sofia-py Documentation

Release 0.2

Christoph Hohnerlein (QU Lab)

Jul 25, 2016

Contents:

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 :doc:'reference.

GENERATORS

Generators explanation. Module contains various generator functions:

awgn Generate additive White Gaussian noise

gaussGrid Gauss-Legendre quadrature grid and weights

lebedev Lebedev quadrature grid and weigths

mf Modal Radial Filter

swg Sampled Wave Generator, emulating discrete sampling

wgc Wave Generator, returns spatial Fourier coefficients

PROCESSING

Processing explanation. Processing functions:

fdt Frequency to time transform

itc Fast Inverse Spatial Fourier Transform

pdc Plane Wave Decomposition

rfi Radial filter Improvement

stc Fast Spatial Fourier Transform

tdt Time Domain Reconstruction

FOUR

PLOTTING

Plotting explanation. Plotting functions - makeMTX: Generate 3D-matrix data - visualize3D: Draw matrix data in 3D

REFERENCE

5.1 Generators

Module contains various generator functions:

awgn Generate additive White Gaussian noise

gaussGrid Gauss-Legendre quadrature grid and weights

lebedev Lebedev quadrature grid and weigths

mf Modal Radial Filter

swg Sampled Wave Generator, emulating discrete sampling

wgc Wave Generator, returns spatial Fourier coefficients

sofia.gen.**awgn** (*fftData*, *noiseLevel=80*, *printInfo=True*)

Adds White Gaussian Noise of approx. 16dB crest to a FFT block. :param fftData: Input fftData block (e.g. from F/D/T or S/W/G) :type fftData: array of complex floats :param noiseLevel: Average noise Level in dB [Default: -80dB] :type noiseLevel: int, optional :param printInfo: Toggle print statements [Default: True] :type printInfo: bool, optional

> **Returns noisyData** – Output fftData block including white gaussian noise

> **Return type** array of complex floats

sofia.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

sofia.gen.**lebedev** (*degree*, *plot=False*, *printInfo=True*)

Compute Lebedev quadrature nodes and weigths.

Parameters

- **Degree** (`int`) – Lebedev Degree. Currently available: 6, 14, 26, 38, 50, 74, 86, 110, 146, 170, 194

- **plot** (`bool, optional`) – Plot selected Lebedev grid [Default: False]

Returns

- **gridData** (*array of floats*) – Lebedev quadrature positions and weigths: [AZ, EL, W]

- **Nmax** (*int*) – Highest stable grid order

sofia.gen.**mf** (*N*, *kr*, *ac*, *amp_maxdB=0*, *plc=0*, *fadeover=0*, *printInfo=True*)
    Generate modal radial filters

Parameters

- **N** (`int`) – Maximum Order

- **kr** (`array of floats`) – Vector or Matrix of kr values

    ```
    First Row  (M=1) N: kr values microphone radius
    Second Row (M=2) N: kr values sphere/microphone radius
    [kr_mic;kr_sphere] for rigid/dual sphere configurations
    ! If only one kr-vector is given using a rigid/dual sphere
    Configuration: kr_sphere = kr_mic
    ```

- **ac** (`int {0, 1, 2, 3, 4}`) –

    Array Configuration:

    - *0*: Open Sphere with p Transducers (NO plc!)

    - *1*: Open Sphere with pGrad Transducers

    - *2*: Rigid Sphere with p Transducers

    - *3*: Rigid Sphere with pGrad Transducers

    - *4*: Dual Open Sphere with p Transducers

- **amp_maxdB** (`int, optional`) – Maximum modal amplification limit in dB [Default: 0]

- **plc** (`int {0, 1, 2}, optional`) – OnAxis powerloss-compensation: - *0*: Off [Default] - *1*: Full kr-spectrum plc - *2*: Low kr only -> set fadeover

- **fadeover** (`int, optional`) – :: Number of kr values to fade over +/- around mindistance gap of powerloss compensated filter and normal N0 filters. 0 is auto fadeover [Default]

