SFS Toolbox

Release 0.6.2

SFS Toolbox Developers

2021-06-05

Contents

A Python library for creating numercial simulations of sound field synthesis methods like Wave Field Synthesis (WFS) or Near-Field Compensated Higher Order Ambisonics (NFC-HOA).

Documentation: https://sfs-python.readthedocs.io/

Source code and issue tracker: https://github.com/sfstoolbox/sfs-python/

License: MIT – see the file `LICENSE` for details.

Quick start:

- Install Python 3, NumPy, SciPy and Matplotlib
- `python3 -m pip install sfs --user`
- Check out the examples in the documentation

More information about the underlying theory can be found at https://sfs.readthedocs.io/. There is also a Sound Field Synthesis Toolbox for Octave/Matlab, see https://sfs-matlab.readthedocs.io/.

---

1 Installation

1.1 Requirements

Obviously, you'll need Python[1]. More specifically, you'll need Python 3. NumPy[2] and SciPy[3] are needed for the calculations. If you want to use the provided functions for plotting sound fields, you'll need Matplotlib[4]. However, since all results are provided as plain NumPy[5] arrays, you should also be able to use any plotting library of your choice to visualize the sound fields.

Instead of installing all of the requirements separately, you should probably get a Python distribution that already includes everything, e.g. Anaconda[6].

1.2 Installation

Once you have installed the above-mentioned dependencies, you can use pip[7] to download and install the latest release of the Sound Field Synthesis Toolbox with a single command:

```
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:

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

If you want to install the latest development version of the SFS Toolbox, have a look at *Contributing*.

[1] https://www.python.org/
[2] http://www.numpy.org/
[3] https://www.scipy.org/scipylib/
[4] https://matplotlib.org/
[5] http://www.numpy.org/
[6] https://docs.anaconda.com/anaconda/
[7] https://pip.pypa.io/en/latest/installing/

2 Examples

The following section was generated from /home/docs/checkouts/readthedocs.org/user_builds/sfs-python/checkouts/0.6.2/doc/examples/sound-field-synthesis.ipynb

2.1 Sound Field Synthesis

Illustrates the usage of the SFS toolbox for the simulation of different sound field synthesis methods.

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

```
/home/docs/checkouts/readthedocs.org/user_builds/sfs-python/conda/0.6.2/lib/
↪python3.9/site-packages/traitlets/traitlets.py:3030: FutureWarning: --rc={
↪'figure.dpi': 96} for dict-traits is deprecated in traitlets 5.0. You can pass -
↪-rc <key=value> ... multiple times to add items to a dict.
  warn(
```

[2]:
```
# Simulation parameters
number_of_secondary_sources = 56
frequency = 680  # in Hz
pw_angle = 30  # traveling direction of plane wave in degree
xs = [-2, -1, 0]  # position of virtual point source in m

grid = sfs.util.xyz_grid([-2, 2], [-2, 2], 0, spacing=0.02)
omega = 2 * np.pi * frequency  # angular frequency
npw = sfs.util.direction_vector(np.radians(pw_angle))  # normal vector of plane␣
↪wave
```

Define a helper function for synthesize and plot the sound field from the given driving signals.

[3]:
```
def sound_field(d, selection, secondary_source, array, grid, tapering=True):
    if tapering:
        tapering_window = sfs.tapering.tukey(selection, alpha=0.3)
    else:
        tapering_window = sfs.tapering.none(selection)
    p = sfs.fd.synthesize(d, tapering_window, array, secondary_source, grid=grid)
    sfs.plot2d.amplitude(p, grid, xnorm=[0, 0, 0])
    sfs.plot2d.loudspeakers(array.x, array.n, tapering_window)
```

Circular loudspeaker arrays

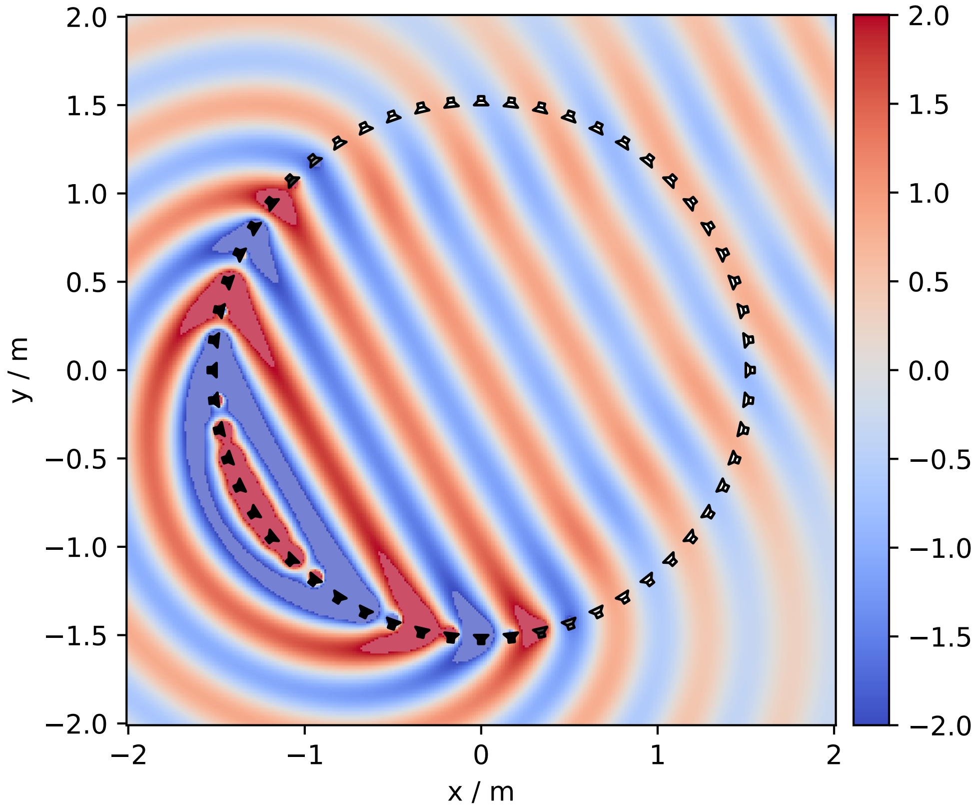
In the following we show different sound field synthesis methods applied to a circular loudspeaker array.

[4]:
```
radius = 1.5  # in m
array = sfs.array.circular(number_of_secondary_sources, radius)
```

Wave Field Synthesis (WFS)

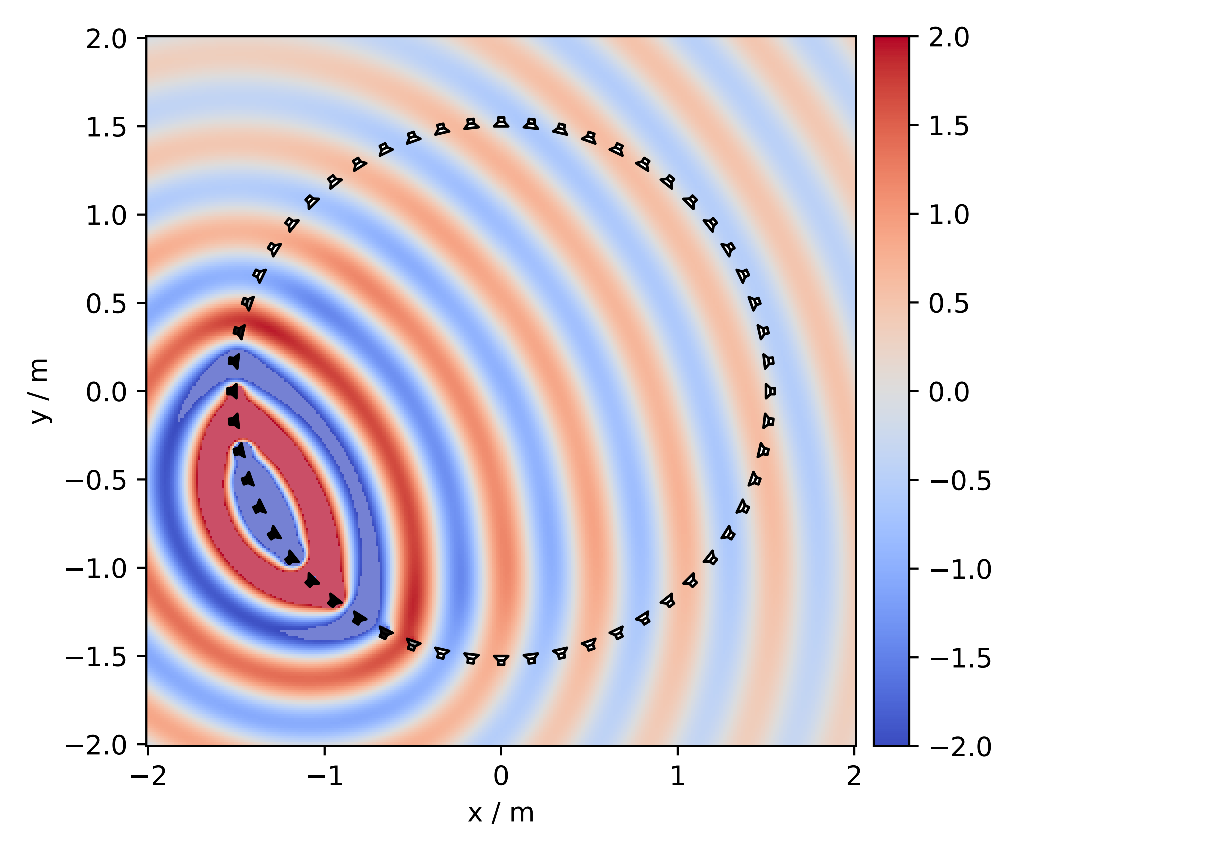
Plane wave

[5]:
```
d, selection, secondary_source = sfs.fd.wfs.plane_25d(omega, array.x, array.n,␣
↪n=npw)
sound_field(d, selection, secondary_source, array, grid)
```



Point source

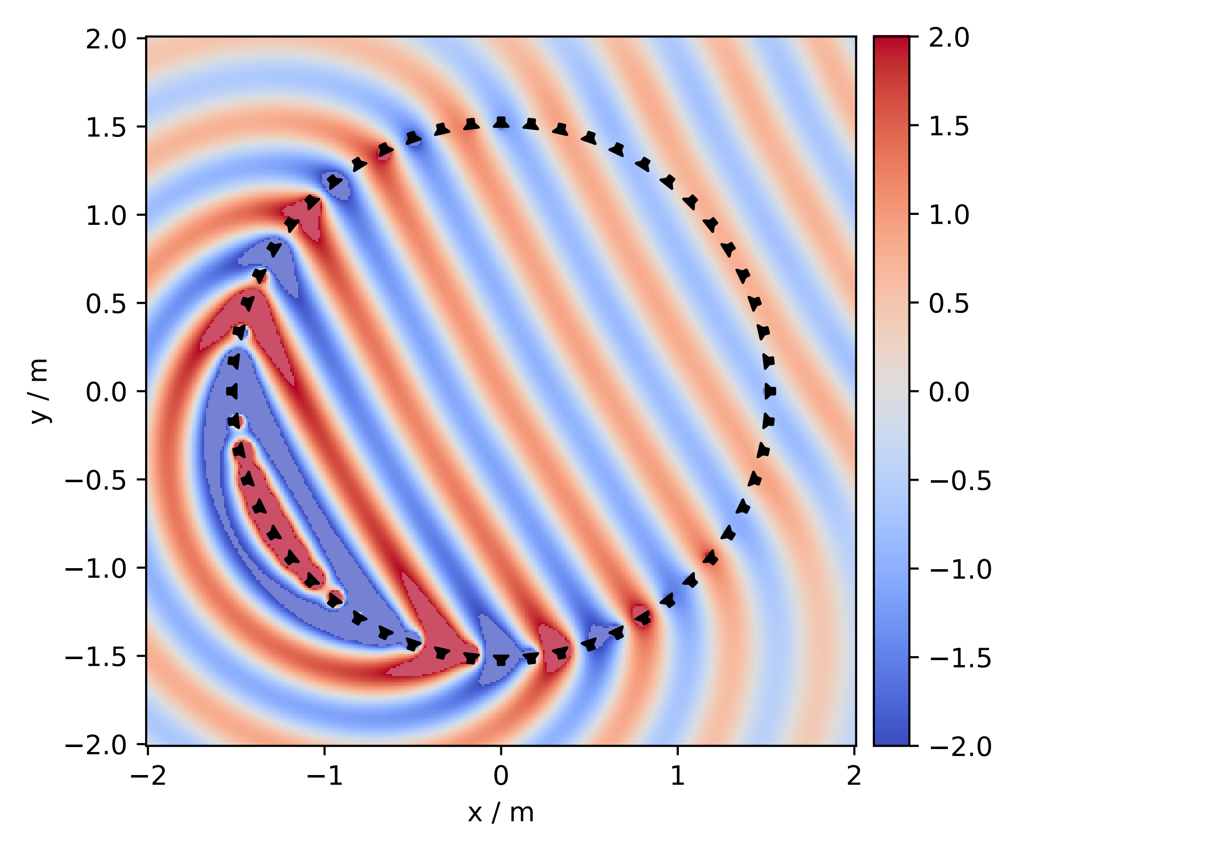
[6]:
```
d, selection, secondary_source = sfs.fd.wfs.point_25d(omega, array.x, array.n, xs)
sound_field(d, selection, secondary_source, array, grid)
```



Near-Field Compensated Higher Order Ambisonics (NFC-HOA)

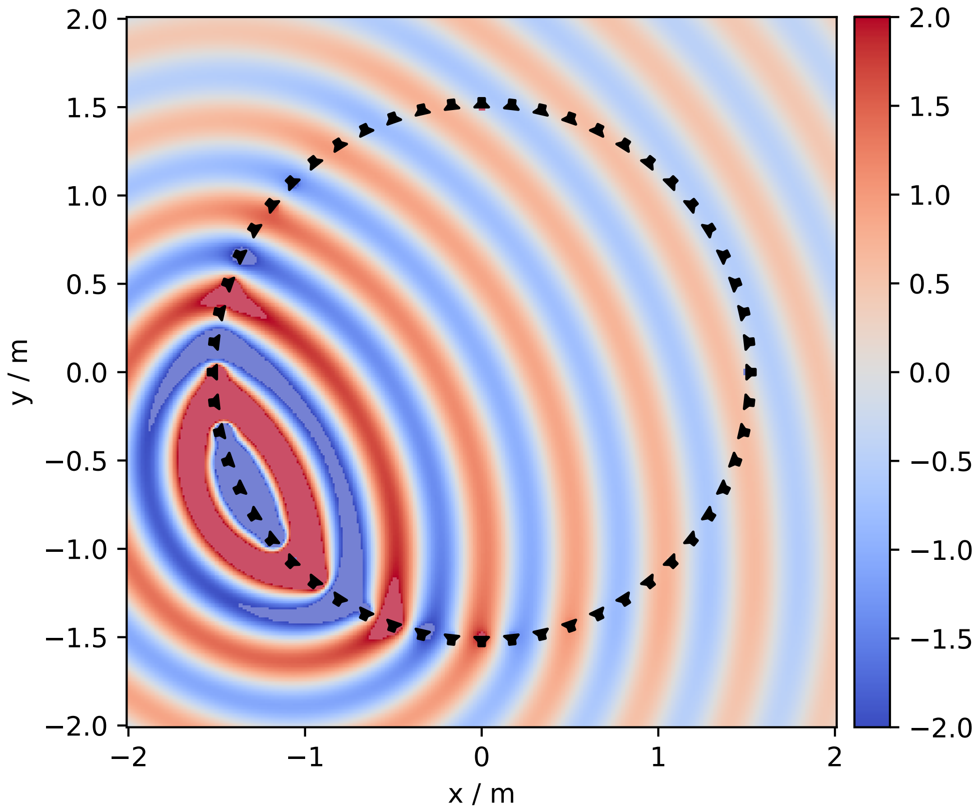
Plane wave

[7]:
```
d, selection, secondary_source = sfs.fd.nfchoa.plane_25d(omega, array.x, radius,
 ↪n=npw)
sound_field(d, selection, secondary_source, array, grid, tapering=False)
```



Point source

[8]:
```
d, selection, secondary_source = sfs.fd.nfchoa.point_25d(omega, array.x, radius,␣
 ↪xs)
sound_field(d, selection, secondary_source, array, grid, tapering=False)
```



Linear loudspeaker array

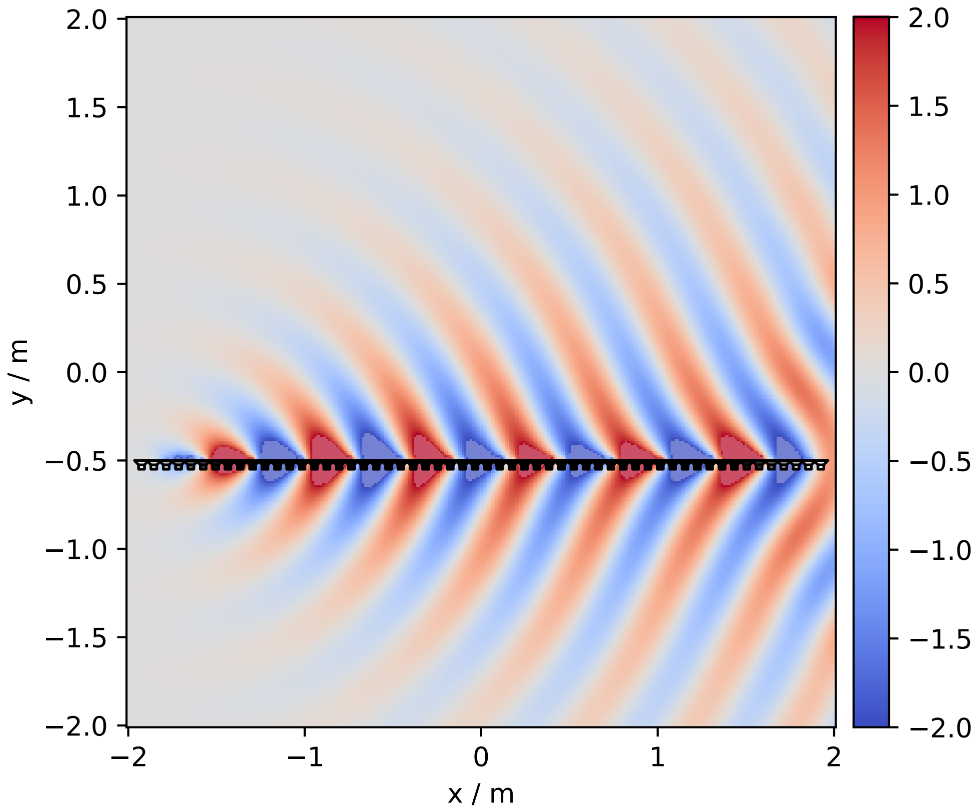
In the following we show different sound field synthesis methods applied to a linear loudspeaker array.

[9]:
```
spacing = 0.07  # in m
array = sfs.array.linear(number_of_secondary_sources, spacing,
                         center=[0, -0.5, 0], orientation=[0, 1, 0])
```

Wave Field Synthesis (WFS)

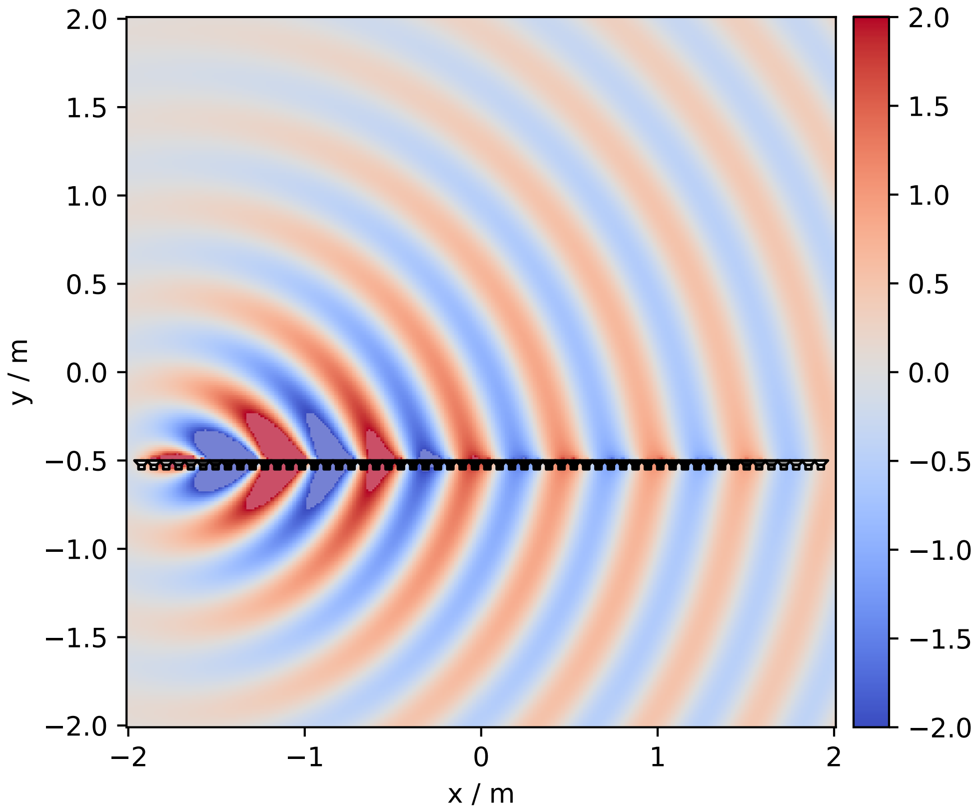
Plane wave

[10]:
```
d, selection, secondary_source = sfs.fd.wfs.plane_25d(omega, array.x, array.n,␣
 ↪npw)
sound_field(d, selection, secondary_source, array, grid)
```



Point source

[11]:
```
d, selection, secondary_source = sfs.fd.wfs.point_25d(omega, array.x, array.n, xs)
sound_field(d, selection, secondary_source, array, grid)
```



. /home/docs/checkouts/readthedocs.org/user_builds/sfs-python/checkouts/0.6.2/doc/examples/sound-field-synthesis.ipynb ends here.

The following section was generated from /home/docs/checkouts/readthedocs.org/user_builds/sfs-python/checkouts/0.6.2/doc/examples/modal-room-acoustics.ipynb ....

2.2 Modal Room Acoustics

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

```
/home/docs/checkouts/readthedocs.org/user_builds/sfs-python/conda/0.6.2/lib/
↪python3.9/site-packages/traitlets/traitlets.py:3030: FutureWarning: --rc={
↪'figure.dpi': 96} for dict-traits is deprecated in traitlets 5.0. You can pass -
↪-rc <key=value> ... multiple times to add items to a dict.
  warn(
```

[2]:
```
%matplotlib inline
```

```
/home/docs/checkouts/readthedocs.org/user_builds/sfs-python/conda/0.6.2/lib/
↪python3.9/site-packages/traitlets/traitlets.py:3030: FutureWarning: --rc={
↪'figure.dpi': 96} for dict-traits is deprecated in traitlets 5.0. You can pass -
↪-rc <key=value> ... multiple times to add items to a dict.
  warn(
```

[3]:
```
x0 = 1, 3, 1.80  # source position
L = 6, 6, 3  # dimensions of room
deltan = 0.01  # absorption factor of walls
N = 20  # maximum order of modes
```

You can experiment with different combinations of modes:

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

Sound Field for One Frequency

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

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

```
[7]: p = sfs.fd.source.point_modal(omega, x0, grid, L, N=N, deltan=deltan)
```

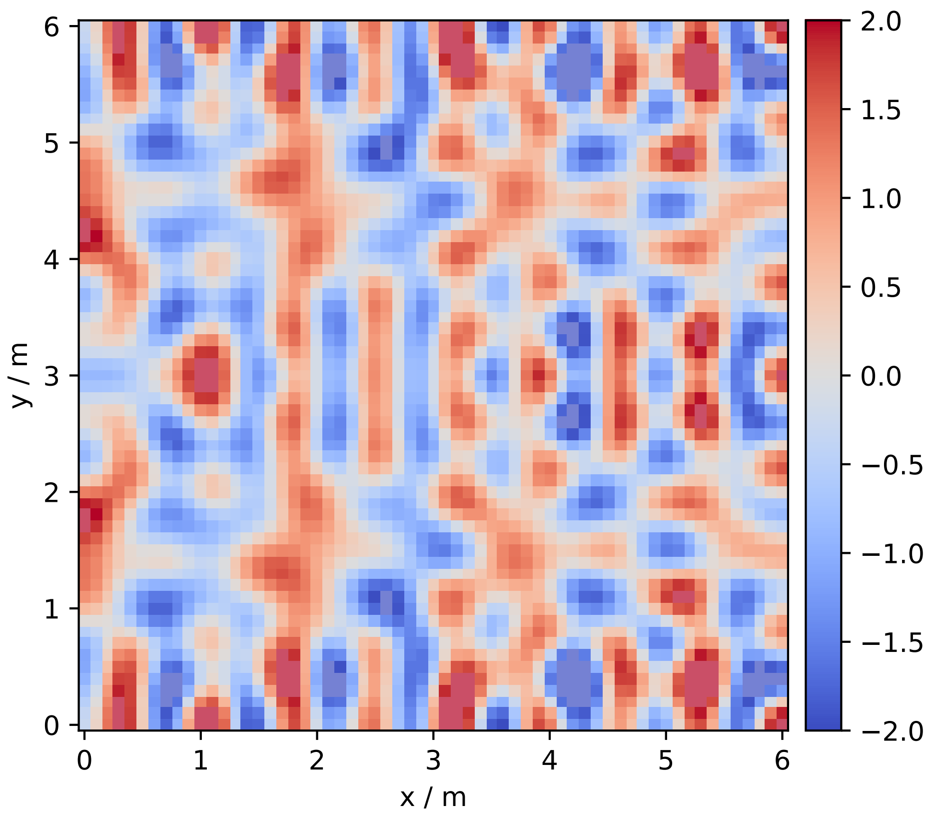
For now, we apply an arbitrary scaling factor to make the plot look good

TODO: proper normalization

```
[8]: p *= 0.05
```

```
[9]: sfs.plot2d.amplitude(p, grid);
```

```
[9]: <matplotlib.image.AxesImage at 0x7fc1a4c03a00>
```



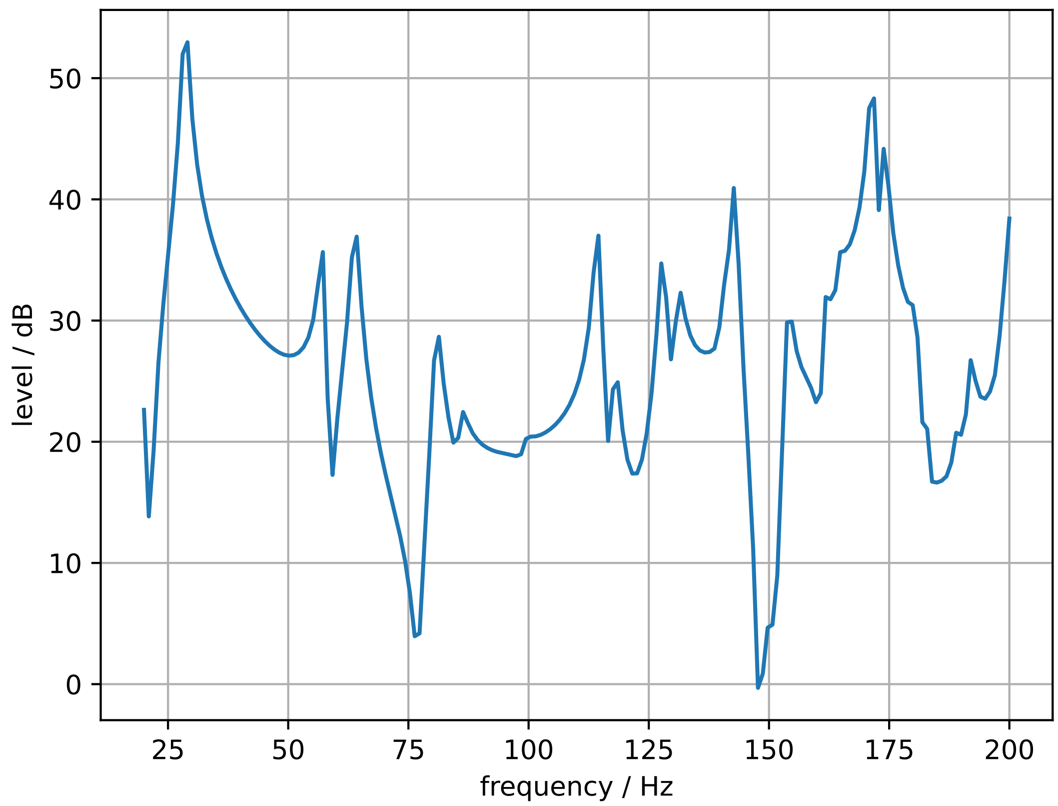
Frequency Response at One Point

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

receiver = 1, 1, 1.8

p = [sfs.fd.source.point_modal(om, x0, 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()
```



.. /home/docs/checkouts/readthedocs.org/user_builds/sfs-python/checkouts/0.6.2/doc/examples/modal-room-acoustics.ipynb ends here.

The following section was generated from /home/docs/checkouts/readthedocs.org/user_builds/sfs-python/checkouts/0.6.2/doc/examples/mirror-image-source-model.ipynb ....

2.3 Mirror Image Sources and the Sound Field in a Rectangular Room

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

```
/home/docs/checkouts/readthedocs.org/user_builds/sfs-python/conda/0.6.2/lib/
↪python3.9/site-packages/traitlets/traitlets.py:3030: FutureWarning: --rc={
↪'figure.dpi': 96} for dict-traits is deprecated in traitlets 5.0. You can pass -
↪-rc <key=value> ... multiple times to add items to a dict.
```

(continues on next page)

(continued from previous page)

```
warn(
```

[2]:
```
L = 2, 2.7, 3  # room dimensions
x0 = 1.2, 1.7, 1.5  # source position
max_order = 2  # maximum order of image sources
coeffs = .8, .8, .6, .6, .7, .7  # wall reflection coefficients
```

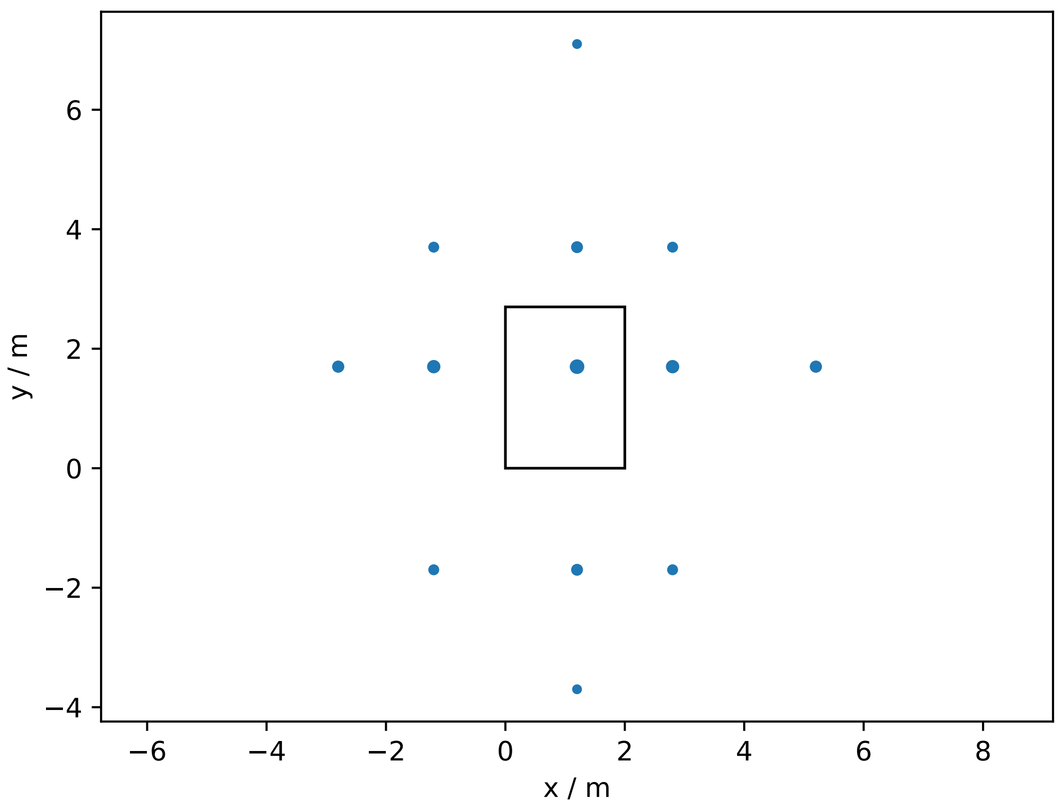
2D Mirror Image Sources

[3]:
```
xs, wall_count = sfs.util.image_sources_for_box(x0[0:2], L[0:2], max_order)
source_strength = np.prod(coeffs[0:4]**wall_count, axis=1)
```

[4]:
```
from matplotlib.patches import Rectangle
```

[5]:
```
fig, ax = plt.subplots()
ax.scatter(*xs.T, source_strength * 20)
ax.add_patch(Rectangle((0, 0), L[0], L[1], fill=False))
ax.set_xlabel('x / m')
ax.set_ylabel('y / m')
ax.axis('equal');
```

[5]:
```
(-3.1999999999999997, 5.6000000000000005, -4.24, 7.640000000000001)
```



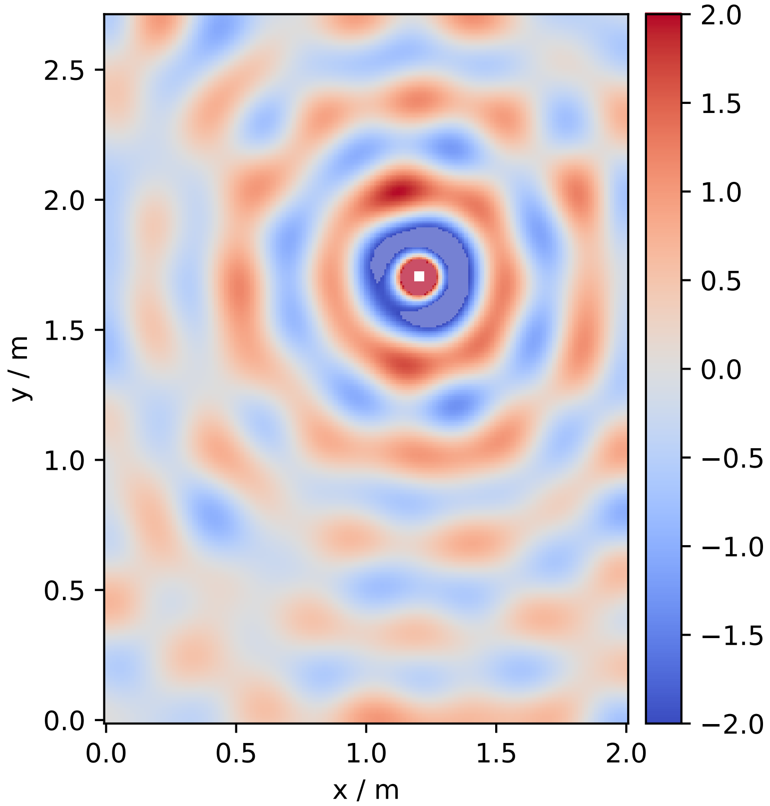
Monochromatic Sound Field

[6]:
```
omega = 2 * np.pi * 1000  # angular frequency
```

[7]:
```
grid = sfs.util.xyz_grid([0, L[0]], [0, L[1]], 1.5, spacing=0.02)
P = sfs.fd.source.point_image_sources(omega, x0, grid, L,
                                      max_order=max_order, coeffs=coeffs)
```

[8]:
```
sfs.plot2d.amplitude(P, grid, xnorm=[L[0]/2, L[1]/2, L[2]/2]);
```

[8]:
```
<matplotlib.image.AxesImage at 0x7f6ac8adb190>
```

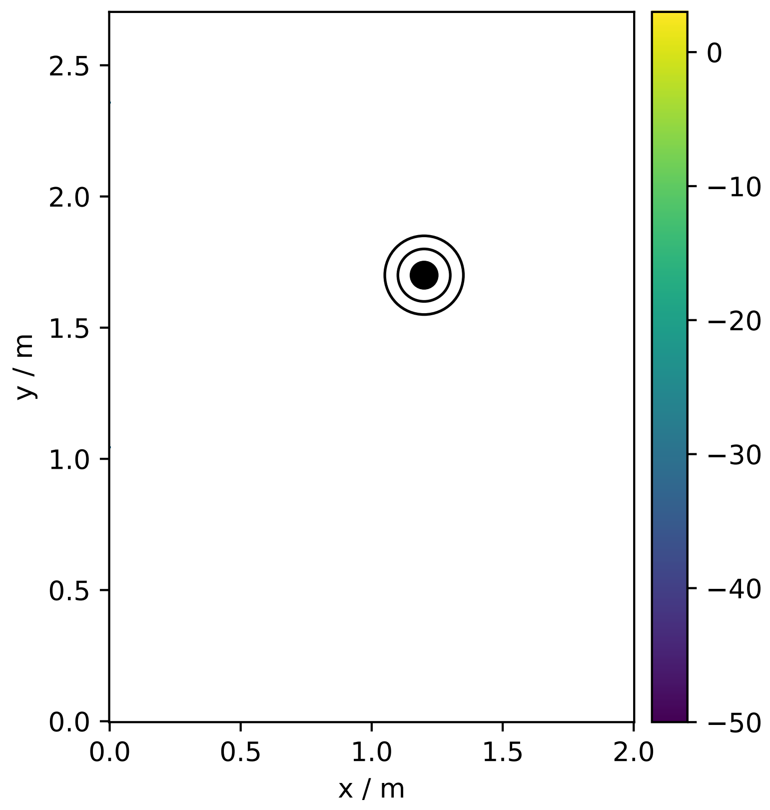


Spatio-temporal Impulse Response

[9]:
```
fs = 44100  # sample rate
signal = [1, 0, 0], fs
```

[10]:
```
grid = sfs.util.xyz_grid([0, L[0]], [0, L[1]], 1.5, spacing=0.005)
p = sfs.td.source.point_image_sources(x0, signal, 0.004, grid, L, max_order,
                                      coeffs=coeffs)
```

[11]:
```
sfs.plot2d.level(p, grid)
sfs.plot2d.virtualsource(x0)
```



.. /home/docs/checkouts/readthedocs.org/user_builds/sfs-python/checkouts/0.6.2/doc/examples/mirror-image-source-model.ipynb ends here.

The following section was generated from /home/docs/checkouts/readthedocs.org/user_builds/sfs-python/checkouts/0.6.2/doc/examples/animations-pulsating-sphere.ipynb ....

2.4 Animations of a Pulsating Sphere

[1]:
```
import sfs
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import HTML
```

```
/home/docs/checkouts/readthedocs.org/user_builds/sfs-python/conda/0.6.2/lib/
↪python3.9/site-packages/traitlets/traitlets.py:3030: FutureWarning: --rc={
↪'figure.dpi': 96} for dict-traits is deprecated in traitlets 5.0. You can pass -
↪-rc <key=value> ... multiple times to add items to a dict.
  warn(
```

In this example, the sound field of a pulsating sphere is visualized. Different acoustic variables, such as sound pressure, particle velocity, and particle displacement, are simulated. The first two quantities are computed with

- *sfs.fd.source.pulsating_sphere()* and
- *sfs.fd.source.pulsating_sphere_velocity()*

while the last one can be obtained by using

- *sfs.fd.displacement()*

which converts the particle velocity into displacement.

