SMUTHI Documentation

Release 2.1.2

Amos Egel

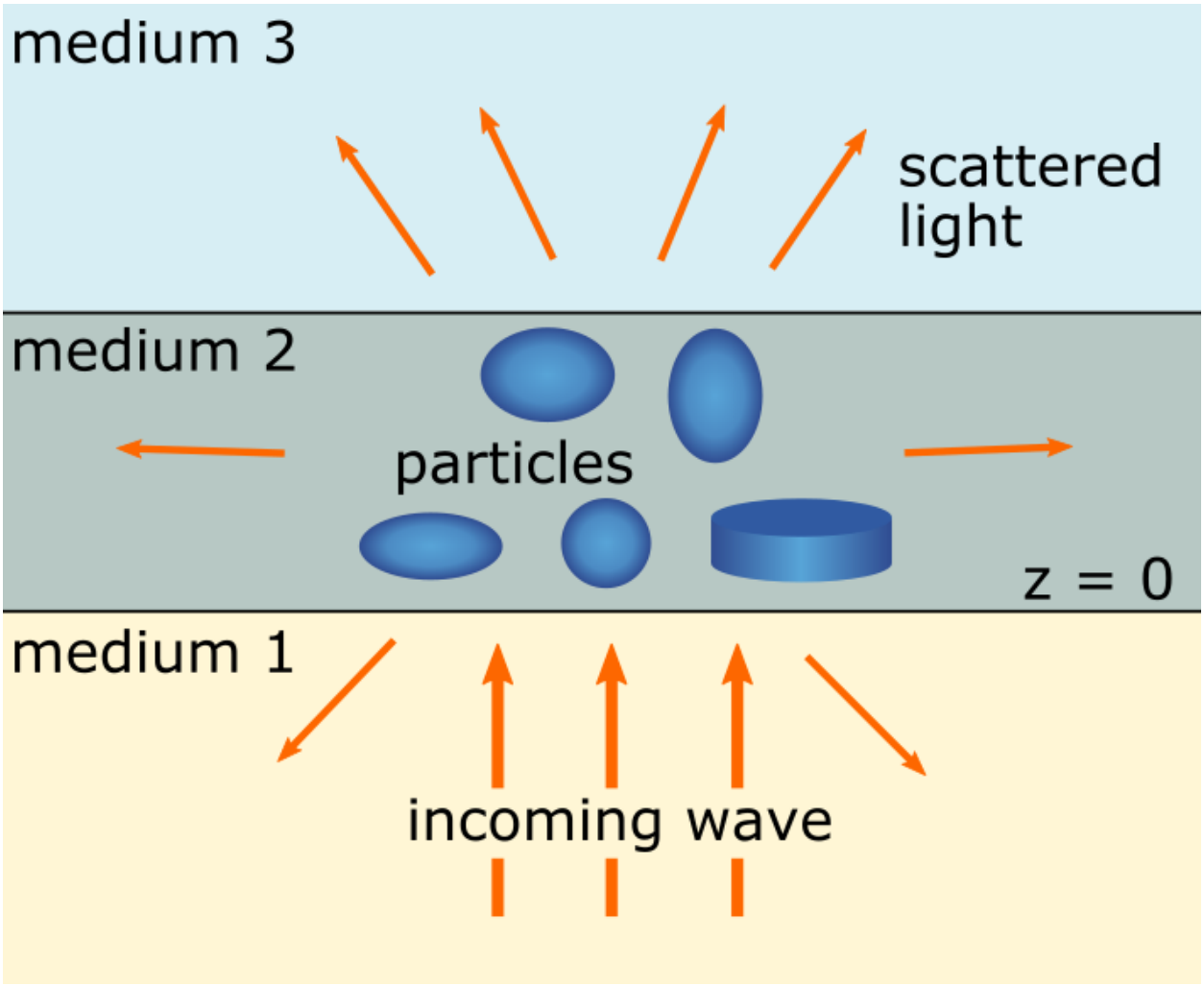
May 16, 2023

Contents

CHAPTER 1

About Smuthi

Smuthi stands for 'scattering by multiple particles in thin-film systems'. It is a Python software that allows to solve light scattering problems involving one ore multiple particles near or inside a system of planar layer interfaces.



It solves the Maxwell equations (3D wave optics) in frequency domain (one wavelength per simulation).

1.1 Simulation method

Smuthi is based on the T-matrix method for the single particle scattering and on the scattering-matrix method for the propagation through the layered medium. See *[Egel 2018]* and other publications listed in the *literature section* for a description of the method.

For non spherical particles, Smuthi calls the NFM-DS by Doicu, Wriedt and Eremin to compute the single particle T-matrix. This is a Fortran software package written by based on the "Null-field method with discrete sources", see *[Doicu et al. 2006]*.

Performance critical parts of the software are implemented in CUDA. When dealing with a large number of particles, Smuthi can benefit from a substantial acceleration if a suitable (NVIDIA) GPU is available.

For CPU-only execution, other acceleration concepts (including MPI parallelization, Numba JIT compilation) are currently tested.

1.2 Range of applications

Smuthi can be applied to any scattering problem in frequency domain involving

- a system of plane parallel layer interfaces separating an arbitrary number of thin metallic or dielectric layers.
- an arbitrary number of wavelength-scale scattering particles (currently available: spheres, spheroids, finite cylinders, custom particle shapes, anisotropic spheres, layered spheroids). The particles can be metallic or dielectric and rotated to an arbitrary orientation.
- an initial field in form of a plane wave, a beam (currently available: beam with Gaussian xy-profile) or a collection of dipole sources

Thus, the range of applications spans from scattering by a single particle on a substrate to scattering by several thousand particles inside a planarly layered medium. For a number of examplary simulations, see the *examples* section.

1.3 Simulation output

Smuthi can compute

- the 3D electric and/or magnetic field, for example along a cut plane and save it in the form of ascii data files, png images or gif animations.
- the far field power flux of the total field, the initial field or the scattered field. For plane wave excitation, it can be processed to the form of *differential scattering and extinction cross sections*.
- For dipole sources, the dissipated power can be computed (Purcell effect).

1.4 Current limitations

The following issues need to be considered when applying Smuthi:

- Particles must not intersect with each other or with layer interfaces.
- Magnetic or anisotropic materials are currently not supported (anisotropic spheres are currently tested).
- The method is in principle valid for a wide range of particle sizes - however, the numerical validity has only been tested for particle diameters up to around one wavelength. For larger particles, note that the number of multipole terms in the spherical wave expansion grows with the particle size. For further details, see the *hints for the selection of the multipole truncation order*.
- Particles in a single homogeneous medium (or in free space) can be treated by setting a trivial two layer system with the same refractive index. However, Smuthi was not designed for that use case and we believe that there is better software for that case.
- Smuthi was designed for particles on a substrate or particles near or inside a thin-film system with layer thicknesses of up to a few wavelengths. Simulations involving thick layers might fail or return wrong results due to numerical instability. Maybe a more stable algorithm for the layer system response does exist - help is welcome.
- Smuthi does not provide error checking of user input, nor does it check if numerical parameters specified by the user are sufficient for accurate simulation results. It is thus required that the user develops some understanding of the influence of various numerical parameters on the validity of the results. See the *simulation guidelines*.
- A consequence of using the T-matrix method is that the electric field inside the circumscribing sphere of a particle cannot be correctly computed, see for example Auguié et al. (2016). In the electric field plots, the circumscribing sphere is displayed as a dashed circle around the particle as a reminder that there, the computed near fields cannot be trusted.

- Particles with initersecting circumscribing spheres can lead to incorrect results. The use of Smuthi is therefore limited to geometries with particles that have disjoint circumscribing spheres.
- If particles are located near interfaces, such that the circumscribing shere of the particle intersects the interface, a correct simulation result can in principle be achieved. However, special care has to be taken regarding the selection of the truncation of the spherical and plane wave expansion, see the *hints for the selection of the wavenumber truncation*.
- Dipole sources must not be placed inside the circumscribing sphere of a non-spherical particle (exception: it is OK if the particle is in a different layer)

1.5 License

The software is licensed under the MIT license.

1.6 How to cite this software

If you use SMUTHI for a publication, please consider to cite *[Egel et al. 2021]*.

1.7 Contact

Email to the author under amos.egel@gmail.com or to the Smuthi mailing list under smuthi@googlegroups.com for questions, feature requests or if you would like to contribute.

1.8 Acknowledgments

The following persons are/were involved in the Smuthi development: Amos Egel, Dominik Theobald, Krzysztof Czajkowski, Konstantin Ladutenko, Lorenzo Pattelli, Alexey Kuznetsov, Parker Wray.

The authors wish to thank Adrian Doicu, Thomas Wriedt and Yuri Eremin for the NFM-DS package, a copy of which is distributed with Smuthi.

Ilia Rasskazov, Giacomo Mazzamuto, Fabio Mangini, Refet Ali Yalcin and Johanne Heitmann Solheim have helped with useful comments, bug reports and code additions.

We thank Håkan T Johansson for making his pywigjxpf software availible through PyPi and also under Windows.

The creation of Smuthi was supervised by Uli Lemmer and Guillaume Gomard during the research project LAMBDA, funded by the DFG in the priority programme tailored disorder.

CHAPTER 2

Getting started

2.1 Installation

We **recommend to use Linux operating systems** to run Smuthi. Otherwise, Smuthi can run on Windows, too, but issues regarding dependencies or performance are more likely.

2.1.1 Installing Smuthi under Ubuntu (recommended)

Prerequisites

python3 with *pip*, *gfortran* and *gcc* usually are shipped with the operating system. However, Smuthi requires a Python version of 3.6 or newer. Check the installed Python version by:

```
python3 --version
```

If the version is 3.5 or less, please install a newer Python version. You can have multiple Python versions installed in parallel. Depending on your configuration, you might need to replace the command `python3` in the below by the command that belongs to the newly installed Python, e.g. `python3.8`.

Make sure that the Foreign Function Interface library is available (needed for pywigxjpf):

```
sudo apt-get install libffi6 libffi-dev
```

Installation

To install Smuthi from PyPi, simply type:

```
sudo python3 -m pip install smuthi
```

Alternatively, you can install it locally from source (see below section *Installing Smuthi from source*).

2.1.2 Installing Smuthi under Windows

Prerequisites

First make sure that a 64 Bit Python 3.6 or newer is installed on your computer. You can install for example Anaconda or WinPython to get a full Python environment.

Warning: Anaconda users are required to update numpy to the latest version from conda-forge before intalling Smuthi. It is also recommended to create a dedicated conda environment for the Smuthi installation. In case the environment gets messed up by destructive interference between Pip and conda, the main Anaconda installation is then still unaffected.

Installation

Open a command window and type:

```
python -m pip install smuthi
```

Depending on where pip will install the package, you might need administrator rights for that.

Alternatively, install locally from source (see below section *Installing Smuthi from source*).

2.1.3 Installing Smuthi from source

This option allows to install a non-release version of Smuthi or to modify the source code and then run your custom version of Smuthi.

Ubuntu

Clone Smuthi and install it locally by:

```
git clone https://gitlab.com/AmosEgel/smuthi.git
cd smuthi/
sudo python3 -m pip install -e .
```

Windows

Local installation requires a Fortran compiler. Visit the MinGW getting started page and follow the instructions to install *gfortran*. Make sure to add the bin folder of your MinGW installation to the Windows PATH variable. See *Environment Settings* section of the MinGW getting started page for instructions.

Note: The MinGW version needs to fit to your Python installation. If you have 64 Bit Python, make sure to download a Mingw-64

Then, download or git clone the Smuthi project folder from the gitlab repository. Open a command prompt and change directory to the Smuthi project folder and enter:

```
python -m pip install -e .
```

If that command fails (e.g. because pip tries to compile the extension modules with the MSVC compiler instead of mingw), you can try:

```
python -m pip install wheel
python -m pip install numpy
python setup.py develop
```

Depending on the Python version, the above commands may fail to create statically linked extensions. This will lead to runtime errors saying that some DLL cannot be found. In that case you can try to overwrite the extension modules statically linked PYD-files by running the command:

```
python setup.py build_ext --inplace --compiler=mingw32 --fcompiler=gnu95 -f
```

Installing Smuthi from source on Windows can be troublesome. If you experience difficulties, please seek support from the Smuthi mailing list or open an issue on the Smuthi GitLab repository.

Verification

After installation from source you can check the unit tests:

Ubuntu:

```
sudo python3 -m pip install nose2
nose2
```

Windows:

```
python -m pip install nose2
nose2
```

2.1.4 GPU-acceleration (optional)

Note: PyCuda support is recommended if you run heavy simulations with many particles. In addition, it can speed up certain post processing steps like the evaluation of the electric field on a grid of points, e.g. when you create images of the field distribution. For simple simiulations involving one particle on a substrate, you might well go without.

If you want to benefit from fast simulations on the GPU, you need:

- A CUDA-capable NVIDIA GPU
- The NVIDIA CUDA toolkit installed
- PyCuda installed

Under Ubuntu, install PyCuda simply by:

```
sudo python3 -m pip install pycuda
```

Under Windows, installing PyCuda this is not as straightforward as under Linux. There exist prebuilt binaries on Christoph Gohlke's homepage. See for example these instructions for the necessary steps to get it running.

2.1.5 Troubleshooting

Windows: Unable to import the nfmds module

Try to install Smuthi from source.

2.2 Running a simulation

2.2.1 Create a simulation script

To start a Smuthi simulation, you need to write a simulation script and save it with the file ending *.py*, for example *my_simulation.py*.

In the *examples* section you can find a number of example scripts that illustrate the use of Smuthi. Edit and run these scripts to get a quick start.

