheses’
- **Reads**:
  - sourcesAzimuthsDistributionHypotheses: ’sourcesDistribution’, ’azimuth’.

FocusComputationKS

- **Description**:
  The FocusComputationKS generates the object to be focused by the robot based on both the *DynamicWeighting* module and the *MultimodalFusion&Inference* module.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - HeadTurningModulationKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  - 'FocusedObject'
- **Reads**:
  - Nothing

AudioVisualFusionKS

- **Description**:
  The AudioVisualFusionKS generates hypothesis about the visual stream that is the most likely to be related to the audio stream momentarily perceived.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - visualLocationKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  - 'audiovisualHypotheses'
- **Reads**:
  - visualStreamsHypotheses: 'present_objects'
  - visualLocationHypotheses: 'theta'
  - sourcesAzimuthsDistributionHypotheses

3.4.2 Segmentation

Based on a given or estimated number of sources/objects, the incoming signals (acoustic or visual) are segmented into the signal components related to the relevant sources/objects.

Identifying the number of sound sources

NumberOfSourcesKS

- **Description**:
  Each instance of NumberOfSourcesKS incorporates a model that generates hypotheses about whether and how many sound sources are present in the audio stream in particular time span (extracted block from earsignals' streams).
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - AuditoryFrontEndKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  - 'NumberOfSourcesHypotheses'
- **Reads**:
  - AFE features – depending on the actual model plugged in. Currently the models commonly use: 'ratemap', 'amsFeatures', 'spectralFeatures', 'onsetStrength', 'ild', 'itd'.
  - SourcesAzimuthsDistributionHypothesis – Source localization estimates.

Source Segregation

FactorialSourceModelKS

- **Description**:
  Uses factorial source models to jointly estimate a segregation mask for the target source
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - AuditoryFrontEndKSs → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' → DnnLocationKS
- **Writes**:
  - 'sourceSegregationHypothesis'
- **Reads**:
  - 'ratemap'

StreamSegregationKS

- **Description**:
  The StreamSegregationKS generates several streams of acoustic features, corresponding to individual sound sources that are present in the scene. This is achieved via a probabilistic masking approach, where masks are generated using estimated source azimuths from DnnLocationKS and the predicted number of sources from NumberOfSourcesKS.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - AuditoryFrontEndKS → ‘KsFiredEvent’
  - DnnLocationKS → ‘KsFiredEvent’
  - NumberOfSourcesKS → ‘KsFiredEvent’
- **Emits**:
  - ‘KsFiredEvent’ (upon generation of a new hypothesis)
- **Writes**:
  - ‘SegregationHypotheses’
- **Reads**:
  - AFE features: ’ild’, ’itd’.
  - SourcesAzimuthsDistributionHypothesis – Source localization estimates.
  - NumberOfSourcesHypothesis – Predicted number of sources.

VisualStreamSegregationKS

- **Description**:
  The VisualStreamSegregationKS processes data from the robot’s vision and generates the number of objects present in its field of view.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - AuditoryIdentityKS → ‘KsFiredEvent’
- **Emits**:
  - ‘KsFiredEvent’ (upon generation of a new hypothesis)
- **Writes**:
  - ‘visualStreamsHypotheses’
- **Reads**:
  - Nothing

3.4.3 Source classification

Sound classification

IdentityKS

- **Description**:
  Each instance of IdentityKS incorporates a model that generates hypotheses about the presence of a certain source-type in particular time span (extracted block from earsignals' streams). Many IdentityKSs can be instantiated – one for each type to be identified.
- **Interfaces**:
  – BlackboardSystem.dataConnect
- **Receives**:
  – AuditoryFrontEndKS → 'KsFiredEvent'
- **Emits**:
  – 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  – 'identityHypotheses'
- **Reads**:
  – AFE features – depending on the actual model plugged in. Currently the models commonly use: 'ratemap', 'amsFeatures', 'spectralFeatures', 'onsetStrength'.

IntegrateFullstreamIdentitiesKS

- **Description**:
  An instance of IntegrateFullstreamIdentitiesKS collects all available identityHypotheses produced for a particular time span and integrates those of each sound type over time. A single IntegrateFullstreamIdentitiesKS collects hypotheses for all instantiated IdentityKSs.
- **Receives**:
  – IdentityKS → 'KsFiredEvent'
- **Emits**:
  – 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  – 'integratedIdentityHypotheses'
- **Reads**:
  – 'identityHypotheses' produced by all instantiated IdentityKSs

SegmentIdentityKS

- **Description**:
  Each instance of SegmentIdentityKS incorporates a model that generates hypotheses about the presence of a certain source-type given a set of masks produced by source segregation in a particular time span (extracted block from earsignals' streams). Many SegmentIdentityKSs can be instantiated – one for each type to be identified.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - AuditoryFrontEndKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  - 'segIdentityHypotheses'
- **Reads**:
  - AFE features – depending on the actual model plugged in. Currently the models commonly use: 'ratemap', 'amsFeatures', 'spectralFeatures', 'onsetStrength'.
  - masks produced by source segregation to apply on the AFE features.

IntegrateSegregatedIdentitiesKS

- **Description**:
  An instance of IntegrateSegregatedIdentitiesKS collects all available hyptheses produced by SegmentIdentityKSs for a particular time span and integrates the hypotheses of each sound type over time and azimuth neighborhood. A hypothesis is produced for each present sound type indicating its location. A single IntegrateSegregatedIdentitiesKS collects hypotheses for all instantiated SegmentIdentityKSs.
- **Receives**:
  - SegmentIdentityKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  - 'singleBlockObjectHypotheses'
- **Reads**:
  - 'segIdentityHypotheses' produced by all instantiated SegmentIdentityKSs

IdentityLocationKS

- **Description**:
  Each instance of IdentityLocationKS drives a model that generates hypotheses about the presence of source-types their respective azimuth location in a particular time span (extracted block from earsignals' streams).
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - AuditoryFrontEndKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  - 'identityHypotheses'
  - 'sourcesAzimuthsDistributionHypotheses'
- **Reads**:
  - AFE features – depending on the actual model plugged in. Currently the models commonly use: 'ratemap', 'amsFeatures', 'spectralFeatures', 'onsetStrength', 'ild'.

IdentityLocationDecisionKS

- **Description**:
  An instance of IdentityLocationDecisionKS collects hypotheses on the joint identity and location of sound types produced for a particular time span. A decision is made regarding whether the sound type is at all present in the scene and decides on its most likely azimuth locations.
- **Receives**:
  - IdentityLocationKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  - 'segIdentityHypotheses'
- **Reads**:
  - 'identityLocationHypotheses'

Gender classification

GenderRecognitionKS

- **Description**:
  Recognizes the speakers' gender from speech audio data.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - AuditoryFrontEndKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  - 'GenderHypotheses'
- **Reads**:
  - AFE features: 'ratemap', 'pitch', 'spectralFeatures'.

Speaker identification

SpeakerRecognitionKS

- **Description**:
  Recognises the speaker identity from speech audio data.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - AuditoryFrontEndKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  - 'speakerIdentityHypotheses'
- **Reads**:
  - 'ratemap'

Keyword recognition

KeywordRecognitionKS

- **Description**:
  Recognises a spoken keyword from speech audio data.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - AuditoryFrontEndKS → ‘KsFiredEvent’
- **Emits**:
  - ‘KsFiredEvent’ (upon generation of a new hypothesis)
- **Writes**:
  - ‘keywordHypotheses’
- **Reads**:
  - ‘ratemap’

Musical genre recognition

MusicalGenreKS

- **Description**:
  Predicts the musical genre from a stream of audio signals containing music. A fixed set of genres, namely ’blues’, ’classic’, ’country’, ’disco’, ’hiphop’, ’jazz’, ’metal’, ’pop’, ’reggae’ and ’rock’ can be classified.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - AuditoryFrontEndKS → ‘KsFiredEvent’
- **Emits**:
  - ‘KsFiredEvent’ (upon generation of a new hypothesis)
- **Writes**:
  - ‘MusicalGenreHypotheses’
- **Reads**:
  - AFE features: ’ratemap’, ’pitch’, ’spectralFeatures’, ’onsetStrength’, ’offsetStrength’.

Turning to a perceived stimulus

This task is also handled by two modules of the HTMKS: the *DynamicWeighting* module and the *MultimodalFusion&Inference* module. The KSs on which it relies on are the same as in Sec. 3.4.1. Hence, here we just describe the KS responsible for computing the motor order on the basis of the *FocusComputationKS*.

MotorOrderKS

- **Description**:
  The MotorOrderKS generates an hypothesis about the angle the head has to turn, according to the computations made by the *HeadTurningModulationKS*.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - FocusComputationKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  - 'motorOrder'
- **Reads**:
  - FocusedObject: 'focus'.
  - headOrientation.

3.4.4 Cognitive Functions

BindingKS

- **Description**:
  The BindingKS generates a set of *binding hypotheses*: each hypothesis in this set relates the location (head-centric azimuth) of a detected sound source to the source's identity. Note: this KS is specifically designed for emulation in the BEFT (cf. D4.3).
- **Interfaces**:
  - BEFT emulator
- **Receives**:
  - UpdateEnvironmentKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  - 'bindingHypotheses'
- **Reads**:
  - from BEFT: emulated reference position (x,y,heading).

AuditoryMetaTaggingKS

- **Description**:
  The AuditoryMetaTaggingKS assigns emulated 'meta tags' to all auditory object hypotheses created by the *AuditoryObjectFormationKS*. Meta tags include source characteristics like category, role, gender, stress level, loudness level, age. Note: the *AuditoryObjectFormationKS* is specifically designed for emulation in the BEFT (s. D4.3).
- **Interfaces**:
  - BEFT emulator
- **Receives**:
  - AuditoryObjectFormation → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon modification of the processed hypotheses)
- **Writes**:
  - 'auditoryObjectHypotheses'
- **Reads**:
  - 'auditoryObjectHypotheses'

AuditoryObjectFormation

- **Description**:
  The emulated identity information stored in each binding hypothesis allows the *AuditoryObjectFormationKS* to create a unique *auditoryObjectHypothesis* for each perceived sound source, and enables straightforward triangulation of the latter on a per-source basis. This approach results in a robust, least-squares estimate of all sources' positions in the azimuthal plane. In addition, the *AuditoryObjectFormationKS* places the *globalLocalizationInstability* hypothesis on the blackboard, for later use in the *PlanningKS*. Note: the *AuditoryObjectFormationKS* is specifically designed for emulation in the BEFT (s. D4.3).
- **Interfaces**:
  - BEFT emulator
- **Receives**:
  - BindingKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon modification of the processed hypotheses)
- **Writes**:
  - 'auditoryObjectHypotheses'
  - 'globalLocalizationInstability'
- **Reads**:
  - 'bindingHypotheses'
  - 'auditoryObjectHypotheses'

Relevance Detection

HazardAssessmentKS

- **Description**:
  The HazardAssessmentKS augments the *auditoryObjectHypotheses* stored in blackboard memory with individual *hazard scores*. To that end, the KS integrates *meta information* provided for each scenario entity by the *AuditoryMetaTaggingKS*. The individual hazard scores are accumulated, and constitute the *globalHazardHypothesis* which is pushed to the blackboard memory. Note: the *HazardAssessmentKS* is specifically designed for emulation in BEFT (D4.3).
- **Interfaces**:
  - BEFT emulator
- **Receives**:
  - AuditoryMetaTaggingKS $\rightarrow$ 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon modification of the processed hypotheses)
- **Writes**:
  - 'auditoryObjectHypotheses'
  - 'globalHazardHypothesis'
- **Reads**:
  - 'auditoryObjectHypotheses'

Task Detection (S&R vs. QoE)

EmergencyDetectionKS

- **Description**:
  This knowledge sources uses the *IdentityKS* to detect whether the current situation contains sound sources indicating an emergency situation. To increase the robustness of the emergency detection and combat possible false alarms, the hypotheses generated by the *IdentityKS* are accumulated over a longer time-frame and an emergency is only triggered, if the probability of sounds indicative of danger (like *fire* or *alarm*) exceed a specified threshold.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - IdentityKS → 'KsFiredEvent'
- **Emits**:
  - 'KsFiredEvent' (upon generation of a new hypothesis)
- **Writes**:
  - 'emergencyHypotheses'
- **Reads**:
  - 'identityHypotheses'

PlanningKS

- **Description**:
  The *PlanningKS* constitutes the cognitive controller employed to drive emulation in the BEFT: this KS implements a task stack together with basic decision rules which allow for active exploration, and the active localization of potential victims in a catastrophe scenario. The cognitive functionality encoded in the *PlanningKS* has to be adapted to novel situations (cf. D4.3), and yields a sequence of tasks and sub-tasks for the robot to follow. Note: the *PlanningKS* is specifically designed for emulation in the BEFT (see D4.3).
- **Interfaces**:
  - BEFT emulator
- **Receives**:
  - HazardAssessmentKS → ‘KsFiredEvent’
- **Emits**:
  - ‘KsFiredEvent’ (upon modification of the processed hypotheses)
- **Writes**:
  - ‘auditoryObjectHypotheses’
  - ‘currentSuperTask’
  - ‘currentSubTask’
- **Reads**:
  - ‘auditoryObjectHypotheses’
  - ‘globalHazardHypothesis’
  - ‘currentSuperTask’
  - ‘currentSubTask’
  - ‘globalLocalizationInstability’

3.5 Robot Interface

In this section, we describe the Robot Interface, which provides basic communication between the blackboard system and the robot.

RobotInterface (abstract)

- **[sig, durSec, durSamples] = getSignal(durSec)** returns an audio signal of *durSec* seconds
- **rotateHead(angleDeg, mode)** rotates the robot head by *angleDeg* degrees, in either 'absolute' or 'relative' mode
- **azimuth = getCurrentHeadOrientation** returns the current head orientation relative to the torso orientation
- **[maxLeft, maxRight] = getHeadTurnLimits** returns the maximum head orientation relative to the torso orientation
- **moveRobot(posX, posY, theta, mode)** moves the robot to a new position
- **[posX, posY, theta] = getCurrentRobotPosition** returns the current robot position
- **b = isActive** returns true if robot is active

The robot simulator class and the interface class for a real robot are subclasses of this robot interface. Such a subclass has been written to interface the blackboard system with the mobile platform from WP 5.

3.5.1 Audio acquisition

The `getSignal()` method of the robot interface retrieves a block of audio signal of chosen duration from the binaural sensor. It returns the latest available data. The implementation on the mobile platform from WP 5 uses the Binaural Audio Stream Server, accessed by the robot interface through a genomix client (cf. D 5.3).

3.5.2 Movement Control and Mapping

The robot is a mobile base moving in a pre-learned map of the environment using Simultaneous Localisation And Mapping (SLAM) techniques. A frame is attached to this map, denoted as the world frame, defining the origin and the `(x,y)` directions. The `moveRobot()` method of the robot interface allows to send a new target position to the navigation system implemented on the platform. This target can be either absolute coordinates in the world frame, or in coordinates relative to the previous position of the mobile base.

The navigation system then computes a path to the target, given obstacles in the map. Using odometry and sensors such as lasers, the mobile base keeps track of its position as motor commands are applied to follow the computed trajectory.

At any moment, the `getCurrentRobotPosition()` method of the robot interface can be called to obtain the current coordinates of the mobile base in the world frame.

Independently from the movements of the mobile base, the embedded binaural sensor can be controlled in rotation. Typically, on a KEMAR Head-And-Torso Simulator (HATS) with a motorised neck, the head can rotate relatively to the torso. The `rotateHead()` method allows to turn the head to a targeted angle, either in absolute or relative mode. The `getCurrentHeadOrientation()` method returns the current angle.

The implementation of movement control on the mobile platform from WP 5 relies on various robotic components, accessed by the robot interface through a genomix client (cf. D 5.3).

3.5.3 Identifying and localizing visual objects

The methods available for identification and localization of visual objects may depend on the implementation on the robotic platform. On the platform from WP 5 for instance, object detection is performed on a dedicated CPU, and Knowledge Sources can access the results of the detection directly from the robot interface through a dedicated method. This led to the creation of the following KSs:

VisualLocationKS

- **Description**:
  The VisualLocationKS generates hypotheses about the locations of detected visual objects, with respect to the head position.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - VisualStreamSegregationKS → ‘KsFiredEvent’
- **Emits**:
  - ‘KsFiredEvent’ (upon generation of a new hypothesis)
- **Writes**:
  - ‘visualLocationHypotheses’
- **Reads**:
  - ‘visualStreamsHypotheses’: ‘present_objects’

VisualIdentityKS

- **Description**:
  The VisualIdentityKS generates hypotheses about the identity of visual objects detected within the robot’s field of view.
- **Interfaces**:
  - BlackboardSystem.dataConnect
- **Receives**:
  - AudioVisualFusionKS → ‘KsFiredEvent’
- **Emits**:
  - ‘KsFiredEvent’ (upon generation of a new hypothesis)
- **Writes**:
  - ‘visualIdentityHypotheses’
- **Reads**:
  - visualStreamsHypotheses: ’present_objects’

Similar features for human detection are functional on the platform from WP5, but they are not used in the blackboard system.

4 Scenario-based implementation and evaluation

All project phases have been carried out in adherence to a scenario-based development paradigm. As this approach has proven valuable throughout all phases, supporting decision making and design, we will also give the following applications guide in the form of the description of a small number of valuable scenarios. A larger number of scenarios is being addressed in detail in the online documentation. To make this document complete independently, we also show all of these examples in Appendix C.

4.1 General aspects of implementation and evaluation

Controlled by the scheduler in the blackboard system[1], the appropriate knowledge sources for each of the respective tasks are called in an appropriate order, which may be determined either by a task-dependent recipe, or by calling knowledge sources in response to the current blackboard state.

4.2 Application of the system in search-and-rescue scenarios

There are a wide range of possible and of available applications of the system. Many of them are described online at `http://docs.twoears.eu/en/latest/examples/`, which can also be found in the attachment, cf. Appendix C.

We therefore focus on two relevant, exemplary applications in the following: we describe the use of the system for multi-speaker localisation, for keyword recognition and for the localisation and characterisation of sources in a multi-room apartment. All of these are pertinent to the search-and-rescue scenario. Applications to the QoE scenario are discussed in the concurrent Deliverable 6.2.3.

1 `http://docs.twoears.eu/en/1.3/blackboard/`
`http://docs.twoears.eu/en/1.3/blackboard/architecture/#dynamic-blackboard-scheduler`

4.2.1 Application to multi-source speaker localisation

Overview: This demonstrates the use of head movement in sound localisation when the robot position is fixed. The robot does not restrict localisation of sound sources to the frontal hemifield. Due to the similarity of binaural cues in the frontal and rear hemifields, front-back confusions often occur. To address this, the robot employs a hypothesis-driven feedback stage that triggers a head movement whenever the source location cannot be unambiguously estimated. One or more sound sources can be present about the robot. When a front-back confusion occurs, the robot actively rotates the head by a few degrees. Information before and after the head rotation is combined to help reduce front-back errors and to decide on the "true" positions of the sound sources.

Tasks: Find the location of all sound sources present, using head movement if necessary.

Measure of success: root mean square (RMS) error of the target azimuth

Application of the blackboard system: For this scenario, the following knowledge sources are used:

KSs involved: multi-source speaker localisation

- **AuditoryFrontEndKS**: receives audio signals from the robot and extracts auditory features
- **DnnLocationKS**: estimates posterior probabilities of all source azimuths given a block of signal
- **LocalisationDecisionKS**: combines the previous location hypothesis with the newly estimated azimuth posterior probabilities to make a localisation decision, and head rotation is triggered in case of a confusion
- **HeadRotationKS**: analyses a location hypothesis to decide how to rotate the head in order to solve a confusion

The interaction of all KSs involved is illustrated in Fig. 4.1.

Evaluation: The scenario is fully evaluated in simulated experimental settings as well as demonstrated on the robot in real environments. Fig. 4.2 shows a screenshot of the blackboard system running when applied to this scenario. In the upper-right panel knowledge source activities are displayed to show the state of the blackboard system, and the size of the bubbles reflects execution time of each knowledge source. The lower-right panel shows the output of the auditory front-end for this scenario. Finally in the lower-left panel the reference source azimuth is shown as the green dot. The estimated posterior probabilities for all azimuths around the head are displayed as bars with the tallest bar indicating the most likely source azimuth. Head rotation is triggered in this case which turns toward the mostly likely azimuth.

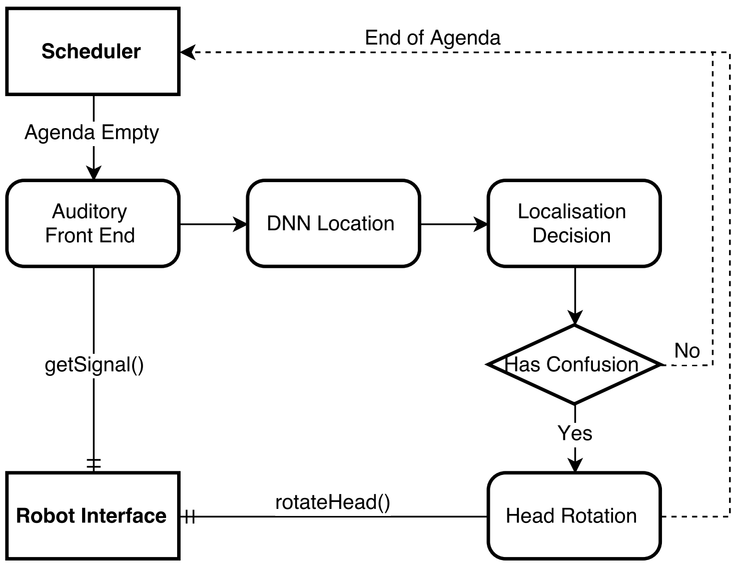


Figure 4.1: Interaction of various knowledge sources (KSs) in the blackboard applied to a sound localisation scenario that triggers a head movement in case of front-back confusions.

4.2.2 Application to keyword recognition

Overview: Recognition of spoken keywords in the presence of noise and reverberation. The database that is used for the evaluation is the CHiME challenge data (Barker *et al.*, 2013), where recordings of domestic noise in a living room (e.g. vacuum cleaners, children playing, music) are superimposed on binaural speech recordings.

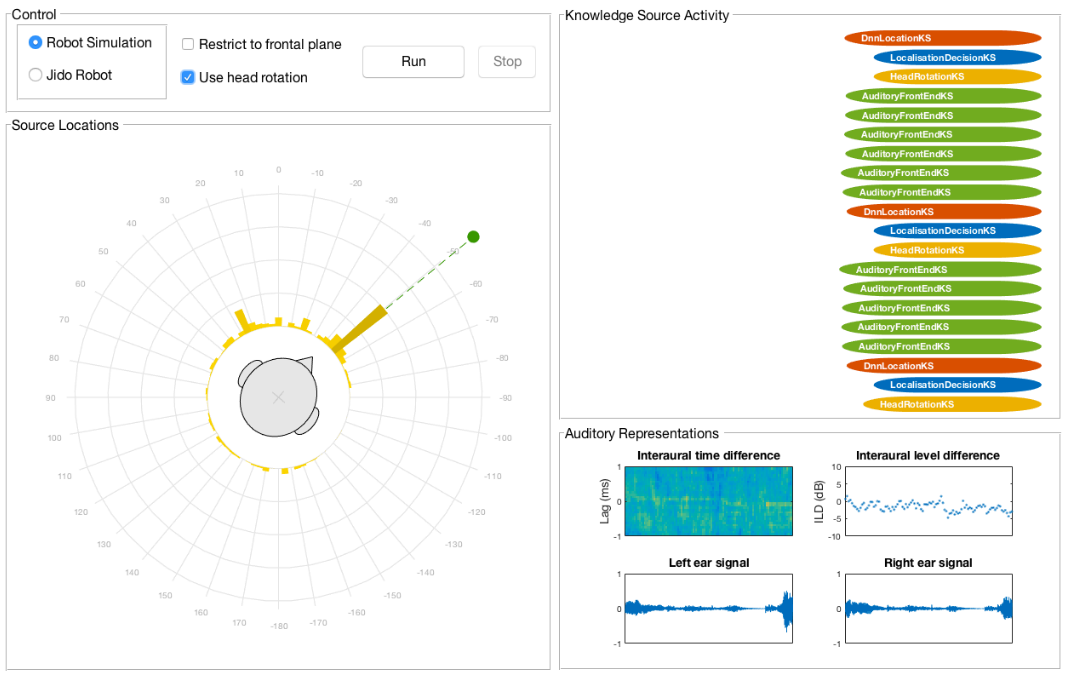
Tasks: Identify the keyword that was spoken.

Measure of success: Keyword recognition accuracy.

Application of the blackboard system Knowledge sources for *source segregation*, *source identification* and *keyword recognition*, can be employed. Alternatively, it is possible to use only the *keyword recognition* KS. This is a viable approach, as long as the noise level is not excessive, and as long as the model has been trained on the respective noise condition, as shown below.

KSs involved

- **AuditoryFrontEndKS**: receives audio signals from the robot and extracts auditory features
- **KeywordRecognitionKS**: carries out keyword recognition, using a deep-neural-network-based approach


Control

[✓ Robot Simulation] Jido Robot

Restrict to frontal plane [ ]

Use head rotation [✓]

Figure 4.2: Application of the blackboard in the sound localisation scenario.

Evaluation: The keyword recognition has been evaluated on binaural speech in noise from the CHiME corpus. Table 4.1 shows the keyword accuracies that were achieved on ratemap features from the auditory front end, when models were trained on noisy data at all SNRs, without any additional application of source separation or signal enhancement.

Features	-6 dB	-3 dB	0 dB	3 dB	6 dB	9 dB	Avg.
Gammatone FB	73.04	77.72	83.42	87.16	89.97	92.26	83.93
Ratemap	73.38	79.68	84.86	88.86	91.58	93.28	85.27

Table 4.1: Keyword accuracies (%) in household noise, using acoustic gammatone or ratemap features.

The results were achieved using deep neural networks, as described in Deliverable 3.5, Section 4.10.1. More details on the training of the recognition model can be found in Meutzner *et al.* (2017).

4.2.3 Application to localisation and characterisation of sources in a multi-room apartment

Overview:

Running in emulation mode, the Bochum Experimental Feedback Testbed (BEFT) allows active exploration in search-and-rescue scenarios of moderate complexity. To that end, it integrates with the blackboard architecture, and relies on a set of specifically designed knowledge sources including the BindingKS, the AuditoryObjectFormationKS, the AuditoryMetaTaggingKS, the HazardAssessmentKS, and the PlanningKS (cf. above). With these, the blackboard driving the BEFT is enabled to locate multiple victims through active search in an emulated indoor scene. For more details, refer to Deliverable D4.3, Section 5.

Tasks:

As indicated above, the primary goal of the emulated robot is to rescue several victims in a multi-compartment building. More concretely, the rescue scenario is located in a synthetic replication of the ADREAM lab in Toulouse, France (cf. D4.3, Section 5.4). The entities found in the scene are enumerated in Table 4.2.

Entity	**category**	**pre event role**	**post event role**	**gender**	**age**
Source001	human	employee	victim	male	25
Source002	animal	dog	victim	male	2
Source003	human	employee	rescuer	female	30
Source004	human	employee	victim	male	40
Source005	alert	siren	siren	NA	NA
Source006	threat	fire	fire	NA	NA
Source007	human	employee	victim	female	20

Table 4.2: Meta characteristics of the entities found in the evaluation scenario described in D4.3, Section 5.

The scenario starts in normal lab conditions, then, after $T_{event} = 60$ seconds, the situation evolves into a catastrophy scenario, namely, after an assumed explosion, attendant lab employees become victims or rescuers, and a fire starts in one corner of the lab. Table 4.2 subsumes the meta characteristics of all entities present in the scenario, including their roles before and after T_{event}' [see D4.3, Section 5.4]. Note that the robot will only save animate entities, thus the rescue attempt ends when sources {'Source001','Source002','Source003','Source004','Source007'} have successfully been evacuated.

Measure of success and evaluation:

BEFT allows us to automatically generate a range of different scenarios with varying characteristics, thus allowing for quantitative assessment of the performance of

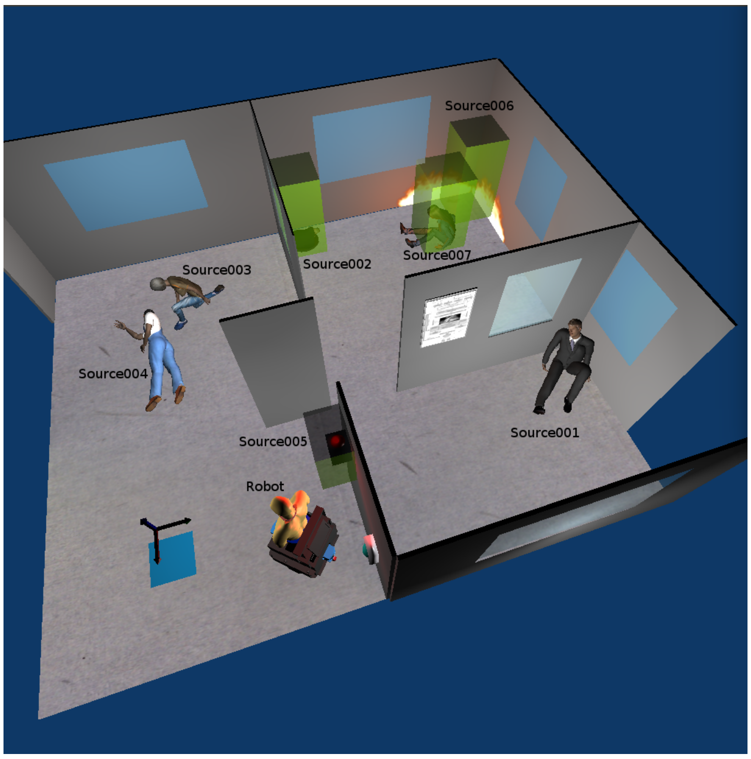


Figure 4.3: A typical S-&-R scenario solved within the Bochum Experimental Feedback Testbed.

search and rescue (S-&-R) schemes encoded in the *PlanningKS*. Focusing on the S-&-R strategy discussed in [... D4.3, Section 5 ...], $N_R = 30$ scenarios [...] are generated by randomly altering the x/y-positions of all animate entities [D4.3, Section 5.4].

