Sound Field Analysis Toolbox Readme

Release 0.2

Christoph Hohnerlein (QU Lab)

Feb 12, 2017

CONTENTS

CHAPTER

ONE

USAGE

1.1 Requirements

Obviously, you'll need Python. The code has been developed and tested on Python 3.x only. NumPy and SciPy are needed for calculations. .. If you also want to plot the resulting sound fields, you'll need matplotlib.

Instead of installing all of them separately, you should probably get a Python distribution that already includes everything, e.g. Anaconda.

1.2 How to Get Started

Various jupyter notebooks are located in the root directory/

- **AE1_IdealPlaneWave.ipynb:** Ideal unity plane wave simulation

1.3 Reference

Feel free to check out the full *Reference*.

CHAPTER

TWO

GENERATORS

Generators explanation. Module contains various generator functions:

whiteNoise Generate additive White Gaussian noise

gaussGrid Gauss-Legendre quadrature grid and weights

lebedev Lebedev quadrature grid and weigths

radial_filter Modal radial filter

radial_filter_fullspec Modal radial filter over the full spectrum

sampledWave Sampled Wave generator, emulating discrete sampling

ideal_wave Ideal wave generator, returns spatial fourier coefficients

CHAPTER

THREE

PROCESSING

Processing explanation. Functions that act on the Spatial Fourier Coefficients

FFT (Fast) Fourier Transform

iFFT Inverse (Fast) Fourier Transform

spatFT Spatial Fourier Transform

iSpatFT Fast Inverse Spatial Fourier Transform

PWDecomp Plane Wave Decomposition

Not yet implemented: *BEMA*

BEMA Spatial Anti-Aliasing

rfi Radial filter improvement

sfe Sound field extrapolation

wdr Wigner-D Rotation

CHAPTER

FOUR

PLOTTING

Plotting explanation. Plotting functions Helps visualizing spherical microphone data.

Generally, you probably want to first extract the amplitude information in spherical coordinates: >> plot.makeMTX(Pnm, dn, Nviz=3, krIndex=1, oversize=1) And then visualize that: >> plot3D(vizMTX, style='shape')

Other valid styles are 'sphere' and 'flat'.

CHAPTER

FIVE

REFERENCE

5.1 Generators

Module contains various generator functions:

whiteNoise Generate additive White Gaussian noise

gaussGrid Gauss-Legendre quadrature grid and weights

lebedev Lebedev quadrature grid and weigths

radial_filter Modal radial filter

radial_filter_fullspec Modal radial filter over the full spectrum

sampledWave Sampled Wave generator, emulating discrete sampling

ideal_wave Ideal wave generator, returns spatial fourier coefficients

`sound_field_analysis.gen.`**`gaussGrid`**(*AZnodes=10*, *ELnodes=5*, *plot=False*)

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

Parameters

- **`ELnodes`** (*`AZnodes,`*) – Number of azimutal / elevation nodes [Default: 10 / 5]
- **`plot`** (*`bool, optional`*) – Show a globe plot of the selected grid [Default: False]

Returns

- **gridData** (*matrix of floats*) – Gauss-Legendre quadrature positions and weigths

```
[AZ_0, EL_0, W_0
     ...
AZ_n, EL_n, W_n]
```

- **Npoints** (*int*) – Total number of nodes
- **Nmax** (*int*) – Highest stable grid order

`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*)

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`** (`ArrayConfiguration`) – List/Tuple/ArrayConfiguration, see io.ArrayConfiguration

- **colatitude** (*azimuth,*) – 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 veolcity in m/s [Default: 343m/s]

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** – Spatial Fourier Coefficients with nm coeffs in cols and FFT coeffs in rows

Return type array of complex floats

`sound_field_analysis.gen.`**`lebedev`**(*max_order=None, degree=None*)

Compute Lebedev quadrature nodes and weigths 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** – Lebedev quadrature positions and weigths: [AZ, EL, W]

Return type array_like

`sound_field_analysis.gen.`**`radial_filter`**(*order, freq, array_configuration, amp_maxdB=40*)

