# **AUTOMATED BATCH ANALYSIS AND CHARACTERISATION OF UNDERWATER SOUNDS** (ABACUS)

## User Guide

v1.0

*by Guillermo Jiménez Arranz*

21 May 2022

# Contents

# 1 Introduction

ABACUS is a script-based data processing software programmed in MATLAB for the analysis of underwater sounds and their spatial distribution. Alike the ancient counting device of the same name, ABACUS is a simple yet effective way of performing relatively complex calculations.

ABACUS is born from the need for a simple tool that can process large audio and navigation datasets for the generation of source-specific sound field maps. This need arises after the author's experience working on acoustic field characterisation (AFC) and sound source verification (SSV) projects. Software such as PAMGuard and Seiche's proprietary software SeicheSSV were initially used for recording and processing audio and navigation data. These programs are suitable for recording, but they do not incorporate robust tools for the automatic detection and processing of the short, low-energy transient signals that may be encountered in a typical AFC project.

ABACUS is inspired on the SPL Toolbox, a piece of software that the author developed at Seiche to process the sound field from target sound sources in complex soundscapes. With ABACUS, the idea was to produce a software similar to the SPL Toolbox but with focus on simplicity, stability and free access. *Simplicity* in its use and content: the GUI has been replaced with JSON configuration files, the code is organised into separate computing modules, and the number of user input parameters has been reduced to only the strictly necessary. *Stability* as to guarantee that the software is fully tested. And *free-access* to allow others to benefit from and contribute to this work.

# 2 General Features

The most relevant features of ABACUS are listed below. Details of these aspects of the software are given in the following chapters.

- User interface based on JSON configuration files.
- Intuitive batch-processing for large audio and navigation datasets.
- Modular structure with six main modules: audio import, audio processing, audio detection, receiver import, source import, and navigation processing.
- Efficient, low-memory solution for reading and resampling audio data. Supports WAV and RAW audio formats.
- Reads source and receiver position information from GPS, AIS and P190 files produced by various platforms (PAMGuard, SeicheSSV, Wildlife Acoustics SM4M).
- Position and resampled audio data is stored in *Navigation* and *Audio Databases* (.mat) for quick access and efficient processing.
- Various detection algorithms are available for the automatic detection of transient sounds (e.g. airguns, piling, sub-bottom profiler, sparker, etc). For the analysis of ambient noise and continuous soundscapes, audio files are divided into segments of equal duration.
- Pre-detection bandpass filtering for noise rejection and improved detection rates.
- Pre-processing DC-offset correction and bandpass filtering.
- Energy-based calculation of processing window's length and position.
- Processing of broadband and band metrics for each detected sound event and background noise segment preceding each detection (peak, peak-to-peak, RMS, exposure).
- Pre-filtering zero-padding approach to minimise energy loss on signals of short duration at low frequencies.
- Time offset correction for syncing detections (PC time) with navigation data (UTC time).
- Calculation of receiver and source navigation data, including *self-parameters* (latitude, longitude, course, speed) and *relative parameters* (source-to-receiver distance, source-to-receiver bearing and source directivity angle)
- Configuration and results produced by the six main modules for each audio file are stored in individual, lightweight *Acoustic Databases* (.mat) for later post-processing and representation. Each Acoustic database can include multiple receivers and sources.

# 3 Flow Chart

**Figure 1** shows the general flow diagram of ABACUS. Colours are used to distinguish between raw data (green), databases (orange), and software modules (blue).

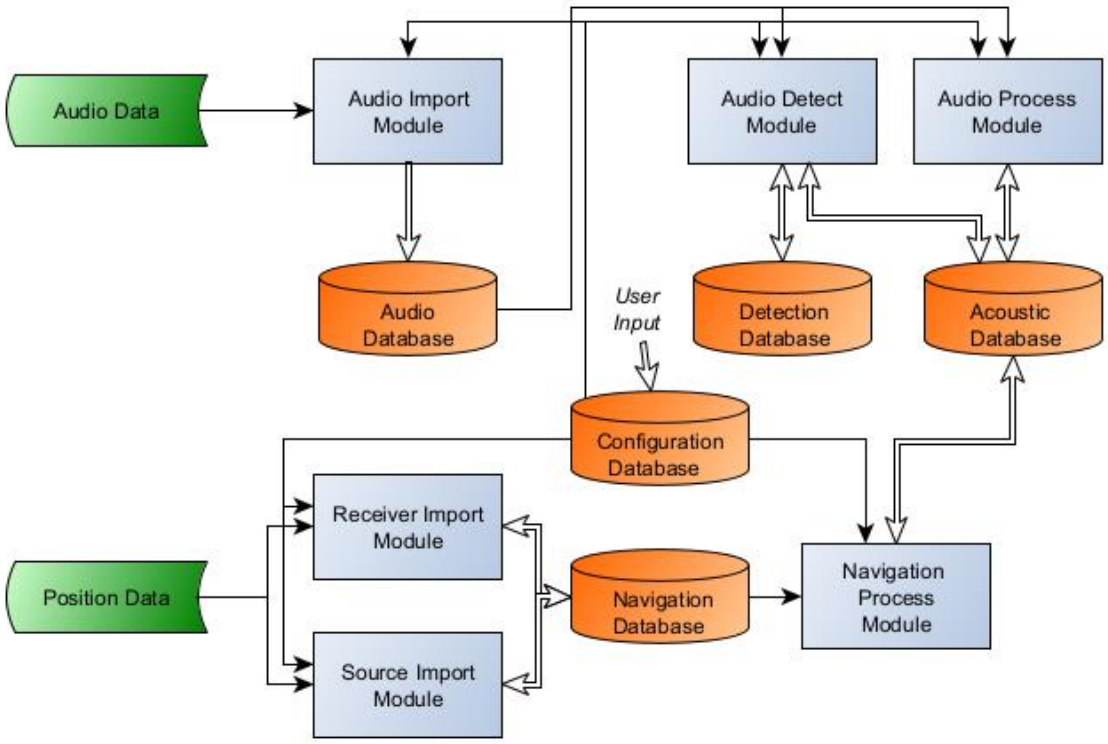


**Figure 1** Flow chart describing the connections between the data (*green*), processing modules (*blue*) and databases (*orange*) in ABACUS. Black arrows represent the exchange of information between parts of the software. White arrows represent changes in a database. Double arrows indicate a two-way information exchange.

# 4 Program Files and Execution

ABACUS root directory contains two files and two folders that are required for running the software: 'Dependencies (Core)', 'Dependencies (External)', *root.json* and *runAbacus.m*.

The folder 'Dependencies (Core)' contains all the MATLAB routines (.m) that are exclusive of ABACUS. The routines are organised in folders corresponding to the different modules. There are also folders dedicated to configuration templates, configuration examples, and general tools. For further details, see the README.txt file in 'Dependencies (Core)'.

The folder 'Dependencies (External)' contains a number of MATLAB packages developed by the author as standalone tools to be used in a wide range of applications. The packages include solutions for filtering (single filter and filter banks; digital and FFT-based), efficient reading and resampling of audio files, reading navigation data (GPS, AIS, P190), calculating geographic positions, distances and bearings, and covariance matrix estimation.

The *root.json* file is a JSON structure with the fields 'audio', 'position', and 'block', followed respectively by the parent directory of the audio files, position files, and databases. JSON is an open-standard file format widely used for the creation of configuration files. A typical MATLAB structure with multiple field-value pairs is easily defined in JSON within curly brackets (*{}*). The field names are enclosed in quotation marks (*""*), followed by colon (*:*), the value of the field, and a comma for separation. Strings representing paths must use double-backslash (\\) rather than single-backslash (\), a reserved special symbol in JSON. The user needs to update the parent directories manually and make sure that *root.json* is in the same directory as *runAbacus.m* for ABACUS to find the location of the audio, position and database files. Below is an example of the content of *root.json*.

```
{
"audio": "F:\\PROJECT_XXX\\Survey Data\\Audio"
"position": "F:\\PROJECT_XXX\\Survey Data\\Position"
"block": "C:\\PROJECT_XXX\\Buoy1"
}
```

**Figure 2** Example of the content in *root.json*. Every time the data or project folder change location, update the paths that accompany the fields "audio", "position" and "block".

The *runAbacus.m* is the file from which the software modules are executed. Each section is identified with % followed by the title. The user must run each section by selecting it and pressing F9. The sections are organised by order of execution. That order must be followed strictly. The only exceptions are the 'Import Receiver' and 'Import Source' modules, which can be run any time before the 'Process Navigation' module, or not run if the position data is not available or relevant.

The script starts with the *'*Add Libraries' and 'Storage Directory' sections, and then calls each software module. 'Add Libraries' adds the paths of all files in 'Dependencies (Core)' and 'Dependencies (External)' and their subfolders to the MATLAB's search path. The 'Storage Directory' section reads the parent directories in *root.json*.

Before running any of the modules in *runAbacus.m*, the user must prepare the configuration files. These files define the rules, parameters and files to process. There are three types of configuration files: global, module-specific and settings-specific. The first two types are compulsory. For details about the configuration files and how to populate them, see **Section 6.1**.

```
% Add Libraries
addpath(genpath(fullfile(pwd,'Dependencies (Core)')))
addpath(genpath(fullfile(pwd,'Dependencies (External)')))

% Storage Directory
root = jsondecode(fileread(fullfile(pwd,'root.json')));

% Import Audio
AudImpConfig = readAudioImportConfig(root);
audioImportFun(root,AudImpConfig)

% Import Receivers
RecImpConfig = readReceiverImportConfig(root);
receiverImportFun(root,RecImpConfig)

% Import Sources
SouImpConfig = readSourceImportConfig(root);
sourceImportFun(root,SouImpConfig)

% Detect Audio
updateAcousticDatabases(root)
AudDetConfig = readAudioDetectConfig(root);
audioDetectFun(root,AudDetConfig)

% Process Audio
updateAcousticDatabases(root)
AudProConfig = readAudioProcessConfig(root);
audioProcessFun(root,AudProConfig)

% Process Navigation
updateAcousticDatabases(root)
NavProConfig = readNavigationProcessConfig(root);
navigationProcessFun(root,NavProConfig)
```

**Figure 3** Content of runAbacus.m script.

# 5 Audio and Position Data

ABACUS makes use of two types of data: audio and position.

The *audio* is recorded from one or more sensors with an external data acquisition system. An audio file has as many channels as sensors and is generated by one platform (e.g., moored buoy). The software supports WAV and RAW formats. For the purpose of syncing the audio and navigation data, the name of the audio files must include the timestamp of the first sample. Any platform can be used to record the data, as long as the timestamp of the first sample is included in the name and the audio format is supported.

The timestamp format can take any form. Examples of formats are `'*yyyymmdd_HHMMSS_FFF'` for PAMGuard, `'yyyymmdd_HHMMSS_FFF*'` for SeicheSSV or `'*yyyymmdd_HHMMSS'` for Wildlife Acoustics SM4M.

The audio files can be stored anywhere within the "audio" parent directory included in *root.json* (ROOT.AUDIO). The selection of specific audio files for the 'Audio Import', 'Audio Detect', 'Audio Process', and 'Navigation Process' modules is done through the partial paths and folders included in *audioPaths.json*. These partial paths and folders are relative to ROOT.AUDIO.

The *position* data is generated by a navigation device and stored with a specific file format. The software supports GPS, AIS and P190 protocols, with some caveats. These protocols are standard and are followed by the corresponding navigation device. However, the format of the file to which the data from a navigation unit is exported is not standard and depends on the hardware and software that creates the file. ABACUS currently supports GPS files from SeicheSSV (.gpstext) and PAMGuard (.csv). For the AIS format, ABACUS only supports files from PAMGuard (.csv).

Note that ABACUS cannot access PAMGuard SQLite (.sqlite3) databases directly. The GPS and AIS data in the .sqlite3 database need to be exported as a .csv table. This is trivial operation that can be easily done with SQLiteStudio. Support for SQL databases may be added in a future release.

The P190 is a file format rather than a protocol, and is supported regardless of the platform. The P190 is an exchange data format specifically designed to store time, position and other information relevant in a seismic survey. P190 files are generated by specialised equipment in the seismic vessel.

The navigation data in the files may refer to the source or receiver themselves or to the position of the navigation unit, which will generally be away from the source or the receiver. For the sake of obtaining accurate sound field maps, the position data should always represent the location of the source or the receiver. In those cases where it doesn't, a relative horizontal offset can be applied (see **Chapter 10** and **11**).

The position files can be stored anywhere within the "position" parent directory included in *root.json* (`ROOT.POSITION`). The selection of specific position files for the 'Receiver Import' and 'Source Import' modules is done through partial paths and folders in the corresponding configuration files (*receiverImportConfig*.json*, *sourceImportConfig*.json*). These partial paths and folders are relative to `ROOT.POSITION`.

# 6 Processing Block

A *processing block* is the folder where all the databases for a particular analysis scenario are stored. There are five databases in a processing block: Configuration, Audio, Detection, Position, and Acoustic. These databases are stored in the folders 'configdb', 'audiodb', 'detectiondb', 'positiondb', and 'acousticdb' under parent directory ROOT.BLOCK, given in the *root.json* file. The name of the database folders cannot be changed. Except for 'configdb', which is created and populated by the user with the necessary configuration files, the rest of folders are generated automatically by the relevant modules.

The Configuration Database encompasses a number of scripts (.json) aimed at controlling what the software will process and how. Audio and Navigation Databases are large, intermediate files (.mat) with pre-processed data created to improve the efficiency of recurrent processing tasks. Acoustic Databases are lightweight files containing the results and configuration data from each module that has been run in ABACUS. These files are the primary output of the software. They contain all the information that is needed to analyse, represent and interpret the sound field from target acoustic sources.

For clarity, a processing block should be limited to a single recording platform. One platform comprises data from one or more sensors (channels) and sound sources (e.g., moored buoy with two hydrophones recording piling blows and ADD pings). Different processing blocks could also be created for specific periods to conveniently process data from long campaigns (e.g., Buoy1_April, Buoy1_May).

Details about each of these databases are given in the **Sections 6.1**, **6.2**, **6.3**, **6.4**, and **6.5**.

## 6.1 Configuration Database

The *configuration database* contains global, module-specific, and setting-specific configuration files. *Global* configuration files affect all modules, *module-specific* configuration files define essential processing parameters and metadata for each module, and *setting-specific* tables are required for particular options on some of the modules. All these files must be stored in directory '<ROOT.BLOCK>\configdb'. Files in subfolders are not read.

Details about these three types of configuration files are given in **Subsections 6.1.1**, **6.1.2**, and **6.1.3**.

### 6.1.1 Global

There are three global configuration files: *channelToReceiver.json, resampleRateToSource.json*, and *audioPaths.json*.

The first two are particularly important, as they define the relationship between channel and receiver, and resampling rate and source. Their importance lies in the fact that ABACUS uses unique source and receiver names to connect the results from all its modules. This approach simplifies the configuration process by taking away from the user the need to introduce the channel, resampling rate, source name and receiver name in each module-specific configuration file, which could lead to mistakes. **Figures Figure 4** and **Figure 5** are two examples of populated *channelToReceiver.json* and *resampleRateToSource.json* files.

```
{
"channel": [1,2],
"receiverName": ["Buoy1_H1","Buoy1_H2"]
}
```

**Figure 4** Example of the content in *channelToReceiver.json*.

```
{
"resampleRate": [ 24000, 96000],
"sourceName":   ["Airgun", "Sparker"]
}
```

**Figure 5** Example of the content in *resampleRateToSource.json*.

Nonetheless, defining these relationships requires some careful planning, since changing the content in *channelToReceiver.json* or *resampleRateToSource.json* will force the software to modify or remove elements in the Acoustic Databases. For example, let's assume that 'Buoy1_H1' and 'Buoy1_H2' are set as the receiver names linked to channels 1 and 2, but after some processing the user realises that Buoy 1 was actually Buoy 2; if the receiver names in *channelToReceiver.json* were to be changed to 'Buoy2_H1' and 'Buoy2_H2', the receiver names in the Acoustic Databases would be updated accordingly. A less desirable situation is having to modify the resampling rate of a source in file *resampleRateToSource.json*; since the results in the Acoustic Database for the source were obtained from an audio file with a different sampling rate, those results have to be automatically removed by the software for consistency. The best approach is to plan in advance how many sources and receivers will be processed, their names, the maximum frequency range of the sources, and the relation between receiver and audio channel, so as to avoid changes in the files *channelToReceiver.json* or *resampleRateToSource.json* at a later stage.

The *audioPaths.json* file contains a list of partial paths and folders pointing at the location of the audio files to be processed. These paths and folders are relative to the audio parent directory ROOT.AUDIO, from *root.json*. *audioPaths.json* is accessed by the modules 'Audio Import', 'Audio Detect', 'Audio Process', and 'Navigation Process' to determine what audio files, audio databases, or acoustic databases need to be processed. Audio and acoustic databases have the same name as the audio file they originate from, except for an appended string (audio databases only) and a different extension (.mat).

Listing the audio and audio-related files to process in a single file is a convenient solution. It avoids having to list within the configuration files of each module the audio files, audio databases or acoustic databases to process, which could lead to mistakes. What makes *audioPaths.json* an even more versatile selection tool is the option of skipping any of the listed paths and folders by simply prepending a supported "commenting" symbol (!, #, %) to the string (just after the opening quotation mark). This option allows the user to test the module's configuration parameters on only a few files before committing to batch-process the entire dataset.

*Partial folders* are an easy way of selecting a group of files stored in the same directory without having to explicitly give their names. However, *partial paths* offer more control by allowing individual files to be tested or skipped. **Figure 6** shows an example of an *audioPaths.json* file. Notice the exclamation mark (!) at the beginning of the last string telling the software not to process that file.

```
[
"RAW\\20140406_183314_4589586_7200000.raw2int16",
"RAW\\20140406_183544_4589586_7200000.raw2int16",
"!RAW\\20140406_183814_4589586_7200000.raw2int16",
]
```

**Figure 6** Example of the content in *audioPaths.json*.

### 6.1.2 Module-Specific

The main configuration parameters are defined in a series of module-specific configuration files. Each module uses one or more configuration file templates. A *template* is a configuration file with a specific set of parameters relevant for a particular configuration scenario. For example, the 'Audio Import' module has two templates, one per audio format (WAV, RAW), and 'Audio Detect' has as many templates as detection algorithms.

The configuration templates are stored in 'Dependencies (Core)\Config_Templates'. Some examples of populated templates are included in 'Dependencies (Core)\Config_Examples'. The content of the configuration templates for each module are described in the following sections.

All module-specific configuration files adopt the name of their respective modules, followed by an underscore (_) and a number indicating the order of the file in the processing queue. For example, configuration files for 'Audio Import' are named '*audioImportConfig_<NUM>.json'*. If the processing order is not relevant, '*_<NUM>*' can be omitted. All configuration files must start with '*<MODULE_NAME>Config*' for the software to be able to recognise them.

#### Audio Import

The configuration files for the 'Audio Import' module are named '*audioImportConfig*.json*'. There are two templates available, one per supported audio format (WAV, RAW). The user must create as many *audioImportConfig*.json* files as unique `channel-resamplingRate` combinations are to be processed.

The role of the 'Audio Import' module is to read, resample and store data from audio files. The idea behind it is to increase the efficiency of the frequent reading and processing of audio data. This is done in two ways: by storing the audio in a format (.mat) that MATLAB can access quickly, and by eliminating redundant information for the specific acoustic source through resampling. This module creates an Audio Database (.mat) file per selected audio file, channel and resampling rate, and stores it under '<ROOT.BLOCK>\audiodb'.

**Figure 7** shows an example of a populated *audioImportConfig*.json* template for the WAV format.

```
{
"audioFormat": "WAV",
"channel": 1,
"resampleRate": 96000
}
```

**Figure 7** Example of WAV template for the configuration file '*audioImportConfig*.json'*.

**Figure 8** shows an example of a populated *audioImportConfig*.json* template for the RAW format. This format does not include a header, which means that the user needs to provide the parameters for reading the audio file (i.e., sampling rate, bit resolution, number of channels, and endianness).

```
{
"audioFormat": "RAW",
"channel": 1,
"resampleRate": 48000,
"sampleRate": 48000,
"bitDepth": 16,
"numChannels": 2,
"endianness": "l"
}
```

**Figure 8** Example of RAW template for the configuration file '*audioImportConfig*.json'*.

For a description of the fields in the two JSON scripts, see **Sub-Section 13.1.1** in the Appendix.

