Applied Acoustics 116 (2017) 50–58

Contents lists available at ScienceDirect

Applied Acoustics

journal homepage: www.elsevier.com/locate/apacoust

A Python framework for microphone array data processing

Ennes Sarradj [a,*], Gert Herold [b]

[a] Institute of Fluid Mechanics and Engineering Acoustics, Technische Universität Berlin, 10587 Berlin, Germany
[b] Chair of Technical Acoustics, Brandenburg University of Technology, 03046 Cottbus, Germany

ARTICLE INFO

Article history:
Received 4 May 2016
Received in revised form 25 August 2016
Accepted 12 September 2016

Keywords:
Microphone array
Beamforming
Python
Software

ABSTRACT

Acoular is an open source object-oriented Python package for microphone array data processing. It supports various methods for sound source characterization and mapping. The background of these methods, which rely on synchronously captured microphone signals, is shortly introduced, and the requirements for a software that implements these methods are discussed. The object-oriented design based on Python allows for easy-to-use scripting and graphical user interfaces, the practical combination with other data handling and scientific computing libraries, and the possibility to extend the software by implementing new processing methods with minimal effort. Built-in result caching and fast C++ based parallelized implementation of core routines is explained. Together with data handling procedures that can accommodate the huge amounts of measured data needed, this makes the application of Acoular to industrial-scale problems possible. Basic examples of Acoular use and extension are given.

1. Introduction

The design of low-noise machinery and vehicles requires analyzing the sources of sound. Information on the characteristics of any sound sources is necessary to find measures to reduce the generation of sound or its propagation. This includes the location and strength of the sources as well as their frequency content. Often, this information is only available through experimental analysis, either on the machine or vehicle itself or on specially designed laboratory setups of noise generating machinery parts. The necessary acoustical measurements can generally be performed using standard equipment such as a microphone, sound level meter or analyzer. However, this approach makes it difficult to reliably characterize sound sources in the case of multiple sound sources, which is a very common scenario.

In such a case, the results are dominated by the strongest source and expensive experimental procedures are needed to get separate results for each source. One solution for this problem is the application of a microphone array, where a number – some ten to some hundred – of microphones is used simultaneously to characterize multiple sound sources at the same time. This is done by computing acoustic source maps (often referred to as acoustic photographs) from the output signals of the microphones. Then, location, strength, and spectrum of the sources can be estimated from these maps.

A number of different methods are available for the computation of acoustic source maps. These methods either rely on the direct simultaneous processing of a large number of time-dependent microphone signals, or they perform the computation in the frequency domain after having transformed the signals accordingly into cross power spectra. Both kinds of methods are computationally demanding and require considerable computer resources. Some of the methods need to solve huge systems of equations, while others need to deal with large-scale optimization problems. The methods have different properties and deliver results of different kind and quality. Depending on the specific acoustic source characterization task, different methods may be appropriate. Consequently, the practical application of microphone arrays benefits from the uncomplicated availability of different methods.

While a larger number of publications on the methods themselves is available, the implementation of the methods is less often discussed. To the knowledge of the authors, no software is publicly available to date that implements more than a few of these methods. The available commercial software products are generally bound to a specific vendor's measuring and data acquisition hardware. Moreover, available software codes are not focused on the extensibility with new or modified methods.

The present contribution introduces Acoular, an open source Python library [1] that was published under the terms of the

* Corresponding author.
E-mail addresses: ennes.sarradj@tu-berlin.de (E. Sarradj), gert.herold@b-tu.de (G. Herold).

http://dx.doi.org/10.1016/j.apacoust.2016.09.015
0003-682X/© 2016 Elsevier Ltd. All rights reserved.

new BSD license and is available for all major operating systems. The library is aimed at applications in acoustic testing where sources of sound need to be characterized. Its design is object-oriented and it is intended to be easily extensible with the incorporation of new methods. Further design goals are computational efficiency and easy application. This concept makes it possible to apply the library for education and for research on methods for sound source characterization. Further, Acoular can be used to efficiently handle applications of industrial size, where larger numbers of measurements need to be analyzed.

The remainder of the paper is organized as follows. First, the theoretical background of the microphone array methods implemented in Acoular is briefly introduced. Second, the object-oriented design and operation of Acoular is addressed. Third, the application of Acoular is demonstrated using some examples. Finally, it is explained how a new method can be introduced into Acoular.

2. Background

Consider the scenario shown in Fig. 1, where a microphone array consisting of N microphones is used to analyze a sound field that is produced by an arbitrary number of sound sources. The sound will need a certain time $\Delta\tau_{mn}$ to travel from source m to microphone n, and its amplitude will be changed by a certain factor a_{mn}. Both will depend on the distance between source and microphone and other factors such as the speed of sound and presence of flow. Because linear superposition can be assumed, the sound pressure at the microphone is the sum of contributions from all sources:

$$p_n(t) = \sum_m a_{mn} q_m(t - \Delta\tau_{mn}). \tag{1}$$

The quantity q_m stands for a measure of source strength such as the flux of a monopole source. As long as both source strength and position of the sources are known, the calculation of all p_n is straightforward. The characterization of sound sources from microphone array measurements represents the inverse problem: to estimate the source strength and position of the sources from the measured p_n.

One possibility to achieve this is to calculate a weighted sum of the delayed and attenuated microphone signals, as shown in Fig. 1:

$$p_m = \sum_n h_{mn} p_n(t + \Delta t_{mn}). \tag{2}$$

Here, the idea is to choose h_{mn} and Δt_{mn} in such a way that the output p_{out} will contain the signal q_m from source m while any other source signals will be suppressed as much as possible. This can be seen as a spatial filter, and there are a number of options to calculate the filter coefficients h_{mn} and Δt_{mn} [2] from a_{mn} and τ_{mn}. Because of the calculation procedure in (2), this approach is called Delay-and-Sum Beamforming. The characterization of multiple sources with unknown positions is possible when the procedure is applied for each possible source position in a grid of source positions (see Fig. 1) in turn. In order to get an acoustic photograph, it is then convenient to calculate the power $\langle p^2_{out}\rangle_T$ over a certain time interval T and map this quantity onto the grid. This can be done for arbitrary kinds of grids including such that are irregular or three-dimensional.

