Sound Field Analysis toolbox Documentation

Release 2021.2.4

Christoph Hohnerlein (QU Labs)
Jens Ahrens (Chalmers)
Hannes Helmholz (Chalmers)

Feb 04, 2021

CONTENTS:

CHAPTER

ONE

USAGE

The *sound_field_analysis* toolbox (short: *sfa*) is a Python port of the Sound Field Analysis Toolbox (SOFiA) toolbox, originally by Benjamin Bernschütz *[1]*. The main goal of the *sfa* toolbox is to analyze, visualize and process sound field data recorded by spherical microphone arrays. Furthermore, various types of test-data may be generated to evaluate the implemented functions. It is an essential building block of ReTiSAR, an implementation of real time binaural rendering of spherical microphone array data.

1.1 Requirements

We use Python 3.9 for development. Chances are that earlier version will work too but this is currently untested.

The following external libraries are required:

- NumPy
- SciPy
- Pysofaconventions
- Jupyter (for running *Notebooks* locally)
- Plotly (for plotting)

1.2 Installation

For performance and convenience reasons we highly recommend to use Conda (miniconda for simplicity) to manage your Python installation. Once installed, you can use the following steps to receive and use *sfa*, depending on your use case:

- From PyPI / `pip`:

  Install into an existing environment (without example Jupyter *Notebooks*):

  ```
  pip install sound_field_analysis
  ```

- By cloning (or downloading) the repository and setting up a new environment:

  ```
  git clone
  https://github.com/AppliedAcousticsChalmers/sound_field_analysis-py.git
  cd sound_field_analysis-py/
  ```

  Create a new Conda environment from the specified dependencies:

  ```
  conda env create --file environment.yml --force
  ```

Activate the environment:

```
source activate sfa
```

Optional: Install additional dependencies for development purposes (locally run Jupyter *Notebooks* with example, run tests, generate documentation):

```
conda env update --file environment_dev.yml
```

1.3 Examples

The following examples are available as Jupyter *Notebooks*, either statically on GitHub or interactively on nbviewer. You can of course also simply download the examples and run them locally!

1.3.1 Exp1: Ideal plane wave

Ideal unity plane wave simulation and 3D plot.

View interactively on nbviewer

1.3.2 Exp2: Measured plane wave

A measured plane wave from AZ=180°, EL=90° in the anechoic chamber using a cardioid mic.

View interactively on nbviewer

1.3.3 Exp4: Binaural rendering

Render a spherical microphone array impulse response measurement binaurally. The example shows examples for loading miro or SOFA files.

View interactively on nbviewer

1.4 Version history

v2021.2.4

- Implement option to use real spherical harmonic basis functions
- Update Exp4 to optionally utilize real spherical harmonics
- Fix testing of spherical harmonics against reference Matlab implementation
- Add testing for generation of real spherical harmonics
- Add evaluation of performance for generation of complex and real spherical harmonics
- Add evaluation of performance for spatial sound field decomposition
- Update Conda environment setup to combine all development dependencies
- Update online and offline documentation

v2021.1.12

- Update MIRO struct loading (quadrature weights are now optional)
- Fix to prevent Python 3.8 syntax warnings

- Improve Exp4 (general code structure and utilizing Spherical Head Filter and Spherical Harmonics Tapering)

v2020.1.30

- Update README and PyPI package

v2019.11.6

- Update internal documentation and string formatting

v2019.8.15

- Change version number scheme to CalVer
- Improve Exp4
- Update *read_SOFA_file()*
- Update 2D plotting functions
- Improve *write_SSR_IRs()*
- Improve Conda environment setup for Jupyter Notebooks
- Update *miro_to_struct()*

2019-07-30 (v0.9)

- Implement SOFA import
- Update Exp4 to contain SOFA import
- Delete obsolete Exp3
- Add named tuple *HRIRSignal*
- Implement *cart2sph()* and *sph2cart()* utility functions
- Add Conda environment file for convenient installation of required packages

2019-07-11 (v0.8)

- Implement Spherical Harmonics coefficients tapering
- Update Spherical Head Filter to consider tapering

2019-06-17 (v0.7)

- Implement Bandwidth Extension for Microphone Arrays (BEMA)
- Edit *read_miro_struct()*, named tuple *ArraySignal* and *miro_to_struct.m* to load center measurements

2019-06-11 (v0.6)

- Implement Radial Filter Improvement from Sound Field Analysis Toolbox (SOFiA) toolbox

2019-05-23 (v0.5)

- Implement Spherical Head Filter
- Implement Spherical Fourier Transform using pseudo-inverse
- Extract real time capable spatial Fourier transform
- Extract reversed m index function (Update Exp4)

1.5 Contribute

See CONTRIBUTE.rst for full details.

You can find the full offline documentation as PDF as well as online at https://appliedacousticschalmers.github.io/sound_field_analysis-py/ .

1.6 License

This software is licensed under the MIT License (see LICENSE for full details).

1.7 References

The *sound_field_analysis* toolbox is based on the Matlab/C++ Sound Field Analysis Toolbox (SOFiA) toolbox by Benjamin Bernschütz. For more information you may refer to the original publication:

[1] Bernschütz, B., Pörschmann, C., Spors, S., and Weinzierl, S. (2011). SOFiA Sound Field Analysis Toolbox. Proceedings of the ICSA International Conference on Spatial Audio

The Lebedev grid generation was adapted from an implementation by Richard P. Muller.

CHAPTER

TWO

MODULES

2.1 Generators

Module containing various generator functions:

whiteNoise Adds White Gaussian Noise of approx. 16dB crest to a FFT block.

gauss_grid Compute Gauss-Legendre quadrature nodes and weights in the SOFiA / VariSphear data format.

lebedev Compute Lebedev quadrature nodes and weights given a maximum stable order. Alternatively, a degree may be supplied.

radial_filter Generate modal radial filter of specified orders and frequencies.

radial_filter_fullspec Generate NFFT/2 + 1 modal radial filter of orders 0:max_order for frequencies 0:fs/2, wraps *radial_filter()*.

spherical_head_filter Generate coloration compensation filter of specified maximum SH order.

spherical_head_filter_spec Generate NFFT/2 + 1 coloration compensation filter of specified maximum SH order for frequencies 0:fs/2, wraps *spherical_head_filter()*.

tapering_window Design tapering window with cosine slope for orders greater than 3.

sampled_wave Returns the frequency domain data of an ideal wave as recorded by a provided array.

ideal_wave Ideal wave generator, returns spatial Fourier coefficients *Pnm* of an ideal wave front hitting a specified array.

spherical_noise Returns order-limited random weights on a spherical surface.

delay_fd Generate delay in frequency domain that resembles a circular shift in time domain.

2.2 I/O

Module containing various input and output functions:

TimeSignal Named tuple to store time domain data and related metadata.

SphericalGrid Named tuple to store spherical sampling grid geometry.

ArrayConfiguration Named tuple to store microphone array characteristics.

ArraySignal Named tuple to store microphone array data in terms of *TimeSignal*, *SphericalGrid* and *ArrayConfiguration*

HrirSignal Named tuple to store Head Related Impulse Response grid data in terms of *TimeSignal* for either ear and *SphericalGrid*

read_miro_struct Read Head Related Impulse Responses (HRIRs) or Array / Directional Impulse Responses (DRIRs) stored as MIRO Matlab files and convert them to *ArraySignal*.

read_SOFA_file Read Head Related Impulse Responses (HRIRs) or Array / Directional Impulse Responses (DRIRs) stored as Spatially Oriented Format for Acoustics (SOFA) files and convert them to *ArraySignal* or *HrirSignal*.

empty_time_signal Returns an empty np rec array that has the proper data structure.

load_array_signal Convenience function to load ArraySignal saved into np data structures.

read_wavefile Reads in WAV files and returns data [Nsig x Nsamples] and fs.

write_SSR_IRs Takes two time signals and writes out the horizontal plane as HRIRs for the SoundScapeRenderer. Ideally, both hold 360 IRs but smaller sets are tried to be scaled up using repeat.

2.3 Lebedev

Module to generate Lebedev grids and quadrature weights for degrees 6, 14, 26, 38, 50, 74, 86, 110, 146, 170, 194:

genGrid(n) Generate Lebedev grid geometry of degree *n*.

Adapted from Richard P. Mullers Python version, https://github.com/gabrielelanaro/pyquante/blob/master/Data/lebedev_write.py C version: Dmitri Laikov F77 version: Christoph van Wuellen, http://www.ccl.net

Users of this code are asked to include reference[3] in their publications, and in the user- and programmers-manuals describing their codes.

References

2.4 Plotting

Module containing various plotting functions:

makeMTX Returns a plane wave decomposition over a full sphere.

makeFullMTX Generates visualization matrix for a set of spatial fourier coefficients over all kr.

plot2D Visualize 2D data using plotly.

plot3D Visualize matrix data, such as from *makeMTX(Pnm, dn)*.

plot3Dgrid Visualize matrix data in a grid, such as from *makeMTX(Pnm, dn)*.