Now, let T_r^A represent the time required to localize all present entities with sufficient precision in scenario r, using emulated acoustic cues, and employing baseline triangulation techniques (s. D4.3, Section 5 for details). Further, let T_r^B be the time it takes to evacuate all animate beings, and to achieve a successful solution in scenario r. This allows to define the arithmetic means

$$\mu_A = \frac{1}{N_R}\sum_{r=1}^{N_R} T_r^A, \qquad \mu_B = \frac{1}{N_R}\sum_{r=1}^{N_R} T_r^B \tag{4.1}$$

and the corresponding standard deviations

$$\sigma_A = \sqrt{\frac{1}{N_R}\sum_{r=1}^{N_R}\left(T_r^A - \mu_A\right)^2}, \qquad \sigma_B = \sqrt{\frac{1}{N_R}\sum_{r=1}^{N_R}\left(T_r^B - \mu_B\right)^2}. \tag{4.2}$$

In the current experiment, the obtained values are $\mu_A = 36.8609\,\mathrm{s}$, $\sigma_A = 6.1922\,\mathrm{s}$, and $\mu_B = 248.6996\,\mathrm{s}$, $\sigma_B = 31.6340\,\mathrm{s}$. In upcoming experiments, these values will have to be compared with results from trials where human assessors guide the robotic agent manually through numerous emulated rescue attempts. This would also set the pace for perceptual evaluation in addition to the instrumental one applied so far [D4.3, Section 5].

Application of the blackboard system: For the DASA-4 scenario, knowledge sources for *visual person detection*, *planning head rotations*, *planning robot movements*, *source segregation*, *source identification* are employed, with provisions for *identification of distressed speech* and *identification of alarm sounds*, *gender recognition*, and *keyword recognition.*

Scheduling is determined dynamically, corresponding to the status of the blackboard. The *PlanningKS* reacts to changing environmental conditions (e.g., positions of sources, room geometries), and enables purposeful behavior of the virtual robot in scenarios of moderate complexity.

4.3 Concluding Remarks

The scenario-based approach to developing our system has proven valuable throughout all project phases. It has allowed us to simultaneously focus our effort on the most relevant application scenarios, while identifying building blocks and components that are of importance through multiple use cases. This has always informed the design of the system components, as specified in Chapter 3. Here, we have focused on exemplary applications that show the range of possibilities within the search-and-rescue context. A larger set of applications of the Two!Ears system is described in Appendix C below.

5 Conclusions

Over the course of the Two!Ears project, we have implemented a dynamic and flexible architecture for the cognitive analysis of acoustic and multi-modal scenes, which has been evaluated in depth in a number of recent publications, e.g. Schymura *et al.* (2014), Ma *et al.* (2015b), Schymura *et al.* (2015), Ma *et al.* (2015a).

This deliverable contains the software specification of the Two!Ears software, with the exception of the preprocessing modules that have already been defined in D2.1, D2.2, D2.3, and D2.4 and that are hence only referenced here. After giving an overview of the software architecture, it specifies all necessary knowledge sources for the blackboard architecture as well as the robot interface.

The deliverable concludes with a brief application guide, discussing three applications. Appendices contain the complete software documentation of the preprocessing modules, the blackboard architecture, and a wider set of application examples, as available online at `http://docs.twoears.eu/en/latest/`.

It is envisaged that this software system, which is fully available under an open-source license, will allow for a wide range of research works in the area of auditory and audio-visual scene understanding, in modeling cognition for perceptual processing, and in utilizing complex world models for the assessment of audio signal and reproduction quality.

Acronyms

HATS	Head-And-Torso Simulator
KEMAR	Knowles Electronics manikin for acoustic research
KS	Knowledge Source
RMS	root mean square
SLAM	Simultaneous Localisation And Mapping

Bibliography

Barker, J., Vincent, E., Ma, N., Christensen, H., and Green, P. (**2013**), "The PASCAL CHiME speech separation and recognition challenge," *Computer Speech and Language* **27**(3), pp. 621–633. (Cited on page 33)

Ma, N., Brown, G. J., and Gonzalez, J. A. (**2015**a), "Exploiting top-down Source Models to improve binaural Localisation of multiple Sources in reverberant Environments," in *Proc. Interspeech.* (Cited on page 39)

Ma, N., Brown, G. J., and May, T. (**2015**b), "Robust localisation of multiple speakers exploiting deep neural networks and head movements," in *Proc. Interspeech*, pp. 3302–3306. (Cited on page 39)

Meutzner, H., Ma, N., Nickel, R., Schymura, C., and Kolossa, D. (**2017**), "Improving Audio-Visual Speech Recognition using Deep Neural Networks with Dynamic Stream Reliability Estimates," in *submitted for Proc. ICASSP.* (Cited on page 34)

Schymura, C., Ma, N., Brown, G. J., Walther, T., and Kolossa, D. (**2014**), "Binaural Sound Source Localisation using a Bayesian-network-based Blackboard System and Hypothesis-driven Feedback," in *Proc. Forum Acusticum*, Kraków, Poland. (Cited on page 39)

Schymura, C., Winter, F., Kolossa, D., and Spors, S. (**2015**), "Binaural Sound Source Localisation and Tracking using a Dynamic Spherical Head Model," in *Proc. Interspeech.* (Cited on page 39)

Appendices

A Documentation of Auditory Frontend

Auditory front-end

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 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 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 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 rate-maps, 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 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 framework supports online processing of the two-channel ear signals.

Credits

The Auditory front-end is developed by Remi Decorsière and Tobias May from DTU, and the rest of the Two!Ears team.

The Auditory front-end includes the following contributions from publicly available Matlab toolboxes or classes:

- Auditory Modeling Toolbox
- LTFAT

- Voicebox
- circVBuf

Overview

The purpose of the Auditory front-end 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 modelling or decision stages. This short description of the role of the Auditory front-end 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 organised 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

v: latest

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.

Getting started

The Auditory front-end 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.

All files 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 Available processors.

The implementation of the Auditory front-end 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.

Computation of an auditory representation

The following sections describe how the Auditory front-end 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 a later section.

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:

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

% Request the computation of ILDs
sOut = managerObj.addProcessor('ild');

% Request the processing
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 cell array containing a handle to the requested signal, here named `sOut`. In the Auditory front-end, signals are also objects. For example, for the output signal just generated:

```
>> sOut{1}

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 the Technical description. 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{1}.Data(:);
```

Note

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 Circular buffer.

v: latest

Input/output signals dimensions

The input signal `sIn`, for which a given auditory representation needs 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 as the 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.

Change parameters used for computation

For the requested representation

v: latest

Each individual processors that is supported by the Auditory front-end 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.,

```
Inter-aural Level Difference Extractor 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 =

  Parameters with properties:

    ild_hSizeSec: 0.0200
    ild_wSizeSec: 0.0400
```

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 processor can be visualised 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 filter bank (`'filterbank'`). The filter bank 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 filter bank, 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{1}` will be based on 16 channels, instead of 31.

Compute multiple auditory representations

Place multiple requests

v: latest

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 an 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 output 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 Matlab´s 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
            ild: {[1x1 TimeFrequencySignal]}
  innerhaircell: {[1x1 TimeFrequencySignal]  [1x1 TimeFrequencySignal]}
          input: {[1x1 TimeDomainSignal]  [1x1 TimeDomainSignal]}
           time: {[1x1 TimeDomainSignal]  [1x1 TimeDomainSignal]}
     filterbank: {[1x1 TimeFrequencySignal]  [1x1 TimeFrequencySignal]}
autocorrelation: {[1x1 CorrelationSignal]  [1x1 CorrelationSignal]}
```

Apart from the properties `bufferSize_s` and `isStereo` which are inherent properties of the data object (and discussed later in the Technical description), 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 filter bank output, type:

v: latest

```
>> 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]
```

How to plot the result

Plotting auditory representations is made very easy in the Auditory front-end. 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:

```
% Request the processing
managerObj.processSignal;

% Plot the ILDs in a separate figure
sOut{1}.plot;

% Create an empty figure with subplots
figure;
h1 = subplot(2,2,1);
h2 = subplot(2,2,2);
h3 = subplot(2,2,3);
h4 = subplot(2,2,4);

% Change plotting options to remove colorbar and reduce title size
p = genParStruct('bColorbar',0,'fsize_title',12);

% Plot additional representations
dataObj.innerhaircell{1}.plot(h1,p);
dataObj.innerhaircell{2}.plot(h2,p);
dataObj.filterbank{1}.plot(h3,p);
dataObj.filterbank{2}.plot(h4,p);
```

This script will produce the two figure windows displayed in Fig. 6. 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).

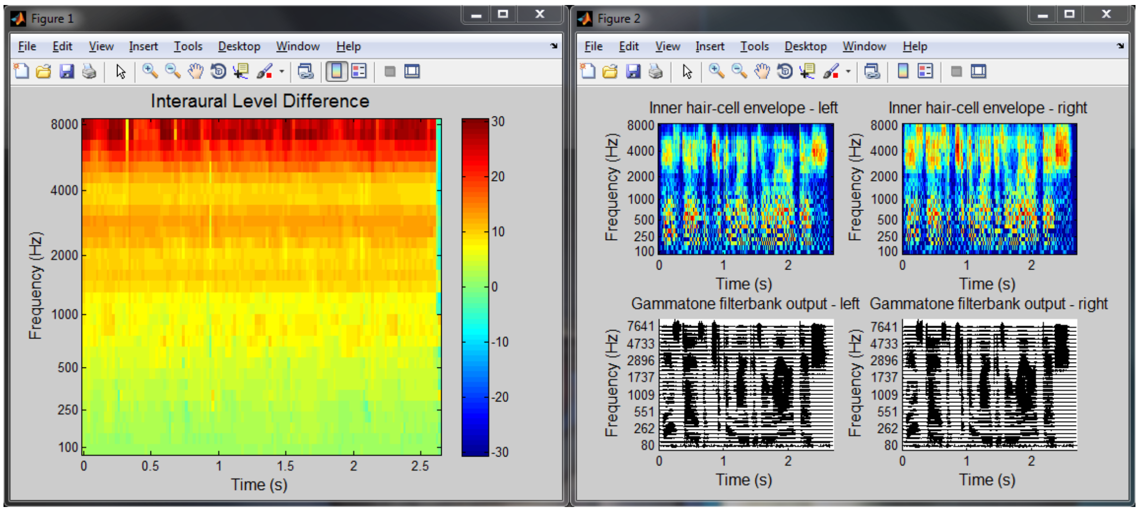


Figure 1
File Edit View Insert Tools Desktop Window Help
Interaural Level Difference
Frequency (Hz)
Time (s)
Figure 2
File Edit View Insert Tools Desktop Window Help
Inner hair-cell envelope - left
Inner hair-cell envelope - right
Gammatone filterbank output - left
Gammatone filterbank output - right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

Fig. 6 The two example figures generated by the demo script.

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 Auditory front-end where more complex operations than simple filtering are involved. A general description of the method used to ensure chunk-based processing is given in processChunk method and chunk-based compatibility.

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 Auditory front-end in a chunk-based scenario, in comparison to a batch scenario:

```
21  % Instantiation of data and manager objects
22  dataObj = dataObject([],fsHz,10,2);
23  managerObj = manager(dataObj);
```

Because the signal is not known before the processing is carried out, the data object cannot be initialised from the input signal. Hence, as is seen on line 22, 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 ope 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.

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

The processing is carried out on line 44 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. 7. 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.

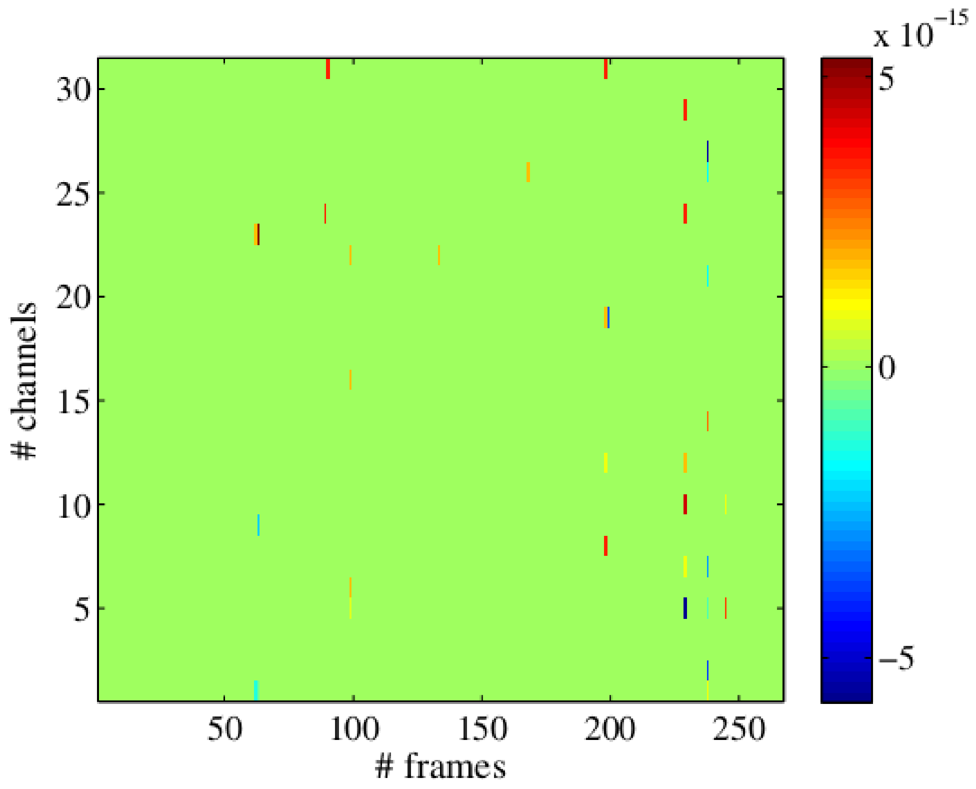


Fig. 7 Difference in ILDs obtained with online and offline processing

Feedback inclusion

A key concept of the Auditory front-end 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.

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

Placing a new request

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

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.

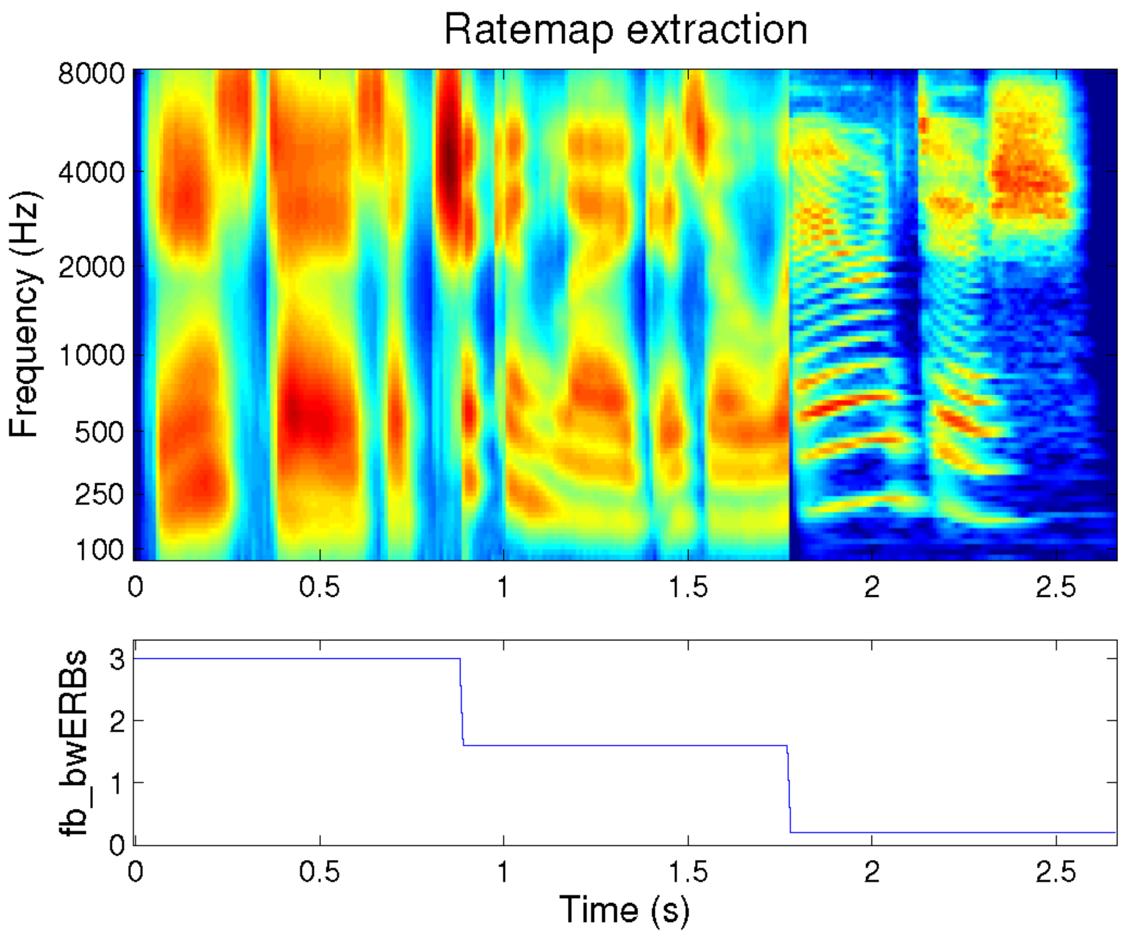


Fig. 8 Sharpening the frequency selectivity of the ear by means of feedback

Fig. 8 illustrates feedback capability of the Auditory front-end. 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). Fig. 8 illustrates how narrower auditory filters will reveal the harmonic structure of speech.

Note

v: latest

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

Deleting a processor

Deleting a processor is simply done by calling its `remove` method. Like 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.

List of commands

- Signal objects `sObj`

This section sums up the commands that could be relevant to a standard user of the Auditory front-end. 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]` (sa
`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 d
`pObj.hasParameters(parStruct)`	True if `pObj` used the exact values in `pa`
`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 subseque

Manager `mObj`

`manager(dObj)`	Constructor
`manager(dObj,name,param)`	Constructor with initial request
`mObj.addProcessor(name,param)`	Adds a processor (including eventual depend
`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

Technical description

Many different auditory models are available that can transform an input signal into an auditory representation. The actual design challenges behind the Auditory front-end 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 Auditory front-end and how the main constraints were tackled conceptually. In an effort to respect encapsulation and the hierarchical organisation of the objects, the sections are arranged in a “bottom-up” way: from the most fundamental objects to the more global processes.

All classes involved in the Auditory front-end 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.

v: latest

Data handling

Circular buffer

Memory pre-allocation of large arrays in Matlab is well known to be a critical operation for optimising computation time. The Auditory front-end, 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 initialised 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 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 Auditory front-end based on the third-party class provided by [Goebbert2014], which has been slightly modified to account for multi-dimensional data (instead of vector-only).

Circular buffer interface

v: latest

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 behaviour 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.

v: latest

Signal objects

Signals are implemented as objects in the Auditory front-end. To avoid code repetition and make better use of object-oriented concepts, signals are grouped according to their dimensions, 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 centre frequencies `cfHz`. Signals of such form are obtained from requesting, for example, `’filterbank’`, `’innerhaircell’`, `’ild’`,... In addition, time-frequency signals containing binary data (used e.g., in onset or offset mapping) have their own `BinaryMask` signal class.
- `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 centre 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 name tag 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 down-sampled (e.g., like a rate-map or an ILD) this value will be different from the input signal’s sampling frequency.

v: latest

- `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 initialising the internal buffer given some data.
- A `clearData` method for re-initialisation.
- 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 General considerations).
- 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 dimensions. Children classes therefore only implement their own constructor (which still calls the super-constructor) and their respective plotting routines.

Data objects

Description

Many signal objects are instantiated by the Auditory front-end (one per representation involved and per channel). To organise 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 Auditory front-end will be grouped according to its class in a distinct property of the `dataObject`, with name given by the signal `signal.Name` unique name tag. 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 (10 s 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 organisation

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).

Processors

Processors are at the core of the Auditory front-end. 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 processChunk method and chunk-based compatibility.

A detailed overview of all processors, with a list of all parameters they accept, is given in Available processors. 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.

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.

Properties

Each processor shares the properties:

- `Type` - describes formally the processing performed
- `Input` - list of input signal object handles
- `Output` - list of output signal object handles

- `isBinaural` - Flag indicating the need of left and right channel as input
- `FsHzIn` - Input signal sampling frequency (Hz)
- `FsHzOut` - Output signal sampling frequency (Hz)
- `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

These properties are populated automatically when using the Auditory front-end by the manager class which is described later in Manager. All of them, apart from `Type` are implemented as `Hidden` properties as they should not be relevant to the user but still need public access by other classes.

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

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.

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 imp

v: latest

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` check that the processor uses a specific set of parameter values
- `getCurrentParameters` returns a structure of the parameter values currently used by the processor.

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
processors with multiple input or outputs)

`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 Available processors. However, most of the available implementations are for batch processing, i.e., using one whole input signal at once. To be included in the Auditory front-end, 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 . 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 Auditory front-end 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 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

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: rate-map

A practical example to better illustrate the concepts described above is given in the following. The rate-map is conceptually a "framed" version of an IHC multi-channel envelope. The IHC envelope is a two-dimensional representation (time versus frequency), and the rate-map extraction is the same procedure repeated for every frequency channel. Hence the following is described for a single channel. To extract the rate-map, 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. 9.

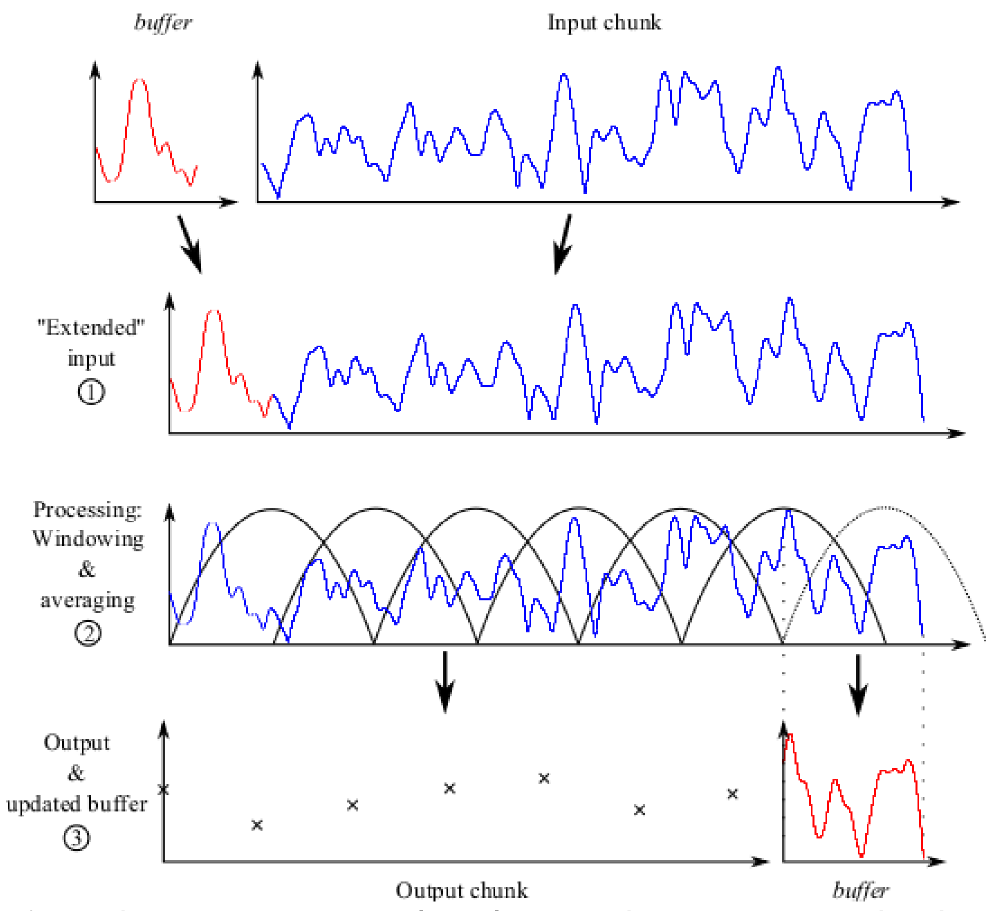


Fig. 9 Three steps for simple online windowing, given a chunk of input and an internal buffer.

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.

v: latest

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. This is unlikely to happen in practice however.

Particular case for filters

The processing performed by the Auditory front-end often involves filtering (e.g., in auditory filter bank processing, inner hair cell 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 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 Auditory front-end, 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.

Manager

The `manager` class is fundamental in the Auditory front-end. 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 Auditory front-end, only a single instance of this class is created. It is with this object that the user interacts.

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 Computation of an auditory representation. 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:
   1. Instantiate a corresponding processor (two if stereo signal)
   2. Instantiate the signal that will contain the output of the processor (two if stereo)
   3. Add the signal(s) to `dataObj`
   4. 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.
   5. A handle to the newly instantiated signal is stored similarly as output. This handle is stored further for the next element in the loop.
   6. 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` 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 re

v: latest

Carrying out the processing

As of the current Auditory front-end 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` methods 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`.

[Goebbert2014] Göbbert, J. H. (2014), "Circular double buffered vector buffer (`circVBuf.m`)," *Matlab file exchange:* http://www.mathworks.com/matlabcentral/fileexchange/4702 circvbuf, accessed: 2014-10-30.

Available processors

This section presents a detailed description of all processors that are currently supported by the Auditory front-end 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 Auditory front-end folder `/test` and can be used to reproduce the individual plots. 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 Computation of an auditory representation.

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:

- DC removal filter
- Pre-emphasis
- RMS normalisation
- Level reference and scaling
- 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 Table 5.

Table 5 List of parameters related to the auditory representation 't

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

v: latest

The influence of each individual pre-processing stage except for the level scaling is illustrated in Fig. 10, 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 panel 3. Panel 4 shows the influence of the pre-emphasis stage. The AGC can be used to equalise 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 synchronise 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.

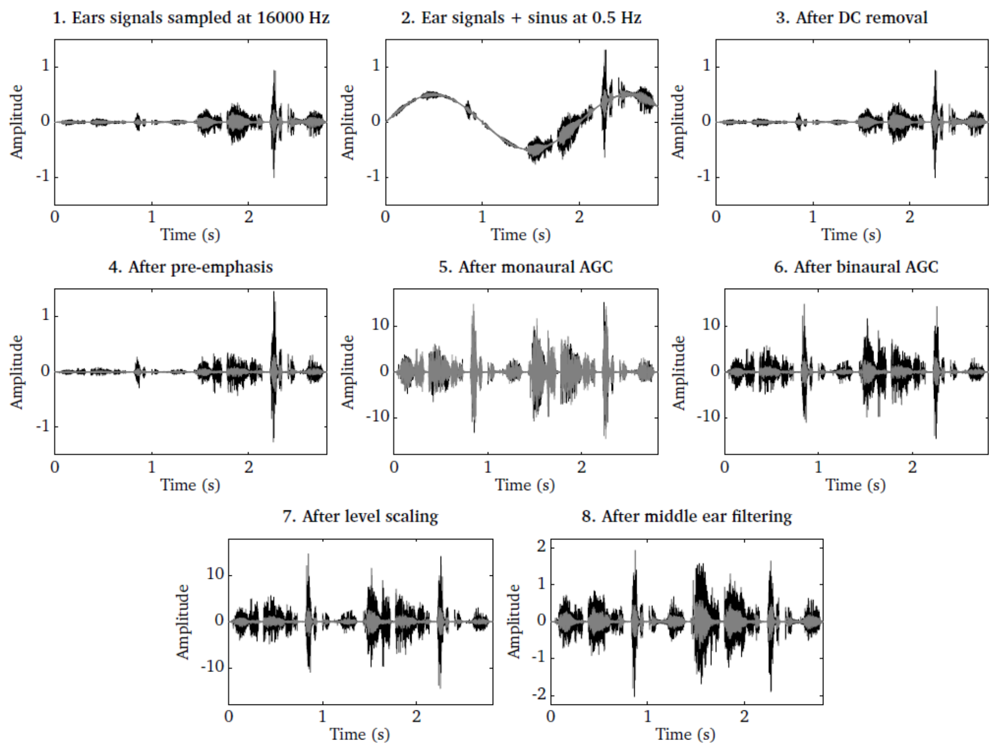


Fig. 10 Illustration of the individual pre-processing steps. 1) Ear signals consisting of two sentences recorded at different levels, 2) ear signals mixed with a 0.5 Hz humming, 3) ear signals after DC removal filter, 4) influence of pre-emphasis filter, 5) monaural RMS normalisation, 6) binaural RMS normalisation, 7) level scaling and 8) middle ear filtering.

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_cutoffHzD`

Pre-emphasis

A common pre-processing stage in the context of 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 and 1. Here, we set the coefficient to `pp_coefPreEmphasis = 0.97` by default according to [Young2006]. This pre-emphasis filter can be activated by setting the flag `pp_bPreEmphasis = true`.

RMS normalisation

A signal level normalisation stage is available which can be used to equalise 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 normalised 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 normalisation stage has also been suggested in the context of AMS feature extraction [Tchorz2003], which are described in Amplitude modulation spectrogram (modulationProc.m). The choice of the time constant is a balance between maintaining the level fluctuations across individual words and allowing the normalisation stage to follow sudden level changes.

The normalisation 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 normalisation, which will preserve the binaural cues (e.g. ITD and ILD) that are encoded in the signal. The RMS normalisation can be activated by the parameter `pp_bNormalizeRMS = true`.

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 Auditory front-end models requires the input to be represented in physical units (such as pascals, see the middle ear filtering stage below). Within the current Auditory front-end framework, the DRNL filter bank model requires this signal representation (see Dual-resonance non-linear filter bank (drnlProc.m)). 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 [Jepsen2008]. The implementation is adopted from the Auditory Modeling Toolbox [Soendergaard2013].

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 [Godde1994]. Its implementation is adopted from the Auditory Modeling Toolbox [Soendergaard2013], which derives the stapes velocity as the output [Lopez-Poveda2001], [Jepsen2008]. 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 filter bank model is used for the BM operation, because of its level-dependent nonlinearity, designed based on that representation (see Dual-resonance non-linear filter bank (drnlProc.m)). 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-Poveda2001] or from [Jepsen2008] can be used for the processing, by specifying the model `pp_middleEarModel = 'lopezpoveda'` or `pp_middleEarModel = 'jepsen'` respectively.

[Godde1994] 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.

[Tchorz2003] 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.

[Young2006] 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, http://htk.eng.cam.ac.uk.

Auditory filter bank

- Gammatone (`gammatoneProc.m`)
- Dual-resonance non-linear filter bank (`drnlProc.m`)

One central processing element of the Auditory front-end is the separation of incoming acoustic signals into different spectral bands, as it happens in the human inner ear. In psychoacoustic modelling, two different approaches have been followed over the years. One is the simulation of this stage by a *linear* filter bank composed of gammatone filters. This linear gammatone filter bank can be considered a standard element for auditory models and has therefore been included in the framework. A computationally more challenging, but at the same time physiologically more plausible simulation of this process can be realised by a *nonlinear* BM model, and we have implemented the DRNL model, as developed by [Meddis2001]. The filter bank representation is requested by using the name tag `'filterbank'`. The filter bank type can be controlled by the parameter `fb_type`. To select a gammatone filter bank, `fb_type` should be set to `’gammatone’` (which is the default), whereas the DRNL filter bank is used when setting `fb_type = 'drnl'`. Some of the parameters are common to the two filter bank, while some are specific, in which case their value is disregarded if the other type of filter bank was requested. Table 6 summarises all parameters corresponding to the `'filterbank'` request. Parameters specific to a filter bank type are separated by a horizontal line. The two filter bank implementations are described in detail in the following two subsections, along with their corresponding parameters.

Table 6 List of parameters related to the auditory representation `'fi`

Parameter	Default	Description
`fb_type`	`'gammatone'`	Filter bank 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

v: latest

Parameter	Default	Description
`fb_bwERBs`	`1.01859`	Filter bandwidth in ERB, `'gammatone'` -only
`fb_lowFreqHz`	`80`	Lowest characteristic frequency in Hz, `'gamm`
`fb_mocIpsi`	`1`	Ipsilateral MOC factor (0 to 1). Given as a scalar (across all frequency channels) or a vector (individual per fre channel), `'drnl'` -only
`fb_mocContra`	`1`	Contralateral MOC factor (0 to 1). Same format a `'fb_mocIpsi'` , `'drnl'` -only
`fb_model`	`'CASP'`	DRNL model (reserved for future extension)

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 Auditory Modeling Toolbox [Soendergaard2013]. The gammatone filters cover a frequency range between `fb_lowFreqHz` and `fb_highFreqHz` and are linearly spaced on the ERB scale [Glasberg1990]. In addition, the distance between adjacent filter centre frequencies on the ERB scale can be specified by `fb_nERBs` , which effectively controls the frequency resolution of the gammatone filter bank. There are three different ways to control the centre frequencies of the individual gammatone filters:

1. Define a vector with centre frequencies, e.g. `fb_cfHz = [100 200 500 ...]` . In this 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 centre 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 neighbouring filters.
3. It is also possible to specify `fb_lowFreqHz` , `fb_highFreqHz` and `fb_nERBs` . Starting at `fb_lowFreqHz` , the centre frequencies will be spaced at a distance of `fb_nERBs` on the ERB scale until the specified frequency range is covered. The centre frequency of the last filter will not necessarily match with `fb_highFreqHz` .

v: latest

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 centre 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 [Glasberg1990]. For instance, increasing `fb_bwERBs` leads to a broader filter shape. A full list of parameters is shown in Table 6.

The gammatone filter bank is illustrated in Fig. 11, 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 Hz and 8000 Hz. The output of each individual filter is shown in the right panel.

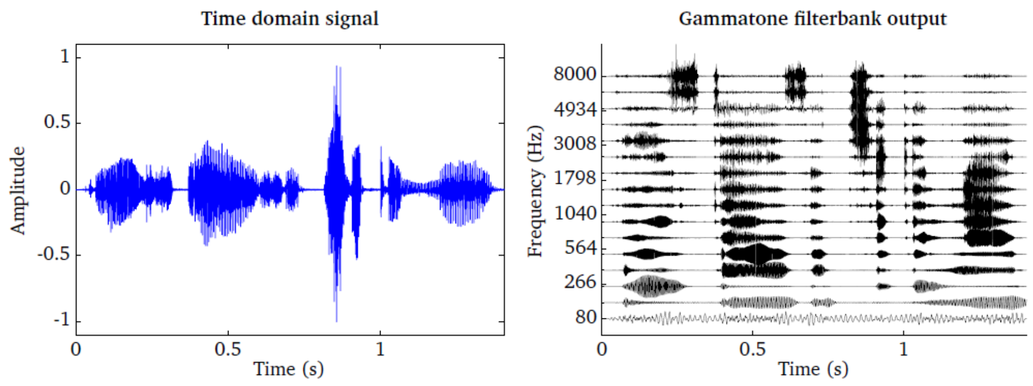


Fig. 11 Time domain signal (left panel) and the corresponding output of the gammatone processor consisting of 16 auditory filters spaced between 80 Hz and 8000 Hz (right panel).

Dual-resonance non-linear filter bank (`drnlProc.m`)

The DRNL filter bank 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 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 filter bank. This can reveal the implication of the BM nonlinearity and MOC feedback for activities such as speech perception in noise (see [Brown2010] for example) or source localisation. It is expected that the use of a nonlinear model, together with the adaptation loops (see Adaptation (adaptationProc.m)), 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 normalisation stage (see AGC in Pre-processing

v: latest

(preProc.m) for example). The structure of DRNL filter bank is based on the work of [Meddis2001]. 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. 12. Currently the implementation follows the model defined as CASP by [Jepsen2008], in terms of the detailed structure and operation, which is specified by the default argument `'CASP'` for `fb_model`.

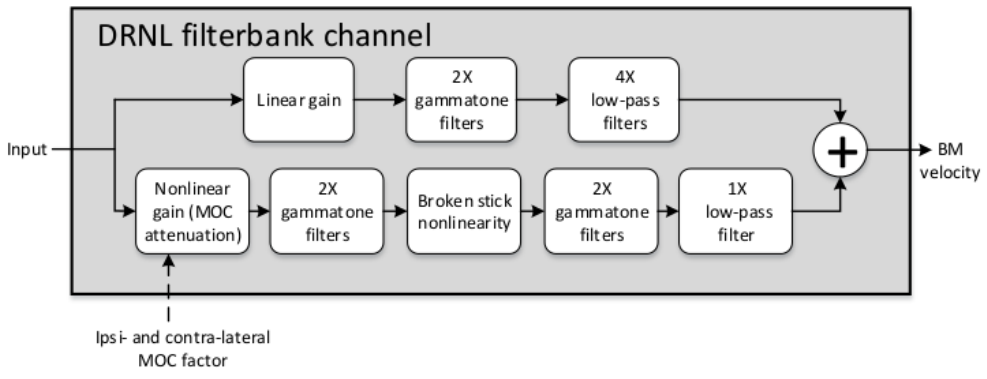


Fig. 12 Filter bank channel structure, following the model specification as default, with an additional nonlinear gain stage to receive feedback.

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 centre 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-Poveda2001], 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 filter bank model as the gain factor to be applied to the nonlinear path. This approach is used by [Ferry2007], where the attenuation caused by MOC the feedback at each of the filter bank 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. Table 6 summarises the parameters for DRNL the processor that can be controlled by the user. Note that `fb_cfHz` corresponds

v: latest

characteristic frequencies and not the *centre* frequencies as used in the gammatone filter bank, although they can have the same values for comparison. Otherwise, the characteristic frequencies can be generated in the same way as the centre frequencies for the gammatone filter bank.

Fig. 13 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 filter bank (left hand side), based on the right ear input signal shown in panel 1 of Fig. 10 (speech excerpt repeated twice with a level difference). The plots can be generated by running the script `DEMO_DRNL.m`. It should be noted that the CASP model of DRNL filter bank expects the input signal to be transformed to 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 filter bank (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 filter bank output, which shows the compressive nature of the nonlinear model responding to input level changes as described earlier.

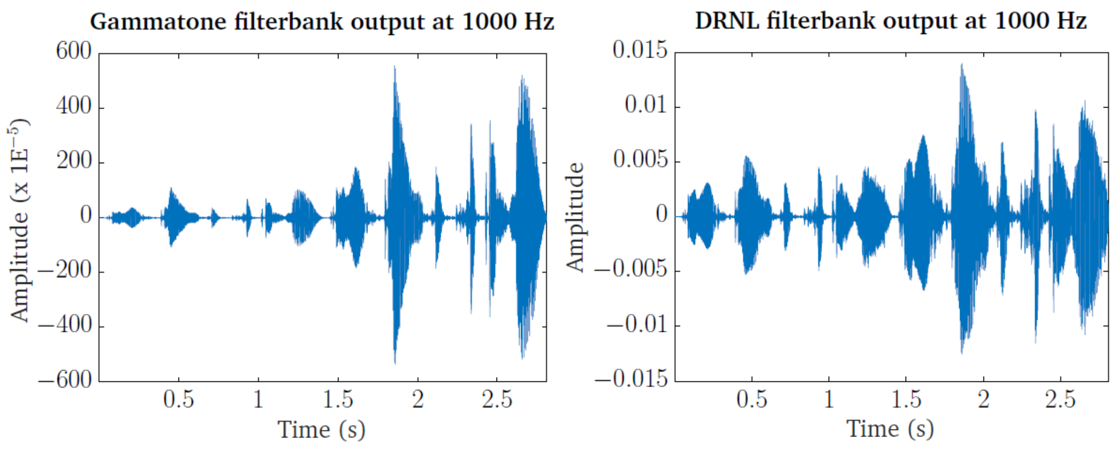


Fig. 13 The gammatone processor output (left panel) compared to the output of the DRNL processor (right panel), based on the right ear signal shown in panel 1 of Fig. 10, at 1 kHz centre or characteristic frequency. Note that the input signal is converted to the stapes velocity before entering both processors 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.

[Brown2010] 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.

[Ferry2007] 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.

v: latest

[Glasberg1990] *(1, 2)* 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.

[Jepsen2008] 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.

[Lopez-Poveda2001] 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.

[Soendergaard2013] 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.

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 Auditory Modeling Toolbox [Soendergaard2013]. 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 [Bernstein1996], [Bernstein1999]. 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 Table 7. A particular model can be selected by using the parameter `ihc_method`.

Table 7 List of supported IHC models¶

`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 [Da
`'joergensen'`	Hilbert transform and low-pass filtering at 150 Hz [Joergense
`'breebart'`	Half-wave rectification and low-pass filtering at 770 Hz [Bree
`'bernstein'`	Half-wave rectification, compression and low-pass filtering at

The effect of the IHC processor is demonstrated in Fig. 14, where the output of the gammatone filter bank 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.

v: latest

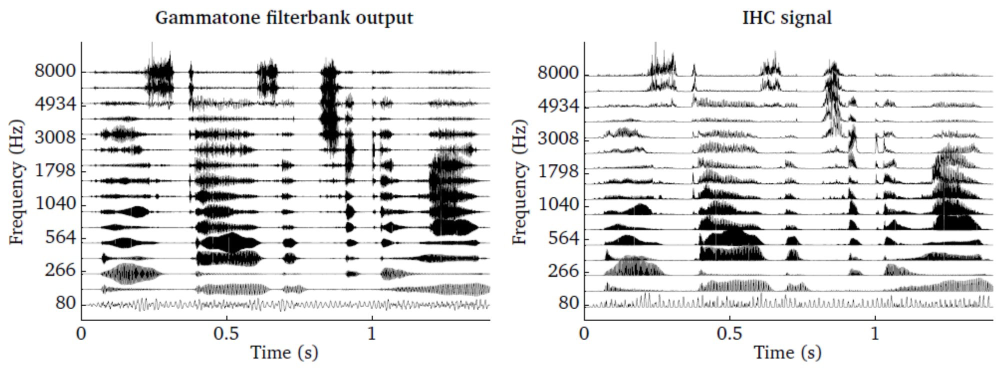


Fig. 14 Illustration of the envelope extraction processor. BM output (left panel) and the corresponding IHC model output using `ihc_method = ’dau’` (right panel).

[Bernstein1996] 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.

[Bernstein1999] *(1, 2)* Bernstein, L. R., van de Par, S., and Trahiotis, C. (1999), “The normalized interaural correlation: Accounting for NoS thresholds obtained with Gaussian and “low-noise” masking noise,” Journal of the Acoustical Society of America 106(2), pp. 870–876.

[Breebart2001] 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.

[Dau1996] 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.

[Joergensen2011] 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.

v: latest

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 [Smith1977], [Smith1983]. The function is adopted from the Auditory Modeling Toolbox [Soendergaard2013]. 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 [Pueschel1988], [Dau1996], [Dau1997a]. 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.

Table 8 List of parameters related to `'adaptation'`.¶

Parameter	Default	Description
`adpt_lim`	`10`	Overshoot limiting ratio
`adpt_mindB`	`0`	Lowest audible threshold of the in dB SPL
`adpt_tau`	`[0.005 0.050 0.129 0.253 0.500]`	Time constants of feedback
`adpt_model`	`''(empty)`	Implementation model `'adt_da` `'adt_puschel'`, or `'adt_breebart` can be used instead of the abov parameters (See Table 9)

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 t

v: latest

`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. Table 8 lists the parameters and their default values, and Table 9 lists the supported models. The output signal is expressed in 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 [Dau1996], [Jepsen2008], 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 filter bank. The adaptation processor recognises whether DRNL or gammatone processor is used in the chain and adjusts the input signal accordingly.

Table 9 List of supported models related to `'adaptation'`.¶

`adpt_model`	**Description**
`'adt_dau'`	Choose the parameters as in the models of [Dau1996], [Dau1997a This consists of 5 adaptation loops with an overshoot limit of 10 a a minimum level of 0 dB. This is a correction in regard to the mode described in [Dau1996], 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 [Pueschel1988]. This consists of 5 adaptation loops without overshoot limiting (`adpt_lim=0`). The adaptation 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

v: latest

The effect of the adaptation processor - the exaggeration of rapid variations - is demonstrated in Fig. 15, where the output of the IHC model from the same input as used in the example of Inner hair-cell (ihcProc.m) (the right panel of Fig. 14) is compared to the adaptation output by running the script `DEMO_Adaptation.m`.

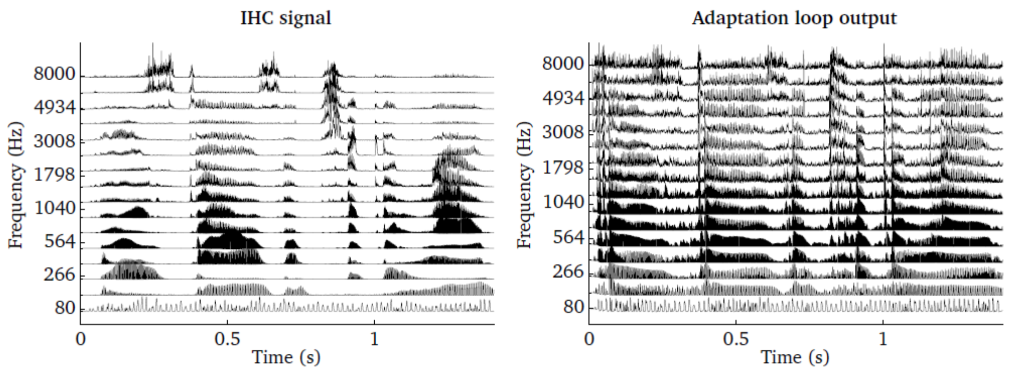


Fig. 15 Illustration of the adaptation processor. IHC output (left panel) as the input to the adaptation processor and the corresponding output using `adpt_model='adt_dau'` (right panel).

[Dau1997a] *(1, 2)* Dau, T., Püschel, D., and Kohlrausch, A. (1997a), "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.

[Pueschel1988] *(1, 2)* Püschel, D. (1988), "Prinzipien der zeitlichen Analyse beim Hören," Ph.D. thesis, University of Göttingen.

[Smith1977] Smith, R. L. (1977), "Short-term adaptation in single auditory nerve fibers: some poststimulatory effects," J Neurophysiol 40(5), pp. 1098–1111.

[Smith1983] 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.

Auto-correlation (`autocorrelationProc.m`)

Auto-correlation is an important computational concept that has been extensively studied in the context of predicting human pitch perception [Licklider1951], [Meddis1991]. To measure the amount of periodicity that is present in individual frequency channels, the ACF is computed in the 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 normalised 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 milliseconds – 30 milliseconds.

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 [Rabiner1977]. This pre-processing can be activated by the flag `ac_bCenterClip` and the following nonlinear operations can be selected for `ac_ccMethod` : centre clip and compress `’clc’` , centre clip `’cc’` , and combined centre and peak clip `’sgn’` . The percentage of centre 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 generalised ACF has been suggested by [Tolonen2000], 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 [Tolonen2000]. A list of all ACF-related parameters is given in Table 10. Note that these parameters will influence the pitch processor, which is described in Pitch (pitchProc.m).