The principle from (2) can be extended in many different ways. Instead of using fixed filter coefficients that depend on the sound propagation model only, it is possible to use variable coefficients that adapt the spatial filter and depend also on the signals p_n themselves [3]. Another option, that can be used if the sound sources are moving on a known trajectory, is to treat the filter coefficients as functions of time [4,5]. Very often (2) is combined with subsequent frequency filtering. With a bandpass filter applied for each frequency band of interest, the method then also allows to estimate the frequency spectrum of the sound sources. In a practical application the microphone signals are containing additional noise not originating from the sound sources. The influence of this noise on the result can be considerably reduced by the following modification of (2):

$$p^2_m(t) = \max\left(\left\langle\left(\sum_n h_{mn} p_n(t+\Delta t_{mn})\right)^2 - \sum_n \left(h_{mn} p_n(t+\Delta t_{mn})\right)^2\right\rangle_T, 0\right). \tag{3}$$

Besides Delay-and-Sum Beamforming in the time domain, a second possibility to estimate the source strength and position of the sources from the measured p_n is analysis in the frequency domain. To this end, the cross spectral matrix of the microphone signals is estimated using Welch's method [6]. All microphone signals are partitioned into n_d blocks of a certain number of samples. These blocks are then Fourier-transformed. An estimate of a matrix containing the N^2 cross power spectra of all possible pairs of microphones is then

$$\mathbf{G}(f_k) = 2\frac{1}{n_d T}\sum_{i=1}^{n_d} \mathbf{p}_i(f_k)\mathbf{p}_i^*(f_k), \tag{4}$$

where $\mathbf{p}(f_k)$ denotes the vector of the values of the Fourier-transform at the frequency f_k for all microphone signals. The cross spectral matrix $\mathbf{G}$ can then be used as a basis to perform the spatial filtering in frequency domain:

$$p^2_m = \mathbf{h}^H_m \mathbf{G} \mathbf{h}_m. \tag{5}$$

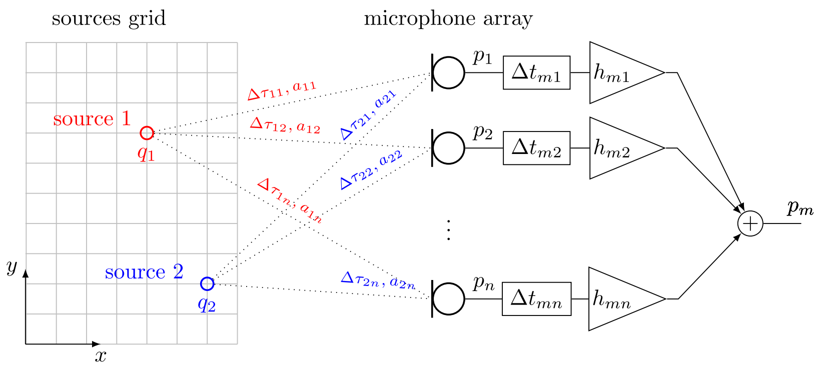

sources grid
microphone array
source 1
q_1
source 2
q_2
y
x
$\Delta\tau_{11}, a_{11}$
$\Delta\tau_{12}, a_{12}$
$\Delta\tau_{1n}, a_{1n}$
$\Delta\tau_{21}, a_{21}$
$\Delta\tau_{22}, a_{22}$
$\Delta\tau_{2n}, a_{2n}$
p_1
p_2
p_n
Δt_{m1}
Δt_{m2}
Δt_{mn}
h_{m1}
h_{m2}
h_{mn}
p_m

Fig. 1. Working principle of beamforming algorithms.

The complex-valued steering vectors $\mathbf{h}_m$ now contain the h_{mn} and Δt_{mn} transformed into frequency domain and H stands for the hermitian transpose. While this is fully equivalent to (2), it allows for a generally more efficient calculation of frequency-dependent acoustic source maps. The steering vector $\mathbf{h}_m$ has to be computed again for each possible source position, but $\mathbf{G}$ needs to be computed only once. More important, the frequency domain approach is the basis for a number of more advanced methods. Among those are methods that make use of an eigen-decomposition of the cross spectral matrix and process the noise space [7] or work with individual eigenvalue/eigenvector pairs separately [8].

Both time and frequency domain spatial filters have imperfections that lead to imperfect acoustic source maps with blurred mappings that may also contain 'ghost' images of actual sources. In many practical situations, this makes it impossible to obtain an accurate source characterization from that approach. A group of methods (e.g. [9–12]) aim to remove the influence of filter imperfections from the source maps. Because the imperfect maps can be seen as a possibly multidimensional convolution of the perfect map with a point spread function that contains the filter imperfections, these methods are referred to as deconvolution methods. Deconvolution methods are computationally demanding, with some methods having very high computational cost.

The last group of methods to be mentioned here attempts to solve the inverse problem

$$\mathbf{G} = \mathbf{A}\mathbf{D}\mathbf{A}^H \tag{6}$$

directly, where $\mathbf{G}$ is the known cross spectral matrix, $\mathbf{A}$ is a matrix of the dimension $N \times M$ that contains the transfer functions from each of the M possible sources to each of the microphones and $\mathbf{D}$ is a diagonal matrix containing the unknown source powers. This is effectively the same as the model in (1). The methods (e.g. [13–15]) dealing with that problem do so without spatial filtering, but apply different solution strategies for the inverse problem that prefer sparse solutions, where only few of the possible source positions are actually occupied by sources.

A great number of further methods are not mentioned here, but do also fall in one of the categories: time domain approach filter methods, frequency domain filter methods, deconvolution methods or inverse methods. It is obvious that the methods must have some computing steps and requirements in common. Once implemented for a certain method, these steps and requirements may be reused for other methods.

3. Design and operation

The design of Acoular is based on some requirements that are identified and briefly discussed here.