2.5 Processing

Module containing functions to act on the Spatial Fourier Coefficients.

BEMA Perform Bandwidth Extension for Microphone Arrays (BEMA) spatial anti-aliasing.

FFT Perform real-valued Fast Fourier Transform (FFT).

iFFT Perform inverse real-valued (Fast) Fourier Transform (iFFT).

spatFT Perform spatial Fourier transform.

spatFT_RT Perform spatial Fourier transform for real-time applications (otherwise use *spatFT()* for more more convenience and flexibility).

spatFT_LSF Perform spatial Fourier transform by least square fit to provided data.

[3] V.I. Lebedev, and D.N. Laikov 'A quadrature formula for the sphere of the 131st algebraic order of accuracy' Doklady Mathematics, Vol. 59, No. 3, 1999, pp. 477-481.

iSpatFT Perform inverse spatial Fourier transform.

plane_wave_decomp Perform plane wave decomposition.

rfi Perform radial filter improvement (RFI)

sfe Perform sound field extrapolation (SFE) - WORK IN PROGRESS.

wdr Perform Wigner-D rotation (WDR) - NOT YET IMPLEMENTED.

convolve Convolve two arrays A & B row-wise where one or both can be one-dimensional for SIMO/SISO convolution.

2.6 Spherical

Module containing various spherical harmonics helper functions:

besselj / neumann Bessel function of first and second kind of order n at kr.

hankel1 / hankel2 Hankel function of first and second kind of order n at kr.

spbessel / spneumann Spherical Bessel function of first and second kind of order n at kr.

sphankel1 / sphankel2 Spherical Hankel of first and second kind of order n at kr.

dspbessel / dspneumann Derivative of spherical Bessel of first and second kind of order n at kr.

dsphankel1 / dsphankel2 Derivative spherical Hankel of first and second kind of order n at kr.

spherical_extrapolation Factor that relate signals recorded on a sphere to it’s center.

array_extrapolation Factor that relate signals recorded on a sphere to it’s center. In the rigid configuration, a scatter_radius that is different to the array radius may be set.

sph_harm Compute spherical harmonics.

sph_harm_large Compute spherical harmonics for large orders > 84.

sph_harm_all Compute all spherical harmonic coefficients up to degree nMax.

mnArrays Generate degrees n and orders m up to nMax.

reverseMnIds Generate reverse indexes according to stacked coefficients of orders m up to nMax.

cart2sph / sph2cart Convert cartesian to spherical coordinates and vice versa.

kr Generate kr vector for given f and array radius.

kr_full_spec Generate full spectrum kr.

2.7 Utilities

Module containing various utility functions:

env_info Guess environment based on *sys.modules*.

progress_bar Display a spinner or a progress bar.

db Convenience function to calculate the 20*log10(abs(x)).

cart2sph / sph2cart Transform cartesian into spherical coordinates and vice versa.

SOFA_grid2acr Transform coordinate grid with specified coordinate system definition from a SOFA file into spherical coordinates in radians.

nearest_to_value / nearest_to_value_IDX / nearest_to_value_logical_IDX Returns nearest value inside an array.

interleave_channels Interleave left and right channels. Style == ‘SSR’ checks if we total 360 channels.

simple_resample Wrap scipy.signal.resample with a simpler API.

scalar_broadcast_match Returns arguments as np.array, if one is a scalar it will broadcast the other one's shape.

frq2kr Returns the kr bin closest to the target frequency.

stack Stacks two 2D vectors along the same-sized dimension or the smaller one.

zero_pad_fd Apply zero padding to frequency domain data by transformation into time domain and back.

current_time Return current system time based on *datetime.now()*.

time_it Measure execution of a specified statement which is useful for the performance analysis. In this way, the execution time and respective results can be directly compared.

CHAPTER

THREE

REFERENCE

3.1 Generators

Module containing various generator functions:

whiteNoise Adds White Gaussian Noise of approx. 16dB crest to a FFT block.

gauss_grid Compute Gauss-Legendre quadrature nodes and weights in the SOFiA / VariSphear data format.

lebedev Compute Lebedev quadrature nodes and weights given a maximum stable order. Alternatively, a degree may be supplied.

radial_filter Generate modal radial filter of specified orders and frequencies.

radial_filter_fullspec Generate NFFT/2 + 1 modal radial filter of orders 0:max_order for frequencies 0:fs/2, wraps *radial_filter()*.

spherical_head_filter Generate coloration compensation filter of specified maximum SH order.

spherical_head_filter_spec Generate NFFT/2 + 1 coloration compensation filter of specified maximum SH order for frequencies 0:fs/2, wraps *spherical_head_filter()*.

tapering_window Design tapering window with cosine slope for orders greater than 3.

sampled_wave Returns the frequency domain data of an ideal wave as recorded by a provided array.

ideal_wave Ideal wave generator, returns spatial Fourier coefficients *Pnm* of an ideal wave front hitting a specified array.

spherical_noise Returns order-limited random weights on a spherical surface.

delay_fd Generate delay in frequency domain that resembles a circular shift in time domain.

`sound_field_analysis.gen.`**`whiteNoise`**(*fftData*, *noiseLevel=80*)

Adds White Gaussian Noise of approx. 16dB crest to a FFT block.

Parameters

- **fftData** (*array of complex floats*) – Input fftData block (e.g. from F/D/T or S/W/G)
- **noiseLevel** (*int, optional*) – Average noise Level in dB [Default: -80dB]

Returns **noisyData** (*array of complex floats*) – Output fftData block including white gaussian noise

`sound_field_analysis.gen.`**`gauss_grid`**(*azimuth_nodes=10*, *colatitude_nodes=5*)

Compute Gauss-Legendre quadrature nodes and weights in the SOFiA / VariSphear data format.

Parameters **azimuth_nodes, colatitude_nodes** (*int, optional*) – Number of azimuth / elevation nodes

Returns **gridData** (*io.SphericalGrid*) – SphericalGrid containing azimuth, colatitude and weights `sound_field_analysis.gen.`**`lebedev`**(*max_order=None*, *degree=None*)

Compute Lebedev quadrature nodes and weights given a maximum stable order. Alternatively, a degree may be supplied.

Parameters

- **max_order** (*int*) – Maximum stable order of the Lebedev grid, [0 ... 11]
- **degree** (*int, optional*) – Lebedev Degree, one of {6, 14, 26, 38, 50, 74, 86, 110, 146, 170, 194}

Returns **gridData** (*array_like*) – Lebedev quadrature positions and weights: [AZ, EL, W]

`sound_field_analysis.gen.`**`radial_filter_fullspec`**(*max_order*, *NFFT*, *fs*, *array_configuration*, *amp_maxdB=40*)

Generate NFFT/2 + 1 modal radial filter of orders 0:max_order for frequencies 0:fs/2, wraps *radial_filter()*.

Parameters

- **max_order** (*int*) – Maximum order
- **NFFT** (*int*) – Order of FFT (number of bins), should be a power of 2.
- **fs** (*int*) – Sampling frequency
- **array_configuration** (*io.ArrayConfiguration*) – List/Tuple/ArrayConfiguration, see io.ArrayConfiguration
- **amp_maxdB** (*int, optional*) – Maximum modal amplification limit in dB [Default: 40]

Returns **dn** (*array_like*) – Vector of modal frequency domain filter of shape [max_order + 1 x NFFT / 2 + 1]

`sound_field_analysis.gen.`**`radial_filter`**(*orders*, *freqs*, *array_configuration*, *amp_maxdB=40*)

Generate modal radial filter of specified orders and frequencies.

Parameters

- **orders** (*array_like*) – orders of filter
- **freqs** (*array_like*) – Frequency of modal filter
- **array_configuration** (*io.ArrayConfiguration*) – List/Tuple/ArrayConfiguration, see io.ArrayConfiguration
- **amp_maxdB** (*int, optional*) – Maximum modal amplification limit in dB [Default: 40]

Returns **dn** (*array_like*) – Vector of modal frequency domain filter of shape [nOrders x nFreq]

`sound_field_analysis.gen.`**`spherical_head_filter`**(*max_order*, *full_order*, *kr*, *is_tapering=False*)

Generate coloration compensation filter of specified maximum SH order, according to[1].

Parameters

- **max_order** (*int*) – Maximum order
- **full_order** (*int*) – Full order necessary to expand sound field in entire modal range
- **kr** (*array_like*) – Vector of corresponding wave numbers
- **is_tapering** (*bool, optional*) – If set, spherical head filter will be adapted applying a Hann window, according to[2]

Returns **G_SHF** (*array_like*) – Vector of frequency domain filter of shape [NFFT / 2 + 1]

[1] Ben-Hur, Z., Brinkmann, F., Sheaffer, J., et al. (2017). "Spectral equalization in binaural signals represented by order-truncated spherical harmonics. The Journal of the Acoustical Society of America".