Table 10 List of parameters related to the auditory representation `'autocorrelation'` .¶

Parameter	Default	Description
`ac_wname`	`'hann'`	Window type
`ac_wSizeSec`	`0.02`	Window duration in s

v: latest

	lt	Description
`ac_hSizeSec`	`0.01`	Window step size in s
`ac_bCenterClip`	`false`	Activate centre clipping
`ac_clipMethod`	`'clp'`	Centre clipping method `'clc'`, `'clp'`, or `'sgn'`
`ac_clipAlpha`	`0.6`	Centre clipping threshold within `[0,1]`
`ac_K`	`2`	Exponent in ACF

A demonstration of the ACF processor is shown in Fig. 16, 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 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 analysed speech signal. This relationship is exploited by the pitch processor, which is described in Pitch (pitchProc.m).

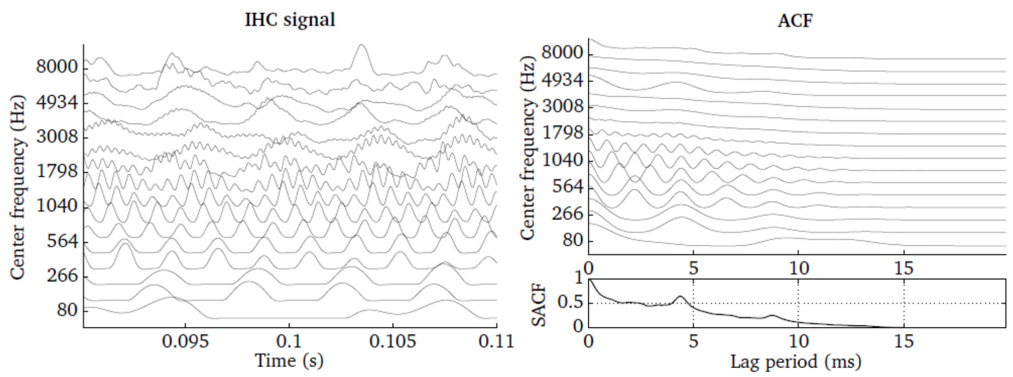


Fig. 16 IHC representation of a speech signal shown for one time frame of 20 ms duration (left panel) and the corresponding ACF (right panel). The SACF summarises the ACF across all frequency channels (bottom right panel).

[Licklider1951] Licklider, J. C. R. (1951), “A duplex theory of pitch perception,” Experientia (4), pp. 128–134.

[Meddis1991] 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.

[Rabiner1977] 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.

[Tolonen2000] *(1, 2)* 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.

Rate-map (`ratemapProc.m`)

The rate-map represents a map of auditory nerve firing rates [Brown1994] and is frequently employed as a spectral feature in CASA systems [Wang2006], ASR [Cooke2001] and speaker identification systems [May2012]. The rate-map 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 rate-map can be interpreted as an auditory spectrogram. Depending on whether the rate-map 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 rate-map parameters are listed in Table 11.

Table 11 List of parameters related to `'ratemap'`.¶

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

The rate-map is demonstrated by the script `DEMO_Ratemap` and the corresponding plots are presented in Fig. 17. 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 rate-map representation scaled in dB is presented in the right panel.

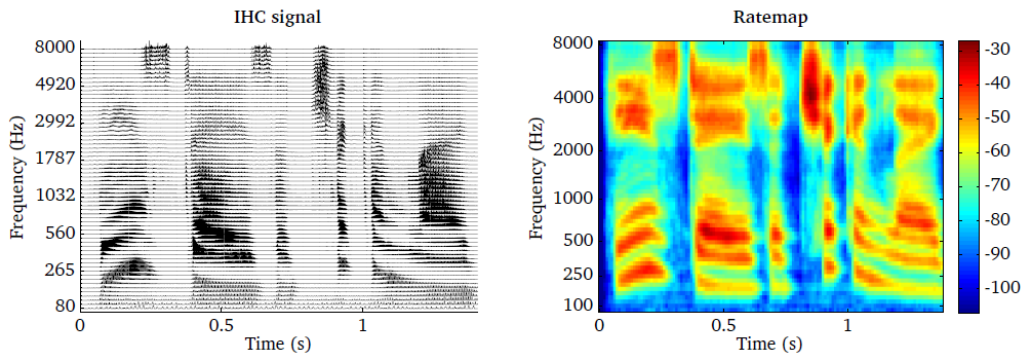


Fig. 17 IHC representation of s speech signal using 64 auditory filters (left panel) and the corresponding rate-map representation (right panel).

[Brown1994] Brown, G. J. and Cooke, M. P. (1994), "Computational auditory scene analysis," Computer Speech and Language 8(4), pp. 297–336.

[Cooke2001] 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.

[May2012] 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.

[Wang2006] Wang, D. L. and Brown, G. J. (Eds.) (2006), Computational Auditory Scene Analysis: Principles, Algorithms and Applications, Wiley / IEEE Press.

Spectral features (`spectralFeaturesProc.m`)

In order to characterise 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 [Peeters2011]. All spectral features summarise the spectral content of the rate-map 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 centre of gravity of the rate-map and is one of the most frequently-used timbre parameters [Tzanetakis2002], [Jensen2004], [Peeters2011]. The centroid is normalised by the highest rate-map centre frequency to reduce the influence of the gammatone parameters.
2. `'spread'` : The spectral spread describes the average deviation of the rate-map 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 normalised by the highest rate-map centre 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 [Jensen2004], [Peeters2011]. 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 rate-map by its squared centre frequency and integrating this representation across all frequency channels [Jensen2004]. To reduce the sensitivity of this feature to the overall signal level, the high-frequency content feature is normalised by the rate-map integrated across-frequency.
5. `'crest'` : The SCM is defined as the ratio between the maximum value and the arithmetic mean and can be used to characterise the peakiness of the rate-map. The feature value is low for signals with a

flat spectrum and high for a rate-map with a distinct spectral peak [Peeters2011], [Lerch2012].

6. `'decrease'` : The spectral decrease describes the average spectral slope of the rate-map representation, putting a stronger emphasis on the low frequencies [Peeters2011].
7. `'entropy'` : The entropy can be used to capture the peakiness of the spectral representation [Misra2004]. The resulting feature is low for a rate-map with many distinct spectral peaks and high for a flat rate-map spectrum.
8. `'flatness'` : The 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) [Peeters2011].
9. `'irregularity'` : The spectral irregularity quantifies the variations of the logarithmically-scaled rate-map across frequencies [Jensen2004].
10. `'kurtosis'` : The excess kurtosis measures whether the spectrum can be characterised by a Gaussian distribution [Lerch2012]. This feature will be zero for a Gaussian distribution.
11. `'skewness'` : The spectral skewness measures the symmetry of the spectrum around its arithmetic mean [Lerch2012]. 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` [Tzanetakis2002] and `sf_ro_perc = 0.95` [Scheirer1997], [Peeters2011]. The roll-off feature is normalised by the highest rate-map centre 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 rate-map across adjacent frames [Lerch2012]. It has been suggested to be useful for the distinction of music and speech signals, since music has a higher rate of change [Scheirer1997].
14. `'variation'` : The spectral variation is defined as one minus the normalised correlation between two adjacent time frames of the rate-map [Peeters2011].

A list of all parameters is presented in Table 12.

Table 12 List of parameters related to `'spectral_features'` .¶

Parameter	Default	Description
`sf_requests`	`'all'`	List of requested spectral features (e.g. `'flux'`). Type `help spectralFeaturesProc` in the Matlab command windo 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

Parameter Default Description

The extraction of spectral features is demonstrated by the script `Demo_SpectralFeatures.m`, which produces the plots shown in Fig. 18. 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 rate-map representation.

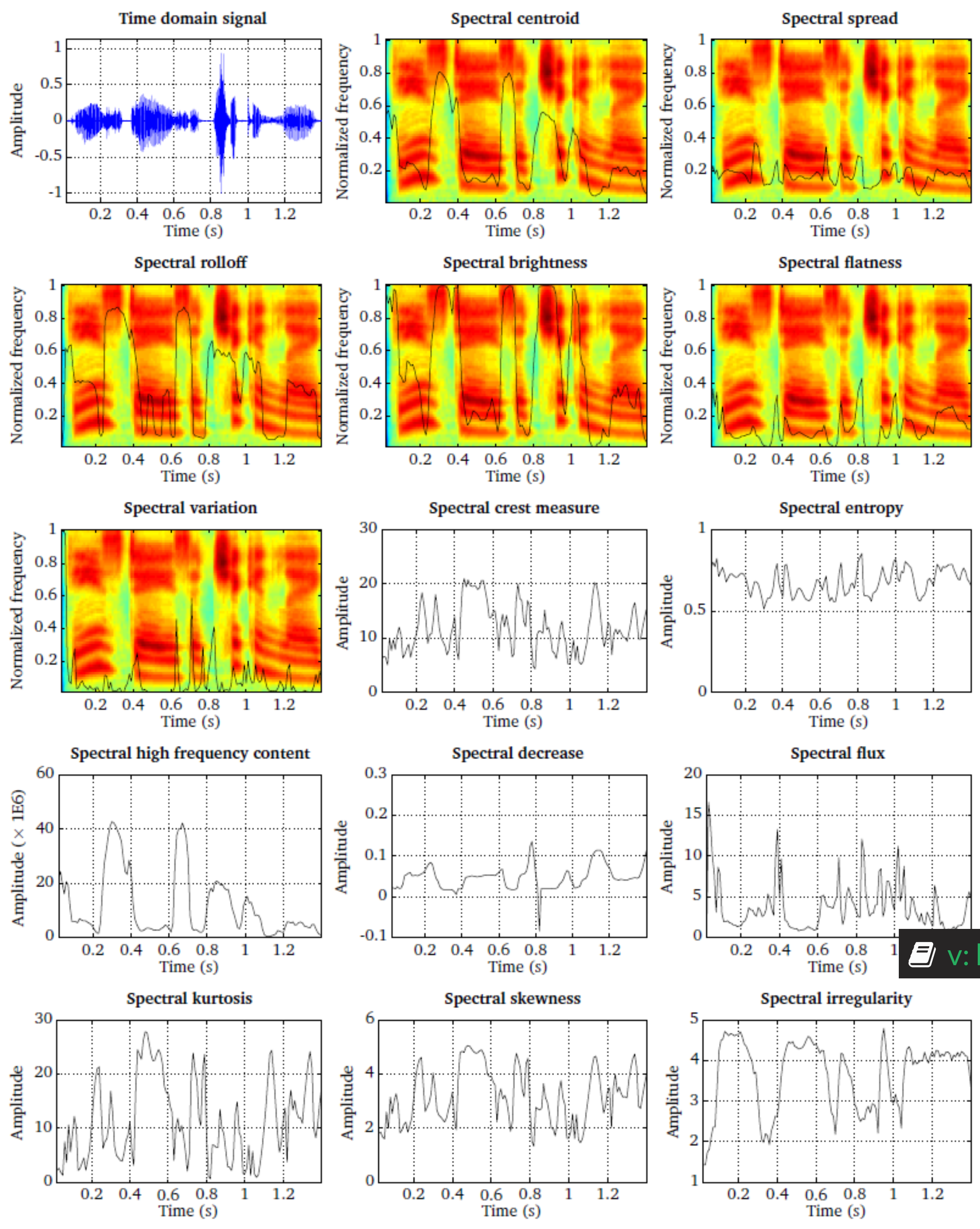


Fig. 18 Speech signal and 14 spectral features that were extracted based on the rate-map representation.

v: latest

[Jensen2004] (1, 2, 3, 4) 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.

[Lerch2012] (1, 2, 3, 4) Lerch, A. (2012), An Introduction to Audio Content Analysis: Applications in Signal Processing and Music Informatics, John Wiley & Sons, Hoboken, NJ, USA.

[Misra2004] 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.

[Peeters2011] *(1, 2, 3, 4, 5, 6, 7, 8)* 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.

[Scheirer1997] *(1, 2)* 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.

[Tzanetakis2002] *(1, 2)* 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.

Onset strength (`onsetProc.m`)

According to [Bregman1990], common onsets and offsets across frequency are important grouping cues that are utilised by the human auditory system to organise and integrate sounds originating from the same source. The onset processor is based on the rate-map representation, and therefore, the choice of the rate-map parameters, as listed in Table 11, 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 rate-map. Onset are detected by measuring the frame-based increase in energy of the rate-map representation. This detection is performed based on the logarithmically-scaled energy, as suggested by [Klapuri1999]. 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 Table 13.

Table 13 List of parameters related to `'onset_strength'` ¶

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

The resulting onset strength expressed in *decibel*, which is a function of time frame and frequency channel, is shown in Fig. 19. 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.

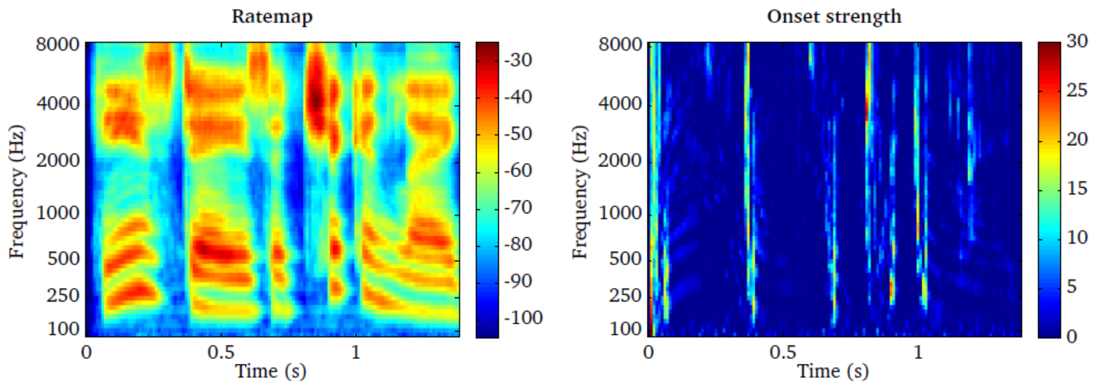


Fig. 19 Rate-map representation (left panel) of speech and the corresponding onset strength in decibel (right panel).

[Klapuri1999] 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.

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 rate-map parameters as listed in Table 11 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 Table 13.

Table 14 List of parameters related to `'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. 20. 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 synchronised across frequency.

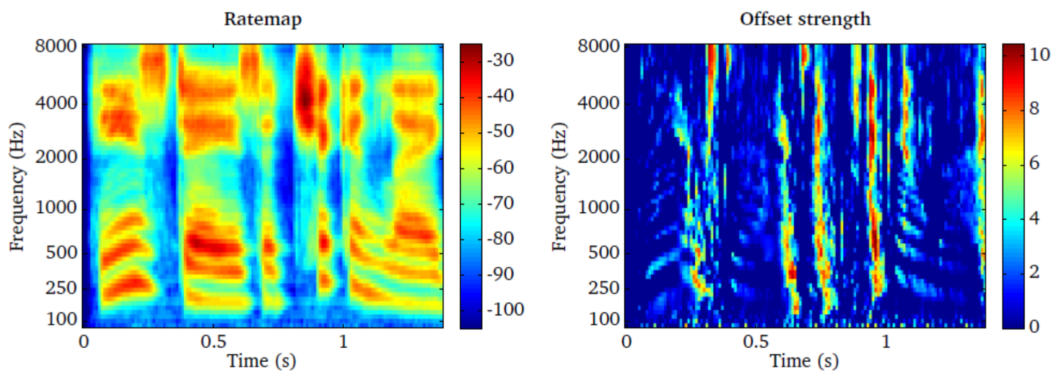


Fig. 20 Rate-map representation (left panel) of speech and the corresponding offset strength in *decibel* (right panel).

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 organise and group the acoustic input according to individual auditory events. The required processing is similar for both onsets and offsets, and is summarised 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 Onset strength (onsetProc.m) and Offset strength (offsetProc.m), 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 [Turgeon2002], 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 TF units are extracted using Matlab’s image labelling tool `bwlabel`. The binary transient map will only retain those transients which consists of at least `trm_minSpread` connected TF 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 rate-map level is below a pre-defined threshold, as determined by `trm_minValuedB`. A summary of all parameters is given in Table 15.

Table 15 List of parameters related to `'onset_map'` and `'offset_map'`.

Parameter	Default	Description
`trm_fuseWithinSec`	`30E-3`	Time constant below which transients are fu
`trm_minSpread`	`5`	Minimum number of connected TF units
`trm_minStrengthdB`	`3`	Minimum onset strength in dB

v: latest

`trm_minValuedB`	`-80`	Minimum rate-map level in dB

To illustrate the benefit of selecting onset and offset information, a rate-map representation is shown in Fig. 21 (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.

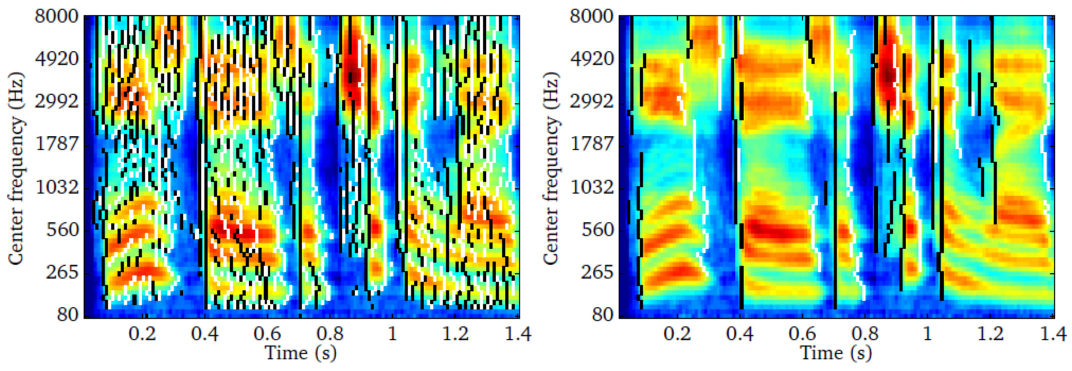


Fig. 21 Detected onsets and offsets indicated by the black and white vertical bars. The left panels shows all onset and offset events, whereas the right panel applies temporal and across-frequency constraints in order to retain the most salient onset and offset events.

[Turgeon2002] 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.

Pitch (`pitchProc.m`)

Following [Slaney1990], [Meddis2001], [Meddis1997], the sub-band 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. 16. 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 Table 16. 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 Auto-correlation (autocorrelationProc.m).

Table 16 List of parameters related to `'pitch'`.¶

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

The task of pitch estimation is demonstrated by the script `DEMO_Pitch` and the corresponding SACF plots are presented in Fig. 22. The pitch is estimated for an anechoic speech signal (top left panel). The corresponding 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

v: latest

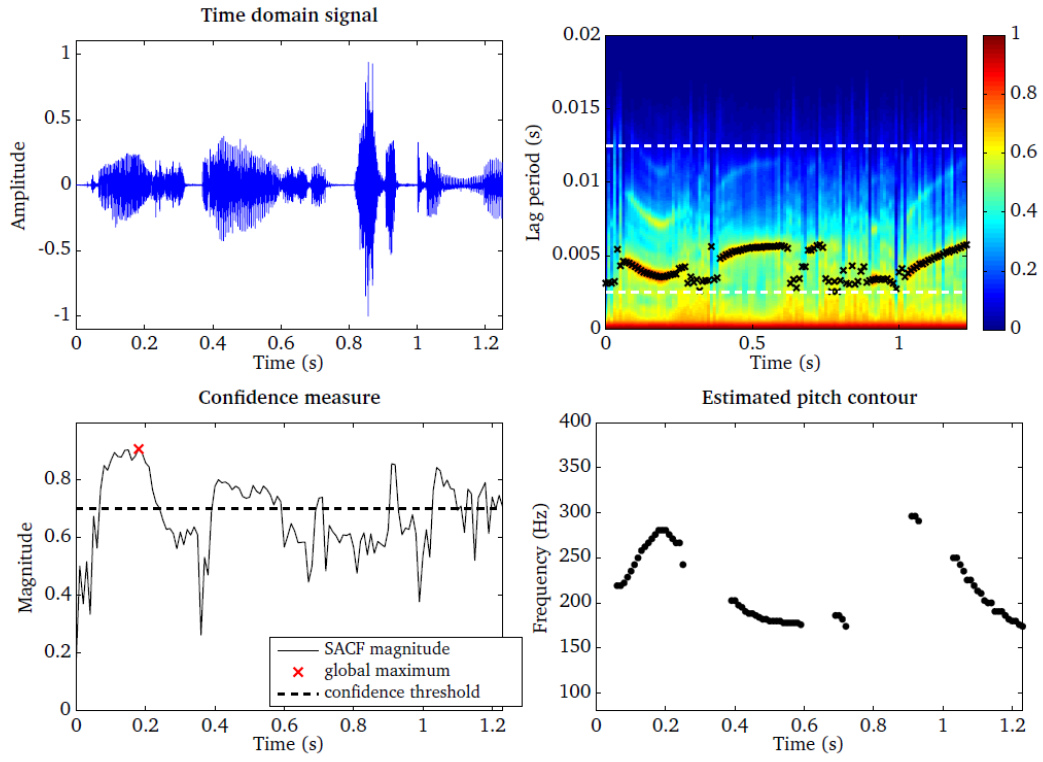


Fig. 22 Time domain signal (top left panel) and the corresponding SACF (top right panel). The confidence measure based on the SACF magnitude is used to select reliable pitch estimates (bottom left panel). The final pitch estimate is post-processed by a median filter (bottom right panel).

[Meddis1997] 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.

[Meddis2001] 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.

[Slaney1990] 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.

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 [Guinan2006]. 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 Auditory front-end, 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 [Clark2012], except that the auditory nerve processing model is simplified as the ratemap processor in Auditory front-end.

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 [Liberman1988]. The amplitude relationship was adopted from the work of [Clark2012]. The time course and delay of the feedback activity, such as in the work of [Backus2006], 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 nonlinear path as the final multiplicative gain factor. Table 17 lists the parameters for the processor, including the above-mentioned two. The other two parameters `moc_mocThresholdRatedB` and `moc_mocMaxAttenuationdB` are specified such that the input level- attenuation relationship is fitted best to the data of [Liberman1988] which is scaled within a range of 0 dB to 40 dB by [Clark2012].

Table 17 List of parameters related to the auditory representation

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

Fig. 23 shows, firstly on the left panel, the input-output characteristics of the MOC processor, using on-frequency stimulation from tones at 520 Hz and 3980 Hz, same as in the work of [Liberman1988]. 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 [Liberman1988], which is also shown on the plot. An example of input signal-DRNL output pair at 40 dB input level is shown on 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`.

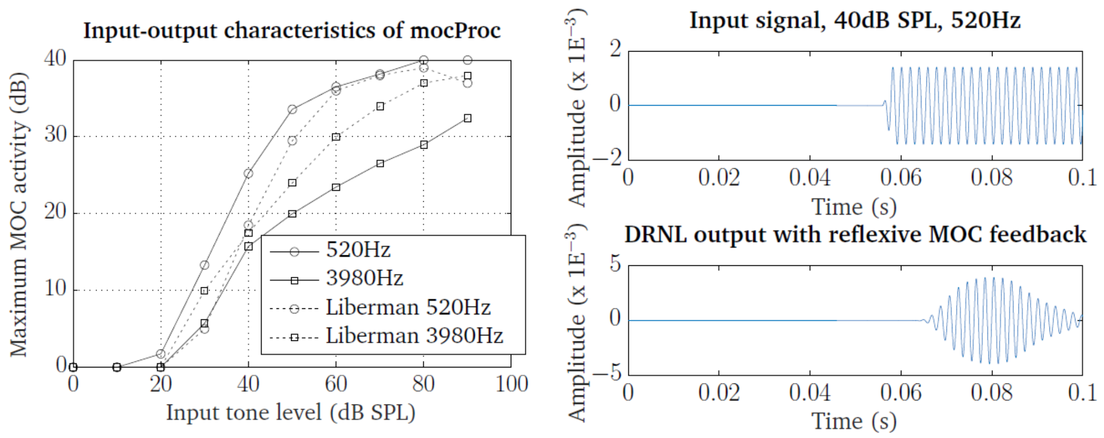


Fig. 23 Left panel: input-output characteristics of the MOC processor for on-frequency tone stimulus at 520 Hz and 3980 Hz. The data set of [Liberman1988], scaled as in the work of [Clark2012], is also shown for comparison. Right panel: DRNL processor output (bottom) from a 50 ms tone at 40 dB SPL and 520 Hz (top), with the reflexive feedback operating.

[Backus2006] 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.

v: latest

[Clark2012] *(1, 2, 3)* 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.

[Guinan2006] 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, http://www.ncbi.nlm.nih.gov/pubmed/17086072.

[Liberman1988] *(1, 2, 3, 4)* 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, http://jn.physiology.org/content/60/5/1779.

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 [Kollmeier1994], [Tchorz2003], [Kim2009], [May2013a], [May2014a], [May2014b]. The use of linearly-scaled modulation filters is, however, not consistent with psychoacoustic data on modulation detection and masking in humans [Bacon1989], [Houtgast1989], [Dau1997a], [Dau1997b], [Ewert2000]. As demonstrated by [Ewert2000], the processing of envelope fluctuations can be described effectively by a second-order band-pass filter bank with logarithmically-spaced centre frequencies. Moreover, it has been shown that an AMS feature representation based on an auditory-inspired modulation filter bank with logarithmically-scaled modulation filters substantially improved the performance of computational speech segregation in the presence of stationary and fluctuating interferers [May2014c]. In addition, such a processing based on auditory-inspired modulation filters has recently also been successful in speech intelligibility prediction studies [Joergensen2011], [Joergensen2013]. 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 analysed 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 filter bank responses are shown in Fig. 24. The linear modulation filter bank is implemented in the frequency domain, whereas the logarithmically-scaled filter bank is realised by a band of second-order IIR Butterworth filters with a constant-Q factor of 1. The modulation filter with the lowest centre frequency is always implemented as a low-pass filter, as illustrated in the right panel of Fig. 24.

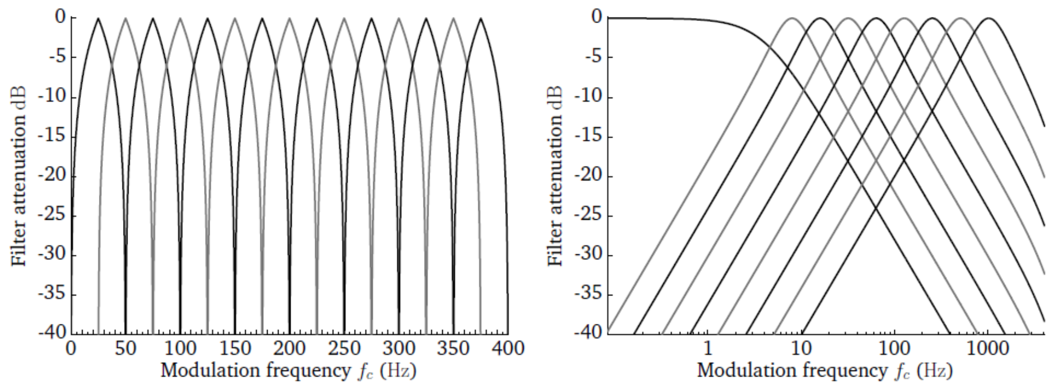


Fig. 24 Transfer functions of 15 linearly-scaled (left panel) and 9 logarithmically-scaled (right panel) modulation filters.

Similarly to the gammatone processor described in Gammatone (gammatoneProc.m), there are different ways to control the centre 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 centre 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 centre frequencies will be logarithmically-spaced at integer powers of two, e.g. 2^2, 2^3, 2^4 ... 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 Table 18.

Table 18 List of parameters related to `'ams_features'`.¶

Parameter	Default	Description
`ams_fbType`	`'log'`	Filter bank type (`'lin'` or `'log'`)
`ams_nFilter`	`[]`	Number of modulation filters (integer)
`ams_lowFreqHz`	`4`	Lowest modulation filter centre frequency in
`ams_highFreqHz`	`1024`	Highest modulation filter centre frequency ir
`ams_cfHz`	`[]`	Vector of modulation filter centre frequencie
`ams_dsRatio`	`4`	Downsampling ratio of the IHC representatic
`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. 25. 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.

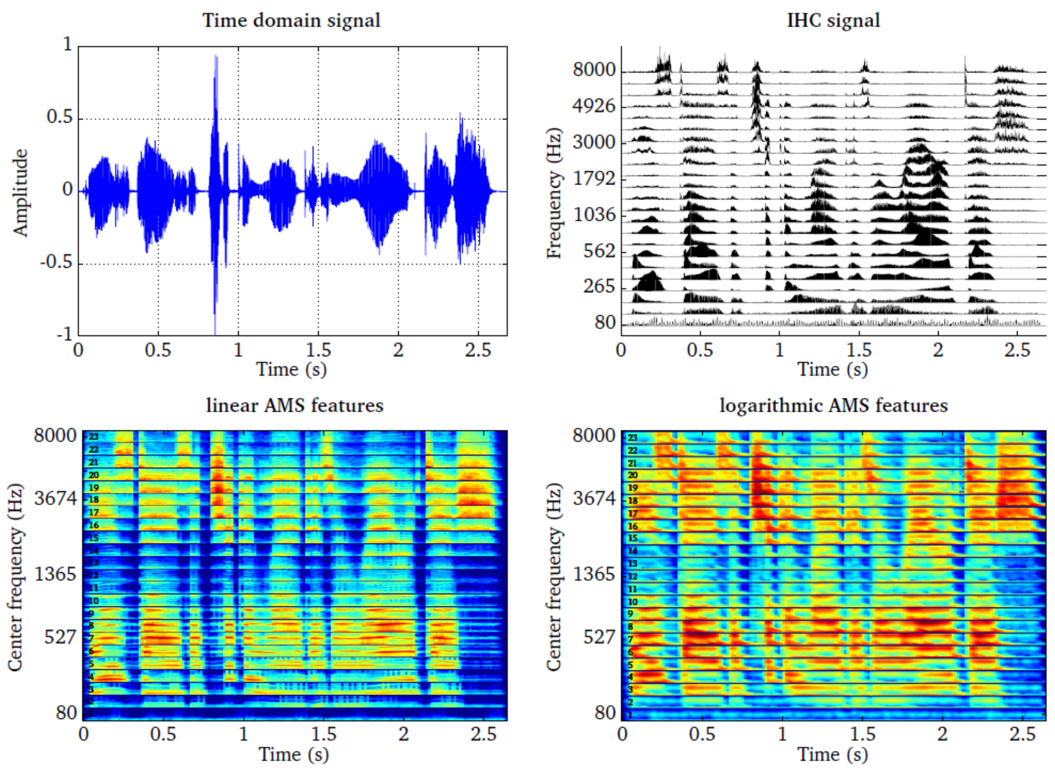


Fig. 25 Speech signal (top left panel) and the corresponding IHC representation (top right panel) using 23 frequency channels spaced between 80 and 8000 Hz. Linear AMS features (bottom left panel) and logarithmic AMS features (bottom right panel). The response of the modulation filters are stacked on 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.

[Bacon1989] 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.

[Dau1997b] Dau, T., Püschel, D., and Kohlrausch, A. (1997b), “Modeling auditory processing of amplitude modulation. II. Spectral and temporal integration,” Journal of the Acoustical Society of America 102(5), pp. 2906–2919.

[Ewert2000] *(1, 2)* 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.

[Houtgast1989] Houtgast, T. (1989), “Frequency selectivity in amplitude-modulation detection,” Journal of the Acoustical Socie
America 85(4), pp. 1676–1680.

v: latest

[Joergensen2013] 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.

[Kim2009] 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.

[Kollmeier1994] 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.

[May2013a] 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.

[May2014a] May, T. and Dau, T. (2014), “Requirements for the evaluation of computational speech segregation systems,” Journal of the Acoustical Society of America 136(6), pp. EL398– EL404.

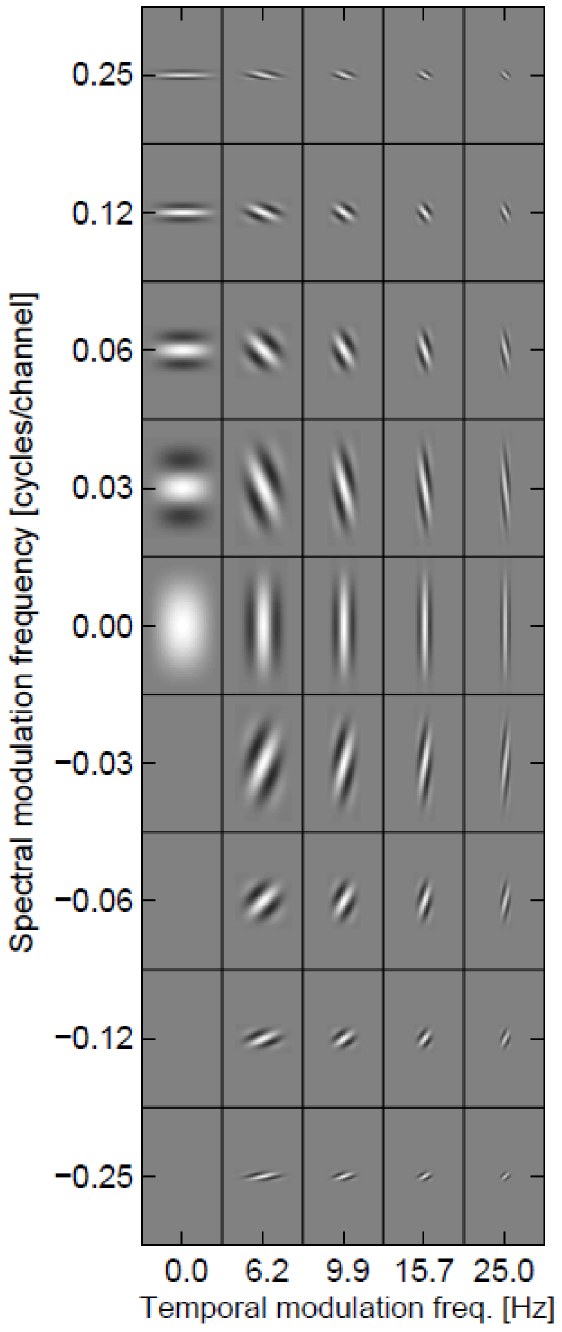
[May2014b] May, T. and Gerkmann, T. (2014), “Generalization of supervised learning for binary mask estimation,” in International Workshop on Acoustic Signal Enhancement, Antibes, France.

[May2014c] May, T. and Dau, T. (2014), “Computational speech segregation based on an auditory-inspired modulation analysis,” Journal of the Acoustical Society of America 136(6), pp. 3350-3359.

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 [Theunissen2001], [Qiu2003]. This response characteristic of neurons can be described by the so-called STRF. As suggested by [Qiu2003], the STRF can be effectively modelled by two-dimensional (2D) Gabor functions. Based on these findings, a spectro-temporal filter bank consisting of 41 Gabor filters has been designed by [Schaedler2012]. This filter bank has been optimised for the task of ASR, and the respective real parts of the 41 Gabor filters is shown in Fig. 26.

The input is a log-compressed rate-map 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 dimensions of the resulting Gabor feature space, a selection of representative rate-map frequency channels will be automatically performed for each Gabor filter [Schaedler2012]. For instance, the reference implementation based on 23 frequency channels produces a 311 dimensional Gabor feature space.



Spectral modulation frequency [cycles/channel]
0.25
0.12
0.06
0.03
0.00
−0.03
−0.06
−0.12
−0.25
0.0
6.2
9.9
15.7
25.0
Temporal modulation freq. [Hz]

Fig. 26 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. 27. A log-compressed rate-map 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 rate-map parameters have been adjusted to meet the specifications as recommended in the ETSI standard [ETSIES]. 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.

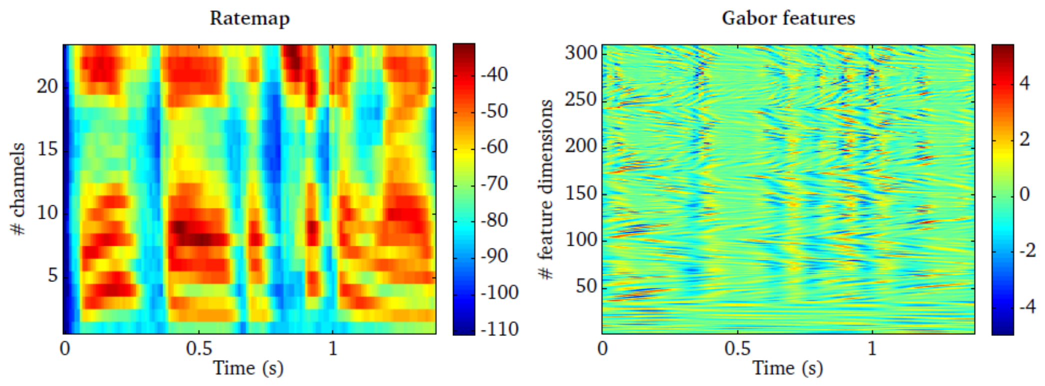


Fig. 27 Rate-map representation of a speech signal (left panel) and the corresponding output of the Gabor feature processor (right panel).

[ETSIES] 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," http://www.etsi.org.

[Qiu2003] *(1, 2)* 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.

[Schaedler2012] *(1, 2)* 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.

[Theunissen2001] 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.

v: latest

Cross-correlation (`crosscorrelationProc.m`)

The IHC representations of the left and the right ear signals is used to compute the normalised CCF in the FFT domain for short time frames of `cc_wSizeSec` duration with a step size of `cc_hSizeSec` . The CCF is normalised by the auto-correlation sequence at lag zero. This normalised CCF is then evaluated for time lags within `cc_maxDelaySec` (e.g., [-1 ms, 1 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 Table 19. Note that the choice of these parameters will influence the computation of the ITD and the IC processors, which are described in Interaural time differences (itdProc.m) and Interaural coherence (icProc.m), respectively.

Table 19 List of parameters related to `'crosscorrelation'` .¶

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

The script `DEMO_Crosscorrelation.m` demonstrates the functionality of the CCF function and the resulting plots are shown in Fig. 28. 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 centred around positive time lags, indicating that the source is closer to the right ear. This is even more evident by looking at the SCCF, as shown in the bottom right panel.

v: latest

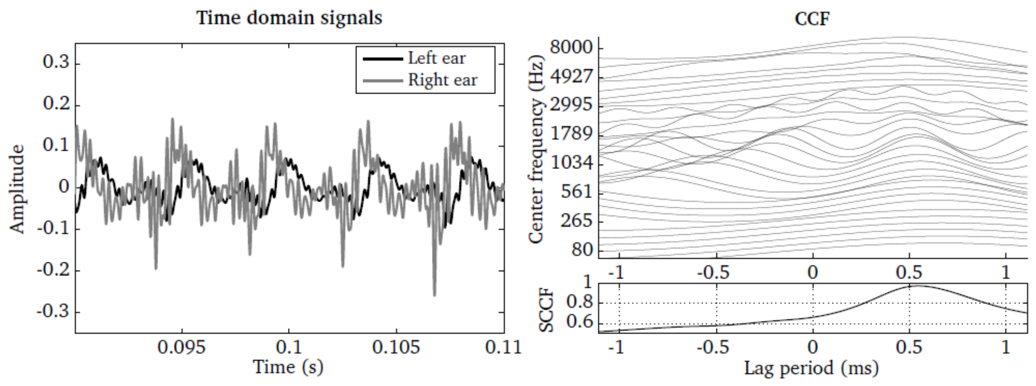


Fig. 28 Left and right ear signals shown for one time frame of 20 ms duration (left panel) and the corresponding CCF (right panel). The SCCF summarises the CCF across all auditory channels (bottom right panel).

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 normalised CCF. This estimation is further refined by a parabolic interpolation stage [May2011], [May2013b]. The ITD processor does not have any adjustable parameters, but it relies on the CCF described in Cross-correlation (crosscorrelationProc.m) and its corresponding parameters (see Table 19). 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. 29. 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 TF 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 TF units.

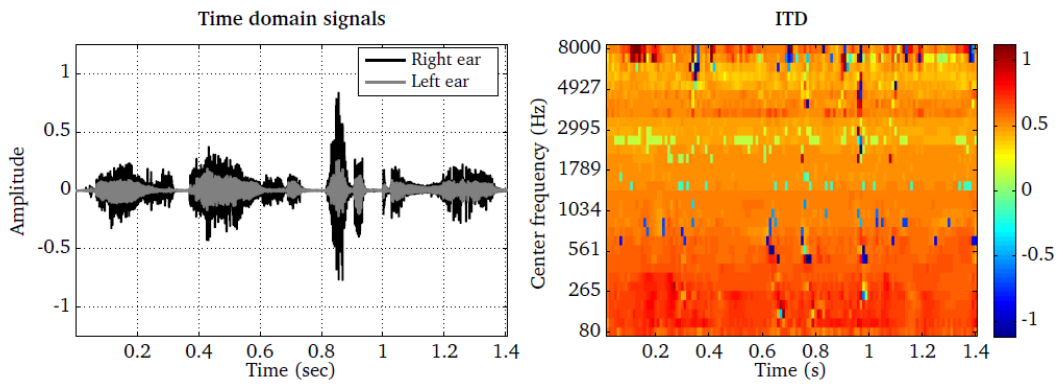


Fig. 29 Binaural speech signal (left panel) and the estimated ITD in ms shown as a function of time frames and frequency channels.

[May2011] 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.

[May2013b] 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.

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 Table 20.

Table 20 List of parameters related to `'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. 30. 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 TF units. It is apparent that the change considerably as a function of the centre 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.

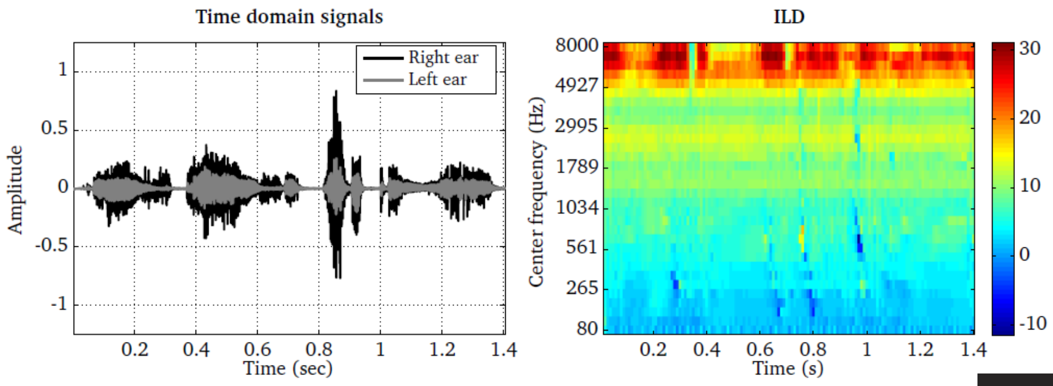
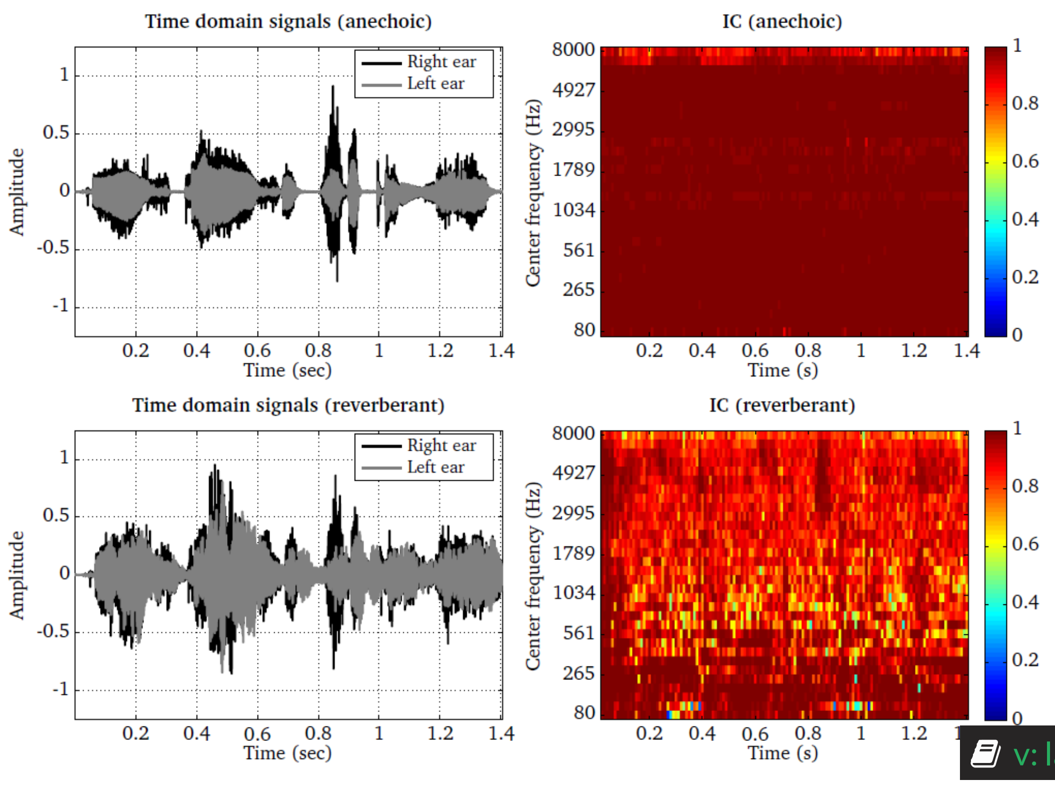


Fig. 30 Binaural speech signal (left panel) and the estimated ILD in dB shown as a function of time frames and frequency channels.

Interaural coherence (`icProc.m`)

The IC is estimated by determining the maximum value of the normalised CCF. It has been suggested that the IC can be used to select TF 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 [Faller2004]. The IC processor does not have any controllable parameters itself, but it depends on the settings of the CCF processor, which is described in Cross-correlation (crosscorrelationProc.m). 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. 31. 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 TF units, which indicates that the corresponding binaural cues are reliable. In contrast, the IC for the reverberant signal is substantially lower for many TF units, suggesting that the corresponding binaural cues might be unreliable due to the impact of the reverberation.



v: latest

Fig. 31 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).

[Faller2004] 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.

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 [Wallach1949]. The effect of the later-arriving sound is suppressed by the first part in the localization process. The precedence effect processor in Auditory front-end models this, with the strategy based on the work of [Braasch2013]. 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 [Stern1988], and through amplitude-weighted summation. Since these ITD/ILD calculation methods of the precedence effect processor are different from what are used for the Auditory front-end ITD and ILD processors, the Auditory front-end 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. Table 21 lists the parameters needed to operate the precedence effect processor.

Table 21 List of parameters related to the auditory representation `’prece`

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 comp

Fig. 32 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.

v: latest

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.

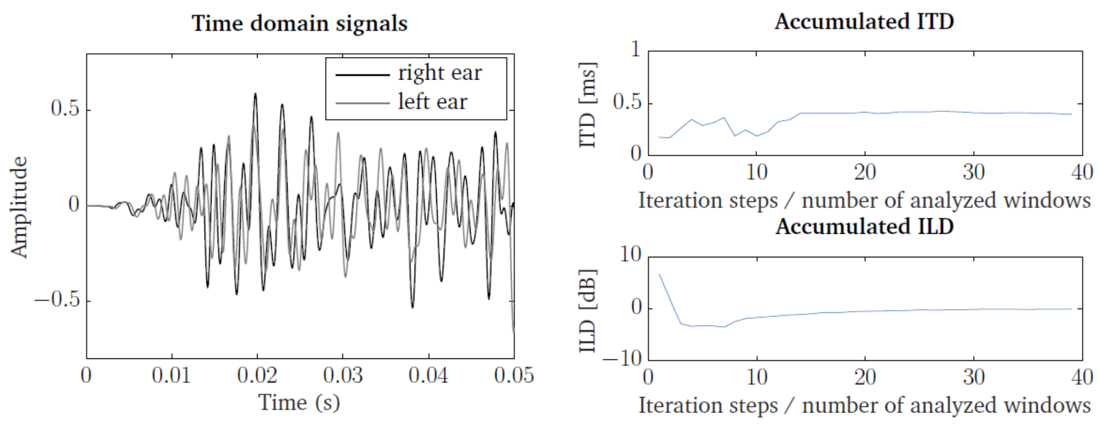


Fig. 32 Left panel: band-pass input noise signal, 400 ms long (only the first 50 ms is shown), 800 Hz wide, centered at 500 Hz, mixed with a reflection of a 2-ms delay, and made binaural with an of 0.4 ms ITD and ILD of 0 dB. Right panel: estimated ITD and ILD shown as a function of time frames.

[Braasch2013] 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.

[Stern1988] Stern, R. M., Zeiberg, A. S., and Trahiotis, C. (1988), “Lateralization
complex binaural stimuli: A weighted-image model,” The Journal of
Acoustical Society of America 84(1), pp. 156–165,
http://scitation.aip.org/content/asa/journal/jasa/84/1/10.1121/1.

[Wallach1949] 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, http://www.jstor.org/stable/1418275.

v: latest

Add your own processors

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

The Auditory front-end 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 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.

v: latest

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.

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.

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 w
virtually make the property invisible and inaccessible to all other objects.

v: latest

v: latest

Implement static methods

- `getDependency`
- `getParameterInfo`
- `getProcessorInfo`

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.

`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.

`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 something like this:

```
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.

`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 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
```

v: latest

Implement 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.

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 Auditory front-end 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
  % ...

end
```

Note

v: latest

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

v: latest

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.

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.

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.

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

v: latest

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.

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.

Implement the core processing method

At this stage, and if the previous tests were successfully passed, your processor should be correctly detected by the Auditory front-end 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.

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 signa
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 Auditory front-end is always indexing time.

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 Auditory front-end 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 "overl

v: latest

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`.

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`).

Override parent methods

The Auditory front-end 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.

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 Auditory front-end 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`

v: latest

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 Auditory front-end 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.

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 Auditory front-end:

- 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.

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 in `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).

Allow 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 as examples `gammatoneProc.isSuitableForRequest` and `drnlProc.isSuitableForRequest`.

Implement a new signal type

The Auditory front-end 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.

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.

B Documentation of Blackboard System

Introduction

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. In order to do so, the system must be able to recognise acoustic sources and optical objects, and achieve perceptual organisation of sound in the same manner as human listeners do. Bregman has referred to the latter phenomenon as ASA [Bregman1990], and to reproduce this ability in a machine system a number of factors must be considered:

- ASA involves diverse sources of knowledge, including both primitive (innate) grouping heuristics and schema-driven (learned) grouping principles.
- Solving the ASA problem requires the close interaction of top-down and bottom-up processes through feedback loops.
- Auditory processing is flexible, adaptive, opportunistic and context-dependent.

The characteristics of ASA are well-matched to those of *blackboard* problem-solving architectures. A blackboard system consists of a group of independent experts (knowledge sources) that communicate by reading and writing data on a globally-accessible data structure (blackboard). The blackboard is typically divided into layers, corresponding to data, hypotheses and partial solutions at different levels of abstraction. Given the contents of the blackboard, each knowledge source indicates the actions that it would like to perform; these actions are then coordinated by a scheduler, which determines the order in which actions will be carried out.

Blackboard systems were introduced by [Erman1980] as an architecture for speech understanding, in their Hearsay-II system. In the 1990s, a number of authors described blackboard-based systems for machine hearing [Cooke1993], [Lesser1995], [Ellis1996], [Godsmark1999]. All of these systems were in most respects conventional blackboard architectures, in which the knowledge sources employed rule-based heuristics. In contrast, the Two!Ears architecture aims to exploit recent developments in machine learning, by combining the flexibility of a blackboard architecture with powerful learning algorithms afforded by probabilistic graphical models.

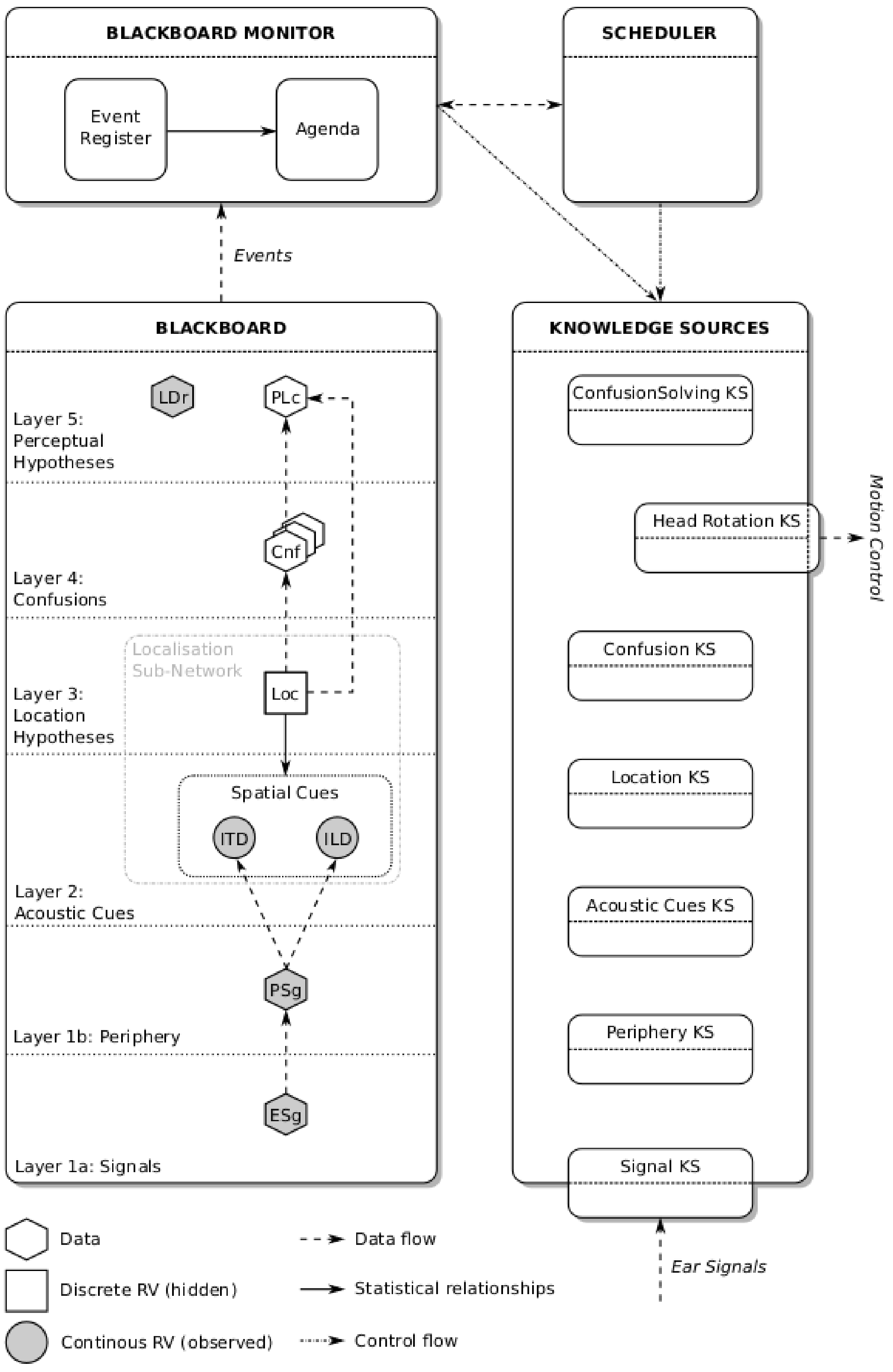


Fig. 30 Overview of the blackboard architecture of Two!Ears.

The general structure of the Blackboard system is shown in Fig. 30. It consists of different knowledge sources that can put data on and receive data from the blackboard. In addition, special knowledge sources can perceive data from outside (ear signals) or send data to the outside (turn the head). The management of the different processes going on in the blackboard is achieved by monitoring and scheduling which is performed by two independent modules.

Read on for further details on the blackboard architecture, details on the knowledge sources, or start with use the blackboard system.

[Bregman1990] Bregman, A. S. (1990), Auditory scene analysis: The perceptual organization of sound, The MIT Press, Cambridge, MA, USA.

[Erman1980] Erman, L. D., Hayes-Roth, F., Lesser, V. R., and Reddy, D. R. (1980), "The Hearsay-II speech understanding system: integrating knowledge to resolve uncertainty," Computing Surveys 12(2), pp. 213–253.

[Cooke1993] Cooke, M., Brown, G. J., Crawford, M., and Green, P. (1993), "Computational auditory scene analysis: listening to several things at once," Endeavour 17(4), pp. 186–190.

[Lesser1995] Lesser, V. R., Nawab, S. H., and Klassner, F. I. (1995), "IPUS: An architecture for the integrated processing and understanding of signals," Artificial Intelligence 77, pp. 129–171.

[Ellis1996] Ellis, D. P. W. (1996), "Prediction-driven computational auditory scene analysis," PhD thesis, Massachusetts Institute of Technology.

[Godsmark1999] Godsmark, D. and Brown, G. J. (1999), "A Blackboard Architecture for Computational Auditory Scene Analysis," Speech Commun. 27(3-4), pp. 351–366, URL http://dx.doi.org/10.1016/S0167-6393(98)00082-X.

Usage

The blackboard system is the heart of the Two!Ears Auditory Model as it provides an architecture that integrates experience formation and active behaviour from a set of different functional modules. These modules can work on different levels of abstraction, independently from each other or in collaboration, in a bottom-up or top-down manner.

Setting up the blackboard

In order to get the model running you first have to decide what is the task that the model should solve. You can get an idea of what is possible if you have a look at the currently available knowledge sources of the Blackboard system. A knowledge source is an independent module that runs inside the blackboard system and has knowledge about an specific topic, which could come from bottom-up or top-down processing.

If you have decided on what you want to do, you configure your blackboard in a XML-file. Let's assume that you want to classify a target speech source in your binaural input signals. A corresponding configuration file could then look like:

```xml
<blackboardsystem>
    <dataConnection Type="AuditoryFrontEndKS"/>

    <KS Name="baby" Type="IdentityKS">
        <Param Type="char">baby</Param>
        <Param Type="char">6687829ce1a73694a1ce41c7c01dec1b</Param>
    </KS>
    <KS Name="femaleSpeech" Type="IdentityKS">
        <Param Type="char">femaleSpeech</Param>
        <Param Type="char">6687829ce1a73694a1ce41c7c01dec1b</Param>
    </KS>
    <KS Name="idDec" Type="IdDecisionKS">
        <Param Type="int">0</Param>
        <Param Type="int">1</Param>
    </KS>

    <Connection Mode="replaceOld" Event="AgendaEmpty">
        <source>scheduler</source>
        <sink>dataConnect</sink>
    </Connection>
    <Connection Mode="replaceOld">
        <source>dataConnect</source>
        <sink>baby</sink>
        <sink>femaleSpeech</sink>
    </Connection>
    <Connection Mode="replaceParallel">
        <source>baby</source>
        <source>femaleSpeech</source>
        <sink>idDec</sink>
    </Connection>
</blackboardsystem>
```

Looking at the configuration file step by step we find the following settings:

`dataConnection`

This specifies where the data that your classifier uses comes from. For most of the available knowledge sources this will be the Auditory front-end which processes the input ear signals in a bottom-up way and provides the knowledge source with auditory features it can use to perform its action. If the data should come from the auditory front-end you have to specify `AuditoryFrontEndKS`, which is by itself a knowledge source.

`KS`

This specifies the knowledge sources that should be part of the blackboard system. In this case we use two identity knowledge sources that have knowledge about features corresponding to a particular sound source. This is given as a parameter `Param` to the identity knowledge source. Each identity knowledge source will provide a hypothesis to the blackboard stating the probability that the corresponding identity is matched by the input signal.

v: 1.3

The second parameter in the `IdentityKS` description (a string of hexadecimal digits) is the version number of a Matlab MAT file that contains a trained acoustic model. For the first knowledge source in the example above, the file name of the corresponding acoustic model file will be `baby.6687829ce1a73694a1ce41c7c01dec1b.model.mat`. For more details, see the description of the identity knowledge source, which explains how to train your own source models.

The second kind of knowledge source we use is the identity decision knowledge source. It will judge the different identity hypothesis it will get from the identity knowledge sources and performs the final decision which identity is matched by the input signals.

`Connection`

The blackboard is a modular system. The `Connection` settings tell the blackboard system which connections should be made between its various modules, so that they can be notified about relevant events. The configuration file shown above describes three connections, each of which describe an event binding between one or more **sources** and one or more **sinks**; let's look at each of them in turn.

Lines 17-20: This describes an event binding between the scheduler and the auditory front end, for an event called `AgendaEmpty`. Hence, the auditory front end will be notified when this event is dispatched by the scheduler, causing it to retrieve a new block of data.

Lines 21-25: Makes an event binding between the auditory front end and the two `IdentityKS` objects defined earlier in the file. Thus, the `IdentityKS` objects will be notified when new audio input is available to classify.

Lines 26-30: Make event bindings between the two `IdentityKS` objects and the `IdDecisionKS`; hence the `IdDecisionKS` is notified when a source has been classified, so that it can make a final decision on which source types are present in the scene.

Each of these connections can have a specific `Mode`, which determines how the blackboard should add triggered knowledge sources into the agenda. For example, the `replaceOld` mode indicates that old instances of the corresponding sink in the agenda should be replaced when a new one occurs. For full details of the different modes, see the section on dynamic blackboard interactions.

Running the blackboard

The following example code fragment shows how to run the blackboard system. It is assumed that the XML files `Scene.xml` and `Blackboard.xml` have previously been created and are on your Matlab path.

v: 1.3

```
1    bbs = BlackboardSystem(0);
2    sim = simulator.SimulatorConvexRoom('Scene.xml');
3    bbs.setRobotConnect(sim);
4    bbs.buildFromXml('Blackboard.xml');
5    bbs.run();
```

In the first line, a `BlackboardSystem` object is created. The parameter to the constructor (in this case, zero) indicates the verbosity level. In line 2 a room simulation is created from the scene description file `Scene.xml`. Line 3 then connects the blackboard to this simulator front end (at this stage you could also connect it to a real robot platform). Finally, in line 4 the blackboard is built from the configuration file `Blackboard.xml` and line 5 runs the blackboard; it will now proceed in a run loop until there is no more data to process.

Further examples

In order to see the whole model in action, including setting up of the Binaural simulator you should have a look at the example chapter.

Blackboard architecture

The Blackboard system is targeted as the front-end for a great variety of applications, providing an architecture that integrates experience formation and active behaviour from a set of different functional modules. These modules can work on different levels of abstraction, independently from each other or in collaboration, in a bottom-up or top-down manner. A key feature of this system should be its ability to evolve, so that easy modification, exchange and/or extension of modules can be achieved within a scalable architecture.

This document will provide you an overview about the blackboard architecture that is used together with the knowledge sources in order to provide such a system.

Architectural considerations

In order to implement this integrative and system-wide view, some core attributes of the system have been established as follows.

Building a flexible system

The system we develop is a *platform*, i.e. it provides functionality to execute other functionality. While the target functionality is clear – auditory and multi-modal experience formation, scene understanding and exploration – it involves many different problems, each with many possible solutions. We therefore design the system with extension in mind, trying not to constrain possible functionality of modules.

In particular, the blackboard system allows the plugging-in of different knowledge sources. Knowledge sources are modules that define their own functionality, to be executed in the organised frame of our system. They define by themselves which data they need for execution and which data they produce – the blackboard system provides the tools for requesting and storing this data, but does not care about the actual contents (while the knowledge sources do not need to care about where and how data is stored). It is also important that the blackboard system has no static knowledge of what types of knowledge sources are available. So long as knowledge sources follow a certain implementation scheme, independent of their actual functionality they can register *dynamically* (i.e. at runtime) as a module in the blackboard system. Thus, a library of knowledge sources can be built during this project that can be extended arbitrarily, without need to modify the blackboard system. Implementors of new modules need only be concerned with implementing their functionality.

The Two!Ears architecture has been designed and implemented using an object-oriented approach. Accordingly, the implementation scheme knowledge sources must adhere to is provided in the form of an abstract class). Additionally, to enable creation of new knowledge sources that depend on auditory signals without needing to hard-code a signal request into the system, an auditory front-end dependent knowledge source superclass has been developed.

Building a dynamic system

Key to providing the described flexibility is to neither hard-code lists of usable knowledge sources nor the interactions between them. Hard-coded (or static) lists and dependencies would be overly restrictive – the system must be open to dynamic change.

At the same time, flexibility for extension is not the only cause for needing a dynamic system. The system is intended to be an *active* system that does not only work in a signal processing bottom-up manner, but also in a cognitive top-down manner. Modules must therefore be allowed to change the system setup at runtime. This means that it is essential for our system to be equipped with functionality for *dynamic* module instantiation, registration and removal. This also implies the need for on-the-fly rewiring of the communication links between modules.

Dynamic system construction

To ensure an easy-to-use system, we implemented as the main class a wrapper that integrates the different main components, and hides their connections where possible. This main class is called **BlackboardSystem**, since the blackboard is the central component of our platform. This is an excerpt of its definition:

```
class BlackboardSystem
    properties
        blackboard;
        blackboardMonitor;
        scheduler;
        robotConnect;
        dataConnect;
    methods
        BlackboardSystem()
        setRobotConnect( robotConnect )
        setDataConnect( connectorClassName )
        buildFromXml( xmlName )
        addKS( ks )
        createKS( ksClassName, ksConstructArgs )
        numKSs()
        run()
```

The *engine* of our system is distributed across the `BlackboardSystem`, `Blackboard`, `BlackboardMonitor` and `Scheduler` classes, with the `BlackboardSystem` class holding instances of the latter three. These four classes each have genuine responsibilities: the `BlackboardSystem` integrates the framework parts, responsible for constructing and setting up the system. The blackboard is the central storage of *functional* data and knowledge sources. It holds a data map that saves arbitrary knowledge source data along time, together with methods to add and recall data from within knowledge source code. Additionally, the knowledge sources themselves are put into blackboard storage by the `BlackboardSystem`.

The `BlackboardMonitor` is responsible for creating bindings on demand between knowledge sources, by instantiating event listeners. It keeps track of these bindings and maintains the agenda of knowledge sources. The `Scheduler` is the executive component of the system. While the `BlackboardMonitor` keeps control of the knowledge sources in the agenda, the

v: latest

`Scheduler` decides about the order of those knowledge sources to be executed. It does that based on the attentional priorities of the knowledge sources. Fig. 34 shows the system class diagram.

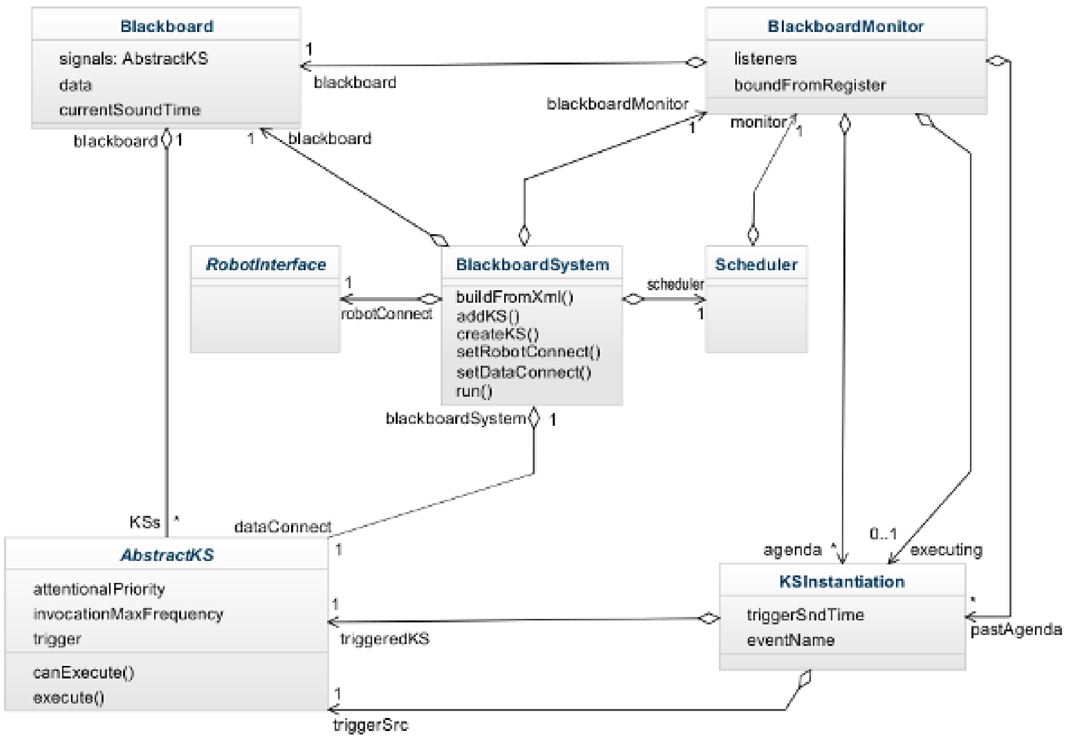


Fig. 34 Class diagram of the whole blackboard system. The `BlackboardSystem` class is the integrative system component holding the other modules and giving access to system functionality.

An example of an XML-configured blackboard system is shown below. Two identity knowledge sources are connected to the Auditory front-end, and triggering an identity decision knowledge source.

```
<blackboardsystem>
    <dataConnection Type="AuditoryFrontEndKS"/>

    <KS Name="baby" Type="IdentityKS">
        <Param Type="char">baby</Param>
        <Param Type="char">6687829ce1a73694a1ce41c7c01dec1b</Param>
    </KS>
    <KS Name="femaleSpeech" Type="IdentityKS">
        <Param Type="char">femaleSpeech</Param>
        <Param Type="char">6687829ce1a73694a1ce41c7c01dec1b</Param>
    </KS>
    <KS Name="idDec" Type="IdDecisionKS">
        <Param Type="int">0</Param>
        <Param Type="int">1</Param>
    </KS>

    <Connection Mode="replaceOld" Event="AgendaEmpty">
        <source>scheduler</source>
        <sink>dataConnect</sink>
    </Connection>
    <Connection Mode="replaceOld">
        <source>dataConnect</source>
        <sink>baby</sink>
        <sink>femaleSpeech</sink>
    </Connection>
    <Connection Mode="replaceParallel">
        <source>baby</source>
        <source>femaleSpeech</source>
        <sink>idDec</sink>
    </Connection>
</blackboardsystem>
```

Several core functionalities are provided through the `BlackboardSystem` class:

- Connecting a Robotic platform or the Binaural simulator to the Blackboard system. The connected robot or simulator must implement the robot interface for delivering audio *ear* signals and commanding movement and head rotation. The blackboard system and all its components including the knowledge sources get access to the audio stream and robot actions through this connection (`setRobotConnect` method).
- Setting the type of the module that integrates with the Auditory front-end, instantiating it and connecting it to the Blackboard system. This module is a knowledge source itself, responsible for processing the ear signals into cues (such as interaural time differences) needed by other knowledge sources. The Auditory front-end itself is connected to the Robotic platform or the Binaural simulator in order to obtain the ear signals. (`setDataConnect` method).
- Instantiating and adding knowledge sources. These knowledge sources must inherit from `AbstractKS` (see abstract knowledge source) or `AuditoryFrontEndDepKS` (see Section Auditory signal dependent knowledge source superclass: AuditoryFrontEndDepKS) to be able to be interfaced and run by the system.

Knowledge sources that inherit from `AuditoryFrontEndDepKS` automatically get connected with the Auditory front-end by the system in order to place their signal/cue requests.

Adding and instantiating knowledge sources can take place both before or *while* running the system; it can be done from outside the system *or from inside knowledge sources*. This enables the development of top-down controlling knowledge sources from higher cognitive experts running in the system (`addKS` or `createKS` method).

- The start-up configuration of the system can completely be defined by an XML file; the system is then constructed before running by loading this file. Of course this configuration can be changed dynamically while executing the system. The XML description needs at least a `dataConnection` node specifying the type of the Auditory front-end module; then, it can also contain `KS` nodes with parameters to construct knowledge sources, and `Connection` nodes that specify event bindings between knowledge sources. See the code listing from above for an example (`buildFromXml` method).
- Starting the execution of the system. This triggers the blackboard system to request data from the robot/binaural simulator connection, and subsequent action by the Auditory front-end and all knowledge sources that are connected. The system will not stop execution before the Robotic platform or the Binaural simulator sends an ending signal (`run` method).

v: latest

Dynamic blackboard memory

The `Blackboard` class holds the central data repository of the platform. It stores the knowledge sources and any shared data, in particular the output of the knowledge sources (e.g. estimates of the location of a sound source). It is accessible to all knowledge sources; and it not only stores current data, but keeps track of the history of this data in order to enable knowledge sources to work on time series data.

Importantly, the `Blackboard` is flexible about data categories, which do not have to be hard-coded into the system. Knowledge sources decide on their own and at runtime what to add and what to request. Thus, the system does not need to be changed in order to implement new knowledge sources that work with new data categories. Of course, knowledge sources can only read data categories that are actually stored in the blackboard by other knowledge sources (or themselves).

The following listing shows an excerpt of the `Blackboard` interface:

```
class Blackboard
        KSs;
        signals;
        currentSoundTime;
    methods
        Blackboard()
        addData( dataLabel, data, append, time )
        getData( dataLabel, reqSndTime )
        getLastData( dataLabel, time )
        getNextData( dataLabel, time )
        getDataBlock( dataLabel, blockSize_s )
```

Prominently featured are methods to add and access data:

`addData`
lets knowledge sources add data to the blackboard storage. The data category has to be named in `dataLabel`, `data` hands over the actual data to store. `append` is an optional flag indicating whether to overwrite or append data at the same time step (there might, for example, be several source identity hypotheses per time step, but only one source number hypothesis might be allowed). `time` sp

v: latest

the time point under which this data shall be stored. It is optional and, if not set, defaults to the current time.

`getData`
: lets knowledge sources read data from the blackboard storage. `dataLabel` indicates the data category requested, `reqSndTime` the time point of interest. `getLastData`, `getNextData` and `getDataBlock` are special cases of `getData` for retrieving the last data, the next data after a particular point in time, or a whole data block of length `blockSize_s`.

The following is an example from the implementation of the `IdDecisionKS` class:

```
idHyps = obj.blackboard.getData( ...
             'identityHypotheses', obj.trigger.tmIdx ).data;
%...
%find the most likely identity hypothesis -> maxProbHyp
%...
obj.blackboard.addData( ...
    'identityDecision', maxProbHyp, false, obj.trigger.tmIdx );
```

Let us assume that we have an instantiation called `bbs` of the Blackboard system. Now we would like to see the currently available data in its memory. For that you can run:

```
>> bbs.blackboard.getDataLabels()
```

or specify explicitly a time for which you would like to see the available data:

```
>> bbs.blackboard.getDataLabels(bbs.blackboard.currentSoundTimeIdx-1)
```

Additionally, the blackboard is used as a storage for pointers to signals from the Auditory front-end requested by knowledge sources inheriting from `AuditoryFrontEndDepKS`. The actual memory in which these signals are stored for recall is implemented in the Auditory front-end through circular buffers.

Dynamic blackboard interactions

Knowledge sources can communicate information through the flexible blackboard storage. However, adding data to the blackboard does not trigger other knowledge sources to be executed. Such interaction – triggering knowledge source execution – is done through an event system. Specifically, knowledge sources do not actually trigger execution of other knowledge sources (since they are decoupled and have no "knowledge" of each other), but knowledge sources *make a request to be triggered* upon the firing of particular *events*.

Each knowledge source has a standard event it can trigger, `KsFiredEvent`, inherited from `AbstractKS`. Beyond that, every knowledge source class is free to define as many additional events as reasonable for its task. Knowledge sources cause the events themselves through a call to `notify` as in the following example, in which the knowledge source induces an event and attaches a `BlackboardEventData` object holding the time that it was triggered:

```
notify( 'KsFiredEvent', BlackboardEventData(obj.trigger.tmIdx) );
```

The blackboard system a priori is totally ignorant of which events exist (clear responsibilities principle, open to extension). It also does not monitor any events by default, until knowledge sources request to be triggered by an event. This request is done through the method `bind` provided by the `BlackboardMonitor` class, whose interface is (partially) listed in the following excerpt:

```
class BlackboardMonitor
    properties
        pastAgenda;
        executing;
        agenda;
    methods
        BlackboardMonitor()
        bind( sources, sinks, addMode, eventName )
```

The `bind` method connects the `sinks` knowledge sources to event `eventName` (optional, defaults to `KsFiredEvent`) caused by the `sources` knowledge sources. `addMode` specifies how the `BlackboardMonitor` shall handle adding the triggered knowledge sources i

v: latest

agenda. It understands the following modes, illustrated in Fig. 35:

`add`
: Add the triggered knowledge source to the end of the agenda, regardless of whether or not there is already a (not yet executed) knowledge source instantiation of this sink in the agenda from a former triggering.