Generate modal radial filter of specified order and frequency

Parameters

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

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

Return type array_like

`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** (*ArrayConfiguration*) – List/Tuple/ArrayConfiguration, see io.ArrayConfiguration

- **amp_maxdB** (*int, optional*) – Maximum modal amplification limit in dB [Default: 40]

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

Return type array_like

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

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

Parameters

- **fs** (*int*) – Sampling frequency
- **NFFT** (*int*) – Order of FFT (number of bins), should be a power of 2.
- **array_configuration** (ArrayConfiguration) – List/Tuple/ArrayConfiguration, see io.ArrayConfiguration
- **gridData** (*array_like*) – Quadrature grid

```
Columns : Position Number 1...M
Rows    : [AZ EL Weight]
```

- **wave_colatitude** (*wave_azimuth,*) – Direction of incoming wave in radians [0-2pi].
- **wavetype** (*{'plane', 'spherical'}, optional*) – Type of the wave. [Default: plane]
- **c** (*float, optional*) – 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

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 **fftData** – Complex sound pressures of size [(N+1)^2 x NFFT]

Return type array_like

Note: This file is a wrapper generating the complex pressures at the positions given in 'gridData' for a full spectrum 0-FS/2 Hz (NFFT Bins) wave impinging on the array, emulating discrete sampling.

sound_field_analysis.gen.**spherical_noise**(*azimuth_grid, colatitude_grid, order_max=8, spherical_harmonic_bases=None*)

Returns band-limited random weights on a spherical surface

Parameters

- **colatitude_grid** (*azimuth_grid,*) – Grids holding azimuthal and colatitudinal angles
- **order_max** (*int, optional*) – Spherical order limit [Default: 8]

Returns **noisy_weights** – Noisy weigths

Return type array_like, complex

`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** – Output fftData block including white gaussian noise

Return type array of complex floats

5.2 Processing

Functions that act on the Spatial Fourier Coefficients

FFT (Fast) Fourier Transform

iFFT Inverse (Fast) Fourier Transform

spatFT Spatial Fourier Transform

iSpatFT Fast Inverse Spatial Fourier Transform

PWDecomp Plane Wave Decomposition

Not yet implemented: *BEMA*

BEMA Spatial Anti-Aliasing

rfi Radial filter improvement

sfe Sound field extrapolation

wdr Wigner-D Rotation

`sound_field_analysis.process.`**`BEMA`**(*Pnm*, *ctSig*, *dn*, *transition*, *avgBandwidth*, *fade=True*)

BEMA Spatial Anti-Aliasing - NOT YET IMPLEMENTED

Parameters

- **`Pnm`** (*`array_like`*) – Spatial Fourier coefficients
- **`ctSig`** (*`array_like`*) – Signal of the center microphone
- **`dn`** (*`array_like`*) – Radial filters for the current array configuration
- **`transition`** (*`int`*) – Highest stable bin, approx: transition = (NFFT/FS+1) * (N*c)/(2*pi*r)
- **`avgBandwidth`** (*`int`*) – Averaging Bandwidth in oct
- **`fade`** (*`bool, optional`*) – Fade over if True, else hard cut {false} [Default: True]

Returns **Pnm** – Alias-free spatial Fourier coefficients

Return type array_like

Note: This was presented at the 2012 AES convention, see[1].

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

References

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

Real-valued Fast Fourier Transform.