The *Simulation guidelines* provide the necessary understanding how a simulation script is built.

For furhter details, the *API section* contains a description of all of Smuthi's modules, classes and functions.

2.2.2 Running the simulation script

To execute the simulation, run the script by

Ubuntu:

```
python3 my_simulation.py
```

Windows:

```
python my_simulation.py
```

CHAPTER 3

Simulation guidelines

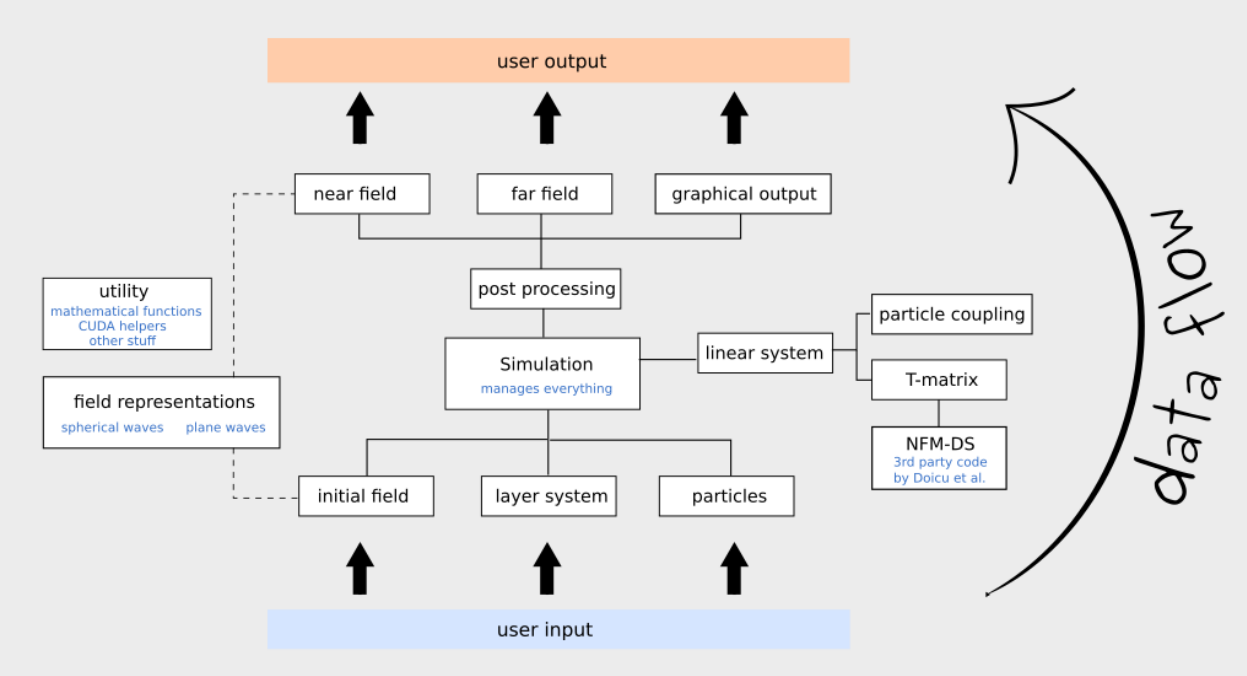
In this section, you can find general hints how to properly run a simulation with Smuthi.

3.1 Building blocks of a Smuthi simulation

In general, a Smuthi simulation script contains the following building blocks:

- Definition of the optical system: the initial field, the layer system and a list of scattering particles are defined
- Definition of the simulation object: the simulation object is initialized with the ingredients of the optical system. Further numerical settings can be applied.
- Simulation start: The calculation is launched with the command *simulation.run()*
- Post processing: The results are processed into the desired output (in our example: scattering cross section).

The following chart illustrates the interaction between the various Smuthi modules:



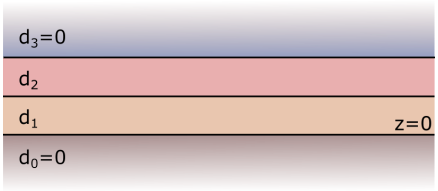
3.1.1 Initial field

Currently, the following classes can be used to define the initial field:

- **Plane waves** are specified by the vacuum wavelength, incident direction, polarization, complex amplitude and reference point. For details, see the API documentation: *smuthi.initial_field.PlaneWave*.
- **Gaussian beams** are specified by the vacuum wavelength, incident direction, polarization, complex amplitude, beam waist and reference point. Note that for oblique incident directions, the Gaussian beam is in fact an elliptical beam, such that the electric field *in the xy-plane, i.e., parallel to the layer interfaces* has a circular Gaussian footprint. For details, see the API documentation: *smuthi.initial_field.GaussianBeam*.
- **A single point dipole source** is specified by the vacuum wavelength, dipole moment vector and position. For details, see the API documentation: *smuthi.initial_field.DipoleSource*.
- **Multiple point dipole sources** can be defined using the *smuthi.initial_field.DipoleCollection* class. A dipole collection is specified by the vacuum wavelength and a list of dipole sources, which can be filled with the *smuthi.initial_field.DipoleCollection.append()* method.

3.1.2 Layer system

The layer system is specified by a list of layer thicknesses and a list of complex refractive indices. Here is the link to the corresponding class in the API documentation: *smuthi.layers.LayerSystem*.

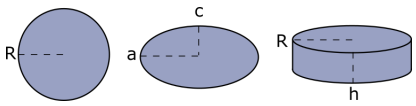


Please note that

- the layer system is built from bottom to top, i.e., the first elements in the lists refer to the bottom layer.
- bottom and top layer are semi-infinite in size. You can specify a layer thickness of zero.
- the interface between the bottom layer and the next layer in the layer system defines the $z = 0$ plane.
- the minimal layer system consists of *two* layers (e.g., a substrate and an ambient medium). Homogeneous media without layer interfaces cannot be defined, but they can be mimicked by a trivial system of two identical layers. However, we don't recommend to use Smuthi for such systems, because there are better software products to simulate systems in homogeneous media.

3.1.3 Particles

When defining a scattering particle, you need to provide the parameters regarding *geometry and material*, as well as the parameters l_{max} and m_{max} which define the *multipole expansion cutoff* (see section *Multipole cut-off*).



The following classes can currently be used:

- **Spheres** are specified by their center position vector, complex refractive index, radius and multipole cutoff. For details, see the API documentation: `smuthi.particles.Sphere`.
- **Spheroids** are specified by their center position vector, euler angles, complex refractive index, two half axis parameters, and multipole cutoff. See: `smuthi.particles.Spheroid`.
- **Cylinders** are specified by their center position vector, euler angles, complex refractive index, radius, height and multipole cutoff. See: `smuthi.particles.FiniteCylinder`.
- **Custom particles** allow to model particles with arbitrary geometry. They are specified by their position vector, euler angles, a FEM file containing the particle surface mesh, a scale parameter to set the physical size of the particle (if it deviates from the size specified by the mesh file) and multipole cutoff. See: `smuthi.particles.CustomParticle`.

Some notes:

- The simulation of nonspherical particles depends on the NFM-DS Fortran code by Adrian Doicu, Thomas Wriedt and Yuri Eremin, see *[Doicu et al. 2006]*.
- Particles must not overlap with each other or with layer interfaces.
- The circumscribing spheres of non-spherical particles may overlap with layer interfaces (e.g. a flat particle on a substrate), but care has to be taken with regard to the selection of the numerical parameters. See *[Egel et al. 2016b] and [Egel et al. 2017]* for a discussion. Use of Smuthi's automatic parameter selection feature is recommended.
- The circumscribing spheres of non-spherical particles must not overlap with each other. There is a Smuthi package to allow for plane-wave mediated particle coupling developed by Dominik Theobald which allows to treat particles with overlaping circumscribing spheres, but this package is still in beta and requires expert knowledge to be used.

3.1.4 The simulation class

The simulation object is the central manager of a Smuthi simulation. To define a simulation, you need to at least specify the optical system, i.e., an initial field, a layer system and a list of scattering particles.

In addition, you can provide a number of input parameters regarding numerical parameters or solver settings which you can view in the API documentation: `smuthi.simulation.Simulation`.

For your first simulations, you can probably just go with the default parameters. However, when approaching numerically challanging systems or if you are interested to optimize the runtime, we recommend to read the sections xyz to get an overview and to study the corresponding tutorial scripts.

Todo: Add links to sections and examples

3.1.5 Post processing

Once the *smuthi.simulation.Simulation.run()* method has successfully terminated, we still need to process the results into the desired simulation output. Smuthi offers data structures to obtain near and far field distributions as well as scattering cross sections. Below, we give a short overview on a couple of convenience functions that can be used to quickly generate some output.

- **Near fields** are electric field distributions as a function of position, $\mathbf{E} = \mathbf{E}(\mathbf{r})$. The term *near field* is opposed to *far field* which is an intensity distribution in direction space. Near field does *not* imply that the field is evaluated very close to the particles. If you want to generate plots or animations of the electric field distribution, we recommend to use the *smuthi.postprocessing.graphical_output.show_near_field()* function. This is a very flexible and powerful function that allows a couple of settings which you can study in the API documentation.

Note: Spheres allow the evaluation of near fields everywhere (inside and outside the particles). Non-spherical particles allow the evaluation only outside the particles. Please also note that the computed near fields inside the circumscribing sphere of non-spherical particles are in general not correct.

- **Far fields** are intensity distributions in direction space (i.e., power per solid angle, measured far away from the scattering centers). We recommend to have a look at the functions *smuthi.postprocessing.graphical_output.show_scattered_far_field()*, *smuthi.postprocessing.graphical_output.show_total_far_field()* and *smuthi.postprocessing.graphical_output.show_scattering_cross_section()* and to study their input parameters in the API documentation.

$$W = \int_0^{2\pi} \int_0^{\pi} I(\alpha, \beta) \sin\beta \mathrm{d}\beta \mathrm{d}\alpha$$

- **Cross sections**: If the initial field was a plane wave, the total scattering cross section as well as the extinction cross section can be evaluated. Please view the section *Cross sections* for details.

If you need post processing that goes beyond the described functionality, we recommend to browse through the API documentation of the *smuthi.postprocessing* package or directly through the source code and construct your own post processing machinery from the provided data structure.

3.2 Physical units

Smuthi is commited to a "relative units" philosophy. That means, all quantities have only relative meaning.

3.2.1 Length units

The user is free to select the unit in which all lengths are provided. Just make sure that particle sizes, layer thicknesses and wavelengths are all specified in the same unit. Results will automatically refer to the same unit. For example, if you specify the wavelength in nanometers, resulting cross sections will be in square nanometers. Besides, quantities with an inverse length dimension (wavenumbers) also implicitly refer to the selected length unit.

3.2.2 Field strength units

When the electromagnetic fields are computed, their absolute value has no physical meaning. Only relative quantities can be used for further analysis. For example, the scattered field strength divided by the amplitude of the initial field *does* have a physical meaning.

3.2.3 Power units

Also power units have no meaning as absolute values. To get meaningful information, power-related figures always need to be guarded in reference to other power-related figures. Some examples:

- Scattering cross section as the quotient of scattered (angular) intensity and incident (power-per-area) intensity.
- Diffuse reflectivity as the total back scattered far field power divided by the initial Gaussian beam power.
- Purcell factor as the dissipated power of a dipole source divided by the dissipated power of the same source in the absence of planar interfaces and scattering particles.

3.3 Cross sections

If the initial excitation is given by a plane wave, it is natural to discuss the far field properties of a scattering structure in terms of cross sections.

However, in the context of scattering particles near planar interfaces, the commonly used concepts of cross sections need further clarification. In the following, we therefore discuss the meaning of cross sections as they are implemented in Smuthi.

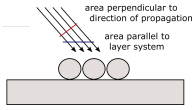
3.3.1 Scattering cross section

The concept of a scattering cross section is straightforward: The incoming wave is specified by an intensity (power per area), whereas the scattered field is characterized by a power, such that the scattered signal divided by the initial signal yields an area.

The total scattering cross section reads

$$C_{\mathrm{scat}} = \frac{W_{\mathrm{scat}}}{I_{\mathrm{inc}}}$$

where W_{scat} is the total scattered power and I_{inc} is the incident irradiance (power per unit area perpendicular to the direction of propagation).

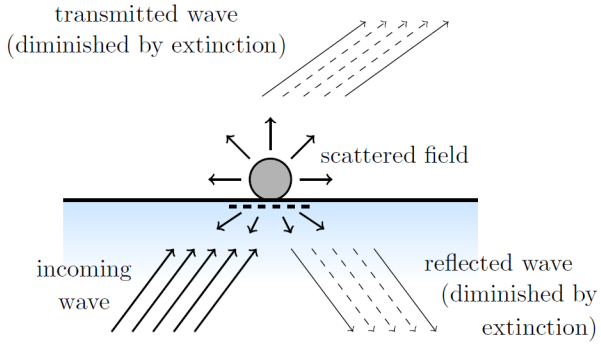


Note: In Smuthi versions < 1.0, a different definition of cross sections was used. In these versions, the incident irradiance was defined as "power per unit area parallel to the layer system", such that cross section figures computed with previous versions can deviate from the current version by a factor $\cos(\beta_{\mathrm{inc}})$, where β_{inc} is the propagation angle of the incoming plane wave.

3.3.2 Extinction cross section

The term "extinction" means that particles take away power from the incindent plane wave, such that they partially extinguish the incindent wave. The power that they take away from the incoming wave is either absorbed or scattered into other channels, such that in the context of scattering of a plane wave by particles in a homogeneous medium, the extinction cross section is usually defined as the sum of the total scattering cross section and the absorption cross section.