[2] Hold, Christoph, Hannes Gamper, Ville Pulkki, Nikunj Raghuvanshi, and Ivan J. Tashev (2019). "Improving Binaural Ambisonics Decoding by Spherical Harmonics Domain Tapering and Coloration Compensation."

References

`sound_field_analysis.gen.`**`spherical_head_filter_spec`**(*max_order*, *NFFT*, *fs*, *radius*, *amp_maxdB=None*, *is_tapering=False*)

Generate NFFT/2 + 1 coloration compensation filter of specified maximum SH order for frequencies 0:fs/2, wraps *spherical_head_filter()*.

Parameters

- **max_order** (*int*) – Maximum order
- **NFFT** (*int*) – Order of FFT (number of bins), should be a power of 2
- **fs** (*int*) – Sampling frequency
- **radius** (*float*) – Array radius
- **amp_maxdB** (*int, optional*) – Maximum modal amplification limit in dB [Default: None]
- **is_tapering** (*bool, optional*) – If true spherical head filter will be adapted for SH tapering. [Default: False]

Returns **G_SHF** (*array_like*) – Vector of frequency domain filter of shape [NFFT / 2 + 1]

Todo: Implement *arctan()* soft-clipping

`sound_field_analysis.gen.`**`tapering_window`**(*max_order*)

Design tapering window with cosine slope for orders greater than 3, according to[2].

Parameters **max_order** (*int*) – Maximum SH order

Returns **hann_window_half** (*array_like*) – Tapering window with cosine slope for orders greater than 3. Ones in case of maximum SH order being smaller than 3.

References

`sound_field_analysis.gen.`**`sampled_wave`**(*order*, *fs*, *NFFT*, *array_configuration*, *gridData*, *wave_azimuth*, *wave_colatitude*, *wavetype='plane'*, *c=343*, *distance=1.0*, *limit_order=85*, *kind='complex'*)

Returns the frequency domain data of an ideal wave as recorded by a provided array.

Parameters

- **order** (*int*) – Maximum transform order
- **fs** (*int*) – Sampling frequency
- **NFFT** (*int*) – Order of FFT (number of bins), should be a power of 2.
- **array_configuration** (*io.ArrayConfiguration*) – List/Tuple/ArrayConfiguration, see io.ArrayConfiguration
- **gridData** (*io.SphericalGrid*) – List/Tuple/gauss_grid, see io.SphericalGrid
- **wave_azimuth, wave_colatitude** (*float, optional*) – Direction of incoming wave in radians [0-2pi].
- **wavetype** (*{'plane', 'spherical'}, optional*) – Type of the wave. [Default: plane]
- **c** (*float, optional*) – __UNUSED__ Speed of sound in [m/s] [Default: 343 m/s]
- **distance** (*float, optional*) – Distance of the source in [m] (For spherical waves only)

- **limit_order** (*int, optional*) – Sets the limit for wave generation
- **kind** (*{'complex', 'real'}, optional*) – Spherical harmonic coefficients data type [Default: 'complex']

Warning: If NFFT is smaller than the time the wavefront needs to travel from the source to the array, the impulse response will by cyclically shifted (cyclic convolution).

Returns **Pnm** (*array_like*) – Spatial fourier coefficients of resampled sound field

Todo: Investigate if *limit_order* works as intended

`sound_field_analysis.gen.`**`ideal_wave`**(*order*, *fs*, *azimuth*, *colatitude*, *array_configuration*, *wavetype='plane'*, *distance=1.0*, *NFFT=128*, *delay=0.0*, *c=343.0*, *kind='complex'*)

Ideal wave generator, returns spatial Fourier coefficients *Pnm* of an ideal wave front hitting a specified array.

Parameters

- **order** (*int*) – Maximum transform order
- **fs** (*int*) – Sampling frequency
- **NFFT** (*int*) – Order of FFT (number of bins), should be a power of 2
- **array_configuration** (*io.ArrayConfiguration*) – List/Tuple/ArrayConfiguration, see io.ArrayConfiguration
- **azimuth, colatitude** (*float*) – Azimuth/Colatitude angle of the wave in [RAD]
- **wavetype** (*{'plane', 'spherical'}, optional*) – Select between plane or spherical wave [Default: Plane wave]
- **distance** (*float, optional*) – Distance of the source in [m] (for spherical waves only)
- **delay** (*float, optional*) – Time Delay in s [default: 0]
- **c** (*float, optional*) – Propagation velocity in m/s [Default: 343m/s]
- **kind** (*{'complex', 'real'}, optional*) – Spherical harmonic coefficients data type [Default: 'complex']

Warning: If NFFT is smaller than the time the wavefront needs to travel from the source to the array, the impulse response will by cyclically shifted.

Returns **Pnm** (*array of complex floats*) – Spatial Fourier Coefficients with nm coeffs in cols and FFT coeffs in rows

`sound_field_analysis.gen.`**`spherical_noise`**(*gridData=None*, *order_max=8*, *kind='complex'*, *spherical_harmonic_bases=None*)

Returns order-limited random weights on a spherical surface.

Parameters

- **gridData** (*io.SphericalGrid*) – SphericalGrid containing azimuth and colatitude
- **order_max** (*int, optional*) – Spherical order limit [Default: 8]
- **kind** (*{'complex', 'real'}, optional*) – Spherical harmonic coefficients data type [Default: 'complex']

- **spherical_harmonic_bases** (*array_like, optional*) – Spherical harmonic base coefficients (not yet weighted by spatial sampling grid) [Default: None]

Returns **noisy_weights** (*array_like, complex*) – Noisy weights

`sound_field_analysis.gen.`**`delay_fd`**(*target_length_fd*, *delay_samples*)
Generate delay in frequency domain that resembles a circular shift in time domain.

Parameters

- **target_length_fd** (*int*) – number of bins in single-sided spectrum
- **delay_samples** (*float*) – delay time in samples (subsample precision not tested yet!)

Returns *numpy.ndarray* – delay spectrum in frequency domain

3.2 I/O

Module containing various input and output functions:

TimeSignal Named tuple to store time domain data and related metadata.

SphericalGrid Named tuple to store spherical sampling grid geometry.

ArrayConfiguration Named tuple to store microphone array characteristics.

ArraySignal Named tuple to store microphone array data in terms of *TimeSignal*, *SphericalGrid* and *ArrayConfiguration*

HrirSignal Named tuple to store Head Related Impulse Response grid data in terms of *TimeSignal* for either ear and *SphericalGrid*

read_miro_struct Read Head Related Impulse Responses (HRIRs) or Array / Directional Impulse Responses (DRIRs) stored as MIRO Matlab files and convert them to *ArraySignal*.

read_SOFA_file Read Head Related Impulse Responses (HRIRs) or Array / Directional Impulse Responses (DRIRs) stored as Spatially Oriented Format for Acoustics (SOFA) files and convert them to *ArraySignal* or *HrirSignal*.

empty_time_signal Returns an empty np rec array that has the proper data structure.

load_array_signal Convenience function to load ArraySignal saved into np data structures.

read_wavefile Reads in WAV files and returns data [Nsig x Nsamples] and fs.

write_SSR_IRs Takes two time signals and writes out the horizontal plane as HRIRs for the SoundScapeRenderer. Ideally, both hold 360 IRs but smaller sets are tried to be scaled up using repeat.

`class` `sound_field_analysis.io.`**`TimeSignal`**(*signal*, *fs*, *delay=None*)
Named tuple to store time domain data and related metadata.

`static __new__`(*cls*, *signal*, *fs*, *delay=None*)

Parameters

- **signal** (*array_like*) – Array of signals of shape [nSignals x nSamples]
- **fs** (*int or array_like*) – Sampling frequency
- **delay** (*float or array_like, optional*) – [Default: None]

`count`(*value*, */*)
Return number of occurrences of value.

`delay`
Alias for field number 2

`fs`
Alias for field number 1

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

signal
Alias for field number 0

class sound_field_analysis.io.**SphericalGrid**(*azimuth*, *colatitude*, *radius=None*, *weight=None*)
Named tuple to store spherical sampling grid geometry.

static __new__(*cls*, *azimuth*, *colatitude*, *radius=None*, *weight=None*)

Parameters

- **azimuth, colatitude** (*array_like*) – Grid sampling point directions in radians
- **radius, weight** (*float or array_like, optional*) – Grid sampling point distances and weights

azimuth
Alias for field number 0

colatitude
Alias for field number 1

count(*value*, */*)
Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

radius
Alias for field number 2

weight
Alias for field number 3

class sound_field_analysis.io.**ArrayConfiguration**(*array_radius*, *array_type*, *transducer_type*, *scatter_radius=None*, *dual_radius=None*)
Named tuple to store microphone array characteristics.

static __new__(*cls*, *array_radius*, *array_type*, *transducer_type*, *scatter_radius=None*, *dual_radius=None*)

Parameters

- **array_radius** (*float or array_like*) – Radius of array
- **array_type** (*{'open', 'rigid'}*) – Type array
- **transducer_type** (*{'omni', 'cardioid'}*) – Type of transducer,
- **scatter_radius** (*float, optional*) – Radius of scatterer, required for *array_type* == 'rigid'. [Default: equal to array_radius]
- **dual_radius** (*float, optional*) – Radius of second array, required for *array_type* == 'dual'

array_radius
Alias for field number 0

array_type
Alias for field number 1

count (*value*, /)
Return number of occurrences of value.

dual_radius
Alias for field number 4

index (*value*, *start=0*, *stop=9223372036854775807*, /)
Return first index of value.