A couple of additional functions are implemented in

- animations_pulsating_sphere.py

in order to help creating animating pictures, which is fun!

[2]:
```
import animations_pulsating_sphere as animation
```

[3]:
```
# Pulsating sphere
center = [0, 0, 0]
radius = 0.25
amplitude = 0.05
f = 1000  # frequency
omega = 2 * np.pi * f  # angular frequency

# Axis limits
figsize = (6, 6)
xmin, xmax = -1, 1
ymin, ymax = -1, 1

# Animations
frames = 20  # frames per period
```

Particle Displacement

[4]:
```
grid = sfs.util.xyz_grid([xmin, xmax], [ymin, ymax], 0, spacing=0.025)
ani = animation.particle_displacement(
        omega, center, radius, amplitude, grid, frames, figsize, c='Gray')
plt.close()
HTML(ani.to_jshtml())
```

[4]:
```
<IPython.core.display.HTML object>
```

Click the arrow button to start the animation. `to_jshtml()` allows you to play with the animation, e.g. speed up/down the animation (+/- button). Try to reverse the playback by clicking the left arrow. You'll see a sound *sink*.

You can also show the animation by using `to_html5_video()`. See the documentation[8] for more detail.

Of course, different types of grid can be chosen. Below is the particle animation using the same parameters but with a hexagonal grid[9].

[5]:
```
def hex_grid(xlim, ylim, hex_edge, align='horizontal'):
    if align is 'vertical':
        umin, umax = ylim
        vmin, vmax = xlim
    else:
        umin, umax = xlim
        vmin, vmax = ylim
    du = np.sqrt(3) * hex_edge
    dv = 1.5 * hex_edge
    num_u = int((umax - umin) / du)
    num_v = int((vmax - vmin) / dv)
```

(continues on next page)

[8] https://matplotlib.org/api/_as_gen/matplotlib.animation.ArtistAnimation.html#matplotlib.animation.ArtistAnimation.to_html5_video

[9] https://www.redblobgames.com/grids/hexagons/

(continued from previous page)

```
    u, v = np.meshgrid(np.linspace(umin, umax, num_u),
                       np.linspace(vmin, vmax, num_v))
    u[::2] += 0.5 * du

    if align is 'vertical':
        grid = v, u, 0
    elif align is 'horizontal':
        grid = u, v, 0
    return  grid
```

```
<>:2: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:16: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:18: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:2: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:16: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:18: SyntaxWarning: "is" with a literal. Did you mean "=="?
<ipython-input-1-17990c1f260e>:2: SyntaxWarning: "is" with a literal. Did you␣
↪mean "=="?
  if align is 'vertical':
<ipython-input-1-17990c1f260e>:16: SyntaxWarning: "is" with a literal. Did you␣
↪mean "=="?
  if align is 'vertical':
<ipython-input-1-17990c1f260e>:18: SyntaxWarning: "is" with a literal. Did you␣
↪mean "=="?
  elif align is 'horizontal':
```

[6]:
```
grid = hex_grid([xmin, xmax], [ymin, ymax], 0.0125, 'vertical')
ani = animation.particle_displacement(
        omega, center, radius, amplitude, grid, frames, figsize, c='Gray')
plt.close()
HTML(ani.to_jshtml())
```

```
/home/docs/checkouts/readthedocs.org/user_builds/sfs-python/checkouts/0.6.2/doc/
↪examples/animations_pulsating_sphere.py:18: VisibleDeprecationWarning: Creating␣
↪an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
↪tuples-or ndarrays with different lengths or shapes) is deprecated. If you␣
↪meant to do this, you must specify 'dtype=object' when creating the ndarray.
  scat = sfs.plot2d.particles(grid + displacement, **kwargs)
/home/docs/checkouts/readthedocs.org/user_builds/sfs-python/checkouts/0.6.2/doc/
↪examples/animations_pulsating_sphere.py:21: VisibleDeprecationWarning: Creating␣
↪an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
↪tuples-or ndarrays with different lengths or shapes) is deprecated. If you␣
↪meant to do this, you must specify 'dtype=object' when creating the ndarray.
  position = (grid + displacement * phasor**i).apply(np.real)
/home/docs/checkouts/readthedocs.org/user_builds/sfs-python/checkouts/0.6.2/doc/
↪examples/animations_pulsating_sphere.py:21: VisibleDeprecationWarning: Creating␣
↪an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
↪tuples-or ndarrays with different lengths or shapes) is deprecated. If you␣
↪meant to do this, you must specify 'dtype=object' when creating the ndarray.
  position = (grid + displacement * phasor**i).apply(np.real)
```

[6]:
```
<IPython.core.display.HTML object>
```

Another one using a random grid.

```
[7]: grid = [np.random.uniform(xmin, xmax, 4000),
             np.random.uniform(ymin, ymax, 4000), 0]
     ani = animation.particle_displacement(
             omega, center, radius, amplitude, grid, frames, figsize, c='Gray')
     plt.close()
     HTML(ani.to_jshtml())
```

```
/home/docs/checkouts/readthedocs.org/user_builds/sfs-python/checkouts/0.6.2/doc/
↪examples/animations_pulsating_sphere.py:18: VisibleDeprecationWarning: Creating␣
↪an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
↪tuples-or ndarrays with different lengths or shapes) is deprecated. If you␣
↪meant to do this, you must specify 'dtype=object' when creating the ndarray.
  scat = sfs.plot2d.particles(grid + displacement, **kwargs)
/home/docs/checkouts/readthedocs.org/user_builds/sfs-python/checkouts/0.6.2/doc/
↪examples/animations_pulsating_sphere.py:21: VisibleDeprecationWarning: Creating␣
↪an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
↪tuples-or ndarrays with different lengths or shapes) is deprecated. If you␣
↪meant to do this, you must specify 'dtype=object' when creating the ndarray.
  position = (grid + displacement * phasor**i).apply(np.real)
```

```
[7]: <IPython.core.display.HTML object>
```

Each grid has its strengths and weaknesses. Please refer to the on-line discussion[10].

Particle Velocity

```
[8]: amplitude = 1e-3
     grid = sfs.util.xyz_grid([xmin, xmax], [ymin, ymax], 0, spacing=0.04)
     ani = animation.particle_velocity(
             omega, center, radius, amplitude, grid, frames, figsize)
     plt.close()
     HTML(ani.to_jshtml())
```

```
[8]: <IPython.core.display.HTML object>
```

Please notice that the amplitude of the pulsating motion is adjusted so that the arrows are neither too short nor too long. This kind of compromise is inevitable since

$$\text{(particle velocity)} = \mathrm{i}\omega \times \text{(amplitude)},$$

thus the absolute value of particle velocity is usually much larger than that of amplitude. It should be also kept in mind that the hole in the middle does not visualizes the exact motion of the pulsating sphere. According to the above equation, the actual amplitude should be much smaller than the arrow lengths. The changing rate of its size is also two times higher than the original frequency.

[10] https://github.com/sfstoolbox/sfs-python/pull/69#issuecomment-468405536

Sound Pressure

[9]:
```
amplitude = 0.05
impedance_pw = sfs.default.rho0 * sfs.default.c
max_pressure = omega * impedance_pw * amplitude

grid = sfs.util.xyz_grid([xmin, xmax], [ymin, ymax], 0, spacing=0.005)
ani = animation.sound_pressure(
        omega, center, radius, amplitude, grid, frames, pulsate=True,
        figsize=figsize, vmin=-max_pressure, vmax=max_pressure)
plt.close()
HTML(ani.to_jshtml())
```

[9]:
```
<IPython.core.display.HTML object>
```

Notice that the sound pressure exceeds the atmospheric pressure ($\approx 10^5$ Pa), which of course makes no sense. This is due to the large amplitude (50 mm) of the pulsating motion. It was chosen to better visualize the particle movements in the earlier animations.

For 1 kHz, the amplitude corresponding to a moderate sound pressure, let say 1 Pa, is in the order of micrometer. As it is very small compared to the corresponding wavelength (0.343 m), the movement of the particles and the spatial structure of the sound field cannot be observed simultaneously. Furthermore, at high frequencies, the sound pressure for a given particle displacement scales with the frequency. The smaller wavelength (higher frequency) we choose, it is more likely to end up with a prohibitively high sound pressure.

In the following examples, the amplitude is set to a realistic value 1 μm. Notice that the pulsating motion of the sphere is no more visible.

[10]:
```
amplitude = 1e-6
impedance_pw = sfs.default.rho0 * sfs.default.c
max_pressure = omega * impedance_pw * amplitude

grid = sfs.util.xyz_grid([xmin, xmax], [ymin, ymax], 0, spacing=0.005)
ani = animation.sound_pressure(
        omega, center, radius, amplitude, grid, frames, pulsate=True,
        figsize=figsize, vmin=-max_pressure, vmax=max_pressure)
plt.close()
HTML(ani.to_jshtml())
```

[10]:
```
<IPython.core.display.HTML object>
```

Let's zoom in closer to the boundary of the sphere.

[11]:
```
L = 10 * amplitude
xmin_zoom, xmax_zoom = radius - L, radius + L
ymin_zoom, ymax_zoom = -L, L
```

[12]:
```
grid = sfs.util.xyz_grid([xmin_zoom, xmax_zoom], [ymin_zoom, ymax_zoom], 0,␣
 ↪spacing=L / 100)
ani = animation.sound_pressure(
        omega, center, radius, amplitude, grid, frames, pulsate=True,
        figsize=figsize, vmin=-max_pressure, vmax=max_pressure)
plt.close()
HTML(ani.to_jshtml())
```

[12]:
```
<IPython.core.display.HTML object>
```

This shows how the vibrating motion of the sphere (left half) changes the sound pressure of the surrounding air (right half). Notice that the sound pressure increases/decreases (more red/blue) when the surface accelerates/decelerates.

.... /home/docs/checkouts/readthedocs.org/user_builds/sfs-python/checkouts/0.6.2/doc/examples/animations-pulsating-sphere.ipynb ends here.

2.5 Example Python Scripts

Various example scripts are located in the directory `doc/examples/`, e.g.

- `examples/horizontal_plane_arrays.py`: Computes the sound fields for various techniques, virtual sources and loudspeaker array configurations
- `examples/animations_pulsating_sphere.py`: Creates animations of a pulsating sphere, see also *the corresponding Jupyter notebook*
- `examples/soundfigures.py`: Illustrates the synthesis of sound figures with Wave Field Synthesis

3 API Documentation

Sound Field Synthesis Toolbox.

https://sfs-python.readthedocs.io/

Submodules

fd	Submodules for monochromatic sound fields.
td	Submodules for broadband sound fields.
array	Compute positions of various secondary source distributions.
tapering	Weights (tapering) for the driving function.
plot2d	2D plots of sound fields etc.
plot3d	3D plots of sound fields etc.
util	Various utility functions.

3.1 sfs.fd

Submodules for monochromatic sound fields.

source	Compute the sound field generated by a sound source.
wfs	Compute WFS driving functions.
nfchoa	Compute NFC-HOA driving functions.
sdm	Compute SDM driving functions.
esa	Compute ESA driving functions for various systems.

sfs.fd.source

Compute the sound field generated by a sound source.

```
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.default.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)
```

Functions

`line`(omega, xo, grid, *[, c])	Line source parallel to the z-axis.
`line_bandlimited`(omega, xo, grid, *[, ...])	Spatially bandlimited (modal) line source parallel to the z-axis.
`line_dipole`(omega, xo, no, grid, *[, c])	Line source with dipole characteristics parallel to the z-axis.
`line_dirichlet_edge`(omega, xo, grid, *[, ...])	Line source scattered at an edge with Dirichlet boundary conditions.
`line_velocity`(omega, xo, grid, *[, c, rhoo])	Velocity of line source parallel to the z-axis.
`plane`(omega, xo, no, grid, *[, c])	Plane wave.
`plane_averaged_intensity`(omega, xo, no, grid, *)	Averaged intensity of a plane wave.
`plane_velocity`(omega, xo, no, grid, *[, c, rhoo])	Velocity of a plane wave.
`point`(omega, xo, grid, *[, c])	Sound pressure of a point source.
`point_averaged_intensity`(omega, xo, grid, *)	Velocity of a point source.
`point_dipole`(omega, xo, no, grid, *[, c])	Point source with dipole characteristics.
`point_image_sources`(omega, xo, grid, L, *, ...)	Point source in a rectangular room using the mirror image source model.
`point_modal`(omega, xo, grid, L, *[, N, ...])	Point source in a rectangular room using a modal room model.
`point_modal_velocity`(omega, xo, grid, L, *)	Velocity of point source in a rectangular room using a modal room model.
`point_velocity`(omega, xo, grid, *[, c, rhoo])	Particle velocity of a point source.
`pulsating_sphere`(omega, center, radius, ...)	Sound pressure of a pulsating sphere.
`pulsating_sphere_velocity`(omega, center, ...)	Particle velocity of a pulsating sphere.

`sfs.fd.source.`**`point`**`(`*omega, xo, grid, *, c=None*`)`

Sound pressure of a point source.

Parameters

- **omega** (*float*) – Frequency of source.
- **xo** (*(3,) array_like*) – Position of source.
- **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.

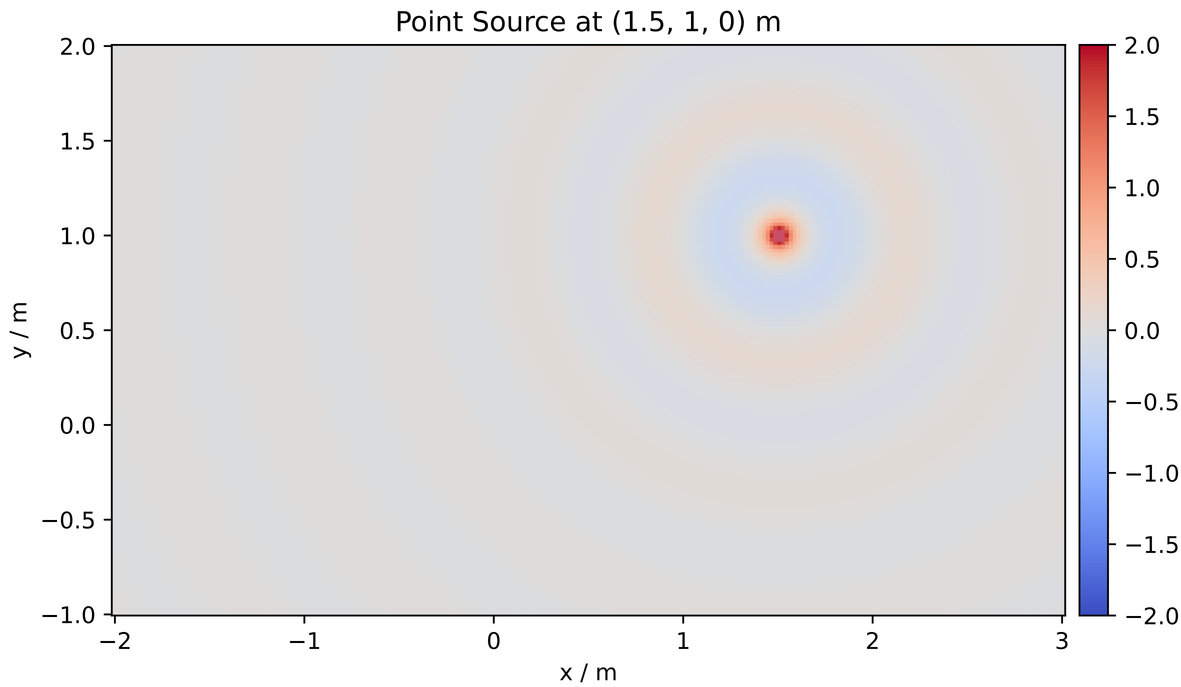
Returns *numpy.ndarray* – Sound pressure at positions given by *grid*.

Notes

$$G(\mathbf{x} - \mathbf{x}_0, \omega) = \frac{1}{4\pi} \frac{\mathrm{e}^{-\mathrm{i}\frac{\omega}{c}|\mathbf{x}-\mathbf{x}_0|}}{|\mathbf{x} - \mathbf{x}_0|}$$

Examples

```
p = sfs.fd.source.point(omega, x0, grid)
sfs.plot2d.amplitude(p, grid)
plt.title("Point Source at {} m".format(x0))
```

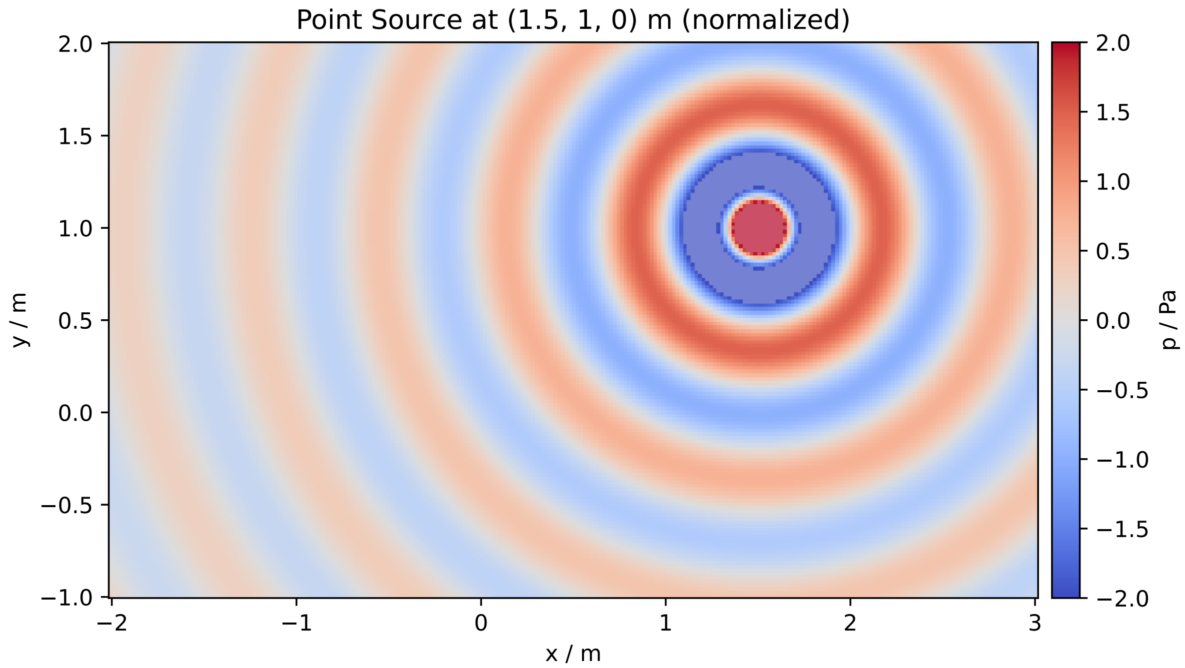


Normalization …

```
sfs.plot2d.amplitude(p * normalization_point, grid,
                     colorbar_kwargs=dict(label="p / Pa"))
plt.title("Point Source at {} m (normalized)".format(x0))
```

`sfs.fd.source.`**`point_velocity`**`(`*omega, xo, grid, *, c=None, rhoo=None*`)`

Particle velocity of a point source.



Parameters

- **omega** (*float*) – Frequency of source.
- **xo** (*(3,) array_like*) – Position of source.
- **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.
- **rhoo** (*float, optional*) – Static density of air.

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

Examples

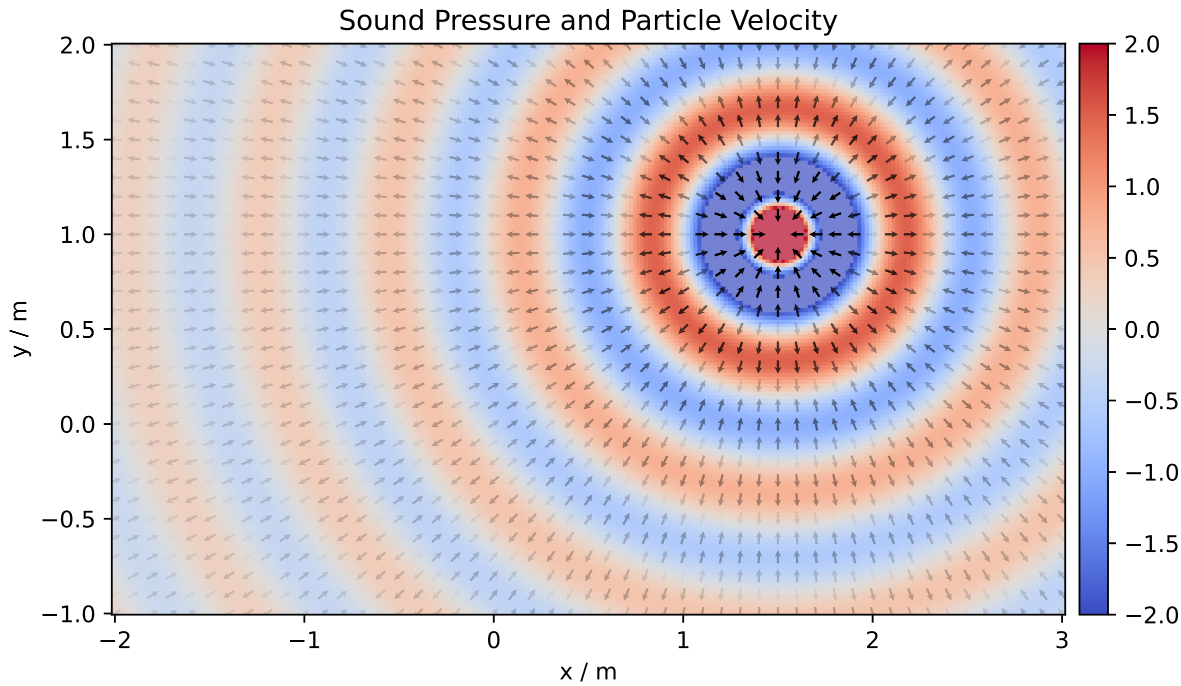
The particle velocity can be plotted on top of the sound pressure:

```
v = sfs.fd.source.point_velocity(omega, x0, vgrid)
sfs.plot2d.amplitude(p * normalization_point, grid)
sfs.plot2d.vectors(v * normalization_point, vgrid)
plt.title("Sound Pressure and Particle Velocity")
```

`sfs.fd.source.`**`point_averaged_intensity`**(*omega, xo, grid, *, c=None, rhoo=None*)
Velocity of a point source.

Parameters

- **omega** (*float*) – Frequency of source.
- **xo** (*(3,) array_like*) – Position of source.
- **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.
- **rhoo** (*float, optional*) – Static density of air.



Returns *XyzComponents* – Averaged intensity at positions given by *grid*.

`sfs.fd.source.`**`point_dipole`**(*omega, xo, no, grid, *, c=None*)

Point source with dipole characteristics.

Parameters

- **omega** (*float*) – Frequency of source.
- **xo** (*(3,) array_like*) – Position of source.
- **no** (*(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

$$G(\mathbf{x}-\mathbf{x}_0,\omega)=\frac{1}{4\pi}\left(\mathrm{i}\frac{\omega}{c}+\frac{1}{|\mathbf{x}-\mathbf{x}_0|}\right)\frac{\langle\mathbf{x}-\mathbf{x}_0,\mathbf{n}_\mathrm{s}\rangle}{|\mathbf{x}-\mathbf{x}_0|^2}\,\mathrm{e}^{-\mathrm{i}\frac{\omega}{c}|\mathbf{x}-\mathbf{x}_0}$$

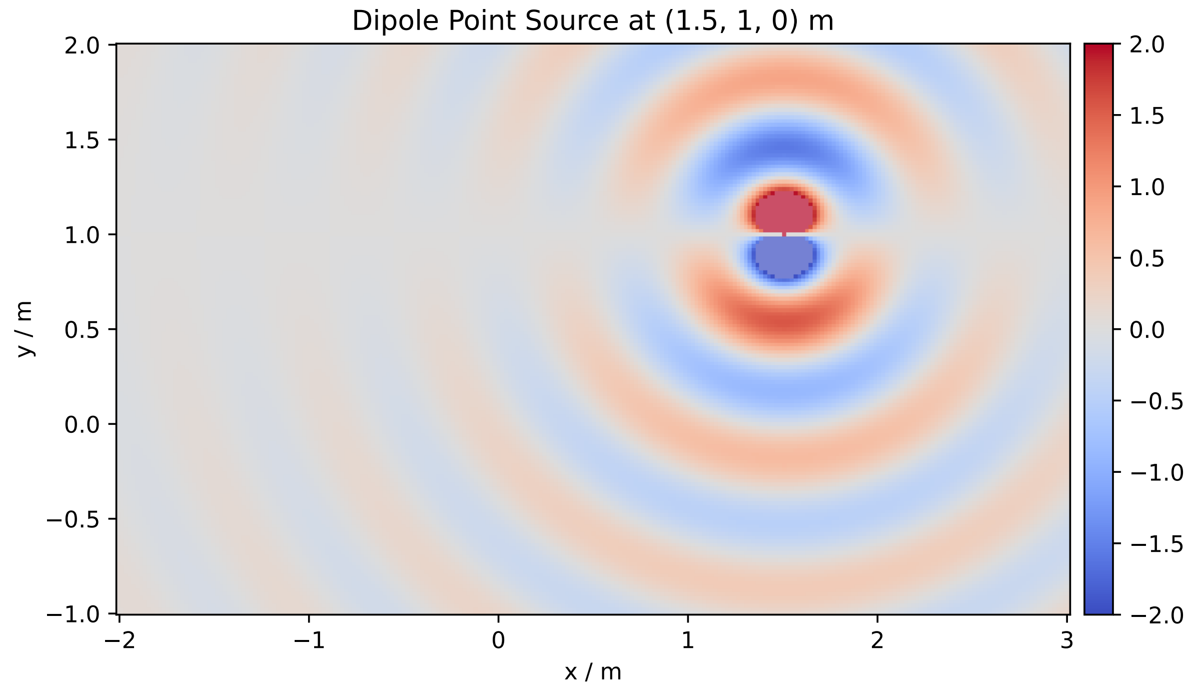
Examples

```
n0 = 0, 1, 0
p = sfs.fd.source.point_dipole(omega, x0, n0, grid)
sfs.plot2d.amplitude(p, grid)
plt.title("Dipole Point Source at {} m".format(x0))
```

`sfs.fd.source.`**`point_modal`**(*omega, xo, grid, L, *, N=None, deltan=o, c=None*)

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

Parameters



- **omega** (*float*) – Frequency of source.
- **xo** (*(3,) array_like*) – Position of source.
- **grid** (*triple of array_like*) – The grid that is used for the sound field calculations. See `sfs.util.xyz_grid()`.
- **L** (*(3,) array_like*) – Dimensionons 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.
- **deltan** (*float, optional*) – Absorption coefficient of the walls.
- **c** (*float, optional*) – Speed of sound.

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

`sfs.fd.source.`**`point_modal_velocity`**(*omega, xo, grid, L, *, N=None, deltan=0, c=None*)

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

Parameters

- **omega** (*float*) – Frequency of source.
- **xo** (*(3,) array_like*) – Position of source.
- **grid** (*triple of array_like*) – The grid that is used for the sound field calculations. See `sfs.util.xyz_grid()`.
- **L** (*(3,) array_like*) – Dimensionons 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.
- **deltan** (*float, optional*) – Absorption coefficient of the walls.
- **c** (*float, optional*) – Speed of sound.

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

`sfs.fd.source.point_image_sources(omega, xo, 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.
- **xo** (*(3,) array_like*) – Position of source.
- **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* – Sound pressure at positions given by *grid*.

`sfs.fd.source.line(omega, xo, grid, *, c=None)`

Line source parallel to the z-axis.

Parameters

- **omega** (*float*) – Frequency of source.
- **xo** (*(3,) array_like*) – Position of source. Note: third component of xo is ignored.
- **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

$$G(\mathbf{x} - \mathbf{x}_0, \omega) = -\frac{\mathrm{i}}{4} H_0^{(2)}\left(\frac{\omega}{c}|\mathbf{x} - \mathbf{x}_0|\right)$$

Examples

```
p = sfs.fd.source.line(omega, x0, grid)
sfs.plot2d.amplitude(p, grid)
plt.title("Line Source at {} m".format(x0[:2]))
```

Normalization . . .

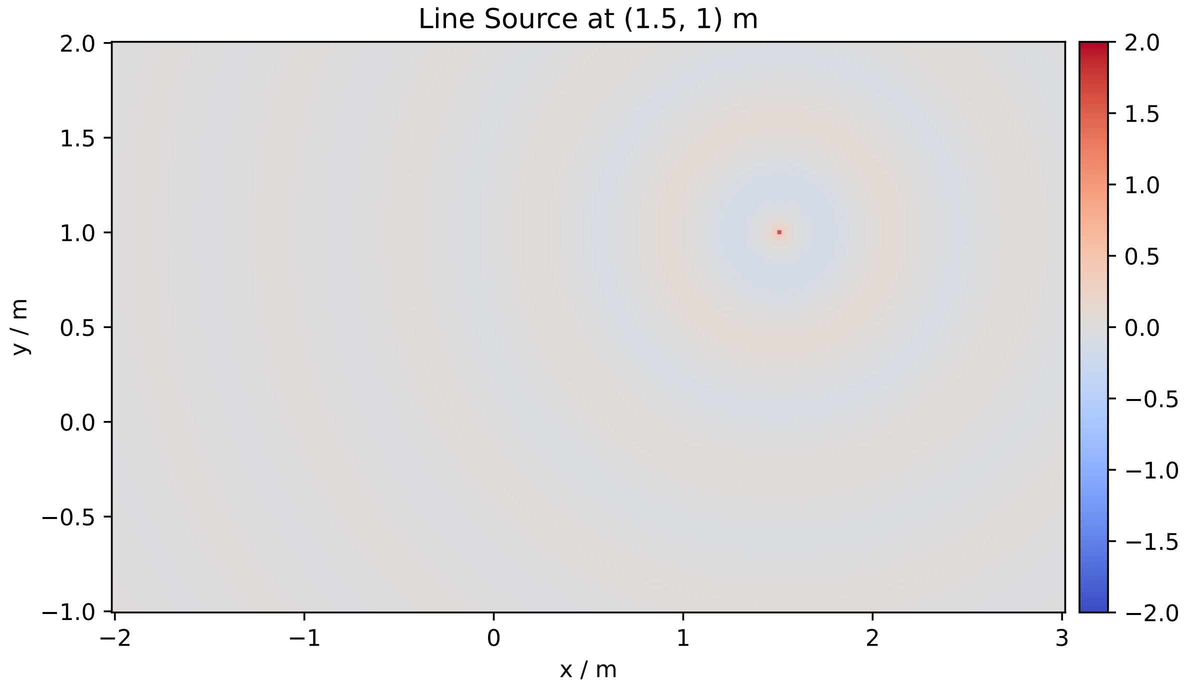
```
sfs.plot2d.amplitude(p * normalization_line, grid,
                     colorbar_kwargs=dict(label="p / Pa"))
plt.title("Line Source at {} m (normalized)".format(x0[:2]))
```

`sfs.fd.source.line_velocity(omega, xo, grid, *, c=None, rhoo=None)`

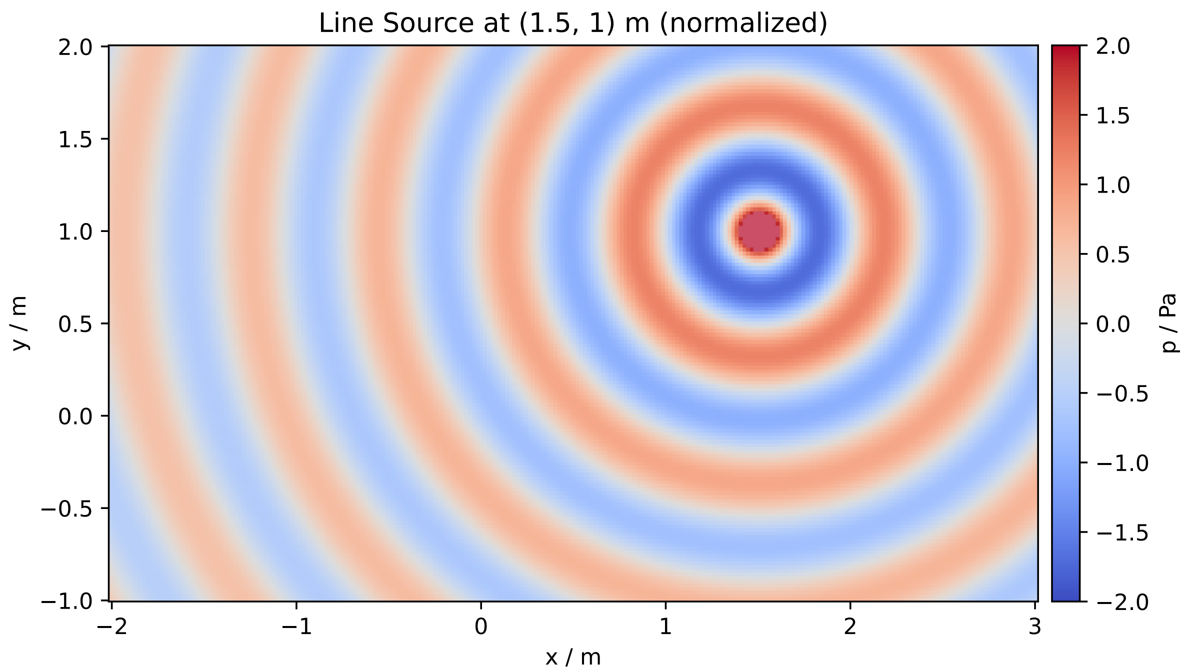
Velocity of line source parallel to the z-axis.

Parameters

- **omega** (*float*) – Frequency of source.



Line Source at (1.5, 1) m
y / m
x / m
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−2
−1
0
1
2
3
−1.5
−2.0



Line Source at (1.5, 1) m (normalized)
y / m
x / m
p / Pa
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−2
−1
0
1
2
3
−1.5
−2.0

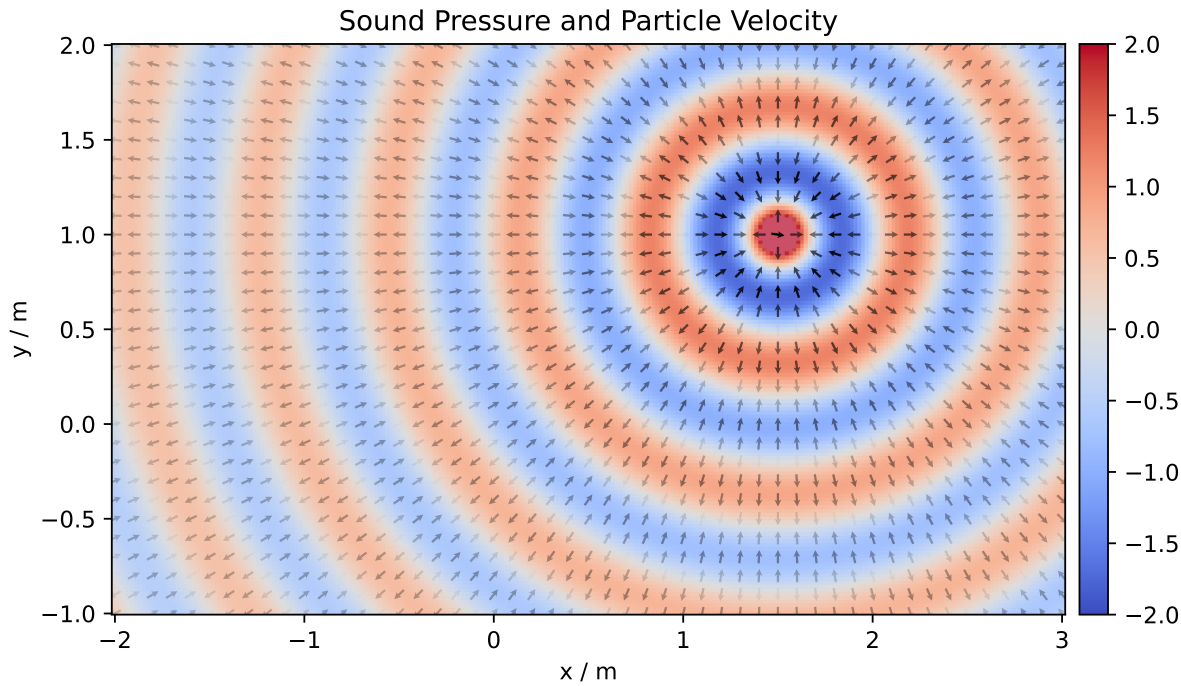
- **xo** (*(3,) array_like*) – Position of source. Note: third component of xo is ignored.
- **grid** (*triple of array_like*) – The grid that is used for the sound field calculations. See *sfs.util.xyz_grid()*.
- **c** (*float, optional*) – Speed of sound.

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

Examples

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

```
v = sfs.fd.source.line_velocity(omega, x0, vgrid)
sfs.plot2d.amplitude(p * normalization_line, grid)
sfs.plot2d.vectors(v * normalization_line, vgrid)
plt.title("Sound Pressure and Particle Velocity")
```



sfs.fd.source.**line_dipole**(*omega, xo, no, grid, *, c=None*)

Line source with dipole characteristics parallel to the z-axis.

Parameters

- **omega** (*float*) – Frequency of source.
- **xo** (*(3,) array_like*) – Position of source. Note: third component of xo is ignored.
- **xo** (*(3,) array_like*) – Normal vector of the source.
- **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.

Notes

$$G(\mathbf{x}-\mathbf{x}_0,\omega) = \frac{\mathrm{i}k}{4} H_1^{(2)}\left(\frac{\omega}{c}|\mathbf{x}-\mathbf{x}_0|\right) \cos\phi$$

`sfs.fd.source.line_bandlimited`(*omega, xo, grid, *, max_order=None, c=None*)

Spatially bandlimited (modal) line source parallel to the z-axis.

Parameters

- **omega** (*float*) – Frequency of source.
- **xo** (*(3,) array_like*) – Position of source. Note: third component of xo is ignored.
- **grid** (*triple of array_like*) – The grid that is used for the sound field calculations. See `sfs.util.xyz_grid()`.
- **max_order** (*int, optional*) – Number of elements for series expansion of the source. No bandlimitation if not given.
- **c** (*float, optional*) – Speed of sound.

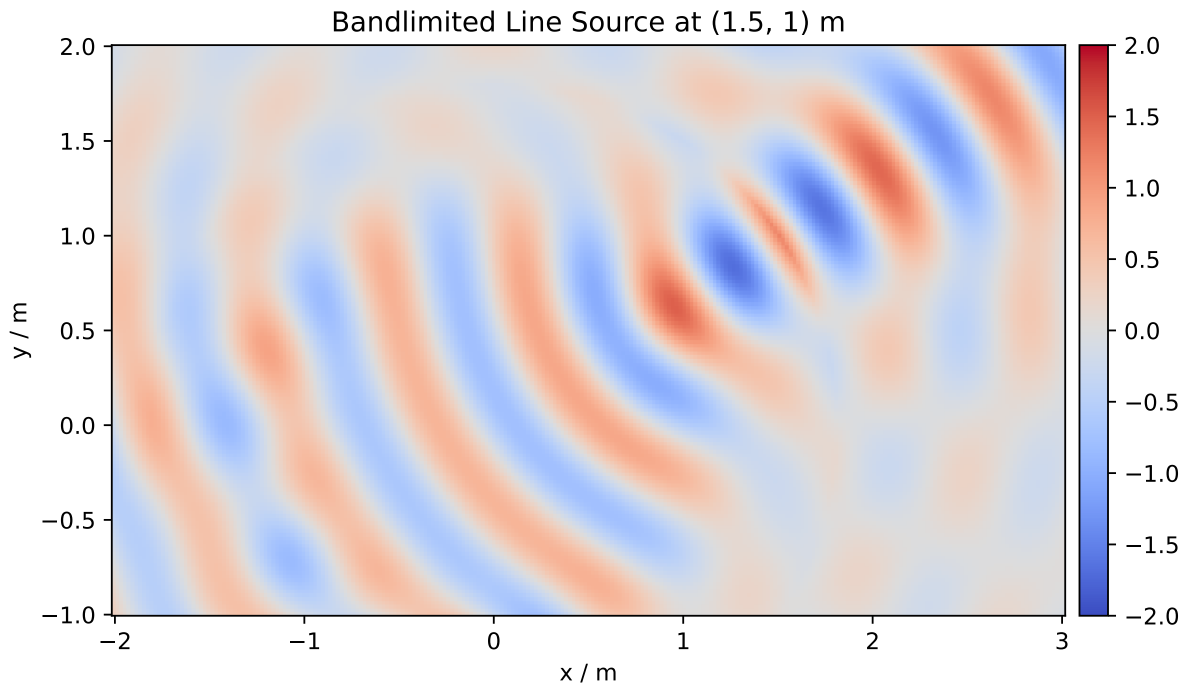
Returns *numpy.ndarray* – Sound pressure at positions given by *grid*.

