A Python framework for microphone array data processing

Ennes Sarradj a,⇑, Gert Herold b

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

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

Keywords:
Microphone array
Beamforming
Python
Software

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

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 $\Delta t_{mn}$ 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:

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

(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 sounds of a monopole source. As long as both source strength and position of the sources from the measured $p_n$ 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:

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

(2)

Here, the idea is to choose $h_{mn}$ and $\Delta t_{mn}$ in such a way that the output $p_m$ 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 $\Delta t_{mn}$ from $a_{mn}$ and $\tau_{mn}$. 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 $p_m^2$ 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):

$$p_m^2(t) = \max \left( \left( \sum_n h_{mn} p_n(t + \Delta t_{mn}) \right)^2 - \sum_n \left( h_{mn} p_n(t + \Delta t_{mn}) \right)^2 \right) / T.$$  

(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 $N$ 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

$$G(\omega) = \frac{1}{N} \sum_{t=1}^{N} \mathbf{P}(f_k) \mathbf{P}^\dagger(f_k).$$

(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 $G$ can then be used as a basis to perform the spatial filtering in frequency domain:

$$p_m^2 = h_{mn}^\dagger G h_{mn}.$$  

(5)

Fig. 1. Working principle of beamforming algorithms.
The complex-valued steering vectors $\mathbf{h}_m$ now contain the $h_{mn}$ and $\Delta t_{mn}$ 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 $G$ 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

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

(6)

directly, where $G$ 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 TimelnOut or of the class PowerSpectra and its subclasses. TimelnOut defines an interface
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 \( s_{mn} \) 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 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
1   import acoular as ac
2   ts = ac.TimeSamples( name = 'three_sources.h5' )
3   ps = ac.PowerSpectra( time_data = ts, block_size = 128,
             window = 'Hanning', overlap = '50%' )
4   mg = ac.MicGeom( from_file = 'array_64.xml' )
5   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 )
6   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.
Table 1

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

Table 2

<table>
<thead>
<tr>
<th>Source</th>
<th>Location</th>
<th>Rms sound pressure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(-0.1, -0.1, 0.3) m</td>
<td>1 Pa</td>
</tr>
<tr>
<td>2</td>
<td>(0.15, 0, 0.3) m</td>
<td>0.7 Pa</td>
</tr>
<tr>
<td>3</td>
<td>(0, 0.1, 0.3) m</td>
<td>0.5 Pa</td>
</tr>
</tbody>
</table>

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 sound pressure in the array center.

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
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

```python
bb.configure_traits()
```
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:

```python
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 TimeAver-age object, the source attribute of avgt also has to be updated:

```python
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
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].
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

$$ p_m^2 = (\mathbf{h}^H \mathbf{G}^{\text{ij}} \mathbf{h}_m)^2 $$

(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 \( \mathbf{G} \).

To implement this method, a subclass of the BeamformerBase has to be defined. In this FunctionalBeamformer class, one extra data member representing \( \gamma \) 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:

```python
class BeamformerFunctional(BeamformerBase):
    def calc(self, ac, fr):
        kji = 2j*pi*freq_data.fftfreq() / self.c
        numchannels = int(freq_data.numchannels)
        e = zeros((numchannels), 'float64')
        h = empty((1, self.grid.size), 'D')
        beamfunc = self.get_beamfunc('_os')
        for i in self.freq_data.indices:
            if not fr[i]:
                eva = array(freq_data.eva[i], 'float64')
                dttype = complex128
                eve = array(freq_data.eve[i], 'float64')
                kji = kji[i, newaxis]
                beamfunc(e, h, self.r0, self.rm, kji, eve, ev, 0, numchannels)
                ac[i] = h*sqrt(sums)

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 parallelization of routine 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