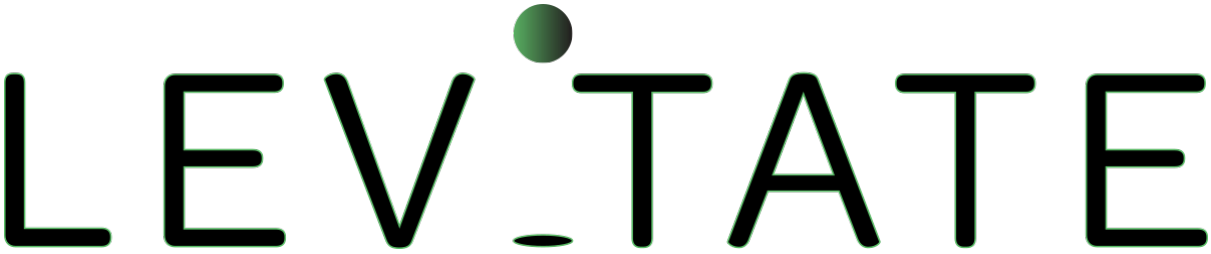

Levitate Python Toolbox

Release 3.0.0

Carl Andersson

Mar 02, 2022

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 737087.

Contents

1 Introduction

This is the documentation for the Levitate research project python toolbox. The toolbox is distributed as an open source python package, hosted on the Chalmers Applied Acoustics GitHub (https://github.com/AppliedAcousticsChalmers/levitate). The primary goal of this toolbox is to provide a collection of algorithms and design patterns to aid researchers working with acoustic levitation and related topics, e.g. mid-air haptic feedback or parametric audio. Included are both basic building blocks for simulating ultrasonic transducer arrays, and beamforming algorithms to design sound fields for specific purposes.

The package targets two major groups: Researchers who primarily focus on developing new algorithms used to design the sound fields, and researchers who use the existing algorithms to investigate areas of application, e.g. within human-computer interaction. The first group requires the possibility of fast prototyping of new algorithms or schemes. The inherent transparency in the python language together with the flexible and extensible design of the toolbox fulfills this requirement. The second group needs simple and reliable tools to quickly design a sound field according to the needs of the application. This is covered by the variety of algorithms existing in the toolbox, and the ease at which they can be applied in varying configurations. Not considered at this point are end-users. The tools still require significant knowledge of the operator and, to a certain degree, understanding of the physical limitations.

2 Package Organization and Functionality

This section covers a broad overview of the package, for a more detailed description see the sections in the full *API Documentation*. The package is created using the de facto numerical and scientific computing libraries for python, Numpy and Scipy. There are three major parts to the toolbox: models, algorithms, and visualization.

2.1 Models

The primary responsibility of the models component is to provide means for easy handling of virtual ultrasonic transducer arrays and their elements. This is the foundation on which the other parts build, with methods to calculate sound fields from arbitrary arrays arrangements. This part of the package in class-oriented, organized in two python modules. The `arrays` module with the primary class `TransducerArray` handles collections of individual transducer elements, i.e. arrays, operating all elements as a whole. The `transducers` module with the primary class `TransducerModel` handles single transducers and their radiation characteristics.

2.2 Algorithms

The algorithms component of the package collects the sound field design methods under a unified framework. The current implementation covers optimization of transducer array amplitudes and phases according to a set of cost functions. All the cost functions have a physical interpretation such as the force on a small bead of a specific size, or the sound pressure at a certain point.

2.3 Visualization

Visualization of data is a complicated topic, and visualization in python in particular comes in many flavors. For the purposes of this toolbox, the main concern is to quickly verify that the design algorithms deliver the intended result. To this end, a small set of tools are implemented using the Plotly graphing library, which has good support for interactive visualizations. The current implementation is focused on providing representations of the sound field generated by an array.

2.4 Workflow

Designing a sound field for a specific application involves choosing appropriate combinations of cost functions and optimization routines. A typical workflow from application specification to result is:

1) Specify what the sound field should do; *Levitate a bead.*
2) Find the appropriate mathematical formulation and choose the equivalent cost functions; *Maximize the Gor'kov Laplacian and minimize the sound pressure*
3) Choose an optimization routine; *minimization with fixed amplitudes*
4) Run the optimization.
5) Visualize the resulting sound field.
6) Export the result to use with a physical array.

For more detailed descriptions of workflows, refer to the *Examples*.

3 Examples

3.1 Simple trap optimization.

A very basic use-case, finding the correct phases to levitate a bead centered 5 cm above a 9x9 element rectangular array, then inspecting the resultant field.

[1]:
```
import numpy as np
import levitate
```

We define a target trap position and a transducer array. The optimizaiton typically converges from random initialization, but we can help it on the way by initializing close to a known nice solution.

[2]:
```
pos = np.array([0, 0, 80e-3])
array = levitate.arrays.RectangularArray(9)
phases = array.focus_phases(pos) + array.signature(stype='twin') + 0.2 * np.random.
↪uniform(-np.pi, np.pi, array.num_transducers)
start = levitate.complex(phases)
```

To find the suitable state of the transducer array, we define a cost function that we minimize with a BFGS-variant optimization.

[3]:
```
point = (levitate.fields.GorkovLaplacian(array) * (-100, -100, -1)).sum() +␣
↪abs(levitate.fields.Pressure(array))**2 * 1e-3
results = levitate.optimization.minimize(point@pos, array, start_values=start)
```

Finally, we visualize the sound field.

[4]:
```
array.visualize[0] = ['Signature', pos]
array.visualize.append('Pressure')
array.visualize(results).show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

3.2 Complicated transducer array setup

The setup shown here is a doublesided array where the two halves are standing vertically 3 cm above a reflecting surface.

In this example no optimization is done, but all optimization functions support complex arrangements like this one.

[1]:
```
import numpy as np
import levitate
```

Reflections from planar reflections are handled as a transducer object. In this case, we wrap a `CircularPiston` object, to include some directivity as well.

[2]:
```
transducer = levitate.transducers.TransducerReflector(
    levitate.transducers.CircularPiston, effective_radius=3e-3,
    plane_intersect=(0, 0, 0), plane_normal=(0, 0, 1))
```

The transducer array is created by using the `DoublesidedArray` class, which takes the type of array to use as the singlesided template as one of the inputs.

[3]:
```
array = levitate.arrays.DoublesidedArray(
    levitate.arrays.RectangularArray, separation=200e-3,
    normal=(1, 0, 0), offset=(0, 0, 50e-3),
    shape=(5, 10), transducer=transducer
)
```

We visualize the sound pressure field, as well as the velocity magnitude field.

[4]:
```
phases = array.focus_phases(np.array([25e-3, 0, 40e-3]))
amps = levitate.complex(phases)
array.visualize.zlimits = (0, 0.1)
array.visualize.append('Pressure')
array.visualize.append('Velocity')
array.visualize(amps).show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

3.3 Superposition of two fields

A more advanced usage, designed to create a field with a levitation trap and a haptics focus point.

[1]:
```
import numpy as np
import levitate
```

[2]:
```
array = levitate.arrays.RectangularArray((21, 12))
trap_pos = np.array([-20e-3, 0, 60e-3])
haptics_pos = np.array([40e-3, 0, 90e-3])
phases = array.focus_phases(trap_pos) + array.signature(trap_pos, stype='twin') + 0.2␣
↪* np.random.uniform(-np.pi, np.pi, array.num_transducers)
start = levitate.complex(phases)
```

The fields are superposed using mutual quiet zones, created by minimizing the pressure and velocity at the secondary point in each field. We will need three fields, calculating the pressure magnitude, the velocity magnitude, and the stiffenss of the trap.

[3]:
```
p = abs(levitate.fields.Pressure(array))**2
v = (abs(levitate.fields.Velocity(array))**2).sum()
s = levitate.fields.RadiationForceStiffness(array).sum()
```

The levitation trap is found using a minimization sequence. First the phases are optimized for just a trap, then the phases and amplitudes are optimized to include the quiet zone.

[4]:
```
trap_result = levitate.optimization.minimize(
    [
        (s + p * 1)@trap_pos,
        (s + p)@trap_pos + (v * 1e3 + p)@haptics_pos
    ],
    array, start_values=start, variable_amplitudes=[False, True]
)[-1]
```

The haptics point can be created using a simple focusing algorithm, so we can optimize for the inclusion of the quiet zone straight away. To retain the focus point we set a negative weight for the pressure, i.e. maximizing the pressure.

[5]:
```
start = levitate.complex(array.focus_phases(haptics_pos))
haptics_result = levitate.optimization.minimize(
    p * (-1)@haptics_pos + (p + v * 1e3)@trap_pos,
    array, start_values=start, variable_amplitudes=True
)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/levitate/envs/stable/lib/python3.7/
↪site-packages/levitate/fields/_transformers.py:268: RuntimeWarning: invalid value␣
↪encountered in true_divide
  jacobians = jacobians * (np.conjugate(values) / abs_values)[self._val_reshape]
```

Finally, we visualize the individual fields, as well as the superposed field.

[6]:
```
array.visualize.append('pressure')
array.visualize(
    trap_result, haptics_result, haptics_result * 0.3 + trap_result * 0.7,
    labels=['Trap', 'Haptics', 'Combined']
).show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

3.4 Visualizing the force around a trap

This example show how to easily visualize the radiation force in the vicinity of an object.

[1]:
```
import numpy as np
import levitate
```

[2]:
```
pos = np.array([0, 0, 60e-3])
array = levitate.arrays.RectangularArray(16)
state = levitate.complex(array.focus_phases(pos) + array.signature(stype='twin'))
```

[3]:
```
radii = [1e-3, 2e-3, 4e-3, 8e-3, 16e-3]
for radius in radii:
```

(continues on next page)

(continued from previous page)

```
    array.force_diagram.append([pos, {'radius': radius, 'name': '{} mm'.format(radius␣
↪* 1e3)}])
array.force_diagram(state).show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

4 API Documentation

Levitate, a python package for simulating acoustic levitation using ultrasonic transducer arrays.