1. There are already many different methods for microphone array data analysis available. They may be improved, and new methods may be developed in the future. It is desirable to take advantage of these improved and new methods with minimal effort. Thus, the implementation of new features and methods should reuse as much as possible of the existing code. One way to make that possible is the use of an object-oriented design approach, as detailed below.
2. Practical applications require the use of further software such as plotting and reporting, database storage or data acquisition tools. Easy integration of the analysis of microphone array data with these tools is important and enables complex applications such as the automated processing of multiple measurements. This is supported by using a script language such as Python, with bindings to many efficient data handling and scientific computing libraries available.
3. The user interface should hide implementation details and allow the simple application of the library within user-generated scripts. Thus, at least the interface should be written in an easy to use script language. This is also addressed by using Python.
4. A microphone array measurement may produce huge amounts of data. For example, an array of 64 microphones with the output signals sampled at a rate of 102.4 kHz and digitized as 32 bit floating point number produces 25 MByte of data per second. For a measurement duration of some minutes this already amounts to some Gigabytes. Consequently, the software should be able to handle input data of such size. Because the data could possibly not fit into the main memory, the software design should be capable to work out-of-core. All algorithms are therefore implemented to sequentially process chunks of the data only that fit into the main memory.
5. Depending on the algorithm and data, the computations necessary can be costly and take a lot of time. It should therefore be possible to use efficient and fast implementations of the time-consuming part of the algorithms and to work on multiple cores in parallel. Besides the use of relatively slow Python for the majority of the code, core parts of the algorithms are implemented in C++ and optimized for speed. Parallelism is realized using OpenMP.
6. Also because of the necessary computational effort, it is sensible to allow the reuse of intermediate and final results. A caching mechanism is therefore implemented that provides the persistence of results between individual runs and makes it unnecessary to repeat computations that were already done before.
7. Computations not necessary for the final result should be avoided because of the computational effort. To this end, a lazy evaluation paradigm was used that triggers a computation or data fetching process only when the result is actually needed.
8. Because some intended applications require frequent user interaction, the software should support the access to a graphical user interface. This requires to provide all information through the interface that is necessary to edit any parameters in a graphical user interface. Acoular makes use of the Enthought Traits library [16], that allows to define a graphical user interface for each object by simply specifying the type of the editable attributes.

The basic design principle of Acoular is to implement all microphone array methods based on a set of common building blocks. For each of these building blocks, a common interface is defined using a base class to define its interface. Then, different algorithms or sub-methods are implemented in subclasses derived from the base class. This design principle is also known as Strategy pattern [17]. The main classes and their relations are shown in Fig. 2. Some of these classes are abstract classes that do not implement actual algorithms, but only the interface.

A class derived from SamplesGenerator implements a mechanism to provide the sampled microphone data time histories. This mechanism uses the Iterator strategy [17] to be able to deal with huge, practically infinite time histories. The SamplesGenerator interface exhibits a Python generator representing an external iterator that produces blocks of data with a limited number of samples at a time. Current implementations of SamplesGenerator do either process stored data from measurements or data that is produced by Acoular itself. For the latter, Acoular contains a facility to simulate multiple sound source scenarios not detailed here for the sake of brevity.

Possible clients for the generator in the SamplesGenerator interface are instances of a subclass of TimeInOut or of the class PowerSpectra and its subclasses. TimeInOut defines an interface

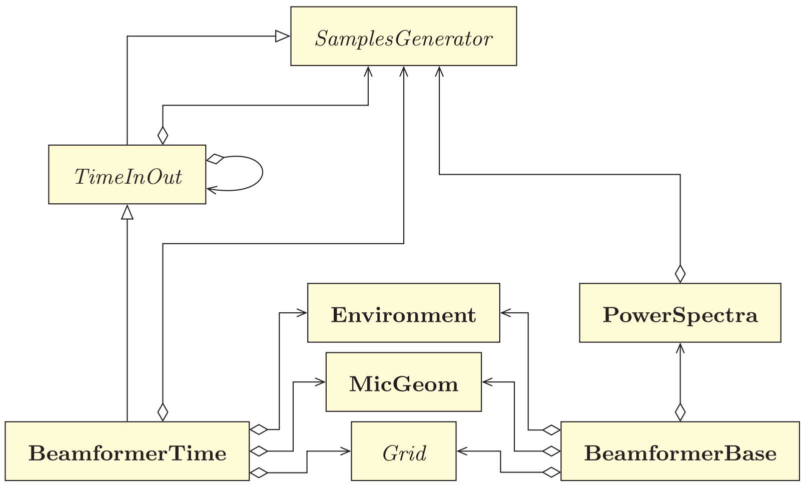


Fig. 2. UML class diagram showing the relation of main Acoular classes.

for a class that implements any form of processing the microphone signals in time domain. While it acts as a client consuming the data provided by a SamplesGenerator instance, it also provides the same generator interface as SamplesGenerator. This way, it is possible to chain a number of instances of TimeInOut subclasses to realize different subsequent operations on the data. Besides filtering, squaring and averaging, Acoular also implements time domain beamforming using a subclass derived from TimeInOut. Fig. 3 shows an example for a chain of TimeInOut objects performing Delay-and-Sum Beamforming, frequency filtering, power calculation and block-wise averaging to compute a time-dependent acoustic mapping or ‘acoustic movie’. More classes are available that implement not only (2), but also (3) as well as time dependent steering vectors.

The class PowerSpectra and its subclasses compute the cross spectral matrix **G**. Objects of this type have to consume the complete multichannel time history that is provided by a SamplesGenerator instance before the result is available. This result is then used by an instance of BeamformerBase or a subclass to compute the resulting map of sound sources using a variant of (5), a deconvolution method or an inverse method solving (6).

All methods that compute acoustic source maps in either time or frequency domain need some additional information. First, the propagation model necessary to compute τ_{mn} and a_{mn} in (1) is provided by the class Environment. Sound propagation in quiescent media, uniform and arbitrary flow is handled by subclasses. Second, the coordinates of the microphones are provided by the class MicGeom or specialized subclasses of it. Third, the Grid class provides the coordinates of all points or possible sound sources that should be considered in the source map. Subclasses are available for rectangular two- and three-dimensional grids.

Some methods do need additional information, such as the trajectory in case of moving sources. This information is also provided using specialized classes not detailed here. Table 1 shows an overview of all classes available in Acoular and a short description of their functionality.