Notes

$$G(\mathbf{x}-\mathbf{x}_0,\omega) = -\frac{\mathrm{i}}{4} \sum_{\nu=-N}^{N} e^{j\nu(\alpha-\alpha_0)} \begin{cases} J_\nu(\frac{\omega}{c}r)H_\nu^{(2)}(\frac{\omega}{c}r_0) & \text{for } r \leq r_0 \\ J_\nu(\frac{\omega}{c}r_0)H_\nu^{(2)}(\frac{\omega}{c}r) & \text{for } r > r_0 \end{cases}$$

Examples

```
p = sfs.fd.source.line_bandlimited(omega, x0, grid, max_order=10)
sfs.plot2d.amplitude(p * normalization_line, grid)
plt.title("Bandlimited Line Source at {} m".format(x0[:2]))
```



`sfs.fd.source.`**`line_dirichlet_edge`**(*omega, xo, 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.
- **xo** (*(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 *numpy.ndarray* – Complex pressure at grid positions.

`sfs.fd.source.`**`plane`**(*omega, xo, no, grid, *, c=None*)

Plane wave.

Parameters

- **omega** (*float*) – Frequency of plane wave.
- **xo** (*(3,) array_like*) – Position of plane wave.
- **no** (*(3,) array_like*) – Normal vector (direction) of plane wave.
- **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

$$G(\mathbf{x}, \omega) = \mathrm{e}^{-\mathrm{i}\frac{\omega}{c}\mathbf{n}\mathbf{x}}$$

Examples

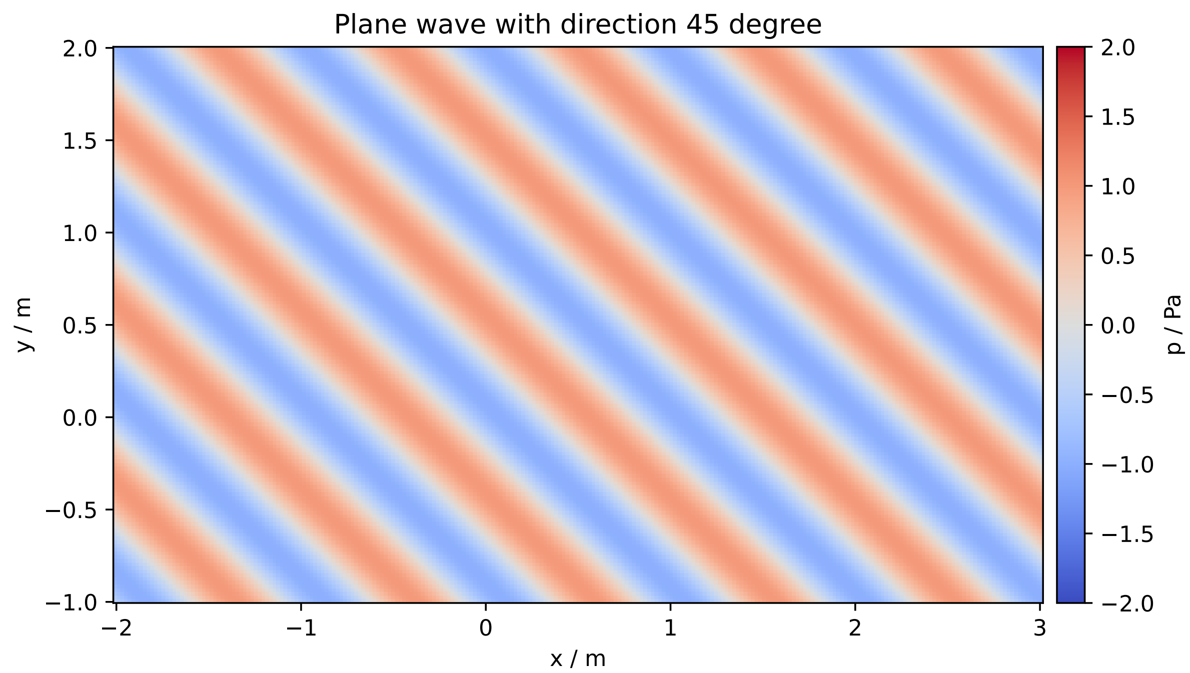
```
direction = 45  # degree
n0 = sfs.util.direction_vector(np.radians(direction))
p = sfs.fd.source.plane(omega, x0, n0, grid)
sfs.plot2d.amplitude(p, grid, colorbar_kwargs=dict(label="p / Pa"))
plt.title("Plane wave with direction {} degree".format(direction))
```

`sfs.fd.source.`**`plane_velocity`**(*omega, xo, no, grid, *, c=None, rhoo=None*)

Velocity of a plane wave.

Parameters

- **omega** (*float*) – Frequency of plane wave.
- **xo** (*(3,) array_like*) – Position of plane wave.
- **no** (*(3,) array_like*) – Normal vector (direction) of plane wave.



- **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.
- **rhoo** (*float, optional*) – Static density of air.

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

Notes

$$V(\mathbf{x},\omega) = \frac{1}{\rho c} \, \mathrm{e}^{-\mathrm{i}\frac{\omega}{c}\mathbf{n}\mathbf{x}} \, \mathbf{n}$$

Examples

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

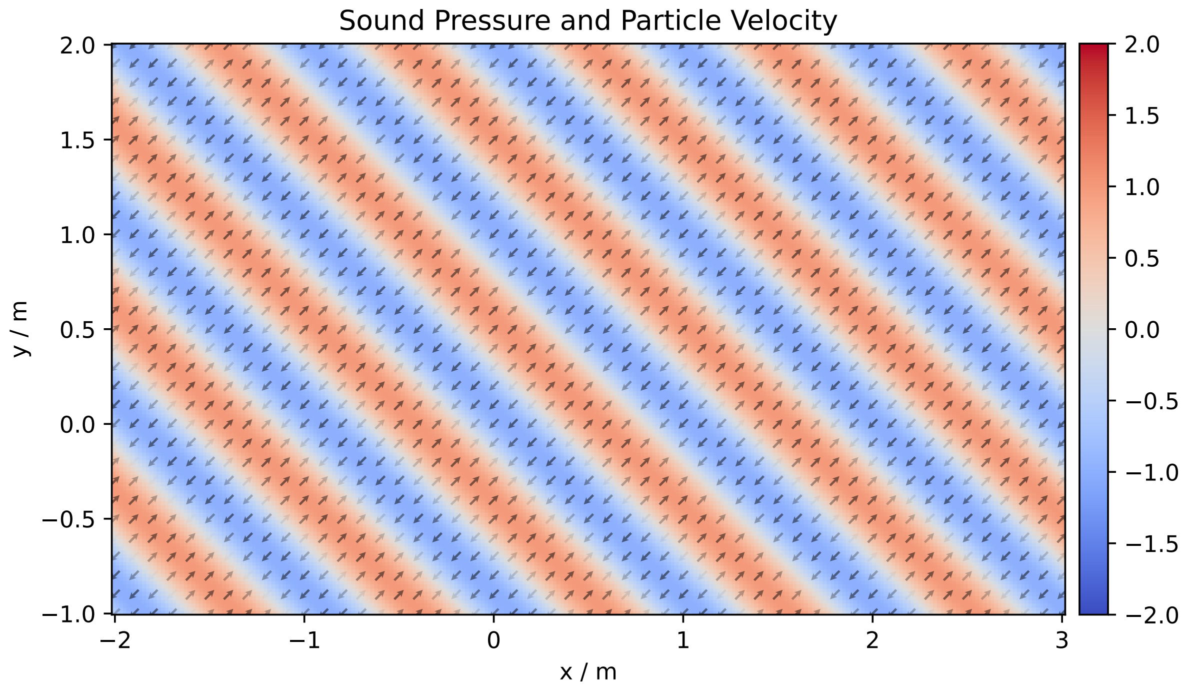
```
v = sfs.fd.source.plane_velocity(omega, x0, n0, vgrid)
sfs.plot2d.amplitude(p, grid)
sfs.plot2d.vectors(v, vgrid)
plt.title("Sound Pressure and Particle Velocity")
```

`sfs.fd.source.`**`plane_averaged_intensity`**(*omega, xo, no, grid, *, c=None, rhoo=None*)

Averaged intensity of a plane wave.

Parameters

- **omega** (*float*) – Frequency of plane wave.
- **xo** (*(3,) array_like*) – Position of plane wave.
- **no** (*(3,) array_like*) – Normal vector (direction) of plane wave.
- **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.
- **rhoo** (*float, optional*) – Static density of air.

Returns *XyzComponents* – Averaged intensity at positions given by *grid*.

Notes

$$I(\mathbf{x}, \omega) = \frac{1}{2\rho c}\mathbf{n}$$

sfs.fd.source.**pulsating_sphere**(*omega*, *center*, *radius*, *amplitude*, *grid*, *, *inside=False*, *c=None*)

Sound pressure of a pulsating sphere.

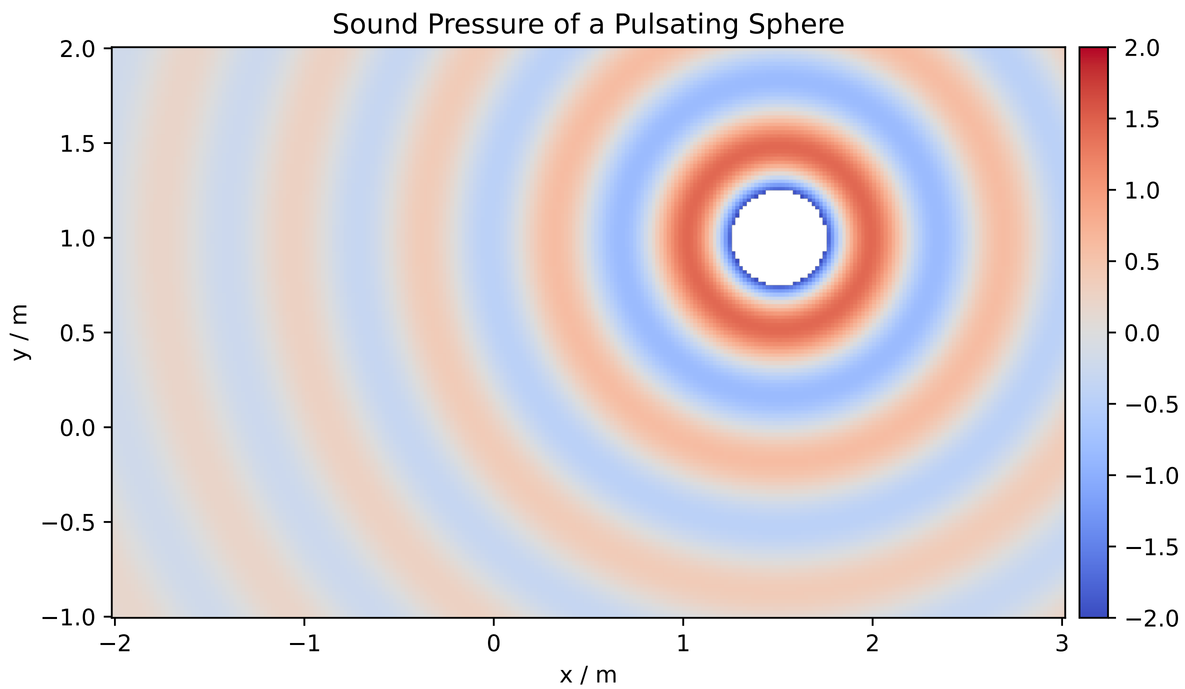
Parameters

- **omega** (*float*) – Frequency of pulsating sphere
- **center** (*(3,) array_like*) – Center of sphere.
- **radius** (*float*) – Radius of sphere.
- **amplitude** (*float*) – Amplitude of displacement.
- **grid** (*triple of array_like*) – The grid that is used for the sound field calculations. See `sfs.util.xyz_grid()`.
- **inside** (*bool, optional*) – As default, `numpy.nan`[11] is returned for inside the sphere. If `inside=True`, the sound field inside the sphere is extrapolated.
- **c** (*float, optional*) – Speed of sound.

Returns *numpy.ndarray* – Sound pressure at positions given by *grid*. If `inside=False`, `numpy.nan`[12] is returned for inside the sphere.

Examples

```
radius = 0.25
amplitude = 1 / (radius * omega * sfs.default.rho0 * sfs.default.c)
p = sfs.fd.source.pulsating_sphere(omega, x0, radius, amplitude, grid)
sfs.plot2d.amplitude(p, grid)
plt.title("Sound Pressure of a Pulsating Sphere")
```



`sfs.fd.source.`**`pulsating_sphere_velocity`**(*omega, center, radius, amplitude, grid, *, c=None*)

Particle velocity of a pulsating sphere.

Parameters

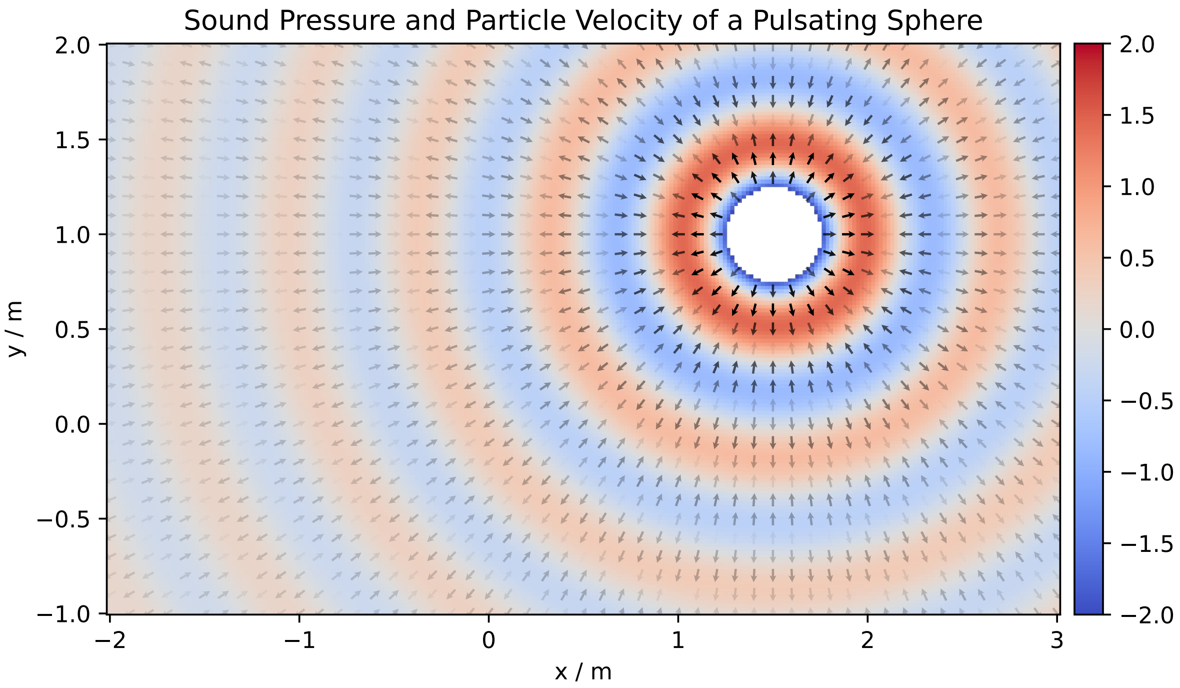
- **omega** (*float*) – Frequency of pulsating sphere
- **center** (*(3,) array_like*) – Center of sphere.
- **radius** (*float*) – Radius of sphere.
- **amplitude** (*float*) – Amplitude of displacement.
- **grid** (*triple of array_like*) – The grid that is used for the sound field calculations. See *`sfs.util.xyz_grid()`*.
- **c** (*float, optional*) – Speed of sound.

Returns *`XyzComponents`* – Particle velocity at positions given by *grid*. `numpy.nan`[13] is returned for inside the sphere.

[11] https://numpy.org/doc/stable/reference/constants.html#numpy.nan
[12] https://numpy.org/doc/stable/reference/constants.html#numpy.nan

Examples

```
v = sfs.fd.source.pulsating_sphere_velocity(omega, x0, radius, amplitude,␣
 ↪vgrid)
sfs.plot2d.amplitude(p, grid)
sfs.plot2d.vectors(v, vgrid)
plt.title("Sound Pressure and Particle Velocity of a Pulsating Sphere")
```



sfs.fd.wfs

Compute WFS driving functions.

```
import matplotlib.pyplot as plt
import numpy as np
import sfs

plt.rcParams['figure.figsize'] = 6, 6

xs = -1.5, 1.5, 0
xs_focused = -0.5, 0.5, 0
# normal vector for plane wave:
npw = sfs.util.direction_vector(np.radians(-45))
# normal vector for focused source:
ns_focused = sfs.util.direction_vector(np.radians(-45))
f = 300  # Hz
omega = 2 * np.pi * f
R = 1.5  # Radius of circular loudspeaker array

grid = sfs.util.xyz_grid([-2, 2], [-2, 2], 0, spacing=0.02)
```

(continues on next page)

[13] https://numpy.org/doc/stable/reference/constants.html#numpy.nan

(continued from previous page)

```
array = sfs.array.circular(N=32, R=R)

def plot(d, selection, secondary_source):
    p = sfs.fd.synthesize(d, selection, array, secondary_source, grid=grid)
    sfs.plot2d.amplitude(p, grid)
    sfs.plot2d.loudspeakers(array.x, array.n, selection * array.a, size=0.15)
```

Functions

`focused_25d`(omega, xo, no, xs, ns, *[, ...])	Driving function for 2.5-dimensional WFS for a focused source.
`focused_2d`(omega, xo, no, xs, ns, *[, c])	Driving function for 2/3-dimensional WFS for a focused source.
`focused_3d`(omega, xo, no, xs, ns, *[, c])	Driving function for 2/3-dimensional WFS for a focused source.
`line_2d`(omega, xo, no, xs, *[, c])	Driving function for 2-dimensional WFS for a virtual line source.
`plane_25d`(omega, xo, no[, n, xref, c, omalias])	Driving function for 2.5-dimensional WFS for a virtual plane wave.
`plane_2d`(omega, xo, no[, n, c])	Driving function for 2/3-dimensional WFS for a virtual plane wave.
`plane_3d`(omega, xo, no[, n, c])	Driving function for 2/3-dimensional WFS for a virtual plane wave.
`plane_3d_delay`(omega, xo, no[, n, c])	Delay-only driving function for a virtual plane wave.
`point_25d`(omega, xo, no, xs[, xref, c, omalias])	Driving function for 2.5-dimensional WFS of a virtual point source.
`point_25d_legacy`(omega, xo, no, xs[, xref, ...])	Driving function for 2.5-dimensional WFS for a virtual point source.
`point_2d`(omega, xo, no, xs, *[, c])	Driving function for 2/3-dimensional WFS for a virtual point source.
`point_3d`(omega, xo, no, xs, *[, c])	Driving function for 2/3-dimensional WFS for a virtual point source.
`preeq_25d`(omega, omalias, c)	Pre-equalization filter for 2.5-dimensional WFS.
`soundfigure_3d`(omega, xo, no, figure[, npw, c])	Compute driving function for a 2D sound figure.

`sfs.fd.wfs.`**`line_2d`**(*omega, xo, no, xs, *, c=None*)

Driving function for 2-dimensional WFS for a virtual line source.

Parameters

- **omega** (*float*) – Angular frequency of line source.
- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **no** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **xs** (*(3,) array_like*) – Position of virtual line source.
- **c** (*float, optional*) – Speed of sound.

Returns

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

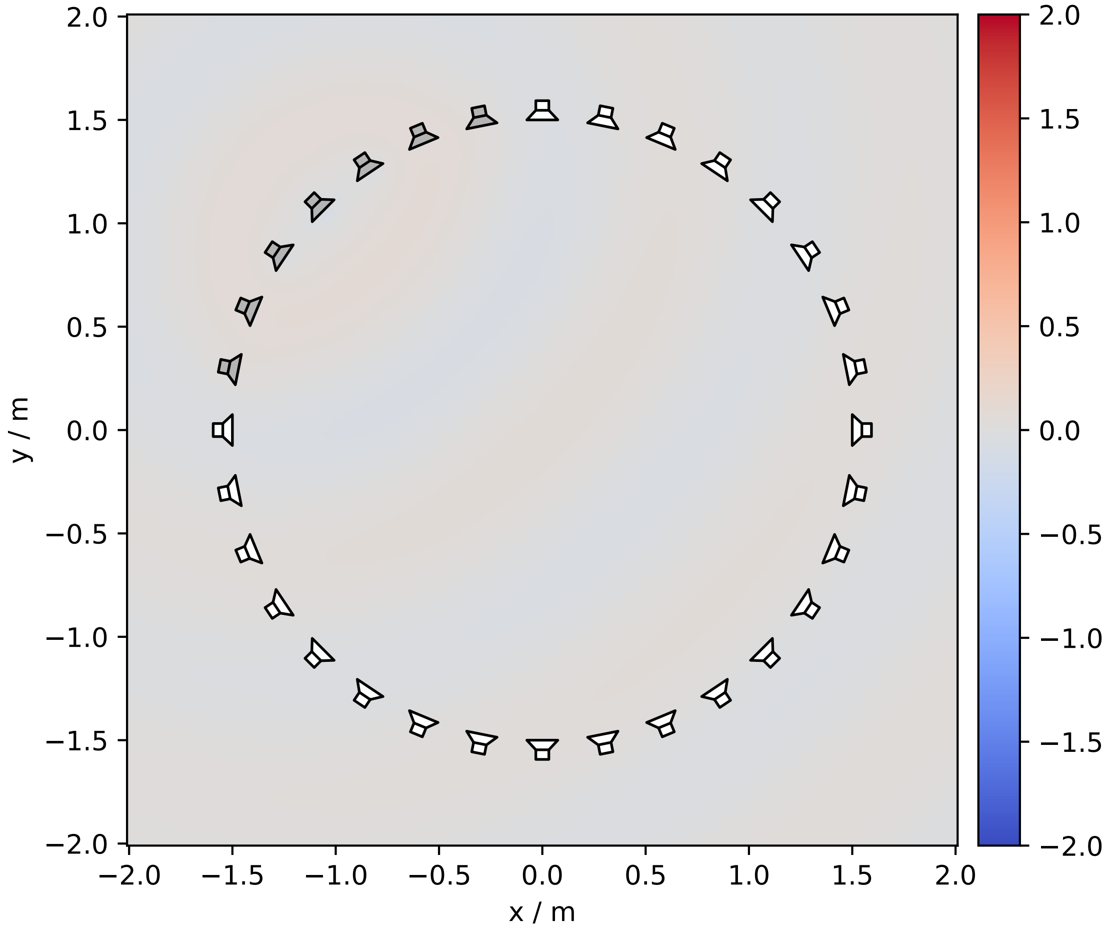
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

$$D(\mathbf{x}_0, \omega) = \frac{\mathrm{i}}{2} \frac{\omega}{c} \frac{\langle \mathbf{x} - \mathbf{x}_0, \mathbf{n}_0 \rangle}{|\mathbf{x} - \mathbf{x}_\mathrm{s}|} H_1^{(2)} \left(\frac{\omega}{c} |\mathbf{x} - \mathbf{x}_\mathrm{s}| \right)$$

Examples

```
d, selection, secondary_source = sfs.fd.wfs.line_2d(
    omega, array.x, array.n, xs)
plot(d, selection, secondary_source)
```



`sfs.fd.wfs.`**`point_2d`**(*omega, xo, no, xs, *, c=None*)

Driving function for 2/3-dimensional WFS for a virtual point source.

Parameters

- **omega** (*float*) – Angular frequency of point source.
- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **no** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.

- **xs** (*(3,) array_like*) – Position of virtual point source.
- **c** (*float, optional*) – Speed of sound.

Returns

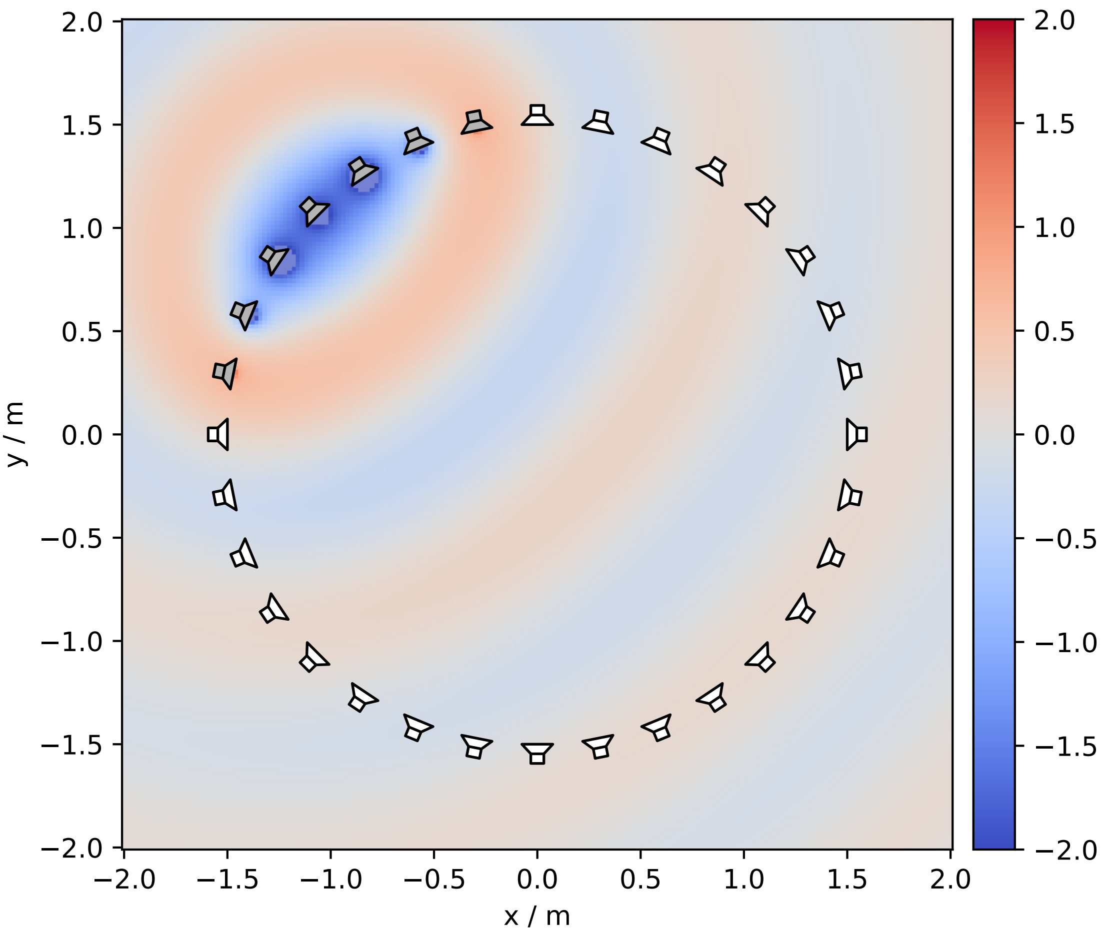
- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

$$D(\mathbf{x}_0, \omega) = \mathrm{i}\frac{\omega}{c} \frac{\langle \mathbf{x}_0 - \mathbf{x}_\mathrm{s}, \mathbf{n}_0 \rangle}{|\mathbf{x}_0 - \mathbf{x}_\mathrm{s}|^{\frac{3}{2}}} \mathrm{e}^{-\mathrm{i}\frac{\omega}{c}|\mathbf{x}_0 - \mathbf{x}_\mathrm{s}|}$$

Examples

```
d, selection, secondary_source = sfs.fd.wfs.point_3d(
    omega, array.x, array.n, xs)
plot(d, selection, secondary_source)
```



`sfs.fd.wfs.`**`point_25d`**(*omega, x0, n0, xs, xref=[0, 0, 0], c=None, omalias=None*)

Driving function for 2.5-dimensional WFS of a virtual point source.

Changed in version 0.5: see notes, old handling of `point_25d()` is now `point_25d_legacy()`

Parameters

- **omega** (*float*) – Angular frequency of point source.
- **x0** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **n0** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **xs** (*(3,) array_like*) – Position of virtual point source.
- **xref** (*(3,) array_like, optional*) – Reference point xref or contour xref(x0) for amplitude correct synthesis.
- **c** (*float, optional*) – Speed of sound in m/s.
- **omalias** (*float, optional*) – Angular frequency where spatial aliasing becomes prominent.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

`point_25d()` derives 2.5D WFS from the 3D Neumann-Rayleigh integral (i.e. the TU Delft approach). The eq. (3.10), (3.11) in [Sta97], equivalent to Eq. (2.137) in [Sch16]

$$D(\mathbf{x}_0,\omega) = \sqrt{8\pi\,\mathrm{i}\frac{\omega}{c}}\sqrt{\frac{|\mathbf{x}_\mathrm{ref}-\mathbf{x}_0|\cdot|\mathbf{x}_0-\mathbf{x}_\mathrm{s}|}{|\mathbf{x}_\mathrm{ref}-\mathbf{x}_0|+|\mathbf{x}_0-\mathbf{x}_\mathrm{s}|}}\left\langle\frac{\mathbf{x}_0-\mathbf{x}_\mathrm{s}}{|\mathbf{x}_0-\mathbf{x}_\mathrm{s}|},\mathbf{n}_0\right\rangle\frac{\mathrm{e}^{-\mathrm{i}\frac{\omega}{c}|\mathbf{x}_0-\mathbf{x}_\mathrm{s}|}}{4\pi\,|\mathbf{x}_0-\mathbf{x}_\mathrm{s}|}$$

is implemented. The theoretical link of `point_25d()` and `point_25d_legacy()` was introduced as *unified WFS framework* in [FFSS17].

Examples

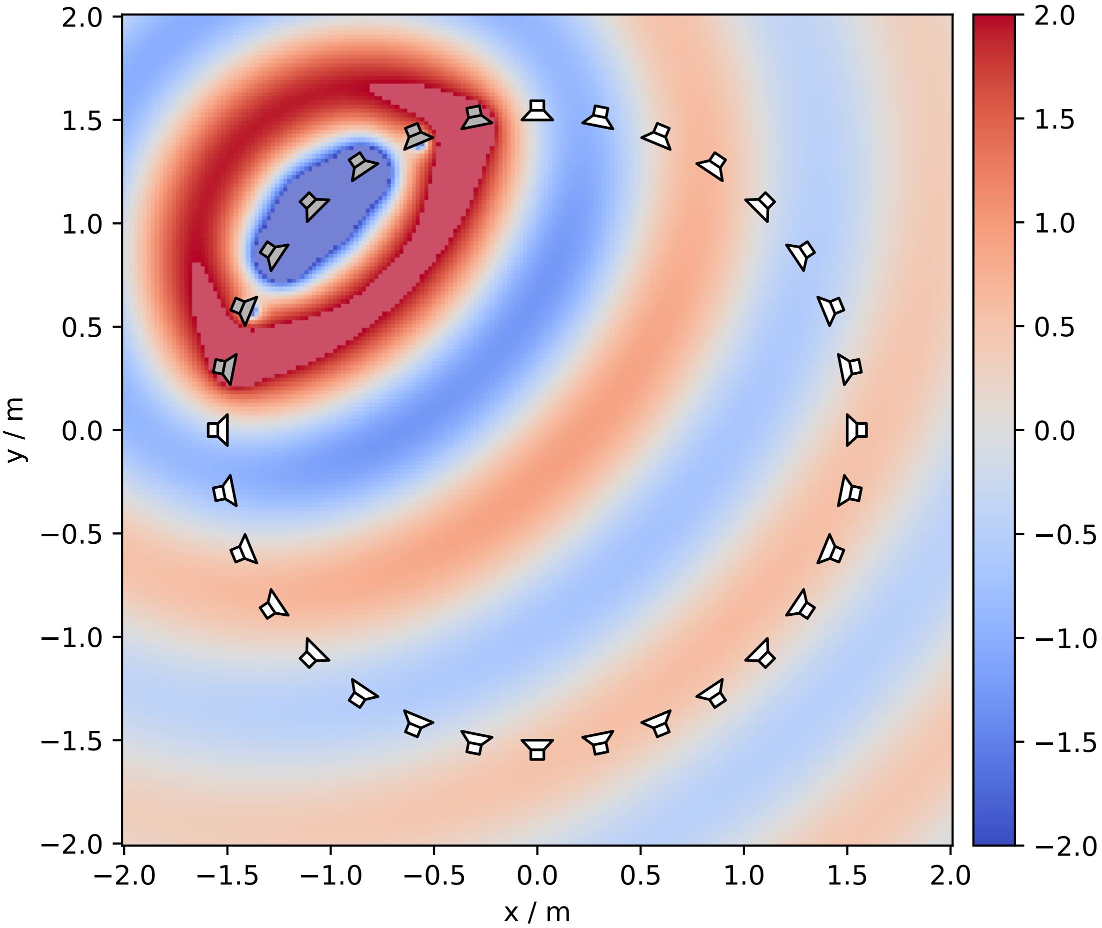
```
d, selection, secondary_source = sfs.fd.wfs.point_25d(
    omega, array.x, array.n, xs)
normalize_gain = 4 * np.pi * np.linalg.norm(xs)
plot(normalize_gain * d, selection, secondary_source)
```

`sfs.fd.wfs.`**`point_3d`**(*omega, x0, n0, xs, *, c=None*)

Driving function for 2/3-dimensional WFS for a virtual point source.

Parameters

- **omega** (*float*) – Angular frequency of point source.
- **x0** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **n0** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **xs** (*(3,) array_like*) – Position of virtual point source.



2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y / m
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x / m

- **c** (*float, optional*) – Speed of sound.

Returns

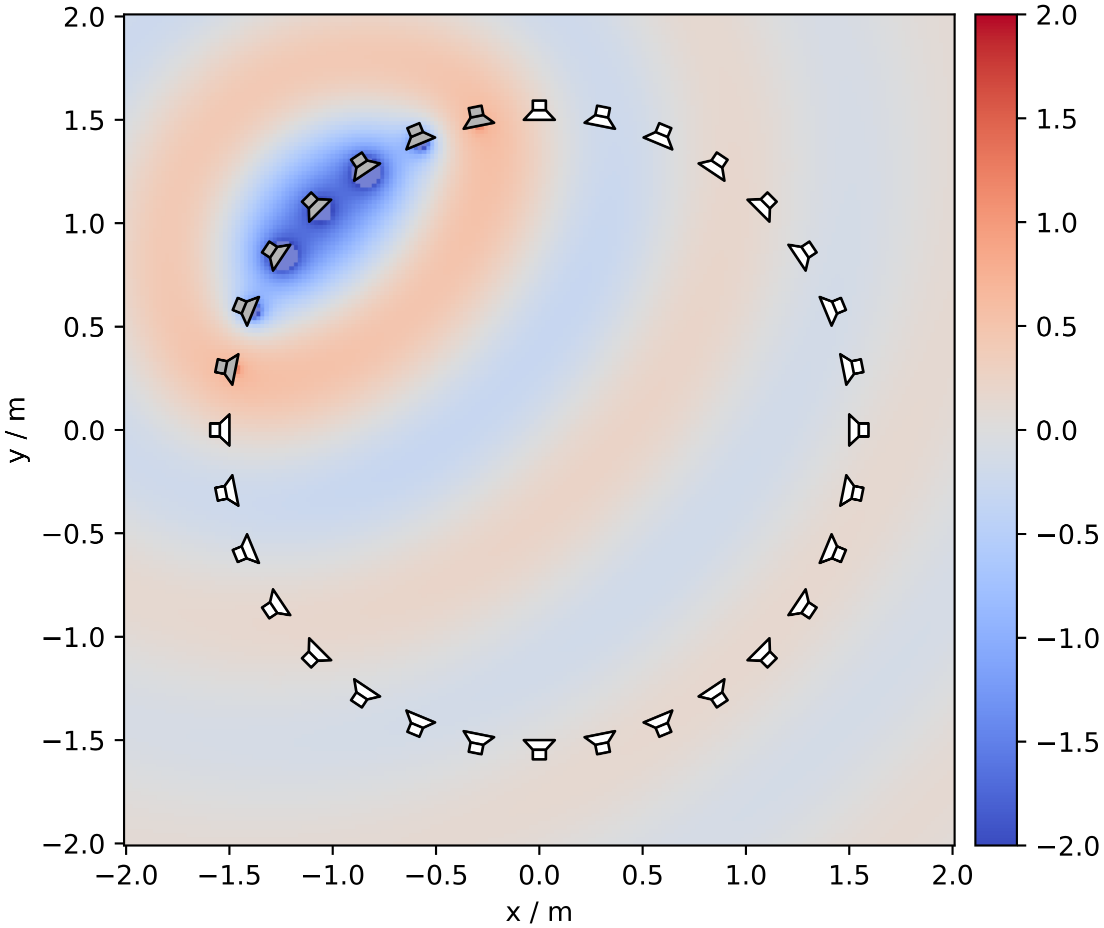
- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

$$D(\mathbf{x}_0, \omega) = \mathrm{i}\frac{\omega}{c}\frac{\langle \mathbf{x}_0 - \mathbf{x}_\mathrm{s}, \mathbf{n}_0 \rangle}{|\mathbf{x}_0 - \mathbf{x}_\mathrm{s}|^{\frac{3}{2}}}\,\mathrm{e}^{-\mathrm{i}\frac{\omega}{c}|\mathbf{x}_0 - \mathbf{x}_\mathrm{s}|}$$

Examples

```
d, selection, secondary_source = sfs.fd.wfs.point_3d(
    omega, array.x, array.n, xs)
plot(d, selection, secondary_source)
```



`sfs.fd.wfs.`**`point_25d_legacy`**(*omega, x0, n0, xs, xref=[0, 0, 0], c=None, omalias=None*)

Driving function for 2.5-dimensional WFS for a virtual point source.

New in version 0.5: `point_25d()` was renamed to `point_25d_legacy()` (and a new function with the name `point_25d()` was introduced). See notes for further details.

Parameters

- **omega** (*float*) – Angular frequency of point source.
- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **no** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **xs** (*(3,) array_like*) – Position of virtual point source.
- **xref** (*(3,) array_like, optional*) – Reference point for synthesized sound field.
- **c** (*float, optional*) – Speed of sound.
- **omalias** (*float, optional*) – Angular frequency where spatial aliasing becomes prominent.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

`point_25d_legacy()` derives 2.5D WFS from the 2D Neumann-Rayleigh integral (i.e. the approach by Rabenstein & Spors), cf. [SRA08].

$$D(\mathbf{x}_0,\omega) = \sqrt{\mathrm{i}\frac{\omega}{c}|\mathbf{x}_\mathrm{ref}-\mathbf{x}_0|}\frac{\langle \mathbf{x}_0-\mathbf{x}_\mathrm{s}, \mathbf{n}_0\rangle}{|\mathbf{x}_0-\mathbf{x}_\mathrm{s}|^{\frac{3}{2}}}\,\mathrm{e}^{-\mathrm{i}\frac{\omega}{c}|\mathbf{x}_0-\mathbf{x}_\mathrm{s}|}$$

The theoretical link of `point_25d()` and `point_25d_legacy()` was introduced as *unified WFS framework* in [FFSS17]. Also cf. Eq. (2.145)-(2.147) [Sch16].