The API consists of four main modules, and a few supporting modules. The main modules contain models to handle transducers and transducer arrays, in the *transducers* and *arrays* modules respectively, algorithms to calculate physical properties in the *fields* module, and some numerical optimization functions in the *optimization* module. There is also a *visualizers* module with some convenience function to show various fields, and some analysis tools in *analysis*. It is possible to use different materials or material properties from the *materials* module.

The *hardware* module includes definitions with array geometries corresponding to some physical prototypes, and python-c++ combined setup to control Ultrahaptics physical hardware directly from python. This implementation of Ultrahaptics control from python is not officially supported by Ultrahaptics, and only enables a very limited subset of the research SDK.

4.1 Transducers

Handling of individual transducers and their directivities.

This module contains classes describing how individual transducer elements radiate sound, e.g. waveforms and directivities. This is also where the various spatial properties, e.g. derivatives, are implemented. Most calculations in this module are fully vectorized, so the models can calculate sound fields for any number of source positions and receiver positions at once.

TransducerModel	Base class for ultrasonic single frequency transducers.
PointSource	Point source transducers.
PlaneWaveTransducer	Class representing planar waves.
CircularPiston	Circular piston transducer model.
CircularRing	Circular ring transducer model.
TransducerReflector	Class for transducers with planar reflectors.

class levitate.transducers.**TransducerModel**(*freq=40000.0*, *p0=6*, *medium=Air(rho=1.204082071218662, dynamic_viscosity=1.85e-05, c=343.23714360505863)*, *physical_size=0.01*)

Base class for ultrasonic single frequency transducers.

Parameters

- **freq** (*float, default 40 kHz*) – The resonant frequency of the transducer.
- **p0** (*float, default 6 Pa*) – The sound pressure created at maximum amplitude at 1m distance, in Pa. Note: This is not an rms value!
- **medium** (Material) – The medium in which the array is operating.
- **physical_size** (*float, default 10e-3*) – The physical dimentions of the transducer. Mainly used for visualization and some geometrical assumptions.

Variables

- **~TransducerModel.k** (*float*) – Wavenumber in the medium.
- **~TransducerModel.wavelength** (*float*) – Wavelength in the medium.
- **~TransducerModel.omega** (*float*) – Angular frequency.
- **~TransducerModel.freq** (*float*) – Wave frequency.

pressure(*source_positions*, *source_normals*, *receiver_positions*, ***kwargs*)

Calculate the complex sound pressure from the transducer.

Parameters

- **source_positions** (*numpy.ndarray*) – The location of the transducer, as a (3, ...) shape array.
- **source_normals** (*numpy.ndarray*) – The look direction of the transducer, as a (3, ...) shape array.
- **receiver_positions** (*numpy.ndarray*) – The location(s) at which to evaluate the radiation, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.

Returns **out** (*numpy.ndarray*) – The pressure at the locations, shape `source_positions.shape[1:] + receiver_positions.shape[1:]`.

pressure_derivs(*source_positions*, *source_normals*, *receiver_positions*, *orders=3*, ***kwargs*)

Calculate the spatial derivatives of the greens function.

Calculates Cartesian spatial derivatives of the pressure Green's function. Should be implemented by concrete subclasses.

Parameters

- **source_positions** (*numpy.ndarray*) – The location of the transducer, as a (3, ...) shape array.
- **source_normals** (*numpy.ndarray*) – The look direction of the transducer, as a (3, ...) shape array.
- **receiver_positions** (*numpy.ndarray*) – The location(s) at which to evaluate the radiation, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.
- **orders** (*int*) – How many orders of derivatives to calculate. Currently three orders are supported.

Returns **derivatives** (*numpy.ndarray*) – Array with the calculated derivatives. Has the shape `(M,) + source_positions.shape[1:] + receiver_positions.shape[1:]`. where `M` is the number of spatial derivatives, see `num_spatial_derivatives` and `spatial_derivative_order`.

class levitate.transducers.**PointSource**(*freq=40000.0*, *p0=6*, *medium=Air(rho=1.2040820712186 62, dynamic_viscosity=1.85e-05, c=343.23714360505863)*, *physical_size=0.01*)

Point source transducers.

A point source is in this context defines as a spherically spreading wave, optionally with a directivity. On its own this class defines a monopole, but subclasses are free to change the directivity to other shapes.

The spherical spreading is defined as

$$G(r) = \frac{e^{ikr}}{r}$$

where r is the distance from the source, and k is the wavenumber of the wave.

directivity(*source_positions*, *source_normals*, *receiver_positions*)
Evaluate transducer directivity.

Subclasses will preferably implement this to create new directivity models. Default implementation is omnidirectional sources.

Parameters

- **source_positions** (*numpy.ndarray*) – The location of the transducer, as a (3, ...) shape array.
- **source_normals** (*numpy.ndarray*) – The look direction of the transducer, as a (3, ...) shape array.
- **receiver_positions** (*numpy.ndarray*) – The location(s) at which to evaluate the radiation, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.

Returns out (*numpy.ndarray*) – The amplitude (and phase) of the directivity, shape `source_positions.shape[1:] + receiver_positions.shape[1:]`.

pressure_derivs(*source_positions*, *source_normals*, *receiver_positions*, *orders=3*, ***kwargs*)
Calculate the spatial derivatives of the greens function.

This is the combination of the derivative of the spherical spreading, and the derivatives of the directivity, including source strength.

Parameters

- **source_positions** (*numpy.ndarray*) – The location of the transducer, as a (3, ...) shape array.
- **source_normals** (*numpy.ndarray*) – The look direction of the transducer, as a (3, ...) shape array.
- **receiver_positions** (*numpy.ndarray*) – The location(s) at which to evaluate the radiation, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.
- **orders** (*int*) – How many orders of derivatives to calculate. Currently three orders are supported.

Returns derivatives (*numpy.ndarray*) – Array with the calculated derivatives. Has the `(M, ) + source_positions.shape[1:] + receiver_positions.shape[1:]`. where `M` is the number of spatial derivatives, see `num_spatial_derivatives` and `spatial_derivative_order`.

wavefront_derivatives(*source_positions*, *receiver_positions*, *orders=3*)
Calculate the spatial derivatives of the spherical spreading.

Parameters

- **source_positions** (*numpy.ndarray*) – The location of the transducer, as a (3, ...) shape array.
- **receiver_positions** (*numpy.ndarray*) – The location(s) at which to evaluate the radiation, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.
- **orders** (*int*) – How many orders of derivatives to calculate. Currently three orders are supported.

Returns derivatives (*ndarray*) – Array with the calculated derivatives. Has the shape `(M, ) + source_positions.shape[1:] + receiver_positions.shape[1:]`. where `M` is the number of spatial derivatives, see `num_spatial_derivatives` and `spatial_derivative_order`.

directivity_derivatives(*source_positions*, *source_normals*, *receiver_positions*, *orders=3*)
Calculate the spatial derivatives of the directivity.

The default implementation uses finite difference stencils to evaluate the derivatives. In principle this means that customized directivity models does not need to implement their own derivatives, but can do so for speed and precision benefits.

Parameters

- **source_positions** (*numpy.ndarray*) – The location of the transducer, as a (3, ...) shape array.
- **source_normals** (*numpy.ndarray*) – The look direction of the transducer, as a (3, ...) shape array.
- **receiver_positions** (*numpy.ndarray*) – The location(s) at which to evaluate the radiation, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.
- **orders** (*int*) – How many orders of derivatives to calculate. Currently three orders are supported.

Returns derivatives (*numpy.ndarray*) – Array with the calculated derivatives. Has the shape `(M,) + source_positions.shape[1:] + receiver_positions.shape[1:]`. where `M` is the number of spatial derivatives, see `num_spatial_derivatives` and `spatial_derivative_order`.

spherical_harmonics(*source_positions*, *source_normals*, *receiver_positions*, *orders=0*, ***kwargs*)
Expand sound field in spherical harmonics.

Performs a spherical harmonics expansion of the sound field created from the transducer model. The expansion is centered at the receiver position(s), and calculated by translating spherical wavefronts from the source position(s).

Parameters

- **source_positions** (*numpy.ndarray*) – The location of the transducer, as a (3, ...) shape array.
- **source_normals** (*numpy.ndarray*) – The look direction of the transducer, as a (3, ...) shape array.
- **receiver_positions** (*numpy.ndarray*) – The location(s) at which to evaluate the radiation, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.
- **orders** (*int*) – How many orders of spherical harmonics coefficients to calculate.

Returns coefficients (*numpy.ndarray*) – Array with the calculated expansion coefficients. Has the shape `(M,) + source_positions.shape[1:] + receiver_positions.shape[1:]`, where `M=len(SphericalHarmonicsIndexer(orders))`, see `SphericalHarmonicsIndexer` for details on the structure of the coefficients.

class levitate.transducers.**TransducerReflector**(*transducer*, *plane_intersect=(0, 0, 0)*, *plane_normal=(0, 0, 1)*, *reflection_coefficient=1*, **args*, ***kwargs*)
Class for transducers with planar reflectors.

This class can be used to add reflectors to all transducer models. This uses the image source method, so only infinite planar reflectors are possible.

Parameters

- **transducer** (`TrnsducerModel` instance or (sub)class) – The base transducer to reflect. If passed a class it will be instantiated with the remaining arguments not used by the reflector.

- **plane_intersect** (*array_like, default (0, 0, 0)*) – A point which the reflection plane intersects.
- **plane_normal** (*array_like, default (0,0,1)*) – 3 element vector with the plane normal.
- **reflection_coefficient** (*complex float, default 1*) – Reflection coefficient to tune the magnitude and phase of the reflection.

