The Sound Field Synthesis Toolbox for Python gives you the possibility to create numerical simulations of sound
field synthesis methods like wave field synthesis (WFS) or near-field compensated higher order Ambisonics (NFC-HOA).

Theory: http://sfstoolbox.org/
Documentation: http://python.sfstoolbox.org/
Source code and issue tracker: https://github.com/sfstoolbox/sfs-python/
Python Package Index: https://pypi.python.org/pypi/sfs/
SFS Toolbox for Matlab: http://matlab.sfstoolbox.org/
License: MIT – see the file LICENSE for details.

Quick start:

- Install NumPy, SciPy and Matplotlib
- python3 -m pip install sfs --user
- python3 doc/examples/horizontal_plane_arrays.py

1 Installation

1.1 Requirements

Obviously, you'll need Python\(^1\). We normally use Python 3.x, but it should also work with Python 2.x. NumPy\(^2\) and SciPy\(^3\) are needed for the calculations. If you also want to plot the resulting sound fields, you'll need matplotlib\(^4\).

Instead of installing all of them separately, you should probably get a Python distribution that already includes everything, e.g. Anaconda\(^5\).

1.2 Installation

Once you have installed the above-mentioned dependencies, you can use pip\(^6\) to download and install the latest release of the Sound Field Synthesis Toolbox with a single command:

```bash
python3 -m pip install sfs --user
```

If you want to install it system-wide for all users (assuming you have the necessary rights), you can just drop the --user option.

To un-install, use:

```bash
python3 -m pip uninstall sfs
```

2 Examples

Various examples are located in the directory doc/examples/ as Python scripts, e.g.

- ```
  sound_field_synthesis.py: Illustrates the general usage of the toolbox
```
• **horizontal_plane_arrays.py**: Computes the sound fields for various techniques, virtual sources and loudspeaker array configurations

• **soundfigures.py**: Illustrates the synthesis of sound figures with Wave Field Synthesis

Or Jupyter notebooks, which are also available online as interactive examples: binder:doc/examples

### 2.1 Modal Room Acoustics

In [1]:
   : import numpy as np
   : import matplotlib.pyplot as plt
   : import sfs

/home/docs/checkouts/readthedocs.org/user_builds/sfs-python/conda/0.4.0/lib/python3.5/importlib/_bootstrap.py:222: RuntimeWarning: numpy.dtype size changed, may indicate binary incompatibility. Expected 96, got 88
   : return f(*args, **kwds)

In [2]: %matplotlib inline

In [3]: x0 = 1, 3, 1.80 # source position
   : L = 6, 6, 3 # dimensions of room
   : deltan = 0.01 # absorption factor of walls
   : n0 = 1, 0, 0 # normal vector of source (only for compatibility)
   : N = 20 # maximum order of modes

You can experiment with different combinations of modes:

In [4]: #N = [[1], 0, 0]

**Sound Field for One Frequency**

In [5]: f = 500 # frequency
   : omega = 2 * np.pi * f # angular frequency

In [6]: grid = sfs.util.xyz_grid([0, L[0]], [0, L[1]], L[2] / 2, spacing=.1)

In [7]: p = sfs.mono.source.point_modal(omega, x0, n0, grid, L, N=N, deltan=deltan)

For now, we apply an arbitrary scaling factor to make the plot look good

**TODO**: proper normalization

In [8]: p *= 0.05

In [9]: sfs.plot.soundfield(p, grid);

---

7 https://mybinder.org/v2/gh/sfstoolbox/sfs-python/0.4.0?filepath=doc/examples
Frequency Response at One Point

In [10]: f = np.linspace(20, 200, 180)  # frequency
omega = 2 * np.pi * f  # angular frequency
receiver = 1, 1, 1.8

p = [sfs.mono.source.point_modal(om, x0, n0, receiver, L, N=N, deltan=deltan)
    for om in omega]

plt.plot(f, sfs.util.db(p))
plt.xlabel('frequency / Hz')
plt.ylabel('level / dB')
plt.grid()
3 Secondary Sources

3.1 Loudspeaker Arrays

Compute positions of various secondary source distributions.

```python
import sfs
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = 8, 4.5  # inch
plt.rcParams['axes.grid'] = True

class sfs.array.ArrayData:
    Create new instance of ArrayData(x, n, a)

    take(indices)
    Return a sub-array given by indices.

sfs.array.linear(N, spacing, center=[0, 0, 0], orientation=[1, 0, 0])
Linear secondary source distribution.

Parameters

- **N (int)** – Number of secondary sources.
- **spacing (float)** – Distance (in metres) between secondary sources.
- **center ((3,) array_like, optional)** – Coordinates of array center.
- **orientation ((3,) array_like, optional)** – Orientation of the array. By default, the loudspeakers have their main axis pointing into positive x-direction.

Returns **ArrayData** – Positions, orientations and weights of secondary sources.
Examples

```python
x0, n0, a0 = sfs.array.linear(16, 0.2, orientation=[0, -1, 0])
sfs.plot.loudspeaker_2d(x0, n0, a0)
plt.axis('equal')
```

**sfs.array.linear_diff** *(distances, center=[0, 0, 0], orientation=[1, 0, 0])*  
Linear secondary source distribution from a list of distances.

**Parameters**
- **distances** *(N-1,) array_like* – Sequence of secondary sources distances in metres.
- **center, orientation** – See **linear()**.

**Returns** *ArrayData* – Positions, orientations and weights of secondary sources.

Examples

```python
x0, n0, a0 = sfs.array.linear_diff(4 * [0.3] + 6 * [0.15] + 4 * [0.3],
                                 orientation=[0, -1, 0])
sfs.plot.loudspeaker_2d(x0, n0, a0)
plt.axis('equal')
```

**sfs.array.linear_random** *(N, min_spacing, max_spacing, center=[0, 0, 0], orientation=[1, 0, 0],
seed=None)*  
Randomly sampled linear array.

**Parameters**
- **N** *(int)* – Number of secondary sources.
- **min_spacing, max_spacing** *(float)* – Minimal and maximal distance (in metres) between secondary sources.
- **center, orientation** – See **linear()**.
• seed ([None, int, array_like], optional) – Random seed. See numpy.random.RandomState.

Returns ArrayData – Positions, orientations and weights of secondary sources.

Examples

```python
x0, n0, a0 = sfs.array.linear_random(12, 0.15, 0.4, orientation=[0, -1, 0])
sfs.plot.loudspeaker_2d(x0, n0, a0)
plt.axis('equal')
```

---

sfs.array.circular\( (N, R, \text{center}=[0, 0, 0]) \)
Circular secondary source distribution parallel to the xy-plane.

**Parameters**
- \( N \) (int) – Number of secondary sources.
- \( R \) (float) – Radius in metres.
- \( \text{center} \) – See \textit{linear}().

**Returns** \textit{ArrayData} – Positions, orientations and weights of secondary sources.

**Examples**