4. Examples

4.1. Three monopole sound sources

The first example uses a planar 64-microphone array with an aperture of 38 cm (see Fig. 4) to analyze the sound produced by three monopole sources with different strengths (Table 2). The input data for the analysis are the (synthesized) sampled time histories for the sound pressure at all microphone locations, stored in an HDF5 [18] file. In order to do the analysis and produce a source map of the three sources, the time histories have to be read and used to compute the cross spectral matrix. Afterwards, beamforming is to be applied to produce results on a previously defined grid. The location of the microphones also needs to be known. As explained before, all necessary information for the computation is provided by instances of the classes (objects) defined in Acoular.

The following Python script assembles the objects needed to perform beamforming in the frequency domain for the example case:

```python
import acoular as ac
ts = ac.TimeSamples( name = 'three_sources.h5' )
ps = ac.PowerSpectra( time_data = ts, block_size = 128,
                      window = 'Hanning', overlap = '50%' )
mg = ac.MicGeom( from_file = 'array_64.xml' )
rg = ac.RectGrid( x_min = -0.2, x_max = 0.2, y_min = -0.2,
                  y_max = 0.2, z = 0.3, increment = 0.01 )
bb = ac.BeamformerBase( freq_data = ps, mpos = mg,
                        grid = rg, c = 343.0 )
```

Here, after importing Acoular into Python, the source of the input data is specified by instantiating a TimeSamples object and giving the name of the HDF5 file. Then, the parameters for the calculation of the cross spectral matrix are set by creating a PowerSpectra object that also knows about the previously defined TimeSamples object as a source of the microphone time histories.

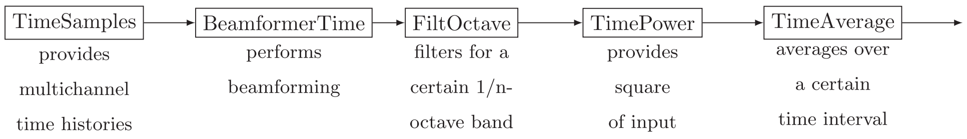

TimeSamples
provides multichannel time histories
BeamformerTime
performs beamforming
FiltOctave
filters for a certain 1/n-octave band
TimePower
provides square of input
TimeAverage
averages over a certain time interval

Fig. 3. Example for a chain of objects that are instances of TimeInOut subclasses, working on multichannel time histories that may have thousands of channels.

Table 1
Classes in Acoular, indentation indicates subclasses.

Class	Description
SamplesGenerator	Base class for any signal generating block
TimeSamples	Container for time data in ∗.h5 format
MaskedTimeSamples	…with sample and channel masking facilities
PointSource	Defines a fixed point source with an arbitrary signal
MovingPointSource	…for a source moving along a given trajectory
TimeInOut	Base class for any time domain signal processing block
Mixer	Mixes the signals from several sources
TimePower	Calculates time-depended power of the signal
TimeAverage	Calculates time-depended average of the signal
TimeReverse	Calculates the time-reversed signal of a source
FiltFiltOctave	Octave or third-octave filter with zero phase delay
FiltOctave	Octave or third-octave filter (non-zero phase delay)
TimeCache	Caches time signal in cache file
WriteWAV	Saves time signal as mono/stereo/multi-channel ∗.wav
WriteH5	Saves time signal as ∗.h5 file
BeamformerTime	Basic time domain beamformer
BeamformerTimeTraj	Time domain beamformer for moving grid
BeamformerTimeSq	Time domain beamformer with possible autopower removal
BeamformerTimeSqTraj	…for moving grid
IntegratorSectorTime	Integrator in the time domain
PowerSpectra	Provides the cross spectral matrix of multichannel time data
EigSpectra	Provides the eigendecomposition of cross spectral matrix
BeamformerBase	Basic beamforming in the frequency domain
BeamformerFunctional	Functional beamforming
BeamformerCapon	Beamforming using the Capon (Mininimum Variance) algorithm
BeamformerEig	Beamforming using eigenvalue and eigenvector techniques
BeamformerMusic	Beamforming using the MUSIC algorithm
BeamformerClean	CLEAN deconvolution
BeamformerDamas	DAMAS deconvolution
BeamformerOrth	Orthogonal beamforming deconvolution
BeamformerCleansc	CLEAN-SC deconvolution
BeamformerCMF	Covariance Matrix Fitting
PointSpreadFunction	Point spread function for CLEAN and DAMAS
Environment	A simple acoustic environment without flow
UniformFlowEnvironment	An acoustic environment with uniform flow
GeneralFlowEnvironment	An acoustic environment with a generic flow field
FlowField	An abstract base class for a spatial flow field
OpenJet	Analytical approximation of the flow field of an open jet
MicGeom	Provides coordinates of microphones in the array
Grid	Base class for grid geometries
RectGrid	Provides a cartesian 2D grid for the beamforming results
RectGrid3D	Provides a cartesian 3D grid for the beamforming results
Calib	Container for calibration data in ∗.xml format
SignalGenerator	Virtual base class for a simple one-channel signal generator
WNoiseGenerator	White noise signal generator
SineGenerator	Sine signal generator with adjustable frequency and phase
Trajectory	Describes a trajectory from sampled points

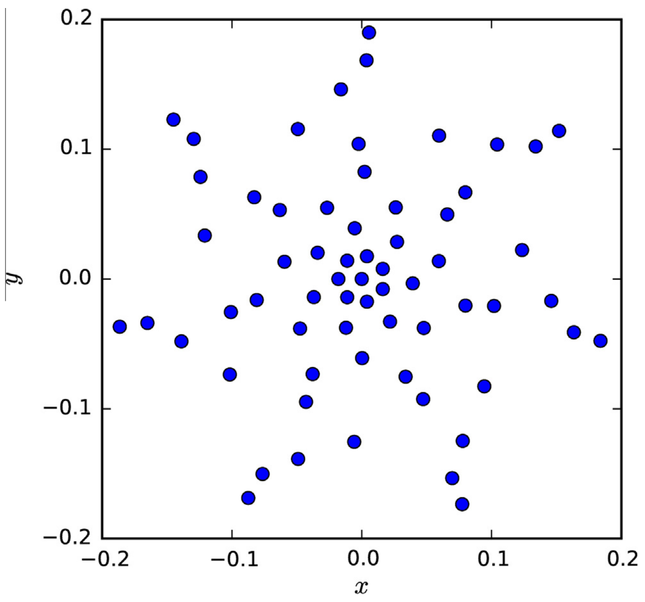


Fig. 4. Layout of the 64 microphones in the array.