Raises ValueError if the value is not present.

scatter_radius
Alias for field number 3

transducer_type
Alias for field number 2

class sound_field_analysis.io.**ArraySignal** (*signal*, *grid*, *center_signal=None*, *configuration=None*, *temperature=None*)
Named tuple to store microphone array data in terms of *TimeSignal*, *SphericalGrid* and *ArrayConfiguration*.

static __new__ (*cls*, *signal*, *grid*, *center_signal=None*, *configuration=None*, *temperature=None*)

Parameters

- **signal** (*TimeSignal*) – Array Time domain signals and sampling frequency
- **grid** (*SphericalGrid*) – Measurement grid of time domain signals
- **center_signal** (*TimeSignal*) – Center measurement time domain signal and sampling frequency
- **configuration** (*ArrayConfiguration*) – Information on array configuration
- **temperature** (*array_like, optional*) – Temperature in room or at each sampling position

center_signal
Alias for field number 2

configuration
Alias for field number 3

count (*value*, /)
Return number of occurrences of value.

grid
Alias for field number 1

index (*value*, *start=0*, *stop=9223372036854775807*, /)
Return first index of value.

Raises ValueError if the value is not present.

signal
Alias for field number 0

temperature
Alias for field number 4

class sound_field_analysis.io.**HrirSignal** (*l*, *r*, *grid*, *center_signal=None*)
Named tuple to store Head Related Impulse Response grid data in terms of *TimeSignal* for either ear and *SphericalGrid*.

static __new__ (*cls*, *l*, *r*, *grid*, *center_signal=None*)

Parameters

- **l** (*TimeSignal*) – Left ear time domain signals and sampling frequency

- **r** (*TimeSignal*) – Right ear time domain signals and sampling frequency
- **grid** (*SphericalGrid*) – Measurement grid of time domain signals
- **center_signal** (*TimeSignal*) – Center measurement time domain signal and sampling frequency

`center_signal`
Alias for field number 3

`count` (*value*, */*)
Return number of occurrences of value.

`grid`
Alias for field number 2

`index` (*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

`l`
Alias for field number 0

`r`
Alias for field number 1

`sound_field_analysis.io.`**`read_miro_struct`** (*file_name*, *channel='irChOne'*, *transducer_type='omni'*, *scatter_radius=None*, *get_center_signal=False*)
Read Head Related Impulse Responses (HRIRs) or Array / Directional Impulse Responses (DRIRs) stored as MIRO Matlab files and convert them to *ArraySignal*.

Parameters

- **file_name** (*filepath*) – Path to file that has been exported as a struct
- **channel** (*string, optional*) – Channel that holds required signals. [Default: 'irChOne']
- **transducer_type** (*{omni, cardioid}, optional*) – Sets the type of transducer used in the recording. [Default: 'omni']
- **scatter_radius** (*float, option*) – Radius of the scatterer. [Default: None]
- **get_center_signal** (*bool, optional*) – If center signal should be loaded. [Default: False]

Returns **array_signal** (*ArraySignal*) – Tuple containing a TimeSignal *signal*, SphericalGrid *grid*, TimeSignal 'center_signal', ArrayConfiguration *configuration* and the air temperature

Notes

This function expects a slightly modified miro file in that it expects a field *colatitude* instead of *elevation*. This is for avoiding confusion as may miro file contain colatitude data in the elevation field.

To import center signal measurements the matlab method miro_to_struct has to be extended. Center measurements are included in every measurement provided at http://audiogroup.web.th-koeln.de/.

`sound_field_analysis.io.`**`read_SOFA_file`** (*file_name*)
Read Head Related Impulse Responses (HRIRs) or Array / Directional Impulse Response (DRIRs) stored as Spatially Oriented Format for Acoustics (SOFA) files and convert them to`ArraySignal` or *HrirSignal*.

Parameters **file_name** (*filepath*) – Path to SOFA file

Returns *ArraySignal or HRIRSignal* – Names tuples containing a the loaded file contents

Raises

- **`NotImplementedError`** – In case SOFA conventions other then "Simple-FreeFieldHRIR" or "SingleRoomDRIR" should be loaded

- `ValueError` – In case source / receiver grid given in units not according to the SOFA convention
- `ValueError` – In case impulse response data is incomplete

`sound_field_analysis.io.`**`empty_time_signal`**(*no_of_signals*, *signal_length*)

Returns an empty np rec array that has the proper data structure.

Parameters

- **no_of_signals** (*int*) – Number of signals to be stored in the recarray
- **signal_length** (*int*) – Length of the signals to be stored in the recarray

Returns

- **time_data** (*recarray*) – Structured array with following fields:
- *::* – .signal [Channels X Samples] .fs Sampling frequency in [Hz] .azimuth Azimuth of sampling points .colatitude Colatitude of sampling points .radius Array radius in [m] .grid_weights Weights of quadrature .air_temperature Average temperature in [C]

`sound_field_analysis.io.`**`load_array_signal`**(*filename*)

Convenience function to load ArraySignal saved into np data structures.

Parameters **filename** (*string*) – File to load

Returns **Y** (*ArraySignal*) – See io.ArraySignal

`sound_field_analysis.io.`**`read_wavefile`**(*filename*)

Reads in WAV files and returns data [Nsig x Nsamples] and fs.

Parameters **filename** (*string*) – Filename of wave file to be read

Returns

- **data** (*array_like*) – Data of dim [Nsig x Nsamples]
- **fs** (*int*) – Sampling frequency of read data

`sound_field_analysis.io.`**`write_SSR_IRs`**(*filename*, *time_data_l*, *time_data_r*, *wavformat='float32'*)

Takes two time signals and writes out the horizontal plane as HRIRs for the SoundScapeRenderer. Ideally, both hold 360 IRs but smaller sets are tried to be scaled up using repeat.

Parameters

- **filename** (*string*) – filename to write to
- **time_data_l, time_data_r** (*io.ArraySignal*) – ArraySignals for left/right ear
- **wavformat** (*{float32, int32, int16}, optional*) – wav file format to write [Default: float32]

Raises

- `ValueError` – in case unknown wavformat is provided
- `ValueError` – in case integer format should be exported and amplitude exceeds 1.0

3.3 Lebedev

Module to generate Lebedev grids and quadrature weights for degrees 6, 14, 26, 38, 50, 74, 86, 110, 146, 170, 194:

genGrid(n) Generate Lebedev grid geometry of degree *n*.

Adapted from Richard P. Mullers Python version, https://github.com/gabrielelanaro/pyquante/blob/master/Data/lebedev_write.py C version: Dmitri Laikov F77 version: Christoph van Wuellen, http://www.ccl.net

Users of this code are asked to include reference[3] in their publications, and in the user- and programmers-manuals describing their codes.

References

`sound_field_analysis.lebedev.`**`genGrid`**`(`*n*`)`

Generate Lebedev grid geometry of degree *n*.

Parameters **n** (*int{6, 14, 26, 38, 50, 74, 86, 110, 146, 170, 194}*) – Lebedev degree

Returns **lebGrid** (*named tuple*) – Named tuple to store *x*, *y*, *z* cartesian coordinates and quadrature weights *w*

Raises **`ValueError`** – in case no grid could be generated for given degree

3.4 Plotting

Module containing various plotting functions:

makeMTX Returns a plane wave decomposition over a full sphere.

makeFullMTX Generates visualization matrix for a set of spatial fourier coefficients over all kr.

plot2D Visualize 2D data using plotly.

plot3D Visualize matrix data, such as from *makeMTX(Pnm, dn)*.

plot3Dgrid Visualize matrix data in a grid, such as from *makeMTX(Pnm, dn)*.

`sound_field_analysis.plot.`**`makeMTX`**`(`*spat_coeffs*, *radial_filter*, *kr_IDX*, *viz_order=None*, *stepsize_deg=1*, *kind='complex'*`)`

Returns a plane wave decomposition over a full sphere.