However, this interpretation of extinction (i.e., the sum of particle absorption and scattering) is not applicable when besides the particle there is also a planarly layered medium involved. The reason is that besides particle absorption and scattering, also absorption in the layered medium has to be taken into account.



Instead, we apply what is usually referred to as the optical theorem to define extinction (please see section 3.8.1 of *[Egel 2018]* for the mathematical details). This way, we take the term "extinction" serious and provide a measure for "how much power is taken away by the particles from the incident plane wave?"

In fact, Smuthi computes two extinction cross sections: one for the reflected incoming wave and one for the transmitted incoming wave. That means, the extinction cross section for reflection (transmission) refers to the destructive interference of the scattered signal with the specular reflection (transmission) of the initial wave. It thereby includes absorption in the particles, scattering, and a modified absorption by the layer system, e.g. through incoupling into waveguide modes.

As a consequence, the extinction cross sections can be negative if (for example due to a modified absorption in the layer system) more light is reflected (or transmitted) in the specular direction than would be without the particles.

Conservation of energy is then expressed by the following statement: "For lossless particles near or inside a lossless planarly layered medium (that doesn't support any waveguide modes), the sum of top and bottom extinction cross section equals the total scattering cross section".

3.4 Multipole cut-off

The scattering properties of each particle are represented by its T-matrix $T_{plm,p'l'm'}$ where plm and $p'l'm'$ are the multipole polarization, degree and order of the scattered and incoming field, respectively, see sections 3.3 and 2.3.2 of *[Egel 2018]*. In practice, the T-matrix is truncated at some multipole degree $l_{max} \geq 1$ and order $0 \leq m_{max} \leq l_{max}$ to obtain a finite system of linear equations.

Specify the cut-off parameters for each particle like this:

```
large_sphere = smuthi.particles.Sphere( ...
                                        l_max=10,
                                        m_max=10,
                                        ...)
```

(continues on next page)

(continued from previous page)

```
small_sphere = smuthi.particles.Sphere( ...
                                        l_max=3,
                                        m_max=3,
                                        ...)
```

In general, we can say:

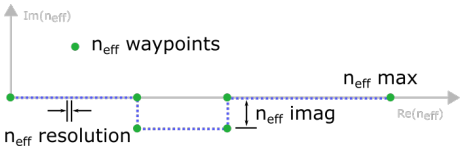
- Large particles require higher multipole orders than small particles.
- Particles very close to each other, very close to an interface or very close to a point dipole source require higher multipole orders than those that stand freely.
- Larger multipole cutoff parameters imply better accuracy, but also a quickly growing numerical effort.
- When simulating flat particles near planar interfaces, the multipole truncation should be chosen with regard to the Sommerfeld integral truncation. See *[Egel et al. 2017]*.

Literature offers various rules of thumb for the selection of the multipole truncation in the case of spherical particles, see for example *[Neves 2012]* or *[Wiscombe 1980]*.

Otherwise, you can use Smuthi's built-in automatic parameter selection feature to estimate a suitable multipole truncation, see section on *Automatic parameter selection*.

3.5 Complex integral contours

Sommerfeld integrals arise in the treatment of the layer system response to the scattered field or to the initial field (in case of dipole excitation). Their numerical evaluation relies on an integral contour that is deflected into the complex plane in order to avoid sharp features stemming from waveguide mode singularities (see the section on SommerfeldAnchor for a short discussion).



3.5.1 Default settings

If you specify no input arguments with regard to the integral contours, default settings are applied. Note, however, that this does not guarantee accurate results in all use cases.

3.5.2 Automatic contour definition

If you want to be on the safe side, use the automatic parameter selection feature to obtain a suitable integral contour, see section on *Automatic parameter selection*. The drawback is a substantially enhanced runtime, as the simulation is repeated multiple times until the result converges.

3.5.3 Manual contour definition

We recommend to use the *neff_imag*, *neff_max* and *neff_resolution* input parameter of the `smuthi.simulation.Simulation` constructor. Smuthi will construct contours based on this input and store them for the duration of the simulation as default contours for multiple scattering and initial fields in the `smuthi.fields` module.

- *neff_imag* states how far into the negative imaginary the contour will be deflected in terms of the dimensionless effective refractive index, $n_{\mathrm{eff}} = \frac{c\kappa}{\omega}$
- *neff_max* is the point where the contour ends (on the real axis). Instead of *neff_max*, you can also provide *neff_max_offset* which specifies, how far *neff_max* should be chosen away from the largest relevant layer refractive index.
- *neff_resolution* denotes the distance Δn_{eff} between two adjacent sampling points (again in terms of effective refractive index).

The locations where the waypoints mark a deflection into the imaginary are chosen with consideration of the involved layer system refractive indices (see the section on SommerfeldAnchor for a discussion why that is necessary).

This is how a call to the simulation contructor could look like:

```
simulation = smuthi.simulation.Simulation( ...
                                           neff_imag=1e-2,
                                           neff_max=2.5,
                                           neff_resolution=5e-3,
                                           ... )
```

Note: If you need more control over the shape of the contour, read through the API documentation or contact the support mailing list (see *Contact*).

Multiple scattering and initial field contours

In some use cases it makes sense to specify the contour for multiple scattering with different parameters than the contour for the initial field. For example, when a dipole is very close to an interface, but the particle centers are not.

In that case you can use the function `reasonable_Sommerfeld_kpar_contour` (see *fields*) to construct an array of *k_parallel* values for each initial field and multiple scattering purposes, like this:

```
# construct contour arrays
init_kpar = smuthi.fields.reasonable_Sommerfeld_kpar_contour( ... )
scat_kpar = smuthi.fields.reasonable_Sommerfeld_kpar_contour( ... )

# assign them to the respective objects
simulation = smuthi.initial_field.DipoleSource( ...
                                                k_parallel=scat_kpar,
                                                ... )

simulation = smuthi.simulation.Simulation( ...
                                           k_parallel=scat_kpar,
                                           ... )
```

3.5.4 Guidelines for parameter selection

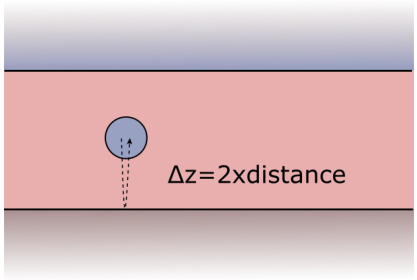
Contour truncation

The contour truncation scale *neff_max* is a real number which specifies where the contour ends. It should be larger than the refractive index of the layer in which the particle resides. The offset $n_{\mathrm{eff}} - n$ should be chosen with regard to the distance between the particles (and point sources) to the next layer interface. If that distance is large, the truncation scale is uncritical, whereas whereas point sources or particles whose center is very close to a layer interface require a larger offset.

At a z-distance of Δz, evanescent waves with an effective refractive index of n_{eff} are damped by a factor of

$$\exp\left(2\pi \mathrm{i} \frac{\Delta z}{\lambda} \sqrt{n_{\text{eff}}^2 - n^2}\right),$$

where λ is the vacuum wavelength and n is the refractive index of the medium.



To select a reasonable *neff_max*, we should consider that the shortest possible interaction path is *twice* the z-distance between some particle center (or dipole position) and the next layer interface.

Uncritical example

A layer system consists of a substrate ($n = 1.5$), covered with a 1000nm thick layer of titania ($n = 2.1$) under air ($n = 1$). A silica sphere is immersed in the middle of the titania layer. The system is illuminated with a plane wave at vacuum wavelength of 550nm.

Then, $\Delta z = 2 \times 500\text{nm}$ such that evanescent waves with $n_{\text{eff}} = 2.3$ are already damped by a factor of $\exp(-2\pi \frac{1000\text{nm}}{550\text{nm}} \sqrt{(2.3^2 - 2.1^2)}) \approx 2 \times 10^{-5}$ when they propagate to the layer interface and back to the sphere. Waves beyond that effective refractive index thus can be safely neglected in the particle-layer system interaction, such that a truncation parameter of $n_{\text{eff,max}} = 2.3$ is reasonable.

Critical example

A layer system consists of a substrate ($n = 1.5$), under air ($n = 1$). A point dipole source of wavelength 550nm is located 10nm above the substrate/air interface.

Here we need to consider $\Delta z = 2 \times 10\text{nm}$ such that Then, evanescent waves with $n_{\text{eff}} = 2.3$ are only damped by a factor of $\exp(-2\pi \frac{20nm}{550nm} \sqrt{(2.3^2 - 1^2)}) \approx 0.62$ when scattered by the layer interface. Even a truncation of $n_{\text{eff,max}} = 10$ would only lead to an evanescent damping of $\exp(-2\pi \frac{20nm}{550nm} \sqrt{(10^2 - 1^2)}) \approx 0.1$ which might still not be enough.

Resolution

In Smuthi, Sommerfeld integrals are addressed numerical by means of the trapezoidal rule. The discretization of the integrand along the integration contour is determined by the parameter `neff_resolution` which specifies the distance of one integration node to the next in terms of the effective refractive index. In general, a finer resolution leads to a better accuracy and a longer runtime during preprocessing (i.e., when the particle coupling lookup is computed) as well as during post processing (when the electric field is computed from a plane wave pattern).

The following situations can require a fine sampling of the integrands:

- when a high accuracy is desired
- when waveguide modes and branch point singularities render a numerically challenging integrand of the Sommerfeld integrals (this can be avoided by a deflection into the imaginary, see below)

- when particles with a large distance to each other are part of the simulation geometry

To understand the latter point, consider the Sommerfeld integral as a Hankel transform. Like in a Fourier transform, a large lateral distance requires a fine sampling of the wavenumber to avoid aliasing. It thus is advised to select `neff_resol` below $2/(k\rho_{\max})$, where $k = 2\pi/\lambda$ is the vacuum wavenumber and $\rho_{\max}$ is the largest lateral distance between two particles.

Deflection into imaginary

Near waveguide mode or branchpoint singularities, the integrand of the Sommerfeld integrals may be a rapidly varying function (in case of lossless media, the waveguide mode singularities are located on the real axis, such that the integrand is even singular). In that case, a deflection of the integral contour into the complex plane can improve the accuracy of the numerical integrals for a given sampling resolution, see also the section on SommerfeldAnchor. The extent of that deflection is set by the `neff_imag` parameter.

Note: Care has to be taken when selecting the `neff_imag` parameter, especially in the case of large lateral distances between the particles.

- The larger `neff_imag`, the stronger is the smoothing effect on the Sommerfeld integrand
- For large lateral distances, a too large `neff_imag` can lead to significant errors! To understand this point, consider the Sommerfeld integral as a Hankel transform, involving expressions of type $J_\nu(\kappa\rho)$, where J_ν is the Bessel function, κ is the in-plane wavenumber (which is proportional to n_{eff}) and ρ is the lateral distance between the particles. Note that the Bessel functions grow rapidly arguments with a large negative imaginary part - which can lead to numerical problems in the integration.

Again, it is thus advised to select `neff_imag` below $2/(k\rho_{\max})$, where $k = 2\pi/\lambda$ is the vacuum wavenumber and $\rho_{\max}$ is the largest lateral distance between two particles.

3.6 Automatic parameter selection

Smuthi offers a module to run an automated convergence test for the following parameters:

- Multipole truncation parameters *l_max* and *m_max* for each particle
- Sommerfeld integral contour parameters *neff_max* and *neff_resolution*.
- Angular resolution of far field data

3.6.1 Parameter selection procedure

The user provides:

- a *simulation* object
- a detector function
- a realtive tolerance
- some other numerical settings

The detector function

The **detector function** is a function defined by the user. It accepts a simulation object (one that has already been run) and returns a single quantity which we call the detector quantity. In other words, the detector function does some *post processing* to yield a value that we use to monitor convergence. If no function but one of the strings "extinction cross section" , "total scattering cross section" or "integrated scattered far field" is specified, the corresponding figure is used as the detector quantity. Other possible detector functions could map to the electric field at a certain point, or the scattered far field in a certain direction or whatever seems to the user to be a suitable measure for convergence of the simulation.

Parameter selection algorithm

The automatic parameter selection routine repeatedly runs the simulation and evaluates the detector quantity with subsequently modified numerical input parameters until the relative deviation of the detector quantity is less than the specified tolerance.

The below animation illustrates the typical graphical output during a parameter selection routine. The left panel shows the extinction cross section as a function of multipole cutoff *l_max*, where each line corresponds to a different Sommerfeld integral cutoff *neff_max*. The right panel shows the resulting converged extinction cross sections, this time as a function of Sommerfeld integral cutoff.