Table 2
Location relative to the array center and strength of the three sources given as rms sound pressure in the array center.

Source	Location	Rms sound pressure
1	(−0.1, −0.1, 0.3) m	1 Pa
2	(0.15, 0, 0.3) m	0.7 Pa
3	(0, 0.1, 0.3) m	0.5 Pa

Both the microphone locations as well as the extent and the resolution of the grid of possible source positions are defined by instantiating a MicGeom and a RectGrid object, respectively. Then, the kind of analysis is chosen to be basic beamforming by creating a BeamformerBase object, and the speed of sound is given along with the previously defined PowerSpectra, MicGeom and RectGrid objects. Consequently, the BeamformerBase object has all the information needed for the computation. However, up to here, no actual computation is performed. The processing starts only when a result is really needed. In this example the map for the 8 kHz third-octave band is requested by the next line where a member function of the BeamformerBase object is called that returns the result for the 8 kHz third-octave band computed from all FFT lines within that band:

```
7   pm = bb.synthetic( 8000, 3 )
```

With this and without any further explicit command, the fetching of input data from the HDF5 file is triggered, followed by the computation of the cross spectral matrix and the subsequent computation of the acoustic source map. With the help of a Python plotting library such as Matplotlib [19] the result stored in the variable pm can be used to produce Fig. 5. For the sake of brevity, the respective lines in the script are omitted here (there is no Acoular-specific code). Fig. 5 shows an acoustic source map with the three different sources, blurred due to the imperfection of the spatial beamforming filter.

If now the same analysis shall be performed for different input data, it is not necessary to repeat all the above steps and define a new object. Instead, only the input file name in the TimeSamples object has to be changed. So, it takes only

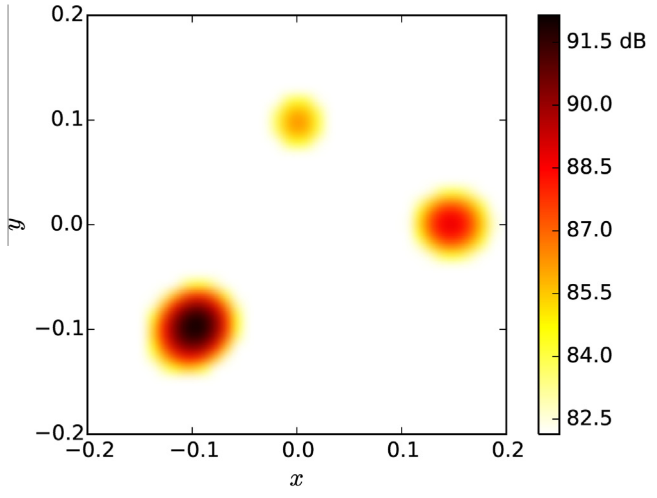


Fig. 5. Acoustic source map of three different sources from the example.

```
8   ts.name = 'two_sources.h5'
9   pm = bb.synthetic( 8000, 3 )
```

to trigger the fetching and subsequent processing of that data. In this example, a different input data file is given that contains the time histories when only the first two sources from Table 2 are active. The resulting image then shows only two sources (Fig. 6).

The first time the script is executed, the input data is processed and the computations are actually performed. Both the intermediate (cross spectral matrix) and the final result (acoustic source map) are saved to a persistent hard disk cache. On subsequent runs of the script, only the results that are needed are read from that cache, and computations are not repeated. While this saves only some seconds in this specific example, for more realistic scenarios some minutes to many hours of unnecessary computation time can be saved.

Finally, if a basic graphical user interaction is required, a special method can be called on each Acoular object, resulting in a simple graphical user interface that allows to edit the data attributes of that object. If, for example, one wants to edit the BeamformerBase object that was instantiated on line 6 of the example script, it suffices to put the command

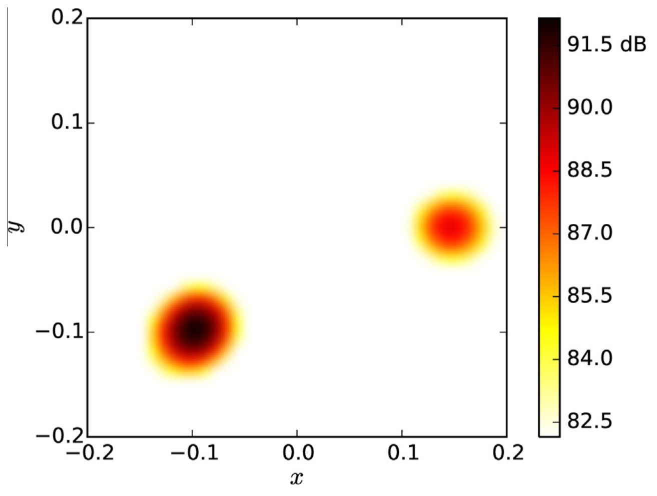


Fig. 6. Acoustic source map of two different sources.

```
10  bb.configure_traits()
```

to get the interface shown in Fig. 7. The appearance and layout of individual elements of that graphical user interface can be customized and the feature can also be used as a part of a more complex application software.

4.2. One rotating sound source

The second example is concerned with a moving sound source. The setup is similar to the first example, but this time the data to be analyzed is from a sound source moving on a circular trajectory in front of the array, as it would be the case for rotating machinery. The rotational speed is known to be 10 s^{-1} and the time history data in this example covers just one revolution (0.1 s). If the array input data is to be analyzed in the time domain, a chain of objects needs to be assembled similar to what is shown in Fig. 3. In order to do so, the source of the input data, the microphone locations, and the grid of possible source positions are given similar to the first example:

```
1   import acoular as ac
2   ts = ac.TimeSamples( name = 'rotating_source.h5' )
3   mg = ac.MicGeom( from_file = 'array_64.xml' )
4   rg = ac.RectGrid( x_min = −0.2, x_max = 0.2, y_min = −0.2,
                       y_max = 0.2, z = 0.3, increment = 0.01 )
```