`replaceOld`
: Replace old knowledge source instantiations of this sink in the agenda with the new one. Only instantiations of the sink triggered by the same source and same event are replaced. This is an important mode for knowledge sources where processing current data is more important than processing all data.

`replaceParallel`
: Replace knowledge source instantiations of this sink from the same time point of parallel sources in the agenda with the new one. Only instantiations of the sink triggered at the same time and by the same event are replaced. This mode avoids sinks being unnecessarily executed several times with the same information.

`replaceParallelOld`
: Replace old or current knowledge source instantiations of this sink triggered by parallel sources in the agenda with the new one. Only instantiations of the sink triggered by the same event are replaced. This mode is a combination of the `replaceOld` and `replaceParallel` modes.

It should be noted that the `addMode` only affects triggered knowledge source instantiations in the *agenda*, i.e. those that are not executed yet. As soon as a knowledge source is executed, it is removed from the agenda (first in `executing`, afterwards in `pastAgenda`).

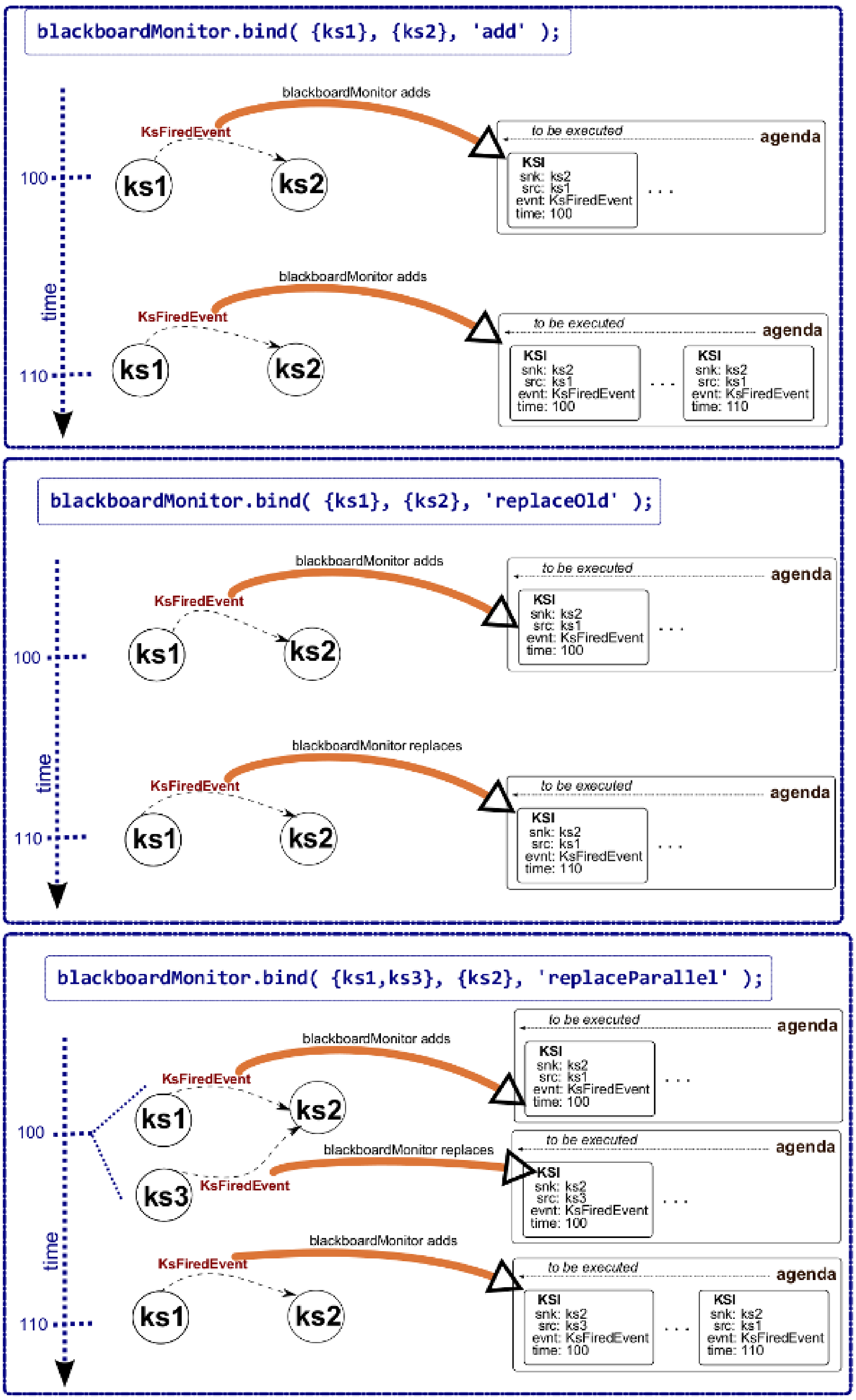


Fig. 35 The different possibilities of event binding between knowledge sources with the blackboard system.

Scheduler

The scheduler is the component of the blackboard system that actually executes the knowledge sources – but first, it *schedules* them, that is, it decides the order in which knowledge sources waiting in the agenda get executed. This order is rescheduled after every execution of a knowledge source, since the conditions determining the order may have changed, or new knowledge sources may be present in the agenda that are more urgent.

The following factors influence the order of execution of knowledge sources:

- Knowledge sources have a property called *attentional priority*. Knowledge sources with higher priority get executed before knowledge sources with lower priority. This priority can be set by the knowledge source itself, by other knowledge sources or from outside the system. The `BlackboardMonitor` provides a method for setting focus on a knowledge source (increasing its priority), along with the option to propagate this higher priority down along the dependency chain of this knowledge source. The dependency chain is determined by the event bindings.
- Knowledge sources must implement a method `canExecute`, that returns whether or not the knowledge source can execute at this moment, and which is called by the scheduler if the knowledge source is first on the scheduling list. If it cannot execute, the knowledge source can decide whether to remain in the agenda or be removed from it.
- Knowledge sources define a maximum invocation frequency, that cannot be exceeded. It is a *maximum* frequency, because knowledge sources get not necessarily executed periodically, since they are triggered by events, but not by timers. The scheduler checks whether the last execution time was long enough ago before considering the knowledge source for execution. Until then, it remains in the agenda.

This listing shows the relevant parts of the interface with respect to influencing the scheduling:

```
class BlackboardMonitor
    methods
        focusOn( ks, propagateDown )
        resetFocus()

class AbstractKS
    properties
        invocationMaxFrequency_Hz;
    methods (Abstract)
        canExecute()
        execute()
    methods
        focus()
        unfocus()
```

C Documentation of Application Examples

Examples

In this part you find a collection of several examples using the complete Two!Ears Auditory Model for common tasks. At the moment the following examples are available:

- Localisation with and without head rotations
- Localisation - looking at the results in detail
- DNN-based localisation under reverberant conditions
- GMM-based localisation under reverberant conditions
- Train sound type identification models
- Identification of sound types
- Segmentation with and without priming
- (Re)train the segmentation stage
- Stream binaural signals from BASS to Matlab
- Prediction of coloration in spatial audio systems
- Prediction of localisation in spatial audio systems

This section will expand with every new major feature that will be added to the model.

Localisation with and without head rotations

The Two!Ears Auditory Model comes with several knowledge sources that work together to estimate the perceived azimuth of a sound source, see Localisation knowledge sources for a summary. The main work is done by the GmmLocationsKS knowledge source that uses ITD and ILD cues provided by the Auditory front-end and compares them with learned cues to azimuth maps. As an output it provides a probability distribution of possible directions for the source. This will be passed on to the ConfusionKS knowledge source which looks at the probabilities and decides if a clear direction can be extracted from this. If not, ConfusionSolvingKS is called which then triggers RotationKS to rotate the head of the listener (could be in the simulation or of a robot) and start the localisation process again.

In this example we will see how to set up the model to perform a localisation task and how to switch on or off the possibility of the model to rotate its head. This example can be found in the `examples/localisation_w_and_wo_head_movements` folder which consists of the following files:

```
BlackboardNoHeadRotation.xml
Blackboard.xml
localise.m
SceneDescription.xml
```

The first file we look at is `SceneDescription.xml`, it defines the actual acoustic scene in which our virtual head and the sound source will be placed in order to simulate binaural signals. It looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<scene
  BlockSize="4096"
  SampleRate="44100"
  MaximumDelay="0.0"
  NumberOfThreads="1"
  LengthOfSimulation = "5"
  HRIRs="impulse_responses/scut_kemar_anechoic/SCUT_KEMAR_anechoic_1m.sofa">
  <source Radius="1.0"
          Mute="false"
          Type="point"
          Name="SoundSource">
    <buffer ChannelMapping="1"
        Type="noise"/>
  </source>
  <sink Name="Head"
        Position="0 0 0"
        UnitX="1 0 0"
        UnitZ="0 0 1"/>
</scene>
```

Here, we define basic things like the sampling rate, the length of the stimulus, the used HRTF, the source material, the listener position, and the distance between listener and source. For more documentation on specifying an acoustic scene, see Configuration using XML Scene Description.

Note

We don't specify the exact source azimuth here, as we will choose different azimuth values later on and set them on the fly from within Matlab.

The next thing we have to do is to specify of what components or model should consists and what it should actually do. This is done by selecting appropriate modules for the Blackboard system stage of the model. This can be configured again in a xml file. First we look at the configuration for localisation including head movements (`Blackboard.xml`):

```xml
<?xml version="1.0" encoding="utf-8"?>
<blackboardsystem>

    <dataConnection Type="AuditoryFrontEndKS"/>

    <KS Name="loc" Type="GmmLocationKS"/>
    <KS Name="conf" Type="ConfusionKS"/>
    <KS Name="confSolv" Type="ConfusionSolvingKS"/>
    <KS Name="rot" Type="RotationKS">
        <Param Type="ref">robotConnect</Param>
    </KS>

    <Connection Mode="replaceOld" Event="AgendaEmpty">
        <source>scheduler</source>
        <sink>dataConnect</sink>
    </Connection>
    <Connection Mode="replaceOld">
        <source>dataConnect</source>
        <sink>loc</sink>
    </Connection>
    <Connection Mode="add">
        <source>loc</source>
        <sink>conf</sink>
    </Connection>
    <Connection Mode="replaceOld" Event="ConfusedLocations">
        <source>conf</source>
        <sink>rot</sink>
    </Connection>
    <Connection Mode="add" Event="ConfusedLocations">
        <source>conf</source>
        <sink>confSolv</sink>
    </Connection>

</blackboardsystem>
```

Here, we use different knowledge sources that work together in order to solve the localisation task. We have AuditoryFrontEndKS for extract auditory cues from the ear signals, GmmLocationsKS, ConfusionKS, ConfusionSolvingKS, and RotationKS for the actual localisation task. The `Param` tags are parameters we can pass to the knowledge sources. After setting up which knowledge sources we will use, we connect them with the `Connection` tags. For more information on configuring the blackboard see Configuration.

In a second configuration file we setting up the same blackboard, but now disabling its ability to turn the head (`BlackboardNoHeadRotation.xml`):

```
<?xml version="1.0" encoding="utf-8"?>
<blackboardsystem>

    <dataConnection Type="AuditoryFrontEndKS"/>

    <KS Name="loc" Type="GmmLocationKS"/>
    <KS Name="conf" Type="ConfusionKS">
        <!-- Disable confusion solving (== no head rotation) -->
        <Param Type="int">0</Param>
    </KS>

    <Connection Mode="replaceOld" Event="AgendaEmpty">
        <source>scheduler</source>
        <sink>dataConnect</sink>
    </Connection>
    <Connection Mode="replaceOld">
        <source>dataConnect</source>
        <sink>loc</sink>
    </Connection>
    <Connection Mode="add">
        <source>loc</source>
        <sink>conf</sink>
    </Connection>
</blackboardsystem>
```

Now, everything is prepared and we can start Matlab in order to perform the localisation. You can just start it and run the following command to see it in action, afterwards we will have a look at what happened:

```
>> localise

-------------------------------------------------------------------
Source direction        Model w head rot.        Model wo head rot.
-------------------------------------------------------------------
   0                          0                        0
  33                         35                       35
  76                         70                       70
-121                       -120                      -55
-------------------------------------------------------------------
```

As you can see the model with head rotation returned better results as the model without head rotation enabled. The reason why we have problems without head rotations is that we have trained the model with another HRTF data set (QU KEMAR) as we used for the creation of the acoustic scene (SCUT KEMAR).

Now, we have a look into the details of the `localise()` function. We will only talk about the parts that are responsible for the task, not for printing out the results onto the screen. First we define the source angles we are going to synthesise and start the Binaural simulator:

v: latest

```
% Different angles the sound source is placed at
sourceAngles = [0 33 76 239];

% === Initialise binaural simulator
sim = simulator.SimulatorConvexRoom('SceneDescription.xml');
sim.Verbose = false;
sim.Init = true;
```

After that we have a loop over the different source angles in which we are setting the source position in the Binaural simulator and run two different blackboards after each other, one with, the other one without head rotations:

```
for direction = sourceAngles

    sim.Sources{1}.set('Azimuth', direction);
    sim.rotateHead(0, 'absolute');
    sim.ReInit = true;

    % GmmLocationKS with head rotation for confusion solving
    bbs = BlackboardSystem(0);
    bbs.setRobotConnect(sim);
    bbs.buildFromXml('Blackboard.xml');
    bbs.run();

    % Reset binaural simulation
    sim.rotateHead(0, 'absolute');
    sim.ReInit = true;

    % GmmLocationKS without head rotation and confusion solving
    bbs = BlackboardSystem(0);
    bbs.setRobotConnect(sim);
    bbs.buildFromXml('BlackboardNoHeadRotation.xml');
    bbs.run();

end
```

Localisation - looking at the results in detail

As seen in the previous example localisation is performed inside the blackboard by different localisation knowledge sources. In this example we will perform a single localisation of a anechoic white noise signal coming from 0° using GmmLocationKS. The example can be found in the `examples/localisation_look_at_details` folder which consists of the following files:

```
Blackboard.xml
localise.m
SceneDescription.xml
```

For details on `Blackboard.xml` and `SceneDescription.xml` have a look at our previous example. Here, we will focus on the details after we performed the localisation. So, first run:

```
>> bbs = localise;
```

in Matlab. This runs the blackboard and does the localisation, but does not print any results, it only returns the blackboard as `bbs`. The blackboard itself stores lots of data in itself, see Dynamic blackboard memory for details. To see what is currently available in the memory run:

```
>> bbs.blackboard.getDataLabels()

ans =

    'confusionHypotheses'
    'headOrientation'
    'sourcesAzimuthsDistributionHypotheses'
    'perceivedAzimuths'
```

To analyse the localisation performance of the model we ask the blackboard to return the localisation data:

```
>> perceivedAzimuths = bbs.blackboard.getData('perceivedAzimuths')

perceivedAzimuths =

1x9 struct array with fields:

    sndTmIdx
    data
```

It returns a relatively complicated struct that comes with time stamps `sndTmIdx` and the corresponding `data` which again is a struct containing different things, here is the output for one time stamp:

```
>> perceivedAzimuths(9).data

ans =

  PerceivedAzimuth with properties:

            azimuth: 20
    headOrientation: 340
    relativeAzimuth: 0
              score: 0.7028
```

But don't worry there is an easy way to get an overview of the results. First, we are only interested in the summary of the localisation result:

```
>> sourceAzimuth = 0; % the actual source position
>> [loc, locError] = evaluateLocalisationResults(perceivedAzimuths, sourceAzimuth)

loc =

     0


locError =

     0
```

As you can see the source was localised at 0° meaning that the localisation error is also 0°. After that we would like to have a more detailed view on what happened during the localisation:

v: latest

```
>> displayLocalisationResults(perceivedAzimuths, sourceAzimuth)

------------------------------------------------------------------------------------
Reference target angle:   0 degrees
------------------------------------------------------------------------------------
Localised source angle:
BlockTime   PerceivedAzimuth    (head orient., relative azimuth)    Probability
------------------------------------------------------------------------------------
  0.56        0 degrees     (  0 degrees,      0 degrees)   1.00
  1.02        0 degrees     (  0 degrees,      0 degrees)   0.65
  1.58        0 degrees     ( 20 degrees,    340 degrees)   0.56
  2.04        0 degrees     (  0 degrees,      0 degrees)   0.61
  2.51        0 degrees     ( 20 degrees,    340 degrees)   0.50
  3.07        0 degrees     (  0 degrees,      0 degrees)   0.69
  3.53        0 degrees     ( 20 degrees,    340 degrees)   0.73
  4.09        0 degrees     (  0 degrees,      0 degrees)   0.62
  4.55        0 degrees     (340 degrees,     20 degrees)   0.70
------------------------------------------------------------------------------------
Mean localisation error: 0
------------------------------------------------------------------------------------
```

Here, we see that the head was turned twice during the localisation and that the perceived location was always at 0°, but the model has not have always the same confidence that the source was really located there, which you can see by the `Probability` values. In the cases when they were to low they triggered a head movement in order to see if the values would be higher for another head position.

So far, we looked at all the details going on in the blackboard. In order to localise the blackboard uses different cues - like ITDs and ILDs - that are provided by the Auditory front-end. It might be of interest to have a detailed look on them. In order to see which are available, run:

```
>>  bbs.listAfeData

Available AFE data:

  'filterbank'
  'time'
  'input'
  'itd'
  'innerhaircell'
  'ild'
  'crosscorrelation'
  'head_rotation'
```

All those cues can be plotted with `bbs.plotAfeData(cueName)`, for example:

```
>> bbs.plotAfeData('time');
```

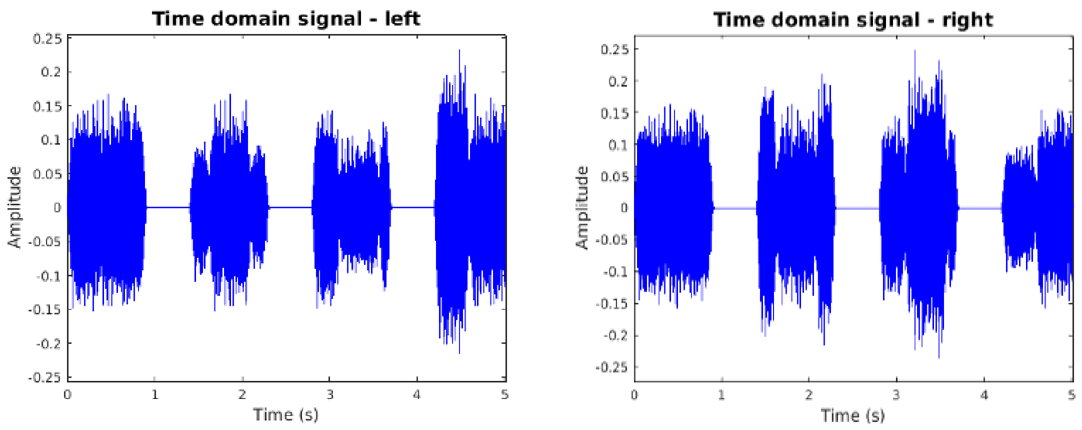


Fig. 59 Left and right ear signals.

```
>> bbs.plotAfeData('ild');
```

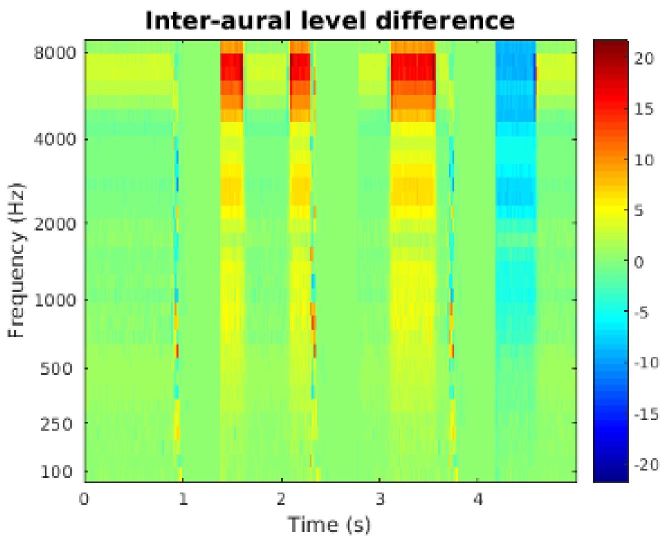


Fig. 60 ILDs between the two ear signals over time.

The next one is not really a cue provided by the Auditory front-end, but is it good to know in which position the head was pointing at what time:

```
>> bbs.plotAfeData('head_rotation');
```

v: latest

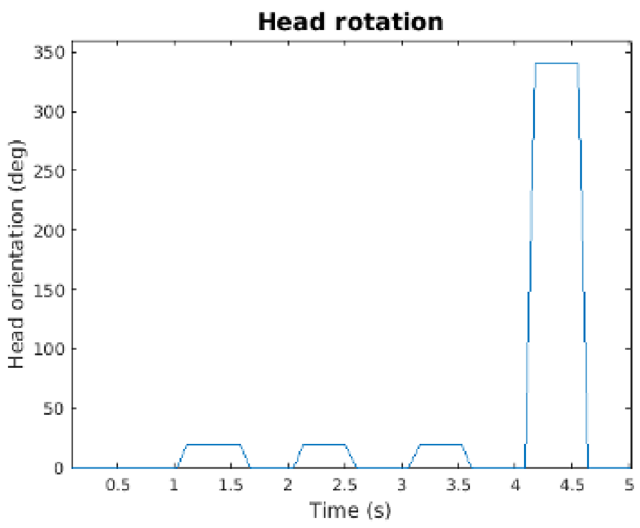


Fig. 61 Head rotations of the model during localisation.

DNN-based localisation under reverberant conditions

The Two!Ears Auditory Model comes with several knowledge sources that work together to estimate the perceived azimuth of a sound source, see Localisation knowledge sources for a summary. One stage of this process is the mapping of the extracted features like ITDs and ILDs to the perceived azimuth angle. This mapping is highly influenced by the environment. For example, if you are in a room the ITD values will look quite different than in the case of an anechoic chamber. That is the reason why we have different knowledge sources that do this mapping: DnnLocationKS, GmmLocationsKS, and ItdLocationKS. ItdLocationKS utilises a simple lookup table for the mapping works well in the case of Prediction of localisation in spatial audio systems. GmmLocationsKS is at the moment trained only for anechoic condition. In this example we have a look at DnnLocationKS which was trained with a multi-conditional training approach to work under reverberant conditions [MaEtAl2015dnn]. Beside this, DnnLocationKS works in the same way as GmmLocationsKS and connects with ConfusionKS, ConfusionSolvingKS, and RotationKS to solve front-back confusions.

In this example we will have a look at localisation in a larger room, namely the BRIR data set measured in TU Berlin, room Auditorium 3, which provides six different loudspeaker positions as possible sound sources. All files can be found in the `examples/localisation_DNNs` folder which consists of the following files:

```
BlackboardDnnNoHeadRotation.xml
BlackboardDnn.xml
estimateAzimuth.m
localise.m
resetBinauralSimulator.m
setupBinauralSimulator.m
```

The setup is very similar to Localisation with and without head rotations with a few exceptions. First, the setup of the Binaural simulator is different as we use BRIRs instead of HRTFs, and have one impulse response set for every sound source. The initial configuration of the Binaural simulator is provided by the `setupBinauralSimulator` function:

```
sim = simulator.SimulatorConvexRoom();
set(sim, ...
    'BlockSize',            4096, ...
    'SampleRate',           44100, ...
    'NumberOfThreads',      1, ...
    'LengthOfSimulation',   1, ...
    'Renderer',             @ssr_brs, ...
    'Verbose',              false, ...
    'Sources',              {simulator.source.Point()}, ...
    'Sinks',                simulator.AudioSink(2) ...
    );
set(sim.Sinks, ...
    'Name',                 'Head', ...
    'Position',             [ 0.00  0.00  0.00]' ...
    );
set(sim.Sources{1}, ...
    'AudioBuffer',          simulator.buffer.Ring(1) ...
    );
set(sim.Sources{1}.AudioBuffer, ...
    'File', 'sound_databases/grid_subset/s1/bbaf2n.wav' ...
    );
```

Here, we configure it to use the `@ssr_brs` renderer which is needed for BRIRs, define the speech signal to use, but don't provide a BRIR yet as this will be done on the fly later on.

We have four different configuration files for setting up the Blackboard system. One important step for the DnnLocationKS is to define a resampling as it is trained for 16000 Hz at the moment. As an example, we list the file `BlackboardDnn.xml``:

v: latest

```xml
<?xml version="1.0" encoding="utf-8"?>
<blackboardsystem>

    <dataConnection Type="AuditoryFrontEndKS">
       <Param Type="double">16000</Param>
    </dataConnection>

    <KS Name="loc" Type="DnnLocationKS">
        <Param Type="int">16</Param>
    </KS>
    <KS Name="conf" Type="ConfusionKS"/>
    <KS Name="confSolv" Type="ConfusionSolvingKS"/>
    <KS Name="rot" Type="RotationKS">
        <Param Type="ref">robotConnect</Param>
    </KS>

    <Connection Mode="replaceOld" Event="AgendaEmpty">
        <source>scheduler</source>
        <sink>dataConnect</sink>
    </Connection>
    <Connection Mode="replaceOld">
        <source>dataConnect</source>
        <sink>loc</sink>
    </Connection>
    <Connection Mode="add">
        <source>loc</source>
        <sink>conf</sink>
    </Connection>
    <Connection Mode="replaceOld" Event="ConfusedLocations">
        <source>conf</source>
        <sink>rot</sink>
    </Connection>
    <Connection Mode="add" Event="ConfusedLocations">
        <source>conf</source>
        <sink>confSolv</sink>
    </Connection>

</blackboardsystem>
```

Here, we use different knowledge sources that work together in order to solve the localisation task. We have AuditoryFrontEndKS for extract auditory cues from the ear signals sampled at 16 kHz, DnnLocationKS with 16 frequency channels, ConfusionKS, ConfusionSolvingKS, and RotationKS for the actual localisation task. The `Param` tags are parameters we can pass to the knowledge sources. After setting up which knowledge sources we will use, we connect them with the `Connection` tags. For more information on configuring the blackboard see Configuration.

In the other blackboard configuration files we set up a blackboard for the case of DnnLocationKS without confusion solving by head rotation.

Now, everything is prepared and we can start Matlab in order to perform the localisation. You can just start it and run the following command to see it in action, afterwards we will have a look at what happened:

```
>> localise

-------------------------------------------------------------------------
Source direction   DnnLocationKS w head rot.   DnnLocationKS wo head rot.
-------------------------------------------------------------------------
       0                -5                         -180
     -52               -60                          -55
    -131              -135                         -135
       0                 0                         -180
      30                25                           25
     -30               -30                          -30
------------------------------------------------------------------------
```

As you can see the model with head rotation returned better results than the model without head rotation enabled.

Now, we have a look into the details of the `localise()` function. We will only talk about the parts that are responsible for the task, not for printing out the results onto the screen. First, we define the sources we are going to synthesise and start the Binaural simulator:

```
brirs = { ...

'impulse_responses/qu_kemar_rooms/auditorium3/QU_KEMAR_Auditorium3_src1_xs+0.00_ys+3.97.sofa'
 ...

'impulse_responses/qu_kemar_rooms/auditorium3/QU_KEMAR_Auditorium3_src2_xs+4.30_ys+3.42.sofa'
 ...
    'impulse_responses/qu_kemar_rooms/auditorium3/QU_KEMAR_Auditorium3_src3_xs+2.20_ys-
1.94.sofa'; ...

'impulse_responses/qu_kemar_rooms/auditorium3/QU_KEMAR_Auditorium3_src4_xs+0.00_ys+1.50.sofa'
 ...
    'impulse_responses/qu_kemar_rooms/auditorium3/QU_KEMAR_Auditorium3_src5_xs-
0.75_ys+1.30.sofa'; ...