For flat particles near planar interfaces, the multipole truncation and the Sommerfeld integral truncation cannot be chosen independently, because we are dealing with a *relative convergence*, see *[Egel et al. 2016b]*. In that case, the user can set the *relative_convergence* flag to true (default). In that case, a convergence test for the multipole truncation parameters is triggered during each iteration of the *neff_max* selection routine:

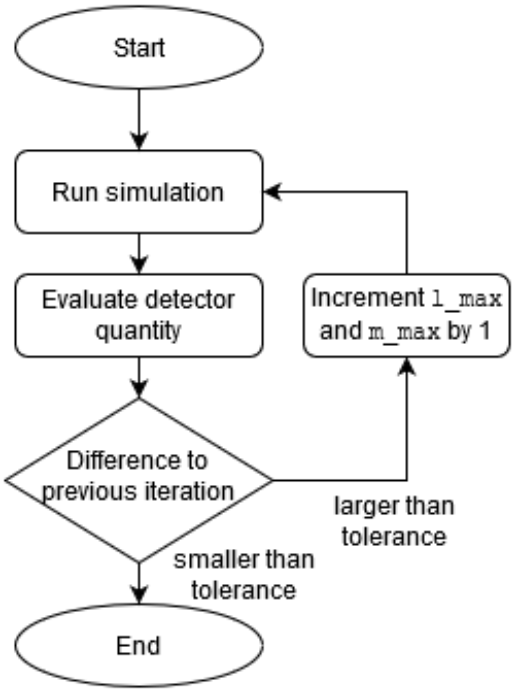


Fig. 1: Selection of *l_max*

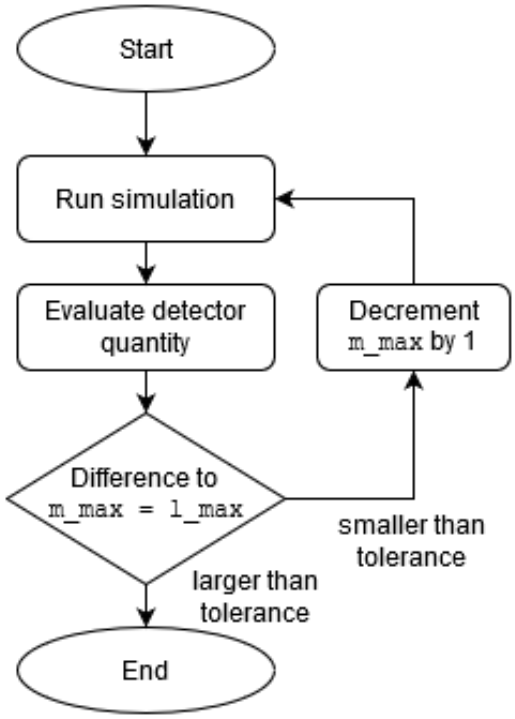


Fig. 2: Selection of *m_max*

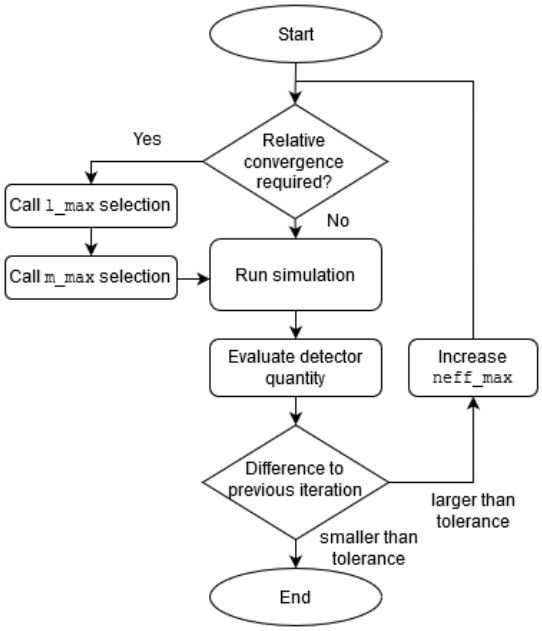


Fig. 3: Selection of *neff_max*

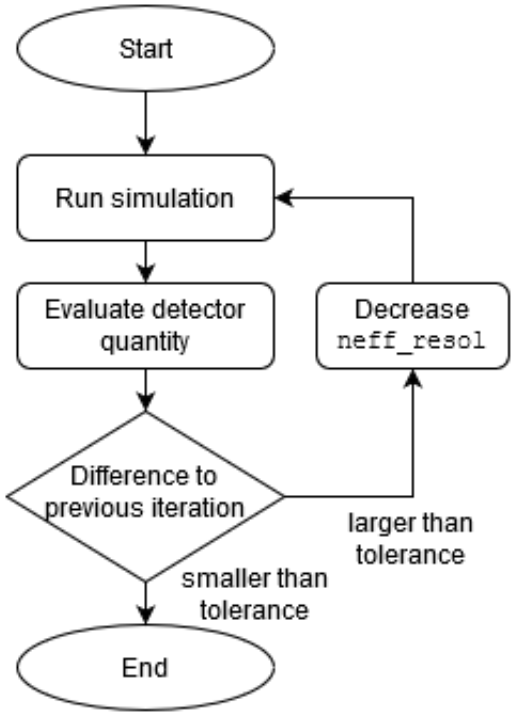


Fig. 4: Selection of *neff_resolution*

Some things to regard when using the automatic parameter selection:

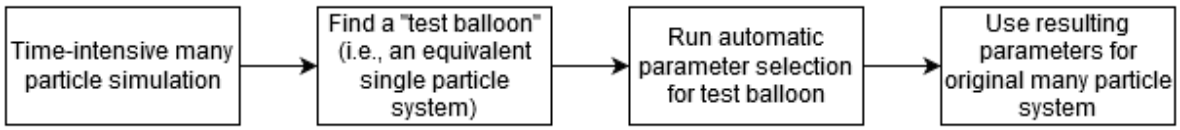
- Both, the multiple scattering and the initial field contour are updated with the same parameters. A seperate optimization of the parameters for initial field and multiple scattering is currently not supported.
- The algorithm compares the detector value for subsequent simulation runs. The idea is that if the simulation results agree for different numerical input parameters, they have probably converged with regard to that parameter. However, in certain cases this assumption can be false, i.e., the simulation results agree although they have not converged. The automatic parameter selection therefore *does not replace critical judging of the results by the user*.
- With the parameter `tolerance_steps`, the user can ask that the tolerance criterion is met multiple times in a row before the routine terminates.
- The simulation is repeated multiple times, such that the automatic parameter selection takes much more time than a single simulation.

For more details, see the API documentation on the *smuthi.utility.automatic_parameter_selection* module.

See also the example on AutoParamExampleAnchor.

3.6.2 Simulations involving many particles

A simulation with many particles can be busy for a considerable runtime. The above described automatic procedure might then be unpractical. In this case, we recommend a strategy of "trial ballooning". The idea is to find a system that takes less time to simulate but that has similar requirements with regard to numerical parameters.



Let us assume that we want to simulate light scattering by one thousand identical flat nano-cylinders located on a thin film system on a substrate. Then, the selection of *neff_max* needs to be done with regard to the distance of the particles to the next planar interface, whereas *l_max* and *m_max* have to be chosen with regard to the particle geometry, material, and to the selected *neff_max*. Finally, *neff_resolution* needs to be chosen with regard to the layer system response. All of these characteristics have nothing to do with the fact that we are interested in a many particles system. We can thus simulate scattering by a single cylinder on the thin film system and let the automatic parameter selection module determine suitable values for *l_max*, *m_max*, *neff_max* and *neff_resolution*. These parameters are then used as input parameters for the 1000-particles simulation which we run without another call to the automatic parameter selection module.

See the example on AutoParamExampleAnchor for an illustration of the procedure.

Note: One needs to be cautious when the many particles simulation involves large lateral distances. In that case, a finer resolution of the complex contour might be required compared to the single-particle test balloon. See the section on *Resolution* for details.

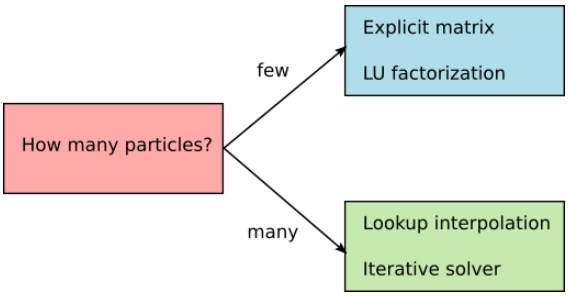
3.7 Solver settings

Note: This section is relevant if you want to simulate systems with many particles

In order to limit the runtime, Smuthi currently offers two numerical strategies for the solution of the scattering problem:

1. LU factorization, that is basically a variant of Gaussian elimination. To this end, the interaction matrix is fully stored in memory.
2. Iterative solution with the GMRES method. In this case, you can either store the full interaction matrix in memory, or use a lookup from which the matrix entries are approximated by interpolation, see Amos Egel's PhD thesis (section 3.10.1) or [Egel, Kettlitz, Lemmer, 2016]

With growing particle number, all involved operations get more expensive, but the costs of LU factorization grow faster than the cost of iterative solution. Similarly, costs of calculating of the full interaction matrix grows faster than the cost of computing a lookup table. For this reason, we recommend the following decision scheme:



The numerical strategy for solving the linear system is defined through the input parameters of the simulation constructor. The relevant parameters are:

- `solver_type`: Either "LU" or "gmres"
- `solver_tolerance`: This parameter defines the abort criterion. If the residual is smaller than the tolerance, the solver halts. The parameter is ignored in case of "LU" solver type.
- `store_coupling_matrix`: If true, the coupling matrix is explicitly calculated and stored in memory. Otherwise, a lookup table is prepared and the matrix-vector multiplications are run on the fly, where the matrix entries are computed using the lookup table. The parameter is ignored in case of "LU" solver type.
- `coupling_matrix_lookup_resolution`: If lookup tables should be used, this needs to be set to a distance value that defines the spatial resolution of the lookup table. The parameter is ignored when the coupling matrix is explicitly calculated.
- `coupling_matrix_interpolator_kind`: If lookup tables should be used, define here either "linear" or "cubic" interpolation. "linear" is faster and "cubic" is more precise for the same resolution. The parameter is ignored when the coupling matrix is explicitly calculated.

This would be a typical setting for a **small number of particles**:

```
simulation = smuthi.simulation.Simulation( ...
                                           solver_type='LU',
                                           store_coupling_matrix=True,
                                           ... )
```

This would be a typical setting for a **large number of particles**:

```
simulation = smuthi.simulation.Simulation( ...
                                           solver_type='gmres',
                                           solver_tolerance=1e-4,
                                           store_coupling_matrix=False,
```

(continues on next page)

(continued from previous page)

```
                                        coupling_matrix_lookup_resolution=5,
                                        coupling_matrix_interpolator_kind='linear',
                                        ... )
```

Note that GPU acceleration is currently only available for particle coupling through lookup interpolation.

3.8 Custom particles

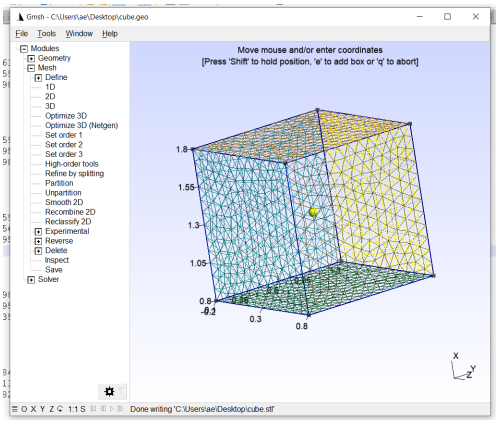
Since version 1.0.0, Smuthi allows to model scattering particles with a user-defined geometry by wrapping the NFM-DS TNONAXSYM functionality.

3.8.1 Creating a FEM file

The particle surface must be specified in a FEM file.

- The first line is the number of surfaces.
- For each surface, the first line is the number of mesh elements
- Each mesh element is specified by a line containing: element location (x, y, z), element normal (x, y, z), element normal

3.8.2 Creating a FEM file using GMSH



One way to produce a FEM file is to use the GMSH package. For example, to generate a cube, do:

- select "Geometry" → "Elementary entities" → "Add" → "Box"
- enter the parameters to achieve a 1 by 1 by 1 box at the center of the coordinate system
- select "Mesh" → 2D
- optionally: refine mesh by clicking "Mesh" → "Refine by splitting"
- Save the mesh in .stl format by "File" → "Export" and then pick "Mesh - STL Surface"
- In the STL options, select "Per surface". This is important, because a clear distinction between surfaces is required.

The so created .stl file can be converted to a FEM file using the `sumuthi.linearsystem.tmatrix.nfmds.stlmanager` module through the `convert_stl_to_fem()` method.

3.8.3 Include custom particle in a Smuthi simulation

To create a particle object with custom geometry, call the *smuthi.particles.CustomParticle* class, for example:

```
cube = smuthi.particles.CustomParticle(position=[0, 0, 100],
                                       refractive_index=1.52,
                                       scale=100,
                                       l_max=3,
                                       fem_filename="cube.fem")
```

3.9 Plane wave coupling

Warning: The plane-wave coupling module is still in development, and its current functionality is experimental.

If non-spherical particles are located such that their circumscribing spheres overlap, the conventional superposition T-matrix method is not applicable. A coupling method based on a temporary plane wave expansion of the scattered field was developed [Theobald 2017] in order to allow for simulations also in such cases.

Right now, the plane wave coupling can be used if

- direct matrix inversion is selected, see *Solver settings*.
- `m_max` is set to `l_max` for all particles.

3.9.1 Use PVWF coupling in a Smuthi simulation

To activate the PVWF coupling feature in a Smuthi simulation, set the `use_pvwf_coupling` parameter of the simulation constructor to `True` and provide a suitable n_{eff} truncation and discretization with the `pvwf_coupling_neff_max` parameter and the `pvwf_coupling_neff_resolution` parameter of the simulation constructor:

```
simul = smuthi.simulation.Simulation( ...,
                                      use_pvwf_coupling=True,
                                      pvwf_coupling_neff_max=7,
                                      pvwf_coupling_neff_resolution=1e-2)
```

CHAPTER 4

Examples

4.1 Tutorials

This section contains a number of exemplary simulation scripts to illustrate the use of Smuthi. Each tutorial is supposed to illustrate a certain aspect of the software. Click on the respective tutorial names to view a brief discussion.

No.	Tutorial	level	script	Google colab
1	Setting up a simulation		`download`	link
2	Plotting the near field		`download`	link
3	Plotting the far field		`download`	link
4	Non-spherical particles		`download`	n/a
5	Dipole sources		TBD	n/a
6	Gaussian beams		TBD	n/a
7	Automatic parameter selection		`download`	n/a
8	Many particle simulations		`download`	n/a
9	Multipole decomposition		`download`	n/a
10	Periodic near field		`download`	n/a
11	Plane wave coupling		`download`	n/a

4.2 Benchmarks

This section contains a number of benchmarks between Smuthi and other codes with regard to accuracy and/or runtime. Click on the respective benchmark names to view a brief discussion.