Examples

```
d, selection, secondary_source = sfs.fd.wfs.point_25d_legacy(
    omega, array.x, array.n, xs)
normalize_gain = np.linalg.norm(xs)
plot(normalize_gain * d, selection, secondary_source)
```

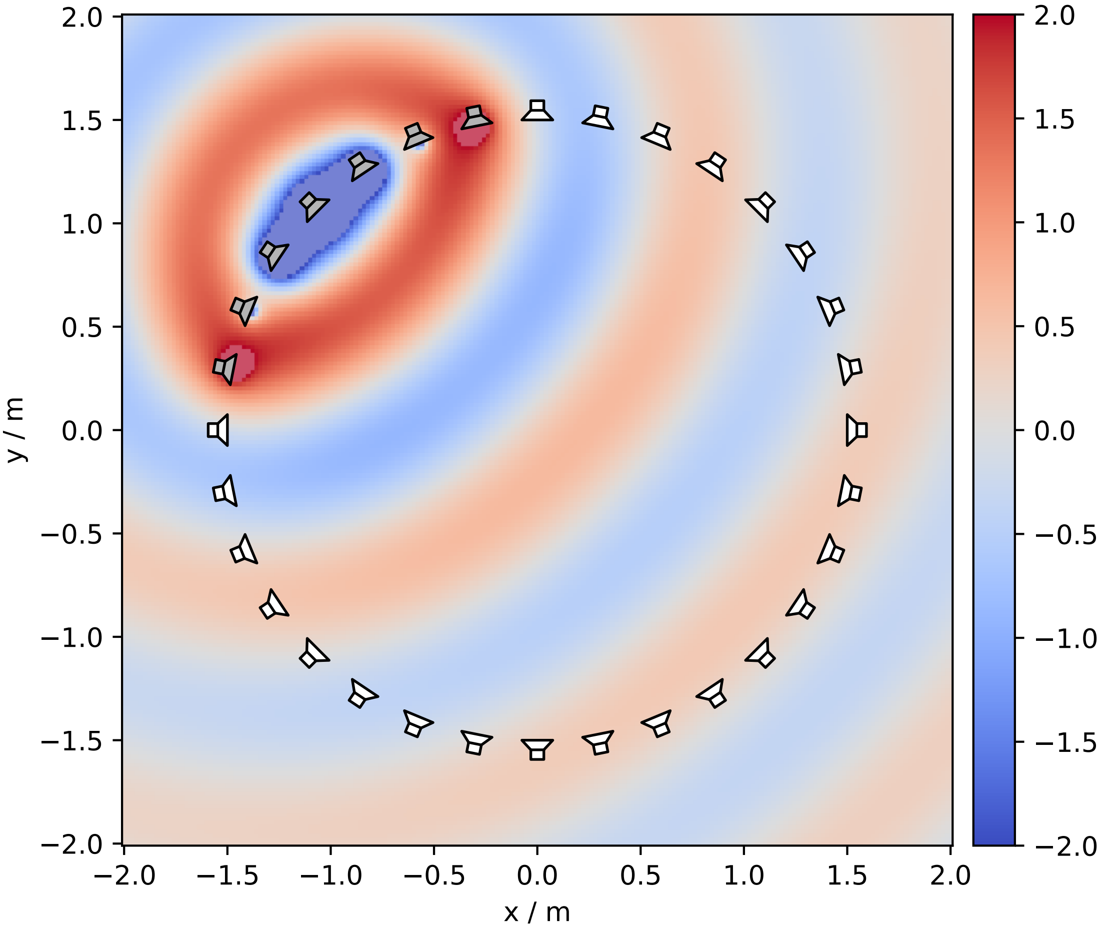
`sfs.fd.wfs.`**`plane_2d`**(*omega, xo, no, n=[0, 1, 0], *, c=None*)

Driving function for 2/3-dimensional WFS for a virtual plane wave.

Parameters

- **omega** (*float*) – Angular frequency of plane wave.
- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **no** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **n** (*(3,) array_like, optional*) – Normal vector (traveling direction) of plane wave.
- **c** (*float, optional*) – Speed of sound.

Returns


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y / m
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x / m

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

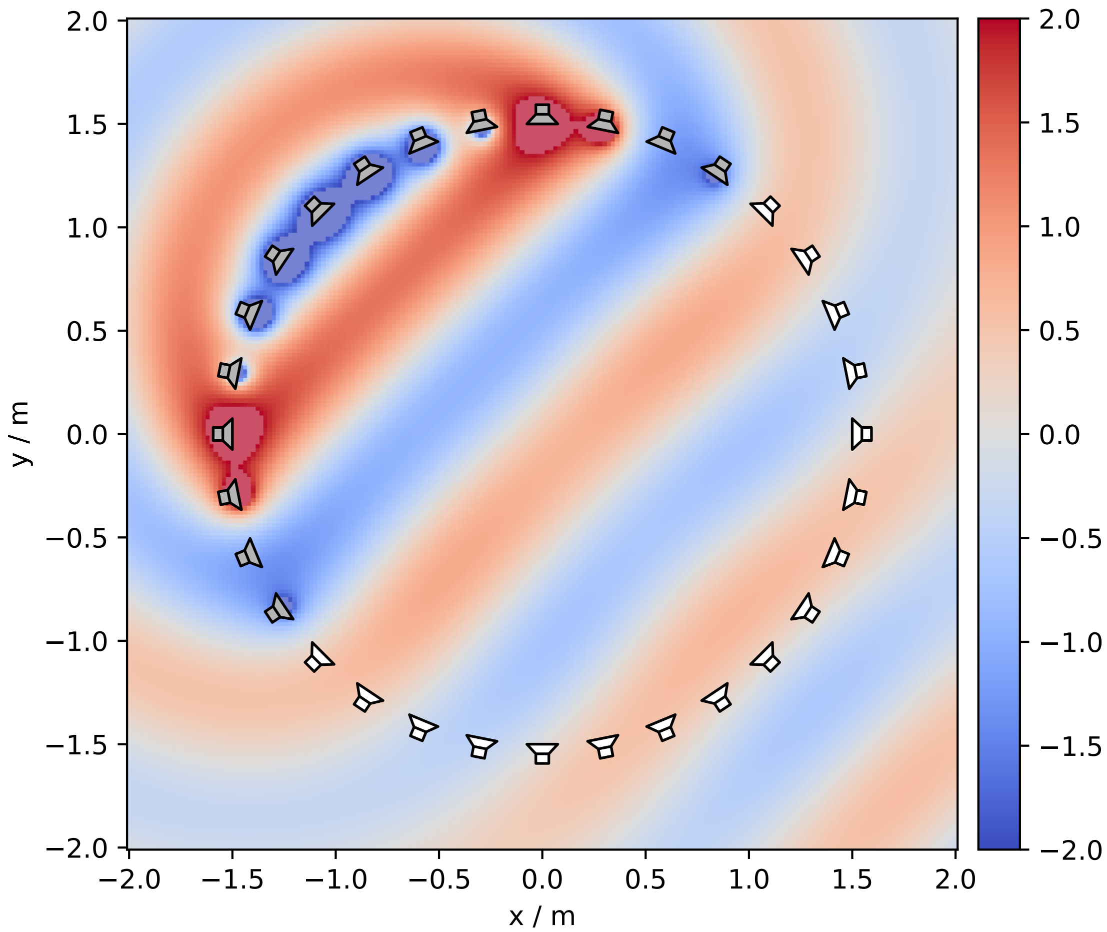
Notes

Eq.(17) from [SRA08]:

$$D(\mathbf{x}_0, \omega) = \mathrm{i}\frac{\omega}{c} \langle \mathbf{n}, \mathbf{n}_0 \rangle \, \mathrm{e}^{-\mathrm{i}\frac{\omega}{c}\langle \mathbf{n}, \mathbf{x}_0 \rangle}$$

Examples

```
d, selection, secondary_source = sfs.fd.wfs.plane_3d(
    omega, array.x, array.n, npw)
plot(d, selection, secondary_source)
```



`sfs.fd.wfs.`**`plane_25d`**(*omega, x0, n0, n=[0, 1, 0], *, xref=[0, 0, 0], c=None, omalias=None*)

Driving function for 2.5-dimensional WFS for a virtual plane wave.

Parameters

- **omega** (*float*) – Angular frequency of plane wave.

- **x0** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **n0** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **n** (*(3,) array_like, optional*) – Normal vector (traveling direction) of plane wave.
- **xref** (*(3,) array_like, optional*) – Reference point for synthesized sound field.
- **c** (*float, optional*) – Speed of sound.
- **omalias** (*float, optional*) – Angular frequency where spatial aliasing becomes prominent.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

$$D_{2.5\mathrm{D}}(\mathbf{x}_0,\omega) = \sqrt{\mathrm{i}\frac{\omega}{c}|\mathbf{x}_\mathrm{ref} - \mathbf{x}_0|}\,\langle\mathbf{n},\mathbf{n}_0\rangle\,\mathrm{e}^{-\mathrm{i}\frac{\omega}{c}\langle\mathbf{n},\mathbf{x}_0\rangle}$$

Examples

```
d, selection, secondary_source = sfs.fd.wfs.plane_25d(
    omega, array.x, array.n, npw)
plot(d, selection, secondary_source)
```

`sfs.fd.wfs.`**`plane_3d`**(*omega, x0, n0, n=[0, 1, 0], *, c=None*)

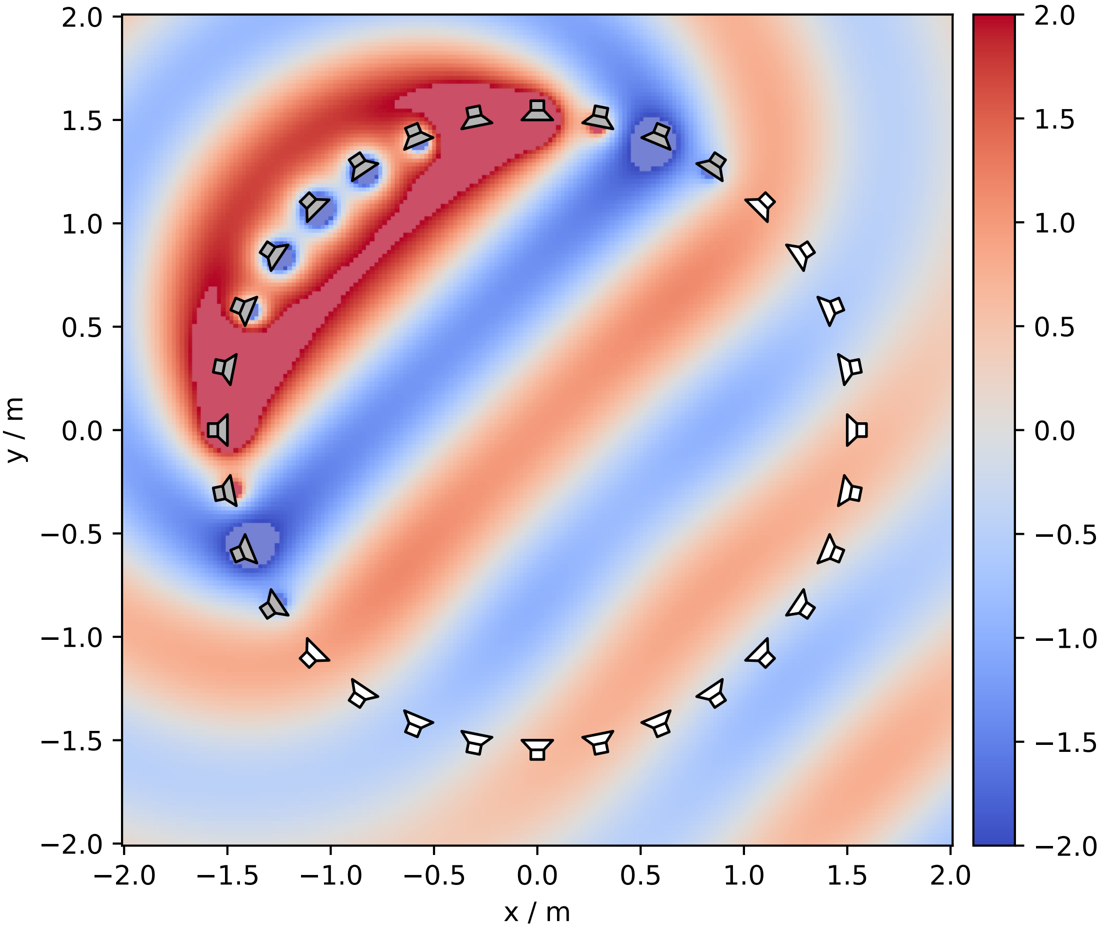
Driving function for 2/3-dimensional WFS for a virtual plane wave.

Parameters

- **omega** (*float*) – Angular frequency of plane wave.
- **x0** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **n0** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **n** (*(3,) array_like, optional*) – Normal vector (traveling direction) of plane wave.
- **c** (*float, optional*) – Speed of sound.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.



2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y / m
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x / m

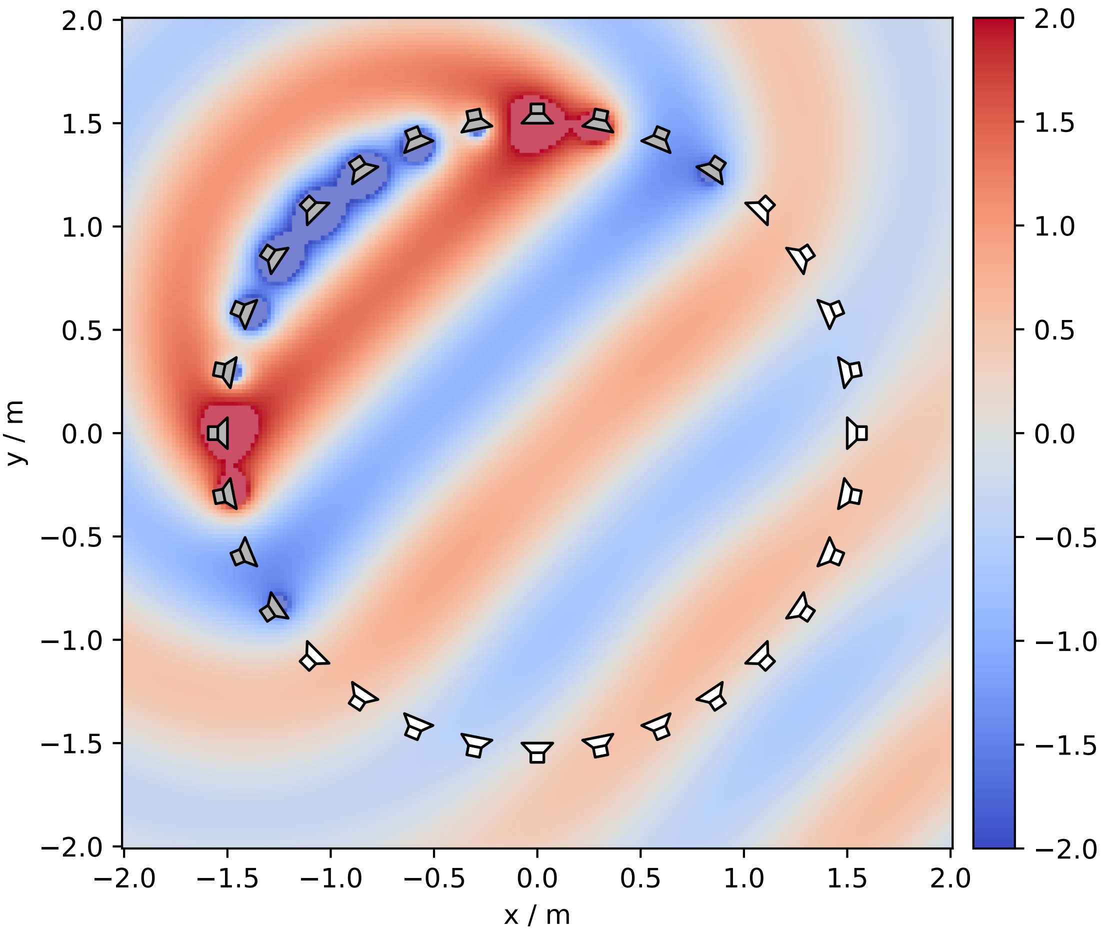
Notes

Eq.(17) from [SRA08]:

$$D(\mathbf{x}_0, \omega) = \mathrm{i}\frac{\omega}{c} \langle \mathbf{n}, \mathbf{n}_0 \rangle \, \mathrm{e}^{-\mathrm{i}\frac{\omega}{c} \langle \mathbf{n}, \mathbf{x}_0 \rangle}$$

Examples

```
d, selection, secondary_source = sfs.fd.wfs.plane_3d(
    omega, array.x, array.n, npw)
plot(d, selection, secondary_source)
```



`sfs.fd.wfs.`**`focused_2d`**(*omega, xo, no, xs, ns, *, c=None*)

Driving function for 2/3-dimensional WFS for a focused source.

Parameters

- **omega** (*float*) – Angular frequency of focused source.
- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **no** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **xs** (*(3,) array_like*) – Position of focused source.
- **ns** (*(3,) array_like*) – Direction of focused source.
- **c** (*float, optional*) – Speed of sound.

Returns

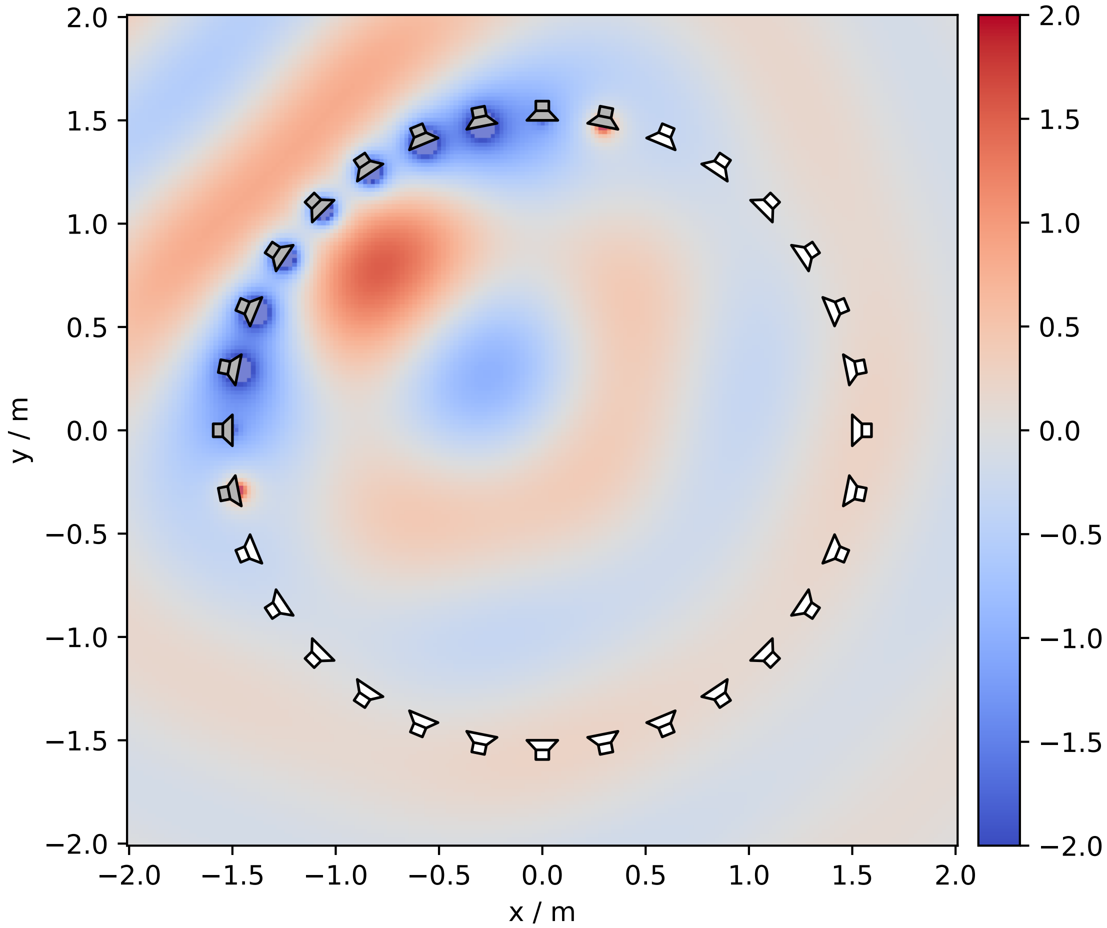
- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See *sfs.fd.synthesize()*.

Notes

$$D(\mathbf{x}_0, \omega) = \mathrm{i}\frac{\omega}{c} \frac{\langle \mathbf{x}_0 - \mathbf{x}_\mathrm{s}, \mathbf{n}_0 \rangle}{|\mathbf{x}_0 - \mathbf{x}_\mathrm{s}|^{\frac{3}{2}}} \mathrm{e}^{\mathrm{i}\frac{\omega}{c}|\mathbf{x}_0 - \mathbf{x}_\mathrm{s}|}$$

Examples

```
d, selection, secondary_source = sfs.fd.wfs.focused_3d(
    omega, array.x, array.n, xs_focused, ns_focused)
plot(d, selection, secondary_source)
```



`sfs.fd.wfs.focused_25d`(*omega, x0, n0, xs, ns, *, xref=[0, 0, 0], c=None, omalias=None*)

Driving function for 2.5-dimensional WFS for a focused source.

Parameters

- **omega** (*float*) – Angular frequency of focused source.
- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **no** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **xs** (*(3,) array_like*) – Position of focused source.
- **ns** (*(3,) array_like*) – Direction of focused source.
- **xref** (*(3,) array_like, optional*) – Reference point for synthesized sound field.
- **c** (*float, optional*) – Speed of sound.
- **omalias** (*float, optional*) – Angular frequency where spatial aliasing becomes prominent.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

$$D(\mathbf{x}_0,\omega) = \sqrt{\mathrm{i}\frac{\omega}{c}|\mathbf{x}_\mathrm{ref}-\mathbf{x}_0|}\frac{\langle \mathbf{x}_0-\mathbf{x}_\mathrm{s}, \mathbf{n}_0\rangle}{|\mathbf{x}_0-\mathbf{x}_\mathrm{s}|^{\frac{3}{2}}}\,\mathrm{e}^{\mathrm{i}\frac{\omega}{c}|\mathbf{x}_0-\mathbf{x}_\mathrm{s}|}$$

Examples

```
d, selection, secondary_source = sfs.fd.wfs.focused_25d(
    omega, array.x, array.n, xs_focused, ns_focused)
plot(d, selection, secondary_source)
```

`sfs.fd.wfs.`**`focused_3d`**(*omega, xo, no, xs, ns, *, c=None*)

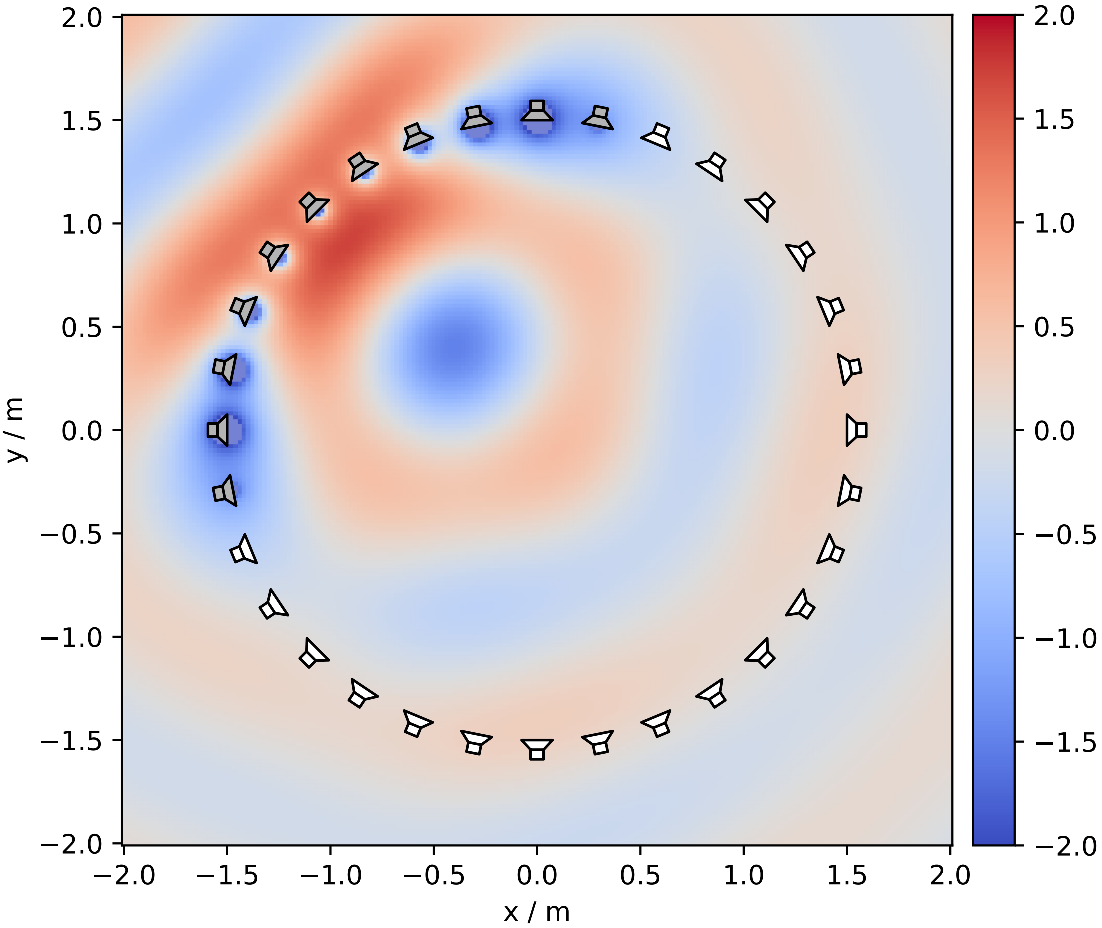
Driving function for 2/3-dimensional WFS for a focused source.

Parameters

- **omega** (*float*) – Angular frequency of focused source.
- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **no** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **xs** (*(3,) array_like*) – Position of focused source.
- **ns** (*(3,) array_like*) – Direction of focused source.
- **c** (*float, optional*) – Speed of sound.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.



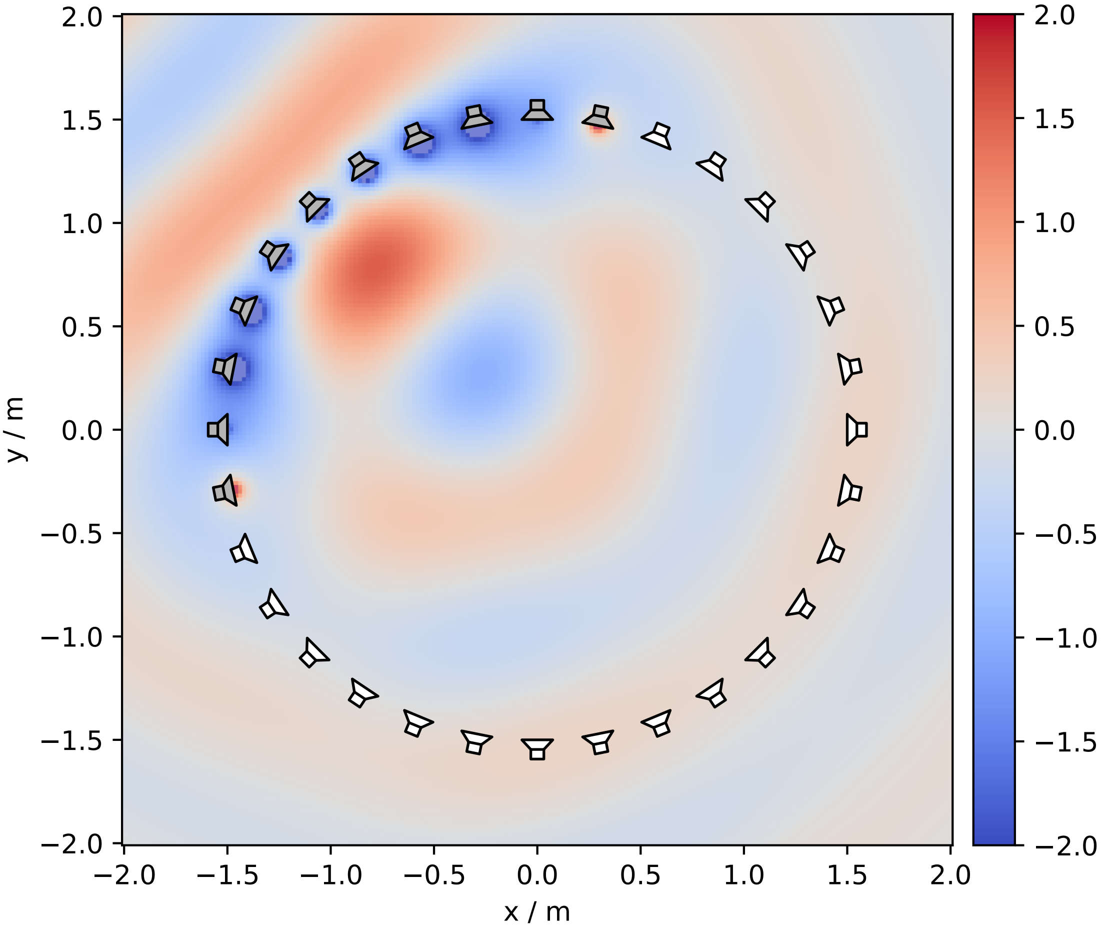
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y / m
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x / m

Notes

$$D(\mathbf{x}_0, \omega) = \mathrm{i}\frac{\omega}{c} \frac{\langle \mathbf{x}_0 - \mathbf{x}_\mathrm{s}, \mathbf{n}_0 \rangle}{|\mathbf{x}_0 - \mathbf{x}_\mathrm{s}|^{\frac{3}{2}}} \mathrm{e}^{\mathrm{i}\frac{\omega}{c}|\mathbf{x}_0 - \mathbf{x}_\mathrm{s}|}$$

Examples

```
d, selection, secondary_source = sfs.fd.wfs.focused_3d(
    omega, array.x, array.n, xs_focused, ns_focused)
plot(d, selection, secondary_source)
```



sfs.fd.wfs.**preeq_25d**(*omega, omalias, c*)

Pre-equalization filter for 2.5-dimensional WFS.

Parameters

- **omega** (*float*) – Angular frequency.
- **omalias** (*float*) – Angular frequency where spatial aliasing becomes prominent.
- **c** (*float*) – Speed of sound.

Returns *float* – Complex weight for given angular frequency.

Notes

$$H(\omega) = \begin{cases} \sqrt{\mathrm{i}\frac{\omega}{c}} & \text{for } \omega \leq \omega_{\text{alias}} \\ \sqrt{\mathrm{i}\frac{\omega_{\text{alias}}}{c}} & \text{for } \omega > \omega_{\text{alias}} \end{cases}$$

`sfs.fd.wfs.`**`plane_3d_delay`**(*omega, xo, no, n=[0, 1, 0], *, c=None*)

Delay-only driving function for a virtual plane wave.

Parameters

- **omega** (*float*) – Angular frequency of plane wave.
- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **no** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **n** (*(3,) array_like, optional*) – Normal vector (traveling direction) of plane wave.
- **c** (*float, optional*) – Speed of sound.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

$$D(\mathbf{x}_0, \omega) = \mathrm{e}^{-\mathrm{i}\frac{\omega}{c}\langle \mathbf{n}, \mathbf{x}_0 \rangle}$$

Examples

```
d, selection, secondary_source = sfs.fd.wfs.plane_3d_delay(
    omega, array.x, array.n, npw)
plot(d, selection, secondary_source)
```

`sfs.fd.wfs.`**`soundfigure_3d`**(*omega, xo, no, figure, npw=[0, 0, 1], *, c=None*)

Compute driving function for a 2D sound figure.

Based on [Helwani et al., The Synthesis of Sound Figures, MSSP, 2013]

sfs.fd.nfchoa

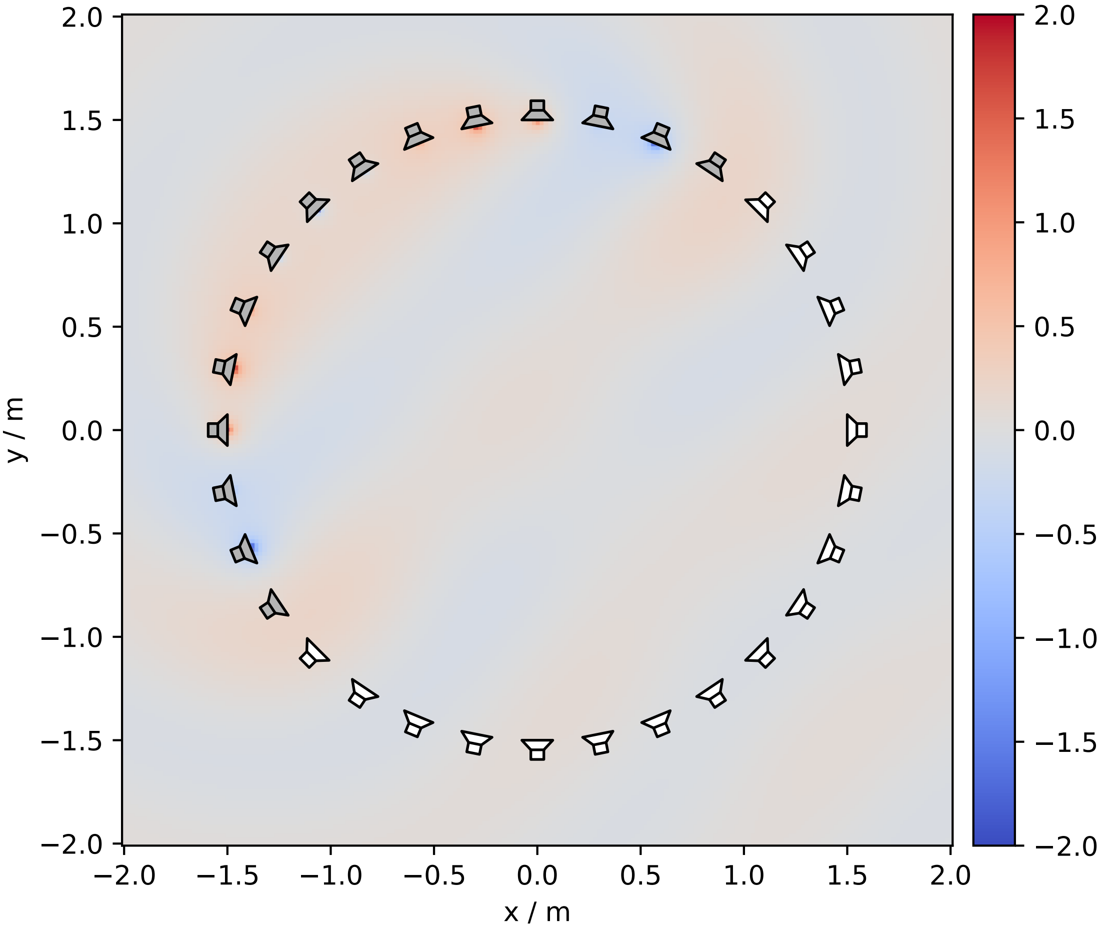
Compute NFC-HOA driving functions.

```
import matplotlib.pyplot as plt
import numpy as np
import sfs

plt.rcParams['figure.figsize'] = 6, 6

xs = -1.5, 1.5, 0
# normal vector for plane wave:
```

(continues on next page)


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y / m
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x / m

(continued from previous page)

```
npw = sfs.util.direction_vector(np.radians(-45))
f = 300  # Hz
omega = 2 * np.pi * f
R = 1.5  # Radius of circular loudspeaker array

grid = sfs.util.xyz_grid([-2, 2], [-2, 2], 0, spacing=0.02)

array = sfs.array.circular(N=32, R=R)

def plot(d, selection, secondary_source):
    p = sfs.fd.synthesize(d, selection, array, secondary_source, grid=grid)
    sfs.plot2d.amplitude(p, grid)
    sfs.plot2d.loudspeakers(array.x, array.n, selection * array.a, size=0.15)
```

Functions

plane_25d(omega, x0, r0[, n, max_order, c])	Driving function for 2.5-dimensional NFC-HOA for a virtual plane wave.
plane_2d(omega, x0, r0[, n, max_order, c])	Driving function for 2-dimensional NFC-HOA for a virtual plane wave.
point_25d(omega, x0, r0, xs, *[, max_order, c])	Driving function for 2.5-dimensional NFC-HOA for a virtual point source.

`sfs.fd.nfchoa.`**`plane_2d`**(*omega*, *x0*, *r0*, *n=[0, 1, 0]*, ***, *max_order=None*, *c=None*)

Driving function for 2-dimensional NFC-HOA for a virtual plane wave.

Parameters

- **omega** (*float*) – Angular frequency of plane wave.
- **x0** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **r0** (*float*) – Radius of circular secondary source distribution.
- **n** (*(3,) array_like, optional*) – Normal vector (traveling direction) of plane wave.
- **max_order** (*float, optional*) – Maximum order of circular harmonics used for the calculation.
- **c** (*float, optional*) – Speed of sound.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing only `True` indicating that all secondary source are “active” for NFC-HOA.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See *sfs.fd.synthesize()*.