```python
x0, n0, a0 = sfs.array.circular(16, 1)
sfs.plot.loudspeaker_2d(x0, n0, a0, size=0.2, show_numbers=True)
plt.axis('equal')
```

sfs.array.rectangular\( (N, \text{spacing, center}=[0, 0, 0], \text{orientation}=[1, 0, 0]) \)
Rectangular secondary source distribution.

**Parameters**
- \( N \) (int or pair of int) – Number of secondary sources on each side of the rectangle. If a pair of numbers is given, the first one specifies the first and third segment, the second number specifies the second and fourth segment.
- \( \text{spacing} \) (float) – Distance (in metres) between secondary sources.
- \( \text{center, orientation} \) – See \textit{linear}(). The \textit{orientation} corresponds to the first linear segment.

**Returns** \textit{ArrayData} – Positions, orientations and weights of secondary sources.

**Examples**
x0, n0, a0 = sfs.array.rectangular((4, 8), 0.2)
sfs.plot.loudspeaker_2d(x0, n0, a0, show_numbers=True)
plt.axis('equal')

\[
\begin{array}{cccccccc}
5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
4 & & & & & & & \\
3 & & & & & & & \\
2 & & & & & & & \\
1 & & & & & & & \\
& -0.25 & 0.00 & 0.25 & 0.50 & 0.75 & 1.00 & 1.25 \\
& -0.25 & 0.00 & 0.25 & 0.50 & 0.75 & 1.00 & 1.25 \\
& -0.25 & 0.00 & 0.25 & 0.50 & 0.75 & 1.00 & 1.25 \\
& -0.25 & 0.00 & 0.25 & 0.50 & 0.75 & 1.00 & 1.25 \\
& -0.25 & 0.00 & 0.25 & 0.50 & 0.75 & 1.00 & 1.25 \\
& -0.25 & 0.00 & 0.25 & 0.50 & 0.75 & 1.00 & 1.25 \\
& -0.25 & 0.00 & 0.25 & 0.50 & 0.75 & 1.00 & 1.25 \\
& -0.25 & 0.00 & 0.25 & 0.50 & 0.75 & 1.00 & 1.25 \\
\end{array}
\]

\textbf{sfs.array.rounded_edge} (Nxy, Nr, dx, \texttt{center=[0, 0, 0]}, \texttt{orientation=[1, 0, 0]})
Array along the \texttt{xy-axis} with rounded edge at the origin.

**Parameters**
- \texttt{Nxy (int)} – Number of secondary sources along \texttt{x-} and \texttt{y-axis}.
- \texttt{Nr (int)} – Number of secondary sources in rounded edge. Radius of edge is adjusted to
equidistant sampling along entire array.
- \texttt{center ((3,) array_like, optional)} – Position of edge.
- \texttt{orientation ((3,) array_like, optional)} – Normal vector of array. Default orientation is
along \texttt{xy-axis}.

**Returns** \texttt{ArrayData} – Positions, orientations and weights of secondary sources.

\textbf{Examples}

\begin{verbatim}
x0, n0, a0 = sfs.array.rounded_edge(8, 5, 0.2)
sfs.plot.loudspeaker_2d(x0, n0, a0)
plt.axis('equal')
\end{verbatim}

\textbf{sfs.array.edge} (Nxy, dx, \texttt{center=[0, 0, 0]}, \texttt{orientation=[1, 0, 0]})
Array along the \texttt{xy-axis} with edge at the origin.

**Parameters**
- \texttt{Nxy (int)} – Number of secondary sources along \texttt{x-} and \texttt{y-axis}.
- \texttt{center ((3,) array_like, optional)} – Position of edge.
- \texttt{orientation ((3,) array_like, optional)} – Normal vector of array. Default orientation is
along \texttt{xy-axis}.

**Returns** \texttt{ArrayData} – Positions, orientations and weights of secondary sources.
Examples

```python
x0, n0, a0 = sfs.array.edge(8, 0.2)
sfs.plot.loudspeaker_2d(x0, n0, a0)
plt.axis('equal')
```

`sfs.array.planar(N, spacing, center=[0, 0, 0], orientation=[1, 0, 0])`

Planar secondary source distribution.

**Parameters**

- **N** *(int or pair of int)* – Number of secondary sources along each edge. If a pair of numbers is given, the first one specifies the number on the horizontal edge, the second one specifies the number on the vertical edge.
• spacing (float) – Distance (in metres) between secondary sources.
• center, orientation – See linear().

Returns ArrayData – Positions, orientations and weights of secondary sources.

sfs.array.cube (N, spacing, center=[0, 0, 0], orientation=[1, 0, 0])
Cube-shaped secondary source distribtion.

Parameters
• N (int or triple of int) – Number of secondary sources along each edge. If a triple of
  numbers is given, the first two specify the edges like in rectangular(), the last one
  specifies the vertical edge.
• spacing (float) – Distance (in metres) between secondary sources.
• center, orientation – See linear(). The orientation corresponds to the first planar
  segment.

Returns ArrayData – Positions, orientations and weights of secondary sources.

sfs.array.sphere_load (fname, radius, center=[0, 0, 0])
Spherical secondary source distribution loaded from datafile.

ASCII Format (see MATLAB SFS Toolbox) with 4 numbers (3 position, 1 weight) per secondary source
located on the unit circle.

Returns ArrayData – Positions, orientations and weights of secondary sources.

sfs.array.load (fname, center=[0, 0, 0], orientation=[1, 0, 0])
Load secondary source positions from datafile.

Comma Seperated Values (CSV) format with 7 values (3 positions, 3 normal vectors, 1 weight) per sec-
ondary source.

Returns ArrayData – Positions, orientations and weights of secondary sources.

sfs.array.weights_midpoint (positions, closed)
Calculate loudspeaker weights for a simply connected array.

The weights are calculated according to the midpoint rule.

Parameters
• positions ((N, 3) array_like) – Sequence of secondary source positions.

Note: The loudspeaker positions have to be ordered on the contour!

• closed (bool) – True if the loudspeaker contour is closed.

Returns (N,) numpy.ndarray – Weights of secondary sources.

sfs.array.concatenate (*arrays)
Concatenate ArrayData objects.

3.2 Tapering

Weights (tapering) for the driving function.

```python
import sfs
import matplotlib.pyplot as plt
import numpy as np
plt.rcParams['figure.figsize'] = 8, 3 # inch
plt.rcParams['axes.grid'] = True
```
active1 = np.zeros(101, dtype=bool)
active1[5:-5] = True

# The active part can wrap around from the end to the beginning:
active2 = np.ones(101, dtype=bool)
active2[30:-10] = False

sfs.tapering.none(active)
No tapering window.

Parameters active(array_like, dtype=bool) – A boolean array containing True for active loudspeakers.

Returns type(active) – The input, unchanged.

Examples

plt.plot(sfs.tapering.none(active1))
plt.axis([-3, 103, -0.1, 1.1])

plt.plot(sfs.tapering.none(active2))
plt.axis([-3, 103, -0.1, 1.1])

sfs.tapering.tukey(active, alpha)
Tukey tapering window.

This uses a function similar to scipy.signal.tukey(), except that the first and last value are not zero.
Parameters

- **active** *(array_like, dtype=bool)* – A boolean array containing True for active loudspeakers.
- **alpha** *(float)* – Shape parameter of the Tukey window, see scipy.signal.tukey().

**Returns** *(len(active),) numpy.ndarray* – Tapering weights.

**Examples**

```python
plt.plot(sfs.tapering.tukey(active1, 0), label='alpha = 0')
plt.plot(sfs.tapering.tukey(active1, 0.25), label='alpha = 0.25')
plt.plot(sfs.tapering.tukey(active1, 0.5), label='alpha = 0.5')
plt.plot(sfs.tapering.tukey(active1, 0.75), label='alpha = 0.75')
plt.plot(sfs.tapering.tukey(active1, 1), label='alpha = 1')
plt.axis([-3, 103, -0.1, 1.1])
plt.legend(loc='lower center')
```

```python
plt.plot(sfs.tapering.tukey(active2, 0.3))
plt.axis([-3, 103, -0.1, 1.1])
```

```python
sfs.tapering.kaiser(active, beta)
```

Kaiser tapering window.

This uses numpy.kaiser().

---
9 [https://docs.scipy.org/doc/numpy/reference/generated/numpy.kaiser.html#numpy.kaiser]
Parameters

- **active** *(array_like, dtype=boo1)* – A boolean array containing True for active loudspeakers.

- **alpha** *(float)* – Shape parameter of the Kaiser window, see `numpy.kaiser()`\(^\text{10}\).

Returns *(len(active),)* *ndarray* – Tapering weights.

**Examples**

```python
plt.plot(sfs.tapering.kaiser(active1, 0), label='beta = 0')
plt.plot(sfs.tapering.kaiser(active1, 2), label='beta = 2')
plt.plot(sfs.tapering.kaiser(active1, 6), label='beta = 6')
plt.plot(sfs.tapering.kaiser(active1, 8.6), label='beta = 8.6')
plt.plot(sfs.tapering.kaiser(active1, 14), label='beta = 14')
plt.axis([-3, 103, -0.1, 1.1])
plt.legend(loc='lower center')
```

![Graph showing various tapering weights with different beta values](image)

```python
plt.plot(sfs.tapering.kaiser(active2, 7))
plt.axis([-3, 103, -0.1, 1.1])
```

![Graph showing tapering weights for active2](image)

---

**4 Frequency Domain**

Submodules for monochromatic sound fields.

\(^{10}\) [https://docs.scipy.org/doc/numpy/reference/generated/numpy.kaiser.html#numpy.kaiser](https://docs.scipy.org/doc/numpy/reference/generated/numpy.kaiser.html#numpy.kaiser)
4.1 Monochromatic Sources

Compute the sound field generated by a sound source.