Returns

- **dn** (*array of floats*) – Vector of modal 0-N frequency domain filters

- **beam** (*array of floats*) – Expected free field on-axis kr-response

sofia.gen.**swg** (*r=0.01*, *gridData=None*, *ac=0*, *FS=48000*, *NFFT=512*, *AZ=0*, *EL=1.5707963267948966*, *c=343*, *wavetype=0*, *ds=1*, *Nlim=120*, *printInfo=True*)
    Sampled Wave Generator Wrapper

Parameters

- **r** (`array of floats, optional`) – Microphone Radius [Default: 0.01]

```
Can also be a vector for rigid sphere configurations:
[1,1] => rm  Microphone Radius
[2,1] => rs  Sphere Radius (Scatterer)
```

- **gridData** (*array of floats*) – Quadrature grid [Default: 110 Lebebdev grid]

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

- **ac** (*int {0, 1, 2, 3, 4}*) –

  Array Configuration:

    – *0*: Open Sphere with p Transducers (NO plc!) [Default]

    – *1*: Open Sphere with pGrad Transducers

    – *2*: Rigid Sphere with p Transducers

    – *3*: Rigid Sphere with pGrad Transducers

    – *4*: Dual Open Sphere with p Transducers

- **FS** (*int, optional*) – Sampling frequency [Default: 48000 Hz]

- **NFFT** (*int, optional*) – Order of FFT (number of bins), should be a power of 2. [Default: 512]

- **AZ** (*float, optional*) – Azimuth angle in radians [0-2pi]. [Default: 0]

- **EL** (*float, optional*) – Elevation angle in in radians [0-pi]. [Default: pi / 2]

- **c** (*float, optional*) – Speed of sound in [m/s] [Default: 343 m/s]

- **wavetype** (*int {0, 1}, optional*) –

  Type of the Wave:

    – 0: Plane Wave [Default]

    – 1: Spherical Wave

- **ds** (*float, optional*) – Distance of the source in [m] (For wavetype = 1 only)

```
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).
```

- **Nlim** (*int, optional*) – Internal generator transform order limit [Default: 120]

Returns

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

- **kr** (*array of floats*) – 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)
```

Notes

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 to an array. The wrapper involves the W/G/C wave generator core and the I/T/C spatial transform core.

S/W/G emulates discrete sampling. You can observe alias artifacts.

sofia.gen.**wgc**(*N*, *r*, *ac*, *fs*, *F_NFFT*, *az*, *el*, *t=0.0*, *c=343.0*, *wavetype=0*, *ds=1.0*, *lowerSegLim=0*, *SegN=None*, *upperSegLim=None*, *printInfo=True*)
Wave Generator Core: Returns Spatial Fourier Coefficients *Pnm* and *kr* vector

> **Parameters**
>
> - **N** (*int*) – Maximum transform order.
>
> - **r** (*list of ints*) – Microphone radius
>
>   ```
>   Can also be a vector for rigid/dual sphere configurations:
>   [1,1] => rm  Microphone radius
>   [2,1] => rs  Sphere or microphone radius
>   ! If only one radius (rm) is given using a rigid/dual sphere
>   Configuration: rs = rm and only one kr-vector is returned!
>   ```
>
> - **ac** (*int {0, 1, 2, 3, 4}*) –
>
>   **Array Configuration:**
>
>   - *0*: Open Sphere with p Transducers (NO plc!)
>
>   - *1*: Open Sphere with pGrad Transducers
>
>   - *2*: Rigid Sphere with p Transducers
>
>   - *3*: Rigid Sphere with pGrad Transducers
>
>   - *4*: Dual Open Sphere with p Transducers
>
> - **FS** (*int*) – Sampling frequency
>
> - **NFFT** (*int*) – Order of FFT (number of bins), should be a power of 2.
>
> - **AZ** (*float*) – Azimuth angle in radians [0-2pi].
>
> - **EL** (*float*) – Elevation angle in in radians [0-pi].
>
> - **t** (*float, optional*) – Time Delay in s.
>
> - **c** (*float, optional*) – Speed of sound in [m/s] [Default: 343m/s]
>
> - **wavetype** (*int {0, 1}, optional*) –
>
>   **Type of the Wave:**
>
>   - 0: Plane Wave [Default]
>
>   - 1: Spherical Wave
>
> - **ds** (*float, optional*) – Distance of the source in [m] (For wavetype = 1 only)
>
>   ```
>   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).
>   ```
>
> - **lSegLim** (*int, optional*) – (Lower Segment Limit) Used by the S/W/G wrapper

- **uSegLim**(`int, optional`) – (Upper Segment Limit) Used by the S/W/G wrapper
- **SegN**(`int, optional`) – (Sement Order) Used by the S/W/G wrapper
- **printInfo**(`bool, optional`) – Toggle print statements

> **Returns**

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

5.2 Processing

Processing functions:

fdt Frequency to time transform

itc Fast Inverse Spatial Fourier Transform

pdc Plane Wave Decomposition

rfi Radial filter Improvement

stc Fast Spatial Fourier Transform

tdt Time Domain Reconstruction

`sofia.process.`**fdt**(*timeData*, *FFToversize=1*, *firstSample=0*, *lastSample=None*)
> F/D/T frequency domain transform

> **Parameters**

- **timeData** (`named tuple`) – timeData tuple with following fields