Parameters

- **`time_signals`** (`array_like`) – Time-domain signals to be transformed, of shapeb [nSig x nSamples]
- **`fs`** (`int`) – Sampling frequency of the time data
- **`NFFT`** (`int, optional`) – Number of frequency bins. Resulting array will have size NFFT//2+1 Default: Next power of 2
- **`oversampling`** (`int, optional`) – Oversamples the incoming signal to increase frequency resolution [Default: 1]
- **`firstSample`** (`int, optional`) – First time domain sample to be included. [Default: 0]
- **`lastSample`** (`int, optional`) – Last time domain sample to be included. [Default: -1]

Returns

- **fftData** (*ndarray*) – Frequency-domain data
- **f** (*ndarray*) – Frequency scale

Note: 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.`**`PWDecomp`**(*N*, *OmegaL*, *Pnm*, *dn*, *cn=None*)

Plane Wave Decomposition

Parameters

- **`N`** (`int`) – Decomposition order
- **`OmegaL`** (`array_like`) – Look directions of shape

```
[AZ1, EL1;
 AZ2, EL2;
   ...
 AZn, ELn]
```

- **`Pnm`** (`matrix of complex floats`) – Spatial Fourier Coefficients (e.g. from spatFT)
- **`dn`** (`matrix of complex floats`) – Radial filters (e.g. from radFilter)
- **`cn`** (`array_like, optional`) – Weighting Function. Either frequency invariant weights as 1xN array or with kr bins in rows over N cols. [Default: None]

Returns **Y** – MxN Matrix of the decomposed wavefield with kr bins in rows

Return type matrix of floats

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

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

Parameters

- **`B`** (`A,`) – Data to perform the convolution on of shape [Nsignals x NSamples]

- **FFT** (*bool, optional*) – Selects wether time or frequency domain convolution is applied. Default: On if Nsamples > 500 for both

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

Return type array

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

Inverse real-valued Fourier Transform

Parameters

- **Y** (*array_like*) – Frequency domain data [Nsignals x Nbins]
- **output_length** (*int, optional*) – Lenght of returned time-domain signal (Default: 2 x len(Y) + 1)
- **win** (*boolean, optional*) – Weights the resulting time-domain signal with a Hann

Returns **y** – Reconstructed time-domain signal

Return type array_like

`sound_field_analysis.process.`**`iSpatFT`**(*spherical_coefficients*, *azimuths*, *colatitudes*, *order_max=None*, *spherical_harmonic_bases=None*)

Inverse spatial Fourier Transform

Parameters

- **spherical_coefficients** (*array_like*) – Spatial Fourier coefficients with columns representing frequncy bins
- **colatitudes** (*azimuths,*) – Azimuth/Colatitude angles of spherical coefficients
- **order_max** (*int, optional*) – Maximum transform order [Default: highest available order]

Returns **P** – Sound pressures with frequency bins in columnss and angles in rows

Return type array_like

`sound_field_analysis.process.`**`rfi`**(*dn*, *kernelDownScale=2*, *highPass=0.0*)

R/F/I Radial Filter Improvement [NOT YET IMPLEMENTED!]

Parameters

- **dn** (*array_like*) – Analytical frequency domain radial filters (e.g. gen.radFilter())
- **kernelDownScale** (*int, optional*) – Downscale factor for the filter kernel [Default: 2]
- **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

---

Note: This function improves the FIR radial filters from gen.radFilter(). 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).

IMPORTANT Remember to choose a fft-oversize factor (.FFT()) being large enough to cover all filter latencies and reponse slopes. Otherwise undesired cyclic convolution artifacts may appear in the output signal.

HIGHPASS If HPF is on (highPass>0) the radial filter kernel is downscaled by a factor of two. Radial Filters and HPF share the available taps and the latency keeps constant. Be careful using very small signal blocks because there may remain too few taps. Observe the filters by plotting their spectra and impulse responses. > Be very carefull if NFFT/max(kr) < 25 > Do not use R/F/I if NFFT/max(kr) < 15

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

S/F/E Sound Field Extrapolation. CURRENTLY WIP

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]

`sound_field_analysis.process.`**`spatFT`**(*data*, *azimuths*, *colatitudes*, *gridweights*, *order_max=10*, *spherical_harmonic_bases=None*)

Spatial Fourier Transform

Parameters

- **`data`** (*`array_like`*) – Data to be transformed, with signals in rows and frequency bins in columns
- **`order_max`** (*`int, optional`*) – Maximum transform order (Default: 10)
- **`colatitudes, gridweights`** (*`azimuths,`*) – Azimuths/Colatitudes/Gridweights of spatial sampling points