Returns *transducer* – The transducer model with reflections.

pressure_derivs(*source_positions*, *source_normals*, *receiver_positions*, **args*, ***kwargs*)
Calculate the spatial derivatives of the greens function.

Parameters

- **source_positions** (*numpy.ndarray*) – The location of the transducer, as a (3, ...) shape array.
- **source_normals** (*numpy.ndarray*) – The look direction of the transducer, as a (3, ...) shape array.
- **receiver_positions** (*numpy.ndarray*) – The location(s) at which to evaluate the radiation, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.
- **orders** (*int*) – How many orders of derivatives to calculate. Currently three orders are supported.

Returns **derivatives** (*numpy.ndarray*) – Array with the calculated derivatives. Has the shape `(M,) + source_positions.shape[1:] + receiver_positions.shape[1:]`. where `M` is the number of spatial derivatives, see `num_spatial_derivatives` and `spatial_derivative_order`.

spherical_harmonics(*source_positions*, *source_normals*, *receiver_positions*, **args*, ***kwargs*)
Evaluate the spherical harmonics expansion at a point.

Mirrors the sources in the reflection plane and calculates the superposition of the expansions from the combined sources. For the full documentation of the parameters and output format, see the documentation of the spherical harmonics method of the underlying transducer model.

class levitate.transducers.**PlaneWaveTransducer**(*freq=40000.0*, *p0=6*, *medium=Air(rho=1.204082071218662, dynamic_viscosity=1.85e-05, c=343.23714360505863)*, *physical_size=0.01*)
Class representing planar waves.

This is not representing a physical transducer per se, but a traveling plane wave.

pressure_derivs(*source_positions*, *source_normals*, *receiver_positions*, *orders=3*, ***kwargs*)
Calculate the spatial derivatives of the greens function.

Parameters

- **source_positions** (*numpy.ndarray*) – The location of the transducer, as a (3, ...) shape array.
- **source_normals** (*numpy.ndarray*) – The look direction of the transducer, as a (3, ...) shape array.
- **receiver_positions** (*numpy.ndarray*) – The location(s) at which to evaluate the radiation, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.
- **orders** (*int*) – How many orders of derivatives to calculate. Currently three orders are supported.

Returns **derivatives** (*numpy.ndarray*) – Array with the calculated derivatives. Has the shape `((M,) + source_positions.shape[1:] + receiver_positions.shape[1:]`. where `M` is the number of spatial derivatives, see `num_spatial_derivatives` and `spatial_derivative_order`.

class levitate.transducers.**CircularPiston**(*effective_radius*, **args*, ***kwargs*)

Circular piston transducer model.

Implementation of the circular piston directivity $D(\theta) = 2\frac{J_1(ka\sin\theta)}{ka\sin\theta}$.

Parameters

- **effective_radius** (*float*) – The radius a in the above.
- ****kwargs** – See *TransducerModel*

Note: This class has no implementation of analytic jacobians yet, and is much slower to use than other models.

---

directivity(*source_positions*, *source_normals*, *receiver_positions*)

Evaluate transducer directivity.

Returns $D(\theta) = 2J_1(ka\sin\theta)/(ka\sin\theta)$ where a is the `effective_radius` of the transducer, k is the wavenumber of the transducer (`k`), θ is the angle between the transducer normal and the vector from the transducer to the receiving point, and and J_1 is the first order Bessel function.

Parameters

- **source_positions** (*numpy.ndarray*) – The location of the transducer, as a (3, ...) shape array.
- **source_normals** (*numpy.ndarray*) – The look direction of the transducer, as a (3, ...) shape array.
- **receiver_positions** (*numpy.ndarray*) – The location(s) at which to evaluate the radiation, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.

Returns **out** (*numpy.ndarray*) – The amplitude (and phase) of the directivity, shape `source_positions.shape[1:] + receiver_positions.shape[1:]`.

class levitate.transducers.**CircularRing**(*effective_radius*, **args*, ***kwargs*)

Circular ring transducer model.

Implementation of the circular ring directivity $D(\theta) = J_0(ka\sin\theta)$.

Parameters

- **effective_radius** (*float*) – The radius a in the above.
- ****kwargs** – See *TransducerModel*

directivity(*source_positions*, *source_normals*, *receiver_positions*)

Evaluate transducer directivity.

Returns $D(\theta) = J_0(ka\sin\theta)$ where a is the `effective_radius` of the transducer, k is the wavenumber of the transducer (`k`), θ is the angle between the transducer normal and the vector from the transducer to the receiving point, and J_0 is the zeroth order Bessel function.

Parameters

- **source_positions** (*numpy.ndarray*) – The location of the transducer, as a (3, ...) shape array.
- **source_normals** (*numpy.ndarray*) – The look direction of the transducer, as a (3, ...) shape array.

- **receiver_positions** (*numpy.ndarray*) – The location(s) at which to evaluate the radiation, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.

Returns out (*numpy.ndarray*) – The amplitude (and phase) of the directivity, shape `source_positions.shape[1:] + receiver_positions.shape[1:]`.

directivity_derivatives(*source_positions*, *source_normals*, *receiver_positions*, *orders=3*)

Calculate the spatial derivatives of the directivity.

Explicit implementation of the derivatives of the directivity, based on analytical differentiation.

Parameters

- **source_positions** (*numpy.ndarray*) – The location of the transducer, as a (3, ...) shape array.
- **source_normals** (*numpy.ndarray*) – The look direction of the transducer, as a (3, ...) shape array.
- **receiver_positions** (*numpy.ndarray*) – The location(s) at which to evaluate the radiation, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.
- **orders** (*int*) – How many orders of derivatives to calculate. Currently three orders are supported.

Returns derivatives (*numpy.ndarray*) – Array with the calculated derivatives. Has the shape `(M,) + source_positions.shape[1:] + receiver_positions.shape[1:]`. where `M` is the number of spatial derivatives, see `num_spatial_derivatives` and `spatial_derivative_order`.

4.2 Arrays

Handling of transducer arrays, grouping multiple transducer elements.

The main class is the *TransducerArray* class, but other classes exist to simplify the creation of the transducer positions for common array geometries.

TransducerArray	Base class to handle transducer arrays.
NormalTransducerArray	Transducer array with a clearly defined normal.
RectangularArray	TransducerArray implementation for rectangular arrays.
SphericalCapArray	Transducer array implementation for spherical caps.
DoublesidedArray	TransducerArray implementation for doublesided arrays.

class levitate.arrays.**TransducerArray**(*positions*, *normals*, *transducer=None*, *medium=None*, ***kwargs*)

Base class to handle transducer arrays.

This class has no notion of the layout. If possible, try to use a more specific implementation instead.

Parameters

- **positions** (*numpy.ndarray*) – The positions of the transducer elements in the array, shape 3xN.
- **normals** (*numpy.ndarray*) – The normals of the transducer elements in the array, shape 3xN.
- **transducer** – An object of *levitate.transducers.TransducerModel* or a subclass. If passed a class it will create a new instance.

- ****kwargs** – All additional keyword arguments will be passed to the a transducer class used when instantiating a new transducer model. Note that this will have no effect on already instantiated transducer models.

Variables

- **~TransducerArray.num_transducers** (*int*) – The number of transducers used.
- **~TransducerArray.positions** (*numpy.ndarray*) – As above.
- **~TransducerArray.normals** (*numpy.ndarray*) – As above.
- **~TransducerArray.transducer** (TransducerModel) – An instance of a specific transducer model implementation.
- **~TransducerArray.freq** (*float*) – Frequency of the transducer model.
- **~TransducerArray.omega** (*float*) – Angular frequency of the transducer model.
- **~TransducerArray.k** (*float*) – Wavenumber in air, corresponding to `freq`.
- **~TransducerArray.wavelength** (*float*) – Wavelength in air, corresponding to `freq`.

class ArrayVisualizer(*array*, **args*, ***kwargs*)

Visualizations of a trandcuer array.

It is possible to set an item using either just a trace specifier, e.g. "Pressure", which create the appropriate trace with default arguments. If arguments are required or wanted, set the item to a tuple where the first element is the trace specifier, and subsequent elements are the arguments. If the last element in the tuple is a dictionary, it will be used as keyword arguments for the trace type.

__call__(**complex_transducer_amplitudes*, ***kwargs*)

Call self as a function.

class ForceDiagram(**args*, *scale_to_gravity=True*, *include_gravity=True*, ***kwargs*)

__call__(**complex_transducer_amplitudes*, ***kwargs*)

Call self as a function.

focus_phases(*focus*)

Focuses the phases to create a focus point.

Parameters **focus** (*array_like*) – Three element array with a location where to focus.

Returns **phases** (*numpy.ndarray*) – Array with the phases for the transducer elements.

signature(*position*, *phases*, *stype=None*)

Calculate the phase signature of the array.

The signature of an array if the phase of the transducer elements when the phase required to focus all elements to a specific point has been removed.

Parameters

- **position** (*array_like*) – Three element array with a position for where the signature is relative to.
- **phases** (*numpy.ndarray*) – The phases of which to calculate the signature.

Returns **signature** (*numpy.ndarray*) – The signature wrapped to the interval [-pi, pi].

pressure_derivs(*positions*, *orders=3*)

Calculate derivatives of the pressure.

Calculates the spatial derivatives of the pressure from all individual transducers in a Cartesian coordinate system.

Parameters

- **positions** (*numpy.ndarray*) – The location(s) at which to evaluate the derivatives, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.
- **orders** (*int*) – How many orders of derivatives to calculate. Currently three orders are supported.