```python
import sfs
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = 8, 4.5  # inch
x0 = 1.5, 1, 0
f = 500  # Hz
omega = 2 * np.pi * f
normalization_point = 4 * np.pi
normalization_line = np.sqrt(8 * np.pi * omega / sfs.defs.c) * np.exp(1j * np.pi / 4)
grid = sfs.util.xyz_grid([-2, 3], [-1, 2], 0, spacing=0.02)
# Grid for vector fields:
vgrid = sfs.util.xyz_grid([-2, 3], [-1, 2], 0, spacing=0.1)
sfs.mono.source.point(omega, x0, n0, grid, c=None)

sfs.mono.source.point_velocity(omega, x0, n0, grid, c=None)

sfs.mono.source.point_dipole(omega, x0, n0, grid, c=None)
```

Notes

\[
G(x-x0, w) = \frac{1}{4\pi |x-x0|} e^{-j w/c |x-x0|}
\]

Examples

```python
p = sfs.mono.source.point(omega, x0, None, grid)
sfs.plot.soundfield(p, grid)
plt.title("Point Source at {} m".format(x0))

sfs.plot.soundfield(p * normalization_point, grid,
colorbar_kwargs=dict(label="p / Pa"))
plt.title("Point Source at {} m (normalized)".format(x0))
```

sfs.mono.source.point_velocity(omega, x0, n0, grid, c=None)

Velocity of a point source.

Returns `XyzComponents` – Particle velocity at positions given by `grid`.

Examples

The particle velocity can be plotted on top of the sound pressure:

```python
v = sfs.mono.source.point_velocity(omega, x0, None, vgrid)
sfs.plot.soundfield(p * normalization_point, grid)
sfs.plot.vectors(v * normalization_point, vgrid)
plt.title("Sound Pressure and Particle Velocity")
```

sfs.mono.source.point_dipole(omega, x0, n0, grid, c=None)

Point source with dipole characteristics.
Parameters

- **omega** *(float)* – Frequency of source.
- **x0** *(3,) array_like* – Position of source.
- **n0** *(3,) array_like* – Normal vector (direction) of dipole.
- **grid** *(triple of array_like)* – The grid that is used for the sound field calculations. See `sfs.util.xyz_grid()`.
- **c** *(float, optional)* – Speed of sound.

Returns **numpy.ndarray** – Sound pressure at positions given by grid.

Notes

\[
\frac{1}{iw} \frac{(x-x0)}{4\pi c} \frac{1}{|x-x0|^2} e^{-iw/c|x-x0|}
\]

Examples

```python
n0 = 0, 1, 0
p = sfs.mono.source.point_dipole(omega, x0, n0, grid)
sfs.plot.soundfield(p, grid)
plt.title("Dipole Point Source at {} m".format(x0))
```

`sfs.mono.source.point_modal` *(omega, x0, n0, grid, L, N=None, deltan=0, c=None)*

Point source in a rectangular room using a modal room model.

Parameters

- **omega** *(float)* – Frequency of source.
- **x0** *(3,) array_like* – Position of source.
Dipole Point Source at (1.5, 1, 0) m

- \(\mathbf{n}_0\) ((3,) array_like) – Normal vector (direction) of source (only required for compatibility).
- \(\mathbf{grid}\) (triple of array_like) – The grid that is used for the sound field calculations. See \texttt{sfs.util.xyz_grid()}.  
- \(L\) ((3,) array_like) – Dimensions of the rectangular room.
- \(N\) ((3,) array_like or int, optional) – For all three spatial dimensions per dimension maximum order or list of orders. A scalar applies to all three dimensions. If no order is provided it is approximately determined.
- \(\text{deltan}\) (float, optional) – Absorption coefficient of the walls.
- \(c\) (float, optional) – Speed of sound.

**Returns** numpy.ndarray – Sound pressure at positions given by \(\mathbf{grid}\).

\texttt{sfs.mono.source.point_modal_velocity(omega, x0, n0, grid, L, N=\text{None}, \text{deltan}=0, c=\text{None})}

Velocity of point source in a rectangular room using a modal room model.

**Parameters**

- \(\omega\) (float) – Frequency of source.
- \(\mathbf{x}_0\) ((3,) array_like) – Position of source.
- \(\mathbf{n}_0\) ((3,) array_like) – Normal vector (direction) of source (only required for compatibility).
- \(\mathbf{grid}\) (triple of array_like) – The grid that is used for the sound field calculations. See \texttt{sfs.util.xyz_grid()}.  
- \(L\) ((3,) array_like) – Dimensions of the rectangular room.
- \(N\) ((3,) array_like or int, optional) – Combination of modal orders in the three-spatial dimensions to calculate the sound field for or maximum order for all dimensions. If not given, the maximum modal order is approximately determined and the sound field is computed up to this maximum order.
- \(\text{deltan}\) (float, optional) – Absorption coefficient of the walls.
• `c (float, optional)` – Speed of sound.

**Returns**

`XyzComponents` – Particle velocity at positions given by `grid`.

### sfs.mono.source.point_image_sources(omega, x0, n0, grid, L, max_order, coeffs=None, c=None)

Point source in a rectangular room using the mirror image source model.

**Parameters**

- `omega (float)` – Frequency of source.
- `x0 ((3,) array_like)` – Position of source.
- `n0 ((3,) array_like)` – Normal vector (direction) of source (only required for compatibility).
- `grid (triple of array_like)` – The grid that is used for the sound field calculations. See `sfs.util.xyz_grid()`.
- `L ((3,) array_like)` – Dimensions of the rectangular room.
- `max_order (int)` – Maximum number of reflections for each image source.
- `coeffs ((6,) array_like, optional)` – Reflection coefficients of the walls. If not given, the reflection coefficients are set to one.
- `c (float, optional)` – Speed of sound.

**Returns**

`numpy.ndarray` – Sound pressure at positions given by `grid`.

### sfs.mono.source.line(omega, x0, n0, grid, c=None)

Line source parallel to the z-axis.

**Notes**

\[
G(x-x_0, w) = -j/4 H_0 \frac{w}{c |x-x_0|}
\]

**Examples**

```python
p = sfs.mono.source.line(omega, x0, None, grid)
sfs.plot.soundfield(p, grid)
plt.title("Line Source at {} m".format(x0[:2]))

Normalization ...

sfs.plot.soundfield(p, grid,
                   colorbar_kwarg_dict=dict(label="p / Pa"))
plt.title("Line Source at {} m (normalized)".format(x0[:2]))
```

### sfs.mono.source.line_velocity(omega, x0, n0, grid, c=None)

Velocity of line source parallel to the z-axis.

**Returns**

`XyzComponents` – Particle velocity at positions given by `grid`.

**Examples**

The particle velocity can be plotted on top of the sound pressure:
v = sfs.mono.source.line_velocity(omega, x0, None, vgrid)
sfs.plot.soundfield(p * normalization_line, grid)
sfs.plot.vectors(v * normalization_line, vgrid)
plt.title("Sound Pressure and Particle Velocity")

sfs.mono.source.line_dipole(omega, x0, n0, grid, c=None)

Line source with dipole characteristics parallel to the z-axis.
Note: third component of x0 is ignored.

Notes

\[(2)\]
\[G(x-x0, w) = \frac{jk}{4} H_1 \left(\frac{w}{c} |x-x0|\right) \cos(\phi)\]

sfs.mono.source.line_dirichlet_edge(omega, x0, grid, alpha=4.71238898038469, Nc=None, c=None)

Line source scattered at an edge with Dirichlet boundary conditions.

[Mos12], eq.(10.18/19)

Parameters

- **omega** (float) – Angular frequency.
- **x0** ((3,) array_like) – Position of line source.
- **grid** (triple of array_like) – The grid that is used for the sound field calculations. See sfs.util.xyz_grid().
- **alpha** (float, optional) – Outer angle of edge.
- **Nc** (int, optional) – Number of elements for series expansion of driving function. Estimated if not given.
- **c** (float, optional) – Speed of sound

Returns  **ndarray** – Complex pressure at grid positions.
sfs.mono.source.plane \( (\omega, x_0, n_0, \text{grid}, c=None) \)

Plane wave.

**Notes**

\[ G(x, w) = e^{-i \frac{w}{c} n x} \]

**Examples**