No.	Benchmark	other method	script and data
1	Four particles in slab waveguide	FEM	`download`
2	Fifteen periodic spheres in slab	FEM	`download`

CHAPTER 5

API

Smuthi is a Python package with the following modules and sub-packages.

5.1 Top level modules

5.1.1 smuthi.simulation

Provide class to manage a simulation.

class smuthi.simulation.**Simulation**(*layer_system=None*, *particle_list=None*, *initial_field=None*, *k_parallel='default'*, *angular_resolution=0.008726646259971648*, *neff_waypoints=None*, *neff_imag=0.01*, *neff_max=None*, *neff_max_offset=1*, *neff_resolution=0.01*, *neff_minimal_branchpoint_distance=None*, *overwrite_default_contours=True*, *solver_type='LU'*, *solver_tolerance=0.0001*, *store_coupling_matrix=True*, *coupling_matrix_lookup_resolution=None*, *coupling_matrix_interpolator_kind='linear'*, *length_unit='length unit'*, *input_file=None*, *output_dir='smuthi_output'*, *save_after_run=False*, *log_to_file=False*, *log_to_terminal=True*, *check_circumscribing_spheres=True*, *do_sanity_check=True*, *periodicity=None*, *ewald_sum_separation_parameter='default'*, *number_of_threads_periodic='default'*, *use_pvwf_coupling=False*, *pvwf_coupling_neff_max=None*, *pvwf_coupling_neff_resolution=0.01*)

Central class to manage a simulation.

Parameters

- **layer_system** (smuthi.layers.LayerSystem) – stratified medium
- **particle_list** (*list*) – list of smuthi.particles.Particle objects
- **initial_field** (smuthi.initial_field.InitialField) – initial field object
- **k_parallel** (*numpy.ndarray or str*) – in-plane wavenumber for Sommerfeld integrals and field expansions. if 'default', use smuthi.fields.default_Sommerfeld_k_parallel_array
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range
- **neff_waypoints** (*list or ndarray*) – Used to set default k_parallel arrays. Corner points through which the contour runs This quantity is dimensionless (effective refractive index, will be multiplied by vacuum wavenumber) *Multipole cut-off* If not provided, reasonable waypoints are estimated.
- **neff_imag** (*float*) – Used to set default k_parallel arrays. Extent of the contour into the negative imaginary direction (in terms of effective refractive index, n_eff=kappa/omega). Only needed when no neff_waypoints are provided
- **neff_max** (*float*) – Used to set default k_parallel arrays. Truncation value of contour (in terms of effective refractive index). Only needed when no neff_waypoints are provided
- **neff_max_offset** (*float*) – Used to set default k_parallel arrays. Use the last estimated singularity location plus this value (in terms of effective refractive index). Default=1 Only needed when no *neff_waypoints* are provided and if no value for *neff_max* is specified.
- **neff_resolution** (*float*) – Used to set default k_parallel arrays. Resolution of contour, again in terms of effective refractive index
- **neff_minimal_branchpoint_distance** (*float*) – Used to set default k_parallel arrays. Minimal distance that contour points shall have from branchpoint singularities (in terms of effective refractive index). This is only relevant if not deflected into imaginary. Default: One fifth of neff_resolution
- **overwrite_default_contours** (*bool*) – If true (default), the default contours are written even if they have already been defined before
- **solver_type** (*str*) – What solver type to use? Options: 'LU' for LU factorization, 'gmres' for GMRES iterative solver
- **coupling_matrix_lookup_resolution** (*float or None*) – If type float, compute particle coupling by interpolation of a lookup table with that spacial resolution. If None (default), don't use a lookup table but compute the coupling directly. This is more suitable for a small particle number.
- **coupling_matrix_interpolator_kind** (*str*) – Set to 'linear' (default) or 'cubic' interpolation of the lookup table.
- **store_coupling_matrix** (*bool*) – If True (default), the coupling matrix is stored. Otherwise it is recomputed on the fly during each iteration of the solver.
- **length_unit** (*str*) – what is the physical length unit? has no influence on the computations
- **input_file** (*str*) – path and filename of input file (for logging purposes)
- **output_dir** (*str*) – path to folder where to export data
- **save_after_run** (*bool*) – if true, the simulation object is exported to disc when over
- **log_to_file** (*bool*) – if true, the simulation log will be written to a log file

- **log_to_terminal** (*bool*) – if true, the simulation progress will be displayed in the terminal
- **check_circumscribing_spheres** (*bool*) – if true, check all particles for overlapping circumscribing spheres and print a warning if detected
- **do_sanity_check** (*bool*) – if true (default), check numerical input for some flaws. Warning: A passing sanity check does not guarantee correct numerical settings. For many particles, the sanity check might take some time and/or occupy large memory.
- **periodicity** (*tuple*) – tuple (a1, a2) specifying two 3-dimensional lattice vectors in Carthesian coordinates with a1, a2 (numpy.ndarrays)
- **ewald_sum_separation_parameter** (*float*) – Used to separate the real and reciprocal lattice sums to evaluate particle coupling in periodic lattices.
- **number_of_threads_periodic** (*int or str*) – sets the number of threats used in a simulation with periodic particle arrangements if 'default', all available CPU cores are used if negative, all but number_of_threads_periodic are used
- **use_pvwf_coupling** (*bool*) – If set to True, plane wave coupling is used to calculate the direct. Currently only possible in combination with direct solver strategy.
- **pvwf_coupling_neff_max** (*float*) – Truncation neff for the integration contour of the PVWF coupling integral
- **pvwf_coupling_neff_resolution** (*float*) – Discretization of the neff integral for PVWF coupling

circumscribing_spheres_disjoint()
Check if all circumscribing spheres are disjoint

initialize_linear_system()

largest_lateral_distance()
Compute the largest lateral distance between any two particles

print_simulation_header()

run()

sanity_check()
Check contour parameters for obvious problems

save(*filename=None*)
Export simulation object to disc.

Parameters **filename** (*str*) – path and file name where to store data

set_default_Sommerfeld_contour()
Set the default Sommerfeld k_parallel array

set_default_angles()
Set the default polar and azimuthal angular arrays for pre-processing (i.e., initial field expansion)

set_default_contours()
Set the default initial field k_parallel array and the default Sommerfeld k_parallel array

set_default_initial_field_contour()
Set the default initial field k_parallel array

set_logging(*log_to_terminal=None*, *log_to_file=None*, *log_filename=None*)
Update logging behavior.

Parameters

- **log_to_terminal** (*logical*) – If true, print output to console.
- **log_to_file** (*logical*) – If true, print output to file
- **log_filename** (*char*) – If *log_to_file* is true, print output to a file with that name in the output directory. If the file already exists, it will be appended.

5.1.2 smuthi.initial_field

This module defines classes to represent the initial excitation.

class smuthi.initial_field.**DipoleCollection**(*vacuum_wavelength*, *k_parallel_array='default'*, *azimuthal_angles_array='default'*, *angular_resolution=None*, *compute_swe_by_pwe=False*, *compute_dissipated_power_by_pwe=False*)

Class for the representation of a set of point dipole sources. Use the append method to add DipoleSource objects.

Parameters

- **vacuum_wavelength** (*float*) – vacuum wavelength (length units)
- **k_parallel_array** (*numpy.ndarray or str*) – In-plane wavenumber. If 'default', use smuthi.fields.default_initial_field_k_parallel_array
- **azimuthal_angles_array** (*numpy.ndarray or str*) – Azimuthal angles for plane wave expansions If 'default', use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range
- **compute_swe_by_pwe** (*bool*) – If True, the initial field coefficients are computed through a plane wave expansion of the whole dipole collection field. This is slower for few dipoles and particles, but can become faster than the default for many dipoles and particles (default=False).
- **compute_dissipated_power_by_pwe** (*bool*) – If True, evaluate dissipated power through a plane wave expansion of the whole scattered field. This is slower for few dipoles, but can be faster than the default for many dipoles (default=False).

append(*dipole*)

Add dipole to collection.

Parameters **dipole** (DipoleSource) – Dipole object to add.

dissipated_power(*particle_list*, *layer_system*, *k_parallel='default'*, *azimuthal_angles='default'*, *angular_resolution=None*)

Compute the power that the dipole collection feeds into the system.

It is computed according to

$$P = \sum_i P_{0,i} + \frac{\omega}{2}\mathrm{Im}(\mu_i^* \cdot \mathbf{E}_i(\mathbf{r}_i))$$

where $P_{0,i}$ is the power that the i-th dipole would feed into an infinte homogeneous medium with the same refractive index as the layer that contains that dipole, $\mathbf{r}_i$ is the location of the i-th dipole, ω is the angular frequency, μ_i is the dipole moment and $\mathbf{E}_i$ includes the reflections of the dipole field from the layer interfaces, as well as the scattered field from all particles and the fields from all other dipoles. In contrast to dissipated_power_alternative, this routine uses the particle coupling routines and might be faster for many particles and few dipoles.

Parameters

- **particle_list** (*list of smuthi.particles.Particle objects*) – scattering particles
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium
- **k_parallel** (*ndarray or str*) – array of in-plane wavenumbers for plane wave expansions. If 'default', use smuthi.fields.default_initial_field_k_parallel_array
- **azimuthal_angles** (*ndarray or str*) – array of azimuthal angles for plane wave expansions. If 'default', use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range

Returns dissipated power of each dipole (list of floats)

dissipated_power_alternative(*particle_list*, *layer_system*, *k_parallel='default'*, *azimuthal_angles='default'*, *angular_resolution=None*)

Compute the power that the dipole collection feeds into the system.

It is computed according to

$$P = \sum_i P_{0,i} + \frac{\omega}{2}\mathrm{Im}(\mu_i^* \cdot \mathbf{E}_i(\mathbf{r}_i))$$

where $P_{0,i}$ is the power that the i-th dipole would feed into an infinte homogeneous medium with the same refractive index as the layer that contains that dipole, $\mathbf{r}_i$ is the location of the i-th dipole, ω is the angular frequency, μ_i is the dipole moment and $\mathbf{E}_i$ includes the reflections of the dipole field from the layer interfaces, as well as the scattered field from all particles and the fields from all other dipoles. In contrast to dissipated_power, this routine uses the scattered field piecewise expansion and might be faster for few particles or many dipoles.

Parameters

- **particle_list** (*list of smuthi.particles.Particle objects*) – scattering particles
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium
- **k_parallel** (*ndarray or str*) – array of in-plane wavenumbers for plane wave expansions. If 'default', use smuthi.fields.default_initial_field_k_parallel_array
- **azimuthal_angles** (*ndarray or str*) – array of azimuthal angles for plane wave expansions. If 'default', use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range

Returns dissipated power of each dipole (list of floats)

electric_field(*x*, *y*, *z*, *layer_system*)

Evaluate the complex electric field of the dipole collection.

Parameters

- **x** (*array like*) – Array of x-values where to evaluate the field (length unit)
- **y** (*array like*) – Array of y-values where to evaluate the field (length unit)
- **z** (*array like*) – Array of z-values where to evaluate the field (length unit)
- **layer_system** (*smuthi.layer.LayerSystem*) – Stratified medium

Returns Tuple (E_x, E_y, E_z) of electric field values

magnetic_field(*x*, *y*, *z*, *layer_system*)
Evaluate the complex magnetic field of the dipole collection.

Parameters

- **x** (*array like*) – Array of x-values where to evaluate the field (length unit)
- **y** (*array like*) – Array of y-values where to evaluate the field (length unit)
- **z** (*array like*) – Array of z-values where to evaluate the field (length unit)
- **layer_system** (*smuthi.layer.LayerSystem*) – Stratified medium

Returns Tuple (H_x, H_y, H_z) of magnetic field values

piecewise_field_expansion(*layer_system*)
Compute a piecewise field expansion of the dipole collection..

Parameters **layer_system** (*smuthi.layer.LayerSystem*) – stratified medium

Returns smuthi.field_expansion.PiecewiseWaveExpansion object

plane_wave_expansion = functools.partial(<bound method Memoize.__call__ of <smuthi.uti

spherical_wave_expansion(*particle*, *layer_system*)
Regular spherical wave expansion of the dipole collection including layer system response, at the locations of the particles. If self.compute_swe_by_pwe is True, use the dipole collection plane wave expansion, otherwise use the individual dipoles spherical_wave_expansion method.

Parameters

- **particle** (smuthi.particles.Particle) – particle relative to which the swe is computed
- **layer_system** (*smuthi.layer.LayerSystem*) – stratified medium

Returns regular smuthi.field_expansion.SphericalWaveExpansion object

class smuthi.initial_field.**DipoleSource**(*vacuum_wavelength*, *dipole_moment*, *position*, *k_parallel_array='default'*, *azimuthal_angles_array='default'*)
Class for the representation of a single point dipole source.

Parameters

- **vacuum_wavelength** (*float*) – vacuum wavelength (length units)
- **dipole_moment** (*list or tuple*) – (x, y, z)-coordinates of dipole moment vector
- **position** (*list or tuple*) – (x, y, z)-coordinates of dipole position
- **k_parallel_array** (*numpy.ndarray or str*) – In-plane wavenumber. If 'default', use smuthi.fields.default_initial_field_k_parallel_array
- **azimuthal_angles_array** (*numpy.ndarray or str*) – Azimuthal angles for plane wave expansions If 'default', use smuthi.fields.default_azimuthal_angles

check_dissipated_power_homogeneous_background(*layer_system*)

current()
The current density takes the form

$$\mathbf{j}(\mathbf{r}) = \delta(\mathbf{r} - \mathbf{r}_D)\mathbf{j}_D,$$

where $\mathbf{j}_D = -j\omega\mu$, $\mathbf{r}_D$ is the location of the dipole, ω is the angular frequency and μ is the dipole moment. For further details, see 'Principles of nano optics' by Novotny and Hecht.