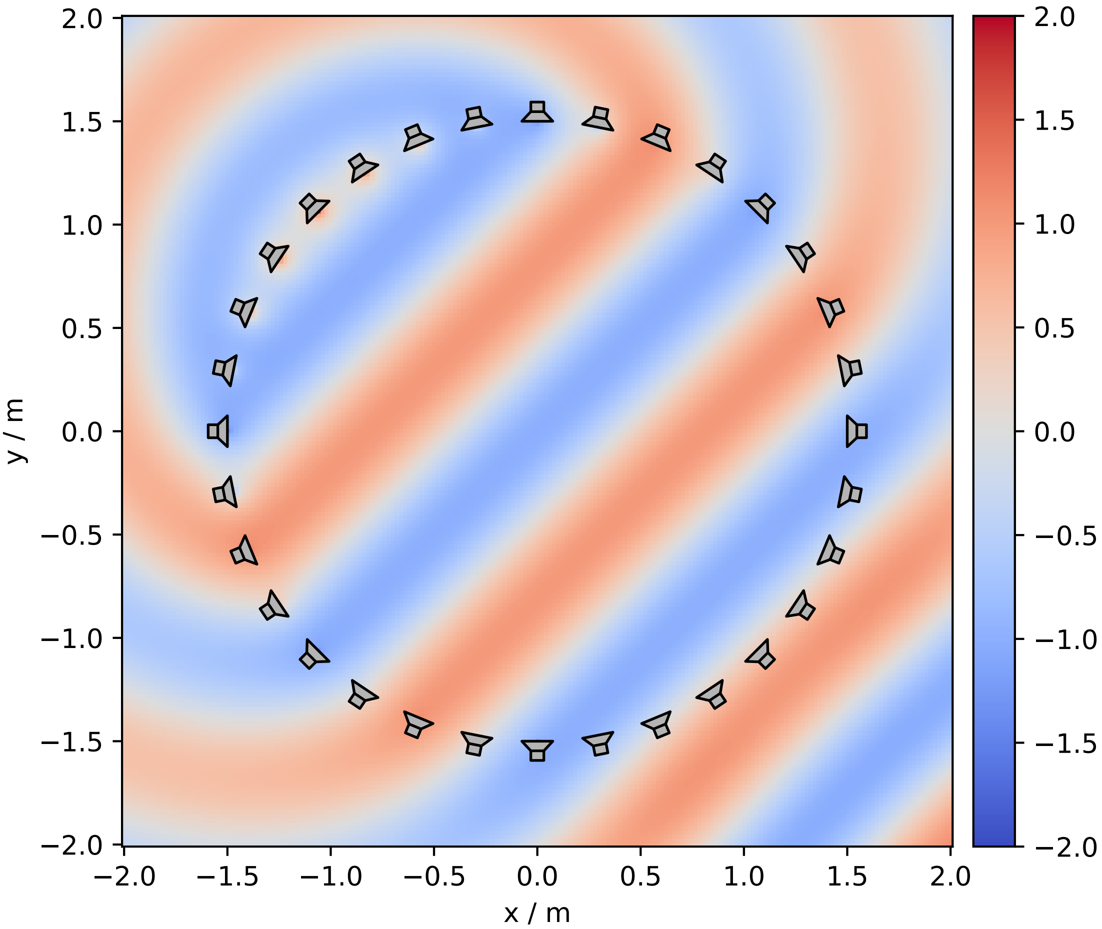
Notes

$$D(\phi_0, \omega) = -\frac{2\mathrm{i}}{\pi r_0} \sum_{m=-M}^{M} \frac{\mathrm{i}^{-m}}{H_m^{(2)}\left(\frac{\omega}{c} r_0\right)} \mathrm{e}^{\mathrm{i}m(\phi_0 - \phi_{\mathrm{pw}})}$$

See https://sfs.rtfd.io/d_nfchoa/#equation-fd-nfchoa-plane-2d[14]

Examples

```
d, selection, secondary_source = sfs.fd.nfchoa.plane_2d(
    omega, array.x, R, npw)
plot(d, selection, secondary_source)
```



sfs.fd.nfchoa.**point_25d**(*omega*, *xo*, *ro*, *xs*, ***, *max_order=None*, *c=None*)

Driving function for 2.5-dimensional NFC-HOA for a virtual point source.

Parameters

- **omega** (*float*) – Angular frequency of point source.
- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **ro** (*float*) – Radius of circular secondary source distribution.
- **xs** (*(3,) array_like*) – Position of point source.

[14] https://sfs.readthedocs.io/en/3.2/d_nfchoa/#equation-fd-nfchoa-plane-2d

- **max_order** (*float, optional*) – Maximum order of circular harmonics used for the calculation.
- **c** (*float, optional*) – Speed of sound.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing only `True` indicating that all secondary source are "active" for NFC-HOA.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

$$D(\phi_0, \omega) = \frac{1}{2\pi r_0} \sum_{m=-M}^{M} \frac{h_{|m|}^{(2)}\left(\frac{\omega}{c} r\right)}{h_{|m|}^{(2)}\left(\frac{\omega}{c} r_0\right)} \mathrm{e}^{\mathrm{i}m(\phi_0 - \phi)}$$

See https://sfs.rtfd.io/d_nfchoa/#equation-fd-nfchoa-point-25d[15]

Examples

```
d, selection, secondary_source = sfs.fd.nfchoa.point_25d(
    omega, array.x, R, xs)
plot(d, selection, secondary_source)
```

`sfs.fd.nfchoa.`**`plane_25d`**(*omega, xo, ro, n=[0, 1, 0], *, max_order=None, c=None*)

Driving function for 2.5-dimensional NFC-HOA for a virtual plane wave.

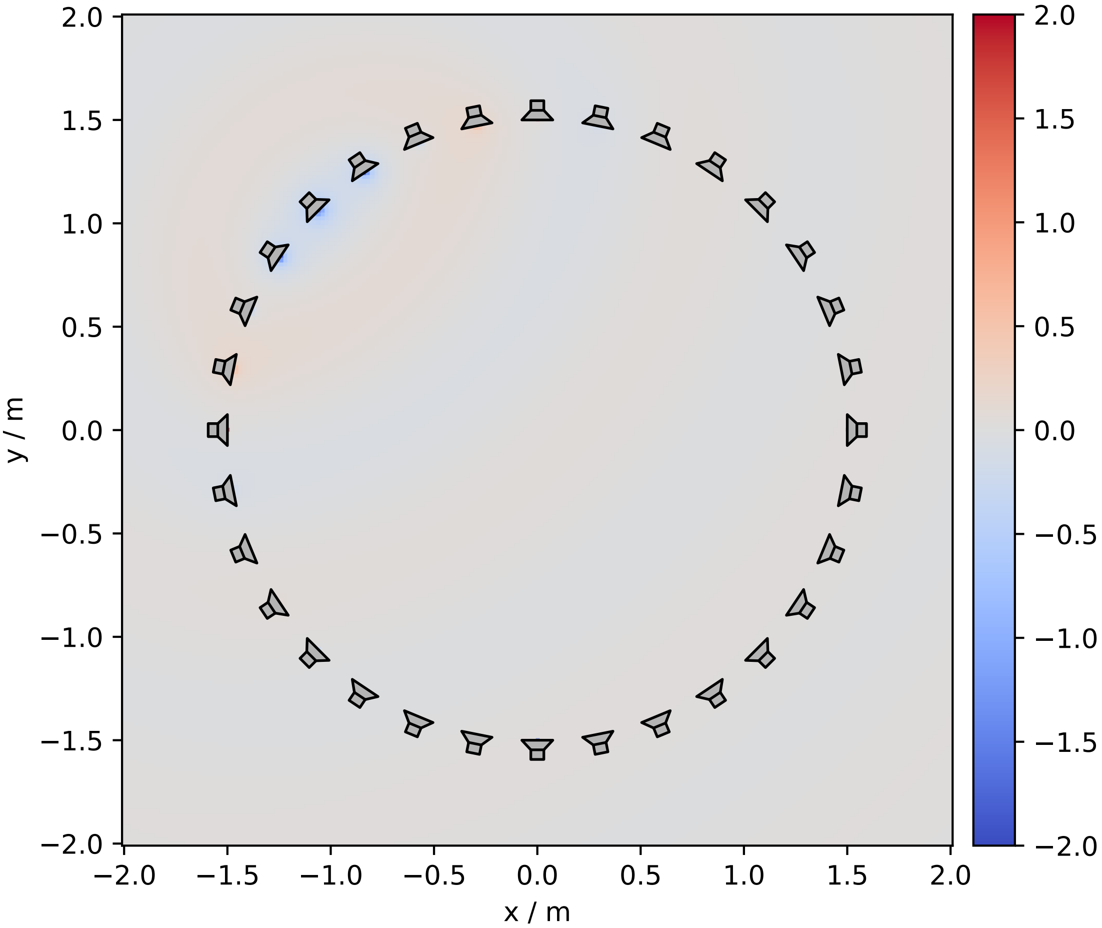
Parameters

- **omega** (*float*) – Angular frequency of point source.
- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **ro** (*float*) – Radius of circular secondary source distribution.
- **n** (*(3,) array_like, optional*) – Normal vector (traveling direction) of plane wave.
- **max_order** (*float, optional*) – Maximum order of circular harmonics used for the calculation.
- **c** (*float, optional*) – Speed of sound.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing only `True` indicating that all secondary source are "active" for NFC-HOA.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

[15] https://sfs.readthedocs.io/en/3.2/d_nfchoa/#equation-fd-nfchoa-point-25d


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y / m
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x / m
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0

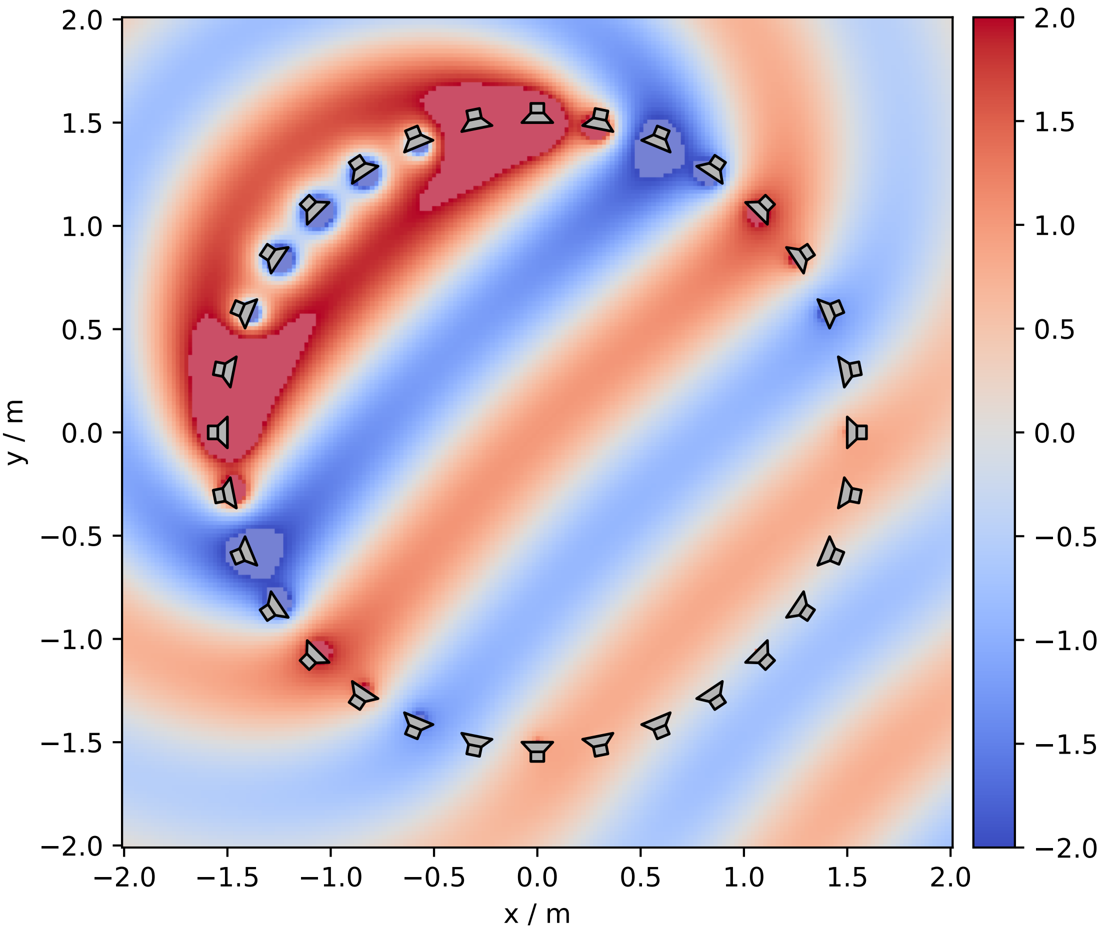
Notes

$$D(\phi_0, \omega) = \frac{2\mathrm{i}}{r_0} \sum_{m=-M}^{M} \frac{\mathrm{i}^{-|m|}}{\frac{\omega}{c} h_{|m|}^{(2)}\left(\frac{\omega}{c} r_0\right)} \mathrm{e}^{\mathrm{i}m(\phi_0 - \phi_{\mathrm{pw}})}$$

See https://sfs.rtfd.io/d_nfchoa/#equation-fd-nfchoa-plane-25d[16]

Examples

```
d, selection, secondary_source = sfs.fd.nfchoa.plane_25d(
    omega, array.x, R, npw)
plot(d, selection, secondary_source)
```



[16] https://sfs.readthedocs.io/en/3.2/d_nfchoa/#equation-fd-nfchoa-plane-25d

sfs.fd.sdm

Compute SDM driving functions.

```
import matplotlib.pyplot as plt
import numpy as np
import sfs

plt.rcParams['figure.figsize'] = 6, 6

xs = -1.5, 1.5, 0
# normal vector for plane wave:
npw = sfs.util.direction_vector(np.radians(-45))
f = 300  # Hz
omega = 2 * np.pi * f

grid = sfs.util.xyz_grid([-2, 2], [-2, 2], 0, spacing=0.02)

array = sfs.array.linear(32, 0.2, orientation=[0, -1, 0])

def plot(d, selection, secondary_source):
    p = sfs.fd.synthesize(d, selection, array, secondary_source, grid=grid)
    sfs.plot2d.amplitude(p, grid)
    sfs.plot2d.loudspeakers(array.x, array.n, selection * array.a, size=0.15)
```

Functions

line_2d(omega, x0, n0, xs, *[, c])	Driving function for 2-dimensional SDM for a virtual line source.
plane_25d(omega, x0, n0[, n, xref, c])	Driving function for 2.5-dimensional SDM for a virtual plane wave.
plane_2d(omega, x0, n0[, n, c])	Driving function for 2-dimensional SDM for a virtual plane wave.
point_25d(omega, x0, n0, xs, *[, xref, c])	Driving function for 2.5-dimensional SDM for a virtual point source.

`sfs.fd.sdm.`**`line_2d`**(*omega, x0, n0, xs, *, c=None*)

Driving function for 2-dimensional SDM for a virtual line source.

Parameters

- **omega** (*float*) – Angular frequency of line source.
- **x0** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **n0** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **xs** (*(3,) array_like*) – Position of line source.
- **c** (*float, optional*) – Speed of sound.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.

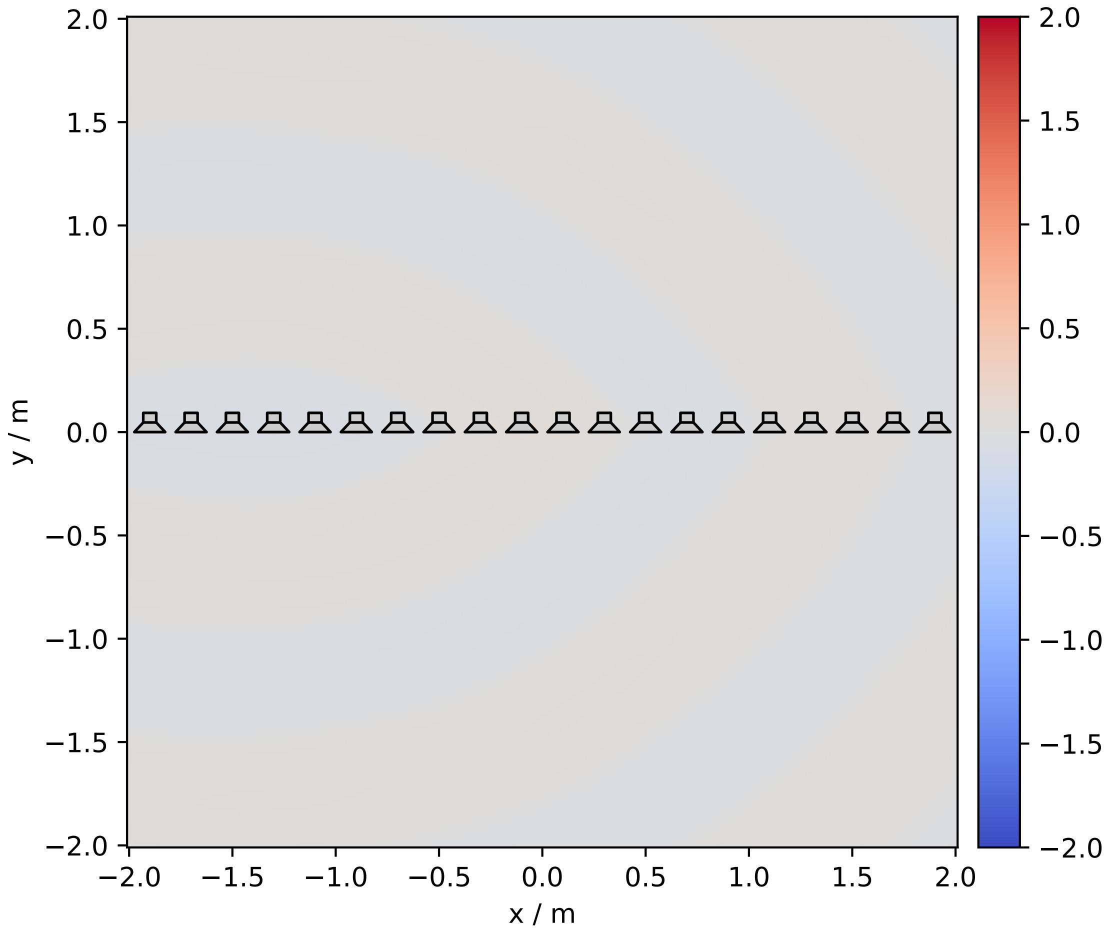
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See *sfs.fd.synthesize()*.

Notes

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

Examples

```
d, selection, secondary_source = sfs.fd.sdm.line_2d(
    omega, array.x, array.n, xs)
plot(d, selection, secondary_source)
```



sfs.fd.sdm.**plane_2d**(*omega, x0, n0, n=[0, 1, 0], *, c=None*)

Driving function for 2-dimensional SDM for a virtual plane wave.

Parameters

- **omega** (*float*) – Angular frequency of plane wave.
- **x0** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **n0** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **n** (*(3,) array_like, optional*) – Normal vector (traveling direction) of plane wave.

- **c** (*float, optional*) – Speed of sound.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

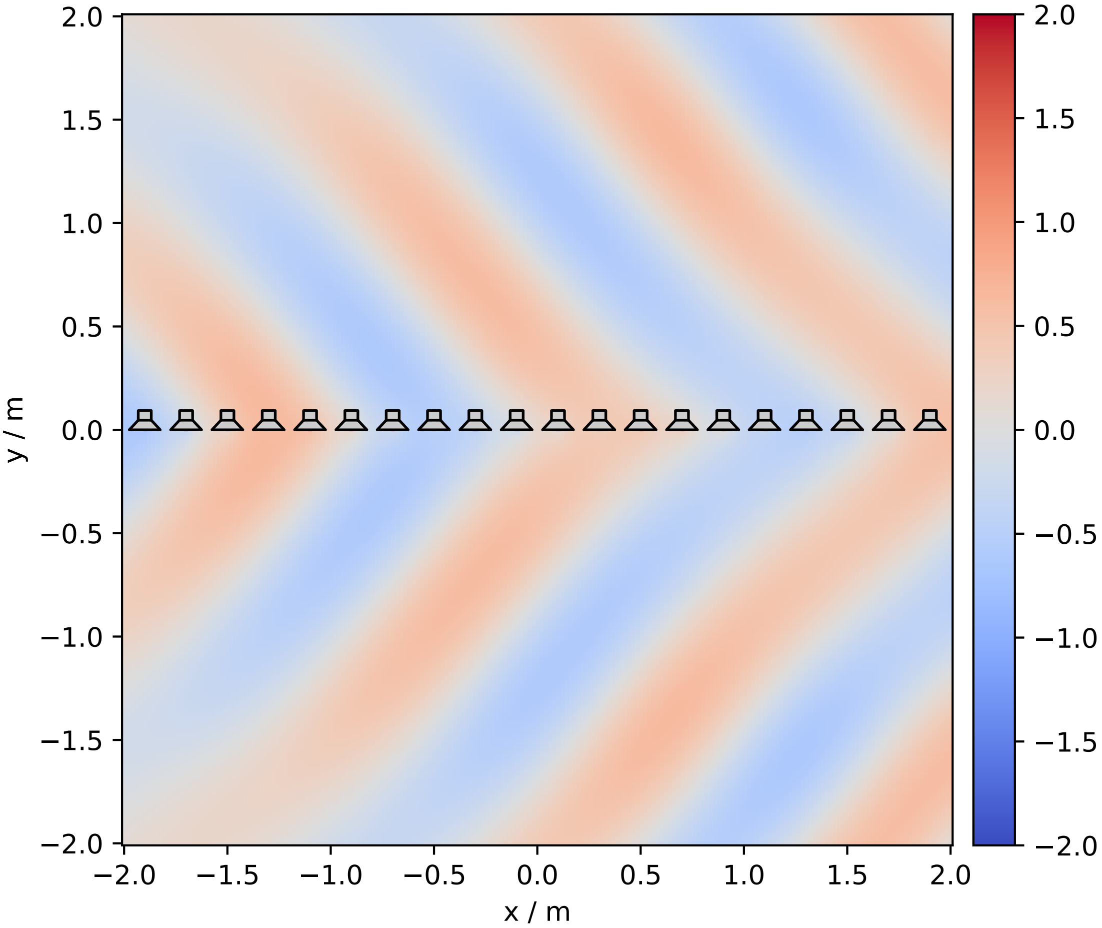
Notes

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):

$$D(\mathbf{x}_0, k) = k_{\text{pw},y}\, \mathrm{e}^{-\mathrm{i}k_{\text{pw},x}x}$$

Examples

```
d, selection, secondary_source = sfs.fd.sdm.plane_2d(
    omega, array.x, array.n, npw)
plot(d, selection, secondary_source)
```



`sfs.fd.sdm.`**`plane_25d`**(*omega, x0, n0, n=[0, 1, 0], *, xref=[0, 0, 0], c=None*)
Driving function for 2.5-dimensional SDM for a virtual plane wave.

Parameters

- **omega** (*float*) – Angular frequency of plane wave.
- **x0** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **n0** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **n** (*(3,) array_like, optional*) – Normal vector (traveling direction) of plane wave.
- **xref** (*(3,) array_like, optional*) – Reference point for synthesized sound field.
- **c** (*float, optional*) – Speed of sound.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

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

Examples

```
d, selection, secondary_source = sfs.fd.sdm.plane_25d(
    omega, array.x, array.n, npw, xref=[0, -1, 0])
plot(d, selection, secondary_source)
```

`sfs.fd.sdm.`**`point_25d`**(*omega, x0, n0, xs, *, xref=[0, 0, 0], c=None*)

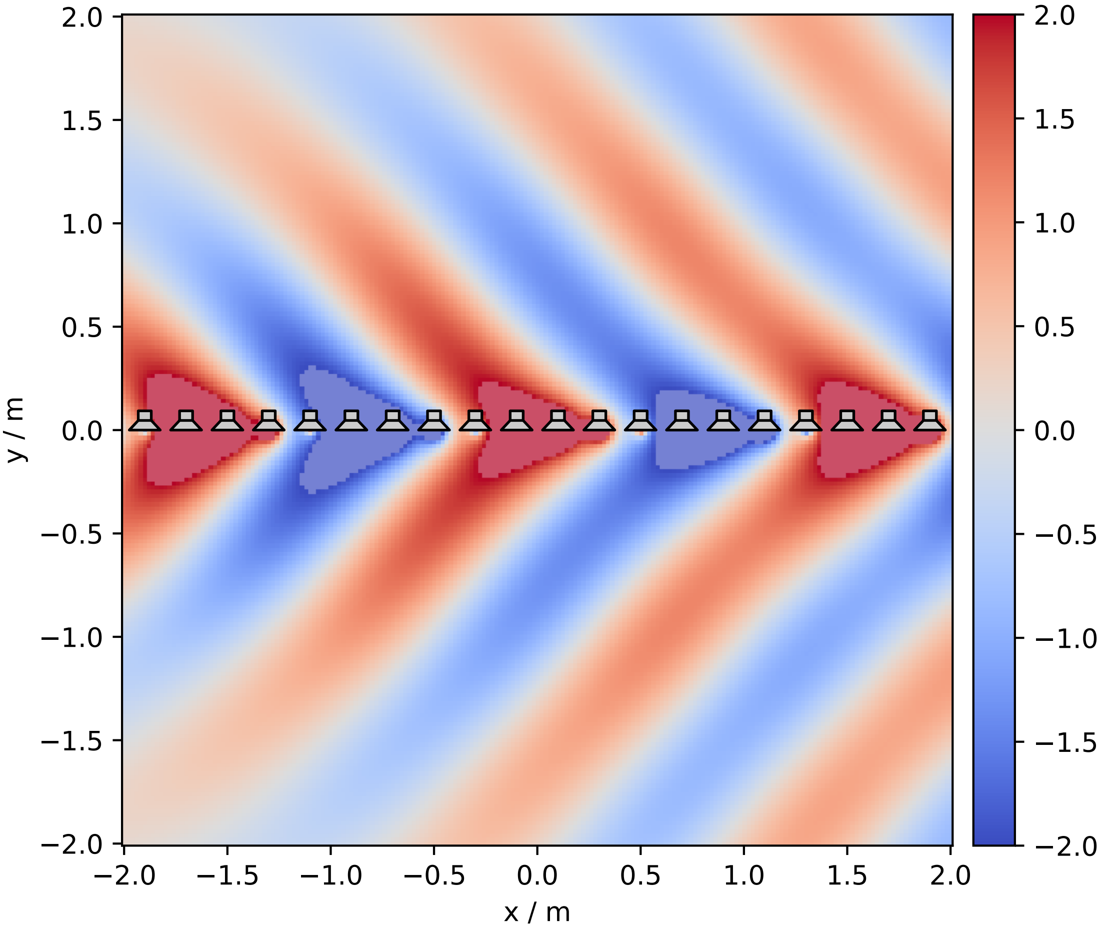
Driving function for 2.5-dimensional SDM for a virtual point source.

Parameters

- **omega** (*float*) – Angular frequency of point source.
- **x0** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **n0** (*(N, 3) array_like*) – Sequence of normal vectors of secondary sources.
- **xs** (*(3,) array_like*) – Position of virtual point source.
- **xref** (*(3,) array_like, optional*) – Reference point for synthesized sound field.
- **c** (*float, optional*) – Speed of sound.

Returns

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

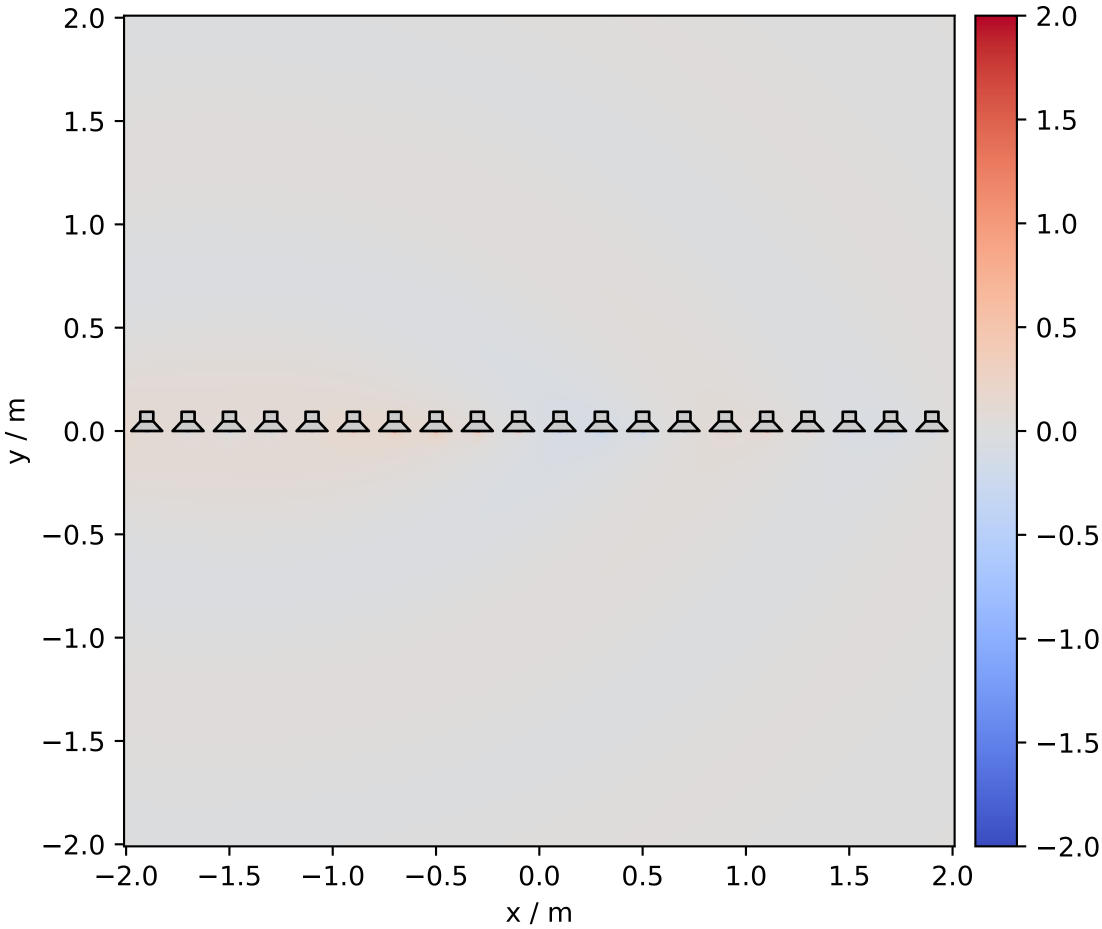

2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y / m
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x / m
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0

Notes

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

Examples

```
d, selection, secondary_source = sfs.fd.sdm.point_25d(
    omega, array.x, array.n, xs, xref=[0, -1, 0])
plot(d, selection, secondary_source)
```



sfs.fd.esa

Compute ESA driving functions for various systems.

ESA is abbreviation for equivalent scattering approach.

ESA driving functions for an edge-shaped SSD are provided below. Further ESA for different geometries might be added here.

Note that mode-matching (such as NFC-HOA, SDM) are equivalent to ESA in their specific geometries (spherical/circular, planar/linear).

Functions

`line_2d_edge`(omega, xo, xs, *[, alpha, Nc, c])	Driving function for 2-dimensional line source with edge ESA.
`line_2d_edge_dipole_ssd`(omega, xo, xs, *[, ...])	Driving function for 2-dimensional line source with edge dipole ESA.
`plane_2d_edge`(omega, xo[, n, alpha, Nc, c])	Driving function for 2-dimensional plane wave with edge ESA.
`plane_2d_edge_dipole_ssd`(omega, xo[, n, ...])	Driving function for 2-dimensional plane wave with edge dipole ESA.
`point_25d_edge`(omega, xo, xs, *[, xref, ...])	Driving function for 2.5-dimensional point source with edge ESA.

`sfs.fd.esa.`**`plane_2d_edge`**(*omega, xo, n=[0, 1, 0], *, alpha=4.71238898038469, Nc=None, c=None*)

Driving function for 2-dimensional plane wave with edge ESA.

Driving function for a virtual plane wave using the 2-dimensional ESA for an edge-shaped secondary source distribution consisting of monopole line sources.

Parameters

- **omega** (*float*) – Angular frequency.
- **xo** (*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

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

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

Derived from [SSR16]

`sfs.fd.esa.`**`plane_2d_edge_dipole_ssd`**(*omega, xo, n=[0, 1, 0], *, alpha=4.71238898038469, Nc=None, c=None*)

Driving function for 2-dimensional plane wave with edge dipole ESA.

Driving function for a virtual plane wave using the 2-dimensional ESA for an edge-shaped secondary source distribution consisting of dipole line sources.

Parameters

- **omega** (*float*) – Angular frequency.
- **xo** (*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

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

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

Derived from [SSR16]

`sfs.fd.esa.`**`line_2d_edge`**(*omega, x0, xs, *, alpha=4.71238898038469, Nc=None, c=None*)

Driving function for 2-dimensional line source with edge ESA.

Driving function for a virtual line source using the 2-dimensional ESA for an edge-shaped secondary source distribution consisting of line sources.

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

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

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

Derived from [SSR16]

`sfs.fd.esa.`**`line_2d_edge_dipole_ssd`**(*omega*, *x0*, *xs*, ***, *alpha=4.71238898038469*, *Nc=None*, *c=None*)

Driving function for 2-dimensional line source with edge dipole ESA.

Driving function for a virtual line source using the 2-dimensional ESA for an edge-shaped secondary source distribution consisting of dipole line sources.

Parameters

- **omega** (*float*) – Angular frequency.
- **x0** (*(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

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

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

Derived from [SSR16]

`sfs.fd.esa.`**`point_25d_edge`**(*omega*, *x0*, *xs*, ***, *xref=[2, - 2, 0]*, *alpha=4.71238898038469*, *Nc=None*, *c=None*)

Driving function for 2.5-dimensional point source with edge ESA.

Driving function for a virtual point source using the 2.5-dimensional ESA for an edge-shaped secondary source distribution consisting of point sources.

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.
- **xref** (*(3,) array_like or float*) – Reference position or reference distance
- **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

- **d** (*(N,) numpy.ndarray*) – Complex weights of secondary sources.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.fd.synthesize()`.

Notes

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

Derived from [SSR16]

Functions

`displacement`(v, omega)	Particle displacement.
`secondary_source_line`(omega, c)	Create a line source for use in `sfs.fd.synthesize()`.
`secondary_source_point`(omega, c)	Create a point source for use in `sfs.fd.synthesize()`.
`shiftphase`(p, phase)	Shift phase of a sound field.
`synthesize`(d, weights, ssd, ...)	Compute sound field for a generic driving function.

`sfs.fd.`**`shiftphase`**(*p, phase*)

Shift phase of a sound field.

`sfs.fd.`**`displacement`**(*v, omega*)

Particle displacement.

$$d(x,t) = \int_{-\infty}^{t} v(x,\tau) d\tau$$

`sfs.fd.`**`synthesize`**(*d, weights, ssd, secondary_source_function, **kwargs*)

Compute sound field for a generic driving function.

Parameters

- **d** (*array_like*) – Driving function.
- **weights** (*array_like*) – Additional weights applied during integration, e.g. source selection and tapering.
- **ssd** (*sequence of between 1 and 3 array_like objects*) – Positions, normal vectors and weights of secondary sources. A `SecondarySourceDistribution` can also be used.
- **secondary_source_function** (*callable*) – A function that generates the sound field of a secondary source. This signature is expected:

```
secondary_source_function(
    position, normal_vector, **kwargs) -> numpy.ndarray
```

- ****kwargs** – All keyword arguments are forwarded to *secondary_source_function*. This is typically used to pass the *grid* argument.

`sfs.fd.`**`secondary_source_point`**`(`*omega, c*`)`

Create a point source for use in *`sfs.fd.synthesize()`*.

`sfs.fd.`**`secondary_source_line`**`(`*omega, c*`)`

Create a line source for use in *`sfs.fd.synthesize()`*.

3.2 sfs.td

Submodules for broadband sound fields.

source	Compute the sound field generated by a sound source.
wfs	Compute WFS driving functions.
nfchoa	Compute NFC-HOA driving functions.

sfs.td.source

Compute the sound field generated by a sound source.

The Green's function describes the spatial sound propagation over time.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.signal import unit_impulse
import sfs

xs = 1.5, 1, 0  # source position
rs = np.linalg.norm(xs)  # distance from origin
ts = rs / sfs.default.c  # time-of-arrival at origin

# Impulsive excitation
fs = 44100
signal = unit_impulse(512), fs

grid = sfs.util.xyz_grid([-2, 3], [-1, 2], 0, spacing=0.02)
```

Functions

point(xs, signal, observation_time, grid[, c])	Source model for a point source: 3D Green's function.
point_image_sources(xo, signal, . . . [, coeffs, c])	Point source in a rectangular room using the mirror image source model.

`sfs.td.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.

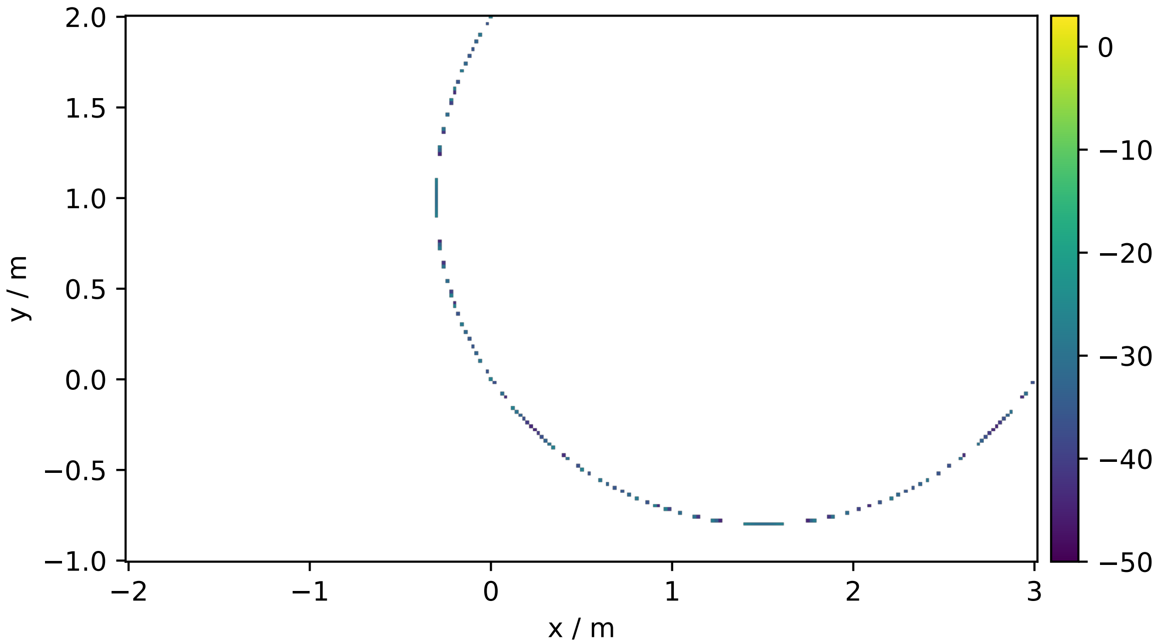
Returns *numpy.ndarray* – Scalar sound pressure field, evaluated at positions given by *grid*.

Notes

$$g(x - x_s, t) = \frac{1}{4\pi|x - x_s|} \delta \left(t - \frac{|x - x_s|}{c} \right)$$

Examples

```
p = sfs.td.source.point(xs, signal, ts, grid)
sfs.plot2d.level(p, grid)
```



`sfs.td.source.point_image_sources`(*xo*, *signal*, *observation_time*, *grid*, *L*, *max_order*, *coeffs=None*, *c=None*)

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

Parameters

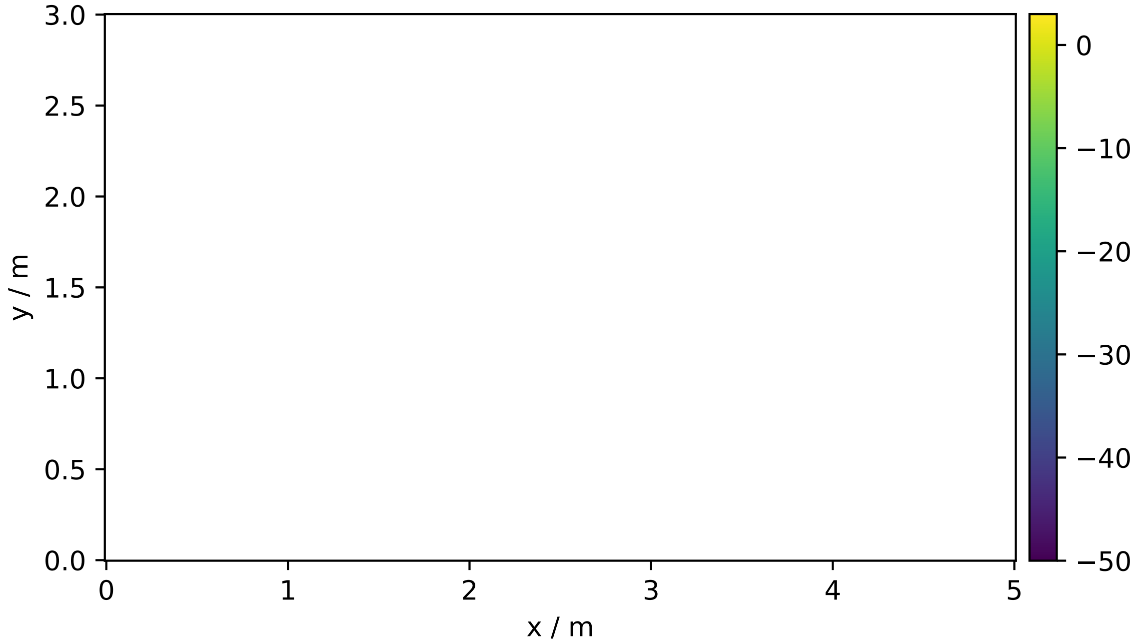
- **xo** (*(3,) array_like*) – Position of source in cartesian coordinates.
- **signal** (*(N,) array_like + float*) – Excitation signal consisting of (mono) audio data and a sampling rate (in Hertz). A `DelayedSignal` object can also be used.
- **observation_time** (*float*) – Observed point in time.

- **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*.

Examples

```
room = 5, 3, 1.5  # room dimensions
order = 2  # image source order
coeffs = .8, .8, .6, .6, .7, .7  # wall reflection coefficients
grid = sfs.util.xyz_grid([0, room[0]], [0, room[1]], 0, spacing=0.01)
p = sfs.td.source.point_image_sources(
        xs, signal, 1.5 * ts, grid, room, order, coeffs)
sfs.plot2d.level(p, grid)
```



sfs.td.wfs

Compute WFS driving functions.