```python
direction = 45  # degree
n0 = sfs.util.direction_vector(np.radians(direction))
p = sfs.mono.source.plane(omega, x0, n0, grid)
sfs.plot.soundfield(p, grid, colorbar_kwargs=dict(label="p / Pa"))
plt.title("Plane wave with direction {} degree".format(direction))
```

sfs.mono.source.plane_velocity \( (\omega, x_0, n_0, \text{grid}, c=None) \)

Velocity of a plane wave.

**Notes**

\[ V(x, w) = \frac{1}{\rho c} e^{-i \frac{w}{c} n x} \]

**Returns** \( \text{XyzComponents} \) – Particle velocity at positions given by \( \text{grid} \).

**Examples**

The particle velocity can be plotted on top of the sound pressure:

```python
v = sfs.mono.source.plane_velocity(omega, x0, n0, vgrid)
sfs.plot.soundfield(p, grid)
sfs.plot.vectors(v, vgrid)
plt.title("Sound Pressure and Particle Velocity")
```
4.2 Monochromatic Driving Functions

Compute driving functions for various systems.

\texttt{sfs.mono.drivingfunction.wfs\_2d\_line(omega, x0, n0, xs, c=None)}

Line source by 2-dimensional WFS.

\[ D(x_0, k) = \frac{j}{2} k (x_0 - x_s) n_0 / |x_0 - x_s| \ast H_1(k |x_0 - x_s|) \]

\texttt{sfs.mono.drivingfunction.wfs\_2d\_point(omega, x0, n0, xs, c=None)}

Point source by two- or three-dimensional WFS.

\[ D(x_0, k) = j \frac{(x_0 - x_s) n_0}{|x_0 - x_s|^{3/2}} e^{-j k |x_0 - x_s|} \]

\texttt{sfs.mono.drivingfunction.wfs\_25d\_point(omega, x0, n0, xs, xref=[0, 0, 0], c=None, omalias=None)}

Point source by 2.5-dimensional WFS.

\[ D(x_0, k) = \frac{j}{k} \frac{(x_0 - x_s) n_0}{|x_0 - x_s|^{3/2}} e^{-j k |x_0 - x_s|} \]

\texttt{sfs.mono.drivingfunction.wfs\_3d\_point(omega, x0, n0, xs, c=None)}

Point source by two- or three-dimensional WFS.

\[ D(x_0, k) = j \frac{(x_0 - x_s) n_0}{|x_0 - x_s|^{3/2}} e^{-j k |x_0 - x_s|} \]

\texttt{sfs.mono.drivingfunction.wfs\_2d\_plane(omega, x0, n0, n=[0, 1, 0], c=None)}

Plane wave by two- or three-dimensional WFS.

Eq.(17) from [SRA08]:
\[ D(x_0, k) = jk n n_0 e^{-j k n x_0} \]

`sfs.mono.drivingfunction.wfs_25d_plane`(
  \( \omega, x_0, n_0, n=\{0, 1, 0\}, xref=\{0, 0, 0\}, c=None, onalias=None \)
)

Plane wave by 2.5-dimensional WFS.

\[ D_{2.5D}(x_0, w) = \sqrt{|j k |xref-x_0| n n_0 e^{-j k n x_0}} \]

`sfs.mono.drivingfunction.wfs_3d_plane`(
  \( \omega, x_0, n_0, n=\{0, 1, 0\}, c=None \)
)

Plane wave by two- or three-dimensional WFS.

Eq.(17) from [SRA08]:

\[ D(x_0, k) = jk n n_0 e^{-j k n x_0} \]

`sfs.mono.drivingfunction.wfs_2d_focused`(
  \( \omega, x_0, n_0, c=None \)
)

Focused source by two- or three-dimensional WFS.

\[ D(x_0, k) = \frac{(x_0-x_0) n_0}{|x_0-x_0|^{3/2}} e^{j k |x_0-x_0|} \]

`sfs.mono.drivingfunction.wfs_25d_focused`(
  \( \omega, x_0, n_0, x_0, c=None, onalias=None \)
)

Focused source by 2.5-dimensional WFS.

\[ D(x_0, w) = \frac{(x_0-x_0) n_0}{|x_0-x_0|^{3/2}} e^{j k |x_0-x_0|} \]

`sfs.mono.drivingfunction.wfs_3d_focused`(
  \( \omega, x_0, n_0, x_0, c=None \)
)

Focused source by two- or three-dimensional WFS.

\[ D(x_0, k) = \frac{(x_0-x_0) n_0}{|x_0-x_0|^{3/2}} e^{j k |x_0-x_0|} \]

`sfs.mono.drivingfunction.wfs_25d_preeq`(
  \( \omega, onalias, c \)
)

Prequalification for 2.5D WFS.

`sfs.mono.drivingfunction.delay_3d_plane`(
  \( \omega, x_0, n_0, n=\{0, 1, 0\}, c=None \)
)

Plane wave by simple delay of secondary sources.

`sfs.mono.drivingfunction.source_selection_plane`(
  \( n_0, n \)
)

Secondary source selection for a plane wave.

Eq.(13) from [SRA08]

`sfs.mono.drivingfunction.source_selection_point`(
  \( n_0, x_0, x_0 \)
)

Secondary source selection for a point source.

Eq.(15) from [SRA08]

`sfs.mono.drivingfunction.source_selection_line`(
  \( n_0, x_0, x_0 \)
)

Secondary source selection for a line source.

compare Eq.(15) from [SRA08]

`sfs.mono.drivingfunction.source_selection_focused`(
  \( ns, x_0, x_0 \)
)

Secondary source selection for a focused source.

Eq.(2.78) from [Wie14]

`sfs.mono.drivingfunction.source_selection_all`(
  \( N \)
)

Select all secondary sources.
Plane wave by two-dimensional NFC-HOA.

\[
D(\phi_0, \omega) = -\frac{2\beta}{\pi r_0} \sum_{m=-M}^{M} \frac{\beta^{-m}}{2mr_0} \delta m(\phi_0 - \phi_{pw})
\]

See http://sfstoolbox.org/#equation-D.nfchoa.pw.2D.

Plane wave by 2.5-dimensional NFC-HOA.

\[
D(\phi_0, \omega) = \frac{2}{r_0} \sum_{m=-M}^{M} \frac{\beta^{-|m|}}{2|m|r_0} \delta m(\phi_0 - \phi_{pw})
\]

See http://sfstoolbox.org/#equation-D.nfchoa.pw.2.5D.

Plane wave by two-dimensional SDM.

The secondary sources have to be located on the x-axis (y0=0). Derived from [SA09], Eq.(9), Eq.(4):

\[
D(x0,k) = -
\]

Plane wave by two-dimensional SDM.

The secondary sources have to be located on the x-axis (y0=0). Derived from [Ahr12], Eq.(3.73), Eq.(C.5), Eq.(C.11):

\[
D(x0,k) = k_{pw,y} \times e^{-jk_{pw,x}x}
\]

Plane wave by 2.5-dimensional SDM.

The secondary sources have to be located on the x-axis (y0=0). Eq.(3.79) from [Ahr12]:

\[
D_{2.5D}(x0,\omega) =
\]

Point source by 2.5-dimensional SDM.

The secondary sources have to be located on the x-axis (y0=0). Driving function from [SA10], Eq.(24):

\[
D(x0,k) =
\]

Plane wave by two-dimensional ESA for an edge-shaped secondary source distribution consisting of monopole line sources.
One leg of the secondary sources has to be located on the x-axis (y0=0), the edge at the origin.

Derived from [SSR16]

**Parameters**

- **omega** (float) – Angular frequency.
- **x0** (int(N, 3) array_like) – Sequence of secondary source positions.
- **n** ((3,) array_like, optional) – Normal vector of synthesized plane wave.
- **alpha** (float, optional) – Outer angle of edge.
- **Nc** (int, optional) – Number of elements for series expansion of driving function. Estimated if not given.
- **c** (float, optional) – Speed of sound

**Returns** (N,) numpy.ndarray – Complex weights of secondary sources.

sfs.mono.drivingfunction.esa_edge_dipole_2d_plane(omega, x0, n=[0, 1, 0], alpha=4.71238898038469, Nc=None, c=None)

**Plane wave by two-dimensional ESA for an edge-shaped secondary source** distribution consisting of dipole line sources.

One leg of the secondary sources has to be located on the x-axis (y0=0), the edge at the origin.

Derived from [SSR16]

**Parameters**

- **omega** (float) – Angular frequency.
- **x0** (int(N, 3) array_like) – Sequence of secondary source positions.
- **xs** ((3,) array_like) – Position of synthesized line source.
- **alpha** (float, optional) – Outer angle of edge.
- **Nc** (int, optional) – Number of elements for series expansion of driving function. Estimated if not given.
- **c** (float, optional) – Speed of sound

**Returns** (N,) numpy.ndarray – Complex weights of secondary sources.

sfs.mono.drivingfunction.esa_edge_2d_line(omega, x0, xs, alpha=4.71238898038469, Nc=None, c=None)

**Line source by two-dimensional ESA for an edge-shaped secondary source** distribution consisting of monopole line sources.

One leg of the secondary sources have to be located on the x-axis (y0=0), the edge at the origin.

Derived from [SSR16]

**Parameters**

- **omega** (float) – Angular frequency.
- **x0** (int(N, 3) array_like) – Sequence of secondary source positions.
- **xs** ((3,) array_like) – Position of synthesized line source.
- **alpha** (float, optional) – Outer angle of edge.
- **Nc** (int, optional) – Number of elements for series expansion of driving function. Estimated if not given.
- **c** (float, optional) – Speed of sound

**Returns** (N,) numpy.ndarray – Complex weights of secondary sources.
### 4.3 Monochromatic Sound Fields

Computation of synthesized sound fields.

- **sfs.mono.synthesized.generic(omega, x0, n0, d, grid, c=None, source=<function point>)**
  
  Compute sound field for a generic driving function.

- **sfs.mono.synthesized.shiftphase(p, phase)**
  
  Shift phase of a sound field.
5 Time Domain

5.1 Time Domain Sources

Compute the sound field generated by a sound source. The Green’s function describes the spatial sound propagation over time.

`sfs.time.source.point(xs, signal, observation_time, grid, c=None)`

Source model for a point source: 3D Green’s function.

Calculates the scalar sound pressure field for a given point in time, evoked by source excitation signal.

Parameters

- `xs` ((3,) array_like) – Position of source in cartesian coordinates.
- `signal` ((N,) array_like + float) – Excitation signal consisting of (mono) audio data and a sampling rate (in Hertz). A `DelayedSignal` object can also be used.
- `observation_time` (float) – Observed point in time.
- `grid` (triple of array_like) – The grid that is used for the sound field calculations. See `sfs.util.xyz_grid()`.
- `c` (float, optional) – Speed of sound.

Returns `numpy.ndarray` – Scalar sound pressure field, evaluated at positions given by `grid`.

Notes

\[
g(x - x_s, t) = \frac{1}{4\pi|x - x_s|} - \frac{|x - x_s|}{c}
\]

`sfs.time.source.point_image_sources(x0, signal, observation_time, grid, L, max_order, coeffs=None, c=None)`

Point source in a rectangular room using the mirror image source model.

Parameters

- `x0` ((3,) array_like) – Position of source in cartesian coordinates.