Returns List of [x, y, z]-components of current density vector $\mathbf{j}_D$

dissipated_power (*particle_list*, *layer_system*, *show_progress=True*)
Compute the power that the dipole feeds into the system.

It is computed according to

$$P = P_0 + \frac{\omega}{2}\mathrm{Im}(\mu^* \cdot \mathbf{E}(\mathbf{r}_D))$$

where P_0 is the power that the dipole would feed into an infinte homogeneous medium with the same refractive index as the layer that contains the dipole, $\mathbf{r}_D$ is the location of the dipole, ω is the angular frequency, μ is the dipole moment and $\mathbf{E}$ includes the reflections of the dipole field from the layer interfaces, as well as the scattered field from all particles.

Parameters

- **particle_list** (*list of smuthi.particles.Particle objects*) – scattering particles
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium
- **show_progress** (*bool*) – if true, display progress

Returns dissipated power as float

dissipated_power_alternative (*particle_list*, *layer_system*)
Compute the power that the dipole feeds into the system.

It is computed according to

$$P = P_0 + \frac{\omega}{2}\mathrm{Im}(\mu^* \cdot \mathbf{E}(\mathbf{r}_D))$$

where P_0 is the power that the dipole would feed into an infinte homogeneous medium with the same refractive index as the layer that contains the dipole, $\mathbf{r}_D$ is the location of the dipole, ω is the angular frequency, μ is the dipole moment and $\mathbf{E}$ includes the reflections of the dipole field from the layer interfaces, as well as the scattered field from all particles. In contrast to dissipated_power, this routine relies on the scattered field piecewise expansion and and might thus be slower.

Parameters

- **particle_list** (*list of smuthi.particles.Particle objects*) – scattering particles
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium

Returns dissipated power as float

dissipated_power_homogeneous_background (*layer_system*)
Compute the power that the dipole would radiate in an infinite homogeneous medium of the same refractive index as the layer that contains the dipole.

$$P_0 = \frac{|\mu| k \omega^3}{12\pi}$$

Parameters **layer_system** (smuthi.layers.LayerSystem) – stratified medium

Returns power (float)

electric_field (*x*, *y*, *z*, *layer_system*, *include_direct_field=True*, *include_layer_response=True*)
Evaluate the complex electric field of the dipole source.

Parameters

- **x** (*array like*) – Array of x-values where to evaluate the field (length unit)
- **y** (*array like*) – Array of y-values where to evaluate the field (length unit)
- **z** (*array like*) – Array of z-values where to evaluate the field (length unit)
- **layer_system** (*smuthi.layer.LayerSystem*) – Stratified medium
- **include_direct_field** (*bool*) – if True (default), the direct dipole field is included. otherwise, only the layer response of the dipole field is returned.
- **include_layer_response** (*bool*) – if True (default), the layer response of the dipole field is included. otherwise, only the direct dipole field is returned.

Returns Tuple (E_x, E_y, E_z) of electric field values

magnetic_field(*x*, *y*, *z*, *layer_system*, *include_direct_field=True*, *include_layer_response=True*)
Evaluate the complex magnetic field of the dipole source.

Parameters

- **x** (*array like*) – Array of x-values where to evaluate the field (length unit)
- **y** (*array like*) – Array of y-values where to evaluate the field (length unit)
- **z** (*array like*) – Array of z-values where to evaluate the field (length unit)
- **layer_system** (*smuthi.layer.LayerSystem*) – Stratified medium
- **include_direct_field** (*bool*) – if True (default), the direct dipole field is included. otherwise, only the layer response of the dipole field is returned.
- **include_layer_response** (*bool*) – if True (default), the layer response of the dipole field is included. otherwise, only the direct dipole field is returned.

Returns Tuple (H_x, H_y, H_z) of electric field values

outgoing_spherical_wave_expansion(*layer_system*)
The dipole field as an expansion in spherical vector wave functions.

Parameters **layer_system** (smuthi.layers.LayerSystem) – stratified medium

Returns outgoing smuthi.field_expansion.SphericalWaveExpansion object

piecewise_field_expansion(*layer_system*, *include_direct_field=True*, *include_layer_response=True*)
Compute a piecewise field expansion of the dipole field.

Parameters

- **layer_system** (*smuthi.layer.LayerSystem*) – stratified medium
- **include_direct_field** (*bool*) – if True (default), the direct dipole field is included. otherwise, only the layer response of the dipole field is returned.
- **include_layer_response** (*bool*) – if True (default), the layer response of the dipole field is included. otherwise, only the direct dipole field is returned.

Returns smuthi.field_expansion.PiecewiseWaveExpansion object

plane_wave_expansion(*layer_system*, *i*, *k_parallel_array=None*, *azimuthal_angles_array=None*)
Plane wave expansion of the dipole field.

Parameters

- **layer_system** (*smuthi.layer.LayerSystem*) – stratified medium
- **i** (*int*) – layer number in which to evaluate the expansion
- **k_parallel_array** (*numpy.ndarray*) – in-plane wavenumber array for the expansion. if none specified, self.k_parallel_array is used
- **azimuthal_angles_array** (*numpy.ndarray*) – azimuthal angles for the expansion. if none specified, self.azimuthal_angles_array is used

Returns tuple of to smuthi.field_expansion.PlaneWaveExpansion objects, one for upgoing and one for downgoing component

spherical_wave_expansion(*particle*, *layer_system*)
Regular spherical wave expansion of the wave including layer system response, at the locations of the particles.

Parameters

- **particle** (smuthi.particles.Particle) – particle relative to which the swe is computed
- **layer_system** (*smuthi.layer.LayerSystem*) – stratified medium

Returns regular smuthi.field_expansion.SphericalWaveExpansion object

class smuthi.initial_field.**GaussianBeam**(*vacuum_wavelength*, *polar_angle*, *azimuthal_angle*, *polarization*, *beam_waist*, *k_parallel_array='default'*, *azimuthal_angles_array='default'*, *amplitude=1*, *reference_point=None*)
Class for the representation of a Gaussian beam as initial field.

initial_intensity(*layer_system*)
Evaluate the incoming intensity of the initial field.

Parameters **layer_system** (smuthi.layers.LayerSystem) – Stratified medium

Returns A smuthi.field_expansion.FarField object holding the initial intensity information.

plane_wave_expansion(*layer_system*, *i*, *k_parallel_array=None*, *azimuthal_angles_array=None*)
Plane wave expansion of the Gaussian beam.

Parameters

- **layer_system** (*smuthi.layer.LayerSystem*) – stratified medium
- **i** (*int*) – layer number in which to evaluate the expansion
- **k_parallel_array** (*numpy.ndarray*) – in-plane wavenumber array for the expansion. if none specified, self.k_parallel_array is used
- **azimuthal_angles_array** (*numpy.ndarray*) – azimuthal angles for the expansion. if none specified, self.azimuthal_angles_array is used

Returns tuple of to smuthi.field_expansion.PlaneWaveExpansion objects, one for upgoing and one for downgoing component

propagated_far_field(*layer_system*)
Evaluate the far field intensity of the reflected / transmitted initial field.

Parameters **layer_system** (smuthi.layers.LayerSystem) – Stratified medium

Returns A tuple of smuthi.field_expansion.FarField objects, one for forward (i.e., into the top hemisphere) and one for backward propagation (bottom hemisphere).

class smuthi.initial_field.**InitialField**(*vacuum_wavelength*)
Base class for initial field classes

angular_frequency()
Angular frequency.

Returns Angular frequency (float) according to the vacuum wavelength in units of c=1.

get_azimuthal_angles_array()
Get azimuthal angles array which is either the default array or the one stored in the object

get_k_parallel_array()
Get k_parallel array which is either the default array or the one stored in the object

piecewise_field_expansion(*layer_system*)

plane_wave_expansion(*layer_system*, *i*)
Virtual method to be overwritten.

spherical_wave_expansion(*particle*, *layer_system*)
Virtual method to be overwritten.

class smuthi.initial_field.**InitialPropagatingWave**(*vacuum_wavelength*, *polar_angle*, *azimuthal_angle*, *polarization*, *amplitude=1*, *reference_point=None*)
Base class for plane waves and Gaussian beams

Parameters

- **vacuum_wavelength** (*float*) –
- **polar_angle** (*float*) – polar propagation angle (0 means, parallel to z-axis)
- **azimuthal_angle** (*float*) – azimuthal propagation angle (0 means, in x-z plane)
- **polarization** (*int*) – 0 for TE/s, 1 for TM/p
- **amplitude** (*float or complex*) – Electric field amplitude
- **reference_point** (*list*) – Location where electric field of incoming wave equals amplitude

electric_field(*x*, *y*, *z*, *layer_system*)
Evaluate the complex electric field corresponding to the wave.

Parameters

- **x** (*array like*) – Array of x-values where to evaluate the field (length unit)
- **y** (*array like*) – Array of y-values where to evaluate the field (length unit)
- **z** (*array like*) – Array of z-values where to evaluate the field (length unit)
- **layer_system** (*smuthi.layer.LayerSystem*) – Stratified medium

Returns Tuple (E_x, E_y, E_z) of electric field values

magnetic_field(*x*, *y*, *z*, *layer_system*)
Evaluate the complex magnetic field corresponding to the wave.

Parameters

- **x** (*array like*) – Array of x-values where to evaluate the field (length unit)
- **y** (*array like*) – Array of y-values where to evaluate the field (length unit)
- **z** (*array like*) – Array of z-values where to evaluate the field (length unit)

- **layer_system** (*smuthi.layer.LayerSystem*) – Stratified medium

Returns Tuple (H_x, H_y, H_z) of magnetic field values

piecewise_field_expansion(*layer_system*)
Compute a piecewise field expansion of the initial field.

Parameters **layer_system** (*smuthi.layer.LayerSystem*) – stratified medium

Returns smuthi.field_expansion.PiecewiseWaveExpansion object

spherical_wave_expansion(*particle*, *layer_system*)
Regular spherical wave expansion of the wave including layer system response, at the locations of the particles.

Parameters

- **particle** (smuthi.particles.Particle) – particle relative to which the swe is computed
- **layer_system** (*smuthi.layer.LayerSystem*) – stratified medium

Returns regular smuthi.field_expansion.SphericalWaveExpansion object

class smuthi.initial_field.**PlaneWave**(*vacuum_wavelength*, *polar_angle*, *azimuthal_angle*, *polarization*, *amplitude=1*, *reference_point=None*)
Class for the representation of a plane wave as initial field.

Parameters

- **vacuum_wavelength** (*float*) –
- **polar_angle** (*float*) – polar angle of k-vector (0 means, k is parallel to z-axis)
- **azimuthal_angle** (*float*) – azimuthal angle of k-vector (0 means, k is in x-z plane)
- **polarization** (*int*) – 0 for TE/s, 1 for TM/p
- **amplitude** (*float or complex*) – Plane wave amplitude at reference point
- **reference_point** (*list*) – Location where electric field of incoming wave equals amplitude

plane_wave_expansion(*layer_system*, *i*)
Plane wave expansion for the plane wave including its layer system response. As it already is a plane wave, the plane wave expansion is somehow trivial (containing only one partial wave, i.e., a discrete plane wave expansion).

Parameters

- **layer_system** (smuthi.layers.LayerSystem) – Layer system object
- **i** (*int*) – layer number in which the plane wave expansion is valid

Returns Tuple of smuthi.field_expansion.PlaneWaveExpansion objects. The first element is an upgoing PWE, whereas the second element is a downgoing PWE.

5.1.3 smuthi.layers

Provide class for the representation of planar layer systems.

class smuthi.layers.**LayerSystem**(*thicknesses=None*, *refractive_indices=None*)
Stack of planar layers.

Parameters

- **thicknesses** (*list*) – layer thicknesses, first and last are semi inf and set to 0 (length unit)
- **refractive_indices** (*list*) – complex refractive indices in the form n+jk

is_degenerate()
Returns True if the layer system consists of only two layers of the same material. This function is useful for detecting if layer mediated coupling can be omitted when calculating the coupling between particles.

Returns True if layer system is degenerate. False otherwise.

layer_number(*z*)
Return number of layer that contains point [0,0,z]

If z is on the interface, the higher layer number is selected.

Parameters **z** (*float*) – z-coordinate of query point (length unit)

Returns number of layer containing z

lower_zlimit(*i*)
Return the z-coordinate of lower boundary

The coordinate system is defined such that z=0 corresponds to the interface between layer 0 and layer 1.

Parameters **i** (*int*) – index of layer in question (must be between 0 and number_of_layers-1)

Returns z-coordinate of lower boundary

number_of_layers()
Return total number of layers

Returns number of layers

reference_z(*i*)
Return the anchor point's z-coordinate.

The coordinate system is defined such that z=0 corresponds to the interface between layer 0 and layer 1.

Parameters **i** (*int*) – index of layer in question (must be between 0 and number_of_layers-1)

Returns anchor point's z-coordinate

response(*pwe*, *from_layer*, *to_layer*)
Evaluate the layer system response to an electromagnetic excitation inside the layer system.