Parameters

- **spat_coeffs** (*array_like*) – Spatial fourier coefficients
- **radial_filter** (*array_like*) – Modal radial filters
- **kr_IDX** (*int*) – Index of kr to be computed
- **viz_order** (*int, optional*) – Order of the spatial fourier transform [Default: Highest available]
- **stepsize_deg** (*float, optional*) – Integer Factor to increase the resolution. [Default: 1]
- **kind** (*{'complex', 'real'}, optional*) – Spherical harmonic coefficients data type [Default: 'complex']

Returns **mtxData** (*array_like*) – Plane wave decomposition (frequency domain)

[3] V.I. Lebedev, and D.N. Laikov 'A quadrature formula for the sphere of the 131st algebraic order of accuracy' Doklady Mathematics, Vol. 59, No. 3, 1999, pp. 477-481.

Note: The file generates a Matrix of 181x360 pixels for the visualisation with visualize3D() in 1[deg] Steps (65160 plane waves).

`sound_field_analysis.plot.`**`makeFullMTX`**(*Pnm*, *dn*, *kr*, *viz_order=None*)

Generates visualization matrix for a set of spatial fourier coefficients over all kr.

Parameters

- **Pnm** (*array_like*) – Spatial Fourier Coefficients (e.g. from S/T/C)
- **dn** (*array_like*) – Modal Radial Filters (e.g. from M/F)
- **kr** (*array_like*) – kr-vector
- **viz_order** (*int, optional*) – Order of the spatial fourier tplane_wave_decompransform [Default: Highest available]

Returns **vizMtx** (*array_like*) – Computed visualization matrix over all kr

`sound_field_analysis.plot.`**`plot2D`**(*data*, *title=None*, *viz_type=None*, *fs=44100*, *line_names=None*)

Visualize 2D data using plotly.

Parameters

- **data** (*array_like*) – Data to be plotted, separated along the first dimension (rows)
- **title** (*str, optional*) – Add title to be displayed on plot
- **viz_type** (*str{None, 'Time', 'ETC', 'LinFFT', 'LogFFT'}, optional*) – Type of data to be displayed [Default: None]
- **fs** (*int, optional*) – Sampling rate in Hz [Default: 44100]
- **line_names** (*list of str, optional*) – Add legend to be displayed on plot, with one entry for each data row [Default: None]

`sound_field_analysis.plot.`**`plot3D`**(*vizMTX*, *style='shape'*, *layout=None*, *normalize=True*, *logScale=False*)

Visualize matrix data, such as from *makeMTX(Pnm, dn)*.

Parameters

- **vizMTX** (*array_like*) – Matrix holding spherical data for visualization
- **layout** (*plotly.graph_objs.Layout, optional*) – Layout of plot to be displayed offline
- **style** (*string{'shape', 'sphere', 'flat'}, optional*) – Style of visualization. [Default: 'shape']
- **normalize** (*bool, optional*) – Toggle normalization of data to [-1 … 1] [Default: True]
- **logScale** (*bool, optional*) – Toggle conversion logScale [Default: False]

Returns *None*

Todo: Add colorization and contour plots

`sound_field_analysis.plot.`**`plot3Dgrid`**(*rows*, *cols*, *viz_data*, *style*, *normalize=True*, *title=None*)

Visualize matrix data in a grid, such as from *makeMTX(Pnm, dn)*.

Parameters

- **rows** (*int*) – Number of grid rows
- **cols** (*int*) – Number of grid columns
- **viz_data** (*array_like*) – Matrix holding data for visualization

- **style** (*string{'shape', 'sphere', 'flat'}, optional*) – Style of visualization. [Default: 'shape']
- **normalize** (*bool, optional*) – Toggle normalization of data to [-1 … 1] [Default: True]
- **title** (*str, optional*) – Add title to be displayed on plot

3.5 Processing

Module containing functions to act on the Spatial Fourier Coefficients.

BEMA Perform Bandwidth Extension for Microphone Arrays (BEMA) spatial anti-aliasing.

FFT Perform real-valued Fast Fourier Transform (FFT).

iFFT Perform inverse real-valued (Fast) Fourier Transform (iFFT).

spatFT Perform spatial Fourier transform.

spatFT_RT Perform spatial Fourier transform for real-time applications (otherwise use *spatFT()* for more more convenience and flexibility).

spatFT_LSF Perform spatial Fourier transform by least square fit to provided data.

iSpatFT Perform inverse spatial Fourier transform.

plane_wave_decomp Perform plane wave decomposition.

rfi Perform radial filter improvement (RFI)

sfe Perform sound field extrapolation (SFE) - WORK IN PROGRESS.

wdr Perform Wigner-D rotation (WDR) - NOT YET IMPLEMENTED.

convolve Convolve two arrays A & B row-wise where one or both can be one-dimensional for SIMO/SISO convolution.

`sound_field_analysis.process.`**`BEMA`**(*Pnm*, *center_sig*, *dn*, *transition*, *avg_band_width=1*, *fade=True*, *max_order=None*)

Perform Bandwidth Extension for Microphone Arrays (BEMA) spatial anti-aliasing, according to[4].

Parameters

- **Pnm** (*array_like*) – Sound field SH spatial Fourier coefficients
- **center_sig** (*array_like*) – Center microphone in shape [0, NFFT]
- **dn** (*array_like*) – Radial filters for the current array configuration
- **transition** (*int*) – Highest stable bin, approx: transition = (NFFT/fs) * (N*c)/(2*pi*r)
- **avg_band_width** (*int, optional*) – Averaging Bandwidth in oct [Default: 1]
- **fade** (*bool, optional*) – Fade over if True, else hard cut [Default: True]
- **max_order** (*int, optional*) – Maximum transform order [Default: highest available order]

Returns **Pnm** (*array_like*) – BEMA optimized sound field SH coefficients

References

[4] B. Bernschütz, "Bandwidth Extension for Microphone Arrays", AES Convention 2012, Convention Paper 8751, 2012. http://www.aes.org/e-lib/browse.cfm?elib=16493

`sound_field_analysis.process.`**`FFT`**(*time_signals*, *fs=None*, *NFFT=None*, *oversampling=1*, *first_sample=0*, *last_sample=None*, *calculate_freqs=True*)

Perform real-valued Fast Fourier Transform (FFT).

Parameters

- **time_signals** (*TimeSignal/tuple/object*) – Time-domain signals to be transformed.
- **fs** (*int, optional*) – Sampling frequency - only optional no frequency vector should be calculated or if a TimeSignal or tuple/array containing fs is passed
- **NFFT** (*int, optional*) – Number of frequency bins. Resulting array will have size NFFT//2+1 [Default: Next power of 2]
- **oversampling** (*int, optional*) – Oversample the incoming signal to increase frequency resolution [Default: 1]
- **first_sample** (*int, optional*) – First time domain sample to be included [Default: 0]
- **last_sample** (*int, optional*) – Last time domain sample to be included [Default: -1]
- **calculate_freqs** (*bool, optional*) – Calculate frequency scale if True, else return only spectrum [Default: True]

Returns

- **fftData** (*array_like*) – One-sided frequency domain spectrum
- **f** (*array_like, optional*) – Vector of frequency bins of one-sided spectrum, in case of calculate_freqs

Notes

An oversampling*NFFT point Fourier Transform is applied to the time domain data, where NFFT is the next power of two of the number of samples. Time-windowing can be used by providing a first_sample and last_sample index.

`sound_field_analysis.process.`**`iFFT`**(*Y*, *output_length=None*, *window=False*)

Perform inverse real-valued (Fast) Fourier Transform (iFFT).

Parameters

- **Y** (*array_like*) – Frequency domain data [Nsignals x Nbins]
- **output_length** (*int, optional*) – Length of returned time-domain signal (Default: 2 x len(Y) + 1)
- **window** (*str, optional*) – Window applied to the resulting time-domain signal

Returns **y** (*array_like*) – Reconstructed time-domain signal

`sound_field_analysis.process.`**`spatFT`**(*data*, *position_grid*, *order_max=10*, *kind='complex'*, *spherical_harmonic_bases=None*)

Perform spatial Fourier transform.

Parameters

- **data** (*array_like*) – Data to be transformed, with signals in rows and frequency bins in columns
- **position_grid** (*array_like or io.SphericalGrid*) – Azimuths/Colatitudes/Gridweights of spatial sampling points
- **order_max** (*int, optional*) – Maximum transform order [Default: 10]
- **kind** (*{'complex', 'real'}, optional*) – Spherical harmonic coefficients data type [Default: 'complex']

- **spherical_harmonic_bases** (*array_like, optional*) – Spherical harmonic base coefficients (not yet weighted by spatial sampling grid) [Default: None]

Returns **Pnm** (*array_like*) – Spatial Fourier Coefficients with nm coeffs in rows and FFT bins in columns

Notes

In case no weights in spatial sampling grid are given, the pseudo inverse of the SH bases is computed according to Eq. 3.34 in[5].

References

`sound_field_analysis.process.`**`spatFT_RT`**(*data*, *spherical_harmonic_weighted*)

Perform spatial Fourier transform for real-time applications (otherwise use *spatFT()* for more more convenience and flexibility).