### Audio Detect

The configuration files for the 'Audio Detect' module are named *audioDetectConfig*.json*. There are five templates available, one per detection algorithm (Moving Average, Constant Rate, Slice, Mirror, and Neyman-Pearson). The user must create as many *audioDetectConfig*.json* files as unique `receiverName-sourceName` combinations are to be processed.

The 'Audio Detect' module determines the start and end of any event of interest in an audio file. This module creates an Acoustic Database (.mat) file, if it does not already exist, and populates it with the configuration and results from the detection process. Acoustic Database files are stored under '<ROOT.BLOCK>\acousticdb'.

**Figure 9** shows an example of a populated *audioDetectConfig*.json* file for a `'MovingAverage'` detector. This type of detector divides the audio file into segments of `windowDuration` seconds and classifies as *detection* any segment with a root-mean-square (RMS) amplitude `threshold` times larger than the RMS amplitude of the previous segment. The detector also applies a band-pass filter before processing the RMS value to maximise the signal-to-noise ratio (SNR). The windows that delimit the detections are then centred by setting their start position `windowOffset` milliseconds before the maximum energy point within the window. The detector also returns the limits for the background noise windows of `windowDuration`, placed immediately before each detection window.

```
{
"receiverName": "Buoy1_H1",
"sourceName": "Airgun",
"detector": "MovingAverage",
"DetectParameters": {
       "windowDuration": 1000,
       "windowOffset": 500,
       "threshold": [],
       "cutoffFreqs": [20,10000]}
}
```

**Figure 9** Example of configuration file '*audioDetectConfig*.json*' for 'MovingAverage' detector.

**Figure 10** shows an example of a populated *audioDetectConfig*.json* file for a `'ConstantRate'` detector. This detector uses an external Pulse Table (.csv) named `filename` and stored in '<ROOT.BLOCK>\ configdb' to locate the sound events. The Pulse Table contains the relative time of the first pulse and the pulse interval for a number of audio files. In those cases where the firing rate of the source is either constant or expected to vary very little (e.g., sparker or sub-bottom profiler), a solution like this can be used to locate the detections given the time of the initial pulse, the pulse repetition interval, `windowDuration`, and `windowOffset`. This approach requires the user to find out the first two values for each file that needs to be analysed. This is not a convenient solution, but it can be extremely useful when a low-energy engineering source with a steady firing rate needs to be processed and a conventional detector is unable to find the events. The detector also returns the limits of background noise windows.

```
{
"receiverName": "Buoy1_H1",
"sourceName": "Airgun",
"detector": "ConstantRate",
"DetectParameters": {
       "windowDuration": 5000,
       "windowOffset": 2500,
       "fileName": "pulseTable.csv"}
}
```

**Figure 10** Example of configuration file '*audioDetectConfig*.json'* for 'ConstantRate' detector.

**Figure 11** shows an example of a populated *audioDetectConfig*.json* template for a `'Slice'` detector. This is not a "detector" strictly speaking, as it doesn't need to access the audio data to determine the position of the target events. The algorithm simply divides the audio file into segments of `windowDuration` milliseconds. This solution is suitable for the analysis of continuous sounds and soundscapes where there is no unique or dominant target source (e.g., ambient sound). The algorithm doesn't return the limits of background noise windows, since all segments are treated as target sound.

```
{
"receiverName": "Buoy1_H1",
"sourceName": "Airgun",
"detector": "Slice",
"DetectParameters": {
       "windowDuration": 1000}
}
```

**Figure 11** Example of configuration file '*audioDetectConfig*.json'* for 'Slice' detector.

**Figure 12** shows an example of a populated *audioDetectConfig*.json* template for a `'Mirror'` solution. This approach copies the detection information from `mirrorReceiver` into the current receiver. This is a convenient solution for comparing the detections from two or more nearby sensors at a later stage, as it ensures that the number of detections and their location is identical, and doesn't require running a detection algorithm. `mirrorReceiver` should point at the channel with highest SNR, as the detections computed from that channel will be the most reliable.

```
{
"receiverName": "Buoy1_H2",
"sourceName": "Airgun",
"mirrorReceiver": "Buoy1_H1"
}
```

**Figure 12** Example of configuration file '*audioDetectConfig*.json'* for mirror solution.

**Figure 13** shows an example of a populated *audioDetectConfig*.json* template for a detector `'NeymanPearson'`. This approach divides the audio file into segments of `kernelDuration` seconds and applies to each segment one of three available Neyman-Peason algorithms, specified by `detectorType`. The detections are then grouped into windows of `windowDuration` seconds, where `windowDuration` is an integer multiple of `kernelDuration`. The three available algorithms are the energy detector (`'ed'`), the estimator-correlator in white Gaussian noise (`'ecw'`) and the estimator-correlator in coloured Gaussian noise (`'ecc'`). For the analysis of each segment,

all three methods use a target probability of false alarm `rtpFalseAlarm` to determine the threshold $\gamma'$ from the probability function of false alarm $P_{FA}(\gamma')$. The `detectorSensitivity` can be used to artificially increase the threshold $\gamma'$ and thus help minimise the number of false alarms produced by high-level, non-target signals. A lower limit in the signal-to-noise ratio can also be defined to avoid classifying as "detection" any segment with a SNR below `minSnrLevel`. This is a useful parameter, as segments with a SNR < -10 dB are generally of no interest due to the large uncertainty that exits when processing their broadband and in-band energy. The SNR of the segments can be improved through *resampling* or *filtering*, by changing the sampling rate to `resampleRate` or by applying a filter with limit frequencies `cutoffFreqs`. These processing stages do not affect the audio database and are only temporarily applied to enhance the performance of the detector. The estimator-correlators rely on training data to build the covariance matrices that will be used for the computation of the performance curves $P_{FA}(\gamma')$ and test statistics $T(\mathbf{x})$. The training data for the target signal and noise is stored in directory `trainFolder`, under sub-folders 'signal' and 'noise'. The covariance `estimator` is used to compute the covariance matrix of the target signal (`detectorType='ecw'`), or the covariance matrices of the target signal and noise (`detectorType='ecc'`). There are nine covariance estimators available: 'oas', 'rblw', 'param1', 'param2', 'corr', 'diag', 'stock', 'looc', and 'sample'. A segment is classified as "detection" when the test statistic $T(\mathbf{x})$ is larger than the threshold $\gamma'(\mathbf{x})$.

```
{
"receiverName": "Buoy1_H2",
"sourceName": "Airgun",
"detector": "NeymanPearson",
"DetectParameters": {
       "detectorType": "ecc",
       "kernelDuration": 0.5,
       "windowDuration": 1,
       "windowOffset": 0.5,
       "rtpFalseAlarm": 0.001,
       "detectorSensitivity": 1,
       "minSnrLevel": -6,
       "cutoffFreqs": [20,8000],
       "trainFolder": "C:\\AIRGUN PULSES",
       "estimator": "sample",
       "resampleRate": 16000
       }
}
```

**Figure 13** Example of configuration file '*audioDetectConfig*.json*' for mirror solution.

For a description of the fields in the five JSON scripts, see **Sub-Section 13.1.2** in the Appendix.

### Audio Process

The configuration files for the 'Audio Process' module are named '*audioProcessConfig*.json*'. There is only one template available. The user must create as many *audioProcessConfig*.json* files as unique `receiverName-sourceName` combinations are to be processed.

The 'Audio Process' module computes the broadband and band acoustic metrics of the events detected by the 'Audio Detect' module. It also calculates the offset-corrected timestamps (UTC) of those events for them to be later synced with the navigation data. This module populates an existing Acoustic Database (.mat) file with the audio processing results and configuration information. Acoustic Database files are stored under '<ROOT.BLOCK>\acousticdb'.

**Figure 14** shows an example of a populated *audioProcessConfig*.json* template. The parameters `freqLimits` and `bandsPerOctave` define the frequency limits and bandwidth of the bandpass filters, from which the standard spectral bands to be processed are set. `audioTimeFormat` is the timestamp format used by the software to extract start PC times of the audio files. The `timeOffset` parameter indicates the difference between the PC and UTC times and is used for syncing audio and navigation data. If `timeOffset = []`, the module looks for a *timeOffset.csv* table with different time offsets for different PC times (see **Sub-Section 6.1.3** for details about this table). The `tags` are descriptive words that can be later assigned to audio segments or entire sections of an audio database.

```
{
"receiverName": "Buoy1_H1",
"sourceName": "Airgun",
"freqLimits": [20, 20000],
"bandsPerOctave": 3,
"cumEnergyRatio": 0.9,
"audioTimeFormat": "yyyymmdd_HHMMSS_FFF*",
"timeOffset": [],
"tags": ["full","rampup"]
}
```

**Figure 14** Example of configuration file '*audioProcessConfig*.json*'.

For a detailed description of the fields in the JSON script, see **Sub-Section 13.1.3** in the Appendix.

### Receiver Import

The configuration files for the 'Receiver Import' module are named '*receiverImportConfig*.json*'. There are two templates available, one per receiver category (`'fixed'`, `'towed'`). The user must create as many *receiverImportConfig*.json* files as unique receivers are to be processed.

The 'Receiver Import' module creates a Navigation Database file, if it doesn't exist, and populates it with information from the receivers. The type of receiver information included in the Navigation Database comprises metadata copied directly from the configuration file, and time-position data (see **Section 6.4** for further details). If the receiver is not fixed (`receiverCategory = 'towed'`), the module reads the receiver's time and position from navigation files (GPS, AIS, or P190). The Navigation Database file *navigationdb.mat* is stored under '<ROOT.BLOCK>\navigationdb'.

**Figure 15** shows an example of a populated *receiverImportConfig*.json* template for a `'fixed'` receiver category. This template is used for receivers that remain static and therefore their position can be set manually to a single fix (e.g., moored buoy).

```
{
"receiverCategory": "fixed",
"receiverName": "Buoy1_H1",
"latitude": 44.2032,
"longitude": -4.5368,
"depth": -2
}
```

**Figure 15** Example of configuration file '*receiverImportConfig*.json*' for 'fixed' receiver category.

**Figure 16** shows an example of a populated *receiverImportConfig*.json* template for a `'towed'` receiver category. This template is used for moving receivers (autonomous or towed) with time-position data sourced from navigation files (e.g., drift buoy, vessel-towed array).

```
{
"receiverCategory": "towed",
"receiverName": "Buoy1_H1",
"receiverOffset": [0,0],
"receiverOffsetMode": "soft",
"positionPaths": "GPS (SeicheSsv).gpstext",
"positionFormat": "GPS",
"positionPlatform": "SeicheSsv",
"vesselId": [],
"mmsi": [],
"depth": -2
}
```

**Figure 16** Example of configuration file '*receiverImportConfig*.json'* for 'towed' receiver category.

For a description of the fields in the two JSON scripts, see **Sub-Section 13.1.4** in the Appendix.

### Source Import

The configuration files for the 'Source Import' module are named '*sourceImportConfig*.json*'. There are four templates available, one per source category (`'fixed'`, `'towed'`, `'vessel'`, `'fleet'`). The user must create as many *sourceImportConfig*.json* files as unique sources are to be processed.

The 'Source Import' module creates a Navigation Database file, if it doesn't exist, and populates it with information from the source. The type of source information included in the Navigation Database comprises metadata copied directly from the configuration file, and time-position data (see **Section 6.4** for further details). If the source is not fixed (`sourceCategory = {'towed', 'vessel', 'fleet'}`), the module reads the source's time and position from navigation files (GPS, AIS, or P190). The Navigation Database file *navigationdb.mat* is stored under '<ROOT.BLOCK>\navigationdb'.

**Figure 17** shows an example of a populated *sourceImportConfig*.json* template for a `'fixed'` source category. This template is used for sources that remain static and therefore their position can be set manually to a single fix (e.g., impact piling).

```
{
"sourceCategory": "fixed",
"sourceName": "Airgun",
"latitude": 44.1890,
"longitude": -4.5045,
"depth": -2
}
```

**Figure 17** Example of configuration file '*sourceImportConfig*.json*' for 'fixed' source category.

**Figure 18** shows an example of a populated *sourceImportConfig*.json* template for a `'towed'` source category. This template is used for moving sources (autonomous or towed) with time-position data obtained from navigation files (e.g., vessel-towed seismic airgun array).

```
{
"sourceCategory": "towed",
"sourceName": "Airgun",
"sourceOffset": [0,0],
"sourceOffsetMode": "hard",
"positionPaths": "P190 (Seismic).p190",
"positionFormat": "P190",
"positionPlatform": "Seismic",
"vesselId": 1,
"sourceId": 1,
"mmsi": [],
"depth": -2
}
```

**Figure 18** Example of configuration file '*sourceImportConfig*.json'* for 'towed' source category.

**Figure 19** shows an example of a populated *sourceImportConfig*.json* template for a `'vessel'` source category. This template is used for individual vessels with time-position data obtained from navigation files. This category should be used when the vessel is the target source and not just the towing platform. The difference with `'towed'` source category with `sourceOffset = [0 0]` is that in this template the user can introduce the specifications of the vessel.

```
{
"sourceCategory": "vessel",
"sourceName": "Isabella Rose",
"positionPaths": "AIS (PAMGuard).csv",
"positionFormat": "AIS",
"positionPlatform": "PamGuard",
"vesselId": [],
"mmsi": 367614560,
"vesselName": "Isabella Rose",
"vesselLength": 45,
"vesselBeam": 11,
"vesselDraft": [],
"vesselGrossTonnage": []
}
```

**Figure 19** Example of configuration file '*sourceImportConfig*.json'* for 'vessel' source category.

**Figure 20** shows an example of a populated *sourceImportConfig*.json* template for a `'fleet'` source category. This template aims at importing the position of a large number of vessels from a navigation file with AIS data. Vessels added with this method are considered *secondary sources*. As opposed to a *target source*, a secondary source cannot be selected for audio processing, but its navigation data will be processed and synced with the detections from all target sources. Importing the position data from a vessel fleet is useful in two ways: it provides an image of the activity in the area at key moments, and it helps assess the proximity of individual vessels to target sources and their masking potential.

```
{
"sourceCategory": "fleet",
"sourceName": "fleet",
"positionPaths": "AIS (PamGuard).csv",
"positionFormat": "AIS",
"positionPlatform": "PamGuard"
}
```

**Figure 20** Example of configuration file '*sourceImportConfig*.json'* for 'fleet' source category.

For a description of the fields in the two JSON scripts, see **Sub-Section 13.1.5** in the Appendix.

Navigation Process

The configuration files for 'Navigation Process' module are named '*navigationProcessConfig*.json*'. There is only one template available. The user must create as many *navigationProcessConfig*.json* files as unique `receiverName-sourceName` combinations are to be processed.

The 'Navigation Process' module computes the navigation parameters of the receiver, target source and secondary sources at the time of the events detected by the 'Audio Detect' module for each unique `receiverName-sourceName` combination included in the configuration files. The processed navigation parameters comprise *self-values* for the source and the receiver (latitude, longitude, speed and course) and *relative-values* between source and receiver (distance, bearing, directivity angle).

The position data is treated with a time-averaging window (low-pass filter) of `smoothWindow` seconds, before calculating the navigation parameters at the UTC times of the detections using `interpMethod` interpolation. The navigation parameters of detections falling within periods longer than `maxTimeGap` seconds with no position information are ignored (set as `NaN`). The 'Navigation Process' module populates the Acoustic Database (.mat) file, from which it read the detection times, with the configuration and results from the navigation data processing. Acoustic Database files are stored under '<ROOT.BLOCK>\acousticdb'.

```
{
"receiverName": "Buoy1_H1",
"sourceName": "Airgun",
"smoothWindow": 20,
"maxTimeGap": 300,
"interpMethod": "linear"
}
```

**Figure 21** Example of configuration file '*navigationProcessConfig*.json*'.

For a description of the fields in the two JSON scripts, see **Sub-Section 13.1.6** in the Appendix.

### 6.1.3 Tables

There are currently three *.csv tables related to settings of specific modules: 'Pulse' table, for the `'ConstantRate'` detector in the 'Audio Detect' module; 'Time Offset' table for the variable PC time offset option in the 'Audio Process' module; and 'Vessel Database' table for the `'fleet'` source category in the 'Source Import' module.

**Figure 22** shows an example of a populated 'Pulse' table for the `'ConstantRate'` detector. This table is used by the `'ConstantRate'` detector to locate sound events emitted at a steady rate (see **Audio Detect** part in **Section 6.1.2** for details). The table starts with a one-line header with the names of the fields, followed by as many data lines as audio files to process. Each data line contains the values for each header name. The header must include the names `'FirstPulse_s'`, `'PulseInterval_ms'` and `'AudioName'`. Fields and values in each line are comma-separated. The order of the fields is not relevant, as long as the name and their values occupy the same column. The 'Pulse' table has to be populated manually and requires the user to inspect the audio files to process with an audio editor to extract the relevant parameters.

The field names are described as follows: `'FirstPulse_s'` is the time of the first pulse, in seconds, relative to the start of the audio file; `'PulseInterval_ms'` is the spacing, in milliseconds, between

consecutive pulses; `'AudioName'` is the name of the audio file, including extension, to which the first two fields refer to. The 'Pulse' table must be stored under '<ROOT.BLOCK>\configdb'. The user can give any name to the table in the *audioDetectConfig*.json* template for a `'ConstantRate'` detector, but for consistency and simplicity it is recommended to call it *pulseTable.csv*.

```
FirstPulse_s,PulseInterval_ms,AudioName
17.710,28680,20140406_183314_4589586_7200000.raw2int16
11.527,28660,20140406_183544_4589586_7200000.raw2int16
5.066,27500,20140406_183814_4589586_7200000.raw2int16
```

**Figure 22** Example of a *pulseTable.csv* file for a `'ConstantRate'` detector in the 'Audio Detect' module.

**Figure 23** shows an example of a populated 'Time Offset' table. This table is used by the 'Audio Detect' module to correct the difference between the detection timestamps, defined by the PC clock and sampling rate, and the UTC timestamps in which the navigation data is given. The table starts with a one-line header containing the field names, followed by data lines with the values for each field name. In general, it is sufficient to use a data line per audio file, unless the files are particularly large and a noticeable time offset is expected within them. The header must include the field names `'Timestamp_yyyymmddTHHMMSS'`, `'TimeOffset_s'`. Field names and values are separated by comma. The order of the fields is not relevant, as long as the name and their values occupy the same column. The 'Time Offset' table has to be populated manually and requires the user to find the time offset for different timestamps, information that can be generally obtained from GPS and AIS files.

The goal of the PC time offset correction is to sync the navigation data with the detected events. As opposed to a constant `timeOffset`, a 'Time Offset' table can compensate for time deviations from UTC that vary between audio files or within them. This solution is particularly important in large audio datasets from long campaigns, since conventional PC clock oscillators can introduce cumulative time errors of ~0.9 s per day (10µs/s). The 'Time Offset' table must be named *timeOffset.csv* and be stored under '<ROOT.BLOCK>\configdb'.

```
Timestamp_yyyymmddTHHMMSS,TimeOffset_s
20140406T183314,3590
20140406T183544,3590
20140406T183814,3590
```

**Figure 23** Example of a *timeOffset.csv* file for variable PC time offset in the 'Audio Process' module.

**Figure 24** shows an example of a populated 'Vessel Database' table. This table provides the specifications of a list of relevant vessels. When the source category `'fleet'` is selected in a *sourceImportConfig*.json* file, the 'Source Import' module reads the time and position data for a number of vessels from an AIS navigation file(s). The 'Vessel Database' determines what vessels to read from the AIS file(s) and provides the specifications for each of them. The table starts with a one-line header with the names of the fields, followed by data lines containing the values for each field name, with as many data lines as vessels. The header must include the names `'mmsi'`, `'vesselName'`, `'vesselLength'`, `'vesselBeam'`, `'vesselDraft'`, `'vesselGrossTonnage'`, separated by comma and not necessarily placed in this same order. The 'Vessel Database' table has to be populated manually and requires the user to find vessel specifications through its unique MMSI number (e.g., through online resources such as www.marinetraffic.com).