Parameters

- **pwe** (*tuple or smuthi.field_expansion.PlaneWaveExpansion*) – Either specify a PlaneWaveExpansion object that that represents the electromagnetic excitation, or a tuple of two PlaneWaveExpansion objects representing the upwards- and downwards propagating partial waves of the excitation.
- **from_layer** (*int*) – Layer number in which the excitation is located
- **to_layer** (*int*) – Layer number in which the layer response is to be evaluated

Returns Tuple (pwe_up, pwe_sown) of PlaneWaveExpansion objects representing the layer system response to the excitation.

upper_zlimit(*i*)
Return the z-coordinate of upper boundary.

The coordinate system is defined such that z=0 corresponds to the interface between layer 0 and layer 1.

Parameters **i** (*int*) – index of layer in question (must be between 0 and number_of_layers-1)

Returns z-coordinate of upper boundary

wavenumber(*layer_number*, *vacuum_wavelength*)

Parameters

- **layer_number** (*int*) – number of layer in question
- **vacuum_wavelength** (*float*) – vacuum wavelength

Returns wavenumber in that layer as float

smuthi.layers.**fresnel_r**(*pol*, *kz1*, *kz2*, *n1*, *n2*)
Fresnel reflection coefficient.

Parameters

- **pol** (*int*) – polarization (0=TE, 1=TM)
- **kz1** (*float or array*) – incoming wave's z-wavenumber (k*cos(alpha1))
- **kz2** (*float or array*) – transmitted wave's z-wavenumber (k*cos(alpha2))
- **n1** (*float or complex*) – first medium's complex refractive index (n+ik)
- **n2** (*float or complex*) – second medium's complex refractive index (n+ik)

Returns Complex Fresnel reflection coefficient (float or array)

smuthi.layers.**fresnel_t**(*pol*, *kz1*, *kz2*, *n1*, *n2*)
Fresnel transmission coefficient.

Parameters

- **pol** (*int*) – polarization (0=TE, 1=TM)
- **kz1** (*float or array*) – incoming wave's z-wavenumber (k*cos(alpha1))
- **kz2** (*float or array*) – transmitted wave's z-wavenumber (k*cos(alpha2))
- **n1** (*float or complex*) – first medium's complex refractive index (n+ik)
- **n2** (*float or complex*) – second medium's complex refractive index (n+ik)

Returns Complex Fresnel transmission coefficient (float or array)

smuthi.layers.**interface_transition_matrix**(*pol*, *kz1*, *kz2*, *n1*, *n2*)
Interface transition matrix to be used in the Transfer matrix algorithm.

Parameters

- **pol** (*int*) – polarization (0=TE, 1=TM)
- **kz1** (*float or array*) – incoming wave's z-wavenumber (k*cos(alpha1))
- **kz2** (*float or array*) – transmitted wave's z-wavenumber (k*cos(alpha2))
- **n1** (*float or complex*) – first medium's complex refractive index (n+ik)
- **n2** (*float or complex*) – second medium's complex refractive index (n+ik)

Returns Interface transition matrix as 2x2 numpy array or as 2x2 mpmath.matrix

smuthi.layers.**layer_propagation_matrix**(*kz*, *d*)
Layer propagation matrix to be used in the Transfer matrix algorithm.

Parameters

- **kz** (*float or complex*) – z-wavenumber (k*cos(alpha))
- **d** (*float*) – thickness of layer

Returns Layer propagation matrix as 2x2 numpy array or as 2x2 mpmath.matrix

smuthi.layers.**layersystem_scattering_matrix**(*pol*, *layer_d*, *layer_n*, *kpar*, *omega*)
Scattering matrix of a planarly layered medium.

Parameters

- **pol** (*int*) – polarization(0=TE, 1=TM)
- **layer_d** (*list*) – layer thicknesses
- **layer_n** (*list*) – complex layer refractive indices
- **kpar** (*float*) – in-plane wavenumber
- **omega** (*float*) – angular frequency in units of c=1: omega=2*pi/lambda

Returns Scattering matrix as 2x2 numpy array or as 2x2 mpmath.matrix

smuthi.layers.**layersystem_transfer_matrix**(*pol*, *layer_d*, *layer_n*, *kpar*, *omega*)
Transfer matrix of a planarly layered medium.

Parameters

- **pol** (*int*) – polarization(0=TE, 1=TM)
- **layer_d** (*list*) – layer thicknesses
- **layer_n** (*list*) – complex layer refractive indices
- **kpar** (*float*) – in-plane wavenumber
- **omega** (*float*) – angular frequency in units of c=1: omega=2*pi/lambda

Returns Transfer matrix as 2x2 numpy array or as 2x2 mpmath.matrix

smuthi.layers.**matrix_inverse**(*m*)

Parameters **m** (*mpmath.matrix or numpy.ndarray*) – matrix to invert

Returns inverse of m with same data type as m1 and m2

smuthi.layers.**matrix_product**(*m1*, *m2*)

Parameters

- **m1** (*mpmath.matrix or numpy.ndarray*) – first matrix
- **m2** (*mpmath.matrix or numpy.ndarray*) – second matrix

Returns matrix product m1 * m2 with same data type as m1 and m2

smuthi.layers.**set_precision**(*prec=None*)
Set the numerical precision of the layer system response. You can use this to evaluate the layer response of unstable systems, for example in the case of evanescent waves in very thick layers. Calculations take longer time if the precision is set to a value other than None (default).

Parameters **prec** (*None or int*) – If None, calculations are done using standard double precision. If int, that many decimal digits are considered in the calculations, using the mpmath package.

5.1.4 smuthi.particles

Classes for the representation of scattering particles.

class smuthi.particles.**AnisotropicSphere**(*position=None*, *euler_angles=None*, *polar_angle=0*, *azimuthal_angle=0*, *refractive_index=(1+0j)*, *radius=1*, *refractive_index_z=(2+0j)*, *l_max=None*, *m_max=None*, *n_rank=None*)

Particle subclass for anisotropic spheres.

Parameters

- **position** (*list*) – Particle position in the format [x, y, z] (length unit)
- **euler_angles** (*list*) – Euler angles [alpha, beta, gamma] in (zy'z"-convention) in radian. Alternatively, you can specify the polar and azimuthal angle of the axis of revolution.
- **polar_angle** (*float*) – Polar angle of axis of revolution.
- **azimuthal_angle** (*float*) – Azimuthal angle of axis of revolution.
- **refractive_index** (*complex*) – Complex refractive index of particle in x-y plane (if not rotated)
- **refractive_index_z** (*complex*) – Complex refractive index of particle along z-axis (if not rotated)
- **radius** (*float*) – Sphere radius
- **l_max** (*int*) – Maximal multipole degree used for the spherical wave expansion of incoming and scattered field
- **m_max** (*int*) – Maximal multipole order used for the spherical wave expansion of incoming and scattered field
- **n_rank** (*int*) – Maximal multipole order used for in NFMDS (default: l_max + 5)

circumscribing_sphere_radius()

Virtual method to be overwritten

compute_t_matrix(*vacuum_wavelength*, *n_medium*)

Return the T-matrix of a particle.

Parameters

- **vacuum_wavelength** (*float*) –
- **n_medium** (*float or complex*) – Refractive index of surrounding medium
- **particle** (smuthi.particles.Particle) – Particle object

Returns T-matrix as ndarray

class smuthi.particles.**CustomParticle**(*position=None*, *euler_angles=None*, *polar_angle=0*, *azimuthal_angle=0*, *refractive_index=(1+0j)*, *geometry_filename=None*, *scale=1*, *l_max=None*, *m_max=None*, *n_rank=None*)

Particle subclass for custom particle shapes defined via FEM file.

Parameters

- **position** (*list*) – Particle position in the format [x, y, z] (length unit)
- **euler_angles** (*list*) – Euler angles [alpha, beta, gamma] in (zy'z"-convention) in radian. Alternatively, you can specify the polar and azimuthal angle of the axis of revolution.

- **polar_angle** (*float*) – Polar angle of axis of revolution.
- **azimuthal_angle** (*float*) – Azimuthal angle of axis of revolution.
- **geometry_filename** (*string*) – Path to FEM file
- **scale** (*float*) – Scaling factor for particle dimensions (relative to provided dimensions)
- **l_max** (*int*) – Maximal multipole degree used for the spherical wave expansion of incoming and scattered field
- **m_max** (*int*) – Maximal multipole order used for the spherical wave expansion of incoming and scattered field
- **n_rank** (*int*) – Maximal multipole order used for in NFMDS (default: l_max + 5)

circumscribing_sphere_radius()
Virtual method to be overwritten

compute_t_matrix(*vacuum_wavelength*, *n_medium*)
Return the T-matrix of a particle.

Parameters

- **vacuum_wavelength** (*float*) –
- **n_medium** (*float or complex*) – Refractive index of surrounding medium
- **particle** (smuthi.particles.Particle) – Particle object

Returns T-matrix as ndarray

class smuthi.particles.**FiniteCylinder**(*position=None*, *euler_angles=None*, *polar_angle=0*, *azimuthal_angle=0*, *refractive_index=(1+0j)*, *cylinder_radius=1*, *cylinder_height=1*, *l_max=None*, *m_max=None*, *n_rank=None*, *use_python_tmatrix=False*, *nint=100*)
Particle subclass for finite cylinders.

Parameters

- **position** (*list*) – Particle position in the format [x, y, z] (length unit)
- **euler_angles** (*list*) – Euler angles [alpha, beta, gamma] in (zy'z"-convention) in radian. Alternatively, you can specify the polar and azimuthal angle of the axis of revolution.
- **polar_angle** (*float*) – Polar angle of axis of revolution.
- **azimuthal_angle** (*float*) – Azimuthal angle of axis of revolution.
- **refractive_index** (*complex*) – Complex refractive index of particle
- **cylinder_radius** (*float*) – Radius of cylinder (length unit)
- **cylinder_height** (*float*) – Height of cylinder, in z-direction if not rotated (length unit)
- **l_max** (*int*) – Maximal multipole degree used for the spherical wave expansion of incoming and scattered field
- **m_max** (*int*) – Maximal multipole order used for the spherical wave expansion of incoming and scattered field
- **n_rank** (*int*) – Maximal multipole order used for in NFMDS (default: l_max + 5)
- **use_python_tmatrix** (*bool*) – If true, use Alan Zhan's Python code to compute the T-matrix rather than NFM-DS

- **nint** (*int*) – Number of angles used in integral (only for python t-mnatrix)

circumscribing_sphere_radius()
Virtual method to be overwritten

compute_t_matrix(*vacuum_wavelength*, *n_medium*)
Return the T-matrix of a particle.

Parameters

- **vacuum_wavelength** (*float*) –
- **n_medium** (*float or complex*) – Refractive index of surrounding medium
- **particle** (smuthi.particles.Particle) – Particle object

Returns T-matrix as ndarray

class smuthi.particles.**LayeredSpheroid**(*position=None*, *euler_angles=None*, *polar_angle=0*, *azimuthal_angle=0*, *layer_refractive_indices=(1+0j)*, *layer_semi_axes_c=1*, *layer_semi_axes_a=1*, *l_max=None*, *m_max=None*, *n_rank=None*)
Particle subclass for layered spheroid.

Parameters

- **position** (*list*) – Particle position in the format [x, y, z] (length unit)
- **euler_angles** (*list*) – Euler angles [alpha, beta, gamma] in (zy'z"-convention) in radian. Alternatively, you can specify the polar and azimuthal angle of the axis of revolution.
- **polar_angle** (*float*) – Polar angle of axis of revolution.
- **azimuthal_angle** (*float*) – Azimuthal angle of axis of revolution.
- **layer_refractive_indices** (*complex*) – Complex refractive index of particle
- **layer_semi_axes_c** (*float*) – Spheroid half axis in direction of axis of revolution (z-axis if not rotated)
- **layer_semi_axes_a** (*float*) – Spheroid half axis in lateral direction (x- and y-axis if not rotated)
- **l_max** (*int*) – Maximal multipole degree used for the spherical wave expansion of incoming and scattered field
- **m_max** (*int*) – Maximal multipole order used for the spherical wave expansion of incoming and scattered field
- **n_rank** (*int*) – Maximal multipole order used in NFMDS (default: l_max + 5)

compute_t_matrix(*vacuum_wavelength*, *n_medium*)
Return the T-matrix of a particle.

Parameters

- **vacuum_wavelength** (*float*) –
- **n_medium** (*float or complex*) – Refractive index of surrounding medium
- **particle** (smuthi.particles.Particle) – Particle object

Returns T-matrix as ndarray

class smuthi.particles.**Particle**(*position=None*, *euler_angles=None*, *refractive_index=(1+0j)*, *l_max=None*, *m_max=None*)

Base class for scattering particles.

Parameters

- **position** (*list*) – Particle position in the format [x, y, z] (length unit)
- **euler_angles** (*list*) – Particle Euler angles in the format [alpha, beta, gamma]
- **refractive_index** (*complex*) – Complex refractive index of particle
- **l_max** (*int*) – Maximal multipole degree used for the spherical wave expansion of incoming and scattered field
- **m_max** (*int*) – Maximal multipole order used for the spherical wave expansion of incoming and scattered field

circumscribing_sphere_radius()

Virtual method to be overwritten

compute_t_matrix(*vacuum_wavelength*, *n_medium*)

Return the T-matrix of a particle.

Parameters

- **vacuum_wavelength** (*float*) –
- **n_medium** (*float or complex*) – Refractive index of surrounding medium
- **particle** (smuthi.particles.Particle) – Particle object

Returns T-matrix as ndarray

is_inside(*x*, *y*, *z*)

Virtual method to be overwritten. Until all child classes implement it: return False

is_outside(*x*, *y*, *z*)

Virtual method to be overwritten. Until all child classes implement it: return True

class smuthi.particles.**Sphere**(*position=None*, *refractive_index=(1+0j)*, *radius=1*, *l_max=None*, *m_max=None*)

Particle subclass for spheres.