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

Return type array_like

`sound_field_analysis.process.`**`spatFT_LSF`**(*data*, *azimuths*, *colatitudes*, *order_max*, *spherical_harmonic_bases=None*)

Returns spherical harmonics coefficients least square fitted to provided data

Parameters

- **`data`** (*`array_like, complex`*) – Data to be fitted to
- **`colatitude_grid`** (*`azimuth_grid,`*) – Azimuth / colatidunenal data locations
- **`order_max`** (*`int`*) – Maximum order N of fit
- **`Returns`** –
- **`coefficients`** (*`array_like, float`*) – Fitted spherical harmonic coefficients (indexing: n**2 + n + m + 1)

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

W/D/R Wigner-D Rotation - NOT YET IMPLEMENTED

Parameters

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

Returns **PnmRot** – Rotated spatial Fourier coefficients

Return type array_like

5.3 Plotting

Plotting functions Helps visualizing spherical microphone data.

Generally, you probably want to first extract the amplitude information in spherical coordinates: >> plot.makeMTX(Pnm, dn, Nviz=3, krIndex=1, oversize=1) And then visualize that: >> plot3D(vizMTX, style='shape')

Other valid styles are 'sphere' and 'flat'.

`sound_field_analysis.plot.`**`frqToKr`**(*fTarget, fVec*)

Returns the kr bin closest to the target frequency

Parameters

- **`fTarget`** (*`float`*) – Target frequency
- **`fVec`** (*`array_like`*) – Array containing the available frequencys

Returns **krTarget** – kr bin closest to target frequency

Return type int

`sound_field_analysis.plot.`**`genFlat`**(*vizMTX*)

Returns trace of flat surface with intensity as surface elevation and color

Parameters **`vizMTX`** (*`array_like`*) – Matrix holding spherical data for visualization

Returns **T** – Trace of desired surface

Return type plotly_trace

Todo

Fix orientation and axis limits

`sound_field_analysis.plot.`**`genShape`**(*vizMTX*)

Returns trace of shape with intensity as radial extension

Parameters **`vizMTX`** (*`array_like`*) – Matrix holding spherical data for visualization

Returns **T** – Trace of desired shape

Return type plotly_trace

Todo

Fix camera position

`sound_field_analysis.plot.`**`genSphCoords`**()

Generates cartesian (x,y,z) and spherical (theta, phi) coordinates of a sphere :returns: **coords** – holds cartesian (x,y,z) and spherical (theta, phi) coordinates :rtype: named tuple

`sound_field_analysis.plot.`**`genSphere`**(*vizMTX*)

Returns trace of sphere with intensity as surface color

Parameters **`vizMTX`** (*`array_like`*) – Matrix holding spherical data for visualization

Returns **T** – Trace of desired sphere

Return type plotly_trace

`sound_field_analysis.plot.`**`genVisual`**(*vizMTX, style='shape', normalize=True, logScale=False*)

Returns desired trace after cleaning the data

Parameters

- **vizMTX** (*array_like*) – Matrix holding spherical 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]

Returns **T** – Trace of desired visualization

Return type plotly_trace

`sound_field_analysis.plot.`**`makeFullMTX`**(*Pnm*, *dn*, *kr*, *Nviz=3*)

Generates visualization matrix for a set of spatial fourier coefficients over all kr :param Pnm: Spatial Fourier Coefficients (e.g. from S/T/C) :type Pnm: array_like :param dn: Modal Radial Filters (e.g. from M/F) :type dn: array_like :param kr: kr-vector

:: Can also be a matrix [krm; krs] for rigid sphere configurations: [1,:] => krm referring to the microphone radius [2,:] => krs referring to the sphere radius (scatterer)

Parameters **Nviz** (*int, optional*) – Order of the spatial fourier transform [Default: 3]