Returns **derivatives** (*ndarray*) – Array with the calculated derivatives. Has the shape (M, N, ...) where M is the number of spatial derivatives, and N is the number of transducers, see `num_spatial_derivatives` and `spatial_derivative_order`, and the remaining dimensions are the same as the `positions` input with the first dimension removed.

spherical_harmonics(*positions*, *orders=0*)

Spherical harmonics expansion of transducer sound fields.

The sound fields generated by the individual transducers in the array are expanded in spherical harmonics around the positions specified. The coefficients are calculated using analytical translation of the transducer radiation patterns. This is a simplified calculation which will not account for the local directivity curve, only an overall scaling for each transducer-position combination.

Parameters

- **positions** (*numpy.ndarray*) – The location(s) at which to evaluate the derivatives, shape (3, ...). The first dimension must have length 3 and represent the coordinates of the points.
- **orders** (*int, default 0*) – The maximum order to expand to.

Returns **spherical_harmonics_coefficients** (*numpy.ndarray*) – Array with the calculated expansion coefficients. The order of the coefficients are described in `SphericalHarmonicsIndexer`. Has shape (M, N, ...) where `M=len(SphericalHarmonicsIndexer(orders))`, `N` is the number of transducers in the array, and the remaining dimensions are the same as the `positions` input with the first dimension removed.

request(*requests*, *position*)

Evaluate a set of requests.

This takes a mapping (e.g. dict) of requests, and evaluates them at a given position. This is independent of the current transducer state. If a certain quantity should be calculated with regards to the current transducer state, use a `FieldImplementation` from the `fields` module.

Parameters

- **position** (*ndarray*) – The position where to calculate the requirements needed, shape (3,...).
- **requests** (*mapping, e.g. dict*) – A mapping of the desired requests. The keys in the mapping should start with the desired output, and the value indicates some kind of parameter set. Possible requests listed below:

  pressure_derivs A number of spatial derivatives of the pressure. Should contain the maximum order of differentiation, see *pressure_derivs*.

  spherical_harmonics Spherical harmonics coefficients for an expansion of the pressure. Should contain the maximum order of expansion, see *spherical_harmonics*.

Returns **evaluated_requests** (*dict*) – A dictionary of the set of calculated data, according to the requests.

class levitate.arrays.**NormalTransducerArray**(*positions*, *normals*, *offset=(0, 0, 0)*, *normal=(0, 0, 1)*, *rotation=0*, ***kwargs*)

Transducer array with a clearly defined normal.

This is mostly intended as a base class for other implementations. The advantage is that a simple arrangement can be created assuming a normal along the z-axis, which is then rotated and moved to the desired orientation.

The positions and normals of the transducers should be input assuming that the overall normal for the array is along the z-axis. The positions and normals will be rotated around the origin to give the desired overall normal. This rotation will take place along the intersection line of the plane specificed by the desired normal, and the xy-plane. If rotation is desired, the positions are further rotated using the normal as the rotation axis. Finally an offset is applied to the entire array.

Parameters

- **positions** (*numpy.ndarray*) – The positions of the transducer elements in the array, shape 3xN.
- **normals** (*numpy.ndarray*) – The normals of the transducer elements in the array, shape 3xN (or 3 elements which will broadcast).
- **offset** (*3 element array_like, default (0, 0, 0)*) – The location of the center of the array.
- **normal** (*3 element array_like, default (0, 0, 1)*) – The normal of the overall array.
- **rotation** (*float, default 0*) – The in-plane rotation of the array around the normal.

signature(*position=None*, **args*, *stype=None*, ***kwargs*)

Calculate phase signatures of the array.

The signature of an array if the phase of the transducer elements when the phase required to focus all elements to a specific point has been removed. If `stype` if set to one of the available signatures: 'twin', 'vortex', or 'bottle', the corresponding signature is returned.

The signatures and the additional keyword parameters for them are:

Current signature (`stype=None`) Calculates the current phase signature. See *TransducerArray.signature*

phases (`numpy.ndarray`, optional) The phases of which to calculate the signature. Will default to the current phases in the array.

Twin signature (`stype='twin'`) Calculates the twin trap signature which shifts the phase of half of the elements by pi, splitting the array along a straight line.

angle (`float`, optional) The angle between the x-axis and the dividing line. Default is to create a line perpendicular to the line from the center of the array to `position`.

Vortex signature (`stype='vortex'`) Calculates the vortex trap signature which phase shifts the elements in the array according to their angle in the coordinate plane.

angle (`float`, optional) Additional angle to rotate the phase signature with.

Bottle signature (`stype='bottle'`) Calculates the bottle trap signature which phase shifts the elements in the array according to their distance from the center, creating an inner zone and an outer zone of equal area with a relative shift of pi.

radius (`float`, optional) A custom radius to use for the division of transducers. The default is to use equal area partition based on the rectangular area occupied by each transducer. This gives the same number of transducers in the two groups for square arrays.

Parameters

- **position** (*array_like*) – Three element array with a location for where the signature is relative to.
- **stype** (*None, 'twin', 'bottle', 'vortex'. Default None*) – Chooses which type of signature to calculate.

Returns **signature** (*numpy.ndarray*) – The signature wrapped to the interval [-pi, pi].

class levitate.arrays.**RectangularArray**(*shape=16*, *spread=0.01*, ***kwargs*)

TransducerArray implementation for rectangular arrays.

Defines the locations and normals of elements (transducers) in an array. See `NormaltransducerArray` for documentation of roration and transslation options.

Parameters

- **shape** (*int or (int, int), default 16*) – The number of transducer elements. Passing a single int will create a square array.
- **spread** (*float, default 10e-3*) – The distance between the array elements.

class levitate.arrays.**SphericalCapArray**(*radius*, *rings*, *spread=0.01*, *packing='distance'*, ***kwargs*)

Transducer array implementation for spherical caps.

The transducers will be placed on a virtual spherical surface, i.e. on the same distance from a given point in space. Control the overall shape of the array with the `radius`, `rings`, and `spead` parameters. See `NormalTransdcerArray` for details on the overall placement of the array, e.g. rotations and offsets.

There are many ways to pack transdcuers on a spherical surface. The 'distance' method will place the transducers on concentric rings where the distance between each ring is pre-determined. Each ring will have as many transducers as possible for the given ring size. This will typically pack the transducers densely, and the outer dimentions of the array is quite consistent. The 'count' method will use a pre-determined number of transducers in each ring, with 6 additional transducers for each successive ring. The inter-ring distance will be set to fit the requested number of transducers. This method will deliver a pre-determined number of transducers, but will not be as dense. If too many rings are requested, the 'count' method will fill a half-spere with transducers and then stop. The 'distance' method can fill the entire sphere with transducers.

Parameters

- **radius** (*float*) – The curvature of the spherical cap, i.e. how for away the focus is.
- **rings** (*int*) – Number of consecutive rings of transducers in the array.
- **packing** (*str, default 'distance'*) – Controlls which packing method is used. One of 'distance' or 'count', see above.
- **spread** (*float, default 10e-3*) – Controls the minimum spacing between individual transducers.

class levitate.arrays.**DoublesidedArray**(*array*, *separation*, *normal=(0, 0, 1)*, *offset=(0, 0, 0)*, *twist=0*, ***kwargs*)

TransducerArray implementation for doublesided arrays.

Creates a doublesided array based on mirroring a singlesided array. This can easily be used to create standard doublesided arrays by using the same normal for the mirroring as for the original array. If a different normal is used it is possible to create e.g. v-shaped arrays.

1) The singlesided array is "centered" at the origin, where "center" is defined as the mean coordinate of the elements.
2) The singlesided array is shifted with half of the separation in the opposite direction of the normal to create the "lower" half.
3) The "upper" half is created by mirroring the "lower" half in the plane described by the normal.
4) Both halves are offset with a specified vector.

Note that only the orientation of the initial array matters, not the overall position.

Parameters

- **array** (Instance or (sub)class of *TransducerArray*.) – The singlesided object used to the creation of the doublesided array. Classes will be instantiated to generate the array, using all input arguments except `array`, `separation`, and `offset`.
- **separation** (*float*) – The distance between the two halves, along the normal.

- **offset** (*array_like, 3 elements*) – The placement of the center between the two arrays.
- **normal** (*array_like, 3 elements*) – The normal of the reflection plane.
- **twist** (*float, default 0*) – By how much the two halves are rotated compared to each other, in radians.

signature(*position=None*, **args*, *stype=None*, ***kwargs*)

Calculate phase signatures of the array.

The signature of an array if the phase of the transducer elements when the phase required to focus all elements to a specific point has been removed. If `stype` if set to one of the available signatures the corresponding signature is returned. The signatures of the array used when creating the doublesided array are also available.

The signatures and the additional keyword parameters for them are:

Current signature (`stype=None`) Calculates the current phase signature. See *TransducerArray.signature*

phases (numpy.ndarray, optional) The phases of which to calculate the signature. Will default to the current phases in the array.

Doublesided signature (`stype='doublesided'`) Calculates the doublesided trap signature which shifts the phase of one side of the array half of the elements by pi.

Parameters

- **position** (*array_like*) – Three element array with a location for where the signature is relative to.
- **stype** (*None, 'doublesided', etc. Default None*) – Chooses which type of signature to calculate.

Returns **signature** (*numpy.ndarray*) – The signature wrapped to the interval [-pi, pi].

4.3 Fields

A collection of levitation related mathematical implementations.

The fields is one of the most important parts of the package, containing implementations of various ways to calculate levitate-related physical properties. To simplify the management and manipulation of the implemented fields they are wrapped in an additional abstraction layer. The short version is that the classes implemented in the *fields* module will not return objects of the called class, but typically objects of `Field`. These objects support algebraic operations, like +, *, and `abs`. The full description of what the different operands do can be found in the documentation of `_field_wrappers`.