```
.impulseResponses [Channels X Samples]
.FS
.radius          Array radius
.averageAirTemp  Temperature in [C]
(.centerIR       [1 x Samples] )
```

- **FFToversize** (`int, optional`) – FFToversize rises the FFT Blocksize. [Default: 1]

```
A FFT of the blocksize (FFToversize*NFFT) is applied
to the time domain data,  where  NFFT is determinded
as the next power of two of the signalSize  which is
signalSize = (lastSample-firstSample).
The function will pick a window of (lastSample-firstSample)
for the FFT.
```

- **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** (*array of floats*) – Frequency domain data ready for the Spatial Fourier Transform (stc)

- **kr** (*array of floats*) – kr-Values of the delivered data

- **f** (*array of floats*) – Absolute frequency scale

- **ctSig** (*array of floats*) – Center signal, if available

Notes

Call this function with a running window (firstSample+td->lastSample+td) iteration increasing td to obtain time slices. This way you resolve the temporal information within the captured sound field.

`sofia.process.`**`itc`**(*Pnm*, *angles*, *N=None*, *printInfo=True*)

I/T/C Fast Inverse spatial Fourier Transform Core

Parameters

- **`Pnm`** (`array of floats`) – Spatial Fourier coefficients with FFT bins as cols and nm coeffs as rows (e.g. from SOFiA S/T/C)

- **`angles`** (`array of floats`) – Target angles of shape

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

- **`[N]`** (`int, optional`) – Maximum transform order [Default: highest available order]

Returns **p** – Sound pressures with FFT bins in cols and specified angles in rows

Return type array of complex floats

Notes

This is a pure ISFT core that does not involve extrapolation. (=The pressures are referred to the original radius)

`sofia.process.`**`pdc`**(*N*, *OmegaL*, *Pnm*, *dn*, *cn=None*, *printInfo=True*)

P/D/C - Plane Wave Decomposition

Parameters

- **`N`** (`int`) – Decomposition order

- **`OmegaL`** (`array of floats`) – Look directions of shape

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

- **`Pnm`** (`matrix of complex floats`) – Spatial Fourier Coefficients (e.g. from SOFiA S/T/C)

- **`dn`** (`matrix of complex floats`) – Modal array filters (e.g. from SOFiA M/F)

- **`cn`** (`array of floats, 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

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

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

> **Parameters**
>
> - **dn** (`array of floats`) – Analytical frequency domain radial filters (e.g. SOFiA M/F)
>
> - **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 of floats*) – 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 SOFiA M/F. 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 (F/D/T) 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

`sofia.process.`**`stc`**(*N*, *fftData*, *grid*)

 S/T/C Fast Spatial Fourier Transform

> **Parameters**
>
> - **N** (`int`) – Maximum transform order
>
> - **fftData** (`array of floats`) – Frequency domain sounfield data, (e.g. from SOFiA FDT) with spatial sampling positions in cols and FFT bins in rows
>
> - **grid** (`array of floats`) – Grid configuration of AZ [0 ... 2pi], EL [0...pi] and W of shape
>
>   ```
>   [AZ1, EL1, W1;
>    AZ2, EL2, W2;
>       ...
>    AZn, ELn, Wn]
>   ```
>
> **Returns** Pnm – Spatial Fourier Coefficients with nm coeffs in cols and FFT bins in rows