All vessels read as part of a `'fleet'` source category are treated by ABACUS as *secondary* sources, that is, non-localised sources that are not to be detected in the audio files and processed, but whose

position is relevant for sound field interpretation. A 'Vessel Database' contains information that is useful for analysing the acoustic masking potential a vessel that may be close to the target source. This type of table is a useful addition in a context where the target source may be affected by intense surrounding vessel activity. However, creating a 'Vessel Database' is not strictly necessary. The 'Vessel Database' must be named *vesseldb.csv* and stored in '<ROOT.BLOCK>\ configdb'.

```
mmsi,vesselName,vesselLength,vesselBeam,vesselDraft,vesselGrossTonnage
218582000,MSC Charleston,324.9,42.8,,89954
236483000,Sanco Star,80,16,,3953
239962000,Makronissos,244,42,,57062
259645000,Artemis Arctic,74.3,18,,3947
```

**Figure 24** Example of a *vesseldb.csv* file for `'fleet'` source category in the 'Source Import' module.

## 6.2 Audio Database

An Audio Database is a *.mat file containing the specifications and resampled data from the selected channel of an audio file stored in ROOT.AUDIO and listed in *audioPaths.json*. The Audio Database is created by the 'Audio Import' module for the selected audio file, `channel` and `resampleRate`, and saved in directory '<ROOT.BLOCK>\audiodb'. All Audio Database files take the name of the audio file they originate from, followed by the channel and the sampling rate after resampling, such as in '<AUDIO_NAME>_ch<CHANNEL>_fr<RESAMPLE_RATE>.mat'.

Reading and processing audio files in MATLAB can be particularly slow (~1.4 MB/s with an Intel Core i5-4300U @ 1.9 GHz). By resampling data at a lower sampling rate, sufficient to characterise the target source, and by saving that data in a format that MATLAB can access quickly (.mat), Acoustic Databases files can dramatically increase the speed of the frequent reading and processing tasks that the 'Audio Detect', 'Audio Process' and 'Revision' modules need to perform on the audio data. The 'Audio Import' process may take long, but it only has to be done once.

In terms of its content, an Audio Database is a MATLAB structure comprising two substructures: `AudImpConfig` and `AudImpData`. The content of these two substructures is discussed in Sub-**Sections 6.2.1** and **6.2.2**.

```
AudioDatabase = struct('AudImpConfig',AudImpConfig,'AudImpData',AudImpData);
```

**Figure 25** Initialised `AudioDatabase` structure.

### 6.2.1 AudImpConfig

`AudImpConfig` is a one-element substructure containing the audio file specifications, obtained directly from the *audioImportConfig*.json* file of name `configFileName`. For further details about its fields see **Sub-Section 13.2.1** in the Appendix.

```
AudImpConfig = struct('inputStatus',[],'configFileName',[],...
  'audioFormat',[],'channel',[],'resampleRate',[],'sampleRate',[],...
  'bitDepth',[],'numChannels',[],'endianness',[],'audioLength',[]);
```

**Figure 26** Initialised `AudImpConfig` substructure from an Audio Database.

### 6.2.2 AudImpData

`AudImpData` is a one-element substructure containing the resampled audio data and the path of its audio file. For further details see **Sub-Section 13.2.2** in the Appendix.

```
AudImpData = struct('audioPath',[],'audioData',[]);
```

**Figure 27** Initialised `AudImpData` substructure from an Audio Database.

## 6.3 Detection Database

The *detection database* folder is created only for the `'NeymanPearson'` detector. The database contains four types of *.mat files: raw scores, covariance, performance data, and eigendata. The raw scores, covariance, and eigendata files are only generated for the estimator-correlators (`detectorType = {'ecw','ecc'}`). The detection database files are generated by the 'Audio Detect' module for each combination of selected Neyman-Pearson detector `detectorType`, covariance `estimator`, detection sampling rate `resampleRate`, and window duration `kernelDuration`. All files are saved in directory '<ROOT.BLOCK>\detectiondb'.

All the detection database files follow the naming convention '<FILETYPE>_ <SOURCE_NAME>_ <DETECTORTYPE> _FA<FA>_FB<FB>_FS<FS>_T<T>_<ESTIMATOR>.mat', where <FILETYPE> is the type of detection database file ('CovarianceNoise', 'CovarianceSignal', 'PerformanceData'), 'EigenData'), <DETECTORTYPE> is the type of NP detector (`detectorType`), <FA> and <FB> are the bottom and top cutoff frequencies of the detection filter (`cutoffFreqs`), <FS> is the sampling rate for detection (`resampleRate`), and <T> is the processing window duration (`kernelDuration`).

The content of the detection database files is described in the following sub-sections.

### 6.3.1 RawScoreData

`RawSignal` and `RawNoise` are a one-element structures containing the raw scores matrices of the target signal and noise and additional information such as the duration of the processing window, the new sampling rate, the SNR from each observation, and the minimum SNR to include. The `RawScoreData` files are generated exclusively for the estimator-correlators.

For further details about the fields see **Sub-Section 13.3.1** in the Appendix.

```
RawScoreData = struct('kernelDuration',[],'sampleRate',[],...
    'minSnrLevel',[],'snrLevels',[],'rawScoreMatrix',[]);
```

**Figure 28** Initialised `RawScoreData` structure from Detection Database.

### 6.3.2 CovarianceData

`CovarianceSignal` and `CovarianceNoise` are one-element structures containing the covariance matrices of the target signal and noise and additional information such as the duration of the processing window and the sampling rate for detection. The `CovarianceData` files are generated exclusively for the estimator-correlators.

For further details about the fields see **Sub-Section 13.3.2** in the Appendix.

```
CovarianceData = struct('kernelDuration',[],'sampleRate',[],...
    'covarianceMatrix',[]);
```

**Figure 29** Initialised `CovarianceData` structure from Detection Database.

### 6.3.3 PerformanceData

`PerformanceData` is a is a multi-element structure where each element contains the probability density functions (PDF) of the null and alternative hypotheses $p_{0,1}(T)$ and the probability functions (PF) of false alarm and detection $P_{FA,D}(T)$ for one of 101 SNR values ranging from -50 dB to 50 dB, for one of three possible Neyman-Pearson detectors. Each element also contains additional information such as SNR, signal and noise variance, type of detector, number of variables, and filter cutoff frequencies.

For further details about the fields see **Sub-Section 13.3.3** in the Appendix.

```
PerformanceData = struct('detectorType',[],'signalVariance',[],...
    'noiseVariance',[],'snrLevel',[],'nVariables',[],'cutoffFreqns',[],...
    'axisFalseAlarm',[],'axisDetection',[],'pdfFalseAlarm',[],...
    'pdfDetection',[],'rtpFalseAlarm',[],'rtpDetection',[]);
```

**Figure 30** Initialised `PerformanceData` structure from Detection Database.

### 6.3.4 EigenData

The *eigendata* file is a one-element structure containing the eigenvalues and eigenvectors of the target signal (for `detectorType = 'ecw'`), or the eigenvalues and eigenvectors of the target noise and the *pre-whitened* target signal (for `detectorType = 'ecc'`). They also include additional information such as duration of the processing window, sample rate for detection, and type of noise (*white* for `'ecw'` or *coloured* for `'ecc'`). The `EigenData` files are generated exclusively for the estimator-correlators.

For further details about the fields see **Sub-Section 13.3.4** in the Appendix.

```
EigenData = struct('kernelDuration',[],'sampleRate',[],'noiseType',[],...
    'signalEigenVectors',[],'signalEigenValuesNorm',[],...
    'noiseEigenVectors',[],'noiseEigenValuesNorm',[]);
```

**Figure 31** Initialised `EigenData` structure from Detection Database.

## 6.4 Navigation Database

A Navigation Database is a *.mat file containing time, position, and general information from multiple receivers and sources. The position data is extracted from navigation files listed in the variable `positionPaths` in the *receiverImportConfig*.json* and *sourceImportConfig*.json* files and stored in `ROOT.POSITION`. The Navigation Database is a single file created and populated by the 'Receiver Import' and 'Source Import' modules and saved in directory '<ROOT.BLOCK>\navigationdb' with the name *navigationdb.mat*.

A Navigation Database is a convenient way of keeping all the source and receiver information in one place. It also gives the user the opportunity to look at the raw position data before syncing it with the

audio detections. Creating a Navigation Database is not strictly necessary, since there will be cases when the position data is either not available or not relevant (e.g., single static source, or ambient noise analysis of multiple non-localised sources).

In terms of its content, a Navigation Database is a MATLAB structure comprising three general variables (`receiverList`, `sourceList`, `vesselList`) and six substructures (`RecImpConfig`, `RecImpData`, `SouImpConfig`, `SouImpData`, `VesImpConfig`, `VesImpData`). The general variables are cell arrays of character vectors containing the names of the receivers, sources and vessels stored in the substructures. The content of the substructures is discussed in **Sub-Sections 6.4.1**, **6.4.2**, **6.4.3**, **6.4.4**, **6.4.5**, and **6.4.6**.

```
NavigationDatabase = struct(...
  'receiverList',[],'sourceList',[],'vesselList',[],...
  'RecImpConfig',RecImpConfig,'RecImpData',RecImpData,...
  'SouImpConfig',SouImpConfig,'SouImpData',SouImpData,...
  'VesImpConfig',VesImpConfig,'VesImpData',VesImpData);
```

**Figure 32** Initialised `NavigationDatabase` structure.

### 6.4.1 RecImpConfig

`RecImpConfig` is a multi-element substructure containing the configuration information for one or more receivers, obtained directly from the *receiverImportConfig*.json* files.

For further details about its fields see **Sub-Section 13.4.2** in the Appendix.

```
RecImpConfig = struct('inputStatus',[],'configFileName',[],...
  'receiverCategory',[],'receiverName',[],'receiverOffset',[],...
  'receiverOffsetMode',[],'positionPaths',[],'positionFormat',[],...
  'positionPlatform',[],'vesselId',[],'mmsi',[],'latitude',[],...
  'longitude',[],'depth',[]);
```

**Figure 33** Initialised `RecImpConfig` substructure from Navigation Database.

### 6.4.2 RecImpData

`RecImpData` is a multi-element substructure containing the time and position data for the receivers. Same-index elements in `RecImpConfig` and `RecImpData` correspond to the same receiver.

For further details see **Sub-Section 13.2.113.4.3** in the Appendix.

```
RecImpData = struct('positionPaths',[],'pcTick',[],'utcTick',[],...
  'latitude',[],'longitude',[],'depth',[]);
```

**Figure 34** Initialised `RecImpData` substructure from Navigation Database.

### 6.4.3 SouImpConfig

`SouImpConfig` is a multi-element substructure containing the configuration information for one or more sources, obtained directly from the *sourceImportConfig*.json* files.

For further details about its fields see **Sub-Section 13.4.4** in the Appendix.

```
SouImpConfig = struct('inputStatus',[],'configFileName',[],...
  'sourceCategory',[],'sourceName',[],'sourceOffset',[],...
  'sourceOffsetMode',[],'positionPaths',[],'positionFormat',[],...
  'positionPlatform',[],'vesselId',[],'sourceId',[],'mmsi',[],...
  'vesselName',[],'vesselLength',[],'vesselBeam',[],'vesselDraft',[],...
  'vesselGrossTonnage',[],'latitude',[],'longitude',[],'depth',[]);
```

**Figure 35** Initialised `SouImpConfig` substructure from Navigation Database.

### 6.4.4 SouImpData

`SouImpData` is a multi-element substructure containing the time and position data of the sources. Same-index elements in `SouImpConfig` and `SouImpData` correspond to the same source.

For further details see **Sub-Section 13.4.5** in the Appendix.

```
SouImpData = struct('positionPaths',[],'pcTick',[],'utcTick',[],...
  'latitude',[],'longitude',[],'depth',[]);
```

**Figure 36** Initialised `SouImpData` substructure from Navigation Database.

### 6.4.5 VesImpConfig

`VesImpConfig` is a multi-element substructure containing the configuration information for one or more vessels, obtained from a *sourceImportConfig*.json* file of `sourceCategory = 'fleet '`. The field names are identical to those in `SouImpConfig`. The number of elements is equal to either the number of unique vessels in the AIS navigation file, or to the number of valid vessels included in the 'Vessel Database' file *vesseldb.csv*, stored in '<ROOT.BLOCK>\configdb'.

For further details about its fields see **Sub-Section 13.4.4** in the Appendix.

```
VesImpConfig = struct('inputStatus',[],'configFileName',[],...
  'sourceCategory',[],'sourceName',[],'sourceOffset',[],...
  'sourceOffsetMode',[],'positionPaths',[],'positionFormat',[],...
  'positionPlatform',[],'vesselId',[],'sourceId',[],'mmsi',[],...
  'vesselName',[],'vesselLength',[],'vesselBeam',[],'vesselDraft',[],...
  'vesselGrossTonnage',[],'latitude',[],'longitude',[],'depth',[]);
```

**Figure 37** Initialised `VesImpConfig` substructure from Navigation Database.

### 6.4.6 VesImpData

`VesImpData` is a multi-element substructure containing the time and position data for the vessels. Same-index elements in `VesImpConfig` and `VesImpData` correspond to the same vessel. The field names are identical to those in `SouImpData`.

For further details see **Sub-Section 13.2.113.4.5** in Appendix.

```
VesImpData = struct('positionPaths',[],'pcTick',[],'utcTick',[],...
  'latitude',[],'longitude',[],'depth',[]);
```

**Figure 38** Initialised `VesImpData` substructure from Navigation Database.

## 6.5 Acoustic Database

An Acoustic Database is a lightweight *.mat file containing the configuration information and results from each module run in ABACUS. These files have all the information that is needed to analyse, represent and interpret the sound field from target acoustic sources.

One Acoustic Database is created for each selected audio file, `receiverName` and `sourceName` when the 'Audio Detect' module is run the first time. The database is saved in directory '<ROOT.BLOCK>\acousticdb'. The 'Audio Detect', 'Audio Process', and 'Navigation Process' modules read from and populate the appropriate Acoustic Database files with their respective outputs and configuration data. Users do not need to select the Acoustic Databases directly, instead they list the audio files to be processed in *audioPaths.json*, and ABACUS finds the corresponding Acoustic Databases to read from and update. All Acoustic Database files take the name of the audio file they originate from.

In terms of its content, an Acoustic Database is a MATLAB structure comprising two substructures (`AcoConfig`, `AcoData`). Their content is discussed in **Sub-Sections 6.5.1** and **6.5.2**.

```
AcousticDatabase = struct('AcoConfig',AcoConfig,'AcoData',AcoData);
```

**Figure 39** Initialised `AcousticDatabase` structure.

### 6.5.1 AcoConfig

`AcoConfig` is a multi-element structure with as many elements as `receiverName-sourceName` combinations. Each element in `AcoConfig` contains five general variables (`audiodbName`, `channel`, `resampleRate`, `receiverName`, `sourceName`) and seven substructures (`AudImpConfig`, `RecImpConfig`, `SouImpConfig`, `VesImpConfig`, `AudDetConfig`, `AudProConfig`, `NavProConfig`).

For a description of the general variables, see **General Fields** in **Sub-Section 13.5.1**. The content of the substructures is briefly discussed in the next points.

```
AcoConfig = struct('audiodbName',[],'channel',[],'resampleRate',[],...
    'receiverName',[],'sourceName',[],'AudImpConfig',AudImpConfig,...
    'RecImpConfig',RecImpConfig,'SouImpConfig',SouImpConfig,...
    'VesImpConfig',VesImpConfig,'AudDetConfig',AudDetConfig',...
    'AudProConfig',AudProConfig,'NavProConfig',NavProConfig);
```

**Figure 40** Initialised `AcoConfig` substructure from Acoustic Database.

#### AudImpConfig

`AcoConfig(m).AudImpConfig` is a one-element substructure containing the 'Audio Import' configuration information for the `receiverName-sourceName` combination in the `m`-element of the Acoustic Database. `AudImpConfig` is obtained directly from the corresponding Audio Database. For further details about its fields see **AudImpConfig** in **Sub-Section 13.5.1** in the Appendix.

#### RecImpConfig

`AcoConfig(m).RecImpConfig` is a one-element substructure containing the 'Receiver Import' configuration information for the `receiverName` in the `m`-element of the Acoustic Database. `RecImpConfig` is obtained directly from the matching receiver in the Navigation Database. For further details about its fields see **inputStatus**: TRUE if the audio import configuration is valid.

- `configFileName`: name of the audio import configuration file from which `AudImpConfig` comes from.
- `audioLength`: number of samples in the audio file.

See **Sub-Section 13.1.1** for a description of the rest of fields in `AudImpConfig`.

RecImpConfig in **Sub-Section 13.5.1** in the Appendix.

SouImpConfig

`AcoConfig(m).SouImpConfig` is a one-element substructure containing the 'Source Import' configuration information for the `sourceName` in the `m`-element of the Acoustic Database. `SouImpConfig` is obtained directly from the matching source in the Navigation Database. For further details about its fields see **inputStatus**: `true` if the audio import configuration is valid.

- `configFileName`: name of the receiver import configuration file '*receiverImportConfig*.json*' from which `RecImpConfig` comes from.

See **Sub-Section 13.1.4** for a description of the rest of fields in `RecImpConfig`.

SouImpConfig in **Sub-Section 13.5.1** in the Appendix.

VesImpConfig

`AcoConfig(m).VesImpConfig` is a multi-element substructure containing the 'Source Import' configuration information for all the vessels imported with the 'Source Import' module as part of a `'fleet'` source category. `VesImpConfig` is a direct copy of the structure of the same name stored in the Navigation Database. For further details about its fields see **inputStatus**: `true` if the audio import configuration is valid.

- `configFileName`: name of the receiver import configuration file '*sourceImportConfig*.json*' from which `SouImpConfig` comes from.

See **Sub-Section 13.1.5** for a description of the rest of fields in `SouImpConfig`.

VesImpConfig in **Sub-Section 13.5.1** in the Appendix.

AudDetConfig

`AcoConfig(m).AudDetConfig` is a one-element substructure containing the 'Audio Detect' configuration information for the `receiverName-sourceName` combination in the `m`-element of the Acoustic Database. `AudDetConfig` is obtained directly from an *audioDetectConfig*.json* file. For further details about its fields see **inputStatus**: `true` if the audio import configuration is valid.

- `configFileName`: name of the receiver import configuration file '*sourceImportConfig*.json*' from which `VesImpConfig` comes from.

See **Sub-Section 13.1.5** for a description of the rest of fields in `VesImpConfig`.

AudDetConfig in **Sub-Section 13.5.1** in the Appendix.

AudProConfig

`AcoConfig(m).AudProConfig` is a one-element substructure containing the 'Audio Process' configuration information for the `receiverName-sourceName` combination in the `m`-element of the

Acoustic Database. `AudProConfig` is obtained directly from an *audioProcessConfig*.json* file. For further details about its fields see **AudProConfig** in **Sub-Section 13.5.1** in the Appendix.

NavProConfig

`AcoConfig(m).NavProConfig` is a one-element substructure containing the 'Navigation Process' configuration information for the `receiverName-sourceName` combination in the `m`-element of the Acoustic Database. `NavProConfig` is obtained directly from a *navigationProcessConfig*.json* file. For further details about its fields see **NavProConfig** in **Sub-Section 13.5.1** in the Appendix.

### 6.5.2 AcoData

`AcoData` is a multi-element structure with as many elements as `receiverName-sourceName` combinations. Each element in `AcoData` contains five general variables (`audiodbName`, `channel`, `resampleRate`, `receiverName`, `sourceName`) and seven substructures (`AudDetData`, `AudProData` `RecProData`, `SouProData`, `VesProData`, `RevData`).

For a description of the general variables, see **General Fields** in **Sub-Section 13.5.2**. The content of the substructures is briefly discussed in the next points.

```
AcoData = struct('audiodbName',[],'channel',[],'resampleRate',[],...
    'receiverName',[],'sourceName',[],'AudDetData',AudDetData,...
    'AudProData',AudProData,'RecProData',RecProData,...
    'SouProData',SouProData,'VesProData',VesProData,'RevData',RevData);
```

**Figure 41** Initialised `AcoData` substructure from Acoustic Database.

AudDetData

`AcoData(m).AudDetData` is a one-element substructure containing the relative start and end times of the detection and background noise windows within the Audio Database `audiodbName` for the `receiverName-sourceName` combination in the `m`-element of the Acoustic Database. For further details about its fields see **AudDetData** in **Sub-Section 13.5.2** in the Appendix.

AudProData

`AcoData(m).AudProData` is a one-element substructure containing the relative times, offset-corrected timestamps and processed acoustic metrics for each detection and background noise segment in `AcoData(m).AudDetData`, for the `receiverName-sourceName` combination in the `m`-element of the Acoustic Database. For further details about its fields see **AudProData** in **Sub-Section 13.5.2** in the Appendix.

RecProData

`AcoData(m).RecProData` is a one-element substructure containing the timestamps and *self* navigation parameters (latitude, longitude, speed, course) of the receiver in the `m`-element of the Acoustic Database for each detection in `AcoData(m).AudDetData`. For further details about its fields see **RecProData** in **Sub-Section 13.5.2** in the Appendix.