```
import matplotlib.pyplot as plt
import numpy as np
import sfs
from scipy.signal import unit_impulse

# Plane wave
npw = sfs.util.direction_vector(np.radians(-45))

# Point source
xs = -1.5, 1.5, 0
rs = np.linalg.norm(xs)  # distance from origin
ts = rs / sfs.default.c  # time-of-arrival at origin

# Focused source
xf = -0.5, 0.5, 0
nf = sfs.util.direction_vector(np.radians(-45))  # normal vector
rf = np.linalg.norm(xf)  # distance from origin
tf = rf / sfs.default.c  # time-of-arrival at origin

# Impulsive excitation
fs = 44100
signal = unit_impulse(512), fs

# Circular loudspeaker array
N = 32  # number of loudspeakers
R = 1.5  # radius
array = sfs.array.circular(N, R)

grid = sfs.util.xyz_grid([-2, 2], [-2, 2], 0, spacing=0.02)

def plot(d, selection, secondary_source, t=0):
    p = sfs.td.synthesize(d, selection, array, secondary_source, grid=grid,
                          observation_time=t)
    sfs.plot2d.level(p, grid)
    sfs.plot2d.loudspeakers(array.x, array.n,
                            selection * array.a, size=0.15)
```

Functions

driving_signals(delays, weights, signal)	Get driving signals per secondary source.
focused_25d(xo, no, xs, ns[, xref, c])	Point source by 2.5-dimensional WFS.
plane_25d(xo, no[, n, xref, c])	Plane wave model by 2.5-dimensional WFS.
point_25d(xo, no, xs[, xref, c])	Driving function for 2.5-dimensional WFS of a virtual point source.
point_25d_legacy(xo, no, xs[, xref, c])	Driving function for 2.5-dimensional WFS of a virtual point source.

`sfs.td.wfs.`**`plane_25d`**(*xo, no, n=[0, 1, 0], xref=[0, 0, 0], c=None*)

Plane wave model by 2.5-dimensional WFS.

Parameters

- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **no** (*(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.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.td.synthesize()`.

Notes

2.5D correction factor

$$g_0 = \sqrt{2\pi |x_\mathrm{ref} - x_0|}$$

d using a plane wave as source model

$$d_{2.5D}(x_0,t) = 2g_0 \langle n, n_0 \rangle \delta \left(t - \frac{1}{c} \langle n, x_0 \rangle \right) *_t h(t)$$

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

See https://sfs.rtfd.io/d_wfs/#equation-td-wfs-plane-25d[17]

Examples

```
delays, weights, selection, secondary_source = \
    sfs.td.wfs.plane_25d(array.x, array.n, npw)
d = sfs.td.wfs.driving_signals(delays, weights, signal)
plot(d, selection, secondary_source)
```

`sfs.td.wfs.`**`point_25d`**`(`*xo, no, xs, xref=[0, 0, 0], c=None*`)`

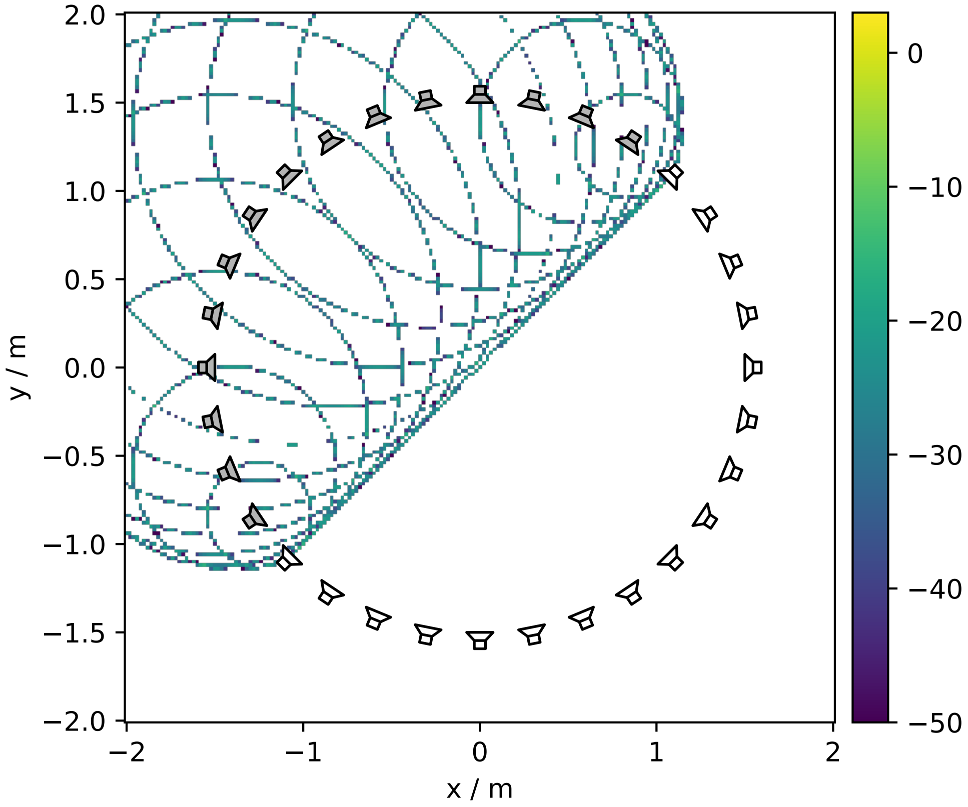
Driving function for 2.5-dimensional WFS of a virtual point source.

Changed in version 0.61: see notes, old handling of `point_25d()` is now `point_25d_legacy()`

Parameters

- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **no** (*(N, 3) array_like*) – Sequence of secondary source orientations.
- **xs** (*(3,) array_like*) – Virtual source position.
- **xref** (*(N, 3) array_like or (3,) array_like*) – Reference curve of correct amplitude xref(xo)

[17] https://sfs.readthedocs.io/en/3.2/d_wfs/#equation-td-wfs-plane-25d



- **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.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See *sfs.td.synthesize()*.

Notes

Eq. (2.138) in [Sch16]:

$$d_{2.5D}(x_0, x_{ref}, t) = \sqrt{8\pi}\frac{\langle(x_0 - x_s), n_0\rangle}{|x_0 - x_s|}\sqrt{\frac{|x_0 - x_s||x_0 - x_{ref}|}{|x_0 - x_s| + |x_0 - x_{ref}|}} \cdot \frac{\delta\left(t - \frac{|x_0 - x_s|}{c}\right)}{4\pi|x_0 - x_s|} *_t h(t)$$

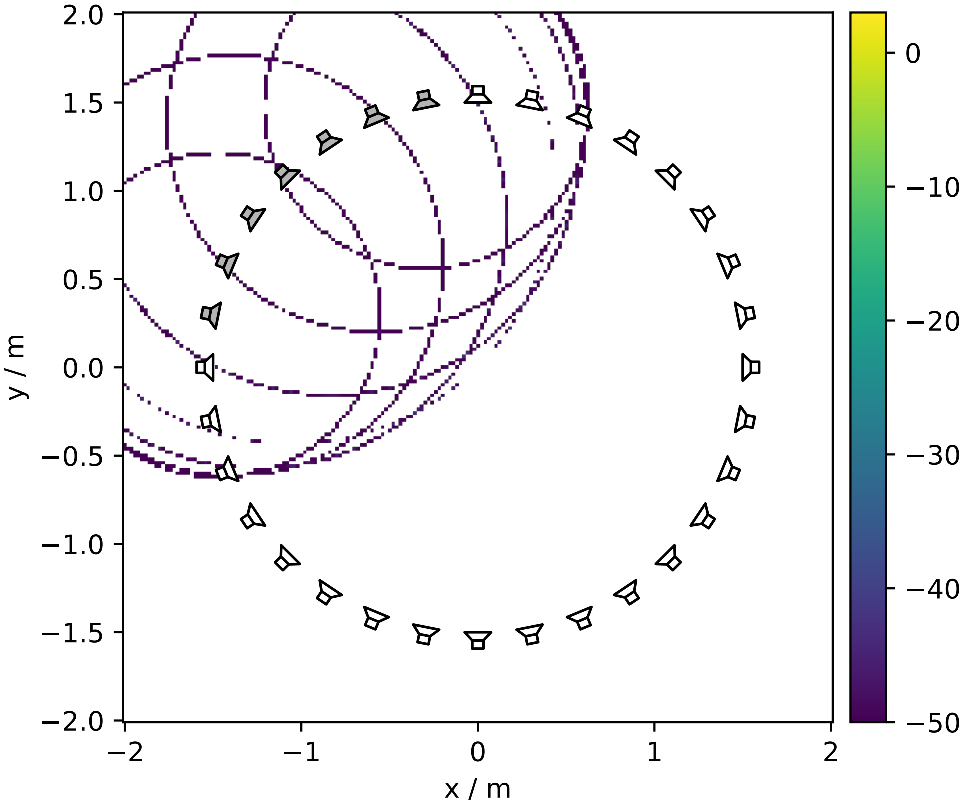
$$h(t) = F^{-1}(\sqrt{\frac{j\omega}{c}})$$

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

point_25d() derives WFS from 3D to 2.5D via the stationary phase approximation approach (i.e. the Delft approach). The theoretical link of *point_25d()* and *point_25d_legacy()* was introduced as *unified WFS framework* in [FFSS17].

Examples

```
delays, weights, selection, secondary_source = \
    sfs.td.wfs.point_25d(array.x, array.n, xs)
d = sfs.td.wfs.driving_signals(delays, weights, signal)
plot(d, selection, secondary_source, t=ts)
```



`sfs.td.wfs.`**`point_25d_legacy`**(*xo*, *no*, *xs*, *xref=[0, 0, 0]*, *c=None*)

Driving function for 2.5-dimensional WFS of a virtual point source.

New in version 0.61: `point_25d()` was renamed to `point_25d_legacy()` (and a new function with the name `point_25d()` was introduced). See notes below for further details.

Parameters

- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **no** (*(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.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.

- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.td.synthesize()`.

Notes

2.5D correction factor

$$g_0 = \sqrt{2\pi|x_\mathrm{ref} - x_0|}$$

d using a point source as source model

$$d_{2.5D}(x_0, t) = \frac{g_0 \left\langle (x_0 - x_s), n_0 \right\rangle}{2\pi |x_0 - x_s|^{3/2}} \delta\left(t - \frac{|x_0 - x_s|}{c}\right) *_t h(t)$$

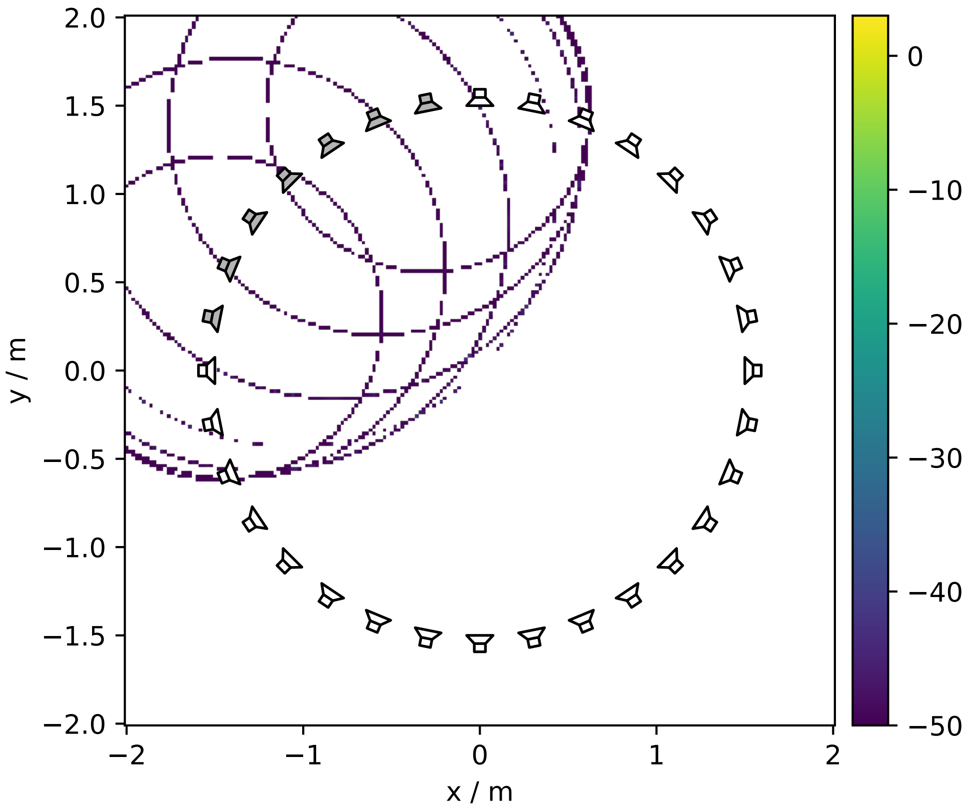
with wfs(2.5D) prefilter h(t), which is not implemented yet.

See https://sfs.rtfd.io/d_wfs/#equation-td-wfs-point-25d[18]

`point_25d_legacy()` derives 2.5D WFS from the 2D Neumann-Rayleigh integral (i.e. the approach by Rabenstein & Spors), cf. [SRA08]. The theoretical link of `point_25d()` and `point_25d_legacy()` was introduced as *unified WFS framework* in [FFSS17].

Examples

```
delays, weights, selection, secondary_source = \
    sfs.td.wfs.point_25d(array.x, array.n, xs)
d = sfs.td.wfs.driving_signals(delays, weights, signal)
plot(d, selection, secondary_source, t=ts)
```



[18] https://sfs.readthedocs.io/en/3.2/d_wfs/#equation-td-wfs-point-25d

`sfs.td.wfs.`**`focused_25d`**(*xo, no, xs, ns, xref=[0, 0, 0], c=None*)

Point source by 2.5-dimensional WFS.

Parameters

- **xo** ((*N, 3*) *array_like*) – Sequence of secondary source positions.
- **no** ((*N, 3*) *array_like*) – Sequence of secondary source orientations.
- **xs** ((*3,*) *array_like*) – Virtual source position.
- **ns** ((*3,*) *array_like*) – Normal vector (propagation direction) of focused source. This is used for secondary source selection, see `sfs.util.source_selection_focused()`.
- **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.
- **selection** ((*N,*) *numpy.ndarray*) – Boolean array containing `True` or `False` depending on whether the corresponding secondary source is "active" or not.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.td.synthesize()`.

Notes

2.5D correction factor

$$g_0 = \sqrt{\frac{|x_\mathrm{ref} - x_0|}{|x_0 - x_s| + |x_\mathrm{ref} - x_0|}}$$

d using a point source as source model

$$d_{2.5D}(x_0, t) = \frac{g_0 \langle (x_0 - x_s), n_0 \rangle}{|x_0 - x_s|^{3/2}} \delta \left(t + \frac{|x_0 - x_s|}{c} \right) *_t h(t)$$

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

See https://sfs.rtfd.io/d_wfs/#equation-td-wfs-focused-25d[19]

Examples

```
delays, weights, selection, secondary_source = \
    sfs.td.wfs.focused_25d(array.x, array.n, xf, nf)
d = sfs.td.wfs.driving_signals(delays, weights, signal)
plot(d, selection, secondary_source, t=tf)
```

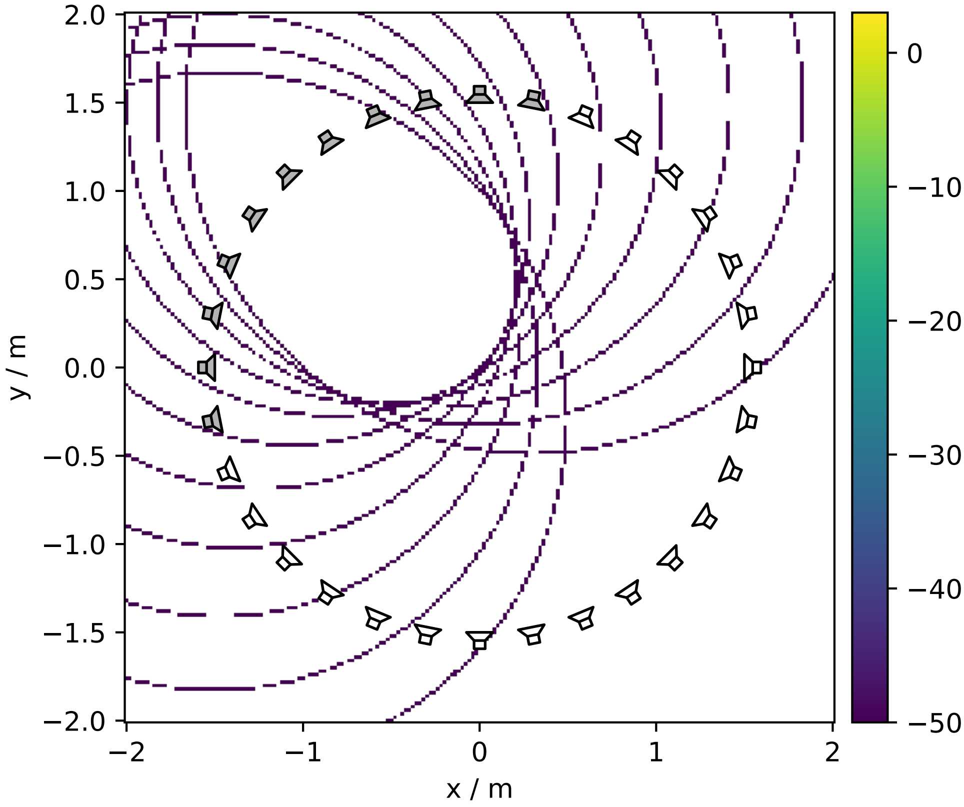
`sfs.td.wfs.`**`driving_signals`**(*delays, weights, signal*)

Get driving signals per secondary source.

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

Parameters

[19] https://sfs.readthedocs.io/en/3.2/d_wfs/#equation-td-wfs-focused-25d



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

Returns `DelayedSignal` – A tuple containing the driving signals (in a `numpy.ndarray`[20] with shape `(N, C)`), followed by the sampling rate (in Hertz) and a (possibly negative) time offset (in seconds).

sfs.td.nfchoa

Compute NFC-HOA driving functions.

```
import matplotlib.pyplot as plt
import numpy as np
import sfs
from scipy.signal import unit_impulse

# Plane wave
npw = sfs.util.direction_vector(np.radians(-45))

# Point source
xs = -1.5, 1.5, 0
rs = np.linalg.norm(xs)  # distance from origin
```

(continues on next page)

[20] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray

(continued from previous page)

```
ts = rs / sfs.default.c  # time-of-arrival at origin

# Impulsive excitation
fs = 44100
signal = unit_impulse(512), fs

# Circular loudspeaker array
N = 32  # number of loudspeakers
R = 1.5  # radius
array = sfs.array.circular(N, R)

grid = sfs.util.xyz_grid([-2, 2], [-2, 2], 0, spacing=0.02)

def plot(d, selection, secondary_source, t=0):
    p = sfs.td.synthesize(d, selection, array, secondary_source, grid=grid,
                          observation_time=t)
    sfs.plot2d.level(p, grid)
    sfs.plot2d.loudspeakers(array.x, array.n, selection * array.a, size=0.15)
```

Functions

driving_signals_25d(delay, weight, sos, ...)	Get 2.5-dimensional NFC-HOA driving signals.
driving_signals_3d(delay, weight, sos, ...)	Get 3-dimensional NFC-HOA driving signals.
matchedz_zpk(s_zeros, s_poles, s_gain, fs)	Matched-z transform of poles and zeros.
plane_25d(xo, ro, npw, fs[, max_order, c, s2z])	Virtual plane wave by 2.5-dimensional NFC-HOA.
plane_3d(xo, ro, npw, fs[, max_order, c, s2z])	Virtual plane wave by 3-dimensional NFC-HOA.
point_25d(xo, ro, xs, fs[, max_order, c, s2z])	Virtual Point source by 2.5-dimensional NFC-HOA.
point_3d(xo, ro, xs, fs[, max_order, c, s2z])	Virtual point source by 3-dimensional NFC-HOA.

`sfs.td.nfchoa.`**`matchedz_zpk`**(*s_zeros*, *s_poles*, *s_gain*, *fs*)

Matched-z transform of poles and zeros.

Parameters

- **s_zeros** (*array_like*) – Zeros in the Laplace domain.
- **s_poles** (*array_like*) – Poles in the Laplace domain.
- **s_gain** (*float*) – System gain in the Laplace domain.
- **fs** (*int*) – Sampling frequency in Hertz.

Returns

- **z_zeros** (*numpy.ndarray*) – Zeros in the z-domain.
- **z_poles** (*numpy.ndarray*) – Poles in the z-domain.
- **z_gain** (*float*) – System gain in the z-domain.

See also:

`scipy.signal.bilinear_zpk()`[21]

[21] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.bilinear_zpk.html#scipy.signal.bilinear_zpk

`sfs.td.nfchoa.`**`plane_25d`**(*xo, ro, npw, fs, max_order=None, c=None, s2z=<function matchedz_zpk>*)

Virtual plane wave by 2.5-dimensional NFC-HOA.

$$D(\phi_0, s) = 2\,\mathrm{e}^{\frac{s}{c} r_0} \sum_{m=-M}^{M} (-1)^m \left(\frac{s}{s - \frac{c}{r_0}\sigma_0} \right)^{\mu} \prod_{l=1}^{\nu} \frac{s^2}{(s - \frac{c}{r_0}\sigma_l)^2 + (\frac{c}{r_0}\omega_l)^2} \, \mathrm{e}^{\mathrm{i}m(\phi_0 - \phi_{\mathrm{pw}})}$$

The driving function is represented in the Laplace domain, from which the recursive filters are designed. $\sigma_l + \mathrm{i}\omega_l$ denotes the complex roots of the reverse Bessel polynomial. The number of second-order sections is $\nu = \big\lfloor \frac{|m|}{2} \big\rfloor$, whereas the number of first-order section μ is either 0 or 1 for even and odd $|m|$, respectively.

Parameters

- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **ro** (*float*) – Radius of the circular secondary source distribution.
- **npw** (*(3,) array_like*) – Unit vector (propagation direction) of plane wave.
- **fs** (*int*) – Sampling frequency in Hertz.
- **max_order** (*int, optional*) – Ambisonics order.
- **c** (*float, optional*) – Speed of sound in m/s.
- **s2z** (*callable, optional*) – Function transforming s-domain poles and zeros into z-domain, e.g. `matchedz_zpk()`, `scipy.signal.bilinear_zpk()`[22].

Returns

- **delay** (*float*) – Overall delay in seconds.
- **weight** (*float*) – Overall weight.
- **sos** (*list of numpy.ndarray*) – Second-order section filters `scipy.signal.sosfilt()`[23].
- **phaseshift** (*(N,) numpy.ndarray*) – Phase shift in radians.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing only `True` indicating that all secondary source are "active" for NFC-HOA.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.td.synthesize()`.

Examples

```
delay, weight, sos, phaseshift, selection, secondary_source = \
    sfs.td.nfchoa.plane_25d(array.x, R, npw, fs)
d = sfs.td.nfchoa.driving_signals_25d(
        delay, weight, sos, phaseshift, signal)
plot(d, selection, secondary_source)
```

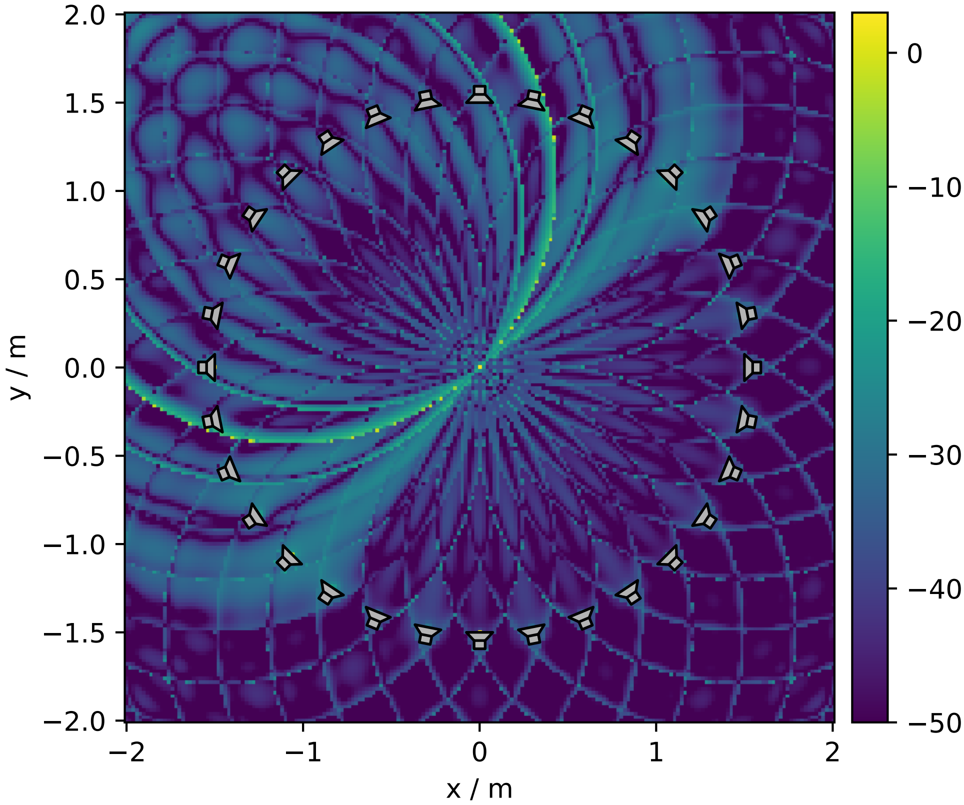
`sfs.td.nfchoa.`**`point_25d`**(*xo, ro, xs, fs, max_order=None, c=None, s2z=<function matchedz_zpk>*)

Virtual Point source by 2.5-dimensional NFC-HOA.

$$D(\phi_0, s) = \frac{1}{2\pi r_{\mathrm{s}}} \mathrm{e}^{\frac{s}{c}(r_0 - r_{\mathrm{s}})} \sum_{m=-M}^{M} \left(\frac{s - \frac{c}{r_{\mathrm{s}}}\sigma_0}{s - \frac{c}{r_0}\sigma_0} \right)^{\mu} \prod_{l=1}^{\nu} \frac{(s - \frac{c}{r_{\mathrm{s}}}\sigma_l)^2 - (\frac{c}{r_{\mathrm{s}}}\omega_l)^2}{(s - \frac{c}{r_0}\sigma_l)^2 + (\frac{c}{r_0}\omega_l)^2} \, \mathrm{e}^{\mathrm{i}m(\phi_0 - \phi_{\mathrm{s}})}$$

The driving function is represented in the Laplace domain, from which the recursive filters are designed. $\sigma_l + \mathrm{i}\omega_l$ denotes the complex roots of the reverse Bessel polynomial. The number of

[22] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.bilinear_zpk.html#scipy.signal.bilinear_zpk
[23] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.sosfilt.html#scipy.signal.sosfilt



second-order sections is $\nu = \lfloor \frac{|m|}{2} \rfloor$, whereas the number of first-order section μ is either 0 or 1 for even and odd $|m|$, respectively.

Parameters

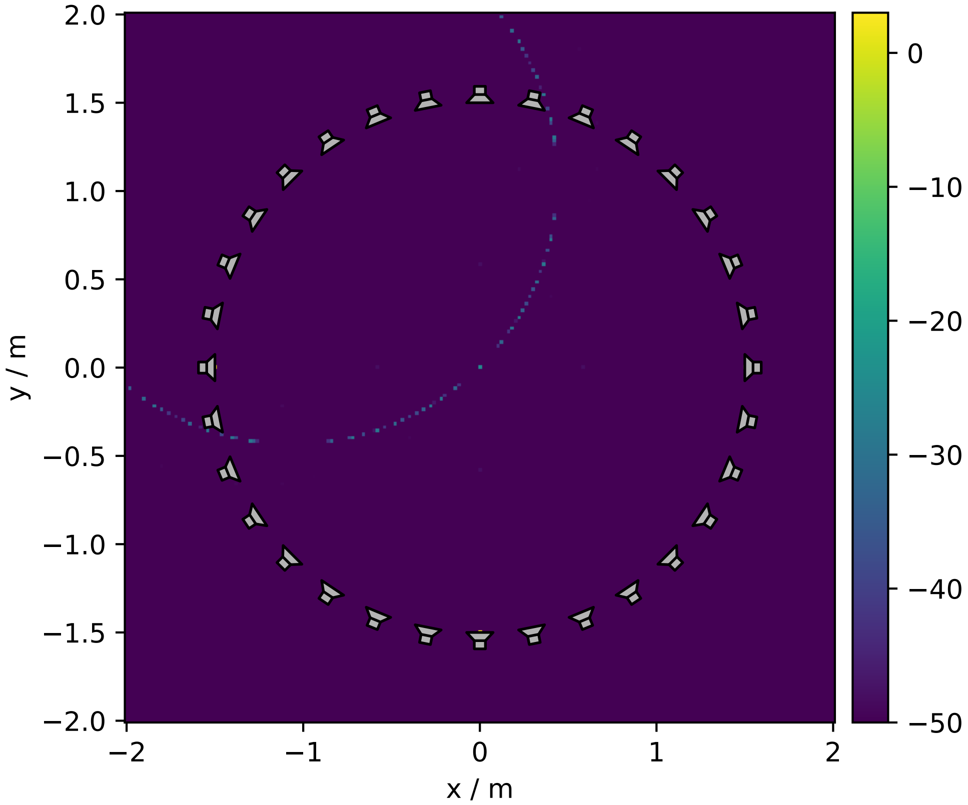
- **x0** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **r0** (*float*) – Radius of the circular secondary source distribution.
- **xs** (*(3,) array_like*) – Virtual source position.
- **fs** (*int*) – Sampling frequency in Hertz.
- **max_order** (*int, optional*) – Ambisonics order.
- **c** (*float, optional*) – Speed of sound in m/s.
- **s2z** (*callable, optional*) – Function transforming s-domain poles and zeros into z-domain, e.g. `matchedz_zpk()`, `scipy.signal.bilinear_zpk()`[24].

Returns

- **delay** (*float*) – Overall delay in seconds.
- **weight** (*float*) – Overall weight.
- **sos** (*list of numpy.ndarray*) – Second-order section filters `scipy.signal.sosfilt()`[25].
- **phaseshift** (*(N,) numpy.ndarray*) – Phase shift in radians.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing only `True` indicating that all secondary source are "active" for NFC-HOA.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.td.synthesize()`.

Examples

```
delay, weight, sos, phaseshift, selection, secondary_source = \
    sfs.td.nfchoa.point_25d(array.x, R, xs, fs)
d = sfs.td.nfchoa.driving_signals_25d(
        delay, weight, sos, phaseshift, signal)
plot(d, selection, secondary_source, t=ts)
```



`sfs.td.nfchoa.`**`plane_3d`**(*xo, ro, npw, fs, max_order=None, c=None, s2z=<function matchedz_zpk>*)

Virtual plane wave by 3-dimensional NFC-HOA.

$$D(\phi_0, s) = \frac{e^{\frac{s}{c} r_0}}{r_0} \sum_{n=0}^{N} (-1)^n (2n+1) P_n(\cos\Theta) \left(\frac{s}{s - \frac{c}{r_0}\sigma_0}\right)^{\mu} \prod_{l=1}^{\nu} \frac{s^2}{(s - \frac{c}{r_0}\sigma_l)^2 + (\frac{c}{r_0}\omega_l)^2}$$

The driving function is represented in the Laplace domain, from which the recursive filters are designed. $\sigma_l + i\omega_l$ denotes the complex roots of the reverse Bessel polynomial. The number of second-order sections is $\nu = \lfloor \frac{|m|}{2} \rfloor$, whereas the number of first-order section μ is either 0 or 1 for even and odd $|m|$, respectively. $P_n(\cdot)$ denotes the Legendre polynomial of degree n, and Θ the angle between (θ, ϕ) and (θ_{pw}, ϕ_{pw}).

Parameters

- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **ro** (*float*) – Radius of the spherical secondary source distribution.
- **npw** (*(3,) array_like*) – Unit vector (propagation direction) of plane wave.
- **fs** (*int*) – Sampling frequency in Hertz.

[24] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.bilinear_zpk.html#scipy.signal.bilinear_zpk
[25] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.sosfilt.html#scipy.signal.sosfilt

- **max_order** (*int, optional*) – Ambisonics order.
- **c** (*float, optional*) – Speed of sound in m/s.
- **s2z** (*callable, optional*) – Function transforming s-domain poles and zeros into z-domain, e.g. `matchedz_zpk()`, `scipy.signal.bilinear_zpk()`[26].

Returns

- **delay** (*float*) – Overall delay in seconds.
- **weight** (*float*) – Overall weight.
- **sos** (*list of numpy.ndarray*) – Second-order section filters `scipy.signal.sosfilt()`[27].
- **phaseshift** (*(N,) numpy.ndarray*) – Phase shift in radians.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing only `True` indicating that all secondary source are "active" for NFC-HOA.
- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.td.synthesize()`.

`sfs.td.nfchoa.`**`point_3d`**(*xo, ro, xs, fs, max_order=None, c=None, s2z=<function matchedz_zpk>*)

Virtual point source by 3-dimensional NFC-HOA.

$$D(\phi_0, s) = \frac{e^{\frac{s}{c}(r_0 - r_s)}}{4\pi r_0 r_s} \sum_{n=0}^{N} (2n+1) P_n(\cos\Theta) \left(\frac{s - \frac{c}{r_s}\sigma_0}{s - \frac{c}{r_0}\sigma_0}\right)^{\mu} \prod_{l=1}^{\nu} \frac{(s - \frac{c}{r_s}\sigma_l)^2 - (\frac{c}{r_s}\omega_l)^2}{(s - \frac{c}{r_0}\sigma_l)^2 + (\frac{c}{r_0}\omega_l)^2}$$

The driving function is represented in the Laplace domain, from which the recursive filters are designed. $\sigma_l + i\omega_l$ denotes the complex roots of the reverse Bessel polynomial. The number of second-order sections is $\nu = \lfloor \frac{|m|}{2} \rfloor$, whereas the number of first-order section μ is either 0 or 1 for even and odd $|m|$, respectively. $P_n(\cdot)$ denotes the Legendre polynomial of degree n, and Θ the angle between (θ, ϕ) and (θ_s, ϕ_s).

Parameters

- **xo** (*(N, 3) array_like*) – Sequence of secondary source positions.
- **ro** (*float*) – Radius of the spherial secondary source distribution.
- **xs** (*(3,) array_like*) – Virtual source position.
- **fs** (*int*) – Sampling frequency in Hertz.
- **max_order** (*int, optional*) – Ambisonics order.
- **c** (*float, optional*) – Speed of sound in m/s.
- **s2z** (*callable, optional*) – Function transforming s-domain poles and zeros into z-domain, e.g. `matchedz_zpk()`, `scipy.signal.bilinear_zpk()`[28].

Returns

- **delay** (*float*) – Overall delay in seconds.
- **weight** (*float*) – Overall weight.
- **sos** (*list of numpy.ndarray*) – Second-order section filters `scipy.signal.sosfilt()`[29].
- **phaseshift** (*(N,) numpy.ndarray*) – Phase shift in radians.
- **selection** (*(N,) numpy.ndarray*) – Boolean array containing only `True` indicating that all secondary source are "active" for NFC-HOA.

[26] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.bilinear_zpk.html#scipy.signal.bilinear_zpk
[27] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.sosfilt.html#scipy.signal.sosfilt

- **secondary_source_function** (*callable*) – A function that can be used to create the sound field of a single secondary source. See `sfs.td.synthesize()`.

`sfs.td.nfchoa.`**`driving_signals_25d`**(*delay*, *weight*, *sos*, *phaseshift*, *signal*)

Get 2.5-dimensional NFC-HOA driving signals.

Parameters

- **delay** (*float*) – Overall delay in seconds.
- **weight** (*float*) – Overall weight.
- **sos** (*list of array_like*) – Second-order section filters `scipy.signal.sosfilt()`[30].
- **phaseshift** (*(N,) array_like*) – Phase shift in radians.
- **signal** (*(L,) array_like + 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 delayed signals (in a `numpy.ndarray`[31] with shape `(L, N)`), followed by the sampling rate (in Hertz) and a (possibly negative) time offset (in seconds).

`sfs.td.nfchoa.`**`driving_signals_3d`**(*delay*, *weight*, *sos*, *phaseshift*, *signal*)

Get 3-dimensional NFC-HOA driving signals.

Parameters

- **delay** (*float*) – Overall delay in seconds.
- **weight** (*float*) – Overall weight.
- **sos** (*list of array_like*) – Second-order section filters `scipy.signal.sosfilt()`[32].
- **phaseshift** (*(N,) array_like*) – Phase shift in radians.
- **signal** (*(L,) array_like + 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 delayed signals (in a `numpy.ndarray`[33] with shape `(L, N)`), followed by the sampling rate (in Hertz) and a (possibly negative) time offset (in seconds).

Functions

`apply_delays`(signal, delays)	Apply delays for every channel.
`secondary_source_point`(c)	Create a point source for use in `sfs.td.synthesize()`.
`synthesize`(signals, weights, ssd, …)	Compute sound field for an array of secondary sources.

`sfs.td.`**`synthesize`**(*signals*, *weights*, *ssd*, *secondary_source_function*, ***kwargs*)

Compute sound field for an array of secondary sources.

Parameters

28 https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.bilinear_zpk.html#scipy.signal.bilinear_zpk
29 https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.sosfilt.html#scipy.signal.sosfilt
30 https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.sosfilt.html#scipy.signal.sosfilt
31 https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
32 https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.sosfilt.html#scipy.signal.sosfilt
33 https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray

- **signals** (*(N, C) array_like + float*) – Driving signals consisting of audio data (C channels) and a sampling rate (in Hertz). A *DelayedSignal* object can also be used.
- **weights** (*(C,) array_like*) – Additional weights applied during integration, e.g. source selection and tapering.
- **ssd** (*sequence of between 1 and 3 array_like objects*) – Positions (shape `(C, 3)`), normal vectors (shape `(C, 3)`) and weights (shape `(C,)`) of secondary sources. A *SecondarySourceDistribution* can also be used.
- **secondary_source_function** (*callable*) – A function that generates the sound field of a secondary source. This signature is expected:

```
secondary_source_function(
    position, normal_vector, **kwargs) -> numpy.ndarray
```

- ****kwargs** – All keyword arguments are forwarded to *secondary_source_function*. This is typically used to pass the *observation_time* and *grid* arguments.

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

`sfs.td.`**`apply_delays`**`(`*signal, delays*`)`
Apply delays for every channel.

Parameters

- **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.
- **delays** (*(C,) array_like*) – Delay in seconds for each channel (C), negative values allowed.

Returns *DelayedSignal* – A tuple containing the delayed signals (in a `numpy.ndarray`[34] with shape `(N, C)`), followed by the sampling rate (in Hertz) and a (possibly negative) time offset (in seconds).

`sfs.td.`**`secondary_source_point`**`(`*c*`)`
Create a point source for use in *sfs.td.synthesize()*.

3.3 sfs.array

Compute positions of various secondary source distributions.

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

[34] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray

Functions

as_secondary_source_distribution(arg, **kwargs)	Create a SecondarySourceDistribution.
circular(N, R, *[, center])	Return circular secondary source distribution parallel to the xy-plane.
concatenate(*arrays)	Concatenate SecondarySourceDistribution objects.
cube(N, spacing, *[, center, orientation])	Return cube-shaped secondary source distribtion.
edge(Nxy, spacing, *[, center, orientation])	Return SSD along the xy-axis with sharp edge at the origin.
linear(N, spacing, *[, center, orientation])	Return linear, equidistantly sampled secondary source distribution.
linear_diff(distances, *[, center, orientation])	Return linear secondary source distribution from a list of distances.
linear_random(N, min_spacing, max_spacing, *)	Return randomly sampled linear array.
load(file, *[, center, orientation])	Load secondary source distribution from file.
planar(N, spacing, *[, center, orientation])	Return planar secondary source distribtion.
rectangular(N, spacing, *[, center, orientation])	Return rectangular secondary source distribution.
rounded_edge(Nxy, Nr, spacing, *[, center, . . .])	Return SSD along the xy-axis with rounded edge at the origin.
sphere_load(file, radius, *[, center])	Load spherical secondary source distribution from file.
weights_midpoint(positions, *, closed)	Calculate loudspeaker weights for a simply connected array.

Classes

SecondarySourceDistribution(x, n, a)	Create new instance of SecondarySourceDistribution(x, n, a)

class sfs.array.**SecondarySourceDistribution**(*x*, *n*, *a*)

Create new instance of SecondarySourceDistribution(x, n, a)

take(*indices*)

Return a sub-array given by *indices*.

sfs.array.**as_secondary_source_distribution**(*arg*, ***kwargs*)

Create a SecondarySourceDistribution.

Parameters

- **arg** (*sequence of between 1 and 3 array_like objects*) – All elements are converted to NumPy arrays. If only 1 element is given, all normal vectors are set to *NaN*. If only 1 or 2 elements are given, all weights are set to 1.0.
- ****kwargs** – All keyword arguments are forwarded to numpy.asarray()[35].

Returns SecondarySourceDistribution – A named tuple consisting of three numpy.ndarray[36]s containing positions, normal vectors and weights.

[35] https://numpy.org/doc/stable/reference/generated/numpy.asarray.html#numpy.asarray
[36] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray

`sfs.array.`**`linear`**(*N*, *spacing*, ***, *center=[0, 0, 0]*, *orientation=[1, 0, 0]*)
Return linear, equidistantly sampled secondary source distribution.

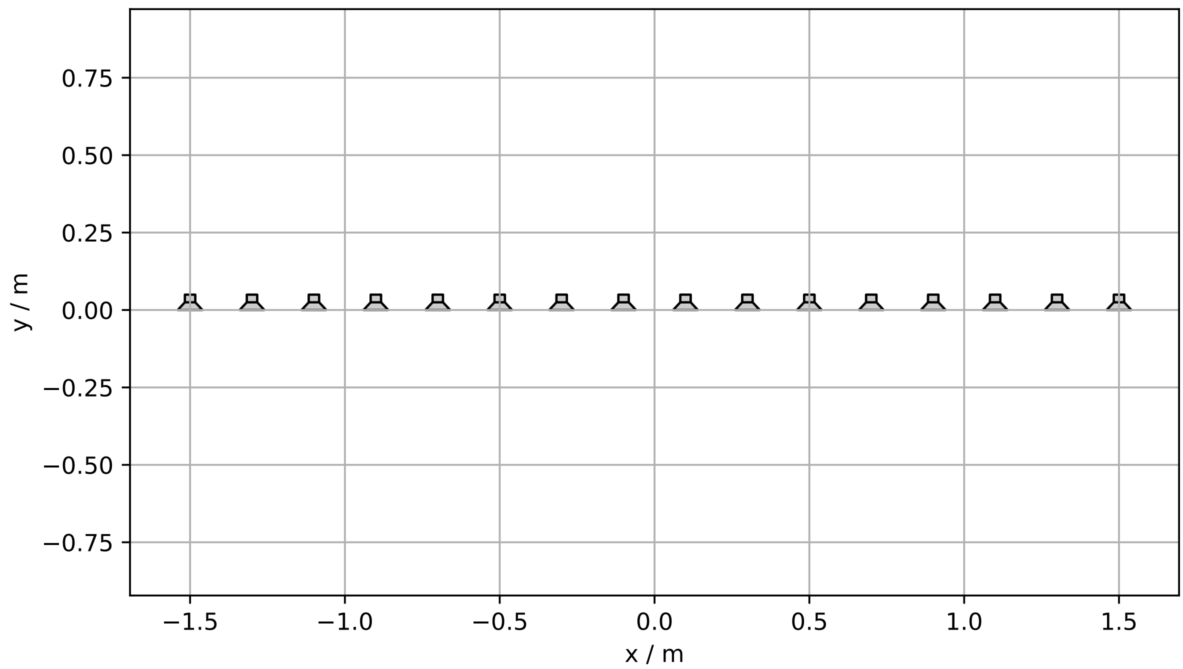
Parameters

- **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 `SecondarySourceDistribution` – Positions, orientations and weights of secondary sources.

Examples

```
x0, n0, a0 = sfs.array.linear(16, 0.2, orientation=[0, -1, 0])
sfs.plot2d.loudspeakers(x0, n0, a0)
plt.axis('equal')
plt.xlabel('x / m')
plt.ylabel('y / m')
```



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

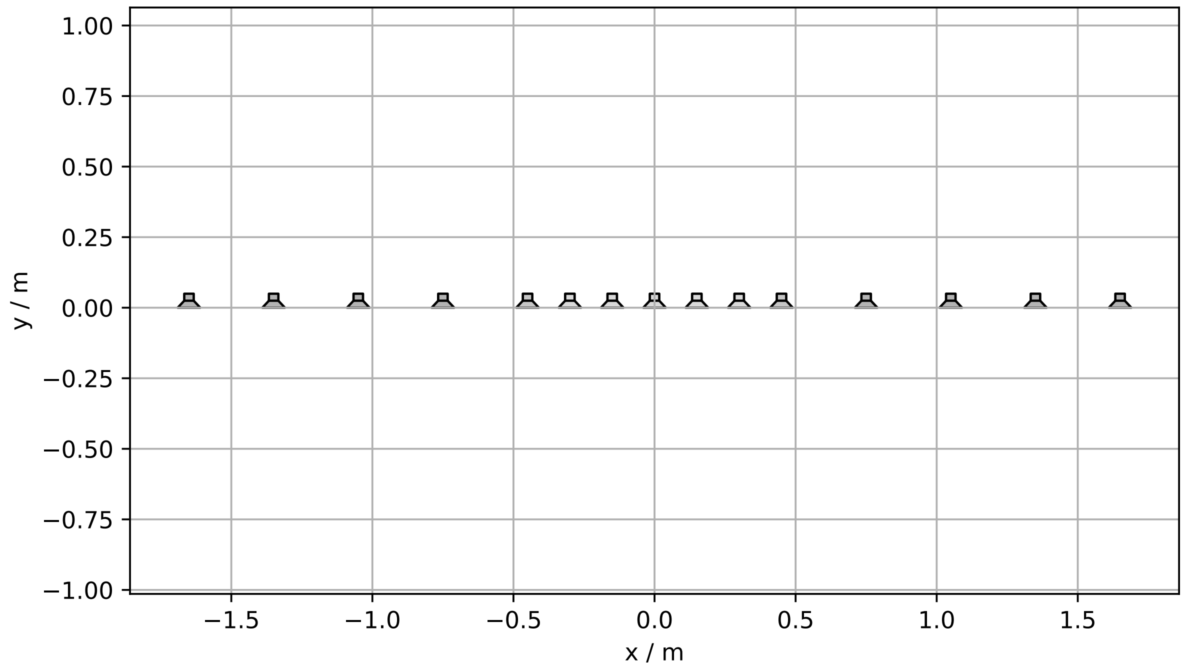
Parameters

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

Returns `SecondarySourceDistribution` – Positions, orientations and weights of secondary sources.

Examples

```
x0, n0, a0 = sfs.array.linear_diff(4 * [0.3] + 6 * [0.15] + 4 * [0.3],
                                   orientation=[0, -1, 0])
sfs.plot2d.loudspeakers(x0, n0, a0)
plt.axis('equal')
plt.xlabel('x / m')
plt.ylabel('y / m')
```



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

Return randomly sampled linear array.

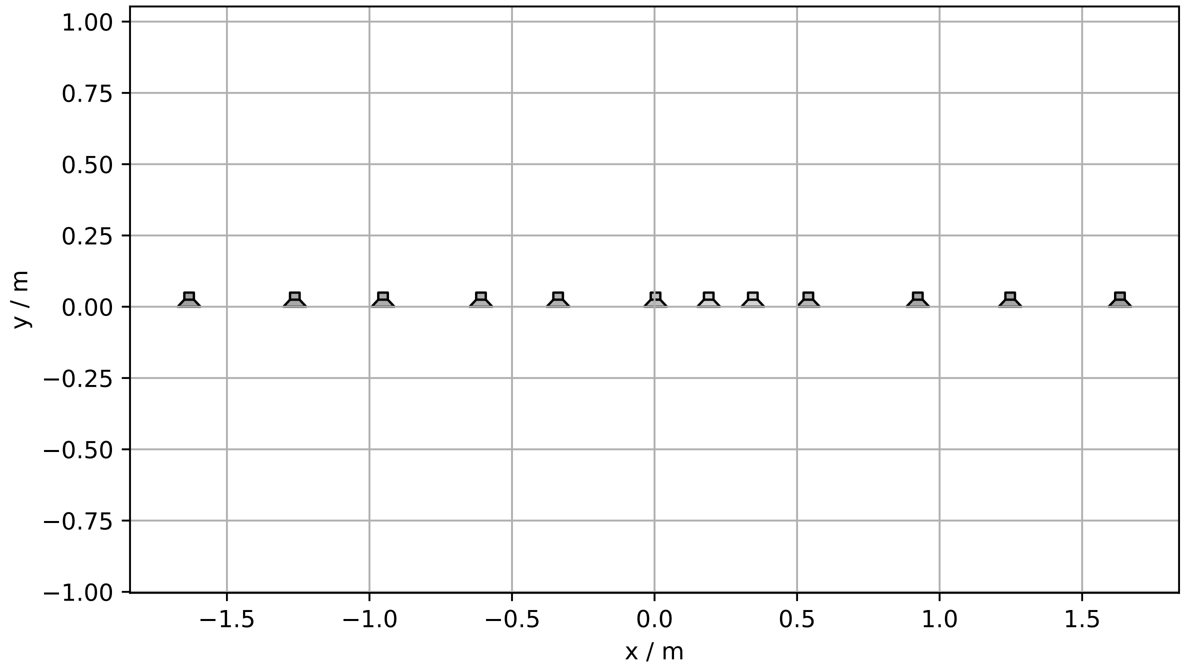
Parameters

- **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`[37].

Returns `SecondarySourceDistribution` – Positions, orientations and weights of secondary sources.

Examples

```
x0, n0, a0 = sfs.array.linear_random(
    N=12,
    min_spacing=0.15, max_spacing=0.4,
    orientation=[0, -1, 0])
sfs.plot2d.loudspeakers(x0, n0, a0)
plt.axis('equal')
plt.xlabel('x / m')
plt.ylabel('y / m')
```



`sfs.array.`**`circular`**(*N*, *R*, ***, *center=[0, 0, 0]*)

Return circular secondary source distribution parallel to the xy-plane.

Parameters

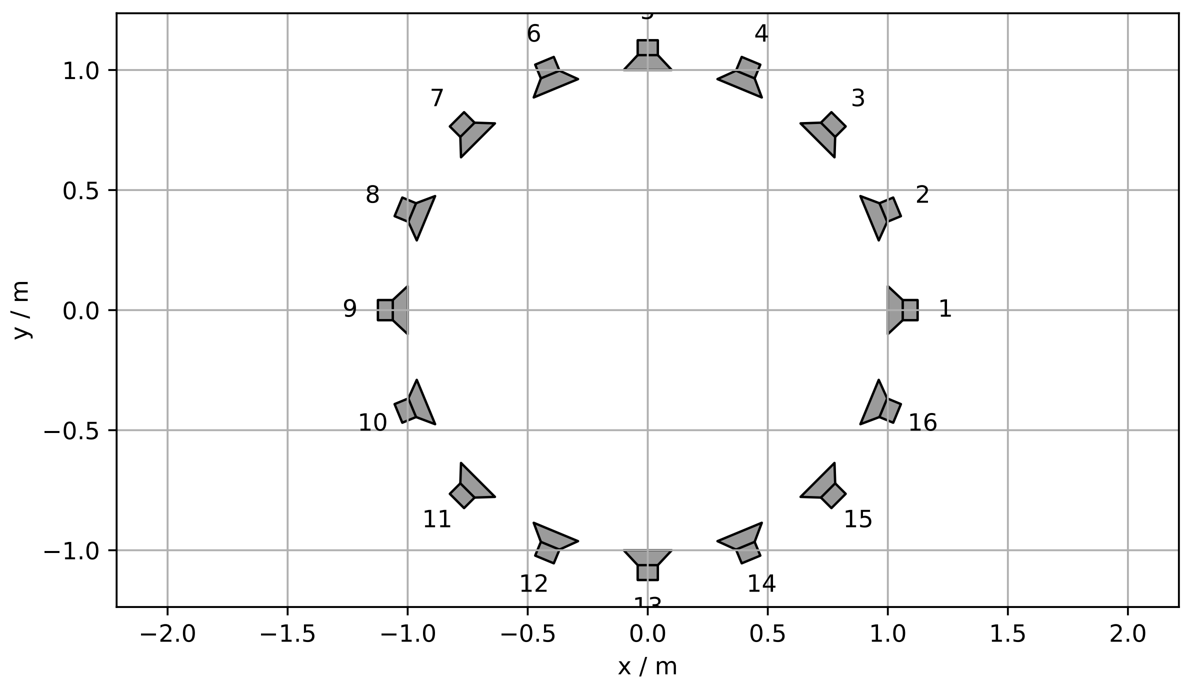
- **N** (*int*) – Number of secondary sources.
- **R** (*float*) – Radius in metres.
- **center** – See *`linear()`*.

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

[37] https://numpy.org/doc/stable/reference/random/legacy.html#numpy.random.RandomState

Examples

```
x0, n0, a0 = sfs.array.circular(16, 1)
sfs.plot2d.loudspeakers(x0, n0, a0, size=0.2, show_numbers=True)
plt.axis('equal')
plt.xlabel('x / m')
plt.ylabel('y / m')
```



sfs.array.**rectangular**(*N*, *spacing*, ***, *center=[0, 0, 0]*, *orientation=[1, 0, 0]*)
Return rectangular secondary source distribution.

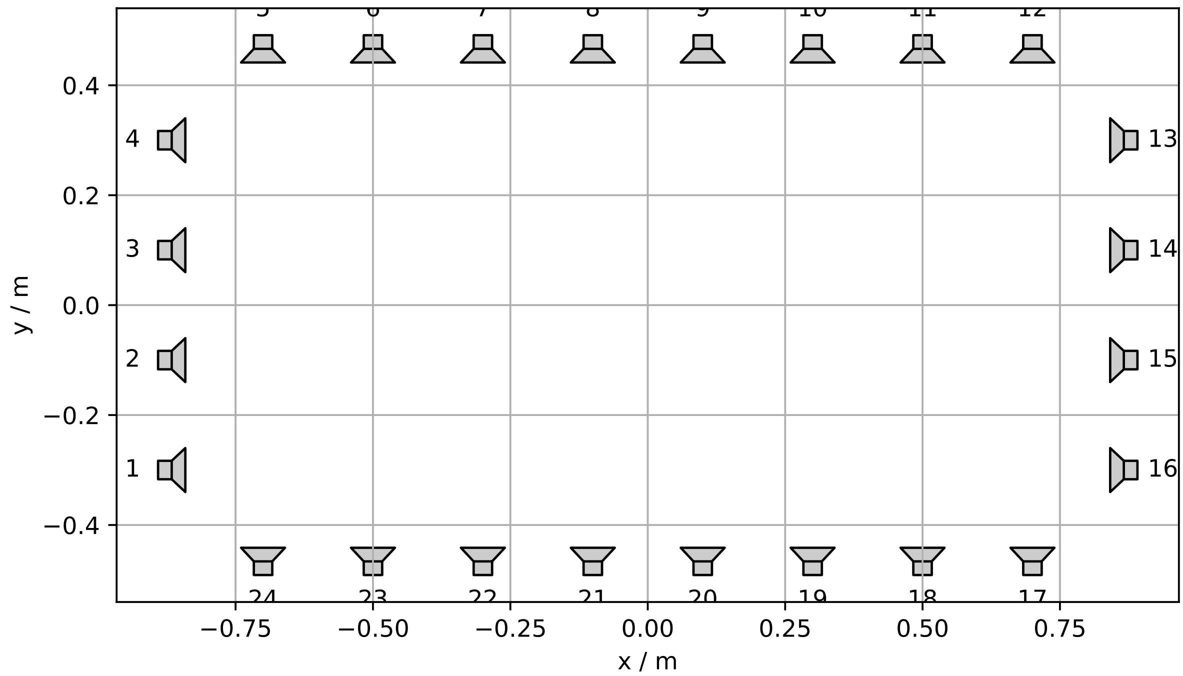
Parameters

- **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.
- **spacing** (*float*) – Distance (in metres) between secondary sources.
- **center, orientation** – See `linear()`. The *orientation* corresponds to the first linear segment.

Returns `SecondarySourceDistribution` – Positions, orientations and weights of secondary sources.

Examples

```
x0, n0, a0 = sfs.array.rectangular((4, 8), 0.2)
sfs.plot2d.loudspeakers(x0, n0, a0, show_numbers=True)
plt.axis('equal')
plt.xlabel('x / m')
plt.ylabel('y / m')
```



sfs.array.**rounded_edge**(*Nxy, Nr, spacing, *, center=[0, 0, 0], orientation=[1, 0, 0]*)

Return SSD along the xy-axis with rounded edge at the origin.

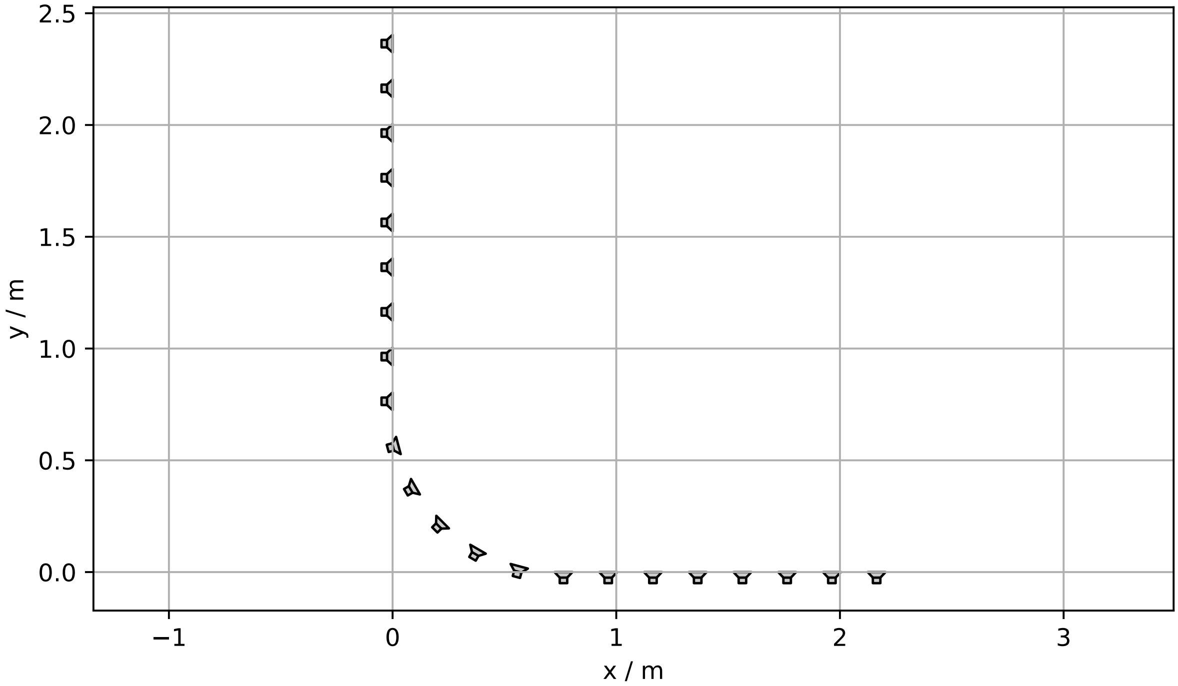
Parameters

- **Nxy** (*int*) – Number of secondary sources along x- and y-axis.
- **Nr** (*int*) – Number of secondary sources in rounded edge. Radius of edge is adjusted to equdistant sampling along entire array.
- **spacing** (*float*) – Distance (in metres) between secondary sources.
- **center** (*(3,) array_like, optional*) – Position of edge.
- **orientation** (*(3,) array_like, optional*) – Normal vector of array. Default orientation is along xy-axis.

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

Examples

```
x0, n0, a0 = sfs.array.rounded_edge(8, 5, 0.2)
sfs.plot2d.loudspeakers(x0, n0, a0)
plt.axis('equal')
plt.xlabel('x / m')
plt.ylabel('y / m')
```



sfs.array.**edge**(*Nxy*, *spacing*, ***, *center=[0, 0, 0]*, *orientation=[1, 0, 0]*)

Return SSD along the xy-axis with sharp edge at the origin.

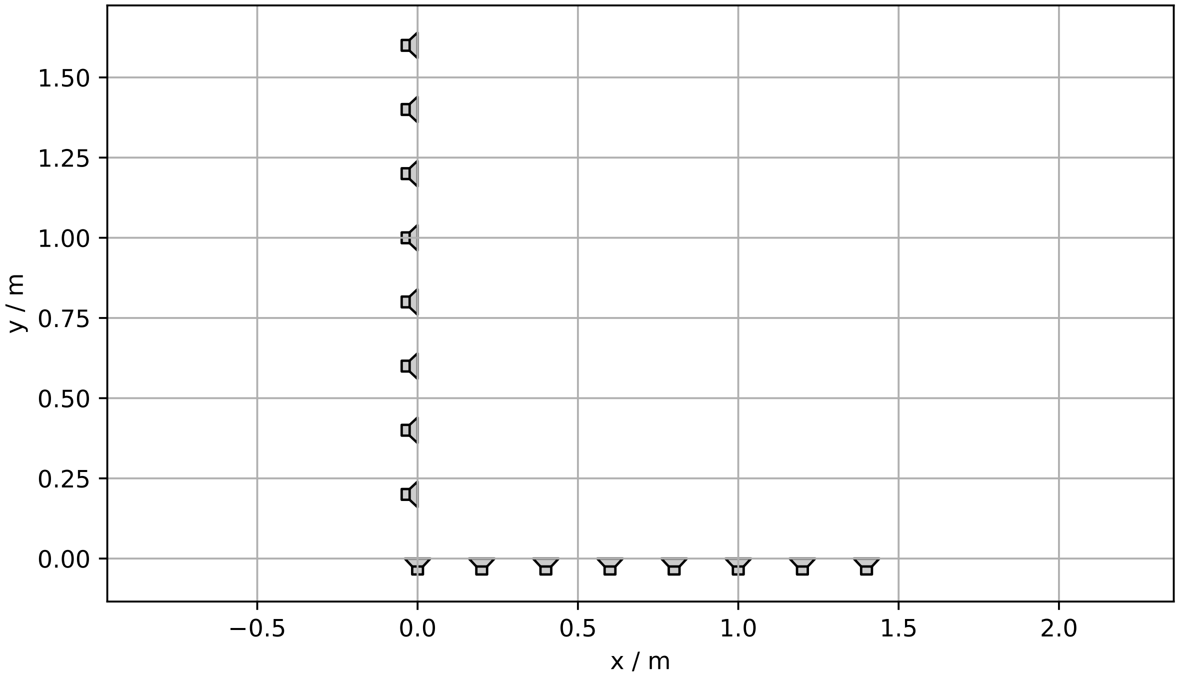
Parameters

- **Nxy** (*int*) – Number of secondary sources along x- and y-axis.
- **spacing** (*float*) – Distance (in metres) between secondary sources.
- **center** (*(3,) array_like, optional*) – Position of edge.
- **orientation** (*(3,) array_like, optional*) – Normal vector of array. Default orientation is along xy-axis.

Returns `SecondarySourceDistribution` – Positions, orientations and weights of secondary sources.

Examples

```
x0, n0, a0 = sfs.array.edge(8, 0.2)
sfs.plot2d.loudspeakers(x0, n0, a0)
plt.axis('equal')
plt.xlabel('x / m')
plt.ylabel('y / m')
```



`sfs.array.planar`(*N*, *spacing*, ***, *center=[0, 0, 0]*, *orientation=[1, 0, 0]*)
Return planar secondary source distribtion.

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 `SecondarySourceDistribution` – Positions, orientations and weights of secondary sources.

Examples

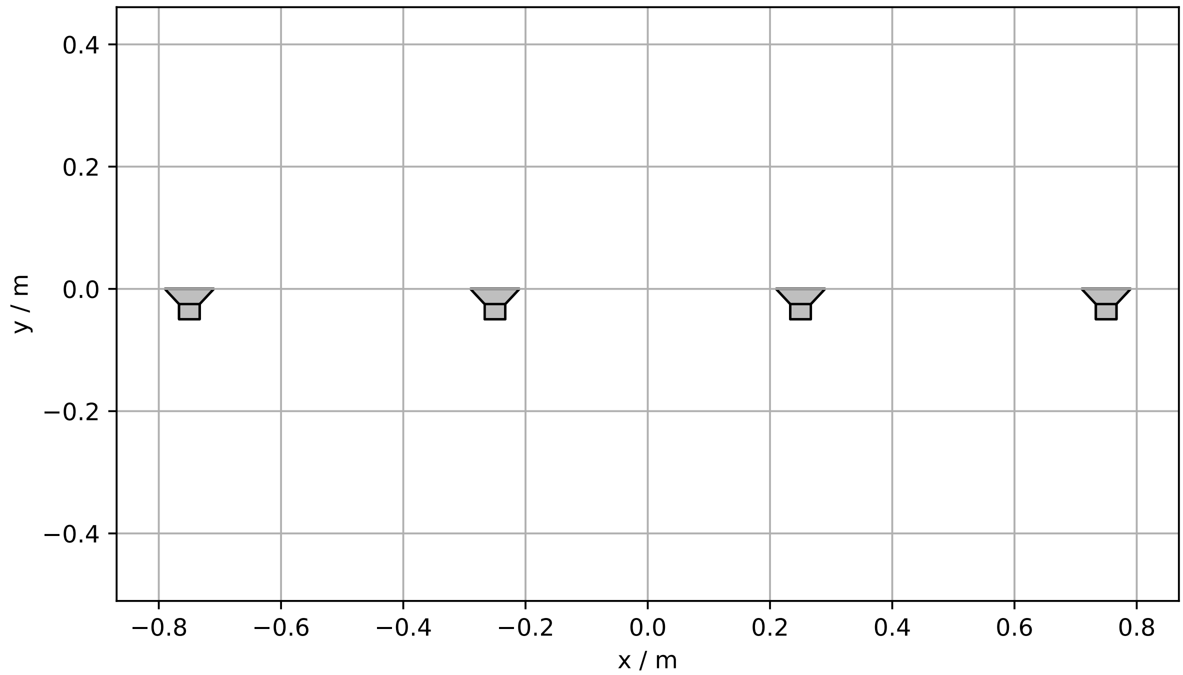
```
x0, n0, a0 = sfs.array.planar(
    (4,3), 0.5, orientation=[0, 0, 1])  # 4 sources along y, 3 sources along
↪x
x0, n0, a0 = sfs.array.planar(
    (4,3), 0.5, orientation=[1, 0, 0])  # 4 sources along y, 3 sources along
↪z

x0, n0, a0 = sfs.array.planar(
```

(continues on next page)

(continued from previous page)

```
    (4,3), 0.5, orientation=[0, 1, 0])  # 4 sources along x, 3 sources along␣
↪z
sfs.plot2d.loudspeakers(x0, n0, a0)  # plot the last ssd in 2D
plt.axis('equal')
plt.xlabel('x / m')
plt.ylabel('y / m')
```



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

Return cube-shaped secondary source distribtion.

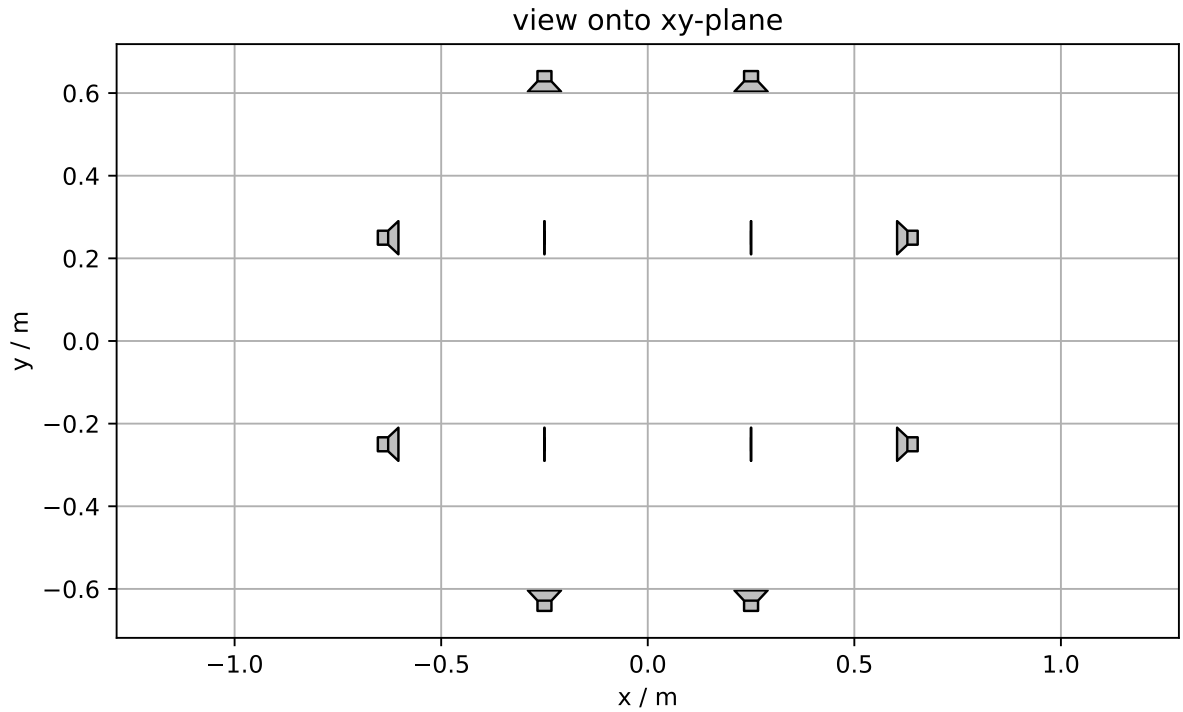
Parameters

- **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 `SecondarySourceDistribution` – Positions, orientations and weights of secondary sources.

Examples

```
x0, n0, a0 = sfs.array.cube(
    N=2, spacing=0.5,
    center=[0, 0, 0], orientation=[1, 0, 0])
sfs.plot2d.loudspeakers(x0, n0, a0)
plt.axis('equal')
plt.xlabel('x / m')
plt.ylabel('y / m')
plt.title('view onto xy-plane')
```



`sfs.array.`**`sphere_load`**(*file, radius, *, center=[0, 0, 0]*)

Load spherical secondary source distribution from file.

ASCII Format (see MATLAB SFS Toolbox) with 4 numbers (3 for the cartesian position vector, 1 for the integration weight) per secondary source located on the unit circle which is resized by the given radius and shifted to the given center.

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

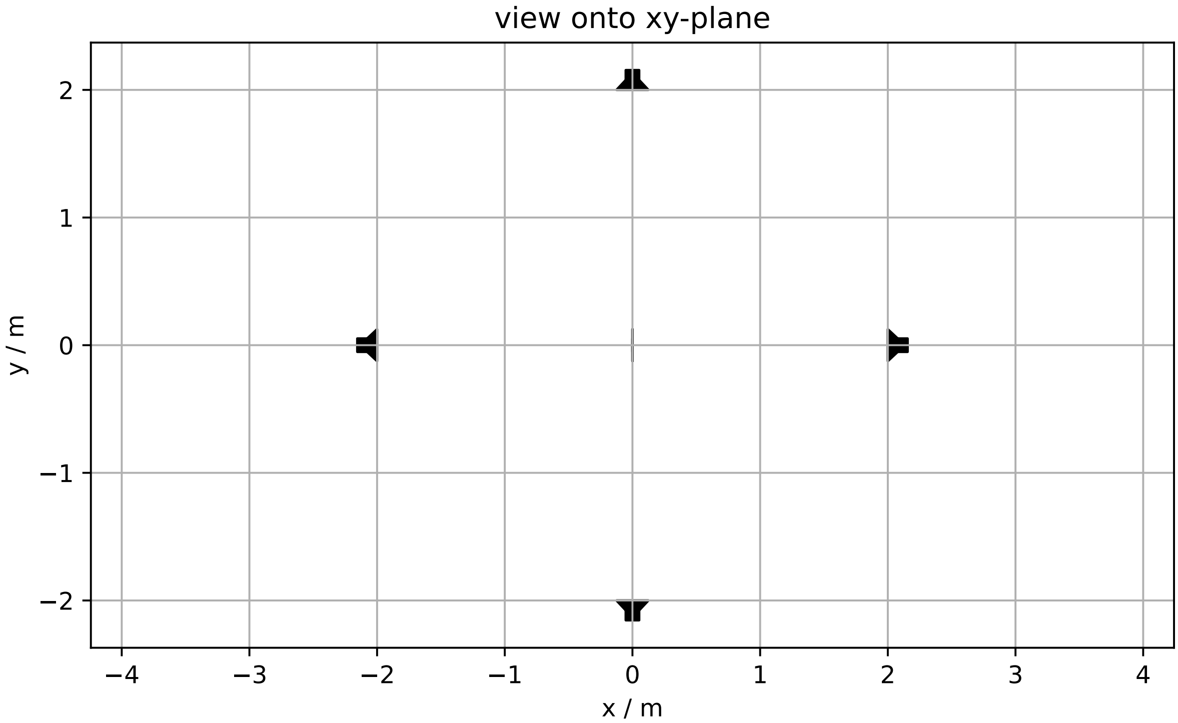
Examples

content of `example_array_6LS_3D.txt`:

```
1 0 0 1
-1 0 0 1
0 1 0 1
0 -1 0 1
0 0 1 1
0 0 -1 1
```

corresponds to the 3-dimensional 6-point spherical 3-design[38].