Returns **vizMtx** – Computed visualization matrix over all kr

Return type array_like

`sound_field_analysis.plot.`**`makeMTX`**(*Pnm*, *dn*, *krIndex=1*, *Nviz=3*, *oversize=1*)

mtxData = makeMTX(Nviz=3, Pnm, dn, krIndex)

Parameters

- **Pnm** (*array_like*) – Spatial Fourier Coefficients (e.g. from S/T/C)
- **dn** (*array_like*) – Modal Radial Filters (e.g. from M/F)
- **krIndex** (*int*) – Index of kr to be computed [Default: 1]
- **Nviz** (*int, optional*) – Order of the spatial fourier transform [Default: 3]
- **oversize** (*int, optional*) – Integer Factor to increase the resolution. [Default: 1]

Returns **mtxData** – 3D-matrix-data in 1[deg] steps

Return type array_like

Note: The file generates a Matrix of 181x360 pixels for the visualisation with visualize3D() in 1[deg] Steps (65160 plane waves). The HD version generally allows to raise the resolution (oversize > 1). (visual3D(), map3D() admit 1[deg] data only, oversize = 1)

`sound_field_analysis.plot.`**`normalizeMTX`**(*MTX*, *logScale=False*)

Normalizes a matrix to [0 ... 1]

Parameters

- **MTX** (*array_like*) – Matrix to be normalized
- **logScale** (*bool*) – Toggle conversion logScale [Default: False]

Returns **MTX** – Normalized Matrix

Return type array_liked

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

Visualize 2D data using plotly.

Parameters

- **data** (*array_like*) – Data to be plotted, separated along the first dimension (rows).

- **title** (*string*) – Add title to be displayed on plot
- **type** (*string{None, 'time', 'linFFT', 'logFFT'}*) – Type of data to be displayed. [Default: None]
- **fs** (*int*) – Sampling rate in Hz. [Default: 44100]

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

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

Parameters

- **vizMTX** (*array_like*) – Matrix holding spherical 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]

Todo

Colorization, contour plot

sound_field_analysis.plot.**plot3Dgrid**(*rows*, *cols*, *vizMTX*, *style*, *normalize=True*)

sound_field_analysis.plot.**showTrace**(*trace*, *layout=None*, *colorize=True*)

Wrapper around plotlys offline .plot() function

Parameters

- **trace** (*plotly_trace*) – Plotly generated trace to be displayed offline
- **colorize** (*Bool, optional*) – Toggles bw / colored plot [Default: True]

Returns **fig** – JSON representation of generated figure

Return type plotly_fig_handle

sound_field_analysis.plot.**sph2cartMTX**(*vizMTX*)

Converts the spherical vizMTX data to named tuple contaibubg .xs/.ys/.zs

Parameters **vizMTX** (*array_like*) – [180 x 360] matrix that hold amplitude information over phi and theta

Returns **V** – Contains .xs, .ys, .zs cartesian coordinates

Return type named_tuple

5.4 Sphericals

Collection of spherical helper functions:

sph_harm More robust spherical harmonic coefficients

spbessel / dspbessel Spherical Bessel and derivative

spneumann / dspneumann Spherical Neumann (Bessel 2nd kind) and derivative

sphankel / dsphankel Spherical Hankel (second kind) and derivative

cart2sph / sph2cart Convert cartesion to spherical coordinates and vice versa

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** (ArrayConfiguration) – List/Tuple/ArrayConfiguration, see io.ArrayConfiguration
- **normalize** (*Bool, optional*) – Normalize by 4 * pi * 1j ** order (Default: True)

Returns **b** – Coefficients of shape [nOrder x nFreqs]

Return type array, complex

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
- **kr** (*array_like*) – Argument

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

Return type array_like

sound_field_analysis.sph.**bn_dual_open_pressure**(*n*, *kr1*, *kr2*)

sound_field_analysis.sph.**bn_open_pressure**(*n*, *krm*)

sound_field_analysis.sph.**bn_open_velocity**(*n*, *krm*)

sound_field_analysis.sph.**bn_rigid_pressure**(*n*, *krm*, *krs*)

sound_field_analysis.sph.**bn_rigid_velocity**(*n*, *krm*, *krs*)

sound_field_analysis.sph.**cart2sph**(*x*, *y*, *z*)

Converts cartesian coordinates x, y, z to spherical coordinates az, el, r.