>
> **Return type** array of floats

sofia.process.**tdt**(*Y*, *win=0*, *minPhase=False*, *resampleFactor=1*, *printInfo=True*)
    T/D/T - Time Domain Transform

> **Parameters**
>
> * **Y** (*array of floats*) – Frequency domain data over multiple channels (cols) with FFT data in rows
>
> * **float, optional** (*win*) – Window Signal tail [0...1] with a HANN window [Default: 0] - NOT YET IMPLEMENTED
>
> * **int, optional** (*resampleFactor*) – Resampling factor (FS_target/FS_source)
>
> * **bool, optional** (*minPhase*) –
>
> * **minimum phase reduction – NOT YET IMPLEMENTED [Default** (*Ensure*) –
>
> **Returns** **y** – Reconstructed time-domain signal of channels in cols and impulse responses in rows
>
> **Return type** array of floats

Notes

This function recombines time domain signals for multiple channels from frequency domain data. It is made to work with half-sided spectrum FFT data. The impulse responses can be windowed. The IFFT blocklength is determined by the Y data itself:

Y should have a size [NumberOfChannels x ((2^n)/2)+1] with n=[1,2,3,...] and the function returns [NumberOfChannels x resampleFactor*2^n] samples.

5.3 Plotting

Plotting functions - makeMTX: Generate 3D-matrix data - visualize3D: Draw matrix data in 3D

sofia.plot.**genScatter**(*vizMTX*, *colorize=False*)

sofia.plot.**genShape**(*vizMTX*, *offset=0*, *scale=1.0*, *colorize=False*)

sofia.plot.**genSphere**(*vizMTX*, *colorize=False*)

sofia.plot.**genVisual**(*vizMTX*, *style='shape'*, *colorize=False*, *offset=0*, *scale=1.0*, *normalize=True*)

sofia.plot.**generateAngles**()
    Returns a [65160 x 1] grid of all radiant angles in 1 deg steps

sofia.plot.**makeMTX**(*Pnm*, *dn*, *Nviz=3*, *krIndex=1*, *oversize=1*)
    mtxData = makeMTX(Nviz=3, Pnm, dn, krIndex)

> **Parameters**
>
> * **Spatial Fourier Coefficients (from S/T/C)** (*Pnm*) –
>
> * **Modal Radial Filters (from M/F)** (*dn*) –
>
> * **Order of the spatial fourier transform [default = 3]** (*N*) –
>
> * **Index of kr Vector [default = 1]** (*krIndex*) –
>
> * **Integer Factor to increase the resolution. Set oversize = 1** (*oversize*) – (default) to use the mtxData matrix for visual3D(), map3D().

- **#Returns** –

- **-------** –

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

Notes

The file generates a SOFiA mtxData 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)

sofia.plot.**normalizeMTX** (*vizMTX*)

sofia.plot.**plotGrid** (*rows*, *cols*, *vizMTX*, *bgcolor='white'*, *style='shape'*, *colorize=False*, *normalize=True*)

sofia.plot.**visualize3D** (*vizMTX*, *style='sphere'*, *colorize=True*, *offset=0.0*, *scale=1.0*, ***kargs*)
   Visualize matrix data, such as from makeMTX(Pnm, dn)

   Parameters

   - **SOFiA 3D-matrix-data [1[deg] steps]** (*vizMTX*) –

   - **'sphere', surface colors indicate the intensity (default)**
     (*style*) – 'flat', surface colors indicate the intensity (TODO) 'scatter', extension indicates the intensity 'shape', extension indicates the intensity

   - **linear offset for shape (Default** (*offset*) –

   - **scaling factor for shape (Default** (*scale*) –

   ### Notes

   TODO: Implement flat style, fix color position in sphere and shape, make colormap selectable, move grid generation into function

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 and derivative

cart2sph / sph2cart Convert cartesion to spherical coordinates and vice versa