```
x0, n0, a0 = sfs.array.sphere_load(
    '../data/arrays/example_array_6LS_3D.txt',
    radius=2,
    center=[0, 0, 0])
sfs.plot2d.loudspeakers(x0, n0, a0, size=0.25)
plt.axis('equal')
plt.xlabel('x / m')
plt.ylabel('y / m')
plt.title('view onto xy-plane')
```



`sfs.array.load`(*file*, ***, *center=[0, 0, 0]*, *orientation=[1, 0, 0]*)

Load secondary source distribution from file.

Comma Separated Values (CSV) format with 7 values (3 for the cartesian position vector, 3 for the cartesian inward normal vector, 1 for the integration weight) per secondary source.

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

Examples

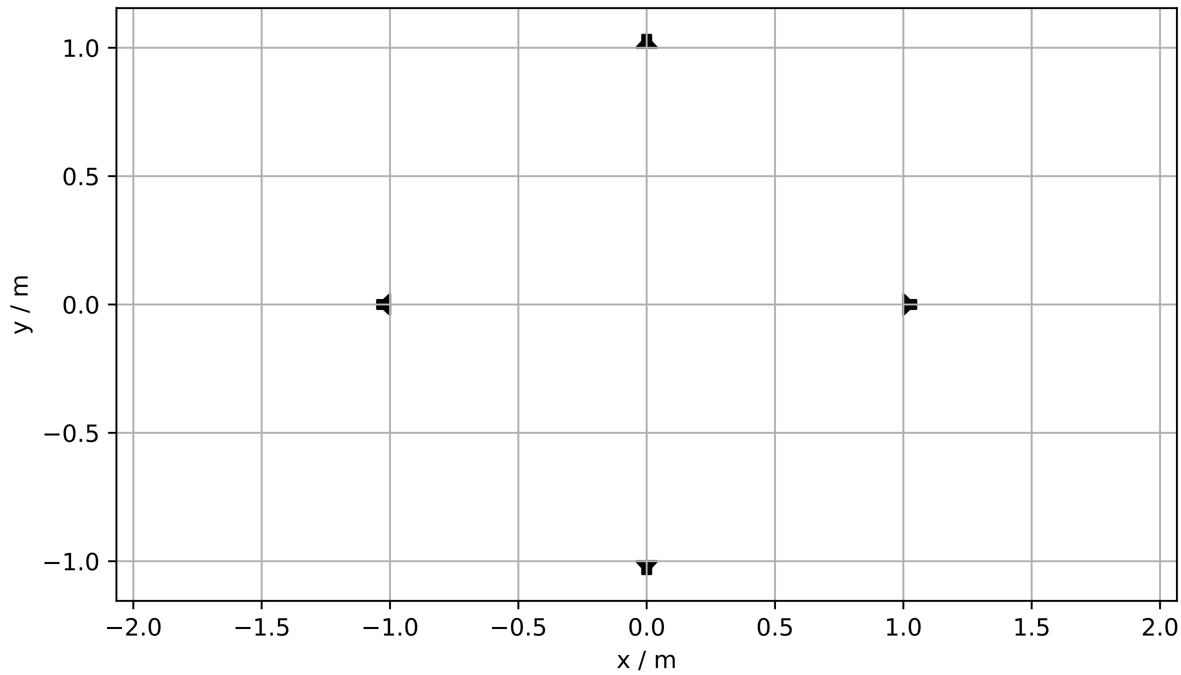
content of `example_array_4LS_2D.csv`:

```
1,0,0,-1,0,0,1
0,1,0,0,-1,0,1
-1,0,0,1,0,0,1
0,-1,0,0,1,0,1
```

corresponds to 4 sources at 1, j, -1, -j in the complex plane. This setup is typically used for Quadraphonic audio reproduction.

[38] http://neilsloane.com/sphdesigns/dim3/des.3.6.3.txt

```
x0, n0, a0 = sfs.array.load('../data/arrays/example_array_4LS_2D.csv')
sfs.plot2d.loudspeakers(x0, n0, a0)
plt.axis('equal')
plt.xlabel('x / m')
plt.ylabel('y / m')
```




```
x0, n0, a0 = sfs.array.load(
    '../data/arrays/wfs_university_rostock_2018.csv')
sfs.plot2d.loudspeakers(x0, n0, a0)
plt.axis('equal')
plt.xlabel('x / m')
plt.ylabel('y / m')
plt.title('top view of 64 channel WFS system at university of Rostock')
```

`sfs.array.`**`weights_midpoint`**(*positions*, *, *closed*)

Calculate loudspeaker weights for a simply connected array.

The weights are calculated according to the midpoint rule.

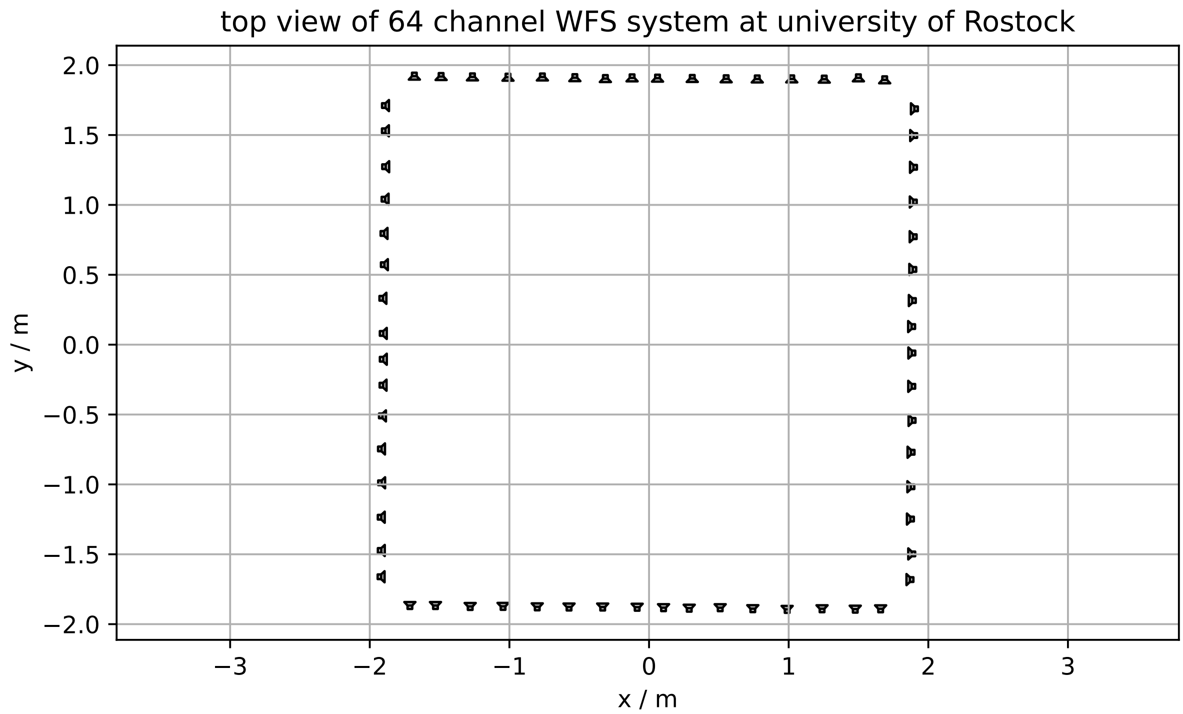
Parameters

- **positions** (*(N, 3) array_like*) – Sequence of secondary source positions.

  Note: The loudspeaker positions have to be ordered along the contour.

- **closed** (*bool*) – `True` if the loudspeaker contour is closed.

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



Examples

```
>>> import sfs
>>> x0, n0, a0 = sfs.array.circular(2**5, 1)
>>> a = sfs.array.weights_midpoint(x0, closed=True)
>>> max(abs(a0-a))
0.0003152601902411123
```

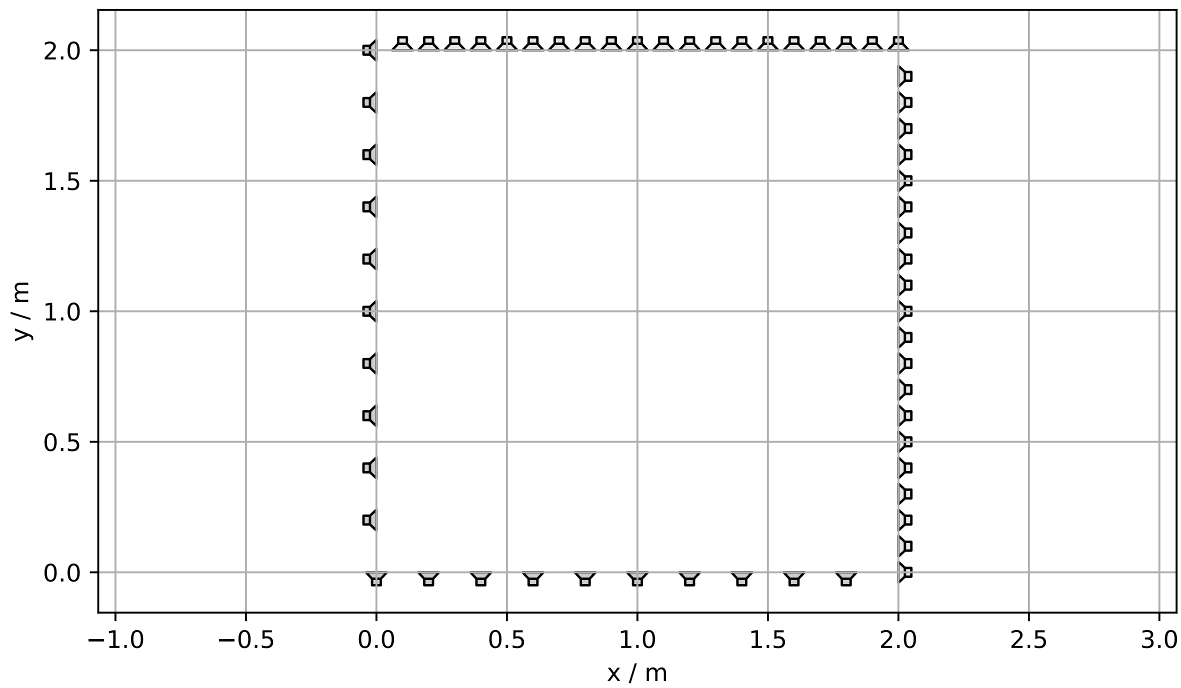
`sfs.array.concatenate(*arrays)`

Concatenate *SecondarySourceDistribution* objects.

Returns *SecondarySourceDistribution* – Positions, orientations and weights of the concatenated secondary sources.

Examples

```
ssd1 = sfs.array.edge(10, 0.2)
ssd2 = sfs.array.edge(20, 0.1, center=[2, 2, 0], orientation=[-1, 0, 0])
x0, n0, a0 = sfs.array.concatenate(ssd1, ssd2)
sfs.plot2d.loudspeakers(x0, n0, a0)
plt.axis('equal')
plt.xlabel('x / m')
plt.ylabel('y / m')
```



3.4 sfs.tapering

Weights (tapering) for the driving function.

```
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
```

Functions

kaiser(active, *, beta)	Kaiser tapering window.
none(active)	No tapering window.
tukey(active, *, alpha)	Tukey tapering window.

`sfs.tapering.none`(*active*)

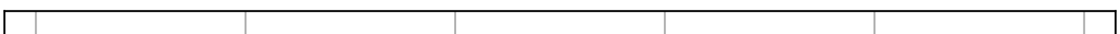
No tapering window.

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

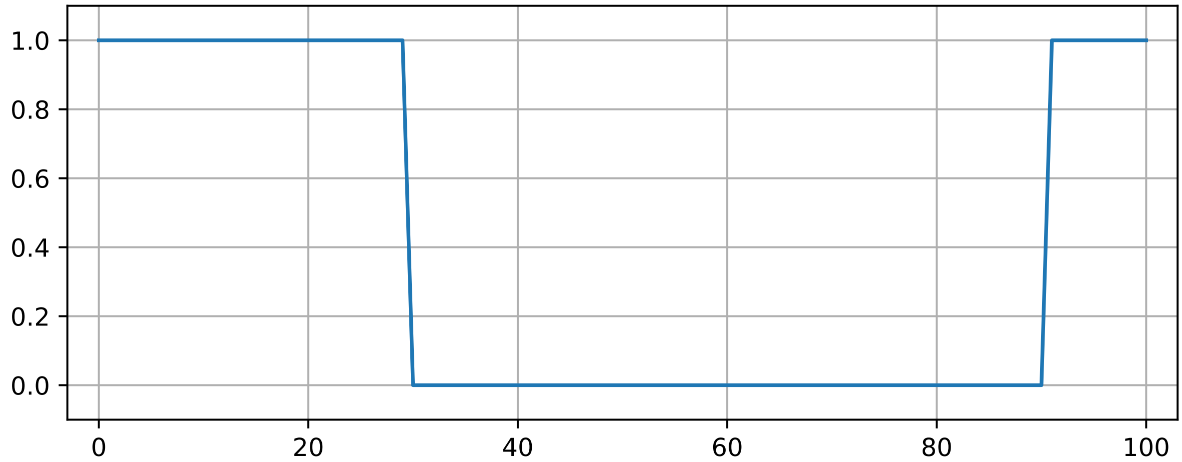
Returns *type(active)* – The input, unchanged.

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`[39] – Tapering weights.

Examples

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

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

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

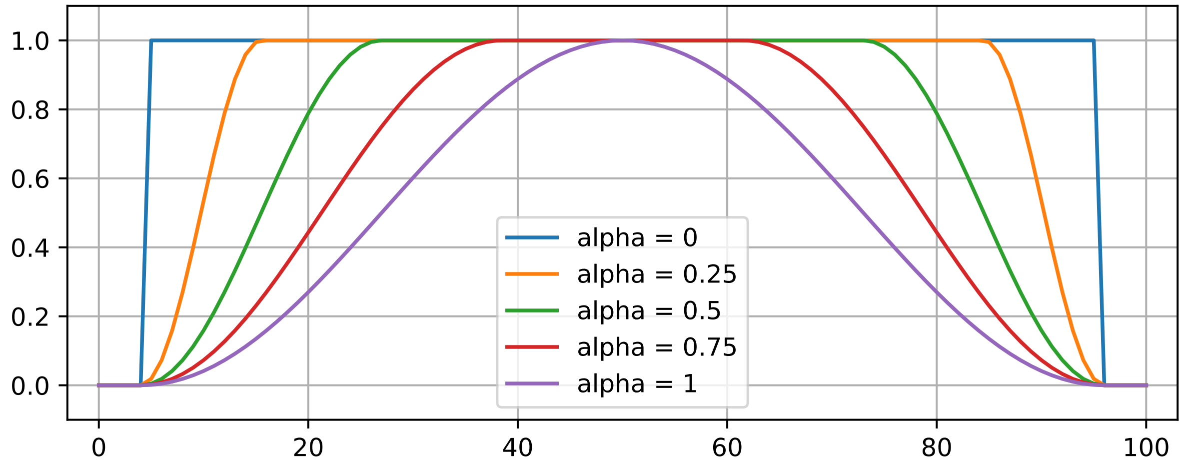
Kaiser tapering window.

This uses `numpy.kaiser()`[40].

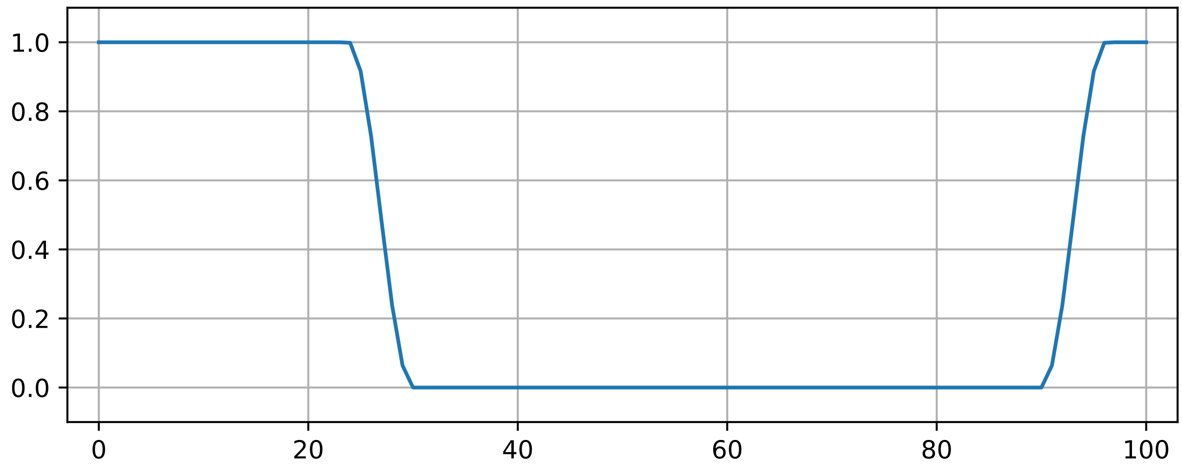
Parameters

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

[39] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray



1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
alpha = 0
alpha = 0.25
alpha = 0.5
alpha = 0.75
alpha = 1



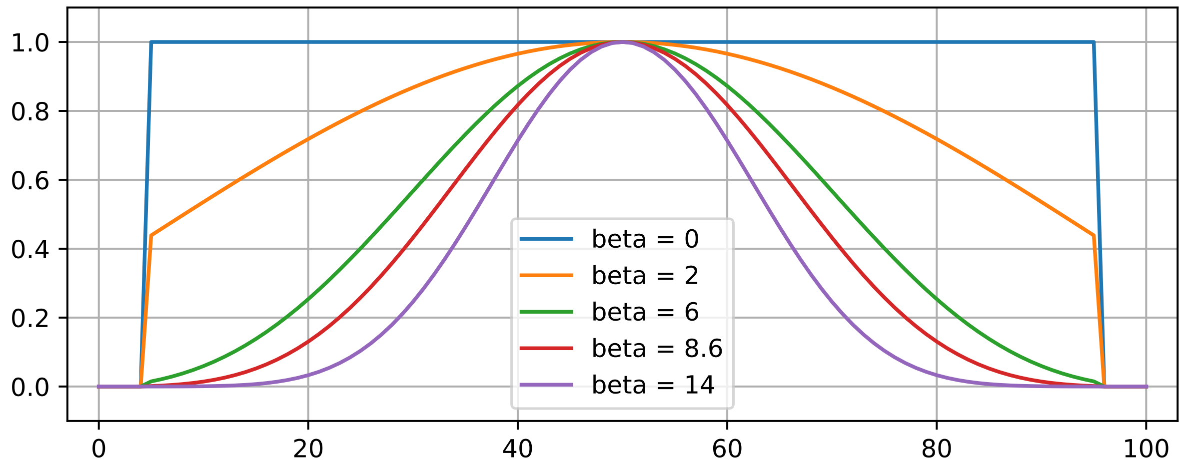
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100

- **alpha** (*float*) – Shape parameter of the Kaiser window, see `numpy.kaiser()`[41].

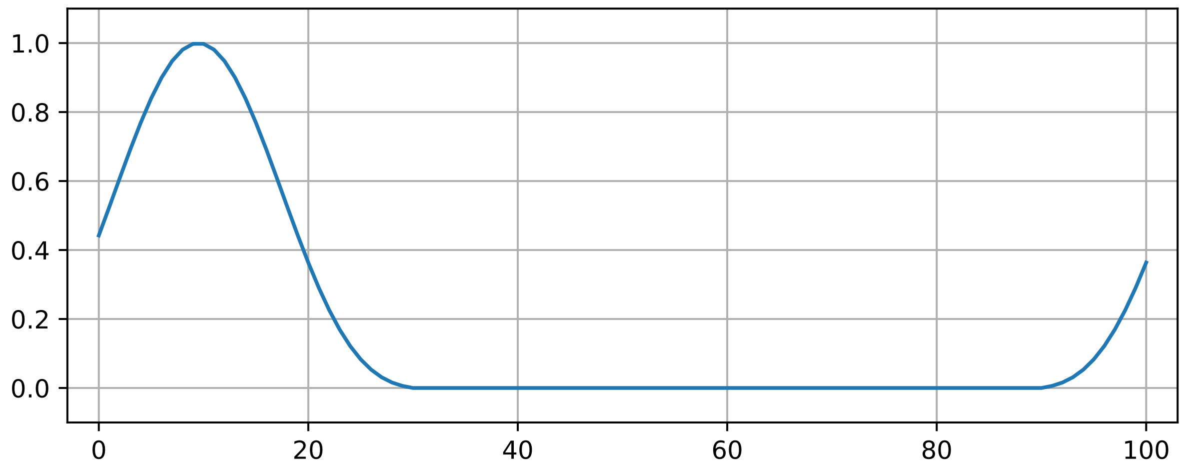
Returns (len(active),) `numpy.ndarray`[42] – Tapering weights.

Examples

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




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



[40] https://numpy.org/doc/stable/reference/generated/numpy.kaiser.html#numpy.kaiser
[41] https://numpy.org/doc/stable/reference/generated/numpy.kaiser.html#numpy.kaiser
[42] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray

3.5 sfs.plot2d

2D plots of sound fields etc.

Functions

`add_colorbar`(im, *[, aspect, pad])	Add a vertical color bar to a plot.
`amplitude`(p, grid, *[, xnorm, cmap, vmin, ...])	Two-dimensional plot of sound field (real part).
`level`(p, grid, *[, xnorm, power, cmap, ...])	Two-dimensional plot of level (dB) of sound field.
`loudspeakers`(x0, n0[, a0, size, ...])	Draw loudspeaker symbols at given locations and angles.
`particles`(x, *[, trim, ax, xlabel, ylabel, ...])	Plot particle positions as scatter plot.
`reference`(xref, *[, size, ax])	Draw reference/normalization point.
`secondary_sources`(x0, n0, *[, size, grid])	Simple visualization of secondary source locations.
`vectors`(v, grid, *[, cmap, headlength, ...])	Plot a vector field in the xy plane.
`virtualsource`(xs[, ns, type, ax])	Draw position/orientation of virtual source.

`sfs.plot2d.`**`virtualsource`**(*xs*, *ns=None*, *type='point'*, *, *ax=None*)

Draw position/orientation of virtual source.

`sfs.plot2d.`**`reference`**(*xref*, *, *size=0.1*, *ax=None*)

Draw reference/normalization point.

`sfs.plot2d.`**`secondary_sources`**(*x0*, *n0*, *, *size=0.05*, *grid=None*)

Simple visualization of secondary source locations.

Parameters

- **x0** (*(N, 3) array_like*) – Loudspeaker positions.
- **n0** (*(N, 3) or (3,) array_like*) – Normal vector(s) of loudspeakers.
- **size** (*float, optional*) – Size of loudspeakers in metres.
- **grid** (*triple of array_like, optional*) – If specified, only loudspeakers within the *grid* are shown.

`sfs.plot2d.`**`loudspeakers`**(*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`[43] object or – if not specified – into the current axes.

[43] https://matplotlib.org/api/axes_api.html#matplotlib.axes.Axes

`sfs.plot2d.amplitude(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 (real part).

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()`[44] was used with a scalar for *z*, *y* or *x*, respectively (and of course with the default `indexing='xy'`).

  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.
- **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()`.

Returns *AxesImage* – See `matplotlib.pyplot.imshow()`[45].

Other Parameters

- **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()`[46] and `matplotlib.pyplot.ylabel()`[47].
- **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()`[48]).
- **cmap, vmin, vmax, **kwargs** – All further parameters are forwarded to `matplotlib.pyplot.imshow()`[49].

See also:

`sfs.plot2d.level`

`sfs.plot2d.level(p, grid, *, xnorm=None, power=False, cmap=None, vmax=3, vmin=-50, **kwargs)`

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

---

[44] https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html#numpy.meshgrid
[45] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.imshow.html#matplotlib.pyplot.imshow
[46] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.xlabel.html#matplotlib.pyplot.xlabel
[47] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.ylabel.html#matplotlib.pyplot.ylabel
[48] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.gca.html#matplotlib.pyplot.gca
[49] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.imshow.html#matplotlib.pyplot.imshow

Takes the same parameters as `sfs.plot2d.amplitude()`.

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

`sfs.plot2d.`**`particles`**`(`*x, *, trim=None, ax=None, xlabel='x (m)', ylabel='y (m)', edgecolors=None, marker='.', s=15, **kwargs*`)`

Plot particle positions as scatter plot.

Parameters **x** (*triple or pair of array_like*) – x, y and optionally z components of particle positions. The z components are ignored. If the values are complex, the imaginary parts are ignored.

Returns *Scatter* – See `matplotlib.pyplot.scatter()`[50].

Other Parameters

- **trim** (*array of float, optional*) – xmin, xmax, ymin, ymax limits for which the particles are plotted.
- **ax** (*Axes, optional*) – If given, the plot is created on *ax* instead of the current axis (see `matplotlib.pyplot.gca()`[51]).
- **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()`[52] and `matplotlib.pyplot.ylabel()`[53].
- **edgecolors, markr, s, **kwargs** – All further parameters are forwarded to `matplotlib.pyplot.scatter()`[54].

`sfs.plot2d.`**`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()`[55].

Other Parameters

- **ax** (*Axes, optional*) – If given, the plot is created on *ax* instead of the current axis (see `matplotlib.pyplot.gca()`[56]).
- **clim** (*pair of float, optional*) – Limits for the scaling of arrow colors. See `matplotlib.pyplot.quiver()`[57].
- **cmap, headlength, headaxislength, **kwargs** – All further parameters are forwarded to `matplotlib.pyplot.quiver()`[58].

`sfs.plot2d.`**`add_colorbar`**`(`*im, *, aspect=20, pad=0.5, **kwargs*`)`

Add a vertical color bar to a plot.

[50] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html#matplotlib.pyplot.scatter
[51] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.gca.html#matplotlib.pyplot.gca
[52] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.xlabel.html#matplotlib.pyplot.xlabel
[53] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.ylabel.html#matplotlib.pyplot.ylabel
[54] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html#matplotlib.pyplot.scatter
[55] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.quiver.html#matplotlib.pyplot.quiver
[56] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.gca.html#matplotlib.pyplot.gca
[57] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.quiver.html#matplotlib.pyplot.quiver
[58] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.quiver.html#matplotlib.pyplot.quiver

Parameters

- **im** (*ScalarMappable*) – The output of `sfs.plot2d.amplitude()`, `sfs.plot2d.level()` or any other `matplotlib.cm.ScalarMappable`[59].
- **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()`[60] has a slightly different meaning ("fraction of original axes")!

- ****kwargs** – All further arguments are forwarded to `matplotlib.figure.Figure.colorbar()`[61].

See also:

`matplotlib.pyplot.colorbar`[62]

3.6 sfs.plot3d

3D plots of sound fields etc.

Functions

`secondary_sources`(x0, n0[, a0, w, h])	Plot positions and normals of a 3D secondary source distribution.

`sfs.plot3d.`**`secondary_sources`**(*x0*, *n0*, *a0=None*, *, *w=0.08*, *h=0.08*)

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

3.7 sfs.util

Various utility functions.

Module Attributes

`DelayedSignal`(data, samplerate, time)	A tuple of audio data, sampling rate and start time.

[59] https://matplotlib.org/api/cm_api.html#matplotlib.cm.ScalarMappable
[60] https://matplotlib.org/api/figure_api.html#matplotlib.figure.Figure.colorbar
[61] https://matplotlib.org/api/figure_api.html#matplotlib.figure.Figure.colorbar
[62] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.colorbar.html#matplotlib.pyplot.colorbar

Functions

as_delayed_signal(arg, **kwargs)	Make sure that the given argument can be used as a signal.
as_xyz_components(components, **kwargs)	Convert *components* to *XyzComponents* of numpy.ndarray[63]s.
asarray_1d(a, **kwargs)	Squeeze the input and check if the result is one-dimensional.
asarray_of_rows(a, **kwargs)	Convert to 2D array, turn column vector into row vector.
broadcast_zip(*args)	Broadcast arguments to the same shape and then use zip()[64].
cart2sph(x, y, z)	Cartesian to spherical coordinate transform.
db(x, *[, power])	Convert *x* to decibel.
direction_vector(alpha[, beta])	Compute normal vector from azimuth, colatitude.
image_sources_for_box(x, L, N, *[, prune])	Image source method for a cuboid room.
max_order_circular_harmonics(N)	Maximum order of 2D/2.5D HOA.
max_order_spherical_harmonics(N)	Maximum order of 3D HOA.
normalize(p, grid, xnorm)	Normalize sound field wrt position *xnorm*.
normalize_vector(x)	Normalize a 1D vector.
probe(p, grid, x)	Determine the value at position *x* in the sound field *p*.
rotation_matrix(n1, n2)	Compute rotation matrix for rotation from *n1* to *n2*.
source_selection_all(N)	Select all secondary sources.
source_selection_focused(ns, x0, xs)	Secondary source selection for a focused source.
source_selection_line(n0, x0, xs)	Secondary source selection for a line source.
source_selection_plane(n0, n)	Secondary source selection for a plane wave.
source_selection_point(n0, x0, xs)	Secondary source selection for a point source.
sph2cart(alpha, beta, r)	Spherical to cartesian coordinate transform.
spherical_hn2(n, z)	Spherical Hankel function of 2nd kind.
strict_arange(start, stop[, step, endpoint, ...])	Like numpy.arange()[65], but compensating numeric errors.
wavenumber(omega[, c])	Compute the wavenumber for a given radial frequency.
xyz_grid(x, y, z, *, spacing[, endpoint])	Create a grid with given range and spacing.

Classes

DelayedSignal(data, samplerate, time)	A tuple of audio data, sampling rate and start time.
XyzComponents(components)	Triple (or pair) of components: x, y, and optionally z.

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.

[63] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[64] https://docs.python.org/3/library/functions.html#zip
[65] https://numpy.org/doc/stable/reference/generated/numpy.arange.html#numpy.arange

Returns (3, 3) `numpy.ndarray`[66] – 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.

$$
\begin{aligned}
x &= r \cos \alpha \sin \beta \\
y &= r \sin \alpha \sin \beta \\
z &= r \cos \beta
\end{aligned}
$$

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

Parameters

- **alpha** (*float or array_like*) – Azimuth angle in radiants
- **beta** (*float or array_like*) – Colatitude angle in radiants (with 0 denoting North pole)
- **r** (*float or array_like*) – Radius

Returns

- **x** (float or `numpy.ndarray`[67]) – x-component of Cartesian coordinates
- **y** (float or `numpy.ndarray`[68]) – y-component of Cartesian coordinates
- **z** (float or `numpy.ndarray`[69]) – z-component of Cartesian coordinates

`sfs.util.`**`cart2sph`**(*x*, *y*, *z*)

Cartesian to spherical coordinate transform.

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

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

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 `numpy.ndarray`[70]) – Azimuth angle in radiants
- **beta** (float or `numpy.ndarray`[71]) – Colatitude angle in radiants (with 0 denoting North pole)
- **r** (float or `numpy.ndarray`[72]) – Radius

[66] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[67] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[68] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[69] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray

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

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

Returns *a* converted to a `numpy.ndarray`[73] 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`[74] 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`[75]s.

The *components* are first converted to NumPy arrays (using `numpy.asarray()`[76]) 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()`[77], 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()`[78].

Returns *`DelayedSignal`* – A named tuple consisting of a `numpy.ndarray`[79] 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:

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

(continues on next page)

[70] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[71] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[72] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[73] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[74] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[75] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[76] https://numpy.org/doc/stable/reference/generated/numpy.asarray.html#numpy.asarray
[77] https://numpy.org/doc/stable/reference/generated/numpy.asarray.html#numpy.asarray

(continued from previous page)

```
>>> signal_offset
0
```

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

Like `numpy.arange()`[80], but compensating numeric errors.

Unlike `numpy.arange()`[81], but similar to `numpy.linspace()`[82], providing `endpoint=True` includes both endpoints.

Parameters

- **start, stop, step, dtype** – See `numpy.arange()`[83].
- **endpoint** – See `numpy.linspace()`[84].

  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()`[85].

Returns `numpy.ndarray`[86] – Array of evenly spaced values. See `numpy.arange()`[87].

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()`[88]. See *`strict_arange()`*.
- ****kwargs** – All further arguments are forwarded to *`strict_arange()`*.

Returns *`XyzComponents`* – A grid that can be used for sound field calculations.

See also:

`strict_arange`, `numpy.meshgrid`[89]

sfs.util.**normalize**(*p*, *grid*, *xnorm*)

Normalize sound field wrt position *xnorm*.

[78] https://numpy.org/doc/stable/reference/generated/numpy.asarray.html#numpy.asarray
[79] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[80] https://numpy.org/doc/stable/reference/generated/numpy.arange.html#numpy.arange
[81] https://numpy.org/doc/stable/reference/generated/numpy.arange.html#numpy.arange
[82] https://numpy.org/doc/stable/reference/generated/numpy.linspace.html#numpy.linspace
[83] https://numpy.org/doc/stable/reference/generated/numpy.arange.html#numpy.arange
[84] https://numpy.org/doc/stable/reference/generated/numpy.linspace.html#numpy.linspace
[85] https://numpy.org/doc/stable/reference/generated/numpy.isclose.html#numpy.isclose
[86] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[87] https://numpy.org/doc/stable/reference/generated/numpy.arange.html#numpy.arange
[88] https://numpy.org/doc/stable/reference/generated/numpy.arange.html#numpy.arange
[89] https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html#numpy.meshgrid

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()[90].

sfs.util.**normalize_vector**(*x*)
Normalize a 1D vector.

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[91]. It is one-dimensional (like a plain list[92]) 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[93]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[94] 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.

property x
x-component.

property y
y-component.

property z
z-component (optional).

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**(*data*, *samplerate*, *time*)
A tuple of audio data, sampling rate and start time.

This class (a collections.namedtuple[95]) is not meant to be instantiated by users.

To pass a signal to a function, just use a simple tuple[96] or list[97] containing the audio data and the sampling rate (in Hertz), with an optional starting time (in seconds) as a third

[90] https://docs.python.org/3/library/functions.html#zip
[91] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[92] https://docs.python.org/3/library/stdtypes.html#list
[93] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[94] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray

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) numpy.ndarray[98]) – original & image source locations.
- **wall_count** ((M, 2D) numpy.ndarray[99]) – number of reflections at individual walls for each source.

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

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

$$h_n^{(2)}(z) = \sqrt{\frac{\pi}{2z}} H_{n+\frac{1}{2}}^{(2)}(z),$$

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

Parameters

- **n** (*array_like*) – Order of the spherical Hankel function (n >= 0).
- **z** (*array_like*) – Argument of the spherical Hankel function.

sfs.util.**source_selection_plane**(*n0*, *n*)
Secondary source selection for a plane wave.

Eq.(13) from [SRA08]

95 https://docs.python.org/3/library/collections.html#collections.namedtuple
96 https://docs.python.org/3/library/stdtypes.html#tuple
97 https://docs.python.org/3/library/stdtypes.html#list
98 https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
99 https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray

`sfs.util.`**`source_selection_point`**`(`*n0, x0, xs*`)`

Secondary source selection for a point source.

Eq.(15) from [SRA08]

`sfs.util.`**`source_selection_line`**`(`*n0, x0, xs*`)`

Secondary source selection for a line source.

compare Eq.(15) from [SRA08]

`sfs.util.`**`source_selection_focused`**`(`*ns, x0, xs*`)`

Secondary source selection for a focused source.

Eq.(2.78) from [Wie14]

`sfs.util.`**`source_selection_all`**`(`*N*`)`

Select all secondary sources.

`sfs.util.`**`max_order_circular_harmonics`**`(`*N*`)`

Maximum order of 2D/2.5D HOA.

It returns the maximum order for which no spatial aliasing appears. It is given on page 132 of [Ahr12] as

$$\mathtt{max\_order} = \begin{cases} N/2 - 1 & \text{even } N \\ (N-1)/2 & \text{odd } N, \end{cases}$$

which is equivalent to

$$\mathtt{max\_order} = \lfloor \frac{N-1}{2} \rfloor.$$

Parameters **N** (*int*) – Number of secondary sources.

`sfs.util.`**`max_order_spherical_harmonics`**`(`*N*`)`

Maximum order of 3D HOA.

$$\mathtt{max\_order} = \lfloor \sqrt{N} \rfloor - 1.$$

Parameters **N** (*int*) – Number of secondary sources.

`class` `sfs.`**`default`**

Get/set defaults for the *sfs* module.

For example, when you want to change the default speed of sound:

```
import sfs
sfs.default.c = 330
```

`c = 343`

Speed of sound.

`rho0 = 1.225`

Static density of air.

`selection_tolerance = 1e-06`

Tolerance used for secondary source selection.

`reset``()`

Reset all attributes to their “factory default”.

4 References

5 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!

5.1 Development Installation

Instead of pip-installing the latest release from PyPI[107], you should get the newest development version from Github[108]:

```
git clone https://github.com/sfstoolbox/sfs-python.git
cd sfs-python
python3 -m pip install --user -e .
```

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

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

5.2 Building the Documentation

If you make changes to the documentation, you can re-create the HTML pages using Sphinx[109]. 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/`.

To create the EPUB file, use:

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

The generated EPUB file will be available in the directory `build/sphinx/epub/`.

To create the PDF file, use:

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

Afterwards go to the folder `build/sphinx/latex/` and run LaTeX to create the PDF file. If you don't know how to create a PDF file from the LaTeX output, you should have a look at Latexmk[110] (see also this Latexmk tutorial[111]).

[107] https://pypi.org/project/sfs/
[108] https://github.com/sfstoolbox/sfs-python/
[109] http://sphinx-doc.org/
[110] http://personal.psu.edu/jcc8/software/latexmk-jcc/
[111] https://mg.readthedocs.io/latexmk.html

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

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

5.3 Running the Tests

You'll need pytest[112] for that. It can be installed with:

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

To execute the tests, simply run:

```
python3 -m pytest
```

5.4 Creating a New Release

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[113] with twine[114]: `python3 -m twine upload dist/*`
10. Push the commit and the tag to Github and add release notes[115] containing a link to PyPI and the bullet points from `NEWS.rst`
11. Check that the new release was built correctly on RTD[116] and select the new release as default version

6 Version History

Version 0.6.2 (2021-06-05):

- build doc fix, use sphinx4, mathjax2, html_css_files

Version 0.6.1 (2021-06-05):

- New default driving function for *`sfs.td.wfs.point_25d()`* for reference curve

Version 0.6.0 (2020-12-01):

- New function *`sfs.fd.source.line_bandlimited()`* computing the sound field of a spatially bandlimited line source

---

[112] https://pytest.org/
[113] https://pypi.org/project/sfs/
[114] https://twine.readthedocs.io/
[115] https://github.com/sfstoolbox/sfs-python/tags
[116] https://readthedocs.org/projects/sfs-python/builds/

- Drop support for Python 3.5

Version 0.5.0 (2019-03-18):

- Switching to separate *sfs.plot2d* and *sfs.plot3d* for plotting functions
- Move `sfs.util.displacement()` to *sfs.fd.displacement()*
- Switch to keyword only arguments
- New default driving function for *sfs.fd.wfs.point_25d()*
- New driving function syntax, e.g. *sfs.fd.wfs.point_25d()*
- Example for the sound field of a pulsating sphere
- Add time domain NFC-HOA driving functions *sfs.td.nfchoa*
- *sfs.fd.synthesize()*, *sfs.td.synthesize()* for soundfield superposition
- Change `sfs.mono` to *sfs.fd* and `sfs.time` to *sfs.td*
- Move source selection helpers to *sfs.util*
- Use *sfs.default* object instead of `sfs.defs` submodule
- Drop support for legacy Python 2.7

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
- *sfs.util.DelayedSignal* class and *sfs.util.as_delayed_signal()*
- Improvements to the documentation
- Start using Jupyter notebooks for examples in documentation
- Spherical Hankel function as *sfs.util.spherical_hn2()*
- Use `scipy.special.spherical_jn`[117], `scipy.special.spherical_yn`[118] instead of `scipy.special.sph_jnyn`
- Generalization of the modal order argument in `sfs.mono.source.point_modal()`
- Rename `sfs.util.normal_vector()` to *sfs.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 *sfs.util.cart2sph()*
- Fix *sfs.tapering.tukey()* for `alpha=1`

Version 0.3.1 (2016-04-08):

- Fixed metadata of release

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 appoach
- Driving functions for the synthesis of focused sources by WFS

[117] https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.spherical_jn.html#scipy.special.spherical_jn
[118] https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.spherical_yn.html#scipy.special.spherical_yn

Version 0.2.0 (2015-12-11):

- Ability to calculate and plot particle velocity and displacement fields
- Several function name and parameter name changes

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

[Ahr12] J. Ahrens. *Analytic Methods of Sound Field Synthesis*. Springer, Berlin Heidelberg, 2012. doi:10.1007/978-3-642-25743-8[100].

[AB79] J. B. Allen and D. A. Berkley. Image method for efficiently simulating small-room acoustics. *Journal of the Acoustical Society of America*, 65:943–950, 1979. doi:10.1121/1.382599[101].

[Bor84] J. Borish. Extension of the image model to arbitrary polyhedra. *Journal of the Acoustical Society of America*, 75:1827–1836, 1984. doi:10.1121/1.390983[102].

[FFSS17] Gergely Firtha, Péter Fiala, Frank Schultz, and Sascha Spors. Improved Referencing Schemes for 2.5D Wave Field Synthesis Driving Functions. *IEEE/ACM Trans. Audio Speech Language Process.*, 25(5):1117–1127, 2017. doi:10.1109/TASLP.2017.2689245[103].

[Mos12] M. Möser. *Technische Akustik*. Springer, Berlin Heidelberg, 2012. doi:10.1007/978-3-642-30933-5[104].

[Sch16] Frank Schultz. *Sound Field Synthesis for Line Source Array Applications in Large-Scale Sound Reinforcement*. PhD thesis, University of Rostock, 2016. doi:10.18453/rosdok_id00001765[105].

[SA09] S. Spors and J. Ahrens. Spatial Sampling Artifacts of Wave Field Synthesis for the Reproduction of Virtual Point Sources. In *126th Convention of the Audio Engineering Society*. 2009. URL: http://bit.ly/2jkfboi.

[SA10] S. Spors and J. Ahrens. Analysis and Improvement of Pre-equalization in 2.5-dimensional Wave Field Synthesis. In *128th Convention of the Audio Engineering Society*. 2010. URL: http://bit.ly/2Ad6RRR.

[SRA08] S. Spors, R. Rabenstein, and J. Ahrens. The Theory of Wave Field Synthesis Revisited. In *124th Convention of the Audio Engineering Society*. 2008. URL: http://bit.ly/2ByRjnB.

[SSR16] S. Spors, F. Schultz, and T. Rettberg. Improved Driving Functions for Rectangular Loudspeaker Arrays Driven by Sound Field Synthesis. In *42nd German Annual Conference on Acoustics (DAGA)*. 2016. URL: http://bit.ly/2AWR07G.

[Sta97] Evert W. Start. *Direct Sound Enhancement by Wave Field Synthesis*. PhD thesis, Delft University of Technology, 1997.

[Wie14] H. Wierstorf. *Perceptual Assessment of Sound Field Synthesis*. PhD thesis, Technische Universität Berlin, 2014. doi:10.14279/depositonce-4310[106].

[100] https://doi.org/10.1007/978-3-642-25743-8
[101] https://doi.org/10.1121/1.382599
[102] https://doi.org/10.1121/1.390983
[103] https://doi.org/10.1109/TASLP.2017.2689245
[104] https://doi.org/10.1007/978-3-642-30933-5
[105] https://doi.org/10.18453/rosdok_id00001765
[106] https://doi.org/10.14279/depositonce-4310