- `signal` ((N,) array_like + float) – Excitation signal consisting of (mono) audio data and a sampling rate (in Hertz). A `DelayedSignal` object can also be used.
- `observation_time` (float) – Observed point in time.
- `grid` (triple of array_like) – The grid that is used for the sound field calculations. See `sfs.util.xyz_grid()`.
- `L` ((3,) array_like) – Dimensions of the rectangular room.
- `max_order` (int) – Maximum number of reflections for each image source.
- `coeffs` ((6,) array_like, optional) – Reflection coeffecients of the walls. If not given, the reflection coefficients are set to one.
- `c` (float, optional) – Speed of sound.

Returns `numpy.ndarray` – Scalar sound pressure field, evaluated at positions given by `grid`.
5.2 Time Domain Driving Functions

Compute time based driving functions for various systems.

sfs.time.drivingfunction.wfs_25d_plane (x0, n0, n=[0, 1, 0], xref=[0, 0, 0], c=None)
Plane wave model by 2.5-dimensional WFS.

Parameters
- x0 ((N, 3) array_like) – Sequence of secondary source positions.
- n0 ((N, 3) array_like) – Sequence of secondary source orientations.
- n ((3,) array_like, optional) – Normal vector (propagation direction) of synthesized plane wave.
- xref ((3,) array_like, optional) – Reference position
- c (float, optional) – Speed of sound

Returns
- delays ((N,) numpy.ndarray) – Delays of secondary sources in seconds.
- weights ((N,) numpy.ndarray) – Weights of secondary sources.

Notes
2.5D correction factor

\[ g_0 = \sqrt{2\pi|\mathbf{x}_\text{ref} - \mathbf{x}_0|} \]

d using a plane wave as source model

\[ d_{2.5D}(x_0, t) = h(t)2g_0 n_0 t - \frac{1}{c}nx_0 \]

with wfs(2.5D) prefilter h(t), which is not implemented yet.

References
See http://sfstoolbox.org/en/latest/#equation-d.wfs.pw.2.5D

sfs.time.drivingfunction.wfs_25d_point (x0, n0, xs, xref=[0, 0, 0], c=None)
Point source by 2.5-dimensional WFS.

Parameters
- x0 ((N, 3) array_like) – Sequence of secondary source positions.
- n0 ((N, 3) array_like) – Sequence of secondary source orientations.
- xs ((3,) array_like) – Virtual source position.
- xref ((3,) array_like, optional) – Reference position
- c (float, optional) – Speed of sound

Returns
- delays ((N,) numpy.ndarray) – Delays of secondary sources in seconds.
- weights ((N,) numpy.ndarray) – Weights of secondary sources.
Notes

2.5D correction factor

\[ g_0 = \sqrt{2\pi|x_{\text{ref}} - x_0|} \]

d using a point source as source model

\[ d_{2.5D}(x_0, t) = h(t) \frac{g_0(x_0 - x_s)n_0}{2\pi|x_0 - x_s|^{3/2}} t + \frac{|x_0 - x_s|}{c} \]

with wfs(2.5D) prefilter h(t), which is not implemented yet.

References

See [http://sfstoolbox.org/en/latest/#equation-d.wfs.ps.2.5D](http://sfstoolbox.org/en/latest/#equation-d.wfs.ps.2.5D)

**sfs.time.drivingfunction**.wfs_25d_focused \((x_0, n_0, x_s, \text{xref}=[0, 0, 0], c=None)\)

Point source by 2.5-dimensional WFS.

**Parameters**

- **x0** \( ((N, 3) \text{ array_like}) \) – Sequence of secondary source positions.
- **n0** \( ((N, 3) \text{ array_like}) \) – Sequence of secondary source orientations.
- **xs** \( ((3,) \text{ array_like}) \) – Virtual source position.
- **xref** \( ((3,) \text{ array_like, optional}) \) – Reference position
- **c** \( \text{(float, optional)} \) – Speed of sound

**Returns**

- **delays** \( ((N,) \text{ numpy.ndarray}) \) – Delays of secondary sources in seconds.
- **weights** \( ((N,) \text{ numpy.ndarray}) \) – Weights of secondary sources.

Notes

2.5D correction factor

\[ g_0 = \sqrt{\frac{|x_{\text{ref}} - x_0|}{|x_0 - x_s| + |x_{\text{ref}} - x_0|}} \]

d using a point source as source model

\[ d_{2.5D}(x_0, t) = h(t) \frac{g_0(x_0 - x_s)n_0}{|x_0 - x_s|^{3/2}} t + \frac{|x_0 - x_s|}{c} \]

with wfs(2.5D) prefilter h(t), which is not implemented yet.

References

See [http://sfstoolbox.org/en/latest/#equation-d.wfs.fs.2.5D](http://sfstoolbox.org/en/latest/#equation-d.wfs.fs.2.5D)

**sfs.time.drivingfunction**.driving_signals \((\text{delays, weights, signal})\)

Get driving signals per secondary source.

Returned signals are the delayed and weighted mono input signal (with N samples) per channel (C).

**Parameters**

- **delays** \( ((C,) \text{ array_like}) \) – Delay in seconds for each channel, negative values allowed.
• weights \((C,) \text{array_like}\) – Amplitude weighting factor for each channel.

• signal \((N,) \text{array_like} + \text{float}\) – Excitation signal consisting of (mono) audio data and a sampling rate (in Hertz). A DelayedSignal object can also be used.

Returns DelayedSignal – A tuple containing the driving signals (in a numpy.ndarray\(^{11}\) with shape \((N, C)\)), followed by the sampling rate (in Hertz) and a (possibly negative) time offset (in seconds).

sfs.time.drivingfunction.apply_delays (signal, delays)
Apply delays for every channel.

Parameters

• signal \((N,) \text{array_like} + \text{float}\) – Excitation signal consisting of (mono) audio data and a sampling rate (in Hertz). A DelayedSignal object can also be used.

• delays \((C,) \text{array_like}\) – Delay in seconds for each channel (C), negative values allowed.

Returns DelayedSignal – A tuple containing the delayed signals (in a numpy.ndarray\(^{12}\) with shape \((N, C)\)), followed by the sampling rate (in Hertz) and a (possibly negative) time offset (in seconds).

5.3 Time Domain Sound Fields

Compute sound field.

sfs.time.soundfield.p_array (x0, signals, weights, observation_time, grid, source=<function point>, c=None)
Compute sound field for an array of secondary sources.

Parameters

• x0 \((N, 3) \text{array_like}\) – Sequence of secondary source positions.

• signals \((N, C) \text{array_like} + \text{float}\) – Driving signals consisting of audio data (C channels) and a sampling rate (in Hertz). A DelayedSignal object can also be used.

• weights \((C,) \text{array_like}\) – Additional weights applied during integration, e.g. source tapering.

• observation_time (float) – Simulation point in time (seconds).

• grid (triple of array_like) – The grid that is used for the sound field calculations. See sfs.util.xyz_grid().

• source (function, optional) – Source type is a function, returning scalar field. For default, see sfs.time.source.point().

• c (float, optional) – Speed of sound.

Returns numpy.ndarray – Sound pressure at grid positions.

6 Plotting

Plot sound fields etc.

sfs.plot.virtualsource_2d (xs, ns=None, type=’point’, ax=None)
Draw position/orientation of virtual source.

sfs.plot.reference_2d (xref, size=0.1, ax=None)
Draw reference/normalization point.

---

\(^{11}\) https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray

\(^{12}\) https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
sfs.plot.secondarysource_2d(x0, n0, grid=None)

Simple plot of secondary source locations.

sfs.plot.loudspeaker_2d(x0, n0, a0=0.5, size=0.08, show_numbers=False, grid=None, ax=None)

Draw loudspeaker symbols at given locations and angles.

Parameters

- **x0** 
  
  \((N, 3)\) array_like
  
  - Loudspeaker positions.

- **n0** 
  
  \((N, 3)\) or \((3,)\) array_like
  
  - Normal vector(s) of loudspeakers.

- **a0** 
  
  float or \((N,)\) array_like, optional
  
  - Weighting factor(s) of loudspeakers.

- **size** 
  
  float, optional
  
  - Size of loudspeakers in metres.

- **show_numbers** 
  
  bool, optional
  
  - If True, loudspeaker numbers are shown.

- **grid** 
  
  triple of array_like, optional
  
  - If specified, only loudspeakers within the grid are shown.

- **ax** 
  
  Axes object, optional
  
  - The loudspeakers are plotted into this matplotlib.axes.Axes object or – if not specified – into the current axes.

sfs.plot.loudspeaker_3d(x0, n0, a0=None, w=0.08, h=0.08)

Plot positions and normals of a 3D secondary source distribution.

sfs.plot.soundfield(p, grid, xnorm=None, cmap='coolwarm_clip', vmin=-2.0, vmax=2.0, xlabel=None, ylabel=None, colorbar=True, colorbar_kwargs={}, ax=None, **kwargs)

Two-dimensional plot of sound field.

Parameters

- **p** 
  
  array_like
  
  - Sound pressure values (or any other scalar quantity if you like). If the values are complex, the imaginary part is ignored. Typically, \(p\) is two-dimensional with a shape of \((Ny, Nx)\), \((Nz, Nx)\) or \((Nz, Ny)\). This is the case if sfs.util.xyz_grid() was used with a single number for \(z\), \(y\) or \(x\), respectively. However, \(p\) can also be three-dimensional with a shape of \((Ny, Nx, 1)\), \((1, Nx, Nz)\) or \((Ny, 1, Nz)\). This is the case if numpy.meshgrid() was used with a scalar for \(z\), \(y\) or \(x\), respectively (and of course with the default indexing='xy').

- **grid** 
  
  triple or pair of numpy.ndarray
  
  - The grid that was used to calculate \(p\), see sfs.util.xyz_grid(). If \(p\) is two-dimensional, but \(grid\) has 3 components, one of them must be scalar.

- **xnorm** 
  
  array_like, optional
  
  - Coordinates of a point to which the sound field should be normalized before plotting. If not specified, no normalization is used. See sfs.util.normalize().