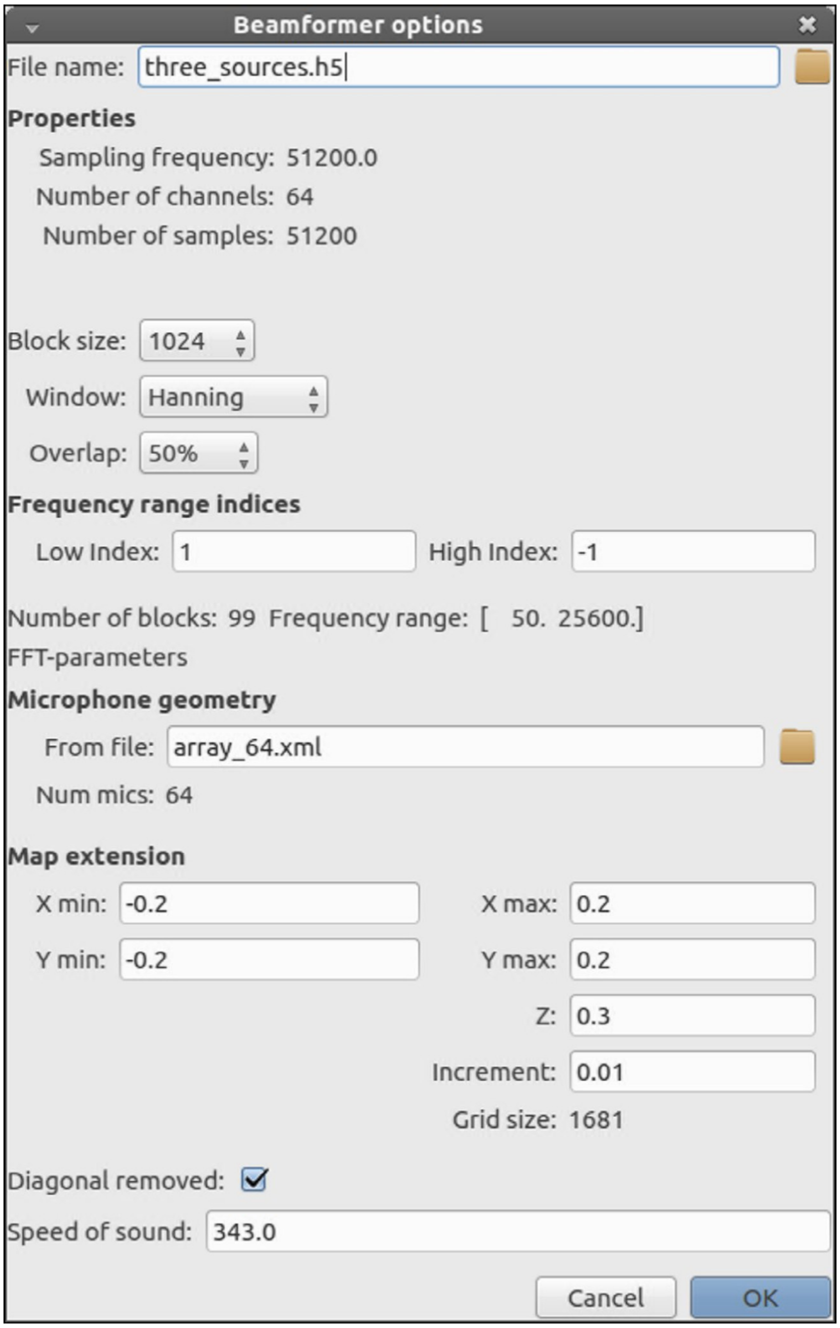
The chain of processing objects is then set up:

```
5   fi = ac.FiltFiltOctave(source = ts, band = 8000,
                        fraction = 'Third octave')
6   bt = ac.BeamformerTimeSq(source = fi, grid = rg, mpos = mg,
                              r_diag = True, c = 343.0)
7   avgt = ac.TimeAverage(source = bt,
                          naverage = ts.sample_freq*0.1/4)
8   cacht = ac.TimeCache(source = avgt)
```

After the TimeSamples object as the first element, a zero-delay third-octave frequency filter (FiltFiltOctave class) is used as the second element. It filters out the frequency range of interest in all microphone channels before the time-domain beamformer (BeamformerTimeSq) as third element in the chain processes them and computes the squared time histories for all grid points. After that, the data is averaged over a time that is equivalent to one fourth of a revolution of the source by the TimeAverage object. All objects classes are derived from the TimeInOut class and have thus a common interface that includes the source attribute which points to the preceding object in the chain.

The TimeCache object as the last element here does no actual processing, but transparently saves the data on disk. This allows the data to be recovered without repeated processing if the script is run without changes to any of the preceding elements in the processing chain. However, due to the lazy evaluation paradigm implemented in Acoular no processing at all is done unless results are actually requested. This would be the case if the resulting source maps are to be plotted by iterating over the output of cacht:

```
9   for res in cacht.result(1):
10      ...(code for plotting the results)
```


Beamformer options

File name: three_sources.h5

Properties

Sampling frequency: 51200.0
Number of channels: 64
Number of samples: 51200

Block size: 1024
Window: Hanning
Overlap: 50%

Frequency range indices

Low Index: 1 High Index: -1

Number of blocks: 99 Frequency range: [50. 25600.]
FFT-parameters

Microphone geometry

From file: array_64.xml
Num mics: 64

Map extension

X min: -0.2 X max: 0.2
Y min: -0.2 Y max: 0.2
Z: 0.3
Increment: 0.01
Grid size: 1681

Diagonal removed: ✓
Speed of sound: 343.0

Fig. 7. Graphical user interface for a BeamformerBase object.

While again the plotting commands are omitted here, the plot is given in Fig. 8. It shows an image of the point source smeared along the quadrant sections of its circular trajectory.