'impulse_responses/qu_kemar_rooms/auditorium3/QU_KEMAR_Auditorium3_src6_xs+0.75_ys+1.30.sofa'
 ...
    };
headOrientation = 90; % towards y-axis (facing src1)
sourceAngles = [90, 38.5, -41.4, 90, 120, 60] - headOrientation; % phi = atan2d(ys,xs)
```

After that we have a loop over the different sources in which we are loading the corresponding BRIR into the Binaural simulator and run the Blackboard system in `estimateAzimuth` function:

v: latest

```
for ii = 1:length(sourceAngles)
    direction = sourceAngles(ii);
    sim.Sources{1}.IRDataset = simulator.DirectionalIR(brirs{ii});
    sim.rotateHead(headOrientation, 'absolute');
    sim.Init = true;
    % DnnLocationKS w head rot.
    phi1 = estimateAzimuth(sim, 'BlackboardDnn.xml');
    resetBinauralSimulator(sim, headOrientation);
    % DnnLocationKS wo head rot.
    phi2 = estimateAzimuth(sim, 'BlackboardDnnNoHeadRotation.xml');
    sim.ShutDown = true;
end
```

As we run four different blackboards after each other, we have to reinitialise the Binaural simulator in between.

GMM-based localisation under reverberant conditions

The Two!Ears Auditory Model comes with several knowledge sources that work together to estimate the perceived azimuth of a sound source, see Localisation knowledge sources for a summary. One stage of this process is the mapping of the extracted features like ITDs and ILDs to the perceived azimuth angle. This mapping is highly influenced by the environment. For example, if you are in a room the ITD values will look quite different than in the case of an anechoic chamber. That is the reason why we have different knowledge sources that do this mapping: DnnLocationKS, GmmLocationsKS, and ItdLocationKS. ItdLocationKS utilises a simple lookup table for the mapping works well in the case of Prediction of localisation in spatial audio systems. GmmLocationsKS is at the moment trained only for anechoic condition. In this example we have a look at GmmLocationsKS which was trained with a multi-conditional training approach to work under reverberant conditions [MaEtAl2015dnn]. Beside this, GmmLocationsKS works in the same way as DnnLocationKS and connects with ConfusionKS, ConfusionSolvingKS, and RotationKS to solve front-back confusions.

In this example we will have a look at localisation in a larger room, namely the BRIR data set measured in TU Berlin, room Auditorium 3, which provides six different loudspeaker positions as possible sound sources. All files can be found in the `examples/localisation_GMMs` folder which consists of the following files:

```
BlackboardDnnNoHeadRotation.xml
BlackboardDnn.xml
estimateAzimuth.m
localise.m
resetBinauralSimulator.m
setupBinauralSimulator.m
```

The setup is very similar to Localisation with and without head rotations with a few exceptions. First, the setup of the Binaural simulator is different as we use BRIRs instead of HRTFs, and have one impulse response set for every sound source. The initial configuration of the Binaural simulator is provided by the `setupBinauralSimulator` function:

```
sim = simulator.SimulatorConvexRoom();
set(sim, ...
    'BlockSize',            4096, ...
    'SampleRate',           44100, ...
    'NumberOfThreads',      1, ...
    'LengthOfSimulation',   1, ...
    'Renderer',             @ssr_brs, ...
    'Verbose',              false, ...
    'Sources',              {simulator.source.Point()}, ...
    'Sinks',                simulator.AudioSink(2) ...
    );
set(sim.Sinks, ...
    'Name',                 'Head', ...
    'Position',             [ 0.00  0.00  0.00]' ...
    );
set(sim.Sources{1}, ...
    'AudioBuffer',          simulator.buffer.Ring(1) ...
    );
set(sim.Sources{1}.AudioBuffer, ...
    'File', 'sound_databases/grid_subset/s1/bbaf2n.wav' ...
    );
```

Here, we configure it to use the `@ssr_brs` renderer which is needed for BRIRs, define the speech signal to use, but don't provide a BRIR yet as this will be done on the fly later on.

We have four different configuration files for setting up the Blackboard system. As an example, we list the file `BlackboardDnn.xml``:

v: latest

```xml
<?xml version="1.0" encoding="utf-8"?>
<blackboardsystem>

    <dataConnection Type="AuditoryFrontEndKS"/>

    <KS Name="loc" Type="GmmLocationKS"/>
    <KS Name="conf" Type="ConfusionKS"/>
    <KS Name="confSolv" Type="ConfusionSolvingKS"/>
    <KS Name="rot" Type="RotationKS">
        <Param Type="ref">robotConnect</Param>
    </KS>

    <Connection Mode="replaceOld" Event="AgendaEmpty">
        <source>scheduler</source>
        <sink>dataConnect</sink>
    </Connection>
    <Connection Mode="replaceOld">
        <source>dataConnect</source>
        <sink>loc</sink>
    </Connection>
    <Connection Mode="add">
        <source>loc</source>
        <sink>conf</sink>
    </Connection>
    <Connection Mode="replaceOld" Event="ConfusedLocations">
        <source>conf</source>
        <sink>rot</sink>
    </Connection>
    <Connection Mode="add" Event="ConfusedLocations">
        <source>conf</source>
        <sink>confSolv</sink>
    </Connection>

</blackboardsystem>
```

Here, we use different knowledge sources that work together in order to solve the localisation task. We have AuditoryFrontEndKS for extract auditory cues from the ear signals, GmmLocationsKS channels, ConfusionKS, ConfusionSolvingKS, and RotationKS for the actual localisation task. The `Param` tags are parameters we can pass to the knowledge sources. After setting up which knowledge sources we will use, we connect them with the `Connection` tags. For more information on configuring the blackboard see Configuration.

In the other blackboard configuration files we set up a blackboard for the case of GmmLocationsKS without confusion solving by head rotation.

Now, everything is prepared and we can start Matlab in order to perform the localisation. You can just start it and run the following command to see it in action, afterwards we will have a look at what happened:

```
>> localise

------------------------------------------------------------------------
Source direction   GmmLocationKS w head rot.   GmmLocationKS wo head rot.
------------------------------------------------------------------------
            0                  0                          -180
          -52                -55                          -100
         -131               -135                          -140
            0                  0                          -180
           30                 30                            30
          -30                -30                           -30
-----------------------------------------------------------------------
```

As you can see the model with head rotation returned better results than the model without head rotation enabled.

Now, we have a look into the details of the `localise()` function. We will only talk about the parts that are responsible for the task, not for printing out the results onto the screen. First, we define the sources we are going to synthesise and start the Binaural simulator:

```
brirs = { ...

'impulse_responses/qu_kemar_rooms/auditorium3/QU_KEMAR_Auditorium3_src1_xs+0.00_ys+3.97.sofa'
 ...

'impulse_responses/qu_kemar_rooms/auditorium3/QU_KEMAR_Auditorium3_src2_xs+4.30_ys+3.42.sofa'
 ...
    'impulse_responses/qu_kemar_rooms/auditorium3/QU_KEMAR_Auditorium3_src3_xs+2.20_ys-
1.94.sofa'; ...

'impulse_responses/qu_kemar_rooms/auditorium3/QU_KEMAR_Auditorium3_src4_xs+0.00_ys+1.50.sofa'
 ...
    'impulse_responses/qu_kemar_rooms/auditorium3/QU_KEMAR_Auditorium3_src5_xs-
0.75_ys+1.30.sofa'; ...

'impulse_responses/qu_kemar_rooms/auditorium3/QU_KEMAR_Auditorium3_src6_xs+0.75_ys+1.30.sofa'
 ...
    };
headOrientation = 90; % towards y-axis (facing src1)
sourceAngles = [90, 38.5, -41.4, 90, 120, 60] - headOrientation; % phi = atan2d(ys,xs)
```

After that we have a loop over the different sources in which we are loading the corresponding BRIR into the Binaural simulator and run the Blackboard system inside the `estimateAzimuth` function:

```
for ii = 1:length(sourceAngles)
    direction = sourceAngles(ii);
    sim.Sources{1}.IRDataset = simulator.DirectionalIR(brirs{ii});
    sim.rotateHead(headOrientation, 'absolute');
    sim.Init = true;
    % GmmLocationKS w head rot.
    phi1 = estimateAzimuth(sim, 'BlackboardDnn.xml');
    resetBinauralSimulator(sim, headOrientation);
    % GmmLocationKS wo head rot.
    phi2 = estimateAzimuth(sim, 'BlackboardDnnNoHeadRotation.xml');
    sim.ShutDown = true;
end
```

As we run four different blackboards after each other, we have to reinitialise the Binaural simulator in between.

Train sound type identification models

- Example step-through

Part of the Two!Ears Auditory Model is the Identity knowledge source: IdentityKS which can be instantiated (multiple times) to identify the type of auditory objects, like “speech”, “fire”, “knock” etc. Each IdentityKS needs a source type model – this example shows one possibility to train such a model. Have a look at the Identification of sound types to see how these models are being used in the Blackboard system.

The base folder for this example is `examples/train_identification_model`, with the example script file being `trainAndTestCleanModel.m`. Later in the model training process, new directories with names like `Training.2015.08.03.14.57.21.786` will be created by the training pipeline, holding log files of the training, file lists of the used training and testing data, and of course the trained models. These are the models to be used in the IdentityKS, then. To see if everything is working, just run

```
>> trainAndTestCleanModel;
```

Example step-through

To dive into the example, load up Matlab, navigate into the example directory, and open `trainAndTestCleanModel.m`, which contains a function (also usable as a script). Let’s have a look before firing it up!

Start-up

First thing happening in there is the

```
startTwoEars();
```

command. This simply start the Two!Ears Auditory Model and adds all necessary paths to your Matlab paths.

Feature and model creators

The next code paragraph first creates the basic pipeline object of type TwoEarsIdTrainPipe, and then sets two defining options: The *feature creator* and the *model creator*.

```
pipe = TwoEarsIdTrainPipe();
pipe.featureCreator = featureCreators.FeatureSet1Blockmean();
pipe.modelCreator = modelTrainers.GlmNetLambdaSelectTrainer( ...
    'performanceMeasure', @performanceMeasures.BAC2, ...
    'cvFolds', 7, ...
    'alpha', 0.99 );
```

In this case, an L1-regularized sparse logistic regression model will be trained through the use of the GlmNetLambdaSelectTrainer, which is a wrapper for GLMNET. A pile of auditory features will be used in this model, processed and compiled by the FeatureSet1Blockmean feature creator. Have a look into the respective sections to learn more!

Training and testing sets

The models will be trained using a particular set of sounds, specified in the trainset flist. For this example, the IEEE AASP single event sounds serve as training material. There are sounds for several classes like “laughter”, “keys”, “speech”, etc. If you don’t call the `trainAndTestCleanModel` function with a different class name, a model for the “speech” class will be trained (this is specified in the third line). Irregardless of the class the model is trained for, all sounds listed in the flist (have a look) will be used for training – but only the ones belonging to the model class will serve as “positive” examples.

```
pipe.trainset =
'learned_models/IdentityKS/trainTestSets/IEEE_AASP_80pTrain_TrainSet_1.flist';
pipe.testset =
'learned_models/IdentityKS/trainTestSets/IEEE_AASP_80pTrain_TestSet_1.flist';
```

The `testset` specifies files used for testing the trained model. This is not necessary for the model creation, it only serves as an immediate way of providing feedback about the model performance after training. Of course the `testset` must only contain files that have not been used for training, to test for generalisation of the model.

Scene configuration

A “clean” scene configuration is used to train this model. That means: the sound sources are positioned at 0° azimuth relative to the head, there is no interfering noise, and no reverberation (free-field conditions). Have a look into the respective training pipeline documentation part to get to know the many possibilities to configure the acoustic training scene.

```
sc = dataProcs.SceneConfiguration(); % clean
pipe.setSceneConfig( [sc] );
```

Running the pipeline

After everything is set up, the pipeline has to be initialised and can then be run.

```
pipe.init();
modelPath = pipe.pipeline.run( {classname}, 0 );
```

Initialisation can take some time depending on the files for training and testing, and whether they are available through a local copy of the Two!Ears database, through the download cache of the remote Two!Ears database, or whether they have to be downloaded from there first. The time needed for actually running the pipeline can vary substantially, depending on

- the total accumulated length of sound files used
- the scene configuration – using reverberation or noise interference makes the binaural simulation take longer
- the features having to be extracted by the Auditory front-end
- the type of model (training) – there are big differences here, as the computational effort can be much higher for some models than for others (GLMNET, the one used here, is pretty fast)
- and whether the files have been processed in this configuration before or not. The pipeline saves intermediate files after each processing stage (binaural simulation, auditory front-end, feature creation) for each sound file and each configuration, and it finds those files later, if a file is to be processed in the same (or partly the same) configuration. This way, a lot of time-consuming preprocessing can be saved. You can try it – interrupt the preprocessing at any moment by hitting *ctrl+c*, and restart the script. You will see that all processed files/stages won’t be done again.

After successful training and testing, you should see something like

```
Running: MultiConfigurationsEarSignalProc
==========================================
.C:\projekte\twoEars\wp1git\tmp\sound_databases\IEEE_AASP\alert\alert11.wav
...

Running: MultiConfigurationsAFEmodule
==========================================
.C:\projekte\twoEars\wp1git\tmp\sound_databases\IEEE_AASP\alert\alert11.wav
...

Running: MultiConfigurationsFeatureProc
==========================================
.C:\projekte\twoEars\wp1git\tmp\sound_databases\IEEE_AASP\alert\alert11.wav
...

Running: GatherFeaturesProc
==========================================
.C:\projekte\twoEars\wp1git\tmp\sound_databases\IEEE_AASP\alert\alert11.wav
...

===================================
##   Training model "speech"
===================================



==  Training model on trainSet...


Run on full trainSet...
GlmNet training with alpha=0.990000
   size(x) = 5040x846



Run cv to determine best lambda...
Starting run 1 of CV... GlmNet training with alpha=0.990000
   size(x) = 4111x846

Applying model to test set...
Done. Performance = 0.842686

...

Calculate Performance for all
lambdas..................................................Done

==  Testing model on testSet...




===================================
##   "speech" Performance: 0.942548
===================================

 -- Model is saved at C:\projekte\twoEars\twoears-
examples\train_identification_model\Training.2015.08.06.15.44.52.582 --
>>
```

v: latest

The stated performance is on the test set, and the path afterwards indicates the location of the model on your drive.

Identification of sound types

This example particularly serves to demonstrate two aspects of the Two!Ears Auditory Model:

- Building a Blackboard system dynamically in code (instead of via xml definition, as demonstrated in the Localisation with and without head rotations example)
- Using identity knowledge sources with source type models to generate hypotheses about the type of sound objects in an auditory scene.

The base folder for this example is `examples/identification`, with the main example script file being `identify.m`. Other than that, there is the file `SceneDescription.xml` which describes the Binaural simulator configuration, there are five directories with names like "Training.2015.07.24.00.23.42.772", which hold the used source type models, a sound file list `shortTest.flist`, and subfunctions (`buildIdentificationBBS.m`, `makeTestSignal.m`, `setDefaultIdModels.m`) used in `identify.m`. Have a look at Train sound type identification models to see how the source type models have been created. To see if everything is working, just run

```
>> identify;
```

Example step-through

To dive into the example, load up Matlab, navigate into the example directory, and open `identify.m`, which contains a function (also usable as a script). Let's have a look before firing it up!

Specifying the identification models

The function `identify` takes a parameter, which shall specify the models to be used for source type identification. However, for this example, you can run without setting this parameter, and let the function `setDefaultIdModels` do this for you.

```
if nargin < 1, idModels = setDefaultIdModels(); end
```

This function specifies five source type models by giving the directories they are located in and the class they are trained to identify, respectively.

Starting Two!Ears

Next, we initialise the Two!Ears Auditory Model:

```
startTwoEars();
```

Creating a test scene

To test the identification models, a test scene is compiled from audio events (IEEE AASP single event sounds database) from several wav-files, listed in `shortTest.flist`. All those wav-files have not been used for training of the tested models (they have all been trained using the same trainset), so the models have never “seen” (or better: “heard”) these actual sounds. The function `makeTestSignal` not only compiles the audio, but also reads the “ground truth”, i.e. the on- and offset times of the respective events from the annotation files. The scene is about 45 seconds long. The events are concatenated in random order (with 0.5s inbetween two events).

Initialising the Binaural Simulator

The next code paragraph deals with initialisation of the Binaural simulator. More specific, the acoustic sources are set to an head-relative azimuth of zero degrees and three meters distance. Free-field conditions (no reverberation) are set through absence of a room definition (either in the `SceneDescription.xml` or in code). The models have been trained under the same conditions.

Building the example Blackboard System

The construction of the Blackboard system for this example is done directly in Matlab (versus via xml) in the function `buildIdentificationBBS`. Let’s look into it, we first cr
new BlackboardSystem object:

v: latest

```
bbs = BlackboardSystem(1);
```

This object is our access point and frame for the simulation. We first set up the connection to the Binaural simulator and the Auditory front-end:

```
bbs.setRobotConnect(sim);
bbs.setDataConnect('AuditoryFrontEndKS');
```

Followed by invocation of the identity knowledge sources using the function `createKS` of the Blackboard system, which also takes care to place the Auditory front-end requests at the AuditoryFrontEndKs. The identity knowledge sources need name and location of the source type models they load and represent in the system. We set their invocation frequency to 100ms (default value is 250ms):

```
for ii = 1 : numel( idModels )
    idKss{ii} = bbs.createKS('IdentityKS', {idModels(ii).name, idModels(ii).dir});
    idKss{ii}.setInvocationFrequency(10);
end
```

We create one more knowledge source, the IdTruthPlotKS – it's not really a knowledge source in the sense of the word in this case, but a handy way to implement a debugging tool for live-inspection of the identity information in the blackboard system. It needs the ground truth information passed to it:

```
idCheat = bbs.createKS('IdTruthPlotKS', {labels, onOffsets});
```

In the last lines dealing with blackboard system construction, we connect the different modules:

```
bbs.blackboardMonitor.bind({bbs.scheduler}, {bbs.dataConnect}, 'replaceOld', 'AgendaEmpty'
);
bbs.blackboardMonitor.bind({bbs.dataConnect}, idKss, 'replaceOld' );
bbs.blackboardMonitor.bind(idKss, {idCheat}, 'replaceParallelOld' );
```

- The `dataConnect` gets bound to the scheduler – this way, the next chunk of acoustic data is fetched whenever no more knowledge source needs to be processed

v: latest

- The identity knowledge sources get bound to the `dataConnect` (which we have instantiated to be the AuditoryFrontEndKS before)
- The `IdTruthPlotKS` gets bound to the identity knowledge sources, which produce the identity hypotheses.

Running the simulation

With the blackboard system set up, we can start the simulation,

```
bbs.run();
```

which will trigger the first fetching of acoustic data by the system, followed by processing and triggering subsequent events and knowledge source executions. The blackboard system is set up to "verbose" mode, printing the events and executions on the Matlab console:

```
-------- [Event Fired:] Scheduler -> (AgendaEmpty) -> AuditoryFrontEndKS
-------- [Executing KS:] AuditoryFrontEndKS
-------- [Event Fired:] AuditoryFrontEndKS -> (KsFiredEvent) -> IdentityKS[clearthroat]
-------- [Event Fired:] AuditoryFrontEndKS -> (KsFiredEvent) -> IdentityKS[knock]
-------- [Event Fired:] AuditoryFrontEndKS -> (KsFiredEvent) -> IdentityKS[switch]
-------- [Event Fired:] AuditoryFrontEndKS -> (KsFiredEvent) -> IdentityKS[keyboard]
-------- [Event Fired:] AuditoryFrontEndKS -> (KsFiredEvent) -> IdentityKS[speech]
-------- [Executing KS:] IdTruthPlotKS
-------- [Executing KS:] IdentityKS[clearthroat]
....Identity Hypothesis: clearthroat with 75% probability.
-------- [Event Fired:] IdentityKS[clearthroat] -> (KsFiredEvent) -> IdTruthPlotKS
-------- [Executing KS:] IdentityKS[knock]
....Identity Hypothesis: knock with 0% probability.
-------- [Event Fired:] IdentityKS[knock] -> (KsFiredEvent) -> IdTruthPlotKS
-------- [Executing KS:] IdentityKS[switch]
....Identity Hypothesis: switch with 6% probability.
-------- [Event Fired:] IdentityKS[switch] -> (KsFiredEvent) -> IdTruthPlotKS
-------- [Executing KS:] IdentityKS[keyboard]
....Identity Hypothesis: keyboard with 0% probability.
-------- [Event Fired:] IdentityKS[keyboard] -> (KsFiredEvent) -> IdTruthPlotKS
-------- [Executing KS:] IdentityKS[speech]
....Identity Hypothesis: speech with 89% probability.
-------- [Event Fired:] IdentityKS[speech] -> (KsFiredEvent) -> IdTruthPlotKS
```

You can see the before installed event bindings in action – the scheduler triggers the AuditoryFrontEndKS, which triggers the IdentityKSs, which place identity hypotheses on the blackboard and trigger the IdTruthPlotKS.

The simulation will take a few minutes – as mentioned, it processes a 45s scene, and at the moment, is not optimised to run in real-time. You can see the progress in the id truth plot, which shows the wave form of the left channel (ear) accompanied by a graphical representation of the events, ground truth versus hypotheses produced by the models:

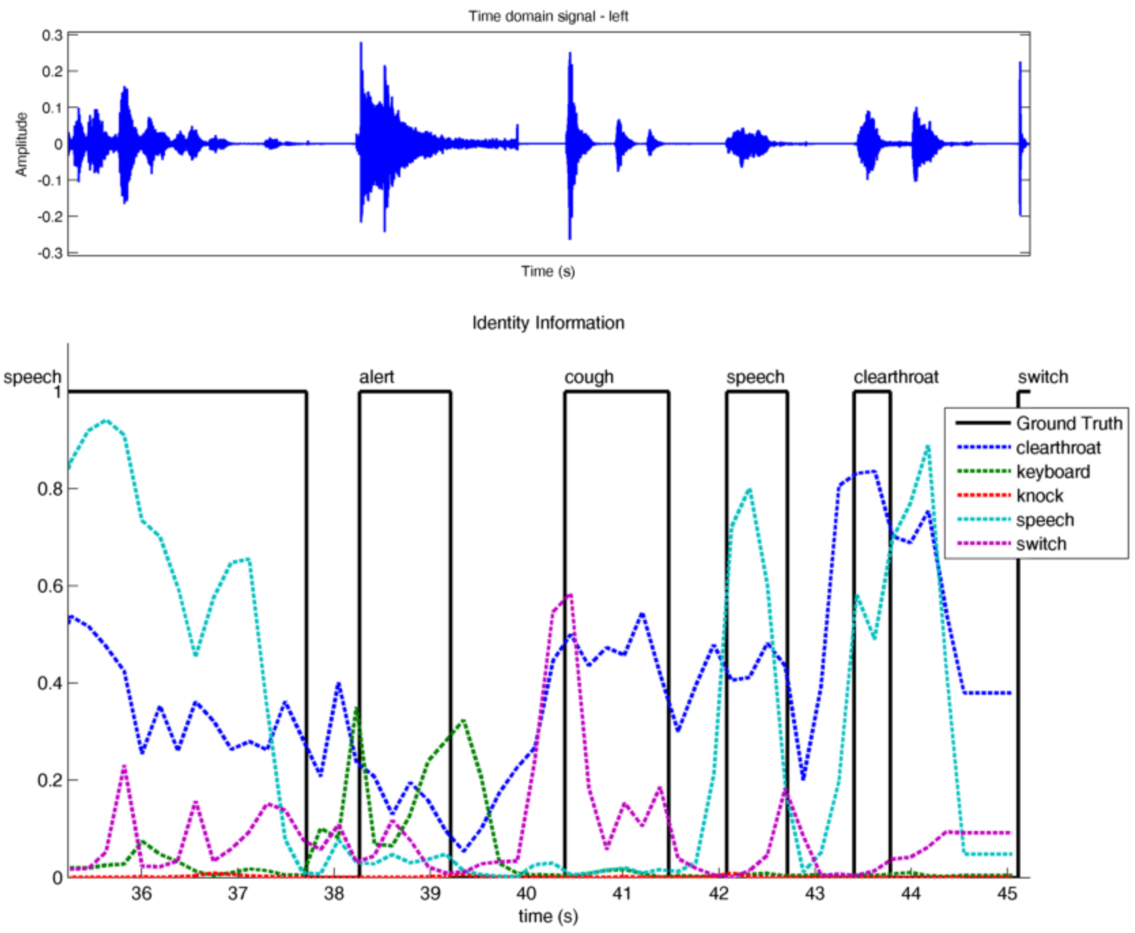


Fig. 58 This is the live plot of hypotheses created by the identity knowledge sources, in comparison to the “ground truth” (as given by annotated on- and offset times for the source sound files).

Evaluating the simulation

Finally, `idScoresRelativeError` calculates an error rate of the tested models for this example:

```
Evaluate scores...

relative error of clearthroat identification model: 0.244339
relative error of keyboard identification model: 0.112274
relative error of knock identification model: 0.034251
relative error of speech identification model: 0.095172
relative error of switch identification model: 0.110843
```

v: latest

The relative error rate here is the over time integrated difference between ground truth and model hypotheses (divided by the length of the simulation).

Segmentation with and without priming

The Blackboard system of the Two!Ears Auditory Model is equipped with a SegmentationKS knowledge source which is capable of generating soft-masks for auditory features in the time-frequency domain. The segmentation framework relies on a probabilistic clustering approach to assign individual time-frequency units to sound sources that are present in the scene. This assignment can either be computed unsupervised or exploit additional prior knowledge about potential source positions provided by the user or e.g. by the DnnLocationKS knowledge source.

Note

To run the examples, an instance of the SegmentationKS knowledge source has to be trained first. Please refer to (Re)train the segmentation stage for details.

This example will demonstrate how the SegmentationKS knowledge source is properly initialised with and without prior knowledge and how the hypotheses which are generated by the segmentation framwork can be used within the Blackboard system. The example can be found in the `examples/segmentation` folder which consists of the following files:

```
demo_segmentation_clean.m
demo_segmentation_noisy.m
demo_segmentation_priming.m
demo_train_segmentation.m
segmentation_blackboard_clean.xml
segmentation_blackboard_noise.xml
segmentation_config.xml
test_scene_clean.xml
test_scene_noise.xml
training_scene.xml
```

Note

The `SegmentationKS` knowledge source is based on Matlab functions that were introduced in release R2013b. Therefore, it is currently not possible to use `SegmentationKS` with earlier versions of Matlab.

The example contains three different demo scenes, namely `demo_segmentation_clean.m`, `demo_segmentation_noisy.m` and `demo_segmentation_priming.m`. The first demo shows the segmentation framework for three speakers in anechoic and undisturbed acoustic conditions without providing prior knowledge about the speaker positions. The scene parameters are specified in the corresponding `test_scene_clean.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<scene
  Renderer="ssr_binaural"
  BlockSize="4096"
  SampleRate="44100"
  LengthOfSimulation = "3"
  HRIRs="impulse_responses/qu_kemar_anechoic/QU_KEMAR_anechoic_3m.sofa">
  <source Name="Speaker1"
          Type="point"
          Position="0.8660 0.5 1.75">
    <buffer ChannelMapping="1"
            Type="fifo"
            File="sound_databases/IEEE_AASP/speech/speech08.wav"/>
  </source>
  <source Name="Speaker2"
          Type="point"
          Position="1 0 1.75">
    <buffer ChannelMapping="1"
            Type="fifo"
            File="sound_databases/IEEE_AASP/speech/speech14.wav"/>
  </source>
  <source Name="Speaker3"
          Type="point"
          Position="0.8660 -0.5 1.75">
    <buffer ChannelMapping="1"
            Type="fifo"
            File="sound_databases/IEEE_AASP/speech/speech07.wav"/>
  </source>
  <sink Name="Head"
        Position="0 0 1.75"
        UnitX="1 0 0"
        UnitZ="0 0 1"/>
</scene>
```

The speaker positions described here correspond to angular positions at -30°, 0° and 30°, respectively. These positions will be fixed for all conditions in this demo. For more documentation on specifying an acoustic scene, see Configuration using XML Scene Description. Additionally, the file `segmentation_blackboard_clean.xml` contains the necessary information to build a Blackboard system with the corresponding SegmentationKS (see Configuration for details):

```
<?xml version="1.0" encoding="utf-8"?>
<blackboardsystem>

    <dataConnection Type="AuditoryFrontEndKS"/>

    <KS Name="seg" Type="SegmentationKS">
        <Param Type="char">DemoKS</Param>
        <Param Type="double">3</Param>
        <Param Type="int">3</Param>
        <Param Type="int">0</Param>
    </KS>

    <Connection Mode="replaceOld" Event="AgendaEmpty">
        <source>scheduler</source>
        <sink>dataConnect</sink>
    </Connection>
    <Connection Mode="replaceOld">
        <source>dataConnect</source>
        <sink>seg</sink>
    </Connection>

</blackboardsystem>
```

The SegmentationKS knowledge source takes four parameters as input arguments. The first parameter is the name of the knowledge source instance which contains previously trained localisation models. For further information about training this specific knowledge source, please refer to (Re)train the segmentation stage. The second parameter defines the block size in seconds on which the segmentation should be performed. In this demo, a block size of 3 seconds is assumed for all cases. The third parameter specifies the number of sources which are assumed to be present in a scene and the fourth parameter is a flag which can be either set to 0 or 1, indicating if an additional background estimation should be performed. If this is the case, the model assumes that individual time-frequency units can either be associated with a sound source or with background noise, which is helpful in noisy acoustic environments but can also degrade performance if no or little background noise is present. As no background noise is assumed in the first demo, this parameter is set to zero accordingly. Running the script `demo_segmentation_clean.m` will produce a result similar to Fig. Fig. 63.

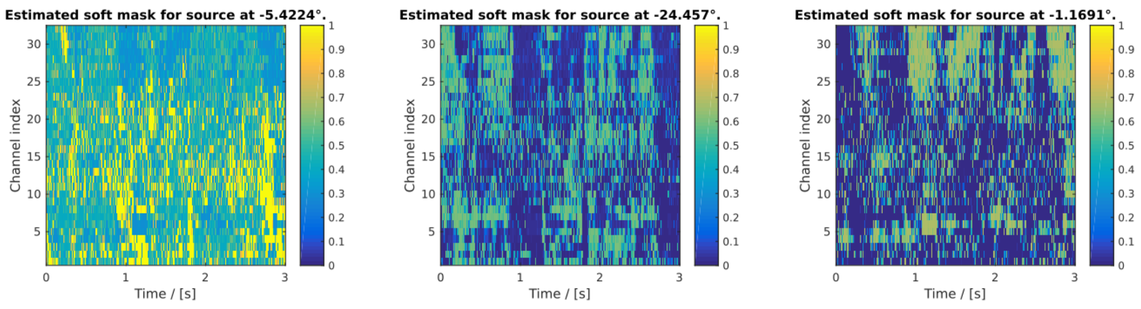


Fig. 63 Figure generated after running the script `demo_segmentation_clean.m`. The figure shows all three soft masks and location estimates for the speech sources that are simulated in this

v: latest

demo. Note that the estimated positions do not necessarily match the true positions (-30°, 0° and 30°) due to the limited localisation capabilities of the SegmentationKS knowledge source. This problem can be circumvented by exploiting prior knowledge about the source positions (see below).

The second demo file `demo_segmentation_noisy.m` provides essentially the same acoustic configuration as the first demo with additional diffuse background noise. This is specified in the corresponding `test_scene_noise.xml` configuration file:

```
<?xml version="1.0" encoding="utf-8"?>
<scene
  Renderer="ssr_binaural"
  BlockSize="4096"
  SampleRate="44100"
  LengthOfSimulation = "3"
  HRIRs="impulse_responses/qu_kemar_anechoic/QU_KEMAR_anechoic_3m.sofa">
  <source Name="Speaker1"
          Type="point"
          Position="0.8660 0.5 1.75">
    <buffer ChannelMapping="1"
            Type="fifo"
            File="sound_databases/IEEE_AASP/speech/speech08.wav"/>
  </source>
  <source Name="Speaker2"
          Type="point"
          Position="1 0 1.75">
    <buffer ChannelMapping="1"
            Type="fifo"
            File="sound_databases/IEEE_AASP/speech/speech14.wav"/>
  </source>
  <source Name="Speaker3"
          Type="point"
          Position="0.8660 -0.5 1.75">
    <buffer ChannelMapping="1"
            Type="fifo"
            File="sound_databases/IEEE_AASP/speech/speech07.wav"/>
  </source>
  <source Type="pwd"
          Name="Noise"
          Azimuths="0 30 60 90 120 150 180 210 240 270 300 330">
    <buffer ChannelMapping="1 2 3 4 5 6 7 8 9 10 11 12"
            Type="noise"
            Variance="0.02"
            Mean="0.0"/>
  </source>
  <sink Name="Head"
        Position="0 0 1.75"
        UnitX="1 0 0"
        UnitZ="0 0 1"/>
</scene>
```

To account for the background noise during the estimation process, the corresponding flag in the blackboard configuration file `segmentation_blackboard_noise.xml` is set to one:

```xml
<?xml version="1.0" encoding="utf-8"?>
<blackboardsystem>

    <dataConnection Type="AuditoryFrontEndKS"/>

    <KS Name="seg" Type="SegmentationKS">
        <Param Type="char">DemoKS</Param>
        <Param Type="double">3</Param>
        <Param Type="int">3</Param>
        <Param Type="int">1</Param>
    </KS>

    <Connection Mode="replaceOld" Event="AgendaEmpty">
        <source>scheduler</source>
        <sink>dataConnect</sink>
    </Connection>
    <Connection Mode="replaceOld">
        <source>dataConnect</source>
        <sink>seg</sink>
    </Connection>

</blackboardsystem>
```

Running the corresponding script `demo_segmentation_noisy.m` will generate an additional soft-mask for the background noise which is shown in Fig. 64.