References

Pressure	Complex sound pressure p.
Velocity	Complex sound particle velocity v.
GorkovPotential	Gor'kov's potential U.
GorkovGradient	Gradient of Gor'kov's potential, ∇U.
GorkovLaplacian	Laplacian of Gor'kov's potential, $\nabla^2 U$.
RadiationForce	Radiation force calculation for small beads in arbitrary sound fields.
RadiationForceStiffness	Radiation force gradient for small beads in arbitrary sound fields.
RadiationForceCurl	Curl or rotation of the radiation force.

continues on next page

Table 3 – continued from previous page

RadiationForceGradient	Full matrix gradient of the radiation force.
SphericalHarmonicsForce	Spherical harmonics based radiation force.
SphericalHarmonicsForceGradient	Spatial gradient of the total spherical radiation force.
SphericalHarmonicsExpansion	Spherical harmonics expansion coefficients of the sound pressure.
SphericalHarmonicsExpansionGradient	Spatial gradient of spherical harmonics expansion coefficients.

4.3.1 Sound Fields

class levitate.fields.**Pressure**(*array*)
Complex sound pressure p.

Calculates the complex-valued sound pressure.

Parameters **array** (*TransducerArray*) – The object modeling the array.

class levitate.fields.**Velocity**(*array*, **args*, ***kwargs*)
Complex sound particle velocity v.

Calculates the sound particle velocity

$$v = \frac{1}{j\omega\rho}\nabla p$$

from the relation $\dot{v} = \rho\nabla p$ applied for monofrequent sound fields. This is a vector value using a Cartesian coordinate system.

Parameters **array** (*TransducerArray*) – The object modeling the array.

class levitate.fields.**SphericalHarmonicsExpansion**(*array*, *orders*, **args*, ***kwargs*)
Spherical harmonics expansion coefficients of the sound pressure.

The expansion coefficients up to a certain order, where the complex amplitudes of the transducers will be accounted for.

Parameters

- **array** (*TransducerArray*) – The object modeling the array.
- **orders** (*int*) – The number of expansion orders to include.

class levitate.fields.**SphericalHarmonicsExpansionGradient**(*array*, *orders*, **args*, ***kwargs*)
Spatial gradient of spherical harmonics expansion coefficients.

Gives the Cartesian gradient of the expansion coefficient with respect to the expansion center. See *SphericalHarmonicsExpansion* for documentation of parameters.

Parameters

- **array** (*TransducerArray*) – The object modeling the array.
- **orders** (*int*) – The number of expansion orders to include.

4.3.2 Radiation Force

class levitate.fields.**RadiationForce**(*array*, *radius=0.001*, *material=Styrofoam(rho=25, poisson_ratio=0.35, c=2350)*, **args*, ***kwargs*)

Radiation force calculation for small beads in arbitrary sound fields.

Calculates the radiation force on a small particle in a sound field which can have both strong standing wave components or strong traveling wave components. The force components $q = x, y, z$ are calculated as

$$F_q = -\frac{\pi}{k^5}\kappa_0 \Re\left\{ ik^2\Psi_0 p\frac{\partial p^*}{\partial q} + ik^2\Psi_1 p^*\frac{\partial p}{\partial q} + 3i\Psi_1\left(\frac{\partial p}{\partial x}\frac{\partial^2 p^*}{\partial x \partial q} + \frac{\partial p}{\partial y}\frac{\partial^2 p^*}{\partial y \partial q} + \frac{\partial p}{\partial z}\frac{\partial^2 p^*}{\partial z \partial q}\right)\right\}$$

where

$$\Psi_0 = -\frac{2(ka)^6}{9}\left(f_1^2 + \frac{f_2^2}{4} + f_1 f_2\right) - i\frac{(ka)^3}{3}(2f_1 + f_2)$$
$$\Psi_1 = -\frac{(ka)^6}{18}f_2^2 + i\frac{(ka)^3}{3}f_2$$
$$f_1 = 1 - \frac{\kappa_p}{\kappa_0}, \qquad f_2 = 2\frac{\rho_p - \rho_0}{2\rho_p + \rho_0}$$

This is more suitable than the Gor'kov formulation for use with progressive wave fiends, e.g. single sided arrays, see [Sapozhnikov]. The actual implementation uses a further algebraic simplification of the above expresion.

Parameters

- **array** (TransducerArray) – The object modeling the array.
- **radius** (*float, default 1e-3*) – Radius of the spherical beads.
- **material** (Material) – The material of the sphere, default styrofoam.

class levitate.fields.**RadiationForceGradient**(*array*, *radius=0.001*, *material=Styrofoam(rho=25, poisson_ratio=0.35, c=2350)*, **args*, ***kwargs*)

Full matrix gradient of the radiation force.

Calculates the full gradient matrix of the radiation force on a small spherical bead. Component (i, j) in the matrix is $\frac{\partial F_i}{\partial q_j}$ i.e. the first index is force the force components and the second index is for derivatives. This is based on analytical differentiation of the radiation force on small beads from [Sapozhnikov], see *RadiationForce*.

Parameters

- **array** (TransducerArray) – The object modeling the array.
- **radius** (*float, default 1e-3*) – Radius of the spherical beads.
- **material** (Material) – The material of the sphere, default styrofoam.

class levitate.fields.**RadiationForceStiffness**(*array*, *radius=0.001*, *material=Styrofoam(rho=25, poisson_ratio=0.35, c=2350)*, **args*, ***kwargs*)

Radiation force gradient for small beads in arbitrary sound fields.

Calculates the non-mixed spatial derivatives of the radiation force,

$$\left(\frac{\partial F_x}{\partial x}, \frac{\partial F_y}{\partial y}, \frac{\partial F_z}{\partial z}\right)$$

where F is the radiation force by [Sapozhnikov], see *RadiationForce*.

Parameters

- **array** (TransducerArray) – The object modeling the array.

- **radius** (*float, default 1e-3*) – Radius of the spherical beads.
- **material** (Material) – The material of the sphere, default styrofoam.

class levitate.fields.**RadiationForceCurl**(**args*, ***kwargs*)

Curl or rotation of the radiation force.

Calculates the curl of the radiation force field as

$$(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z}, \frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x}, \frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y})$$

where F is the radiation force by [Sapozhnikov], see *RadiationForce*.

Parameters

- **array** (TransducerArray) – The object modeling the array.
- **radius** (*float, default 1e-3*) – Radius of the spherical beads.
- **material** (Material) – The material of the sphere, default styrofoam.

4.3.3 Gor'kov

class levitate.fields.**GorkovPotential**(*array*, *radius=0.001*, *material=Styrofoam(rho=25, poisson_ratio=0.35, c=2350)*, **args*, ***kwargs*)

Gor'kov's potential U.

Calculates the Gor'kov potential [Gorkov]

$$U = \frac{V}{4}(f_1 \kappa_0 |p|^2 - \frac{3}{2} f_2 \rho_0 |v|^2)$$

where

$$f_1 = 1 - \frac{\kappa_p}{\kappa_0}, \qquad f_2 = 2\frac{\rho_p - \rho_0}{2\rho_p + \rho_0}$$

and V is the volume of the particle. Note that this is only a suitable measure for small particles, i.e. $ka << 1$, where a is the radius of the particle.

Parameters

- **array** (TransducerArray) – The object modeling the array.
- **radius** (*float, default 1e-3*) – Radius of the spherical beads.
- **material** (Material) – The material of the sphere, default styrofoam.

class levitate.fields.**GorkovGradient**(*array*, *radius=0.001*, *material=Styrofoam(rho=25, poisson_ratio=0.35, c=2350)*, **args*, ***kwargs*)

Gradient of Gor'kov's potential, ∇U.

Calculates the Cartesian spatial gradient of Gor'kov's potential, see *GorkovPotential* and [Gorkov]. This is a vector value used to calculate the radiation force as

$$F = -\nabla U.$$

Note that this value is not suitable for sound fields with strong traveling wave components. If this is the case, use the *RadiationForce* field instead.

Parameters

- **array** (TransducerArray) – The object modeling the array.
- **radius** (*float, default 1e-3*) – Radius of the spherical beads.
- **material** (Material) – The material of the sphere, default styrofoam.

class levitate.fields.**GorkovLaplacian**(*array*, *radius=0.001*, *material=Styrofoam(rho=25, poisson_ratio=0.35, c=2350)*, **args*, ***kwargs*)

Laplacian of Gor'kov's potential, $\nabla^2 U$.

This calculates the Cartesian parts of the Laplacian of Gor'kov's potential, see *GorkovPotential* and [Gorkov]. This is not really the Laplacian, since the components are not summed. The results can be seen as the local linear spring stiffness of the radiation force.

Note that this value is not suitable for sound fields with strong traveling wave components. If this is the case, use the *RadiationForceStiffness* field instead.

Parameters

- **array** (TransducerArray) – The object modeling the array.
- **radius** (*float, default 1e-3*) – Radius of the spherical beads.
- **material** (Material) – The material of the sphere, default styrofoam.

4.3.4 Spherical Harmonics Forces

class levitate.fields.**SphericalHarmonicsForce**(*array*, *radius*, *orders=None*, *material=Styrofoam(rho=25, poisson_ratio=0.35, c=2350)*, *scattering_model='Hard sphere'*, **args*, ***kwargs*)

Spherical harmonics based radiation force.

Expands the local sound field in spherical harmonics and calculates the radiation force in the spherical harmonics domain. The expansion coefficients are calculated using superposition of the translated expansions of the transducer radiation patterns. The radiation force is calculated using a similar derivation as [Sapozhnikov], but without any plane wave decomposition.

Parameters

- **array** (TransducerArray) – The object modeling the array.
- **radius** (*float*) – Radius of the spherical beads.
- **orders** (*int*) – The number of force orders to include. Note that the sound field will be expanded at one order higher that the force order. Will default to floor(ka) + 3, where `k` is the wavenumber and `a` is the radius.
- **material** (Material) – The material of the sphere, default styrofoam.
- **scattering_model** – Chooses which scattering model to use. Currently `Hard sphere`, `Soft sphere`, and `Compressible sphere` are implemented.

class levitate.fields.**SphericalHarmonicsForceGradient**(**args*, ***kwargs*)

Spatial gradient of the total spherical radiation force.