SouProData

`AcoData(m).SouProData` is a one-element substructure containing the timestamps, *self* navigation parameters (latitude, longitude, speed, course) and *relative* navigation parameters (source to receiver

distance, bearing and source directivity angle) of the source in the `m`-element of the Acoustic Database for each detection in `AcoData(m).AudDetData`. For further details about its fields see **SouProData** in **Sub-Section 13.5.2** in the Appendix.

### VesProData

`AcoData(m).VesProData` is a multi-element substructure containing the timestamps, *self* navigation parameters (latitude, longitude, speed, course) and *relative* navigation parameters (vessel-receiver distance, bearing and vessel directivity angle) of all selected `'fleet'` vessels for each detection in `AcoData(m).AudDetData`. The relative navigation parameters are referred to the receiver in `m`-element of the Acoustic Database. For further details about its fields see **VesProData** in **Sub-Section 13.5.2** in the Appendix.

### RevData

`AcoData(m).RevData` is a one-element substructure containing a list of the true and false detections `AcoData(m).AudDetData` and settings of the Revision Module for the `receiverName-sourceName` combination in the `m`-element of the Acoustic Database. The fields in this substructure are deemed to be updated in a future software update when the Revision Module is implemented. For further details about its fields see **RevData** in **Sub-Section 13.5.2** in the Appendix.

## 6.6 Update Acoustic Databases

`updateAcousticDatabases` reads the *channelToReceiver.json* and *resampleRateToSource.json* files in '<ROOT.BLOCK>\configdb' and verifies that the Acoustic Datababases in '<ROOT.BLOCK>\acousticdb' show the same `channel/receiverName` and `resampleRate/sourceName` relationships as defined in the two JSON files. Differences in those relationships may result in the modification or deletion of elements in the Acoustic Databases. For details about the effects that particular changes to the JSON files will have on the Acoustic Databases, see the `help` in `updateAcousticDatabases`.

# 7 Audio Import Module

'Audio Import' is the first module run in ABACUS. The module reads, resamples and stores the audio data that will later be used by the 'Audio Detect' and 'Audio Process' modules.

**Figure 42** shows the flow chart of 'Audio Import'. The module calls `readAudioImportConfig` and `audioImportFun`, described in **Sections 7.1** and **7.2**. It also calls the Audio Filtering and Audio Resampling packages, stored in the 'Dependencies (External)' folder and briefly described in **Sub-Sections 7.3.1** and **7.3.2**.

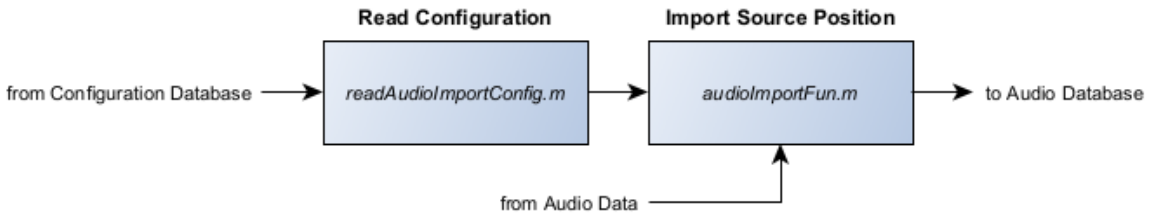


**Figure 42** Flow diagram of 'Audio Import' module.

## 7.1 Read Configuration Files

The *audioImportConfig*.json* files, stored in directory '<ROOT.BLOCK>\configdb', are read with `readAudioImportConfig`. Reading the configuration files is the first step in every module, where the information from the User Interface (manually-populated JSON scripts) is read, verified, and organised for the software routines to perform the appropriate actions.

`readAudioImportConfig` reads the configuration scripts in the order specified by the '_<NUM>' string appended to the file name. Each configuration script goes through a four-step process:

1. The content of the file is read with MATLAB's `jsondecode`, which creates a structure `AudImpConfigFile` containing the parameters of the specific configuration file;
2. `updateAudioImportConfig` verifies the input parameters in `AudImpConfigFile` one-by-one and assigns them to a single-element configuration structure `AudImpConfigOne`.
3. `verifyAudioImportConfig` looks for any non-supported parameter combination and any individual parameter in `AudImpConfigOne` that `updateAudioImportConfig` flagged as non-valid. The function then sets the flag `inputStatus` in `AudImpConfigOne` to `false` if any non-valid parameter or combination of them is found in `AudImpConfigOne`.
4. The process is repeated with all *audioImportConfig*.json* files. A multi-element structure `AudImpConfig` is generated from all the individual `AudImpConfigOne` structures.
5. `readAudioImportConfig` returns the verified multi-element `AudImpConfig` structure.

`AudImpConfig` contains as many elements as *audioImportConfig*.json* scripts are found in the Configuration Database folder and has the same fields regardless of the configuration template (see **AudImpConfig** in **Sub-Section 6.5.1**). The aim of `inputStatus` is to indicate that a mistake has been made in the configuration file `configFileName`. Elements in `AudImpConfig` where `inputStatus = false` are ignored by `audioImportFun`.

## 7.2 Import and Resample Audio Data

`audioImportFun` reads, resamples and stores the data from the audio files listed in *audioPaths.json*. The reading and resampling is done according to the configuration parameters in each element of

`AudImpConfig`. The data and configuration information from each audio file is stored on a *.mat Audio Database under '<ROOT.BLOCK>\audiodb'. One Audio Database file is created for each selected audio file and `channel-resampleRate` combination in `AudImpConfig`.

The function loops through all elements in `AudImpConfig` and audio files in *audioPaths.json*. Any element in `AudImpConfig` with an `inputStatus = false` is ignored. If any Audio Database in '<ROOT.BLOCK>\audiodb' matches the name of an audio file in *audioPaths.json* and the `channel-resampleRate` from any element of `AudImpConfig`, that audio file is ignored before any processing takes place. If `resampleRate` matches the `sampleRate` of an audio file, that file is simply read with `readAudioFile` (see **Sub-Section 7.3.1**), otherwise the file is resampled with `resampleAudioFile` (see **Sub-Section 7.3.2**).

## 7.3 Dependencies

### 7.3.1 Audio Read Package

The *Audio Read* package reads WAV audio files with corrupt header information that cannot be read with MATLAB's `audioread`. The package also supports RAW files.

It is not uncommon for audio recording software to experience an amount of data loss, which in some cases can stop the recording process, unexpectedly closing the audio file. This is an issue frequently seen in PAMGuard and is the result of limitations in the hardware, the connection between the hardware and PAMGuard (e.g., telemetry), or the storage device (e.g., low-quality flash memory). In most cases, this sudden interruption of the recording only affects the WAV header and a few bytes of audio data, and the audio information can still be recovered.

The package comprises two functions: `readwavHeader`, for reading the header information of a WAV file; and `readAudioFile` for reading the data from a WAV/RAW audio file or a portion of it. For further details, check the `help` of these functions.

### 7.3.2 Audio Resampling Package

The *Audio Resample* package includes functions for *resampling* and *filtering* audio files using the Fast-Fourier Transform (FFT). The functions divide an audio file into manageable ~50 MB segments. Those segments are filtered in the frequency domain, converted back into time waveforms, and joined again using the *overlap-add* reconstruction. This "small-segment" approach is faster and more memory-efficient than applying MATLAB's `resample` or `filter` functions to an entire audio file. In most cases, using those two functions directly is simply impractical due to the computer's limited physical memory (RAM). The package comprises three functions: `resampleAudioFile`, `resamplingFactors`, and `filterAudioFile`.

`resampleAudioFile` is an alternative to MATLAB's `resample` for resampling audio files efficiently using FFT filtering with overlap-save method. The function first designs a low-pass FIR filter of length N that works both as a *reconstruction filter* for the upsampled signal and as an *anti-alias filter*, applied before downsampling. Then, the filter is zero-padded to a length L = M + N – 1 and its FFT is calculated. The audio file is divided into segments of length M/P. The steps below are followed for every segment in the audio file:

1. A segment of length M/P is read from the audio file and upsampled by P.
2. The result is appended to the overlapping portion kept from the previous upsampled segment (*overlap save*), making the signal L = M + N - 1 samples long. The last N – 1 samples are kept for the next segment.

3. The FFT is computed for the upsampled segment of length L.
4. The FFT of the upsampled segment and recovery/anti-alias filter are multiplied.
5. The IFFT of the latter product is computed.
6. The last N – 1 samples, affected by the filter's distortion effects, are discarded. Note that the filter causes leading and trailing distortion effects. Both effects are combined at the end of the segment by applying circular convolution of length N – 1 to the FIR filter. This can be alternatively achieved by using the complex conjugate of the filter's FFT.
7. The result is downsampled by Q

`resamplingFactors` calculates the optimal resampling factors for `resampleAudioFile` given the original and target sampling rates, the size of the individual resampling segments and the error tolerance for the target sampling rate. An error tolerance for the new sampling rate is necessary to avoid large processing times and memory usage when the upsampling factor is large (i.e. ratio between new and original sampling rates contains many decimal points).

`filterAudioFile` filters audio files efficiently using FFT filtering and overlap-save reconstruction. It supports FIR and IIR filters. The function divides the audio signal into segments of length M and filters each segment with a filter of input coefficients (`a`,`b`). The steps below are followed for every segment in the audio file:

1. A segment of length M is read from the audio file.
2. The result is appended to the overlapping portion kept from the previous segment (*overlap save*), making a signal of length L = M + N - 1 samples. The last N – 1 samples are kept for the next segment.
3. The segment is filtered. The filtering approach is different for a FIR or IIR filter.
    a. *FIR filter*: the FFT of the segment and filter are calculated. Then, the IFFT of their product is computed.
    b. IIR filter: the segment is filtered using the `filter` function with the vector of final conditions from the previously filtered segment.
4. The last N – 1 samples, affected by the filter's distortion effects, are discarded.

# 8 Audio Detect Module

'Audio Detect' determines from the selected audio files the start and end limits of every detection and associated background noise.

**Figure 43** shows the flow chart of 'Audio Detect'. The module calls `readAudioDetectConfig`, `audioDetectFun`, and `updateAcousticDatabases`. The first two are described in **Sections 8.1** and **8.3**, while the latter was addressed in **Section 6.6**. The module also calls the *FFT Filtering* package stored in the 'Dependencies (External)' folder and briefly described in **Sub-Section 8.4.1**.

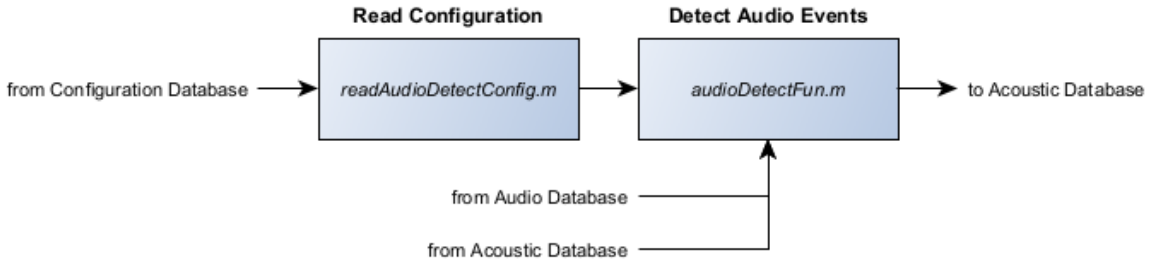


**Figure 43** Flow diagram of 'Audio Detect' module.

The above flow chart changes if the `'NeymanPerson'` detector is used. In this case, the function `preProcessNeymanPearson` needs to be executed before `audioDetectFun`. The function is described in **Section 8.2**. The *Covariance Estimation* package on which `preProcessNeymanPearson` relies for estimating the covariance matrices is described in **Sub-Section 8.4.2**.

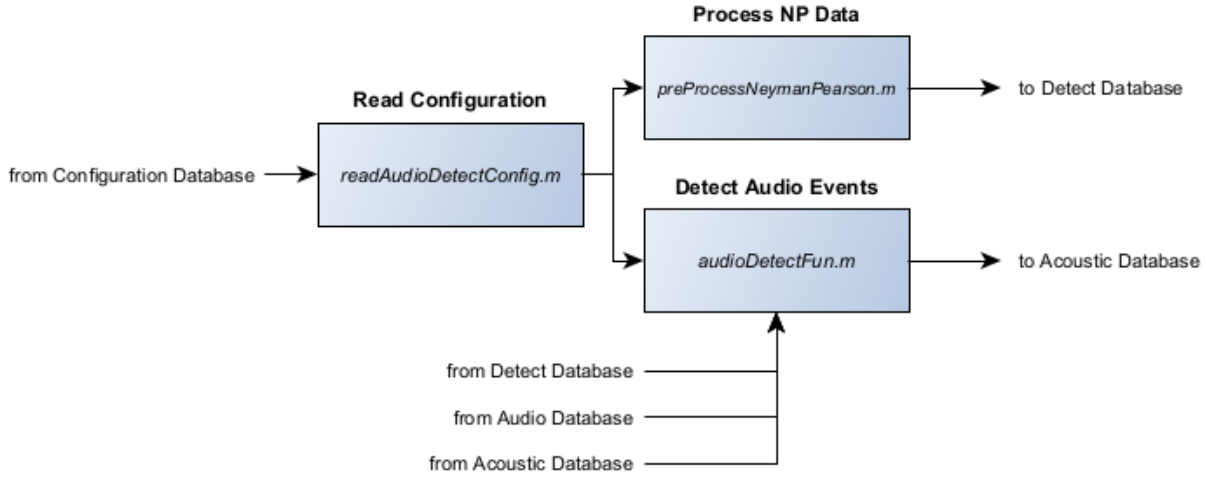


**Figure 44** Flow diagram of 'Audio Detect' module when the Neyman-Pearson detector is used.

## 8.1 Read Configuration Files

The *audioDetectConfig*.json* files, stored in directory '<ROOT.BLOCK>\configdb', are read with `readAudioDetectConfig`. In this stage, the information in the *audioDetectConfig*.json* files is verified and organised for the internal software routines to perform the appropriate actions.

`readAudioDetectConfig` reads the configuration scripts in the order specified by the '_<NUM>' string appended to the file name. Each configuration script goes through a four-step process:

1. The content of the file is read with MATLAB's `jsondecode`, which creates a structure `AudDetConfigFile` containing the parameters of the specific configuration file.
2. `updateAudioDetectConfig` verifies the input parameters in `AudDetConfigFile` one-by-one and assigns them to a single-element configuration structure `AudDetConfigOne`.
3. `verifyAudioDetectConfig` looks for any non-supported parameter combination and any individual parameter in `AudDetConfigOne` that `updateAudioDetectConfig` flagged as non-valid, and sets the flag `inputStatus` in `AudDetConfigOne` to `false` if any non-valid parameter or combination of them is found in `AudDetConfigOne`.
4. The process is repeated with all *audioDetectConfig*.json* files. A multi-element structure `AudDetConfig` is generated from all the individual `AudDetConfigOne` structures.
5. `readAudioDetectConfig` returns the verified `AudDetConfig` structure.

`AudDetConfig` contains as many elements as *audioDetectConfig*.json* scripts are found in the Configuration Database folder and has the same fields regardless of the configuration template (see **AudDetConfig** in **Sub-Section 6.5.1**). The aim of `inputStatus` is to indicate that a mistake has been made in the configuration file `configFileName`. Elements in `AudDetConfig` where `inputStatus = false` are ignored by `audioDetectFun`.

## 8.2 Pre-Process Neyman-Pearson Data

If `'NeymanPearson'` is selected as the detector in `AudDetConfig(m)`, some calculations need to be carried out before `audioDetectFun` can be executed. `preProcessNeymanPearson` computes the raw scores and covariance matrices of the target signal and noise, the eigendata, and the performance curves. This information is stored in '<ROOT.BLOCK>\detectiondb' as individual *.mat files. The files are later used by the Neyman-Pearson (NP) algorithm to calculate the *threshold* $\gamma'(\mathbf{x})$ and *test statistic* $T(\mathbf{x})$ on each audio segment, which is classified as "detection" if $T(\mathbf{x}) > \gamma'(\mathbf{x})$.

Note that there are three types of NP detectors: energy detector (`'ed'`), estimator-correlator in white Gaussian noise (`'ecw'`), and estimator-correlator in coloured Gaussian noise (`'ecc'`). In the following sub-sections we describe what the four types of data consist in (raw scores, covariance, eigendata, and performance curves) and which of them are required by which type of NP detector.

### 8.2.1 Raw Scores Matrix

A matrix of *raw scores* contains all the information that is necessary to build a covariance matrix. It comprises as many rows as variables $N$ and as many columns as *training* observations $M$.

There are two raw scores matrices, one for the target signal and one for the background noise. They are built from training audio segments stored in subfolders 'signal' and 'noise' under the parent folder `trainFolder` specified in `AudDetConfig(m).DetectParameters`.

The raw scores matrices are generated with `rawScores`. The function collates the training segments, and then resamples and trims them based on the parameters `resampleRate` and `kernelDuration` specified in `DetectParameters`. There is only one raw scores file per combination of target signal, sampling rate (after resampling), and segment duration. The `'ecw'` only uses the raw scores matrix of the target signal, while the `'ecc'` uses both. The `'ed'` doesn't require training data since the signal and noise are both assumed white Gaussian.

### 8.2.2 Covariance Matrix

A covariance matrix contains the variances of all $N$ variables from the $M$ training observations (diagonal entries) and the covariances between every variable pair (off-diagonal entries). It is built from a $[N, M]$ matrix of raw scores and it comprises as many rows and columns as variables.

In a high-dimensional setting ($M < N$), the *sample covariance matrix* (SCM) becomes inaccurate and a better covariance estimator is needed. The *Covariance Estimation* package (**Sub-Section 8.4.2**) addresses this issue by offering a range of *shrinkage* covariance estimators that can help minimise the impact that a noisy covariance matrix estimate may have on the detection performance.

There are two covariance matrices, one for the target signal $\mathbf{\Sigma}_s$ and one for the background noise $\mathbf{\Sigma}_w$. In practice, we are interested in the normalised covariance matrices $\breve{\mathbf{\Sigma}}_s = \sigma_s^{-2}\mathbf{\Sigma}_s$ and $\breve{\mathbf{\Sigma}}_w = \sigma_w^{-2}\mathbf{\Sigma}_w$, as these are independent of the variance of the training observations. The normalised covariance matrices $\breve{\mathbf{\Sigma}}_{s,w}$ are calculated from training observations that have been previously normalised by their variance. Note that the mean of the diagonal entries in $\breve{\mathbf{\Sigma}}_{s,w}$ is 1. The function `covariance` computes the *normalised* covariance matrices from the matrices of raw scores. The latter are stored in directory '<ROOT.BLOCK>\detectiondb'.

Provisional *resampling* and *filtering* can be applied to the segments of an audio file to processed for detections. This is done to improve the SNR and, as a result, the performance of the detector. `covariance` applies that same filtering and resampling to the input matrix of raw scores. There is one covariance file per combination of target signal, detector type (`'ed'`, `'ecw'`, `'ecc'`), sampling rate (after *detection* resampling), cutoff frequencies of the *detection* filter, and segment duration. The `'ecw'` only uses the covariance matrix of the target signal, while the `'ecc'` uses both. The `'ed'` doesn't require covariance matrices.

### 8.2.3 EigenData

A square matrix $\mathbf{A}$ can be expressed as the product $\mathbf{V\Lambda V}^{-1}$, where $\mathbf{V}$ is the matrix of eigenvectors (or *modal* matrix) and $\mathbf{\Lambda}$ is the matrix of eigenvalues (or *diagonal* matrix). This is commonly known as the *eigen decomposition* of $\mathbf{A}$. In $\mathbf{V}$, each column is an eigenvector, and $\mathbf{\Lambda} = \mathrm{diag}\{\lambda_1, \lambda_2, \ldots \lambda_N\}$ with $\lambda_n$ representing the $n^{th}$ eigenvalue. The modal and diagonal matrices are referred to as the *eigen data*.

The eigen data is of particular interest as it can be used to *diagonalise* a covariance matrix $\mathbf{\Sigma}$ through the expression $\mathbf{V}^{-1}\mathbf{\Sigma V}$ and *decorrelate* an observation $\mathbf{x} = \{x_1, x_2, \ldots x_N\}$ with $\mathbf{V}^{-1}\mathbf{x}$. The decorrelation transforms the fraction of energy in $\mathbf{x}$ that is statistically similar to $\mathbf{\Sigma}$ into white Gaussian noise. Note that for a symmetric matrix (such as the covariance $\mathbf{\Sigma}$), the modal matrix is *orthogonal* ($\mathbf{V}^{-1} = \mathbf{V}^T$).