Note: If you want to plot a single slice of a pre-computed “full” 3D sound field, make sure that the slice still has three dimensions (including one singleton dimension). This way, you can use the original grid of the full volume without changes. This works because the grid component corresponding to the singleton dimension is simply ignored.

- **grid** 
  
  triple or pair of numpy.ndarray
  
  - The grid that was used to calculate \(p\), see sfs.util.xyz_grid(). If \(p\) is two-dimensional, but \(grid\) has 3 components, one of them must be scalar.

Returns AxesImage – See matplotlib.pyplot.imshow()\(^{15}\).

Other Parameters

\(^{13}\) https://matplotlib.org/api/axes_api.html#matplotlib.axes.Axes

\(^{14}\) https://docs.scipy.org/doc/numpy/reference/generated/numpy.meshgrid.html#numpy.meshgrid

\(^{15}\) https://matplotlib.org/api/_as_gen/matplotlib.pyplot.imshow.html#matplotlib.pyplot.imshow
• xlabel, ylabel (str) – Overwrite default x/y labels. Use xlabel=' ' and ylabel=' ' to remove x/y labels. The labels can be changed afterwards with matplotlib.pyplot.xlabel()\(^{16}\) and matplotlib.pyplot.ylabel()\(^{17}\).

• colorbar (bool, optional) – If False, no colorbar is created.

• colorbar_kwargs (dict, optional) – Further colorbar arguments, see add_colorbar().

• ax (Axes, optional) – If given, the plot is created on ax instead of the current axis (see matplotlib.pyplot.gca()\(^{18}\)).

• cmap, vmin, vmax, **kwargs – All further parameters are forwarded to matplotlib.pyplot.imshow()\(^{19}\).

See also:

sfs.plot.level()

sfs.plot.level(p, grid, xnorm=None, power=False, cmap=None, vmax=3, vmin=-50, **kwargs)

Two-dimensional plot of level (dB) of sound field.

Takes the same parameters as sfs.plot.soundfield().

Other Parameters power (bool, optional) – See sfs.util.db().

sfs.plot.particles(x, trim=None, ax=None, xlabel='x (m)', ylabel='y (m)', edgecolor=' ', **kwargs)

Plot particle positions as scatter plot

sfs.plot.vectors(v, grid, cmap='blacktransparent', headlength=3, headaxislength=2.5, ax=None, clim=None, **kwargs)

Plot a vector field in the xy plane.

Parameters

• v (triple or pair of array_like) – x, y and optionally z components of vector field. The z components are ignored. If the values are complex, the imaginary parts are ignored.

• grid (triple or pair of array_like) – The grid that was used to calculate v, see sfs.util.xyz_grid(). Any z components are ignored.

Returns Quiver – See matplotlib.pyplot.quiver()\(^{20}\).

Other Parameters

• ax (Axes, optional) – If given, the plot is created on ax instead of the current axis (see matplotlib.pyplot.gca()\(^{21}\)).

• clim (pair of float, optional) – Limits for the scaling of arrow colors. See matplotlib.pyplot.quiver()\(^{22}\).

• cmap, headlength, headaxislength, **kwargs – All further parameters are forwarded to matplotlib.pyplot.quiver()\(^{23}\).

sfs.plot.add_colorbar(im, aspect=20, pad=0.5, **kwargs)

Add a vertical color bar to a plot.

Parameters

• im (ScalarMappable) – The output of sfs.plot.soundfield(), sfs.plot.level() or any other matplotlib.cm.ScalarMappable\(^{24}\).

---

\(^{16}\) https://matplotlib.org/api/_as_gen/matplotlib.pyplot.xlabel.html#matplotlib.pyplot.xlabel
\(^{17}\) https://matplotlib.org/api/_as_gen/matplotlib.pyplot.ylabel.html#matplotlib.pyplot.ylabel
\(^{18}\) https://matplotlib.org/api/_as_gen/matplotlib.pyplot.gca.html#matplotlib.pyplot.gca
\(^{19}\) https://matplotlib.org/api/_as_gen/matplotlib.pyplot.imshow.html#matplotlib.pyplot.imshow
\(^{20}\) https://matplotlib.org/api/_as_gen/matplotlib.pyplot.quiver.html#matplotlib.pyplot.quiver
\(^{21}\) https://matplotlib.org/api/_as_gen/matplotlib.pyplot.gca.html#matplotlib.pyplot.gca
\(^{22}\) https://matplotlib.org/api/_as_gen/matplotlib.pyplot.quiver.html#matplotlib.pyplot.quiver
\(^{23}\) https://matplotlib.org/api/_as_gen/matplotlib.pyplot.quiver.html#matplotlib.pyplot.quiver
\(^{24}\) https://matplotlib.org/api/cm_api.html#matplotlib.cm.ScalarMappable
• **aspect** *(float, optional)* – Aspect ratio of the colorbar. Strictly speaking, since the colorbar is vertical, it’s actually the inverse of the aspect ratio.

• **pad** *(float, optional)* – Space between image plot and colorbar, as a fraction of the width of the colorbar.

___

**Note:** The *pad* argument of `matplotlib.figure.Figure.colorbar()`\(^{25}\) has a slightly different meaning (“fraction of original axes”!)

• **kwargs** – All further arguments are forwarded to `matplotlib.figure.Figure.colorbar()`\(^{26}\).

See also:

`matplotlib.pyplot.colorbar()`\(^{27}\)

### 7 Utilities

Various utility functions.

**sfs.util.rotation_matrix** *(n1, n2)*

Compute rotation matrix for rotation from *n1* to *n2*.

**Parameters**

- *n1, n2* *(3,) array_like* – Two vectors. They don’t have to be normalized.

**Returns** *(3, 3) numpy.ndarray* – Rotation matrix.

**sfs.util.wavenumber** *(omega, c=None)*

Compute the wavenumber for a given radial frequency.

**sfs.util.direction_vector** *(alpha, beta=1.5707963267948966)*

Compute normal vector from azimuth, colatitude.

**sfs.util.sph2cart** *(alpha, beta, r)*

Spherical to cartesian coordinate transform.

\[
\begin{align*}
x &= r \cos \alpha \sin \beta \\
y &= r \sin \alpha \sin \beta \\
z &= r \cos \beta
\end{align*}
\]

with $\alpha \in [0, 2\pi), \beta \in [0, \pi], r \geq 0$

**Parameters**

• **alpha** *(float or array_like)* – Azimuth angle in radians

• **beta** *(float or array_like)* – Elevation angle in radians (with 0 denoting North pole)

• **r** *(float or array_like)* – Radius

**Returns**

• **x** *(float or array_like)* – x-component of Cartesian coordinates

• **y** *(float or array_like)* – y-component of Cartesian coordinates

• **z** *(float or array_like)* – z-component of Cartesian coordinates

\(^{25}\) https://matplotlib.org/api/_as_gen/matplotlib.figure.Figure.colorbar.html#matplotlib.figure.Figure.colorbar
\(^{26}\) https://matplotlib.org/api/_as_gen/matplotlib.figure.Figure.colorbar.html#matplotlib.figure.Figure.colorbar
\(^{27}\) https://matplotlib.org/api/_as_gen/matplotlib.pyplot.colorbar.html#matplotlib.pyplot.colorbar

34
sfs.util.cart2sph(x, y, z)

Cartesian to spherical coordinate transform.

\[
\begin{align*}
\alpha &= \arctan\left(\frac{y}{x}\right) \\
\beta &= \arccos\left(\frac{z}{r}\right) \\
r &= \sqrt{x^2 + y^2 + z^2}
\end{align*}
\]

with \(\alpha \in [0, 2\pi], \beta \in [0, \pi], r \geq 0\)

**Parameters**

- `x` (float or array_like) – x-component of Cartesian coordinates
- `y` (float or array_like) – y-component of Cartesian coordinates
- `z` (float or array_like) – z-component of Cartesian coordinates

**Returns**

- `alpha` (float or array_like) – Azimuth angle in radians
- `beta` (float or array_like) – Elevation angle in radians (with 0 denoting North pole)
- `r` (float or array_like) – Radius

sfs.util.asarray_1d(a, **kwargs)

Squeeze the input and check if the result is one-dimensional.

Returns `a` converted to a `numpy.ndarray` and stripped of all singleton dimensions. Scalars are “upgraded” to 1D arrays. The result must have exactly one dimension. If not, an error is raised.

sfs.util.asarray_of_rows(a, **kwargs)

Convert to 2D array, turn column vector into row vector.

Returns `a` converted to a `numpy.ndarray` and stripped of all singleton dimensions. If the result has exactly one dimension, it is re-shaped into a 2D row vector.

sfs.util.as_xyz_components(components, **kwargs)

Convert `components` to `XyzComponents` of `numpy.ndarray`.

The `components` are first converted to NumPy arrays (using `numpy.asarray()`) which are then assembled into an `XyzComponents` object.

**Parameters**

- `components` (triple or pair of array_like) – The values to be used as X, Y and Z arrays. Z is optional.
- `**kwargs` – All further arguments are forwarded to `numpy.asarray()`, which is applied to the elements of `components`.

sfs.util.as_delayed_signal(arg, **kwargs)

Make sure that the given argument can be used as a signal.

**Parameters**

- `arg` (sequence of 1 array_like followed by 1 or 2 scalars) – The first element is converted to a NumPy array, the second element is used as the sampling rate (in Hertz) and the optional third element is used as the starting time of the signal (in seconds). Default starting time is 0.
- `**kwargs` – All keyword arguments are forwarded to `numpy.asarray()`.

---

28 https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
29 https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
30 https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
31 https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
32 https://docs.scipy.org/doc/numpy/reference/generated/numpy.asarray.html#numpy.asarray
33 https://docs.scipy.org/doc/numpy/reference/generated/numpy.asarray.html#numpy.asarray
Returns **DelayedSignal** – A named tuple consisting of a `numpy.ndarray` containing the audio data, followed by the sampling rate (in Hertz) and the starting time (in seconds) of the signal.

**Examples**

Typically, this is used together with tuple unpacking to assign the audio data, the sampling rate and the starting time to separate variables:

```python
>>> import sfs
>>> sig = [1], 44100
>>> data, fs, signal_offset = sfs.util.as_delayed_signal(sig)
>>> data
array([1])
>>> fs
44100
>>> signal_offset
0
```

`sfs.util.strict_arange(start, stop, step=1, endpoint=False, dtype=None, **kwargs)`

Like `numpy.arange()`\(^{35}\), but compensating numeric errors.

Unlike `numpy.arange()`\(^{36}\), but similar to `numpy.linspace()`\(^{37}\), providing `endpoint=True` includes both endpoints.

**Parameters**

- `start, stop, step, dtype` – See `numpy.arange()`\(^{38}\).
- `endpoint` – See `numpy.linspace()`\(^{39}\).

**Note**: With `endpoint=True`, the difference between `start` and `end` value must be an integer multiple of the corresponding `spacing` value!

- `**kwargs` – All further arguments are forwarded to `numpy.isclose()`\(^{40}\).

**Returns** `numpy.ndarray` – Array of evenly spaced values. See `numpy.arange()`\(^{41}\).

`sfs.util.xyz_grid(x, y, z, spacing, endpoint=True, **kwargs)`

Create a grid with given range and spacing.

**Parameters**

- `x, y, z (float or pair of float)` – Inclusive range of the respective coordinate or a single value if only a slice along this dimension is needed.
- `spacing (float or triple of float)` – Grid spacing. If a single value is specified, it is used for all dimensions, if multiple values are given, one value is used per dimension. If a dimension (x, y or z) has only a single value, the corresponding spacing is ignored.
- `endpoint (bool, optional)` – If `True` (the default), the endpoint of each range is included in the grid. Use `False` to get a result similar to `numpy.arange()`\(^{42}\). See `strict_arange()`.
- `**kwargs` – All further arguments are forwarded to `strict_arange()`.

---

\(^{34}\) https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray  
\(^{35}\) https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html#numpy.arange  
\(^{36}\) https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html#numpy.arange  
\(^{37}\) https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html#numpy.linspace  
\(^{38}\) https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html#numpy.arange  
\(^{39}\) https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html#numpy.linspace  
\(^{40}\) https://docs.scipy.org/doc/numpy/reference/generated/numpy.isclose.html#numpy.isclose  
\(^{41}\) https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html#numpy.arange  
\(^{42}\) https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html#numpy.arange
Returns `XyzComponents` – A grid that can be used for sound field calculations.

See also:

- `strict_arange()`, `numpy.meshgrid()`
- `sfs.util.normalize(p, grid, xnorm)`
  Normalize sound field wrt position `xnorm`.
- `sfs.util.probe(p, grid, x)`
  Determine the value at position `x` in the sound field `p`.
- `sfs.util.broadcast_zip(*args)`
  Broadcast arguments to the same shape and then use `zip()`.
- `sfs.util.normalize_vector(x)`
  Normalize a 1D vector.
- `sfs.util.displacement(v, omega)`
  Particle displacement
  \[
  d(x, t) = \int_{0}^{t} v(x, t) dt
  \]
- `sfs.util.db(x, power=False)`
  Convert `x` to decibel.

Parameters

- `x (array_like)` – Input data. Values of 0 lead to negative infinity.
- `power (bool, optional)` – If `power=False` (the default), `x` is squared before conversion.

**class** `sfs.util.XyzComponents(components)`

Triple (or pair) of components: `x`, `y`, and optionally `z`.

Instances of this class can be used to store coordinate grids (either regular grids like in `xyz_grid()` or arbitrary point clouds) or vector fields (e.g. particle velocity).

This class is a subclass of `numpy.ndarray`. It is one-dimensional (like a plain list) and has a length of 3 (or 2, if no z-component is available). It uses `dtype=object` in order to be able to store other `numpy.ndarray`s of arbitrary shapes but also scalars, if needed. Because it is a NumPy array subclass, it can be used in operations with scalars and “normal” NumPy arrays, as long as they have a compatible shape. Like any NumPy array, instances of this class are iterable and can be used, e.g., in for-loops and tuple unpacking. If slicing or broadcasting leads to an incompatible shape, a plain `numpy.ndarray` with `dtype=object` is returned.

To make sure the `components` are NumPy arrays themselves, use `as_xyz_components()`.

Parameters

- `components ((3,) or (2,) array_like)` – The values to be used as X, Y and Z data. Z is optional.
  - `x`: x-component.
  - `y`: y-component.
  - `z`: z-component (optional).

---

43 https://docs.scipy.org/doc/numpy/reference/generated/numpy.meshgrid.html#numpy.meshgrid
44 https://docs.python.org/3/library/functions.html#zip
45 https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
46 https://docs.python.org/3/library/stdtypes.html#list
47 https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
48 https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
apply (func, *args, **kwargs)
Apply a function to each component.

The function `func` will be called once for each component, passing the current component as first argument. All further arguments are passed after that. The results are returned as a new `XyzComponents` object.

class sfs.util.DelayedSignal
A tuple of audio data, sampling rate and start time.

This class (a `collections.namedtuple`) is not meant to be instantiated by users.

To pass a signal to a function, just use a simple `tuple` or `list` containing the audio data and the sampling rate (in Hertz), with an optional starting time (in seconds) as a third item. If you want to ensure that a given variable contains a valid signal, use `sfs.util.as_delayed_signal()`.

data
Alias for field number 0

samplerate
Alias for field number 1

time
Alias for field number 2

sfs.util.image_sources_for_box(x, L, N, prune=True)
Image source method for a cuboid room.

The classical method by Allen and Berkley [AB79].

Parameters
- x ((D,) array_like) – Original source location within box. Values between 0 and corresponding side length.
- L ((D,) array_like) – side lengths of room.
- N (int) – Maximum number of reflections per image source, see below.
- prune (bool, optional) – selection of image sources:
  – If True (default): Returns all images reflected up to N times. This is the usual interpretation of N as “maximum order”.
  – If False: Returns reflected up to N times between individual wall pairs, a total number of \( M := (2N + 1)^D \). This larger set is useful e.g. to select image sources based on distance to listener, as suggested by [Bor84].

Returns
- xs ((M, D) array_like) – original & image source locations.
- wall_count ((M, 2D) array_like) – number of reflections at individual walls for each source.

sfs.util.spherical_hn2(n, z)
Spherical Hankel function of 2nd kind.

Defined as http://dlmf.nist.gov/10.47.E6,

\[
2nz = \sqrt{\pi z} 2n + \frac{1}{2},
\]

where \( 2n \cdot z \) is the Hankel function of the second kind and n-th order, and \( z \) its complex argument.

Parameters

49 https://docs.python.org/3/library/collections.html#collections.namedtuple
50 https://docs.python.org/3/library/stdtypes.html#tuple
51 https://docs.python.org/3/library/stdtypes.html#list
• `n (array_like)` – Order of the spherical Hankel function (n >= 0).
• `z (array_like)` – Argument of the spherical Hankel function.

8 References

9 Contributing

If you find errors, omissions, inconsistencies or other things that need improvement, please create an issue or a pull request at https://github.com/sfstoolbox/sfs-python/. Contributions are always welcome!

9.1 Development Installation

Instead of pip-installing the latest release from PyPI, you should get the newest development version from Github:

```
git clone https://github.com/sfstoolbox/sfs-python.git
cd sfs-python
python3 setup.py develop --user
```

This way, your installation always stays up-to-date, even if you pull new changes from the Github repository.

If you prefer, you can also replace the last command with:

```
python3 -m pip install --user -e .
```

... where `-e` stands for `--editable`.

9.2 Building the Documentation

If you make changes to the documentation, you can re-create the HTML pages using Sphinx. You can install it and a few other necessary packages with:

```
python3 -m pip install -r doc/requirements.txt --user
```

To create the HTML pages, use:

```
python3 setup.py build_sphinx
```

The generated files will be available in the directory `build/sphinx/html/`.

It is also possible to automatically check if all links are still valid:

```
python3 setup.py build_sphinx -b linkcheck
```

9.3 Running the Tests

You'll need pytest for that. It can be installed with:

```
python3 -m pip install -r tests/requirements.txt --user
```

To execute the tests, simply run:

```
```
New releases are made using the following steps:

1. Bump version number in `sfs/__init__.py`
2. Update `NEWS.rst`
3. Commit those changes as “Release x.y.z”
4. Create an (annotated) tag with `git tag -a x.y.z`
5. Clear the `dist/` directory
6. Create a source distribution with `python3 setup.py sdist`
7. Create a wheel distribution with `python3 setup.py bdist_wheel`
8. Check that both files have the correct content
9. Upload them to PyPI with `twine upload dist/*`
10. Push the commit and the tag to Github and add release notes containing a link to PyPI and the bullet points from `NEWS.rst`
11. Check that the new release was built correctly on RTD, delete the “stable” version and select the new release as default version

## 10 Version History

**Version 0.4.0 (2018-03-14):**

- Driving functions in time domain for a plane wave, point source, and focused source
- Image source model for a point source in a rectangular room
- DelayedSignal class and `as_delayed_signal()`
- Improvements to the documentation
- Start using Jupyter notebooks for examples in documentation
- Spherical Hankel function as `util.spherical_hn2`
- Use `spherical_jn, spherical_yn` from `scipy.special` instead of `sph_jnyn`
- Generalization of the modal order argument in `mono.source.point_modal()`
- Rename `util.normal_vector()` to `util.normalize_vector()`
- Add parameter `max_order` to NFCHOA driving functions
- Add beta parameter to Kaiser tapering window
- Fix clipping problem of sound field plots with `matplotlib 2.1`
- Fix elevation in `util.cart2sph`
- Fix `tapering.tukey()` for `alpha=1`

**Version 0.3.1 (2016-04-08):**

- Fixed metadata of release

---

60 https://pypi.python.org/pypi/twine
61 https://github.com/sfstoolbox/sfs-python/tags

---

40
Version 0.3.0 (2016-04-08):

• Dirichlet Green’s function for the scattering of a line source at an edge
• Driving functions for the synthesis of various virtual source types with edge-shaped arrays by the equivalent scattering approach
• Driving functions for the synthesis of focused sources by WFS
• Several refactorings, bugfixes and other improvements

Version 0.2.0 (2015-12-11):

• Ability to calculate and plot particle velocity and displacement fields
• Several function name and parameter name changes
• Several refactorings, bugfixes and other improvements

Version 0.1.1 (2015-10-08):

• Fix missing sfs.mono subpackage in PyPI packages

Version 0.1.0 (2015-09-22): Initial release.

References


^52 https://doi.org/10.1007/978-3-642-25743-8
^53 https://doi.org/10.1121/1.382599
^54 https://doi.org/10.1121/1.390983
^55 https://doi.org/10.1007/978-3-642-30933-5
^56 https://doi.org/10.14279/depositonce-4310