The three Cartesian derivatives of the radiation force on a spherical object, calculated using spherical harmonics expansion of the sound field. See *SphericalHarmonicsForce* for details on the parameters.

Parameters

- **array** (TransducerArray) – The object modeling the array.
- **radius** (*float*) – Radius of the spherical beads.
- **orders** (*int*) – The number of force orders to include. Note that the sound field will be expanded at one order higher that the force order. Will default to floor(ka) + 3, where `k` is the wavenumber and `a` is the radius.
- **material** (Material) – The material of the sphere, default styrofoam.
- **scattering_model** – Chooses which scattering model to use. Currently `Hard sphere`, `Soft sphere`, and `Compressible sphere` are implemented.

class levitate.fields.**SphericalHarmonicsForceDecomposition**(*array*, *radius*, *orders=None*, *material=Styrofoam(rho=25, poisson_ratio=0.35, c=2350)*, *scattering_model='Hard sphere'*, **args*, ***kwargs*)

Radiation force decomposed in spherical harmonics.

This is mostly intended for research purposes, when the radiation force decomposed in individual spherical harmonics bases is of interest.

Parameters

- **array** (TransducerArray) – The object modeling the array.
- **radius** (*float*) – Radius of the spherical beads.
- **orders** (*int*) – The number of force orders to include. Note that the sound field will be expanded at one order higher that the force order. Will default to floor(ka) + 3, where k is the wavenumber and a is the radius.
- **material** (Material) – The material of the sphere, default styrofoam.
- **scattering_model** – Chooses which scattering model to use. Currently `Hard sphere`, `Soft sphere`, and `Compressible sphere` are implemented.

class levitate.fields.**SphericalHarmonicsForceGradientDecomposition**(**args*, ***kwargs*)

Spatial gradient of spherical harmonics force decomposition.

Takes the spatial gradient in Cartesian coordinates of each order and mode of the radiation force calculated from a spherical harmonics expansion. See *SphericalHarmonicsForce* for details on algorithms and parameters.

Parameters

- **array** (TransducerArray) – The object modeling the array.
- **radius** (*float*) – Radius of the spherical beads.
- **orders** (*int*) – The number of force orders to include. Note that the sound field will be expanded at one order higher that the force order. Will default to floor(ka) + 3, where k is the wavenumber and a is the radius.
- **material** (Material) – The material of the sphere, default styrofoam.
- **scattering_model** – Chooses which scattering model to use. Currently `Hard sphere`, `Soft sphere`, and `Compressible sphere` are implemented.

4.4 Optimization

Procedures and algorithms for numerical optimization.

The main method currently in use for acoustic levitation (in this package) is nonlinear numerical minimization of a cost function. The cost funcion should be constructed using the *fields* module.

levitate.optimization.**phase_alignment**(**states*, *method='parallel'*, *output='states'*)

Align independet states with respect to the global phase.

Parameters

- ***states** (*array_like*) – The states to align. THis can be passed in as a sequence of arguments, of as a (P, N)-shaped array, where P is the number of states and N is the number of elements in each state.
- **method** (*str, optional, keyword-only*) – Which method to use for the alignment, default 'parallel'. Should be one of 'parallel' or 'sequential', see below for description.

- **output** (*str, optional, keyword-only*) – What the function should return, default 'states'. The string should contain some combination of 'states' and/or 'phases'. The funtion will return the aligned states and/or the obtained phases in the order found in the string. If only one of 'states' or 'phases' is found, only that one will be returned.

A single state for an array is only unique up to a global phase. When multiple states are considered, the global phase of each state can be shifted to minimize the difference between the states. This function takes a number of states and finds the optimal phase shifts for each of the states. This can operate in two distinct modes, a parallel mode and a sequential mode.

Method `'parallel'` minimizes the sum of all magnitude differences of the states

$$\sum_k \sum_l ||S_k e^{i\phi_k} - S_l e^{i\phi_l}||^2$$

or equivalently, maximizes the sum of the states

$$||\sum_k S_k e^{i\phi_k}||^2.$$

Explicitly, this is done by numerically minimizing the cost function

$$O = \Re\{cAc^*\}$$

where c is a vector with the phases written on complex form, and $A[i,j] = -\sum_n S_i[n]S_j^*[n](1-\delta_{ij})$. This is suitable for superposition, where we want the states to have the most power output.

Method `sequential` minimizes the difference between consecutive states in an iterative fashion. In each step, the difference

$$||S_k e^{i\phi_k} - S_{k-1} e^{i\phi_{k-1}}||$$

is minimized. This is done explicitly as

$$\phi_k = \phi_{k-1} - \arg\{\sum_n S_k[n]S_{n-1}^*[n]\}$$

with $\phi_0 = 0$. This procedure is suitable for state transitions, where the difference between non-consecutive states is irrelevant.

`levitate.optimization.`**`minimize`**(*functions*, *array*, *start_values=None*, *use_real_imag=False*, *constrain_transducers=None*, *variable_amplitudes=False*, *callback=None*, *precall=None*, *basinhopping=False*, *minimize_kwargs=None*, *return_optim_status=False*)

Minimizes a set of cost functions.

The cost function should have the signature `f(complex_amplitudes)` where `complex_amplitudes` is an ndarray with weight of each element in the transducer array. The function should return `value, jacobians` where the jacobians are the derivatives of the value w.r.t the transducers as defined in the full documentation. Also see the documentation of the field wrappers for further details.

This function supports minimization sequences. Pass an iterable of functions to start sequenced minimization, e.g. a list of cost functions. The arguments: `use_real_imag`, `variable_amplitudes`, `constrain_transducers`, `callback`, `precall`, `basinhopping`, and `minimize_kwargs` can be given as single values or as iterables of the same length as `functions`.

Parameters

- **functions** – The cost function that should be minimized. A single callable, or an iterable of callables, as described above.
- **array** (`TransducerArray`) – The array from which the cost functions are created.
- **start_values** (*complex ndarray, optional*) – The start values for the optimization. Will default to 1 for all transducers if not given. Note that the precall for minimization sequences can overrule this value.

- **use_real_imag** (*bool, default False*) – Toggles if the optimization should run using the phase-amplitude formulation or the real-imag formulation.
- **constrain_transducers** (*array_like*) – Specifies a number of transducers which are constant elements in the minimization. Will be used as the second argument in `np.delete`
- **variable_amplitudes** (*bool*) – Toggles the usage of varying amplitudes in the minimization. If `use_real_imag` is False 'phases first' is also a valid argument for this parameter. The minimizer will then automatically sequence to optimize first with fixed then with variable amplitudes, returning only the last result.
- **callback** (*callable*) – A callback function which will be called after each step in sequenced minimization. Return false from the callback to break the sequence. Should have the signature : `callback(array=array, result=result, optim_status=opt_res, idx=idx)`
- **precall** (*callable*) – Initialization function which will be called with the array phases, amplitudes, and the sequence index before each sequence step. Must return the initial phases and amplitudes for the sequence step. Default sets the phases and amplitudes to the solution of the previous sequence step, or the original state for the first iteration. Should have the signature : `precall(complex_amplitudes, idx)`
- **basinhopping** (*bool or int*) – Specifies if basinhopping should be used. Pass an int to specify the number of basinhopping iterations, or True to use default value.
- **return_optim_status** (*bool*) – Toggles the `optim_status` output.
- **minimize_kwargs** (*dict*) – Extra keyword arguments which will be passed to `scipy.minimize`.

Returns

- **result** (`ndarray`) – The array phases and amplitudes after minimization. Stacks sequenced results in the first dimension.
- **optim_status** (`OptimizeResult`) – Scipy optimization result structure. Optional output, toggle with the corresponding input argument.

4.5 Utilities

This page documents four modules, *analysis*, *visualizers*, *materials*, and *hardware*.

Some tools for analysis of sound fields and levitation traps.

`levitate.analysis.`**dB**(*x*, *power=False*)

Convert ratio to decibels.

Converting a ratio to decibels depends on whether the ratio is a ratio of amplitudes or a ratio of powers. For amplitudes the decibel value is $20\log(|x|)$, while for power ratios the value is $10\log(|x|)$ where $\log$ is the base 10 logarithm.

Parameters

- **x** (*numeric*) – Linear amplitude or radio, can be complex.
- **power** (*bool, default False*) – Toggles if the ration is proportional to power.

Returns **L** (*numeric*) – The decibel value.

`levitate.analysis.`**SPL**(*p*)

Convert sound pressure to sound pressure level.

Uses the standard reference value for airborne acoustics: 20 µPa. Note that the input is the pressure amplitude, not the RMS value.

Parameters **p** (*numeric, complex*) – The complex sound pressure amplitude.

Returns **SPL** (*numeric*) – The sound pressure level

levitate.analysis.**SVL**(*u*)

Convert sound particle velocity to sound velocity level.

Uses the standard reference value for airborne acoustics: 50 nm/s, which is approximately 20 µPa / c_0 / rho_0 Note that the input the velocity amplitude(s), not the RMS values.

If the first axis of the velocity input has length 3, it will be assumed to be the three Cartesian components of the velocity.

Parameters **u** (*numeric, complex*) – The complex sound velocity amplitude, or the vector velocity.

Returns **SVL** (*numeric*) – The sound velocity level

levitate.analysis.**find_trap**(*array*, *state*, *position*, *tolerance=1e-05*, *time_interval=50*, *path_points=1*, ***kwargs*)

Find the approximate location of a levitation trap.