The eigen decomposition is necessary for the estimator-correlators, since only with the eigen data one can arrive at the expressions of the *test statistic* $T(\mathbf{x})$ and *performance curves* $p_{0,1}(T)$ **(Kay, 1998)**. The energy detector doesn't need covariance matrices or eigen data: the test statistic is simply the energy of the observation $\mathbf{x}$ and the performance curves are given by the Chi-squared PDF of the $N$-variable vector $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$, where $\sigma^2$ is the variance of the null ($\sigma_w^2$) or alternative ($\sigma_s^2 + \sigma_w^2$) hypotheses.

The function `eigenEquation` computes the eigen data for the two estimator-correlators, returned as a structure `EigenData`. The content of `EigenData` depends on the type of estimator-correlator (`'ecw'`, `'ecc'`). For the estimator-correlator in WGN (`'ecw'`), `EigenData` contains the eigenvectors $\mathbf{V}_s$ and eigenvalues $\breve{\boldsymbol{\lambda}}_s$ of the *normalised* covariance matrix of the target signal $\breve{\mathbf{\Sigma}}_s$. For the estimator-correlator in CGN (`'ecc'`), `EigenData` contains the eigenvectors $\mathbf{V}_w$ and eigenvalues $\breve{\boldsymbol{\lambda}}_w$ of the *normalised* covariance matrix of the target noise $\breve{\mathbf{\Sigma}}_w$, and the eigenvectors $\mathbf{V}_B$ and eigenvalues $\breve{\boldsymbol{\lambda}}_B$ of the compound signal-noise matrix $\breve{\mathbf{B}} = \breve{\mathbf{A}}^T\breve{\mathbf{\Sigma}}_s\breve{\mathbf{A}}$, where $\breve{\mathbf{A}} = \mathbf{V}_w\breve{\mathbf{\Lambda}}_w^{-1/2}$.

There is one eigen data file per combination of target signal, detector type (`'ed'`, `'ecw'`, `'ecc'`), sampling rate (after *detection* resampling), cutoff frequencies of the *detection* filter, and segment duration. `'ed'` doesn't require eigen data.

### 8.2.4 Performance Data

The performance data refers to the probability density functions (PDF) of the null and alternative hypotheses $p_{0,1}(T)$, or alternatively, the probability functions (PF) of false alarm and detection $P_{FA,D}(T)$. The latter is the right-tail probability of the former. The performance of a detector is fully characterised by $p_{0,1}(T)$ or $P_{FA,D}(T)$, insofar as the covariance matrix estimate is accurate. Given a target probability of false alarm $\alpha$, the threshold $\gamma'(\mathbf{x})$ for observation $\mathbf{x}$ is obtained from $P_{FA,D}(T)$. The observation is classified as "detection" if the test statistic $T(\mathbf{x})$ is larger than the threshold $\gamma'(\mathbf{x})$.

$p_{0,1}(T)$ and $P_{FA,D}(T)$ represent the PDF and PF of a test statistic $T(\mathbf{x})$, where $\mathbf{x}\sim\mathcal{N}(0,\boldsymbol{\Sigma}_w)$ for the null hypothesis and $\mathbf{x}\sim\mathcal{N}(0,\boldsymbol{\Sigma}_s+\boldsymbol{\Sigma}_w)$ for the alternative hypothesis. $T(\mathbf{x})$ depends on the variance of the observation $\sigma_t^2$ $(=\sigma_w^2+\sigma_s^2)$, and so do $p_{0,1}(T)$ and $P_{FA,D}(T)$. Fortunately, there is no need to compute the performance curves for every single combination of signal and noise variances. It is enough to calculate $p_{0,1}(T)$ for a number of $SNR$ and scale the curves by the noise variance of $\mathbf{x}$, that is, $p_0(T,\sigma_w^2)=\sigma_w^{-2}p_0(T\sigma_w^{-2},1)$ and $p_1(T,SNR,\sigma_w^2)=\sigma_w^{-2}p_1(T\sigma_w^{-2},SNR,1)$.

`characterisePerformance` calculates $p_{0,1}(T)$ and $P_{FA,D}(T)$ for multiple $SNR$ (-50 dB to 50 dB in 1 dB steps) and a reference noise variance $\sigma_w^2=1$. The function returns a multi-element structure `PerformanceData` where each element corresponds to one $SNR$. For the estimator-correlators (`'ecw'`, `'ecc'`), the performance curves are calculated from the normalised eigenvalues $\check{\boldsymbol{\lambda}}_B$ and $\check{\boldsymbol{\lambda}}_s$, the signal variance $\sigma_s^2$, and the noise variance $\sigma_w^2$. The energy detector (`'ed'`), only requires the number of variables $N$, the signal variance $\sigma_s^2$, the noise variance $\sigma_w^2$, and the cutoff frequencies of the detection filter $f_{1,2}$ to calculate the performance curves.

## 8.3 Detect Audio Events

`audioDetectFun` calculates the relative time limits of every detection and associated background noise using the detection algorithm and parameters given in each element of `AudDetConfig`. The function is executed on the audio files listed in *audioPaths.json*. Note, that `audioDetectFun` works with the data in the Audio Databases in '<ROOT.BLOCK>\audiodb' rather than with the original audio files to improve the reading and processing speed.

`audioDetectFun` creates a *.mat Acoustic Database file per audio file listed in *audioPaths.json* and saves it in '<ROOT.BLOCK>\acousticdb'. The configuration information and detection results from each `receiverName-sourceName` combination is stored as one element in Acoustic Database sub-structures `AcoConfig(m).AudDetConfig` and `AcoData(m).AudDetData`. Any element in `AudDetConfig` with a repeated `receiverName-sourceName` combination or an `inputStatus = false` is ignored.

The `'MovingAverage'` and `'NeymanPearson'` detectors call the FFT Filtering package to bandpass-filter each segment with `fftFilter` before applying the corresponding detection algorithm (see **Sub-Section 8.4.1**). The filter aims at improving the detection performance by limiting the analysis to the dominant frequency region of the target signal.

## 8.4 Dependencies

### 8.4.1 FFT Filtering (Single) Package

The *FFT Filtering (Single)* package calculates the RMS and exposure of an input signal within a specific frequency band. The package uses the Fast Fourier Transform (FFT) to compute the power spectrum

of the signal. The algorithm adds the spectral samples within the specified band to obtain the corresponding power, from which the in-band RMS and exposure are calculated.

Unlike the conventional FIR-type filtering based on FFT and reconstruction methods (overlap-add, overlap-save), the *FFT Filtering (Single)* package computes the metrics within a band directly from the spectrum rather than from the time-domain version of the filtered signal. When the metrics of a filtered signal (but not the filtered signal itself) are of interest, the approach used in the *FFT Filtering (Single)* package is considerably more efficient.

The package comprises two main functions: `fftSingleFilterDesign`, for designing the FFT filter; and `fftSingleFilter`, for filtering a signal using a previously-designed FFT filter.

`fftSingleFilterDesign` designs a digital FFT filter to work with `fftSingleFilter`. The function returns a structure `FftFilter` containing the sampling rate, cutoff frequencies and normalised cutoff frequencies. `fftSingleFilterDesign` is lightweight as it only performs error control operations on the input filter parameters. For consistency with other filtering packages that require more demanding filter design calculations, the *design* and *filtering* functionalities are kept separate. In general, a filter should be designed once but applied many times.

`fftSingleFilter` filters an input signal using the filter designed with `fftSingleFilterDesign`. The function is a wrapper of `fftFilter`. The functionality of `fftSingleFilter` is limited to error control operations and to the calculation of the exposure metric from the filtered RMS value returned by `fftFilter`. The latter computes the RMS amplitudes of a signal within the frequency bands given by two vectors containing the bottom and top half-power frequencies. In this case, there is only one bottom and one top half-power frequency. `fftFilter` also applies a suitable amount of zero-padding to the signal before calculating the FFT to avoid *energy leakage* on short signals at low frequencies.

The zero-padding solution attempts to address a problem arises from the *uncertainty principle* in signal processing, which states that a signal and its Fourier transform cannot be *highly concentrated*. It means that the product of a signal duration $\Delta t$ and its bandwidth $\Delta f$ must be larger than a constant ($\Delta t \cdot \Delta f \geq K$). In practical terms, that constant is of the order $K \approx 10$. It is common practice to filter signals in bands with a bandwidth that is proportional their central frequency, which means that lower bands are more likely to be affected by the uncertainty principle.

Therefore, trying to filter short signals at low frequency bands will lead to problems. Those problems manifest in a different way in FFT and digital filters. In an FFT, not meeting the condition $\Delta t \cdot \Delta f \geq K$ means that the signal bandwidth $\Delta f$ will be lower than $K$ times the frequency resolution given by $f_s/N = 1/\Delta t$, where $f_s$ is the sampling rate and $N$ the number of samples in time and frequency. In a digital filter, not meeting the uncertainty condition will delay the filtered signal, displacing it outside its time window. The end effect in either case is *energy leakage* on the filtered signal. That leakage is progressive for digital filters (larger the smaller $\Delta t \cdot \Delta f$) and abrupt for FFT filtering.

A simple but effective solution for addressing the problems filtering short signals is to extend the duration of the signal by applying zero-padding. Adding zeroes does not introduce new signal information, but it does address the energy leakage effect to a great extent. Nonetheless, the zero-padding approach is not a magic solution. The end effect of zero-padding is to reduce the constant $K$ from $10$ to a more reasonable value $K \approx 1$, for an error in spectral level estimation of ~1 dB. In general, the accuracy of any band will deteriorate dramatically the lower the factor $\Delta t \cdot \Delta f$ is from 1.

For further details, check the `help` of these functions.

### 8.4.2 Covariance Estimation

The *Covariance Estimation* package contains eight different *shrinkage* algorithms for estimating covariance matrices. All the algorithms in the package belong to a group of shrinkage estimators based

on a linear combination of the sample covariance matrix (SCM) and a target matrix. Formally, the shrinkage estimator is expressed as $\mathbf{\Sigma} = \mathbf{S}(1-\delta) + \mathbf{F}\delta$, where $\mathbf{S}$ is the *SCM*, $\mathbf{F}$ is the *target matrix*, and $\delta$ is the *shrinkage coefficient*. This type of estimators was first proposed by **Ledoit & Wolf (2003b)**.

The sample covariance matrix (SCM) is an optimal estimator when the number of *training* observations greatly exceeds their number of variables ($M \gg N$). However, when $M$ is lower than or comparable to $N$, the SCM becomes a "noisy" estimator and alternative solutions are needed. An inaccurate covariance estimate will have a negative impact on the performance of a Neyman-Pearson estimator-correlator. In particular, it will affect the simulated probability density functions $p_{0,1}(T)$ and the computed test statistic $T(\mathbf{x})$, and how well the former characterises the statistical behaviour of the second.

Fortunately, the SCM is not the only covariance estimator available. The field of covariance estimation in high-dimensions has experienced great progress from the early 2000s, encouraged by advances in the fields of finance, bioinformatics, climate studies, risk management, and audio processing. Among the most popular covariance estimation methods are the banding, tapering, thresholding, maximum likelihood, sparse Gaussian graphical models, and shrinkage. Only the last method makes no structural assumptions about the covariance matrix and can be applied to a wide range of covariance structures with positive results. In the next paragraphs, we briefly describe the various shrinkage algorithms implemented as part of the *Covariance Estimation* package.

The `covParam1` estimator uses a *scaled identity* target matrix. The *scaling factor* is equal to the mean variance of the SCM, that is, the mean of its diagonal entries. The shrinkage coefficient $\delta$ is efficiently calculated as the ratio $p/c$, where $p$ is a simple function of the SCM and $c$ is the squared Frobenious norm of the difference $\mathbf{S} - \mathbf{F}$. The method was proposed by **Ledoit & Wolf (2004)** and is one of the simplest and more widely used covariance estimators.

The `covRblw` and `covOas` estimators are approximations to the linear shrinkage estimator from **Ledoit & Wolf (2004)**. Both methods were proposed by **Chen et al (2010)**. The two consist in simple analytical expressions that improve over the previous algorithm when a Gaussian distribution is assumed. `covRblw` applies the *Rao-Blackwell* theorem to the method from **Ledoit & Wolf (2004)**, and is therefore denoted as RBLW. `covOas`, referred to as oracle approximating shrinkage (OAS), is a closed form expression derived for the limit of an iterative approach which approximates the shrinkage estimator from **Ledoit &Wolf (2004)**. Both algorithms use the scaled diagonal as target matrix. The expression for the shrinkage coefficient is identical in both cases, except for the value of two specific constants. Simulations demonstrate that the OAS approach can perform better in high-dimensions ($M \ll N$).

The `covStock` and `covCorr` are designed for the estimation of the covariance matrix of stock returns in portfolio optimisation and were proposed by **Ledoit & Wolf (2003a; 2003b)**. These methods use specialised structured shrinkage targets suitable for financial applications. The shrinkage coefficient $\delta$ is calculated as the ratio $(p-r)/c$. This is the same expression used in Ledoit & Wolf (2004), except for the $r$ parameter, calculated by `covStock` and `covCorr` which carry out different elaborate operations on the elements of the matrix of raw scores.

The `covDiag` estimator is a variation of the estimator from **Ledoit & Wolf (2004)** that uses a *diagonal* target matrix and an extra parameter for the shrinkage coefficient. In the diagonal target matrix the diagonal entries are identical to those in the SCM and the off-diagonal entries are all zero. The shrinkage coefficient $\delta$ is calculated as the ratio $(p-r)/c$. The parameter $r$ is obtained as a mid-step from the calculation of $p$. The algorithm was developed by Olivier Ledoit and can be found at http://ledoit.net.

The `covParam2` estimator is another variation of the estimator from **Ledoit & Wolf (2004)**. In this case, the algorithm uses a two-parameter target matrix with constant values for the diagonal and off-diagonal entries. The shrinkage coefficient $\delta$ is calculated as the ratio $(p - r)/c$. The parameter $r$ is obtained from *direct* and *recursive* operations on the matrix of raw scores. The latter can be computationally demanding but it can be disabled, since its contribution to $\delta$ is generally small. The algorithm was developed by Olivier Ledoit and can be found at http://ledoit.net.

The `covLooc` estimator employs a computationally efficient shrinkage estimator that uses Leave-One-Out Cross Validation (LOOCV) to compute the shrinkage coefficient based on **Hoffbeck & Landgrebe's (1996)** manipulation of the likelihood function. The method was proposed by **Theiler (2012)**, who referred to this estimator as the Leave-One-Out Covariance or *LOOC*. We have updated the method to use a shrinkage target different from the proposed scaled diagonal. `covLooc` can take any shrinkage target. The function `targetCovariance` produces a shrinkage target equal to the optimal covariance matrix of a white Gaussian process processed with a specified filter. The *filtered target* outperforms the *scaled identity target* when used for the estimation of covariance matrices of filtered or band-limited signals.

# 9 Audio Process Module

'Audio Process' computes the offset-corrected timestamps and the broadband and band acoustic metrics for every detection and associated background noise obtained by the 'Audio Detect' module.

**Figure 45** shows the flow chart of 'Audio Process'. The module calls `readAudioProcessConfig`, `audioProcessFun`, and `updateAcousticDatabases`. The first two are described in **Sections 9.1** and **9.2**, while the latter was addressed in **Section 6.6**. 'Audio Process' also calls the *Digital Filtering (Bank)* and *FFT Filtering (Bank)* packages, stored in the 'Dependencies (External)' folder and briefly described in **Sub-Sections 9.3.1** and **9.3.2**.

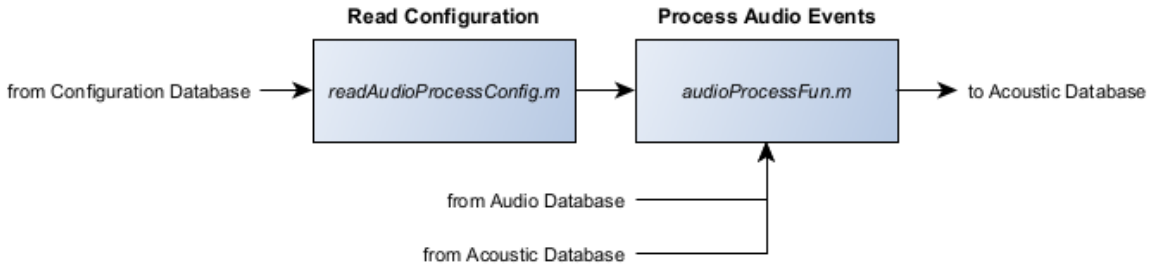


**Figure 45** Flow diagram of 'Audio Process' module.

## 9.1 Read Configuration Files

The *audioProcessConfig*.json* files, stored in directory '<ROOT.BLOCK>\configdb', are read with `readAudioProcessConfig`. In this stage, the information in the *audioProcessConfig*.json* files is verified and organised for the internal software routines to perform the appropriate actions.

`readAudioProcessConfig` reads the configuration scripts in the order specified by the '_<NUM>' string appended to the file name. Each configuration script goes through a five-step process:

1. The content of the file is read with MATLAB's `jsondecode`, which creates a structure `AudProConfigFile` containing the parameters of the specific configuration file.
2. `updateAudioProcessConfig` verifies the input parameters in `AudProConfigFile` one-by-one and assigns them to a single-element configuration structure `AudProConfigOne`.
3. `verifyAudioProcessConfig` looks for any unsupported parameter combination and any individual parameter in `AudProConfigOne` that `updateAudioProcessConfig` flagged as non-valid, and sets the flag `inputStatus` in `AudProConfigOne` to `false` if any non-valid parameter or combination of them is found in `AudProConfigOne`.
4. The process is repeated with all *audioProcessConfig*.json* files. A multi-element structure `AudProConfig` is generated from all the individual `AudProConfigOne` structures.
5. `readAudioProcessConfig` returns the verified `AudProConfig` structure.

`AudProConfig` contains as many elements as *audioDetectConfig*.json* files are found in the Configuration Database folder and has the same fields regardless of the configuration template (see **AudProConfig** in **Sub-Section 6.5.1**). The aim of `inputStatus` is to indicate that a mistake has been made in the configuration file `configFileName`. Elements in `AudProConfig` where `inputStatus = false` are ignored by `audioProcessFun`.

## 9.2 Process Audio Events

`audioProcessFun` computes the offset-corrected timestamps and acoustic metrics for every detection and background noise segments found within the audio files listed in *audioPaths.json*. The calculations are carried out for each `receiverName-sourceName` combination from each element of `AudProConfig`.

For every audio file in *audioPaths.json* and element `m` in `AudProConfig`, `audioProcessFun` first locates the Audio Database with the same `channel` and `resampleRate` as in `AudProConfig(m)`. The function then accesses the Acoustic Database of the same name as the audio file and retrieves the time limits of the detection and background noise segments. The time limits are obtained from the Acoustic Database sub-structure `AcoData(n).AudDetData`, where `n` is the element with a `receiverName-sourceName` that matches that in `AudProConfig(m)`. `audioProcessFun` then processes the offset-corrected timestamps and acoustic metrics for each detection and associated background noise and saves the results in the *.mat Acoustic Database. The configuration information and processed results from each `receiverName-sourceName` combination is stored as one element in sub-structures `AcoConfig(m).AudDetConfig` and `AcoData(m).AudDetData`. Any element in `AudProConfig` with a repeated `receiverName-sourceName` combination or an `inputStatus = false` is ignored.

`audioProcessFun` locates the detections and background noise segments within the Audio Database and computes the broadband (peak, peak-to-peak, RMS, exposure) and the band (RMS, exposure) acoustic metrics. Before processing the metrics, the function applies a bandpass filter to each detection and noise segment to remove the energy outside the frequency range of interest. For that purpose, it uses the functions `digitalSingleFilterDesign` and `digitalSingleFilter` from the Digital Filtering package (see **Sub-Section 9.3.1**). `audioProcessFun` then estimates the exact limits of each detection from their cumulative energy curves and a target energy. The same relative limits are applied to the associated background noise segments. The broadband and band acoustic metrics are calculated over these shorter windows better adapted to the target event. The band metrics (RMS, exposure) are computed with `fftBankFilter` and `fftBankFilterDesign` functions from the *FFT Filtering (Bank)* package (see **Sub-Section 9.3.2**).

Note that the values of the acoustic metrics represent digital amplitudes. For instance, the digital amplitudes in a 16-bit audio file range from -32768 to 32767. If we refer to the instantaneous digital amplitude unit as U, the units for the acoustic metrics are U for peak, peak-to-peak and RMS, and $U^2s$ for exposure.