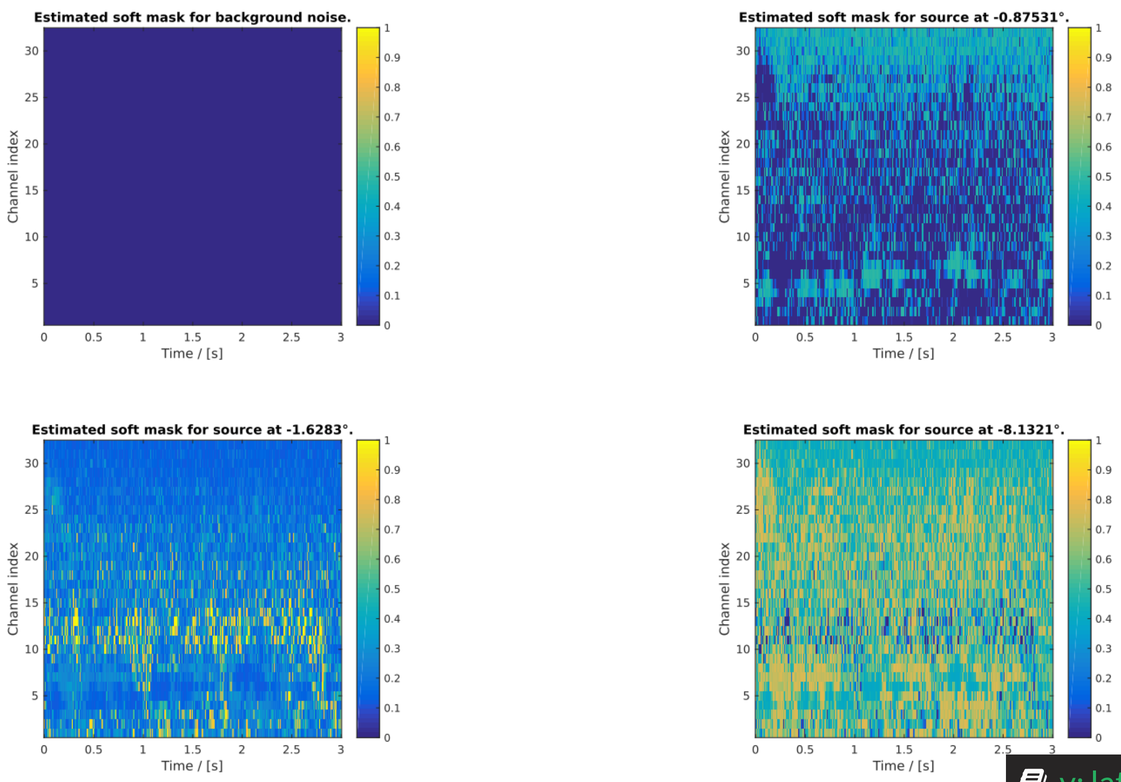


Fig. 64 Figure generated after running the script `demo_segmentation_noisy.m`. The figure shows four soft masks of which three correspond to the individual sources and the remaining one is a soft-mask for the background noise. Note that the latter one only contains very small probabilities for all time-frequency units in this demo. This is due to the fact that stationary white noise was used in this case and the (partially overlapping) speech sources cover a broad range of the whole time-frequency spectrum.

Note

The background noise estimation procedure is based on the assumption that the noise present in the scene is diffuse and hence its directions of arrival follow a uniform distribution around the unit circle. If this condition is not valid and directional noise sources are present, considering them as additional sources instead of using the background estimation procedure might yield better results.

Finally, the third demo shows the possibilities of priming the SegmentationKS knowledge source, which means providing prior knowledge about the source positions before the segmentation is actually performed. For this purpose, the implementation of SegmentationKS provides an additional function `setFixedPositions()` which can be used to manually specify the positions of the sound sources. In the script `demo_segmentation_priming.m`, this is done in the following way:

```
% Provide prior knowledge of the two speaker locations
prior = [-deg2rad(30); deg2rad(30); 0];
bbs.blackboard.KSs{2}.setFixedPositions(prior);
```

It is also possible to exploit this functionality dynamically during runtime by using intermediate results of the DnnLocationKS knowledge source from the blackboard. Note that angular positions are handled in radians within the SegmentationKS framework, hence position estimates in degrees must be converted accordingly. A possible result for this demo is shown in Fig. Fig. 65.

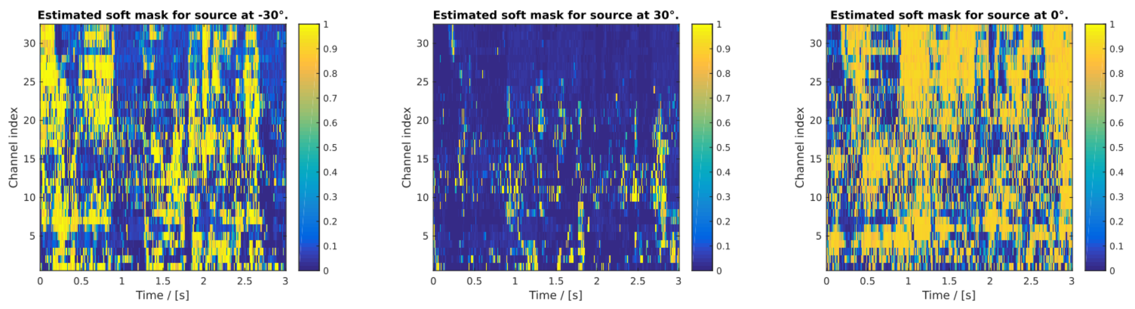


Fig. 65 Figure generated after running the script `demo_segmentation_priming.m`.

v: latest

v: latest

(Re)train the segmentation stage

The SegmentationKS knowledge source of the Two!Ears Auditory Model depends on a localisation model which is based on support vector machine regression. This regression model has to be trained using a set of HRTFs. A demo of how a specific instance of the SegmentationKS can be trained is provided by the script `demo_train_segmentation.m` in the `examples/segmentation` folder. This script shows how the `default` setting of the SegmentationKS knowledge source which is used for all demos is generated.

Before starting with the training of a new model, the configuration of the Binaural simulator for which this model should be used has to be specified. This is done by setting up a training scene. In this case, the training scene is specified in the `training_scene.xml` file:

```xml
<?xml version="1.0" encoding="utf-8"?>
<scene
  BlockSize="4096"
  SampleRate="44100"
  MaximumDelay="0.0"
  NumberOfThreads="1"
  LengthOfSimulation = "5"
  HRIRs="impulse_responses/qu_kemar_anechoic/QU_KEMAR_anechoic_3m.sofa">
  <source Radius="3.0"
          Mute="false"
          Type="point"
          Name="SoundSource">
    <buffer ChannelMapping="1"
        Type="noise"/>
  </source>
  <sink Name="Head"
        Position="0 0 0"
        UnitX="1 0 0"
        UnitZ="0 0 1"/>
</scene>
```

The only parameter that is relevant for the training process is the set of HRTFs, which is taken from the Database `impulse_responses/qu_kemar_anechoic/QU_KEMAR_anechoic_3m.sofa` for this demo. All other parameters only have to match the HRTFs specifications. The current implementation of the training framework uses white noise as a stimulus signal during training.

Besides the scene description file, the only requirement to generate a training script is to provide a unique identifier for the SegmentationKS instance that should be trained. In the file `demo_train_segmentation.m` this is done via

```matlab
ksName = 'DemoKS';
```

Furthermore, some additional parameters that should be used for training can be specified directly in the training script. The additional parameters are optional and will be initialised by the default Auditory front-end values if not explicitly specified. The possible configuration parameters are provided as an example in `demo_train_segmentation.m`:

```matlab
nChannels = 32;             % Number of filterbank channels
winSize = 0.02;             % Size of the processing window in [s]
hopSize = 0.01;             % Frame shift in [s]
fLow = 80;                  % Lowest filterbank center frequency in [Hz]
fHigh = 8000;               % Highest filterbank center frequency in [Hz]
```

If all of the described prerequisites are met, an instance of the SegmentationKS can be created:

```matlab
segKS = SegmentationKS(ksName, ...
    'NumChannels', nChannels, ...
    'WindowSize', winSize, ...
    'HopSize', hopSize, ...
    'FLow', fLow, ...
    'FHigh', fHigh, ...
    'Verbosity', true);     % Enable status messages during training
```

This instance can subsequently be used to automatically generate all files required for the training process:

```matlab
xmlSceneDescription = 'training_scene.xml';
segKS.generateTrainingData(xmlSceneDescription);
```

If this is completed, the `train()` function can be used to start training of the regression models.

```matlab
segKS.train();
```

v: latest

The `train()` command will produce an error message if a set of trained models already exist for the identifier the SegmentationKS was instantiated with. Overwriting existing models has to be explicitly enforced by calling the `train()` method with a additional `doOverwrite` flag which has to be set to `true`:

```
segKS.train(true);
```

Note

The training process may take up to several hours depending on the available computational ressources. It is generally recommended to set the `Verbosity` flag to `true` at instantiation, in order to receive important status and progress messages during the training process.

If training is completed, the generated training files are not needed anymore and can be deleted if no re-training should be performed. This can be done by calling the `removeTrainingData` method:

```
segKS.removeTrainingData();
```

Stream binaural signals from BASS to Matlab

This tutorial shows an example of how to control the BASS component and retrieve audio streams in Matlab, using the *matlab-genomix* bridge.

Preliminary steps

In order to follow this tutorial, you will need:

- A Linux system with the robotic tools and BASS installed (*c.f.* Installation of the robotic tools). We will call this system the **BASS host**.
- An ALSA-compliant sound acquisition interface with at least two input channels, and two microphones plugged into it. The interface must be connected to the BASS host.

  Note

  Alternatively, if you do not possess an external sound interface but the BASS host has an integrated sound card and microphone, you still might be able to follow the tutorial. Keep in mind though that if there is only one microphone, you will not have a genuine stereo signal, but a simulated one from your mono input.

- A computer with Matlab and the *matlab-genomix* bridge installed. We will call it the **remote client**. The BASS host and the remote client could possibly, but not necessarily, be the same computer.

On the BASS host, open 3 new terminals. In the first terminal, run the command:

```
$ roscore
```

This launches the ROS middleware. ROS nodes can now connect to this node called the ROS master. In the second terminal, run the command:

```
$ genomixd
```

This launches a *genomix* server, now waiting for incoming connections from clients on port 8080 by default. In the third terminal, run the command:

```
$ bass-ros
```

This is the BASS component, now running on the system. The name `bass-ros` specifies that this GenoM3 component uses the ROS middleware. So it is actually a ROS node, connected to the ROS master running in the first terminal.

For the moment, the BASS component is not doing anything. It is waiting for requests from a client (which will be Matlab here) to start services. This is the followed process:

1. The client emits a HTTP message destined for the *genomix* server, requesting to call a service of the BASS component.
2. *genomix* executes the call directed at the BASS component.
3. When the service is completed, BASS returns its output to *genomix*, and *genomix* relays it back to the client.

Keep the third terminal running BASS visible on the screen. When we will call some services, we will notice their effect on the component's standard output stream (*stdout*).

Control BASS to start an acquisition

On the remote client, start a Matlab session and make sure that *matlab-genomix* is in the Matlab path (*c.f.* Installation of the robotic tools).

Connect to *genomix* and load BASS

If you have Matlab on the same computer where the *genomix* server is running, you can simply connect to *genomix* with:

```
>> client = genomix.client
client =

  client with no properties.
```

v: latest

This will attempt a connection on `localhost:8080` by default. Otherwise if your BASS host and your remote client are two different computers, get the IP address of the BASS host and override the default value with:

```
>> client = genomix.client('xxx.xxx.xxx.xxx:8080') % write the IP address of BASS host
```

Then, load BASS:

```
>> bass = client.load('bass')

bass =

  component with properties:

        genom_state: [function_handle]
               kill: [function_handle]
       connect_port: [function_handle]
    connect_service: [function_handle]
               Stop: [function_handle]
        ListDevices: [function_handle]
    DedicatedSocket: [function_handle]
              Audio: [function_handle]
     abort_activity: [function_handle]
            Acquire: [function_handle]
        CloseSocket: [function_handle]
```

The returned handle `bass` has a list of properties either corresponding to services (*e.g.* `Acquire`) or ports (*e.g.* `Audio`) of the component.

Get the name of your sound interface

Invoke the `ListDevices` service to get the name of your ALSA device:

```
>> bass.ListDevices();
```

The detected sound devices are listed on the components's standard output stream (*stdout*). On the BASS host, look in the terminal where the component is running, and find a line that matches your interface, something like:

```
hw:1,0 [Babyface2361116] [USB Audio]
```

v: latest

The leading string, `hw:1,0` in the example, is the name of your ALSA device.

Start an acquisition

We will now use the `Acquire` service to start an acquisition.

Caution

By default, services are invoked synchronously, *i.e.* the command to invoke them only returns after completion of the service. As the acquisition runs indefinitely, the `Acquire` service never completes unless you explicitly stop it. So you must invoke this service asynchronously, *i.e.* the command invoking the service returns immediately and the service output can be retrieved later on. Otherwise you will be blocked in the Matlab command window without control, including stopping the service. If this happens, a solution is to kill the Matlab process and start again.

The service can be invoked asynchronously by providing the `'-a'` option:

```
>> r = bass.Acquire('-a')
 string device: ALSA name of the sound device (hw:1,0) >
```

The `Acquire` service expects input arguments. As we did not passed them to the function directly, they are prompted interactively. Enter values according to your sound interface (see the example below):

- For the `device` parameter, take the value you obtained at the previous step.
- For the `sampleRate` parameter, choose a sampling rate that your device supports. The default value (44100 Hz) is most likely to work.
- For the `nFramesPerChunk` parameter, choose a chunk size that your device supports. Some devices only support powers of 2 (*e.g.* 512, 1024, 2048...), refer to your device manual.
- For the `nChunksOnPort` parameter, choose a value that is big enough so that the output port of BASS streams a few seconds of audio data. For instance, with the default values (44100 Hz for the sampling rate and 2205 frames for the chunk size), keep 80 chunks on the port to have 4 seconds:

$$\begin{aligned} duration &= nChunksOnPort * nFramesPerChunk / sampleRate \\ &= 80 * 2205/44100 \\ &= 4s \end{aligned}$$

```
>> r = bass.Acquire('-a')
 string device: ALSA name of the sound device (hw:1,0) > 'hw:1,0'
 unsigned long sampleRate: Sample rate in Hz (44100) > 44100
 unsigned long nFramesPerChunk: Chunk size in frames (2205) > 2205
 unsigned long nChunksOnPort: Port size in chunks (20) > 80

r =

  request with properties:

       status: 'sent'
       result: []
    exception: []
```

If starting the acquisition succeeded, you should see the status `'sent'` in the returned handle. Otherwise, the status would be `'error'`, check then the error message printed in the terminal on the BASS host. It could be an invalid input parameter.

Note

The parameter prompts like `string device: ALSA name of the sound device (hw:1,0) >` contains valuable information, *i.e.* the data type of the parameter, its name, a short description and a default value between parenthesis that will be used if you press enter without specifying another value. All this information comes from the dotgen file of the component, and is part of its definition.

Get audio data in Matlab

You can read the output port of BASS, named `Audio`, in Matlab:

```
>> p = bass.Audio()
p =

    Audio: [1x1 struct]

>> p.Audio
ans =

         sampleRate: 44100
      nChunksOnPort: 80
    nFramesPerChunk: 2205
     lastFrameIndex: 251370
               left: {176400x1 cell}
              right: {176400x1 cell}
```

The data structure shown here is retrieved when reading the port with function `bass.Audio()`. The audio signals are stored in the `left` and `right` fields. Note the presence of the index `lastFrameIndex` for keeping track of the data.

If your remote client computer has speakers, you can listen to the retrieved signals:

```
% Speak in the microphones for a few seconds

% Read the last few recorded seconds
>> p = bass.Audio();

% Play the recorded sound, on left channel for instance
>> soundsc(cell2mat(p.Audio.left), p.Audio.sampleRate);
```

Notice how the duration of the sound matches the one you selected with parameter `nChunkOnPort` when starting the acquisition.

End the session

When you are done, you can clear the used objects in Matlab:

```
>> delete(bass);
>> delete(client); % This closes the connection to genomix
```

On the BASS host, you can kill processes `roscore`, `genomixd` and `bass-ros` by typing `Control-c` in each terminal.

Prediction of coloration in spatial audio systems

Assume we have a live performance of three singers at different positions. Using spatial audio systems like Wave Field Synthesis we might be able to synthesize their corresponding sound field in a way that we are convinced they are exactly at those positions they were during their actual performance. But what we most probably not be able to synthesize correctly is the timbre of those singers and listeners will perceive a coloration compared to the original performance.

We did different listening tests where we investigated the amount of coloration listeners perceive in different Wave Field Synthesis systems with a varying number of loudspeakers and the distance between adjacent loudspeakers.

The Two!Ears model has a ColorationKS that is able to predict the amount of coloration compared to a reference. The model learns this reference on the fly by choosing the first audio input it gets after the start of the Blackboard system.

In the following we show you how to get the results for listening tests from our database and how to use the Two!Ears model to predict the data.

Getting listening test data

The Database contains lots of so called human labels, which are varying data, that all comes from listening tests and has somehow human behavior as input. This can be in the form of direct rating of specific attributes like localisation or coloration, but also indirect input like measurements of the head movements of the listeners during their tasks.

In the following we are interested in modelling the results for the coloration experiment performed with WFS, which is described in 2015-10-01: Coloration of a point source in Wave Field Synthesis revisited. We did the experiment for different listening positions loudspeaker arrays, and audio source materials. In this example, we will focus on the central listening position, a circular loudspeaker array, and music as source material.

First, we will get the source signal and the listening test results and corresponding BRS files:

```
sourceMaterial = audioread(db.getFile('stimuli/anechoic/aipa/music1_48k.wav'));
humanLabels = readHumanLabels(['experiments/2015-10-01_wfs_coloration/', ...
                'human_label_coloration_wfs_circular_offcenter_music.csv']);
```

You could listen to the audio source material with and have a look at the results from the listening test:

```
>> sound(sourceMaterial, 48000);
>> humanLabels

humanLabels =

    'experiments/2015-10-01_wfs_coloration/brs/wfs_...'    [-0.9622]    [0.0419]
    'experiments/2015-10-01_wfs_coloration/brs/wfs_...'    [ 0.0075]    [0.2127]
    'experiments/2015-10-01_wfs_coloration/brs/wfs_...'    [ 0.0730]    [0.2107]
    'experiments/2015-10-01_wfs_coloration/brs/wfs_...'    [-0.1764]    [0.2212]
    'experiments/2015-10-01_wfs_coloration/brs/wfs_...'    [-0.1103]    [0.2410]
    'experiments/2015-10-01_wfs_coloration/brs/wfs_...'    [-0.2899]    [0.2165]
    'experiments/2015-10-01_wfs_coloration/brs/wfs_...'    [-0.3714]    [0.2348]
    'experiments/2015-10-01_wfs_coloration/brs/wfs_...'    [-0.7415]    [0.1406]
    'experiments/2015-10-01_wfs_coloration/brs/wfs_...'    [-0.4208]    [0.1803]
    'experiments/2015-10-01_wfs_coloration/brs/wfs_...'    [ 0.7443]    [0.1587]
```

The first column of `humanLabels` lists the corresponding BRS file used during the experiment, the second column the median coloration rating – ranging from -1 for *not colored* to 1 for *strongly colored* – and the third column the confidence interval of the ratings.

Setting up the Binaural Simulator

The Binaural simulator is then setup with the BRS renderer:

v: latest

```
sim = simulator.SimulatorConvexRoom();
set(sim, ...
    'BlockSize',            48000, ...
    'SampleRate',           48000, ...
    'NumberOfThreads',      3, ...
    'LengthOfSimulation',   5, ...
    'Renderer',             @ssr_brs, ...
    'Verbose',              false, ...
    'Sources',              {simulator.source.Point()}, ...
    'Sinks',                simulator.AudioSink(2) ...
    );
set(sim.Sinks, ...
    'Name',                 'Head', ...
    'UnitX',                [ 0.00 -1.00  0.00]', ...
    'UnitZ',                [ 0.00  0.00  1.00]', ...
    'Position',             [ 0.00  0.00  1.75]' ...
    );
set(sim.Sources{1}, ...
    'AudioBuffer',          simulator.buffer.FIFO(1) ...
    );
% First BRS entry corresponds to the reference condition
sim.Sources{1}.IRDataset = simulator.DirectionalIR(humanLabels{1,1});
sim.Sources{1}.setData(sourceMaterial);
sim.Init = true;
```

Estimating the coloration with the Blackboard

Now the audio part is prepared and we only have to setup the Blackboard system and estimate a coloration value for every condition with the ColorationKS:

```
% === Estimate reference
bbs = BlackboardSystem(0);
bbs.setRobotConnect(sim);
bbs.setDataConnect('AuditoryFrontEndKS');
ColorationKS = bbs.createKS('ColorationKS', {'music'});
bbs.blackboardMonitor.bind({bbs.scheduler}, {bbs.dataConnect}, 'replaceOld', ...
                           'AgendaEmpty' );
bbs.blackboardMonitor.bind({bbs.dataConnect}, {ColorationKS}, 'replaceOld' );
bbs.run(); % The ColorationKS runs automatically until the end of the signal
sim.ShutDown = true;

% === Estimate coloration for every conditions
for jj = 1:size(humanLabels,1)
    sim.Sources{1}.IRDataset = simulator.DirectionalIR(humanLabels{jj,1});
    sim.Sources{1}.setData(sourceMaterial);
    sim.Init = true;
    bbs.run()
    prediction(jj) = ...
        bbs.blackboard.getLastData('colorationHypotheses').data.differenceValue;
    sim.ShutDown = true;
end
```

The model returned a coloration estimation for every condition in the range of `0..1` where `0` means no coloration and `1` strongly colored:

```
>> prediction

prediction =

    0.0035    0.5316    0.5566    0.4907    0.5321    0.3595    0.3080    0.2261

    0.2608    1.7576
```

The anchor condition, which is the very last entry, was rated to be even more degraded than 1. This reflects that the model is more optimised for comb-filter like spectra at the moment and not for strongly low- or high-passed signals.

Verify the results

Now, its time to compare the results with the ones from the listening test. Note, that the listening test ratings were in the range -1..1 and we have to transfer them to 0..1:

```
figure; hold on;
% Plot listening test results as points with errorbars
errorbar(([humanLabels{:,2}]+1)./2, [humanLabels{:,3}]./2, 'og');
% Plot model predictions as line
plot(prediction, '-g');
axis([0 11 0 1]);
xlabel('System');
ylabel('Coloration');
set(gca, 'XTick', [1:10]);
set(gca, 'XTickLabel', ...
    {'ref', 'st.', '67cm', '34cm', '17cm', '8cm', '4cm', '2cm', '1cm', 'anch.'});
```

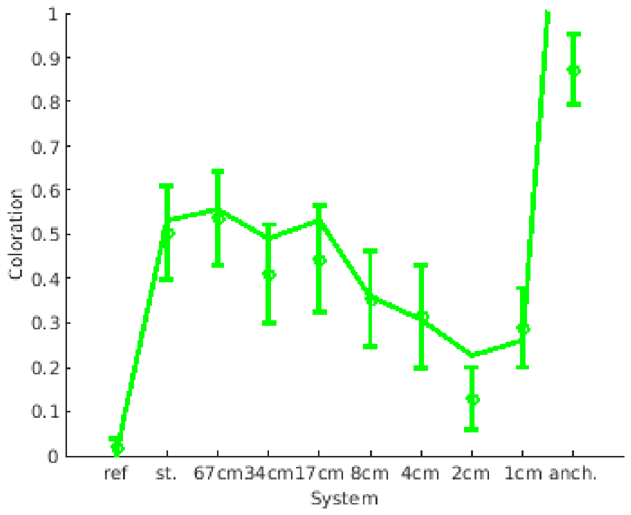


Fig. 66 Median of coloration ratings (points) together with confidence intervals and model prediction (line). The centimeter values describe the inter-loudspeaker distances for the investigated WFS systems.

To see the prediction results for all listening test data, go to the `examples/qoe_coloration` folder. There you will find the following functions that you can run, and that will pop up with a figure showing the results at the end:

```
colorationWfsCircularCenter
colorationWfsCircularOffcenter
colorationWfsLinearCenter
colorationWfsLinearOffcenter
```

Prediction of localisation in spatial audio systems

As we have seen in the example on Prediction of coloration in spatial audio systems WFS systems introduce errors at higher frequencies in the synthesized sound field that are perceivable as coloration compared to a reference sound field, that was the goal of the synthesis. Those errors do not exist at lower frequencies, which implies that localisation of the synthesised sound source deviates only slightly from performance in the reference sound field.

We did a huge set of listening test investigating localisation performance in WFS and NFC-HOA that are presented in 2013-11-01: Localisation of different source types in sound field synthesis.

In this example we choose the WFS system with a circular loudspeaker array and three different number of used loudspeakers synthesising a point source in front of the listener. The goal is to predict the perceived directions of that synthesised point source with the Two!Ears model.

Getting the listening test data

The experiment provides directly BRS files and a xml-file with the settings for the Binaural simulator. In the following we do an example run for one condition:

```
humanLabels = readHumanLabels(['experiments/2013-11-01_sfs_localisation/', ...
                               'human_label_localization_wfs_ps_circular.txt']);
brsFile = humanLabels{8,1};
```

If you look at the BRS file, you can easily decode the condition:

```
>> brsFile

brsFile =

experiments/2013-11-01_sfs_localisation/brs/wfs_nls14_X-0.50_Y0.75_src_ps_xs0.00_ys2.50.wav
```

This means that we had a WFS system with a circular loudspeaker array consisting of 14 loudspeakers synthesising a point source placed at (0.0, 2.5) m. The listener was placed at (-0.50, 0.75) m, which means slightly to the left and to the front inside the listening area.

Setting up the Binaural Simulator

Now, we start the Binaural simulator using the provided configuration file of the experiment and make some final adjustments like setting the length of the noise stimulus and rotate the head to the front as the localisation results will be relative to the head orientation:

```
sim = simulator.SimulatorConvexRoom(['experiments/', ...
   '2013-11-01_sfs_localisation/2013-11-01_sfs_localisation.xml']);
sim.Sources{1}.IRDataset = simulator.DirectionalIR(brsFile);
sim.LengthOfSimulation = 5;
sim.rotateHead(0, 'absolute');
sim.Init = true;
```

Estimating the localisation with the Blackboard

For the actual prediction of the perceived direction we use the DnnLocationKS knowledge source, limit its upper frequency range to 1400 Hz, and setup the Blackboard system for localisation without head rotations, see Localisation with and without head rotations for details:

```
bbs = BlackboardSystem(0);
bbs.setRobotConnect(sim);
bbs.buildFromXml('Blackboard.xml');
bbs.run();
```

Verify the results

Now the prediction has finished and we have to inspect the result and compare it to the result from the listening test. The prediction is stored inside the Blackboard system and can be requested with:

```
>> predictedAzimuths = bbs.blackboard.getData('perceivedAzimuths')

predictedAzimuths =

1x8 struct array with fields:

    sndTmIdx
    data
```

As the `perceivedAzimuths` is a data structure containing the results for every time step of the block-based processing, we provide a function that evaluates that data and provides us with an average value over time:

```
>> predictedAzimuth = evaluateLocalisationResults(predictedAzimuths)

predictedAzimuth =

  -14.8087
```

During the experiment a jitter was applied to the zero degree head orientation of the binaural synthesis system in order to have a larger spread of possible perceived directions. This jitter is stored as well in the results of the listening test and has to be added to the predicted azimuth:

```
headRotationOffset = humanLabels{8,9};
predictedAzimuth = predictedAzimuth + headRotationOffset;
perceivedAzimuth = humanLabels{8,4};
```

After that we can compare the result to the one from the listening test:

```
>> fprintf(1, ['\nPerceived direction: %.1f deg\n', ...
               'Predicted direction: %.1f deg\n'], ...
           perceivedAzimuth, predictedAzimuth);

Perceived direction: -14.5 deg
Predicted direction: -16.8 deg
```

If you would like to do this for all listening positions, you can go to the `examples/qoe_localisation` folder and execute the following command:

```
>> localisationWfsCircularPointSource
```

This will predict the directions for every listener position and plots the results at the end in comparison to the listening test results. Figure Fig. 67 shows the result.

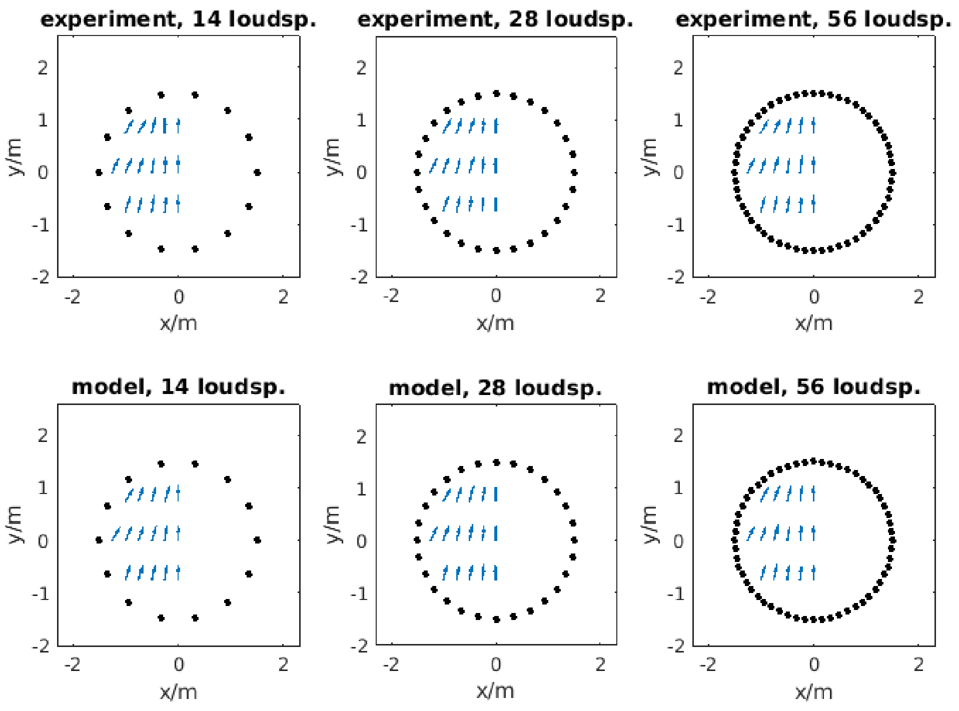


Fig. 67 Localisation results and model predictions. The black symbols indicate the loudspeakers. On every listening position an arrow is pointing into the direction the listener perceived the corresponding auditory event.

v: latest