Find an approximate position of a acoustic levitation trap close to a starting point. This is done by following the radiation force in the sound field using an differential equation solver. The differential equation is the un-physical equation $d\vec{x}/dt = \vec{F}(x,t)$, i.e. interpreting the force field as a velocity field. This works for finding the location of a trap and the field line from the starting position to the trap position, but it can not be seen as a proper kinematic simulation of the system.

The solving of the above equation takes place until the whole time interval is covered, or the tolerance is met. The tolerance is evaluated using the assumption that the force is zero at the trap, evaluating the distance from the zero-force position using the force gradient.

Parameters

- **array** (TransducerArray) – The transducer array to use for the solving.
- **state** (*complex array like*) – The complex transducer amplitudes to use for the solving.
- **position** (*array_like, 3 elements*) – The starting point for the solving.
- **tolerance** (*numeric, default 10e-6*) – The approximate tolerance of the solution, i.e. how close should the found position be to the true position, in meters.
- **time_interval** (*numeric, default 50*) – The un-physical time of the solution range in the differential equation above.
- **path_points** (*int, default 1*) – Sets the number of points to return the path at. A single evaluation point will only return the found position of the trap.

Returns **trap_pos** (*numpy.ndarray*) – The found trap position, or the path from the starting position to the trap position.

class levitate.analysis.**KineticSimulation**(*array*, *t_end=1*, *radius=0.001*, *material=Styrofoam(rho=25, poisson_ratio=0.35, c=2350)*, *force=None*, *force_gradient=None*, ***solver_kwargs*)

Performs kinetic simulations for levitated spherical objects.

Initialize with the relevant parameters. Call the object with a state and a position to start the simulation. After a simulation, the object stores attributes for the results.

If the simulation is not started at the center of something resembling a trap, the energy tracking will not work properly.

Variables

- **~KineticSimulation.t** (*ndarray, shape (T,)*) – Time vector for the simulated positions.

- **~KineticSimulation.position** (*ndarray, shape (3, T)*) – Simulated positions.
- **~KineticSimulation.velocity** (*ndarray, shape (3, T)*) – Simulated velocities.
- **~KineticSimulation.kinetic_energy** (*ndarray, shape (T,)*) – Simulated kinetic energy.
- **~KineticSimulation.potential_energy** (*ndarray, shape (T,)*) – Approximate potential energy. Calculated from a linear approximation at the starting position.
- **~KineticSimulation.total_energy** (*ndarray, shape (T,)*) – Sum of kinetic and potential energy.

__call__(*state*, *initial_position*)
Call self as a function.

levitate.analysis.**linear_stability_metric**(*array*, *radius*, *force_divergence=None*, *force_curl=None*, *material=Styrofoam(rho=25, poisson_ratio=0.35, c=2350)*)

A crude stability metric for traps.

This is based on considering the dynamics of a linear approximation of a trap which has equal axial stiffness. The metric is larger than one for traps which are empirically stable, and smaller than one for traps which empirically are unstable. This is not a guaranteed metric, but relies on very heavy assumptions of the trap behavior. See "Sound Field Design for Transducer Array-Based Acoustic Levitation", Andersson, 2022, PhD thesis, for further information.

The output is a field object, which can be called with a state and a position like the other field objects in this toolbox. The implemented relation is

$$-\frac{27\pi\mu^2}{a\rho_*}\frac{\nabla\cdot\vec{F}}{|\nabla\times\vec{F}|^2}$$

where F is the force field in the trap, μ is the dynamic viscosity of the medium, ρ_* is the density of the object, and a its radius.

4.5.1 Visualization

Visualization classes based on the plotly graphing library.

class levitate.visualizers.**Visualizer**(*array*, **traces*, *display_scale='mm'*)

insert(*index*, *value*)
S.insert(index, value) – insert value before index

class levitate.visualizers.**ArrayVisualizer**(*array*, **args*, ***kwargs*)
Visualizations of a trandcuer array.

It is possible to set an item using either just a trace specifier, e.g. "Pressure", which create the appropriate trace with default arguments. If arguments are required or wanted, set the item to a tuple where the first element is the trace specifier, and subsequent elements are the arguments. If the last element in the tuple is a dictionary, it will be used as keyword arguments for the trace type.

__call__(**complex_transducer_amplitudes*, ***kwargs*)
Call self as a function.

class levitate.visualizers.**ForceDiagram**(**args*, *scale_to_gravity=True*, *include_gravity=True*, ***kwargs*)

__call__(**complex_transducer_amplitudes*, ***kwargs*)
Call self as a function.

4.5.2 Materials

Manages material properties.

Many functions need access to some material properties. In order to ensure that the properties of a specific material is the same everywhere, they are all collected in classes here.

Pickling

Pickling materials require some special consideration. In general there should be a single set of properties defining a material, but loading some data which is saved with modified properties will create a conflict. The newly loaded material will be in a "Local" state, using modified properties different from the global ones. It is recommended to resolve this by avoiding the problem entirely by modifying the global properties before loading the old data. If this is not possible or preferable, there are three functions intended to resolve the conflict using either the global or the local properties.

Note: Updating global material properties will change the properties throughout the entire package, but some classes (notably the fields) pre-calculate a lot of material-dependent properties. These properties will **NOT** be updated after a material update. It is therefore highly recommended to define the material properties once in the beginning of a session.

---

class levitate.materials.**MaterialMeta**(*name*, *bases*, *dct*)

Metaclass for materials.

This metaclass will automatically create the properties defines in the `properties` variable in the class or its bases. The properties implement a local/global system, where each instance will default to use the global properties unless otherwise specified.

static class_instance_property(*name*, *doc=None*)

Create a local/global property.

class levitate.materials.**Material**(***kwargs*)

Main base class for materials.

This class handles most of the functionality of the materials in the package. Each material is required to have (at least) a speed of sound and a density, from which the impedance and the compressibility can be calculated. In most cases there should only be a single instance of each material class, defining the properties of said material. Multiple instances might be created while pickling see the section below. If a new material of an existing material class is created with modified properties, it will also be created in a "Local" state.

property compressibility

Compressibility $\frac{1}{\rho c^2}$, non settable.

property impedance

(Specific) Acoustic (wave) impedance ρc, non settable.

load_from_global()

Load properties from the global state.

Useful only on materials in a local state to resolve conflicts. Replaces the current local properties with the global properties and goes to global mode, completely removing the stored values.

push_to_global()

Push the local properties to the global state.

Useful only on materials in a local state to resolve conflicts. Replaces the current global properties with the modified local ones, completely overriding the global properties for all global instances.

classmethod force_all_to_global()

Force all instances of this material to use global properties.

Useful to resolve material conflicts by choosing the global state for all instances of the material. Will never change the global properties, even if called from a locally modified instance.

property c
The (longitudinal) speed of sound in the material, in m/s.

property rho
The density of the material, in kg/m^3.

class levitate.materials.**Gas**(***kwargs*)
Base class for ideal gases.

All ideal gases can determine the wave speed and density from the ambient temperature and pressure using more basic material constants, see *update_properties*.

update_properties(*temperature=None*, *pressure=None*)
Update the material properties with the ambient conditions.

Sets the material properties of air according to

$$c = \sqrt{\gamma R T}$$
$$\rho = \frac{P}{RT}$$

where T is the ambient temperature in Kelvin, P is the ambient pressure, γ is the adiabatic index, and R is the specific gas constant for the gas.

Parameters

- **temperature** (*float*) – The ambient temperature, in degrees Celsius. Defaults to 20.
- **pressure** (*float*) – The static ambient air pressure, in Pa. Defaults to 101325.

property kinematic_viscosity
Kinematic viscosity, in m^2/s.

property c
The (longitudinal) speed of sound in the material, in m/s.

property dynamic_viscosity
Dynamic viscosity, in Pa * s.

property rho
The density of the material, in kg/m^3.

class levitate.materials.**Solid**(***kwargs*)
Base class for elastic solids.

Solids can support shear waves, which is important for some scattering problems.

property c_transversal
Transversal wave speed.

The speed of sound for transversal waves, i.e. shear waves. Calculated as $c\sqrt{\frac{1-2\nu}{2-2\nu}}$, where ν is the Poisson's ratio.

property c
The (longitudinal) speed of sound in the material, in m/s.

property poisson_ratio
Poisson's ratio, related to shear wave speed.

property rho
The density of the material, in kg/m^3.

class levitate.materials.**Air**(***kwargs*)
Properties of air.

Has default values:

```
c = 343.2367605312694
rho = 1.2040847588826422
```

property c
The (longitudinal) speed of sound in the material, in m/s.

property dynamic_viscosity
Dynamic viscosity, in Pa * s.

property rho
The density of the material, in kg/m^3.

class levitate.materials.**Styrofoam**(***kwargs*)
Properties of styrofoam.

Has default values:

```
c = 2350
rho = 25
poisson_ratio = 0.35
```

property c
The (longitudinal) speed of sound in the material, in m/s.

property poisson_ratio
Poisson's ratio, related to shear wave speed.

property rho
The density of the material, in kg/m^3.

4.5.3 Hardware

Hardware related classes and functions.

Various classes and functions to simulate and interface with physical hardware.

Disclaimer

This module is not related to Ultraleap as a company, and it not part of their SDK. It is simply a non-programmer's way around testing everything in C++. Similarly for the other implemented arrays.

Use at your own risk!

DragonflyArray	Rectangular array with Ultraleap Dragonfly U5 layout.
AcoustophoreticBoard	

data_to_cpp	Write data to a file suitable for c++.
data_from_cpp	Read data previously written for c++.

levitate.hardware.**data_to_cpp**(*complex_values*, *filename*)
Write data to a file suitable for c++.

Takes numpy data and writes it to a file which is simple to read from c++. Internally uses `numpy.tofile` for the actual write.