Parameters

- **data** (*array_like*) – Data to be transformed, with signals in rows and frequency bins in columns
- **spherical_harmonic_weighted** (*array_like*) – Spherical harmonic base coefficients (already weighted by spatial sampling grid)

Returns **Pnm** (*array_like*) – Spatial Fourier Coefficients with nm coeffs in rows and FFT bins in columns

`sound_field_analysis.process.`**`spatFT_LSF`**(*data*, *position_grid*, *order_max=10*, *kind='complex'*, *spherical_harmonic_bases=None*)

Perform spatial Fourier transform by least square fit to provided data.

Parameters

- **data** (*array_like, complex*) – Data to be fitted to
- **position_grid** (*array_like, or io.SphericalGrid*) – Azimuth / colatitude data locations
- **order_max** (*int, optional*) – Maximum transform order [Default: 10]
- **kind** (*{'complex', 'real'}, optional*) – Spherical harmonic coefficients data type [Default: 'complex']
- **spherical_harmonic_bases** (*array_like, optional*) – Spherical harmonic base coefficients (not yet weighted by spatial sampling grid) [Default: None]

Returns **coefficients** (*array_like, float*) – Fitted spherical harmonic coefficients (indexing: n**2 + n + m + 1)

`sound_field_analysis.process.`**`iSpatFT`**(*spherical_coefficients*, *position_grid*, *order_max=None*, *kind='complex'*, *spherical_harmonic_bases=None*)

Perform inverse spatial Fourier transform.

Parameters

- **spherical_coefficients** (*array_like*) – Spatial Fourier coefficients with columns representing frequency bins
- **position_grid** (*array_like or io.SphericalGrid*) – Azimuth/Colatitude angles of spherical coefficients

[5] Boaz Rafaely: Fundamentals of spherical array processing. In. Springer topics in signal processing. Benesty, J.; Kellermann, W. (Eds.), Springer, Heidelberg et al. (2015).

- **order_max** (*int, optional*) – Maximum transform order [Default: highest available order]
- **kind** (*{'complex', 'real'}, optional*) – Spherical harmonic coefficients data type [Default: 'complex']
- **spherical_harmonic_bases** (*array_like, optional*) – Spherical harmonic base coefficients (not yet weighted by spatial sampling grid) [Default: None]

Returns **P** (*array_like*) – Sound pressures with frequency bins in columns and angles in rows

Todo: Check *spherical_coefficients* and *spherical_harmonic_bases* length correspond with *order_max*

`sound_field_analysis.process.`**`plane_wave_decomp`**(*order*, *wave_direction*, *field_coeffs*, *radial_filter*, *weights=None*, *kind='complex'*)

Perform plane wave decomposition.

Parameters

- **order** (*int*) – Decomposition order
- **wave_direction** (*array_like*) – Direction of plane wave as [azimuth, colatitude] pair. io.SphericalGrid is used internally
- **field_coeffs** (*array_like*) – Spatial fourier coefficients
- **radial_filter** (*array_like*) – Radial filters
- **weights** (*array_like, optional*) – Weighting function. Either scalar, one per directions or of dimension (nKR_bins x nDirections). [Default: None]
- **kind** (*{'complex', 'real'}, optional*) – Spherical harmonic coefficients data type [Default: 'complex']

Returns **Y** (*matrix of floats*) – Matrix of the decomposed wave field with kr bins in rows

`sound_field_analysis.process.`**`rfi`**(*dn*, *kernelSize=512*, *highPass=0.0*)

Perform radial filter improvement (RFI), according to[6].

Parameters

- **dn** (*array_like*) – Analytical frequency domain radial filters (e.g. *gen.radial_filter_fullspec()*)
- **kernelSize** (*int, optional*) – Target filter kernel size [Default: 512]
- **highPass** (*float, optional*) – Highpass Filter from 0.0 (off) to 1.0 (maximum kr) [Default: 0.0]

Returns

- **dn** (*array_like*) – Improved radial filters
- **kernelSize** (*int*) – Filter kernel size (total)
- **latency** (*float*) – Approximate signal latency due to the filters

Notes

This function improves the FIR radial filters from *gen.radial_filter_fullspec()*. The filters are made causal and are windowed in time domain. The DC components are estimated. The R/F/I module should always be inserted to the filter path when treating measured data even if no use is made of the included kernel downscaling or highpass filters.

Do NOT use R/F/I for single open sphere filters (e.g. simulations).

[6] B. Bernschütz, C. Pörschmann, S. Spors, and S. Weinzierl, "SOFiA Sound Field Analysis Toolbox," in International Conference on Spatial Audio, 2011, pp. 7–15.

IMPORTANT Remember to choose a kernel size being large enough to cover all filter latencies and response slopes. Otherwise undesired cyclic convolution artifacts may appear in the output signal.

HIGHPASS If HPF is on (0<highPass<=1) the radial filters and HPF share the available taps and the latency keeps constant. Be careful when using very small kernel sizes because since there might be too few taps. Observe the filters by plotting their spectra and impulse responses! > Be very careful if NFFT/max(kr) < 25 > Do not use R/F/I if NFFT/max(kr) < 15

References

Todo: Implement *kernelsize* extension

`sound_field_analysis.process.`**`sfe`**(*Pnm_kra*, *kra*, *krb*, *problem='interior'*)

Perform sound field extrapolation (SFE) - WORK IN PROGRESS.

Parameters

- **Pnm_kra** (*array_like*) – Spatial Fourier coefficients (e.g. from *spatFT()*)
- **kra,krb** (*array_like*) – k * ra/rb vector
- **problem** (*string{'interior', 'exterior'}*) – Select between interior and exterior problem [Default: interior]

Returns **Pnm_kra** (*array_like*) – Extrapolated spatial Fourier coefficients

---

Todo: Verify sound field extrapolation

`sound_field_analysis.process.`**`wdr`**(*Pnm*, *xAngle*, *yAngle*, *zAngle*)

Perform Wigner-D rotation (WDR) - NOT YET IMPLEMENTED.

Parameters

- **Pnm** (*array_like*) – Spatial Fourier coefficients
- **xAngle, yAngle, zAngle** (*float*) – Rotation angle around the x/y/z-Axis

Returns **PnmRot** (*array_like*) – Rotated spatial Fourier coefficients

---

Todo: Implement Wigner-D rotations

`sound_field_analysis.process.`**`convolve`**(*A*, *B*, *FFT=None*)

Convolve two arrays A & B row-wise where one or both can be one-dimensional for SIMO/SISO convolution.

Parameters

- **A, B** (*array_like*) – Data to perform the convolution on of shape [Nsignals x NSamples]
- **FFT** (*bool, optional*) – Selects whether time or frequency domain convolution is applied. [Default: On if Nsamples > 500 for both]

Returns **out** (*array*) – Array containing row-wise, linear convolution of A and B

3.6 Spherical

Module containing various spherical harmonics helper functions:

besselj* / *neumann Bessel function of first and second kind of order n at kr.

hankel1* / *hankel2 Hankel function of first and second kind of order n at kr.

spbessel* / *spneumann Spherical Bessel function of first and second kind of order n at kr.

sphankel1* / *sphankel2 Spherical Hankel of first and second kind of order n at kr.

dspbessel* / *dspneumann Derivative of spherical Bessel of first and second kind of order n at kr.

dsphankel1* / *dsphankel2 Derivative spherical Hankel of first and second kind of order n at kr.

spherical_extrapolation Factor that relate signals recorded on a sphere to it's center.

array_extrapolation Factor that relate signals recorded on a sphere to it's center. In the rigid configuration, a scatter_radius that is different to the array radius may be set.

sph_harm Compute spherical harmonics.

sph_harm_large Compute spherical harmonics for large orders > 84.

sph_harm_all Compute all spherical harmonic coefficients up to degree nMax.

mnArrays Generate degrees n and orders m up to nMax.

reverseMnIds Generate reverse indexes according to stacked coefficients of orders m up to nMax.

cart2sph* / *sph2cart Convert cartesian to spherical coordinates and vice versa.

kr Generate kr vector for given f and array radius.

kr_full_spec Generate full spectrum kr.

`sound_field_analysis.sph.`**`besselj`**`(`*n*, *z*`)`

Bessel function of first kind of order n at kr. Wraps *scipy.special.jn(n, z)*.

Parameters

- **n** (*array_like*) – Order
- **z** (*array_like*) – Argument

Returns **J** (*array_like*) – Values of Bessel function of order n at position z

`sound_field_analysis.sph.`**`neumann`**`(`*n*, *z*`)`

Bessel function of second kind (Neumann / Weber function) of order n at kr. Implemented as *(hankel1(n, z) - besselj(n, z)) / 1j*.

Parameters

- **n** (*array_like*) – Order
- **z** (*array_like*) – Argument

Returns **Y** (*array_like*) – Values of Hankel function of order n at position z

`sound_field_analysis.sph.`**`hankel1`**`(`*n*, *z*`)`

Hankel function of first kind of order n at kr. Wraps *scipy.special.hankel2(n, z)*.