`audioProcessFun` also computes the offset-corrected ticks for the detections. In ABACUS, a *tick* is a decimal number representing an absolute time, in seconds, referred to '00 Jan 0000'. The offset-corrected time, if calculated correctly, should closely match the UTC time. The offset-corrected time is calculated as the difference between the PC time and the time offset. The *time offset* may be caused by a time drift in the PC clock (typically 10µs/s for conventional oscillators found on a PC), the use of a time zone other than UTC+0, or having the 'Daylight Saving Time' option enabled on the PC. The time offset can be obtained from navigation files with a double UTC-PC timestamp. If those files are not available, a note about the PC time settings should be taken at the start of the recording process. In general, the best practice is to always sync the time of any audio recording device to UTC.

Each audio file is named with the timestamp of its first sample, obtained from the PC clock. The PC timestamps of the detections are calculated by adding their *relative times* to the *starting PC timestamp* from the audio file. Through their PC timestamps, the detections can be linked to known time offsets.

The time drift of the PC oscillator is considered negligible for the typical duration of an audio file. Therefore, the time offset for the starting PC timestamp can be used for all the detections in that audio

file. In exceptional cases, an audio recording system that experiences data loss may not able to introduce the appropriate number of "filling" samples during downtimes. In this situation, the assumption that the audio file contains a continuous stream of data at a fixed sampling rate is no longer applicable, resulting in inaccurate timestamps for the detections. Since UTC timestamps are used for syncing audio data with navigation information from sources and receivers, an error in the timestamps will lead to position errors and spatial distortion in the sound field map.

## 9.3 Dependencies

### 9.3.1 Digital Filtering (Single) Package

The *Digital Filtering (Single)* package includes routines to design, represent and apply a single digital filter. The package uses IIR filter objects of *digitalFilter* class designed with MATLAB's `designfilt`. Lowpass, highpass, bandpass, and arbmag ("arbitrary magnitude") filters are supported.

The package comprises three main functions: `digitalSingleFilterDesign` for designing the digital filter, `digitalSingleFilter` for filtering a signal using a previously-designed digital filter, and `digitalSingleFilterPlot` for representing the response of the digital filter and decimation filters (if applicable).

The filter design and filtering functions are kept separate. The reason for this approach is that the *filter design* function can be computationally demanding but only needs to be executed once, while the *filtering* function is more efficient but may need to be applied multiple times.

`digitalSingleFilterDesign` designs a IIR filter for `digitalSingleFilter`. The function returns a structure `DigitalFilter` containing the filtering object and information about the decimator and digital filter. `DigitalFilter` is used as input of `digitalSingleFilter` to extract the *metrics* (RMS, exposure, peak, peak-to-peak) or the *filtered waveform* from a broadband signal.

In order to improve the efficiency of the filtering process when applying a lowpass or bandpass filter to long signals with large sampling rates, a multi-stage decimation algorithm is implemented to reduce the number of samples that need to be filtered and ensure accurate results through simple and stable filters. The function designs a single decimation filter to be applied as many times as needed.

`digitalSingleFilter` filters an input signal using the filter previously designed with function `digitalSingleFilterDesign`. The function decimates the input signal multiple times until the Nyquist frequency is as close as it can be to the top cutoff frequency of the filter. The decimation step improves the overall speed considerably by reducing the number of samples to be filtered. Note that decimation is only applied when a lowpass or bandpass filter is selected.

`digitalSingleFilter` filters the input vector by calling MATLAB's `filter` function. *Zero-phase* filtering is also supported with `myfiltfilt`, a custom version of MATLAB's `filtfilt` that doesn't experience the strong oscillations that may appear at the start and end of a filtered signal when using `filtfilt`.

`digitalSingleFilter` also applies a suitable amount of zero-padding to the signal before filtering to avoid *energy leakage* on short signals at low frequencies. For details about the effects and benefits of zero-padding in this context, see the explanation in **Sub-Section 8.4.1**. Zero-padding will increase the overall length of the filtered waveforms, an effect that is in general not desirable if one is interested in the waveform itself. The function uses MATLAB's `datawrap` to get around this issue. The `datawrap` function divides the filtered vector into segments of length equal to the original (unfiltered) vector and adds them together. The function does a good job conserving energy and phase.

`digitalSingleFilterPlot` takes an input structure `DigitalFilter` and represents the decimators (red) and filter (magenta). A marker on the response curves of the decimator and digital filter designates the half-power point (-3 dB).

### 9.3.2 FFT Filtering (Bank) Package

The *FFT Filtering (Bank)* package calculates the RMS and exposure of an input signal for a set of standard frequency bands within a specific frequency range. The package uses the Fast Fourier Transform (FFT) to compute the power spectrum of the signal. The algorithm calculates the *band RMS* by adding the spectral samples within the band. The *band exposure* is derived from the band RMS.

Unlike the conventional FIR-type filtering based on FFT and reconstruction methods (overlap-add, overlap-save), *FFT Filtering (Bank)* computes the band metrics directly from the spectrum rather than from the filtered waveform. When the metrics of a filtered signal (but not the filtered waveform) are of interest, the approach used in *FFT Filtering (Bank)* is way more efficient.

The package comprises two main functions: `fftBankFilterDesign` for designing the FFT filter bank, and `fftBankFilter` for filtering a signal in bands using a FFT filter bank designed previously.

`fftBankFilterDesign` designs a digital FFT filter bank to work with `fftBankFilter`. The function returns a structure `FftFilterBank` containing the sampling rate, the nominal central frequencies, the standard and normalised central frequencies, and the standard and normalised cutoff frequencies. `fftBankFilterDesign` is lightweight as it only performs error control operations on the input filter parameters. Nonetheless, for consistency with other filtering packages that require more demanding filter design calculations, the *design* and *filtering* functionalities are kept separate. In general, a filter should be designed once but applied many times.

`fftBankFilter` filters an input signal using the filter created with `fftBankFilterDesign`. The function is a wrapper of `fftFilter`. The functionality of `fftBankFilter` is limited to error control operations and to the calculation of the exposure metric from the filtered RMS values returned by `fftFilter`. The latter computes the RMS amplitudes of a signal within specified frequency bands. The frequency bands are given by two vectors containing the bottom and top half-power frequencies. `fftFilter` also applies a suitable amount of zero-padding to the signal before calculating the FFT to avoid *energy leakage* on short signals at low frequencies. For details about the effects and benefits of zero-padding in this context, see the explanation in **Sub-Section 8.4.1**.

For further details, check the `help` of these functions.

### 9.3.3 Digital Filtering (Bank) Package

The *Digital Filtering (Bank)* package includes routines to design, represent and apply a standard bank of digital filters. The package uses IIR filter objects of *digitalFilter* class designed with MATLAB's `designfilt`. Two types of filters are used: lowpass (LPF) for decimation and bandpass (BPF) for computing the bands in the bank. The specifications of the bandpass filters in the bank meet the ANSI S1.11 standard **(ANSI, 2004; BSI, 2014)**. As with all previous filtering packages (see **sub-sections 8.4.1**, **9.3.1** and **9.3.3**), the filter design and filtering functions are kept separate to ensure computational efficiency.

The package comprises four main functions: `digitalBankFilterDesign` for designing the digital filter bank, `digitalBankFilter` for filtering a signal using a previously-designed digital filter bank, `ansiBankFilterCompliance` for determining the class (0, 1, 2) that the designed filter complies with, and `digitalBankFilterPlot` for representing the response of the digital filters in the bank and decimation filters (if applicable).

`digitalBankFilterDesign` designs a bank of IIR filters for `digitalBankFilter`. The function returns a structure `AnsiFilterBank` containing the filtering objects and information about the decimators and digital filters. `AnsiFilterBank` is used as input of `digitalBankFilter` to extract the *band metrics* (RMS, exposure, peak, peak-to-peak) or the *filtered waveform per band* from a given broadband signal.

Similar to the *Digital Filtering (Single)* package, multi-stage decimation is implemented to reduce the number of samples that need to be filtered and ensure accurate results from simpler and stable bandpass filters. This process is particularly useful for low-frequency bands and large sampling rates. The function designs a single decimation filter to be applied as many times as needed.

`digitalBankFilter` filters an input signal using the bank of digital filters previously designed with function `digitalBankFilterDesign`. The function decimates the input signal multiple times until the Nyquist frequency is as close as it can be to the top cutoff frequency of highest bandpass filter. The decimation step improves the overall speed considerably by reducing the number of samples to be filtered. Note that decimation is only applied when a lowpass or bandpass filter is selected. In each decimation step the sampling frequency is reduced by two, meaning that multiple sub-octave bands will be computed at each decimation step. The decimation process improves the overall processing speed.

`digitalBankFilter` filters the input vector by calling MATLAB's `filter` function. *Zero-phase* filtering is also supported with `myfiltfilt`, a custom version of MATLAB's `filtfilt` that eliminates the transitory oscillations that may appear at the start and end of the filtered vector.

`digitalBankFilter` also applies a suitable amount of zero-padding to the signal before filtering to avoid *energy leakage* on short signals at low frequencies. For details about the effects and benefits of zero-padding in this context, see the explanation in **Sub-Section 8.4.1**. Zero-padding will increase the overall length of the filtered waveforms, an undesirable effect when we want to recover the waveform itself. The function uses MATLAB's `datawrap` to get around this issue, alike in the *Digital Filtering (Single)* package.

`ansiBankFilterCompliance` returns the ANSI S1.11 class that the input bandpass filter is compliant with (0,1,2). The function returns -1 if the input filter does not meet the least restrictive specification (class 2). `ansiBankFilterCompliance` is used by `digitalBankFilterDesign` to calculate the necessary filter order based on a target filter class.

`digitalSingleFilterPlot` takes an input structure `AnsiFilterBank` and represents the decimators (red) and filters (magenta). A marker on the response curves of the decimator and digital filter designates the half-power point (-3 dB).

# 10 Receiver Import Module

'Receiver Import' creates a Navigation Database, if it does not exist already, and populates it with general information and position data from individual receivers. The position data is introduced manually or read from navigation files in GPS, AIS and P190 formats.

**Figure 46** shows the flow chart of 'Receiver Import'. The module calls `readReceiverImportConfig` and `receiverImportFun`. These functions are described in **Sections 10.1** and **10.2**. The module also calls the Read GPS, Read AIS and Read P190 packages, stored in the 'Dependencies (External)' folder and briefly described in **Sub-Sections 10.3.1**, **10.3.2**, and **10.3.3**.

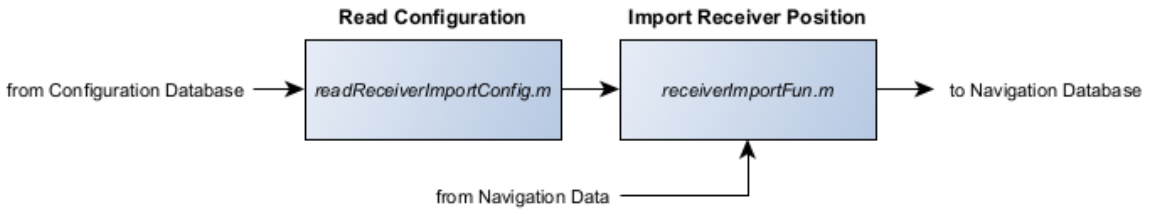


**Figure 46** Flow diagram of 'Receiver Import' module.

## 10.1 Read Configuration Files

The *receiverImportConfig*.json* files, stored in directory '<ROOT.BLOCK>\configdb', are read with `readReceiverImportConfig`. In this stage, the information in the *receiverImportConfig*.json* files is verified and organised for the internal software routines to perform the appropriate actions.

`readReceiverImportConfig` reads the configuration scripts in the order given by the '_<NUM>' string appended to the file name. Each configuration script goes through a five-step process:

1. The content of the file is read with MATLAB's `jsondecode`, which creates a structure `RecImpConfigFile` containing the parameters of the specific configuration file.
2. `updateReceiverImportConfig` verifies the input parameters in `RecImpConfigFile` one-by-one and assigns them to a single-element configuration structure `RecImpConfigOne`.
3. `verifyReceiverImportConfig` looks for any unsupported parameter combination and any individual parameter in `RecImpConfigOne` that `updateReceiverImportConfig` flagged as non-valid, and sets the flag `inputStatus` in `RecImpConfigOne` to `false` if any non-valid parameter or combination of them is found in `RecImpConfigOne`.
4. The process is repeated with all *receiverImportConfig*.json* files. A multi-element structure `RecImpConfig` is generated from all the individual `RecImpConfigOne` structures.
5. `readReceiverImportConfig` returns the verified `RecImpConfig` structure.

`RecImpConfig` contains as many elements as *receiverImportConfig*.json* files are found in the Configuration Database folder and has the same fields regardless of the configuration template (see **RecImpConfig** in **Sub-Section 6.5.1**). The aim of `inputStatus` is to indicate that a mistake has been made in the configuration file `configFileName`. Elements in `RecImpConfig` where `inputStatus = false` are ignored by `receiverImportFun`.

## 10.2 Import Receiver Position Data

`receiverImportFun` creates a *.mat Navigation Database in '<ROOT.BLOCK>\navigationdb', if it doesn't already exist, and populates it with general information and position data from the receivers in `RecImpConfig`. The *general information* is extracted directly from `RecImpConfig`. The *position*

*data* may be introduced manually (`receiverCategory = 'fixed'`) or read from navigation files specified in `RecImpConfig` (`receiverCategory = 'towed'`). The function calls `readgps`, `readais` and `readp190` to extract the time and position data from the respective file types.

The receiver information is stored in the Navigation Database in two multi-element structures: `RecImpConfig`, containing the configuration information; and `RecImpData`, containing the time and position data. Each element in the two structures corresponds to one receiver (for details, see **Sub-Sections 6.4.1** and **6.4.2**).

If a `receiverName` in `RecImpConfig` already exists in the Navigation Database, that receiver is not processed. To reprocess an existing receiver, the user must delete it first with `deleteReceivers`. Any receiver in the Navigation Database can also be renamed with `renameReceiver`.

## 10.3 Dependencies

### 10.3.1 Read GPS Package

The *Read GPS* package reads the navigation data from one or more supported files containing GPS sentences (*.gpstext) or decoded GPS data (*.csv). There are two file formats currently supported: *.gpstext, generated by SeicheSSV software; and *.csv, extracted in table form from PAMGuard's SQLite3 database.

The package utilises one main function: `readgps`. The function reads the GPS data from one or more supported files and returns it in a `GpsData` structure. `readgps` can read sentences of RMC, GGA and GSA type, as these are the ones that provide time, latitude and longitude. Regardless of the file format and selected sentence type, `GpsData` returns PC and UTC ticks, latitude, longitude, course and speed.

In SeicheSSV files (*.gpstext), `readgps` calls the custom function `nmeaChecksum` to indicate whether the checksum in the NMEA sentence is correct. Any sentence with a `false` checksum is ignored.

For further details, check the `help` of these functions.

### 10.3.2 Read AIS Package

The *Read AIS* package reads the navigation data from one or more supported files containing decoded AIS data (*.csv). The current version does not support AIS files generated with SeicheSSV software (*.aistext), as it does not include a functionality to decode AIS sentences. The supported *.csv files are extracted in table form from PAMGuard's SQLite3 database.

The package contains one main function: `readais`. The function reads the AIS data from a *.csv table extracted from a PAMGuard database and returns it in a `AisData` structure. Regardless of the file format, `AisData` returns PC and UTC ticks, latitude, longitude, course, speed, heading and vessels specifications. AIS files contain information from all vessels in the area. Data from specific vessels can be extracted by introducing a list of MMSI numbers.

For further details, check the `help` of this function.

### 10.3.3 Read P190 Package

The *Read P190* package reads the navigation data from one or more *.p190 files produced by the navigation system of a seismic vessel during exploration activities. P1-90 is a format used in seismic surveys for logging position and field data. It comprises a *header*, for general information and format specs, and a *main body*, for real time data (line id, geographic/projected coordinates, time and various other parameters for every pulse emitted by the seismic source).

The package contains one main function: `readp190`. The function returns a `P190Data` structure containing the time, location and operational information from the source and the seismic vessel.

For further details, check the `help` of this function.

# 11 Source Import Module

'Source Import' creates a Navigation Database, if it does not exist already, and populates it with general information and position data from individual sources. The position data is introduced manually or read from navigation files in GPS, AIS and P190 formats.

**Figure 47** shows the flow chart of 'Source Import'. The module calls `readSourceImportConfig` and `sourceImportFun`. These functions are described in **Sections 11.1** and **11.2**. The module also calls the *Read GPS*, *Read AIS* and *Read P190* packages, stored in the 'Dependencies (External)' folder and briefly described in **Sub-Sections 11.3.1**, **11.3.2** and **11.3.3**.

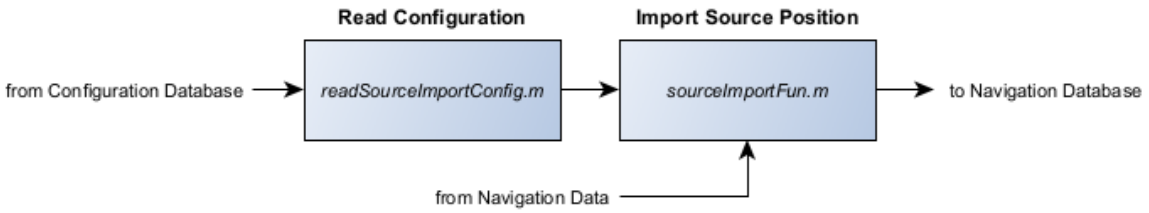


**Figure 47** Flow diagram of 'Source Import' module.

## 11.1 Read Configuration Files

The *sourceImportConfig*.json* files, stored in directory '<ROOT.BLOCK>\configdb', are read with `readSourceImportConfig`. In this stage, the information in the *sourceImportConfig*.json* files is verified and organised for the internal software routines to perform the appropriate actions.

`readSourceImportConfig` reads the configuration scripts in the order specified by the '_<NUM>' string appended to the file name. Each configuration script goes through a five-step process:

1. The content of the file is read with MATLAB's `jsondecode`, which creates a structure `SouImpConfigFile` containing the parameters of the specific configuration file;
2. `updateSourceImportConfig` verifies the input parameters in `SouImpConfigFile` one-by-one and assigns them to a single-element configuration structure `SouImpConfigOne`.
3. `verifySourceImportConfig` looks for any unsupported parameter combination and any individual parameter in `SouImpConfigOne` that `updateSourceImportConfig` flagged as non-valid, and sets the flag `inputStatus` in `SouImpConfigOne` to `false` if any non-valid parameter or combination of them is found in `SouImpConfigOne`.
4. The process is repeated with all *sourceImportConfig*.json* files. A multi-element structure `SouImpConfig` is generated from all the individual `SouImpConfigOne` structures.
5. `readSourceImportConfig` returns the verified `SouImpConfig` structure.

`SouImpConfig` contains as many elements as *sourceImportConfig*.json* files are found in the Configuration Database folder and has the same fields regardless of the configuration template (see **SouImpConfig** in **Sub-Section 6.5.1**). The aim of `inputStatus` is to indicate that a mistake has been made in the configuration file `configFileName`. Elements in `SouImpConfig` where `inputStatus = false` are ignored by `sourceImportFun`.

## 11.2 Import Source Position Data

`sourceImportFun` creates a *.mat Navigation Database in '<ROOT.BLOCK>\navigationdb', if it doesn't already exist, and populates it with general information and position data from the sources in

`SouImpConfig`. The *general information* is extracted directly from `SouImpConfig`. A special case is the 'fleet' source category, consisting of a set of vessels whose specifications are obtained from an external 'Vessel Database' (if available). The *position data* may be introduced manually (`sourceCategory = 'fixed'`) or read from navigation files in `SouImpConfig` (`sourceCategory = {'towed','vessel','fleet'}`). The function calls `readgps`, `readais` and `readp190` to extract the time and position data from the respective file types.

The source information is stored in the Navigation Database in four multi-element structures: `SouImpConfig`, containing the configuration information of target (primary) sources; `SouImpData`, containing the time and position data of target (primary) sources; `VesImpConfig`, containing the configuration information of vessels from `'fleet'` category; and `VesImpData`, containing the time and position data of vessels from `'fleet'` category. Each element in `SouImpConfig` and `SouImpData` corresponds to one target source, whereas each element in `VesImpConfig` and `VesImpData` corresponds to one vessel from `'fleet'` category (for details, see **Sub-Sections 6.4.3**, **6.4.4**, **6.4.5** and **6.4.6**).