sofia.sph.**bn** (*n*, *krm*, *krs*, *ac*)

sofia.sph.**bn_dualOpenP** (*n*, *kr1*, *kr2*)

sofia.sph.**bn_npf** (*n*, *krm*, *krs*, *ac*)
   Microphone scaling

   Parameters

   - **n** (*int*) – Order

- **krm** (*array of floats*) – Microphone radius

- **krs** (*array of floats*) – Sphere radius

- **ac** (*int {0, 1, 2, 3, 4}*) –

  Array Configuration:

  – *0*: Open Sphere with p Transducers (NO plc!) [Default]

  – *1*: Open Sphere with pGrad Transducers

  – *2*: Rigid Sphere with p Transducers

  – *3*: Rigid Sphere with pGrad Transducers

  – *4*: Dual Open Sphere with p Transducers

Returns b

Return type array of floats

sofia.sph.**bn_openP** (*n*, *krm*)

sofia.sph.**bn_openPG** (*n*, *krm*)

sofia.sph.**bn_rigidP** (*n*, *krm*, *krs*)

sofia.sph.**bn_rigidPG** (*n*, *krm*, *krs*)

sofia.sph.**cart2sph** (*x*, *y*, *z*)
    Converts cartesian coordinates x, y, z to spherical coordinates az, el, r.

sofia.sph.**dspbessel** (*n*, *kr*)
    Derivative of spherical Bessel

Parameters

- **n** (*int*) – Order

- **kr** (*int*) – Degree

Returns J' – Derivative of spherical Bessel

Return type complex float

sofia.sph.**dsphankel** (*n*, *kr*)
    Derivative spherical Hankel function hn'

Parameters

- **n** (*int*) – Order

- **kr** (*int*) – Degree

Returns hn' – Derivative of spherical Hankel function hn'

Return type complex float

sofia.sph.**dspneumann** (*n*, *kr*)
    Derivative spherical Neumann (Bessel second kind) of order n

Parameters

- **n** (*int*) – Order

- **kr** (*int*) – Degree

Returns Yv' – Derivative of spherical Neumann (Bessel second kind)

> **Return type** complex float

`sofia.sph.` **`printBNs`** (*n*, *krm*, *krs*)

`sofia.sph.` **`printSPH`** (*n*, *k*)

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

> Spherical Bessel function
>
> > **Parameters**
> >
> > - **n** (*int*) – Order
> >
> > - **kr** (*int*) – Degree
> >
> > **Returns** J – Spherical Bessel
> >
> > **Return type** complex float

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

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

`sofia.sph.` **`sph_harm`** (*m*, *n*, *az*, *el*)

> 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 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*

`sofia.sph.` **`sphankel`** (*n*, *kr*)

> Spherical Hankel hn
>
> > **Parameters**
> >
> > - **n** (*int*) – Order
> >
> > - **kr** (*int*) – Degree
> >
> > **Returns** hn – Spherical Hankel function hn
> >
> > **Return type** complex float

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

> Spherical Neumann (Bessel second kind)
>
> > **Parameters**
> >
> > - **n** (*int*) – Order

- **kr** (*int*) – Degree

Returns Yv – Spherical Neumann (Bessel second kind)

Return type complex float

5.5 I/O

Input-Output functions

`sofia.io.`**`readMiroStruct`**(*matFile*)

Reads miro matlab files.

Parameters **matFile** (*filepath*) – .miro file that has been exported as a struct like so

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

Returns

- **timeData** (*named tuple*)
- *timeData* tuple with following fields
- *:* – .impulseResponses [Channels X Samples] .FS .radius Array radius .averageAirTemp Temperature in [C] (.centerIR [1 x Samples])

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.

`sofia.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

SIX

INDICES AND TABLES

- genindex
- search

S