Parameters

- **n** (*array_like*) – Order
- **z** (*array_like*) – Argument

Returns **H1** (*array_like*) – Values of Hankel function of order n at position z

`sound_field_analysis.sph.`**`hankel2`**`(`*n*, *z*`)`

Hankel function of second kind of order n at kr. Wraps *scipy.special.hankel2(n, z)*.

Parameters

- **n** (*array_like*) – Order
- **z** (*array_like*) – Argument

Returns **H2** (*array_like*) – Values of Hankel function of order n at position z

`sound_field_analysis.sph.`**`spbessel`**(*n*, *kr*)

Spherical Bessel function (first kind) of order n at kr.

Parameters

- **n** (*array_like*) – Order
- **kr** (*array_like*) – Argument

Returns **J** (*complex float*) – Spherical Bessel

`sound_field_analysis.sph.`**`spneumann`**(*n*, *kr*)

Spherical Neumann (Bessel second kind) of order n at kr.

Parameters

- **n** (*array_like*) – Order
- **kr** (*array_like*) – Argument

Returns **Yv** (*complex float*) – Spherical Neumann (Bessel second kind)

`sound_field_analysis.sph.`**`sphankel1`**(*n*, *kr*)

Spherical Hankel (first kind) of order n at kr.

Parameters

- **n** (*array_like*) – Order
- **kr** (*array_like*) – Argument

Returns **hn1** (*complex float*) – Spherical Hankel function hn (first kind)

`sound_field_analysis.sph.`**`sphankel2`**(*n*, *kr*)

Spherical Hankel (second kind) of order n at kr

Parameters

- **n** (*array_like*) – Order
- **kr** (*array_like*) – Argument

Returns **hn2** (*complex float*) – Spherical Hankel function hn (second kind)

`sound_field_analysis.sph.`**`dspbessel`**(*n*, *kr*)

Derivative of spherical Bessel (first kind) of order n at kr.

Parameters

- **n** (*array_like*) – Order
- **kr** (*array_like*) – Argument

Returns **J'** (*complex float*) – Derivative of spherical Bessel

`sound_field_analysis.sph.`**`dspneumann`**(*n*, *kr*)

Derivative spherical Neumann (Bessel second kind) of order n at kr.

Parameters

- **n** (*array_like*) – Order
- **kr** (*array_like*) – Argument

Returns **Yv'** (*complex float*) – Derivative of spherical Neumann (Bessel second kind) `sound_field_analysis.sph.`**`dsphankel1`**(*n*, *kr*)

Derivative spherical Hankel (first kind) of order n at kr.

Parameters

- **n** (*array_like*) – Order
- **kr** (*array_like*) – Argument

Returns **dhn1** (*complex float*) – Derivative of spherical Hankel function hn' (second kind)

`sound_field_analysis.sph.`**`dsphankel2`**(*n*, *kr*)

Derivative spherical Hankel (second kind) of order n at kr.

Parameters

- **n** (*array_like*) – Order
- **kr** (*array_like*) – Argument

Returns **dhn2** (*complex float*) – Derivative of spherical Hankel function hn' (second kind)

`sound_field_analysis.sph.`**`spherical_extrapolation`**(*order*, *array_configuration*, *k_mic*, *k_scatter=None*, *k_dual=None*)

Factor that relate signals recorded on a sphere to it's center.

Parameters

- **order** (*int*) – Order
- **array_configuration** (*io.ArrayConfiguration*) – List/Tuple/ArrayConfiguration, see io.ArrayConfiguration
- **k_mic** (*array_like*) – K vector for microphone array
- **k_scatter** (*array_like, optional*) – K vector for scatterer [Default: same as k_mic]
- **k_dual** (*float, optional*) – Radius of second array, required for *array_type* == 'dual'

Returns **b** (*array, complex*)

`sound_field_analysis.sph.`**`array_extrapolation`**(*order*, *freqs*, *array_configuration*, *normalize=True*)

Factor that relate signals recorded on a sphere to it's center. In the rigid configuration, a scatter_radius that is different to the array radius may be set.

Parameters

- **order** (*int*) – Order
- **freqs** (*array_like*) – Frequencies
- **array_configuration** (*io.ArrayConfiguration*) – List/Tuple/ArrayConfiguration, see io.ArrayConfiguration
- **normalize** (*Bool, optional*) – Normalize by 4 * pi * 1j ** order [Default: True]

Returns **b** (*array, complex*) – Coefficients of shape [nOrder x nFreqs]

`sound_field_analysis.sph.`**`bn_open_omni`**(*n*, *krm*)

`sound_field_analysis.sph.`**`bn_open_cardioid`**(*n*, *krm*)

`sound_field_analysis.sph.`**`bn_rigid_omni`**(*n*, *krm*, *krs*)

`sound_field_analysis.sph.`**`bn_rigid_cardioid`**(*n*, *krm*, *krs*)

`sound_field_analysis.sph.`**`bn_dual_open_omni`**(*n*, *kr1*, *kr2*)

`sound_field_analysis.sph.`**`sph_harm`**(*m*, *n*, *az*, *co*, *kind='complex'*)

Compute spherical harmonics.

Parameters

- **m** (*(int)*) – Order of the spherical harmonic. abs(m) <= n
- **n** (*(int)*) – Degree of the harmonic, sometimes called l. n >= 0
- **az** (*(float)*) – Azimuthal (longitudinal) coordinate [0, 2pi], also called Theta.
- **co** (*(float)*) – Polar (colatitudinal) coordinate [0, pi], also called Phi.
- **kind** (*{'complex', 'real'}, optional*) – Spherical harmonic coefficients data type according to complex[7] or real definition[8] [Default: 'complex']

Returns **y_mn** (*(complex float) or (float)*) – Spherical harmonic of order m and degree n, sampled at theta = az, phi = co

References

`sound_field_analysis.sph.`**`sph_harm_large`**(*m*, *n*, *az*, *co*, *kind='complex'*)

Compute spherical harmonics for large orders > 84.

Parameters

- **m** (*(int)*) – Order of the spherical harmonic. abs(m) <= n
- **n** (*(int)*) – Degree of the harmonic, sometimes called l. n >= 0
- **az** (*(float)*) – Azimuthal (longitudinal) coordinate [0, 2pi], also called Theta.
- **co** (*(float)*) – Polar (colatitudinal) coordinate [0, pi], also called Phi.
- **kind** (*{'complex', 'real'}, optional*) – Spherical harmonic coefficients data type according to complex[6] or real definition[7] [Default: 'complex']

Returns **y_mn** (*(complex float) or (float)*) – Spherical harmonic of order m and degree n, sampled at theta = az, phi = co

Notes

Y_n,m (theta, phi) = ((n - m)! * (2l + 1)) / (4pi * (l + m))^0.5 * exp(i m phi) * P_n^m(cos(theta)) as per http://dlmf.nist.gov/14.30 Pmn(z) are the associated Legendre functions of the first kind, like scipy.special.lpmv, which calculates P(0…m 0…n) and its derivative but won't return +inf at high orders.

Todo: Confirm that the correct SH definition is used

`sound_field_analysis.sph.`**`sph_harm_all`**(*nMax*, *az*, *co*, *kind='complex'*)

Compute all spherical harmonic coefficients up to degree nMax.

Parameters

- **nMax** (*(int)*) – Maximum degree of coefficients to be returned. n >= 0
- **az** (*(float), array_like*) – Azimuthal (longitudinal) coordinate [0, 2pi], also called Theta.
- **co** (*(float), array_like*) – Polar (colatitudinal) coordinate [0, pi], also called Phi.
- **kind** (*{'complex', 'real'}, optional*) – Spherical harmonic coefficients data type [Default: 'complex']

Returns **y_mn** (*(complex float) or (float), array_like*) – Spherical harmonics of degrees n [0 … nMax] and all corresponding orders m [-n … n], sampled at [az, co]. dim1 corresponds to az/co pairs, dim2 to oder/degree (m, n) pairs like 0/0, -1/1, 0/1, 1/1, -2/2, -1/2 …

[7] scipy.special.sph_harm

[8] F. Zotter, "Analysis and synthesis of sound-radiation with spherical arrays," University of Music and Performing Arts, 2009.

`sound_field_analysis.sph.`**`mnArrays`**(*nMax*)
Generate degrees n and orders m up to nMax.

Parameters **nMax** (*(int)*) – Maximum degree of coefficients to be returned. n >= 0

Returns

- **m** (*(int), array_like*) – 0, -1, 0, 1, -2, -1, 0, 1, 2, … , -nMax …, nMax
- **n** (*(int), array_like*) – 0, 1, 1, 1, 2, 2, 2, 2, 2, … nMax, nMax, nMax

`sound_field_analysis.sph.`**`reverseMnIds`**(*nMax*)
Generate reverse indexes according to stacked coefficients of orders m up to nMax.