If a `sourceName` within `SouImpConfig` already exists in the Navigation Database, that source is not processed. To reprocess an existing source, the user must delete it first with `deleteSources`. Any source in the Navigation Database can also be renamed with `renameSource`. Additionally, if `SouImpConfig` contains a source of `'fleet'` category and this category already exists in the Navigation Database (i.e., `VesImpConfig` and `VesImpData` are not empty), the source is not processed. To reprocess the vessels of `'fleet'` category, the user must delete them first with `deleteVessels`.

## 11.3 Dependencies

### 11.3.1 Read GPS Package

See **Sub-Section 10.3.1**.

### 11.3.2 Read AIS Package

See **Sub-Section 10.3.2**.

### 11.3.3 Read P190 Package

See **Sub-Section 10.3.3**.

# 12 Navigation Process Module

'Navigation Process' computes the "smoothed" navigation parameters of the receivers, sources and vessels at the times of the detections obtained by the 'Audio Detect' module.

**Figure 48** shows the flow chart of 'Navigation Process'. The module calls `navigationProcessFun`, `readNavigationProcessConfig` and `updateAcousticDatabases`. The first two are described in **Sections 12.2** and **12.1**, while the later was addressed in **Section 6.6**. The module also calls the *Distance & Bearing* package, stored in the 'Dependencies (External)' folder and briefly described in **Sub-Section 12.3.1**.

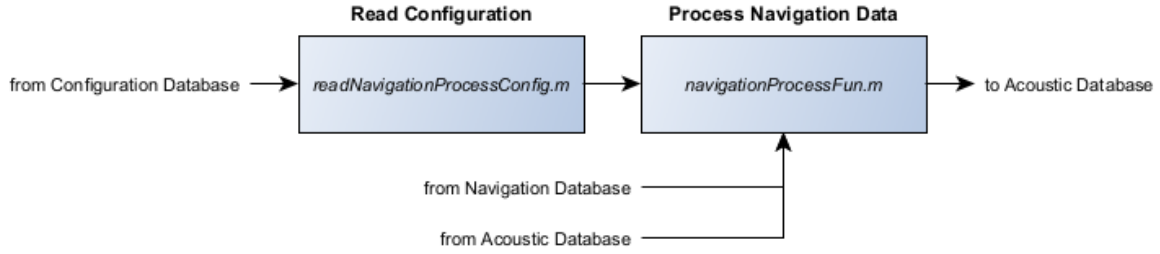


**Figure 48** Flow diagram of 'Navigation Process' module.

## 12.1 Read Configuration Files

The *navigationProcessConfig*.json* files, stored in directory '<ROOT.BLOCK>\configdb', are read with `readNavigationProcessConfig`. In this stage, the information in *navigationProcessConfig*.json* files is verified and organised for the internal software routines to perform the appropriate actions.

`readNavigationProcessConfig` reads the configuration scripts in the order specified by the '_<NUM>' string appended to the file name. Each configuration script goes through five stages:

1. The content of the file is read with MATLAB's `jsondecode`, which creates a structure `NavProConfigFile` containing the parameters of the specific configuration file.
2. `updateNavigationProcessConfig` verifies the input parameters in `NavProConfigFile` one-by-one and assigns them to a single-element configuration structure `NavProConfigOne`.
3. `verifyNavigationProcessConfig` looks for any unsupported parameter combination and any single parameter in `NavProConfigOne` that `updateNavigationProcessConfig` flagged as non-valid, and sets the flag `inputStatus` in `NavProConfigOne` to `false` if any non-valid parameter or combination of them is found in `NavProConfigOne`.
4. The process is repeated with all *navigationProcessConfig*.json* files. A multi-element structure `NavProConfig` is generated from all the individual `NavProConfigOne` structures.
5. `readNavigationProcessConfig` returns the verified `NavProConfig` structure.

`NavProConfig` contains as many elements as *navigationProcessConfig*.json* files are found in the Configuration Database folder and has the same fields regardless of the configuration template (see **NavProConfig** in **Sub-Section 6.5.1**). The aim of `inputStatus` is to indicate that a mistake has been made in the configuration file `configFileName`. Elements in `NavProConfig` where `inputStatus = false` are ignored by `navigationProcessFun`.

## 12.2 Process Source and Receiver Navigation Data

`navigationProcessFun` computes the "smoothed" navigation parameters of receivers, sources and vessels for each detection stored in the selected Acoustic Databases. The selected Acoustic Databases are those linked to the audio files in *audioPaths.json*.

`navigationProcessFun` retrieves the UTC ticks of the detections from the elements of `AcoData` structure that match the `receiverName-sourceName` combinations specified in `NavProConfig`. The Navigation Database is then accessed to extract the time and position data for `receiverName`, `sourceName` and all the vessels of `'fleet'` category. The navigation parameters of the receiver, the source and all the vessels are interpolated to the times of the detections and stored in the corresponding element of the `AcoData` structure. In particular, the navigation parameters of `receiverName`, `sourceName` and `'fleet'` vessels are stored in `AcoData(m).RecProConfig`, `AcoData(m).SouProConfig`, and `AcoData(m).VesProConfig`. One can see that each element in the Acoustic Database (`AcoData(m)` and `AcoConfig(m)`) stores navigation information for one receiver, one target (primary) source, and multiple `'fleet'` vessels. Remember that `AcoConfig` and `AcoData` are the two multi-element sub-structures that make up an Acoustic Database (see **sub-sections 6.5.1** and **6.5.2** for details about the `AcoData` structure).

The navigation parameters of the receiver are the latitude, longitude, course and speed, referred to as *self-parameters*. For every source or vessel, the function calculates *self* and *relative* parameters. The latter comprise source/vessel to receiver distance, source/vessel to receiver bearing and source/vessel direction of emission.

`navigationProcessFun` computes the navigation parameters of any receiver, source or vessel by: 1. Applying an averaging window of duration `smoothWindow` seconds to the raw position data from the Navigation Database, 2. Calculating the course and speed, 3. Calculating the position of the towed element (if applicable), 4. Interpolating the navigation parameters to the UTC times of the detections using the interpolation method `interpMethod`. Interpolation is only applied when the time gap between the two positions immediately before and after a detection is less than `maxTimeGap`, otherwise the navigation parameters for that detection are set to `NaN`. The *relative* parameters of the source and vessels are calculated directly from the navigation parameters of the receiver and the source/vessels.

Any element in `NavProConfig` with a repeated `receiverName-sourceName` combination or an `inputStatus = false` is ignored.

## 12.3 Dependencies

### 12.3.1 Distance and Bearing Package

The *Distance and Bearing* package contains various routines for calculating the distance and bearing between two points in the earth (*inverse* method) or the final point (latitude, longitude) and bearing given a starting point, distance and initial bearing (*direct* method).

The package contains six main functions: `pitagorasDirect`, `pitagoras`, `haversineDirect`, `haversine`, `vincenty` and `vincentyDirect`. Each pair corresponds to the *direct* and *inverse* versions of three different calculation methods: trigonometry, haversine and Vincenty.

The *Pitagoras* approach assumes a "regionally-flat earth", thus simple trigonometry can be applied for calculating distances, bearings, latitudes and longitudes. A constant distance 111320 m per degree of latitude is used. This value comes from assuming a perfectly spherical earth (radius $R = 6378137$ m).

The trigonometry method is the least accurate and should not be used for distances larger than 5 km. Since its implementation is particularly easy, it is a good option for spreadsheet formulas and casual calculations. With this method, the final bearing cannot be calculated (assumed identical to the initial bearing).

The *Haversine* approach assumes a spherical model of the earth. The calculations involved are somewhat more complicated than the Pitagoras method, but their analytical, non-recursive nature makes the Haversine solution a good option for a spreadsheet. Although the calculations are slightly more involved, it is as computationally efficient than the Pitagoras method. And unlike the latter, it computes both the initial and final bearings.

The *Vincenty* approach uses an elliptical model of the Earth. This is a recursive, numerical solution more involved than the previous two. It provides an accuracy of 1 mm when the correct *reference ellipsoid* is used. The parameters from a wide range of reference ellipsoids (radius at the equator, radius at the poles, and flattening) are obtained with the custom function `refEllip`. This is the recommended approach due to its greater accuracy and more than reasonable computational speed.

For further details, check the `help` of these functions.

# 13 Appendix I. Description of Structure Fields

## 13.1 Configuration Database

The parameters in the two JSON files are described in the following sub-sections.

### 13.1.1 Audio Import Module

- `audioFormat`: format of the audio files in *audioPaths.json*.
  - `'RAW'`: raw audio format (.raw, .pcm, .raw2int16)
  - `'WAV'`: wave audio format (.wav)
- `channel`: channel to import and resample.
- `resampleRate`: sampling rate after resampling [Hz]
- `sampleRate`: original sampling rate of the audio files [Hz].
- `bitDepth`: bit resolution (bits per sample)
- `numChannels`: number of channels in the audio files.
- `endianness`: endianess (`'l'` = little endian, `'b'` = big endian).

### 13.1.2 Audio Detect Module

- `receiverName`: name of the receiver to be processed.
- `sourceName`: name of the primary source to be processed.
- `mirrorReceiver`: name of the receiver to be mirrored. The current receiver `receiverName` copies the detection results from receiver `mirrorReceiver`. Useful for comparing the acoustic metrics between two adjacent sensors.
- `detector`: character vector specifying the detection algorithm.
  - `'Slice'`: the file is split into segments of equal duration. For ambient noise analysis.
  - `'MovingAverage'`: computes the RMS of consecutive windows of equal duration and flags a detection when the ratio between a front and back window is higher than a threshold.
  - `'ConstantRate'`: detections located with equally spaced windows of fixed duration and specific starting time. Suitable low-energy enegineering sources (e.g., sparker, SBP).
  - `'NeymanPearson'`: applies one of three Neyman-Pearson detectors to each segment. A detection occurs when the test statistic exceeds $T(\mathbf{x})$ the threshold $\gamma'(\boldsymbol{x})$ for the segment. The NP algorithms are the *energy detector* (`'ed'`) and the *estimator-correlators* in *white* (`'ecw'`) and *coloured* (`'ecc'`) Gaussian noise.
- `DetectParameters`: structure containing the specific parameters for the selected detection algorithm. The fields vary depending on the detection algorithm.
  - `detectorType`: type of Neyman-Pearson algorithm. There are three options: energy detector (`'ed'`), estimator-correlator in white Gaussian noise (`'ecw'`), and estimator-correlator in coloured Gaussian noise (`'ecc'`). Only for `'NeymanPearson'` detector.
  - `kernelDuration`: duration of the detection sub-window, in seconds. Only for `'Neyman-Pearson'` detector.
  - `windowDuration`: duration of the detection window, in seconds. This same duration is applied to the front (signal) and the back (noise) windows. For the `'NeymanPearson'`

detector, this is the *minimum* duration of the detection window, which contains one or more detections of duration `kernelDuration`.

- `windowOffset`: backward displacement of the front window, in milliseconds, relative to the time where the maximum energy of the detection occurs.
- `threshold`: minimum RMS ratio between the front and back window for the front segment to be classified as "detection". Note that low values have a higher risk of false alarm and high values have a higher risk of misses. Values between 1.2 (1.5 dB) and 2 (6 dB) generally work well. Use `threshold = []` for automatic thresholding. Only for `'MovingAverage'` detector.
- `rtpFalseAlarm`: target probability of false alarm. It determines the detection threshold $\gamma'$ from the probability function of false alarm $P_{FA}(\gamma')$. Only for `'NeymanPearson'` detector.
- `detectorSensitivity`: value between `0` and `1` for reducing the sensitivity of the detector (`detectorSensitivity < 1`). A value lower than `1` artificially increases the threshold $\gamma'$ to reduce the number of false alarm detections from high-energy non-target signals. Only for `'NeymanPearson'` detector.
- `minSnrLevel`: lower SNR limit for a segment to be classified as "detection". Only for `'NeymanPearson'` detector.
- `cutoffFreqs`: two-element numeric array specifying the cutoff frequencies of the detection bandpass filter. This filtering stage is temporary and is only applied for detection purposes, to enhance the SNR of the processed segment. Only for `'MovingAverage'` and `'NeymanPearson'` detectors.
- `trainFolder`: directory containing the training audio data from the target signal and noise used to build their respective covariance matrices. The respective training data is stored under sub-folders 'signal' and 'noise'. Only for `'NeymanPearson'` detector.
- `estimator`: covariance matrix estimator. There are nine types available: 'oas', 'rblw', 'param1', 'param2', 'corr', 'diag', 'stock', 'looc', and 'sample'. For details about the various estimators see *audioDetectConfig_NeymanPearson.txt*.
- `fileName`: name of the Pulse Table (.csv) containing detection parameters specific for various audio files. The table must be stored in directory '<ROOT.BLOCK>\configdb'. Only for `'ConstantRate'` detector. For details on the table format see **Sub-Section 6.1.3** or *audioDetectConfig_ConstantRate.txt*.
- `resampleRate`: new sampling rate for the `kernelDuration` segments. This resampling stage is temporary and is only applied for detection purposes, to enhance the computational speed and the SNR of the processed segment. Only for `'NeymanPearson'` detector.

### 13.1.3 Audio Process Module

- `receiverName`: name of the receiver to be processed.
- `sourceName`: name of the primary source to be processed.
- `freqLimits`: two-element array representing the bottom and top frequency limits for processing, in Hertz.
- `bandsPerOctave`: number of bands per octave (e.g., 3 for third-octave).

- `cumEnergyRatio`: ratio of the total energy of an audio segment that delimits the signal of interest. Value between 0 and 1.
- `audioTimeFormat`: timestamp format string. Typical formats are `'yyyymmdd_HHMMSS_FFF*'` for SeicheSSV, `'*yyyymmdd_HHMMSS_FFF'` for PamGuard, `'*yyyymmdd_HHMMSS'` for Wildlife Acoustics SM4M.
- `timeOffset`: PC clock offset, in seconds (UTC = PC - `timeOffset`).
- `tags`: cell vector of tag names. These names will be available in the revision module to identify specific sections of the data.

### 13.1.4 Receiver Import Module

- `receiverCategory`: category of the receiver. The category relates to the type of information necessary to process the receiver. There are two categories available for receivers.
  - `'fixed'`: any fixed receiver with known position (e.g., moored buoy).
  - `'towed'`: any towed or automonous receiver (e.g., USV, drift buoy, towed hydrophone) with available position data (GPS, AIS, P190).
- `receiverName`: name of the receiver. This is used as a unique identifier.
- `receiverOffset`: two-element numeric array (x,y) representing the position of the receiver, in metres, relative to the position of the towinig platform for which position data is available.
- `receiverOffsetMode`: character vector indicating the nature of the physical connection between the receiver and the towing platform.
  - `'hard'`: the receiver remains at the same relative position from the towing platform at all times (e.g., hydrophone vertically-deployed from vessel).
  - `'soft'`: the receiver is loosely connected to the towing platform, changing its relative position as the former moves and changes course (e.g., vessel-towed hydrophone array).
- `positionPaths`: character vector or cell array of character vectors containing the relative directories and paths where the receiver position files are stored. The directories and paths are relative to `ROOT.POSITION` (see *root.json*).
- `positionFormat`: format of the receiver position files.
  - `'GPS'`: GPS format. Only extensions .gpstext (SeicheSSV GPS database) and .csv (PAMGuard exported table) are supported.
  - `'AIS'`: AIS format. Only .csv extension (PAMGuard exported table) is supported.
  - `'P190'`: P190 format. Only .p190 extension (seismic) is supported.
- `positionPlatform`: software platform used to produce the receiver position files. Combined with `positionFormat`, it helps determine the expected file extensions.
  - `'SeicheSsv'`: position file recorded with SeicheSSV software. It supports GPS (.gpstext).
  - `'PamGuard'`: position file recorded with PAMGuard. It supports GPS (.csv) and AIS (.csv).
  - `'Seismic'`: position file recorded with a seismic vessel. It only supports P190 (.p190).
- `vesselId`: identification number for vessel in a P190 file. Only for `positionFormat='P190'`
- `mmsi`: MMSI number of vessel. Only for `positionFormat='AIS'`.
- `latitude`: latitude, in degrees, of the towing platform (`receiverCategory = 'towed'`) or receiver itself (`receiverCategory = 'towed'` with `receiverOffset = [0 0]`, or `receiverCategory = 'fixed'`).

- `longitude`: longitude, in degrees, of the towing platform (`receiverCategory = 'towed'`) or receiver itself (`receiverCategory = 'towed'` with `receiverOffset = [0 0]`, or `receiverCategory` = `'fixed'`).
- `depth`: depth of the receiver, in metres. `depth` is a negative value.

### 13.1.5 Source Import Module

- `sourceCategory`: category of the source. The category relates to the type of information necessary to process the source. There are four categories available for sources.
  - `'fixed'`: any fixed source with known position (e.g., pile).
  - `'towed'`: any towed or autonomous source (e.g., vessel-towed airgun) with available position data (GPS, AIS, P190).
  - `'vessel'`: any vessel with available position data (GPS, AIS, P190).
  - `'fleet'`: multiple vessels with available position data (AIS).
- `sourceName`: name of the source. This is used as a unique identifier.
- `sourceOffset`: two-element numeric array (x,y) representing the position of the source, in metres, relative to the position of the towinig platform for which position data is available.
- `sourceOffsetMode`: character vector indicating the nature of the physical connection between the source and the towing platform.
  - `'hard'`: the source remains at the same relative position from the towing platform at all times (e.g., hydrophone vertically-deployed from vessel, side-mounted SBP).
  - `'soft'`: the source is loosely connected to the towing platform, changing its relative position as the former moves in different directions (e.g., vessel-towed airgun array).
- `positionPaths`: character vector or cell array of character vectors containing the relative directories and paths where the source position files are stored. The directories and paths are relative to `ROOT.POSITION` (see *root.json*).
- `positionFormat`: format of the source position files.
  - `'GPS'`: GPS format. Only the extensions .gpstext (SeicheSSV GPS database) and .csv (PAMGuard exported table) are supported.
  - `'AIS'`: AIS format. Only the .csv extension (PAMGuard exported table) is supported. Support for .aistext extension (SeicheSSV AIS database) will be added in a future release.
  - `'P190'`: P190 format. Only .p190 extension (seismic) is supported.
- `positionPlatform`: software platform used to produce the receiver position files. Combined with `positionFormat`, it helps determine the expected file extensions.
  - `'SeicheSsv'`: position file recorded with SeicheSSV software. It supports GPS (.gpstext).
  - `'PamGuard'`: position file recorded with PAMGuard. It supports GPS (.csv) and AIS (.csv).
  - `'Seismic'`: position file recorded with a seismic vessel. It only supports P190 (.p190).
- `vesselId`: identification number for vessel in a P190 file. Only for `positionFormat='P190'`
- `sourceId`: identification number for vessel in a P190 file. Only for `positionFormat='P190'`. `sourceId` does not need to be given for `sourceCategory = 'vessel'`.
- `mmsi`: MMSI number of the vessel. Only for `positionFormat = 'AIS'`.
- `vesselName`: name of the vessel. This is a unique identifier.

- `vesselLength`: length of the vessel, in metres.
- `vesselBeam`: width of the vessel, in metres.
- `vesselDraft`: draft of the vessel, in metres.
- `vesselGrossTonnage`: gross tonnage of the vessel, in tonnes.
- `latitude`: latitude, in degrees, of the towing platform (`sourceCategory = 'towed'`) or source itself (`sourceCategory = 'towed'` with `sourceOffset = [0 0]`, or `sourceCategory` = {`'fixed'`,`'vessel'`,`'fleet'`}).
- `longitude`: longitude, in degrees, of the towing platform (`sourceCategory = 'towed'`) or source itself (`sourceCategory = 'towed'` with `sourceOffset = [0 0]`, or `sourceCategory` = {`'fixed'`,`'vessel'`,`'fleet'`}).
- `depth`: depth of the receiver, in metres. `depth` is a negative value.

### 13.1.6 Navigation Process Module

- `receiverName`: name of the receiver to be processed.
- `sourceName`: name of the primary source to be processed.
- `smoothWindow`: time window used for averaging position information, in seconds. Using a `smoothWindow` of several seconds will help improve the accuracy of the navigation parameters. The slower the platform, the longer the averaging period (recommended 10-30 s).
- `maxTimeGap`: maximum time interval, in seconds, between two consecutive sentences for applying spatial interpolation to a sound event detected within that period. If the period exceeds `maxTimeGap` and a sound event is detected within it, the navigation parameters for that sound event are set to `NaN`.
- `interpMethod`: interpolation method used for calculating the position of a detected sound event. See `interp1` for available methods.