If the actual trajectory of the source is known (which is possible in this example by monitoring the rotational speed), then the Beamforming filter coefficients can be changed accordingly with time. This has the effect of virtually rotating the grid of possible source positions with the source. The circular trajectory is defined by 20 points that are passed by the source within one revolution of 0.1 s duration:

```
11  tr = ac.Trajectory()
12  for t in linspace(0, 0.1, 20):
13      phi = 10*t*2*pi
14      tr.points[t] = (0.1*cos(phi), 0.1*sin(phi), 0)
```

The Trajectory object uses spline interpolation to estimate the position at any instant needed. Finally, the BeamformerTimeSq object from script line 6 has to be replaced with one of the type BeamformerTimeSqTraj. Because it acts as a data source for the TimeAverage object, the source attribute of avgt also has to be updated:

```
15  bs = ac.BeamformerTimeSqTraj(source = fi, grid = rg,
                                 mpos = mg, trajectory = tr,
                                 rvec  = array((0,0,1.0)))
16  avgt.source = bs
```

The same procedure as before can now be used to produce the virtually rotating source map (see Fig. 9). As expected, this does not show any difference between the four quadrants, because it

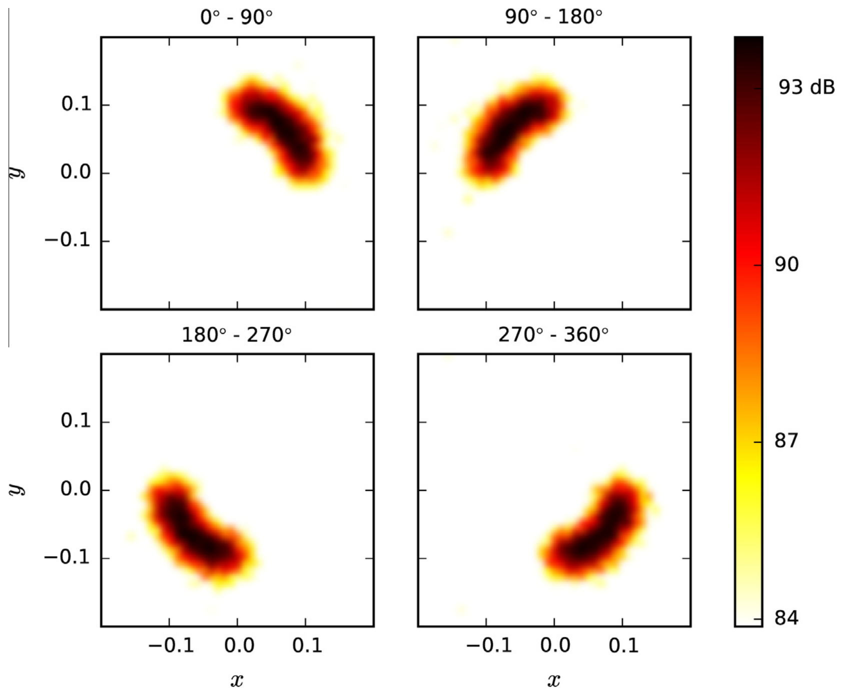


Fig. 8. Acoustic source map for a rotating source for time intervals equivalent one fourth of a revolution, analyzed with fixed beamformer filter coefficients.

uses the co-ordinate system relative to the source and not the array.

4.3. Further examples

Acoular has already been in use prior to its publication as open source software. Therefore, further examples of its application were already published. For example, some thousand measurements in an aeroacoustic wind tunnel were automatically processed and entered into a large database, which was used for research on the sound generation at porous airfoils [20]. Another example concerns the application for the experimental estimation of the sound generation by different bird species in flight, where Acoular was combined with a Python-based software to track the three-dimensional trajectory of the flight from multiple camera images [5]. A last example to be mentioned is the application in an industrial-scale context, were the library was used for large scale three-dimensional mapping of sound sources at a pantograph of a high speed train [21].

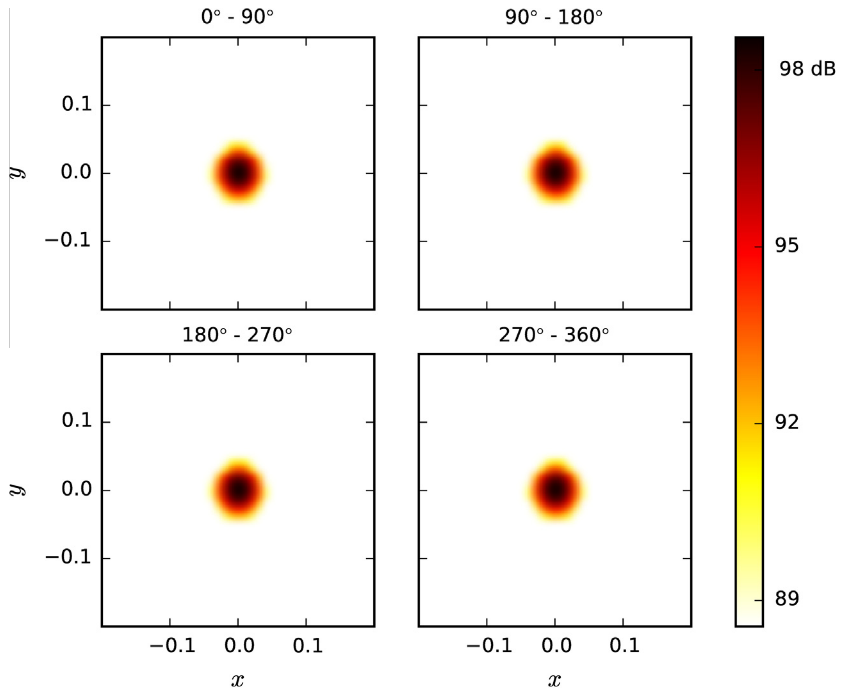


Fig. 9. Acoustic source map for a rotating source for time intervals equivalent one fourth of a revolution, analyzed with time-dependent beamformer filter coefficients to rotate the map along with the source.

5. Extending

As an example for a possible extension of Acoular to new processing algorithms, the functional beamforming algorithm of Dougherty [22] shall be discussed. It is a method working in the frequency domain and it uses

$$p_m^2 = \left(\mathbf{h}_m^H \mathbf{G}^{(\frac{1}{\gamma})} \mathbf{h}_m\right)^{\gamma} \tag{7}$$

instead of (5) to provide a result with less ambiguous side lobes in the source map. The matrix root in (7) has to be computed from the eigen-decomposition of **G**.