Parameters

- **complex_values** (*numpy.ndarray*) – The complex transducer values for the array. The order of the transducers must match the internal order of the array.
- **filename** (*string*) – The filename of the file to create.

Note:

- The data will be normalized to have a maximum amplitude of 1.
- The data will be written as 64 bit complex floats, i.e. 32 bit real + 32 bit imaginary.
- The data will be conjugated: Ultrahaptics uses a different phase convention.

levitate.hardware.**data_from_cpp**(*file*, *num_transducers*)
Read data previously written for c++.

This is the inverse of *data_to_cpp*, and is used to read data previously written with said function, or with similar conventions.

Parameters

- **file** (*String or open file*) – The file to read. See numpy.fromfile.
- **num_transducers** (*int*) – The number of transducers in the array. This is important to be able to reshape the data and have the correct number of states in the output.

Returns **data** (*numpy.ndarray*) – The data read from the file. Shape `(M, N)` where `M` is the number of states in the file, and `N` is the number of transducers specified.

Note:

- The data is assumed to be 64 bit complex floats, i.e. 32 bit real + 32 bit imaginary.
- The data will be conjugated: Ultrahaptics uses a different phase convention.

class levitate.hardware.**AcoustophoreticBoard**(*id=None*, *linearize_amplitude=True*, *compensate_phase=True*, *normalize=False*, *use_phase_calibration=True*, *use_amplitude_calibration=False*, ***kwargs*)

class levitate.hardware.**DragonflyArray**(***kwargs*)
Rectangular array with Ultraleap Dragonfly U5 layout.

This is a 16x16 element array where the order of the transducer elements are the same as the iteration order in the Ultraleap SDK. Otherwise behaves exactly like a `RectangularArray`.

4.6 Field Wrappers

4.6.1 Class list

Public API

These are the only classes and functions regarded as part of the public API, but they will only be used directly when implementing new algorithm types.

class levitate.fields._wrappers.**FieldImplementation**(*array*)
Base class for FieldImplementations.

The attributes listed below are part of the API and should be implemented in subclasses.

Parameters **array** (TransducerArray) – The array object to use for calculations.

Variables

- **~FieldImplementation.values_require** (*dict*) – Each key in this dictionary specifies a requirement for the *values* method. The wrapper classes will manage calling the method with the specified arguments.

- **~FieldImplementation.jacobians_require** (*dict*) – Each key in this dictionary specifies a requirement for the *jacobians* method. The wrapper classes will manage calling the method with the specified arguments.

values()
Method to calculate the value(s) for the field.

jacobians()
Method to calculate the jacobians for the field. This method is optional if the implementation is not used as a cost function in optimizations.

Parameters **array** (TransducerArray) – The object modeling the array.

class requirement(**args*, ***kwargs*)
Parse a set of requirements.

FieldImplementation objects should define requirements for values and jacobians. This class parses the requirements and checks that the request can be met upon call. The requirements are stored as a non-mutable custom dictionary. Requirements can be added to each other to find the combined requirements.

Keyword Arguments

- **complex_transducer_amplitudes** – The field requires the actual complex transducer amplitudes directly. This is a fallback requirement when it is not possible to implement the field with the other requirements, and no performance optimization is possible.
- **pressure_derivs_summed** – The number of orders of Cartesian spatial derivatives of the total sound pressure field. Currently implemented to third order derivatives. See `levitate._indexing.pressure_derivs_order` and `levitate._indexing.num_pressure_derivs` for a description of the structure.
- **pressure_derivs_summed** – Like pressure_derivs_summed, but for individual transducers.
- **spherical_harmonics_summed** – A spherical harmonics decomposition of the total sound pressure field, up to and including the order specified. where remaining dimensions are determined by the positions.
- **spherical_harmonics_individual** – Like spherical_harmonics_summed, but for individual transducers.

Raises NotImplementedError – If one or more of the requested keys is not implemented.

Basic Types

class levitate.fields._wrappers.**Field**(*field*, ***kwargs*)
Primary class for single point, single field.

This is a wrapper class for *FieldImplementation* to simplify the manipulation and evaluation of the implemented fields. Normally it is not necessary to manually create the wrapper, since it should be done automagically. Many properties are inherited from the underlying field implementation, e.g. `ndim`, `array`, `values`, `jacobians`.

Parameters **field** (*FieldImplementation*) – The implemented field to use for calculations.

__call__(*complex_transducer_amplitudes*, *position*)
Evaluate the field implementation.

Parameters

- **compelx_transducer_amplitudes** (*complex numpy.ndarray*) – Complex representation of the transducer phases and amplitudes of the array used to create the field.

- **position** (*array-like*) – The position(s) where to evaluate the field. The first dimension needs to have 3 elements.

Returns **values** (*ndarray*) – The values of the implemented field used to create the wrapper.

class levitate.fields._wrappers.**FieldPoint**(*field*, *position*, ***kwargs*)

Position-bound class for single point, single field.

See *Field* for more precise description.

Parameters

- **field** (FieldImplementation) – The implemented field to use for calculations.
- **position** (*numpy.ndarray*) – The position to bind to.

__call__(*complex_transducer_amplitudes*)

Evaluate the field implementation.

Parameters **compelx_transducer_amplitudes** (*complex numpy.ndarray*) – Complex representation of the transducer phases and amplitudes of the array used to create the field.

Returns **values** (*ndarray*) – The values of the implemented field used to create the wrapper.

Magnitude Squared Types

MultiFields

class levitate.fields._wrappers.**MultiField**(**fields*, ***kwargs*)

Class for multiple fields, single position calculations.

This class collects multiple *Field* objects for simultaneous evaluation at the same position(s). Since the fields can use the same spatial structures this is more efficient than to evaluate all the fields one by one.

Parameters ***fields** (Field) – Any number of *Field* objects.

__call__(*complex_transducer_amplitudes*, *position*)

Evaluate all fields.

Parameters

- **complex_transducer_amplitudes** (*complex numpy.ndarray*) – Complex representation of the transducer phases and amplitudes of the array used to create the field.
- **position** (*array-like*) – The position(s) where to evaluate the fields. The first dimension needs to have 3 elements.

Returns **values** (*list*) – A list of the return values from the individual fields. Depending on the number of dimensions of the fields, the arrays in the list might not have compatible shapes.

class levitate.fields._wrappers.**MultiFieldPoint**(**fields*, ***kwargs*)

Class for multiple field, single fixed position calculations.

This class collects multiple *FieldPoint* bound to the same position(s) for simultaneous evaluation. Since the fields can use the same spatial structures this is more efficient than to evaluate all the fields one by one.

Parameters ***fields** (FieldPoint) – Any number of *FieldPoint* objects.

Warning: If the class is initialized with fields bound to different points, some of the fields are simply discarded.

__call__(*complex_transducer_amplitudes*)
Evaluate all fields.

Parameters **compelx_transducer_amplitudes** (*complex numpy.ndarray*) – Complex representation of the transducer phases and amplitudes of the array used to create the field.

Returns **values** (*list*) – A list of the return values from the individual fields. Depending on the number of dimensions of the fields, the arrays in the list might not have compatible shapes.

MultiPoints

Private Classes

These classes are not considered part of the public API, and should not appear other than as superclasses.

class levitate.fields._wrappers.**FieldBase**(*, *transforms=None*)
Base class for all field type objects.

This wraps a few common procedures for fields, primarily dealing with preparation and evaluation of requirements for fields implementations. The fields support some numeric manipulations to simplify the creation of variants of the basic types. Not all types of fields support all operations, and the order of operation can matter in some cases. If unsure if the arithmetics return the desired outcome, print the resulting object to inspect the new structure.

Note: This class should not be instantiated directly.

---

class levitate.fields._wrappers.**FieldImplementationMeta**(*clsname*, *bases*, *attrs*)
Metaclass to wrap *FieldImplementation* objects in *Field* objects.

API-wise it is nice to call the implementation classes when requesting a field. Since the behavior of the objects should change depending on if they are added etc, it would be very difficult to keep track of both the current state and the actual field in the same top level object. This class will upon object creation instantiate the called class, but also instantiate and return a *Field*-type object.

__call__(**cls_args*, ***cls_kwargs*)
Instantiate an *Field*-type object, using the `cls` as the base field implementation.

The actual *Field*-type will be chosen based on which optional parameters are passed. If no parameters are passed (default) a *Field* object is returned. If `weight` is passed a `CostField` object is returned. If `position` is passed a *FieldPoint* object is returned. If both `weight` and `position` is passed a `CostFieldPoint` object is returned.

Parameters

- **cls** (*class*) – The *FieldImplementation* class to use for calculations.
- ***cls_args** – Args passed to the `cls`.
- ****cls_kwargs** – Keyword arguments passed to `cls`.

5 Changelog

Changes are documented here.

5.1 Unreleased

5.2 2.4.2 - 2020-03-09

5.2.1 Removed

- Python 3.5 is no longer supported

5.2.2 Added

- Improved hover info on transducer visualizations
- Support for controlling hardware from Interact Lab
- Function to phase align states

5.2.3 Changed

- Force diagrams will now subtract the gravitational force
- Force cone visualization will not include the point in the center

5.3 2.4.1 - 2020-02-17

5.3.1 Added

- Spherical cap array class

5.3.2 Changed

- Visualizers will not allocate memory until used the first time

5.4 Pre 2.4

No changelog was kept beyond this point.

References

[Gorkov] L. P. Gorkov, “On the Forces Acting on a Small Particle in an Acoustical Field in an Ideal Fluid” Soviet Physics Doklady, vol. 6, p. 773, Mar. 1962.

[Sapozhnikov] O. A. Sapozhnikov and M. R. Bailey, “Radiation force of an arbitrary acoustic beam on an elastic sphere in a fluid” J Acoust Soc Am, vol. 133, no. 2, pp. 661–676, Feb. 2013.