## 13.2 Audio Database

### 13.2.1 AudImpConfig

- `inputStatus`: TRUE if the audio import configuration is valid.
- `configFileName`: name of the audio import configuration file from which `AudImpConfig` comes from.
- `audioLength`: number of samples in the audio file.

See **Sub-Section 13.1.1** for a description of the rest of fields in `AudImpConfig`.

### 13.2.2 AudImpData

- `audioPath`: absolute path of audio file
- `audioData`: single-precision resampled audio data from `channel`.

## 13.3 Detection Database

### 13.3.1 RawScoreData

- `kernelDuration`: duration of the observations [s]

- `sampleRate`: new sampling rate for the observations [Hz]. No resampling is applied if equal to the sampling rate of the observations.
- `minSnrLevel`: minimum signal to noise ratio for the observations [dB]
- `snrLevels`: vector of signal to noise ratios of the observations [dB]
- `rawScoreMatrix`: normalised matrix of raw scores. It has as many rows as observations (audio segments) and as many columns as variables (audio samples). The values are single-precision.

### 13.3.2 CovarianceData

- `kernelDuration`: duration of raw score observations used to build the covariance matrix [s].
- `sampleRate`: sampling rate, after resampling, of the raw score observations used to build the covariance matrix [Hz].
- `covarianceMatrix`: normalised covariance matrix built from the raw scores *training* data. This is a square matrix with as many rows and columns as variables (audio samples). The values are single-precision.

### 13.3.3 PerformanceData

- `detectorType`: character vector specifying the type of Neyman-Pearson detector.
  - `'ed'`: energy detector. The signal and background noise are assumed to be White Gaussian Noise (WGN) processes.
  - `'ecw'`: estimator-correlator in white Gaussian noise. The signal is characterised by its covariance matrix and the background noise is considered a White Gaussian Noise (WGN) process.
  - `'ecc'`: estimator-correlator in coloured Gaussian noise. The signal and the noise are characterised by their respective covariance matrices. The background noise is a Coloured Gaussian Noise (CGN) process.
- `signalVariance`: variance of the signal.
- `noiseVariance`: variance of the background noise.
- `snrLevel`: selected SNR given by `10*log10(signalVariance/noiseVariance)`.
- `nVariables`: number of variables (samples) in the data segment (= `NDOF`).
- `cutoffFreqns`: two-element vector with the normalised cutoff frequencies of the detection filter. The values must be between `0` and `1`. `cutoffFreqns = 2*cutoffFreqs/resampleRate`, where cutoffFreqs and resampleRate are fields in the configuration file *audioDetectConfig_NeymanPearson.json*.
- `axisFalseAlarm`: test-statistic axis for `pdfFalseAlarm` and `rtpFalseAlarm`.
- `axisDetection`: test-statistic axis for `pdfDetection` and `rtpDetection`.
- `pdfFalseAlarm`: null hypothesis probability density function (PDF).
- `pdfDetection`: alternative hypothesis probability density function (PDF).
- `rtpFalseAlarm`: false alarm right-tail probability (RTP) function.
- `rtpDetection`: detection right-tail probability (RTP) function.

### 13.3.4 EigenData

- `kernelDuration`: duration of the observations used to build the covariance matrices of the target signal and noise, in seconds. The number of variables is `ROUND(kernelDuration *sampleRate)`.

- `sampleRate`: sample rate of the observations used to build the covariance matrices of the target signal and noise (`covarianceSignal`, `covarianceNoise`), in Hertz.
- `noiseType`: type of background noise. `noiseType = 'wgn'` for the estimator-correlator in white Gaussian noise (`'ecw'`) and `noiseType = 'cgn'` for the estimator-correlator in coloured Gaussian noise (`'ecc'`).
- `signalEigenVectors`: matrix of eigenvectors, organised in columns. For `'ecw'`, this is the modal matrix of the covariance matrix of the target signal `covarianceSignal`. For `'ecc'`, `signalEigenVectors` is the modal matrix of `B = A'*covarianceSignal*A`.
- `signalEigenValuesNorm`: vector of normalised eigenvalues of the eigenvectors in `signalEigenVectors`. For `'ecw'`, this is the vector of normalised eigenvalues of `covarianceSignal`. For `'ecc'`, this is the vector of normalised eigenvalues of `B = A'* covarianceSignal*A`.
- `noiseEigenVectors`: matrix of eigenvectors, organised in columns. For `'ecw'`, `'noiseEigenVectors' = []`. For `'ecc'`, `noiseEigenVectors` is the modal matrix of `covarianceNoise`.
- `noiseEigenValuesNorm`: vector of normalised eigenvalues of the eigenvectors in `noiseEigenVectors`. For `'ecw'`, `'noiseEigenValuesNorm' = []`. For `'ecc'`, this is the vector of normalised eigenvalues of `covarianceNoise`.

## 13.4 Navigation Database

### 13.4.1 General Fields

- `receiverList`: cell array of all receiver names stored in the database.
- `sourceList`: cell array of all source names stored in the database.
- `vesselList`: cell array of all vessel names stored in the database.

### 13.4.2 RecImpConfig

- `inputStatus`: `true` if the audio import configuration is valid.
- `configFileName`: name of the receiver import configuration file '*receiverImportConfig*.json*' from which `RecImpConfig` comes from.

See **Sub-Section 13.1.4** for a description of the rest of fields in `RecImpConfig`.

### 13.4.3 RecImpData

- `positionPaths`: absolute paths of the receiver position files.
- `pcTick`: vector of PC times of position sentences. Not available on P190 files.
- `utcTick`: vector of UTC times of position sentences.
- `latitude`: receiver latitude, in degrees. May refer to the towing platform or receiver itself.
- `longitude`: receiver longitude, in degrees. May refer to the towing platform or receiver itself.
- `depth`: depth of the receiver, in metres.

### 13.4.4 SouImpConfig & VesImpConfig

- `inputStatus`: `true` if the audio import configuration is valid.

- `configFileName`: name of the receiver import configuration file '*sourceImportConfig*.json*' from which `SouImpConfig` or `VesImpConfig` from.

See **Sub-Section 13.1.5** for a description of the rest of fields in `SouImpConfig` and `VesImpConfig`.

### 13.4.5 SouImpData & VesImpData

- `positionPaths`: absolute paths of the source position files.
- `pcTick`: vector of PC times of position sentences. Not available on P190 files.
- `utcTick`: vector of UTC times of position sentences.
- `latitude`: source latitude, in degrees. May refer to the towing platform or receiver itself.
- `longitude`: source longitude, in degrees. May refer to the towing platform or source itself.
- `depth`: depth of the source, in metres.

## 13.5 Acoustic Database

### 13.5.1 AcoConfig

#### General Fields

- `audiodbName`: name of the Audio Database on which the Acoustic Database is based.
- `channel`: channel of the Audio Database on which the Acoustic Database is based.
- `resampleRate`: sampling rate (after resampling) of the Audio Database on which the Acoustic Database is based [Hz]
- `receiverName`: name of the receiver for the current element of `AcoConfig`.
- `sourceName`: name of the source for the current element of `AcoConfig`.

#### AudImpConfig

- `inputStatus`: TRUE if the audio import configuration is valid.
- `configFileName`: name of the audio import configuration file from which `AudImpConfig` comes from.
- `audioLength`: number of samples in the audio file.

See **Sub-Section 13.1.1** for a description of the rest of fields in `AudImpConfig`.

#### RecImpConfig

- `inputStatus`: `true` if the audio import configuration is valid.
- `configFileName`: name of the receiver import configuration file '*receiverImportConfig*.json*' from which `RecImpConfig` comes from.

See **Sub-Section 13.1.4** for a description of the rest of fields in `RecImpConfig`.

#### SouImpConfig

- `inputStatus`: `true` if the audio import configuration is valid.
- `configFileName`: name of the receiver import configuration file '*sourceImportConfig*.json*' from which `SouImpConfig` comes from.

See **Sub-Section 13.1.5** for a description of the rest of fields in `SouImpConfig`.

VesImpConfig

- `inputStatus`: `true` if the audio import configuration is valid.
- `configFileName`: name of the receiver import configuration file '*sourceImportConfig*.json*' from which `VesImpConfig` comes from.

See **Sub-Section 13.1.5** for a description of the rest of fields in `VesImpConfig`.

AudDetConfig

- `inputStatus`: `true` if the audio detect configuration is valid.
- `configFileName`: name of the audio detect configuration file from which `AudDetConfig` comes from.
- `channel`: channel to be processed.
- `resampleRate`: sample rate (after resampling) of the audio data [Hz]

See **Sub-Section 13.1.2** for a description of the rest of fields in `AudDetConfig`.

AudProConfig

- `inputStatus`: `true` if the audio detect configuration is valid.
- `configFileName`: name of the audio detect configuration file from which `AudProConfig` comes from.
- `channel`: processed channel.
- `resampleRate`: sample rate (after resampling) of the audio data [Hz]

See **Sub-Section 13.1.3** for a description of the rest of fields in `AudProConfig`.

NavProConfig

- `inputStatus`: `true` if the audio detect configuration is valid.
- `configFileName`: name of the audio detect configuration file from which `NavProConfig` comes from.
- `channel`: processed channel.
- `resampleRate`: sample rate (after resampling) of the audio data [Hz]

See **Sub-Section 13.1.6** for a description of the rest of fields in `NavProConfig`.

### 13.5.2 AcoData

General Fields

- `audiodbName`: name of the Audio Database on which the Acoustic Database is based.
- `channel`: channel of the Audio Database on which the Acoustic Database is based.
- `resampleRate`: sampling rate (after resampling) of the Audio Database on which the Acoustic Database is based [Hz]
- `receiverName`: name of the receiver for the current element of `AcoData`.
- `sourceName`: name of the source for the current element of `AcoData`.

AudDetData

- `signalTime`: vector of exact times for the detections, in seconds referred to the beginning of the audio file. The reference point for the exact time can be the highest pulse energy (for `detector = 'MovingAverage'`) or the centre of the detection window.
- `signalTime1`: vector of times for the start of the detection windows, in seconds referred to the beginning of the audio file.
- `signalTime2`: vector of times for the end of the detection windows, in seconds referred to the beginning of the audio file.
- `noiseTime1`: vector of times for the start of the noise windows, in seconds referred to the beginning of the audio file.
- `noiseTime2`: vector of times for the end of the noise windows, in seconds referred to the beginning of the audio file.

AudProData

- `signalPcTick`: vector of detection PC ticks, in seconds referred to '00 Jan 0000'. `signalPcTick = fileTick + signalTime`.
- `signalUtcTick`: vector of detection PC ticks, in seconds referred to '00 Jan 0000'. `signalUtcTick = fileTick + signalTime - timeOffset`.
- `timeOffset`: vector of time offsets at each detection instant. `timeOffset = signalPcTick - signalUtcTick`.
- `fileTimestamp`: cell array of `fileTick` timestamps with format `'yyyy-mmm-dd HH:MM:SS.FFF'`.
- `fileTick`: tick of the start of the audio file, in seconds referred to '00 Jan 0000'. The value is computed with `audioFileTick` from the name of the audio file, which contains the timestamp of its first sample.
- `signalTime1`: vector of times for the start of the detection windows, in seconds referred to the beginning of the audio file.
- `signalTime2`: vector of times for the end of the detection windows, in seconds referred to the beginning of the audio file.
- `signalEnergyTime1`: vector of times for the start of the cumulative energy window for the detections, in seconds referred to the beginning of the audio file.
- `signalEnergyTime2`: vector of times for the end of the cumulative energy window for the detections, in seconds referred to the beginning of the audio file.
- `noiseTime1`: vector of times for the start of the noise windows, in seconds referred to the beginning of the audio file.
- `noiseTime2`: vector of times for the end of the noise windows, in seconds referred to the beginning of the audio file.
- `noiseEnergyTime1`: vector of times for the start of the cumulative energy window for the background noise, in seconds referred to the beginning of the audio file.
- `noiseEnergyTime2`: vector of times for the end of the cumulative energy window for the background noise, in seconds referred to the beginning of the audio file.
- `signalZ2p`: vector of zero-to-peak amplitudes of detections, calculated over the cumulative energy windows. Single-precision values ranging from `-2^(nbits-1)` to `2^(nbits-1)-1`, where `nbits` is the bit depth of the ADC.

- `signalP2p`: vector of peak-to-peak amplitudes of detections, calculated over the cumulative energy windows. Single-precision values ranging from `-2^(nbits-1)` to `2^(nbits-1)-1`, where `nbits` is the bit depth of the ADC.
- `signalRms`: vector of broadband RMS amplitudes of detections, calculated over the cumulative energy windows. Single-precision values ranging from `-2^(nbits-1)` to `2^(nbits-1)-1`, where `nbits` is the bit depth of the ADC.
- `signalExp`: vector of broadband exposures of detections, calculated over the cumulative energy windows. Single-precision values ranging `-2^(nbits-1)` to `2^(nbits-1)-1`, where `nbits` is the bit depth of the ADC.
- `noiseZ2p`: vector of zero-to-peak background noise amplitudes, calculated over the cumulative energy windows. Single-precision values ranging from `-2^(nbits-1)` to `2^(nbits-1)-1`, where `nbits` is the bit depth of the ADC.
- `noiseP2p`: vector of peak-to-peak background noise amplitudes, calculated over the cumulative energy windows. Single-precision values ranging from `-2^(nbits-1)` to `2^(nbits-1)-1`, where `nbits` is the bit depth of the ADC.
- `noiseRms`: vector of broadband backgroud noise RMS amplitudes, calculated over the cumulative energy windows. Single-precision values ranging from `-2^(nbits-1)` to `2^(nbits-1)-1`, where `nbits` is the bit depth of the ADC.
- `noiseExp`: vector of broadband background noise exposures, calculated over the cumulative energy windows. Single-precision values ranging from `-2^(nbits-1)` to `2^(nbits-1)-1`, where `nbits` is the bit depth of the ADC.
- `signalRmsBand`: array of band RMS amplitudes of detections, calculated over the cumulative energy windows. As many rows as frequency bands and as many columns as detections. Single-precision values ranging from `-2^(nbits-1)` to `2^(nbits-1)-1`, where `nbits` is the bit depth of the ADC.
- `signalExpBand`: array of band exposures of detections, calculated over the cumulative energy windows. As many rows as frequency bands and as many columns as detections. Single-precision values ranging from `-2^(nbits-1)` to `2^(nbits-1)-1`, where `nbits` is the bit depth of the ADC.
- `noiseRmsBand`: array of band background noise RMS amplitudes, calculated over the cumulative energy windows. As many rows as frequency bands and as many columns as detections. Single-precision values ranging from `-2^(nbits-1)` to `2^(nbits-1)-1`, where `nbits` is the bit depth of the ADC.
- `noiseExpBand`: array of band background noise exposures, calculated over the cumulative energy windows. As many rows as frequency bands and as many columns as detections. Single-precision values ranging from `-2^(nbits-1)` to `2^(nbits-1)-1`, where `nbits` is the bit depth of the ADC.
- `nominalFreq`: vector of nominal band frequencies, in Hertz.
- `centralFreq`: vector of central band frequencies, in Hertz.
- `cutoffFreq1`: vector of band low cutoff frequencies, in Hertz.
- `cutoffFreq2`: vector of band high cutoff frequencies, in Hertz.

RecProData

- `pcTick`: vector of PC ticks, in seconds referred to '00 Jan 0000'.

- `utcTick`: vector of UTC ticks, in seconds referred to '00 Jan 0000'.
- `latitude`: vector of receiver latitudes, in degrees.
- `longitude`: vector of receiver longitudes, in degrees.
- `depth`: receiver depth, in metres.
- `course`: vector of receiver courses, in degrees (0 N, 90 E)
- `speed`: vector of receiver speeds, in m/s.

SouProData

- `pcTick`: vector of PC ticks, in seconds referred to '00 Jan 0000'.
- `utcTick`: vector of UTC ticks, in seconds referred to '00 Jan 0000'.
- `latitude`: vector of source latitudes, in degrees.
- `longitude`: vector of source longitudes, in degrees.
- `depth`: source depth, in metres.
- `course`: vector of source courses, in degrees (0 N, 90 E)
- `speed`: vector of source speeds, in m/s.
- `sou2recDistance`: source to receiver distance, in metres.
- `sou2recBearing`: source to receiver bearing, in degrees (0 N, 90 E)
- `sourceHeading`: source heading, in degress (0 N, 90 E)
- `sourceEmitAngle`: source directivity angle, in degrees (0 N, 90 E)

VesProData

- `pcTick`: vector of PC ticks, in seconds referred to '00 Jan 0000'.
- `utcTick`: vector of UTC ticks, in seconds referred to '00 Jan 0000'.
- `latitude`: vector of vessel latitudes, in degrees.
- `longitude`: vector of vessel longitudes, in degrees.
- `depth`: vessel depth, in metres.
- `course`: vector of vessel courses, in degrees (0 N, 90 E)
- `speed`: vector of vessel speeds, in m/s.
- `sou2recDistance`: vessel to receiver distance, in metres.
- `sou2recBearing`: vessel to receiver bearing, in degrees (0 N, 90 E)
- `sourceHeading`: vessel heading, in degress (0 N, 90 E)
- `sourceEmitAngle`: vessel directivity angle, in degrees (0 N, 90 E)

RevData

- `idet`: indices of unclassified detections.
- `ival`: indices of valid detections.
- `iwro`: indices of wrong detections.
- `isat`: indices of saturating detections.
- `imrk1`: index of left marker of selection window.
- `imrk2`: index of right marker of selection window.

- `showSelec`: `true` if selection is shown.
- `showBckgnd`: `true` is background noise is shown
- `showThres`: `true` if amplitude threshold selection window is shown.
- `threshold`: absolute amplitude threshold.
- `thresDir`: direction of the vertical selection area referred to the threshold.
- `snapOpt`: `true` for the selection bar to move only to valid detections when pressing the 'next' or 'previous' buttons.
- `playTimeOpt`: `true` if play duration is enabled
- `playTime`: play duration, in seconds
- `playSpeed`: play speed by original speed ratio (e.g., 2 for twice the speed)
- `soundType`: type of sound ('transient', 'continuous')
- `ipropName`: index of property name.
- `ipropValue`: index of property value
- `graph1`: string specifying the type of graph for graph 1.
- `graph2`: string specifying the type of graph for graph 2.
- `graph3`: string specifying the type of graph for graph 3.
- `fmax`: maximum frequency to display, in Hertz.
- `rmax`: maximum range to display in the 'Operations Map' graph.
- `showOpsTxt`: `true` if receiver, source and vessel names are to be shown in the Operations Map
- `comments`: cell array of comments for each detection.

# 14 References

Kay, S. M. (1998). *Fundamentals of statistical signal processing. Vol 2: Detection theory.* 1st edn. New Jersey: Prentice Hall.

Ledoit, O., and Wolf, M. (2003a) "Honey, I shrunk the sample covariance matrix", UPF Economics and Business Working Paper No. 691. doi: 10.2139/ssrn.433840

Ledoit, O., and Wolf, M. (2003b) "Improved estimation of the covariance matrix of stock returns with an application to portfolio selection", *Journal of Empirical Finance*, **10**, 603-621. doi: 10.1016/S0927-5398(03)00007-0

Ledoit, O., and Wolf, M. (2004) "A well-conditioned estimator for large-dimensional covariance matrices", *Journal of Multivariate Analysis*, **88**(2), 365-411. doi: 10.1016/S0047-259X(03)00096-4

Chen, Y., Wiesel, A., Eldar, Y. C., and Hero, A. O. (2010) "Shrinkage algorithms for MMSE covariance estimation", *IEEE Transactions On Signal Processing*, **58**(10), 5016-5029. doi: 10.1109/TSP.2010.2053029

Hoffbeck, J. P., and Landgrebe, D. A. "Covariance matrix estimation and classification with limited training data", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **18**(7), 763-767. doi: 10.1109/34.506799

Theiler, J. (2012) "The incredible shrinking covariance estimator". In: SPIE Defense Security and Sensing Symposium, 23rd April 2012, Baltimore, MD (USA), pp. 12.

ANSI (2004). *ANSI S1.11: Specification for Octave, Half-Octave, and Third Octave Band Filter Sets*. Melville, NY (USA). American National Standards Institute (ANSI) [online]. Available at: https://ansi.org/

BSI (2014). *BS EN 61260-1:2014 Electroacoustics – Octave-band and fractional-octave-band filters. Part 1: Specifications*. British Standards Institution (BSI) [online]. Available at: https://webstore.ansi.org/Standards/BSI/BSEN612602014