To implement this method, a subclass of the BeamformerBase has to be defined. In this FunctionalBeamformer class, one extra data member representing γ has to be defined using the mechanism provided by the Traits package. A second data member (freq_data) has to be redefined to accommodate objects that provide eigenvalues and eigenvectors of the cross-spectral matrix instead of only the cross-spectral matrix. Finally, the only function that needs to be redefined is the function that performs the actual computation:

```
1   class BeamformerFunctional(BeamformerBase):
2
3     gamma = Float(1, desc = ”functional exponent”)
4
5     freq_data = Trait(EigSpectra, desc = ”freq data object”)
6
7     def calc(self, ac, fr):
8         kj = 2j*pi*self.freq_data.fftfreq()/self.c
9         numchannels  = int (self.freq_data.numchannels)
10        e = zeros((numchannels), 'D')
11        h = empty((1, self.grid.size), 'd')
12        beamfunc = self.get_beamfunc('_os')
13        for i in self.freq_data.indices:
14            if not fr[i]:
15                eva = array(self.freq_data.eva [ i ] [ newaxis ],
                          dtype = 'float64')**(1.0/self.gamma)
16                eve = array(self.freq_data.eve [ i ] [ newaxis ],
                          dtype = 'complex128')
17                kji = kj[i, newaxis]
18                beamfunc(e, h, self.r0, self.rm, kji, eva, eve, 0,
                      numchannels)
19                ac[i] = h**self.gamma
20                fr[i] = True
```

In this function, lines 8–12 initialize variables necessary for the computation. The loop in lines 15–20 iterates over all frequencies that need to be considered. In line 15 and 16 the eigenvalue roots and eigenvectors are stored, while in line 17 the correct wavenumber for the frequency is provided. The vector-matrix-vector calculation from (7) is performed in line 18 by a call to a fast C++ routine that is provided by Acoular and also used for other processing methods. While all other data members and methods such as those responsible for the graphical user interface, the caching algorithm and further processing do not need to be redefined, the actual implementation of functional beamforming in Acoular contains many lines of comments that were stripped off here for the sake of brevity.

6. Conclusive summary

The open source Python library Acoular implements various methods for microphone array signal processing. Published under the terms of the new BSD license, it is aimed at applications in acoustic testing where sources of sound need to be characterized and mapped. It covers both time-domain and frequency-domain operation and can be applied to stationary and moving sound sources. Due to the consequent object-oriented approach, the library can be easily extended to incorporate new processing algorithms or methods. Intelligent caching of the results and the parallelized implementation of core routines in C++ make the library efficient and applicable to large, industrial-scale problems. Acoular supports both scripting and a graphical user interface so that the library can easily be integrated with other libraries available for data handling, visualization and scientific computing.

References

[1] Acoular acoustic testing and source mapping software; 2015. <http://www.acoular.org>.

[2] Sarradj E. Three-dimensional acoustic source mapping with different beamforming steering vector formulations. Adv Acoust Vib 2012;2012 (292695):1–12. http://dx.doi.org/10.1155/2012/292695.

[3] Capon J. High-resolution frequency-wavenumber spectrum analysis. Proc IEEE 1969;57(8):1408–18.

[4] Sijtsma P, Stoker R. Determination of absolute contributions of aircraft noise components using fly-over array measurements. In: Proceedings of the 10th AIAA/CEAS aeroacoustics conference, AIAA paper 2004-2958; 2004.

[5] Sarradj E, Fritzsche C, Geyer T. Silent owl flight: bird flyover noise measurements. AIAA J 2011;49(4):769–79. http://dx.doi.org/10.2514/1.53992.

[6] Bendat JS, Piersol AG. Random data: analysis and measurement procedures. John Wiley & Sons; 2011.

[7] Schmidt R. Multiple emitter location and signal parameter estimation. IEEE Trans Antennas Propagat 1986;34(3):276–80.

[8] Sarradj E. A fast signal subspace approach for the determination of absolute levels from phased microphone array measurements. J Sound Vib 2010;329:1553–69.

[9] Brühl S, Röder A. Acoustic noise source modelling based on microphone array measurements. J Sound Vib 2000;231:611–7.

[10] Brooks TF, Humphreys WM. A deconvolution approach for the mapping of acoustic sources (DAMAS) determined from phased microphone arrays. In: Proceedings of the 10th AIAA/CEAS aeroacoustics conference, AIAA paper 2004-2954; 2004.

[11] Brooks TF, Humphreys WM. Extension of DAMAS phased array processing for spatial coherence determination (DAMAS-C). In: Proceedings of the12th AIAA/CEAS aeroacoustics conference, AIAA paper 2006-2654; 2006.

[12] Sijtsma P. CLEAN based on spatial source coherence. In: Proceedings of the 13th AIAA/CEAS aeroacoustics conference, AIAA paper 2007-3436; 2007.

[13] Blacodon D, Elias G. Level estimation of extended acoustic sources using a parametric method. J Aircraft 2004;41:1360–9.

[14] Yardibi T, Li J, Stoica P, Zawodny N, Cattafesta III L. A covariance fitting approach for correlated acoustic source mapping. J Acoust Soc Am 2010;127:2920.

[15] Michel U, Funke S. Noise source analysis of an aeroengine with a new inverse method SODIX. In: Proceedings of the 14th AIAA/CEAS aeroacoustics conference, AIAA paper 2008-2860; 2008.

[16] Enthought, Inc. The traits framework for validation and event-driven programming in python; 2001–2015. <http://docs.enthought.com/traits/>.

[17] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: elements of reusable object-oriented software. Pearson Education; 1994.

[18] The HDF Group. Hierarchical Data Format, version 5; 1997–2015. <http://www.hdfgroup.org/HDF5/>.

[19] Hunter JD. Matplotlib: a 2d graphics environment. Comput Sci Eng 2007;9 (3):90–5.

[20] Geyer T, Sarradj E, Fritzsche C. Measurement of the noise generation at the trailing edge of porous airfoils. Exp Fluids 2010;48:291–308.

[21] Sarradj E, Geyer T, Brick H, Kirchner K-R, Kohrs T. Application of beamforming and deconvolution techniques to aeroacoustic sources at highspeed trains. In: NOVEM – noise and vibration: emerging methods, Sorrento.

[22] Dougherty R. Functional beamforming. In: Proceedings on CD of the 5th Berlin beamforming conference, 19–20 February 2014. Berlin: GFaI, Gesellschaft zur Förderung angewandter Informatik e.V.; 2014.