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'** – Derivative of spherical Bessel

Return type complex float

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** – Derivative of spherical Hankel function hn' (second kind)

Return type complex float

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** – Derivative of spherical Hankel function hn' (second kind)

Return type complex float

`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'** – Derivative of spherical Neumann (Bessel second kind)

Return type complex float

`sound_field_analysis.sph.`**`hankel1`**(*n*, *z*)
Bessel function of third kind (Hankel function) of order n at kr. Wraps scipy.special.hankel1(n, z)

Parameters

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

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

Return type array_like

`sound_field_analysis.sph.`**`hankel2`**(*n*, *z*)
Bessel function of third kind (Hankel function) of order n at kr. Wraps scipy.special.hankel2(n, z)

Parameters

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

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

Return type array_like

`sound_field_analysis.sph.`**`kr`**(*f*, *radius*, *temperature=20*)
Return 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 Celcius [Default: 20]

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

Return type array_like

`sound_field_analysis.sph.`**`kr_full_spec`**(*fs*, *radius*, *NFFT*, *temperature=20*)
Returns 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 Celcius (Default: 20 C)

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

Return type array_like

`sound_field_analysis.sph.`**`mnArrays`**(*nMax*)

Returns 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.`**`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
- **kr** (*array_like*) – Argument

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

Return type array_like

`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** – Spherical Bessel

Return type complex float

`sound_field_analysis.sph.`**`sph2cart`**(*az*, *el*, *r*)

Converts spherical coordinates az, el, r to cartesian coordinates x, y, z.

`sound_field_analysis.sph.`**`sph_harm`**(*m*, *n*, *az*, *el*, *type='complex'*)

Compute sphercial 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.
- **el** (*(float)*) – Elevation (colatitudinal) coordinate [0, pi], also called Phi.

Returns **y_mn** – Complex spherical harmonic of order m and degree n, sampled at theta = az, phi = el

Return type (complex float)

`sound_field_analysis.sph.`**`sph_harm_all`**(*nMax*, *az*, *el*, *type='complex'*)

Compute all sphercial 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.
- **el** (*(float), array_like*) – Elevation (colatitudinal) coordinate [0, pi], also called Phi.

Returns **y_mn** – Complex spherical harmonics of degrees n [0 ... nMax] and all corresponding orders m [-n ... n], sampled at [az, el]. dim1 corresponds to az/el pairs, dim2 to oder/degree (m, n) pairs like 0/0, -1/1, 0/1, 1/1, -2/2, -1/2 ...

Return type (complex float), array_like

`sound_field_analysis.sph.`**`sph_harm_large`**(*m*, *n*, *az*, *el*)

Compute sphercial 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.
- **el** (*(float)*) – Elevation (colatitudinal) coordinate [0, pi], also called Phi.

Returns

- **y_mn** (*(complex float)*) – Complex spherical harmonic of order m and degree n, sampled at theta = az, phi = el
- *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) is the associated Legendre function of the first kind, like scipy.special.lpmv*
- *scipy.special.lpmn calculates P(0...m 0...n) and its derivative but won't return +inf at high orders*

`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** – Spherical Hankel function hn (first kind)

Return type complex float

`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** – Spherical Hankel function hn (second kind)

Return type complex float

`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** (*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)
- **transducer_type** (*string {pressure, velocity}*) – Transducer type [Default: pressure]

Returns b

Return type array, complex

`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** – Spherical Neumann (Bessel second kind)

Return type complex float

5.5 I/O

Input-Output functions

class `sound_field_analysis.io.`**`ArrayConfiguration`**
Tuple of type ArrayConfiguration