Parameters **nMax** (*(int)*) – Maximum degree of coefficients reverse indexes to be returned. n >= 0

Returns **rev_ids** (*(int), array_like*) – 0, 3, 2, 1, 8, 7, 6, 5, 4, …

`sound_field_analysis.sph.`**`cart2sph`**(*x*, *y*, *z*)
Convert cartesian coordinates x, y, z to spherical coordinates az, el, r.

`sound_field_analysis.sph.`**`sph2cart`**(*az*, *el*, *r*)
Convert spherical coordinates az, el, r to cartesian coordinates x, y, z.

`sound_field_analysis.sph.`**`kr`**(*f*, *radius*, *temperature=20*)
Generate kr vector for given f and array radius.

Parameters

- **f** (*array_like*) – Frequencies to calculate the kr for
- **radius** (*float*) – Radius of array
- **temperature** (*float, optional*) – Room temperature in degree Celsius [Default: 20]

Returns **kr** (*array_like*) – 2 * pi * f / c(temperature) * r

`sound_field_analysis.sph.`**`kr_full_spec`**(*fs*, *radius*, *NFFT*, *temperature=20*)
Generate full spectrum kr.

Parameters

- **fs** (*int*) – Sampling rate in Hertz
- **radius** (*float*) – Radius
- **NFFT** (*int*) – Number of frequency bins
- **temperature** (*float, optional*) – Temperature in degree Celsius [Default: 20 C]

Returns **kr** (*array_like*) – kr vector of length NFFT/2 + 1 spanning the frequencies of 0:fs/2

3.7 Utilities

Module containing various utility functions:

env_info Guess environment based on *sys.modules*.

progress_bar Display a spinner or a progress bar.

db Convenience function to calculate the 20*log10(abs(x)).

cart2sph / sph2cart Transform cartesian into spherical coordinates and vice versa.

SOFA_grid2acr Transform coordinate grid with specified coordinate system definition from a SOFA file into spherical coordinates in radians.

nearest_to_value / nearest_to_value_IDX / nearest_to_value_logical_IDX Returns nearest value inside an array.

interleave_channels Interleave left and right channels. Style == 'SSR' checks if we total 360 channels.

simple_resample Wrap scipy.signal.resample with a simpler API.

scalar_broadcast_match Returns arguments as np.array, if one is a scalar it will broadcast the other one's shape.

frq2kr Returns the kr bin closest to the target frequency.

stack Stacks two 2D vectors along the same-sized dimension or the smaller one.

zero_pad_fd Apply zero padding to frequency domain data by transformation into time domain and back.

current_time Return current system time based on *datetime.now()*.

time_it Measure execution of a specified statement which is useful for the performance analysis. In this way, the execution time and respective results can be directly compared.

`sound_field_analysis.utils.`**`env_info`**`()`

Guess environment based on *sys.modules*.

Returns env (*string{'jupyter_notebook', 'ipython_terminal', 'terminal'}*) – Guessed environment

`sound_field_analysis.utils.`**`progress_bar`**`(`*curIDX*, *maxIDX=None*, *description='Progress'*`)`

Display a spinner or a progress bar.

Parameters

- **curIDX** (*int*) – Current position in the loop
- **maxIDX** (*int, optional*) – Number of iterations. Will force a spinner if set to None. [Default: None]
- **description** (*string, optional*) – Clarify what's taking time

`sound_field_analysis.utils.`**`db`**`(`*data*, *power=False*`)`

Convenience function to calculate the 20*log10(abs(x)).

Parameters

- **data** (*array_like*) – signals to be converted to db
- **power** (*boolean*) – data is a power signal and only needs factor 10

Returns db (*array_like*) – 20 * log10(abs(data))

`sound_field_analysis.utils.`**`cart2sph`**`(`*cartesian_coords*, *is_deg=False*`)`

Transform cartesian into spherical coordinates.

Parameters

- **cartesian_coords** (*numpy.ndarray*) – cartesian coordinates (x, y, z) of size [3; number of coordinates]
- **is_deg** (*bool, optional*) – if values should be calculated in degrees (radians otherwise) [Default: False]

Returns *numpy.ndarray* – spherical coordinates (azimuth [0 … 2pi or 0 … 360deg], colatitude [0 … pi or 0 … 180deg], radius [meter]) of size [3; number of coordinates]

`sound_field_analysis.utils.`**`sph2cart`**`(`*spherical_coords*, *is_deg=False*`)`

Transform spherical into cartesian coordinates.

Parameters

- **spherical_coords** (*numpy.ndarray*) – spherical coordinates (azimuth, colatitude, radius) of size [3; number of coordinates]
- **is_deg** (*bool, optional*) – True if values are given in degrees (radians otherwise) [Default: False]

Returns *numpy.ndarray* – cartesian coordinates (x, y, z) of size [3; number of coordinates] `sound_field_analysis.utils.`**`SOFA_grid2acr`**(*grid_values*, *grid_info*)

Transform coordinate grid with specified coordinate system definition from a SOFA file into spherical coordinates in radians.

Parameters

- **grid_values** (*numpy.ndarray*) – Coordinates either spherical or cartesian of size [3; number of coordinates]
- **grid_info** (*list of str*) – Definition of coordinate system contained in the provided values according to SOFA convention, i.e. either ('degree, degree, metre', 'spherical') or ('metre, metre, metre', 'cartesian')

Returns *numpy.ndarray* – Spherical coordinates (azimuth [0 … 2pi], colatitude [0 … pi], radius [meter]) of size [3; number of coordinates]

Raises **`ValueError`** – In case unknown coordinate system definition is given

Notes

This is used for source position of "SimpleFreeFieldHRIR" and receiver position of "SingleRoomDRIR". These conventions technically require different specific coordinate systems. Experience showed, that this is not exactly met by all SOFA files, hence cartesian or spherical coordinates will be transformed in either case.

Todo: Validate data units against individual convention in *pysofaconventions*

`sound_field_analysis.utils.`**`nearest_to_value_IDX`**(*array*, *target_val*)

Returns nearest value inside an array.

`sound_field_analysis.utils.`**`nearest_to_value`**(*array*, *target_val*)

Returns nearest value inside an array.

`sound_field_analysis.utils.`**`nearest_to_value_logical_IDX`**(*array*, *target_val*)

Returns logical indices of nearest values inside array.

`sound_field_analysis.utils.`**`interleave_channels`**(*left_channel*, *right_channel*, *style=None*)

Interleave left and right channels. Style == 'SSR' checks if we total 360 channels.

`sound_field_analysis.utils.`**`simple_resample`**(*data*, *original_fs*, *target_fs*)

Wrap scipy.signal.resample with a simpler API.

`sound_field_analysis.utils.`**`scalar_broadcast_match`**(*a*, *b*)

Returns arguments as np.array, if one is a scalar it will broadcast the other one's shape.

`sound_field_analysis.utils.`**`frq2kr`**(*target_frequency*, *freq_vector*)

Returns the kr bin closest to the target frequency.

Parameters

- **target_frequency** (*float*) – Target frequency
- **freq_vector** (*array_like*) – Array containing the available frequencys

Returns **krTarget** (*int*) – kr bin closest to target frequency

`sound_field_analysis.utils.`**`stack`**(*vector_1*, *vector_2*)

Stacks two 2D vectors along the same-sized dimension or the smaller one.

`sound_field_analysis.utils.`**`zero_pad_fd`**(*data_fd*, *target_length_td*)

Apply zero padding to frequency domain data by transformation into time domain and back.

Parameters

- **data_fd** (*numpy.ndarray*) – Single-sided spectrum

- **target_length_td** (*int*) – target length of time domain representation in samples

Returns *numpy.ndarray* – Zero padded single-sided spectrum

`sound_field_analysis.utils.`**`current_time`**`()`

Return current system time based on *datetime.now()*.

`sound_field_analysis.utils.`**`get_named_tuple__repr__`**`(`*namedtuple*`)`

`sound_field_analysis.utils.`**`time_it`**`(`*description*, *stmt*, *setup*, *_globals*, *repeat*, *number*, *reference=None*, *check_dtype=None*`)`

Measure execution of a specified statement which is useful for the performance analysis. In this way, the execution time and respective results can be directly compared.

Parameters

- **description** (*str*) – Descriptive name of configuration to be shown in the log
- **stmt** (*str*) – Statement to be timed
- **setup** (*str*) – Additional statement used for setup
- **_globals** – Namespace the code will be executed in (as opposed to inside timeit's namespace)
- **repeat** (*int*) – How many times to repeat the timeit measurement (results will be gathered as a list)
- **number** (*int*) – How many times to execute the statement to be timed
- **reference** (*list of (float, any), optional*) – Result with the same data structure as returned by this function, which will be referenced in order to compare execution time and similarity of the result data
- **check_dtype** (*str, optional*) – Data type of the result to check

Returns *(float, any)* – Tuple of execution time in seconds (minimum of all repetitions) and execution result of first repetition (data depends on the specified statement to be timed)

PYTHON MODULE INDEX

INDEX