Parameters

- **array_radius** (*float*) – Radius of array
- **array_type** (*{'open', 'rigid'}*) – Type array
- **transducer_type** (*{'pressure', 'velocity'}*) – Type of transducer,
- **scatter_radius** (*float, optional*) – Radius of scatterer, required for *array_type* == 'rigid'
- **dual_radius** (*float, optional*) – Radius of second array, required for *array_type* == 'dual'

class `sound_field_analysis.io.`**`ArraySignal`**
Tuple of type ArraySignal

Parameters

- **signals** (TimeSignal) – Holds time domain signals and sampling frequency fs
- **grid** (SphericalGrid) – Location grid of all time domain signals
- **configuration** (ArrayConfiguration) – Information on array configuration
- **temperature** (*array_like, optional*) – Temperature in room or at each sampling position

class `sound_field_analysis.io.`**`SphericalGrid`**
Tuple of type SphericalGrid

Parameters

- **Colatitude, Radius** (*Azimuth,*) –
- **Weights** (*float, optional*) –

class `sound_field_analysis.io.`**`TimeSignal`**
Tuple of type SphericalGrid

Parameters

- **signal** (*array_like*) – Array of signals of shape [nSignals x nSamples]
- **fs** (*int*) – Sampling frequency
- **delay** (*float*) –

`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_time_signal`**(*filename*)

Convenience function to load saved np data structures

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

Returns

- **time_data** (*recarray*) – Structured array with following fields:
- *::* – .IR [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.`**`read_miro_struct`**(*file_name*, *channel='irChOne'*)

Reads miro matlab files.

Parameters

- **matFile** (*filepath*) – Path to file that has been exported as a struct like so

```
load SOFiA_A1;
SOFiA_A1_struct = struct(SOFiA_A1);
save('SOFiA_A1_struct.mat', , '-struct', 'SOFiA_A1_struct');
```

- **channel** (*string, optional*) – Channel that holds required signals. Default: 'irChOne'

Returns

- **td** (*recarray*)
- *time_data* array with fields from empty_time_signal()
- *::* – .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.`**`read_wavefile`**(*filename*)

Reads in wavefiles and returns data [Nsig x Nsamples] and fs :param 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*)
Takes two time signals and writes out the horizontal plane as HRIRs for the SoundScapeRenderer

Parameters

- **`filename`** (*`string`*) – filename to write to
- **`time_data_l`** (*`time_data_l,`*) – time_data arrays for left/right channel.

5.6 lebedev

Generate Lebedev grid and coefficients This module only exposes the function *lebGrid = lebedev.genGrid(degree)*.

lebGrid is a named tuple containing the coordinates .x, .y, .z and the weights .w Possible degrees: 6, 14, 26, 38, 50, 74, 86, 110, 146, 170, 194

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 [1] in their publications, and in the user- and programmers-manuals describing their codes.

[1] 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.

`sound_field_analysis.lebedev.`**`genGrid`**(*n*)
Returns Lebedev coefficients of n'th degree

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

Returns **lebGrid** – lebGrid is a named tuple containing .x, .y, .z and .w

Return type named tuple

5.7 Utilities

Miscellenious utility functions

`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** – 20 * log10(abs(data))

Return type array_like

`sound_field_analysis.utils.`**`deg2rad`**(*deg*)
Converts from degree [0 ... 360] to radiant [0 ... 2 pi]

`sound_field_analysis.utils.`**`env_info`**()
Guess environment based on sys.modules.

Returns **env** – Guesed environment

Return type string{'jupyter_notebook', 'ipython_terminal', 'terminal'}

`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.`**`logical_IDX_of_nearest`**(*array*, *value*)
Returns logical indices of nearest values inside array

`sound_field_analysis.utils.`**`nearest_to_value`**(*array*, *value*)
Returns nearest value inside an array

`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.`**`rad2deg`**(*rad*)
Converts from radiant [0 ... 2 pi] to degree [0 ... 360]

`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.`**`simple_resample`**(*data*, *original_fs*, *target_fs*)
Wrap scipy.signal.resample with a simpler API

PYTHON MODULE INDEX

S

INDEX

R

S

T

W