Parameters

- **position** (*list*) – Particle position in the format [x, y, z] (length unit)
- **refractive_index** (*complex*) – Complex refractive index of particle
- **radius** (*float*) – Particle radius (length unit)
- **l_max** (*int*) – Maximal multipole degree used for the spherical wave expansion of incoming and scattered field
- **m_max** (*int*) – Maximal multipole order used for the spherical wave expansion of incoming and scattered field

circumscribing_sphere_radius()

Virtual method to be overwritten

compute_t_matrix(*vacuum_wavelength*, *n_medium*)

Return the T-matrix of a particle.

Parameters

- **vacuum_wavelength** (*float*) –

- **n_medium** (*float or complex*) – Refractive index of surrounding medium
- **particle** (smuthi.particles.Particle) – Particle object

Returns T-matrix as ndarray

is_inside(*x*, *y*, *z*)
Virtual method to be overwritten. Until all child classes implement it: return False

is_outside(*x*, *y*, *z*)
Virtual method to be overwritten. Until all child classes implement it: return True

class smuthi.particles.**Spheroid**(*position=None*, *euler_angles=None*, *polar_angle=0*, *azimuthal_angle=0*, *refractive_index=(1+0j)*, *semi_axis_c=1*, *semi_axis_a=1*, *l_max=None*, *m_max=None*, *n_rank=None*)
Particle subclass for spheroids.

Parameters

- **position** (*list*) – Particle position in the format [x, y, z] (length unit)
- **euler_angles** (*list*) – Euler angles [alpha, beta, gamma] in (zy'z"-convention) in radian. Alternatively, you can specify the polar and azimuthal angle of the axis of revolution.
- **polar_angle** (*float*) – Polar angle of axis of revolution.
- **azimuthal_angle** (*float*) – Azimuthal angle of axis of revolution.
- **refractive_index** (*complex*) – Complex refractive index of particle
- **semi_axis_c** (*float*) – Spheroid half axis in direction of axis of revolution (z-axis if not rotated)
- **semi_axis_a** (*float*) – Spheroid half axis in lateral direction (x- and y-axis if not rotated)
- **l_max** (*int*) – Maximal multipole degree used for the spherical wave expansion of incoming and scattered field
- **m_max** (*int*) – Maximal multipole order used for the spherical wave expansion of incoming and scattered field
- **n_rank** (*int*) – Maximal multipole order used in NFMDS (default: l_max + 5)

circumscribing_sphere_radius()
Virtual method to be overwritten

compute_t_matrix(*vacuum_wavelength*, *n_medium*)
Return the T-matrix of a particle.

Parameters

- **vacuum_wavelength** (*float*) –
- **n_medium** (*float or complex*) – Refractive index of surrounding medium
- **particle** (smuthi.particles.Particle) – Particle object

Returns T-matrix as ndarray

5.2 The smuthi.fields package

5.2.1 fields

This subpackage contains functionality that has to do with the representation of electromagnetic fields in spherical or plane vector wave functions. The __init__ module contains some helper functions (e.g. with respect to SVWF indexing) and is the place to store default coordinate arrays for Sommerfeld integrals and field expansions.

smuthi.fields.**angular_arrays**(*angular_resolution=0.008726646259971648*)
Return azimuthal and polar angular arrays with a certain angular resolution

smuthi.fields.**angular_frequency**(*vacuum_wavelength*)
Angular frequency $\omega = 2\pi c/\lambda$

Parameters **vacuum_wavelength** (*float*) – Vacuum wavelength in length unit

Returns Angular frequency in the units of c=1 (time units=length units). This is at the same time the vacuum wavenumber.

smuthi.fields.**blocksize**
Number of coefficients in outgoing or regular spherical wave expansion for a single particle.

Parameters

- **l_max** (*int*) – Maximal multipole degree
- **m_max** (*int*) – Maximal multipole order

Returns Number of indices for one particle, which is the maximal index plus 1.

smuthi.fields.**branchpoint_correction**(*layer_refractive_indices*, *n_effective_array*, *neff_minimal_branchpoint_distance*)
Check if an array of complex effective refractive index values (e.g. for Sommerfeld integration) contains possible branchpoint singularities and if so, replace them by nearby non-singular locations.

Parameters

- **layer_refractive_indices** (*list or array*) – Complex refractive indices of planarly layered medium
- **n_effective_array** (*1d numpy.array*) – Complex effective refractive indexc values that are to be checked for branchpoint collision This array is changed during the function evaluation!
- **neff_minimal_branchpoint_distance** (*float*) – Minimal distance that contour points shall have from branchpoint singularities

Returns corrected n_effective_array

smuthi.fields.**create_k_parallel_array**(*vacuum_wavelength*, *neff_waypoints*, *neff_resolution*)
Construct an array of complex in-plane wavenumbers (i.e., the radial component of the cylindrical coordinates of the wave-vector). This is used for the plane wave expansion of fields and for Sommerfeld integrals. Complex contours are used to improve numerical stability (see section 3.10.2.1 of [Egel 2018 dissertation]).

Parameters

- **vacuum_wavelength** (*float*) – Vacuum wavelength λ (length)
- **neff_waypoints** (*list or ndarray*) – Corner points through which the contour runs This quantity is dimensionless (effective refractive index, will be multiplied by vacuum wavenumber)

- **neff_resolution** (*float*) – Resolution of contour, again in terms of effective refractive index

Returns Array κ_i of in-plane wavenumbers (inverse length)

smuthi.fields.**create_neff_array**(*neff_waypoints*, *neff_resolution*)

Construct an array of complex effective refractive index values. The effective refractive index is a dimensionless quantity that will be multiplied by vacuum wavenumber to yield the in-plane component of a wave vector. This is used for the plane wave expansion of fields and for Sommerfeld integrals. Complex contours are used to improve numerical stability (see section 3.10.2.1 of [Egel 2018 dissertation]).

Parameters

- **neff_waypoints** (*list or ndarray*) – Corner points through which the contour runs
- **neff_resolution** (*float*) – Resolution of contour (i.e., distance between adjacent elements)

Returns Array of complex effective refractive index values

smuthi.fields.**default_Sommerfeld_k_parallel_array = None**

Default n_effective array for the initial field (beams, dipoles) - needs to be set, e.g. at beginning of simulation

smuthi.fields.**default_polar_angles = None**

Default n_effective array for Sommerfeld integrals - needs to be set, e.g. at beginning of simulation

smuthi.fields.**k_z**(*k_parallel=None*, *n_effective=None*, *k=None*, *omega=None*, *vacuum_wavelength=None*, *refractive_index=None*)

z-component $k_z = \sqrt{k^2 - \kappa^2}$ of the wavevector. The branch cut is defined such that the imaginary part is not negative, compare section 2.3.1 of [Egel 2018 dissertation]. Not all of the arguments need to be specified.

Parameters

- **k_parallel** (*numpy ndarray*) – In-plane wavenumber κ (inverse length)
- **n_effective** (*numpy ndarray*) – Effective refractive index n_{eff}
- **k** (*float*) – Wavenumber (inverse length)
- **omega** (*float*) – Angular frequency ω or vacuum wavenumber (inverse length, c=1)
- **vacuum_wavelength** (*float*) – Vacuum wavelength λ (length)
- **refractive_index** (*complex*) – Refractive index n_i of material

Returns z-component k_z of wavenumber with non-negative imaginary part (inverse length)

smuthi.fields.**multi_to_single_index**

Unique single index for the totality of indices characterizing a svwf expansion coefficient.

The mapping follows the scheme:

single index	spherical wave expansion indices		
n	τ	l	m
1	1	1	-1
2	1	1	0
3	1	1	1
4	1	2	-2
5	1	2	-1
6	1	2	0
...	...	...	...
...	1	l_max	m_max
...	2	1	-1
...	...	...	...

Parameters

- **tau** (*int*) – Polarization index τ (0=spherical TE, 1=spherical TM)
- **l** (*int*) – Degree l (1, ..., lmax)
- **m** (*int*) – Order m (-min(l,mmax),...,min(l,mmax))
- **l_max** (*int*) – Maximal multipole degree
- **m_max** (*int*) – Maximal multipole order

Returns single index (int) subsuming (τ, l, m)

smuthi.fields.**reasonable_Sommerfeld_kpar_contour**(*vacuum_wavelength*, *neff_waypoints=None*, *layer_refractive_indices=None*, *neff_imag=0.01*, *neff_max=None*, *neff_max_offset=1*, *neff_resolution=0.01*, *neff_minimal_branchpoint_distance=None*)

Return a reasonable k_parallel array that is suitable as a Sommerfeld integral contour. Use this function if you don't want to care for numerical details of your simulation.

Parameters

- **vacuum_wavelength** (*float*) – Vacuum wavelength λ (length)
- **neff_waypoints** (*list or ndarray*) – Corner points through which the contour runs This quantity is dimensionless (effective refractive index, will be multiplied by vacuum wavenumber) If not provided, reasonable waypoints are estimated.
- **layer_refractive_indices** (*list*) – Complex refractive indices of planarly layered medium Only needed when no neff_waypoints are provided
- **neff_imag** (*float*) – Extent of the contour into the negative imaginary direction (in terms of effective refractive index, n_eff=kappa/omega). Only needed when no neff_waypoints are provided
- **neff_max** (*float*) – Truncation value of contour (in terms of effective refractive index). Only needed when no neff_waypoints are provided
- **neff_max_offset** (*float*) – Use the last estimated singularity location plus this value (in terms of effective refractive index). Default=1 Only needed when no neff_waypoints are provided and if no value for *neff_max* is specified.

- **neff_resolution** (*float*) – Resolution of contour, again in terms of effective refractive index
- **neff_minimal_branchpoint_distance** (*float*) – Minimal distance that contour points shall have from branchpoint singularities (in terms of effective refractive index). This is only relevant if not deflected into imaginary. Default: One fifth of neff_resolution

Returns Array κ_i of in-plane wavenumbers (inverse length)

smuthi.fields.**reasonable_Sommerfeld_neff_contour**(*neff_waypoints=None*, *layer_refractive_indices=None*, *neff_imag=0.01*, *neff_max=None*, *neff_max_offset=1*, *neff_resolution=0.01*, *neff_minimal_branchpoint_distance=None*)

Return a reasonable n_effective array that is suitable for the construction of a Sommerfeld k_parallel integral contour. Use this function if you don't want to care for numerical details of your simulation.

Parameters

- **neff_waypoints** (*list or ndarray*) – Corner points through which the contour runs This quantity is dimensionless (effective refractive index, will be multiplied by vacuum wavenumber) If not provided, reasonable waypoints are estimated.
- **layer_refractive_indices** (*list*) – Complex refractive indices of planarly layered medium Only needed when no neff_waypoints are provided
- **neff_imag** (*float*) – Extent of the contour into the negative imaginary direction (in terms of effective refractive index, n_eff=kappa/omega). Only needed when no neff_waypoints are provided
- **neff_max** (*float*) – Truncation value of contour (in terms of effective refractive index). Only needed when no neff_waypoints are provided
- **neff_max_offset** (*float*) – Use the last estimated singularity location plus this value (in terms of effective refractive index). Default=1 Only needed when no neff_waypoints are provided and if no value for *neff_max* is specified.
- **neff_resolution** (*float*) – Resolution of contour, again in terms of effective refractive index
- **neff_minimal_branchpoint_distance** (*float*) – Minimal distance that contour points shall have from branchpoint singularities (in terms of effective refractive index). This is only relevant if not deflected into imaginary. Default: One fifth of neff_resolution

Returns Array of complex effective refractive index values

smuthi.fields.**reasonable_neff_waypoints**(*layer_refractive_indices=None*, *neff_imag=0.01*, *neff_max=None*, *neff_max_offset=1*)

Construct a reasonable list of waypoints for a k_parallel array of plane wave expansions. The waypoints mark a contour through the complex plane such that possible waveguide mode and branchpoint singularity locations are avoided (see section 3.10.2.1 of [Egel 2018 dissertation]).

Parameters

- **layer_refractive_indices** (*list or array*) – Complex refractive indices of the plane layer system
- **neff_imag** (*float*) – Extent of the contour into the negative imaginary direction (in terms of effective refractive index, n_eff=kappa/omega).
- **neff_max** (*float*) – Truncation value of contour (in terms of effective refractive index).

- **neff_max_offset** (*float*) – If no value for *neff_max* is specified, use the last estimated singularity location plus this value (in terms of effective refractive index). Default=1

Returns List of complex waypoint values.

5.2.2 fields.expansions

Classes to manage the expansion of the electric field in plane wave and spherical wave basis sets.

class smuthi.fields.expansions.**FieldExpansion**

Base class for field expansions.

diverging(*x*, *y*, *z*)

Test if points are in domain where expansion could diverge. Virtual method to be overwritten in child classes.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points

Returns numpy.ndarray of bool datatype indicating if points are inside divergence domain.

electric_field(*x*, *y*, *z*)

Evaluate electric field. Virtual method to be overwritten in child classes.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points

Returns Tuple of (E_x, E_y, E_z) numpy.ndarray objects with the Cartesian coordinates of complex electric field.

magnetic_field(*x*, *y*, *z*, *vacuum_wavelength*)

Evaluate magnetic field. Virtual method to be overwritten in child classes.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points
- **vacuum_wavelength** (*float*) – Vacuum wavelength in length units

Returns Tuple of (H_x, H_y, H_z) numpy.ndarray objects with the Cartesian coordinates of complex magnetic field.

valid(*x*, *y*, *z*)

Test if points are in definition range of the expansion. Abstract method to be overwritten in child classes.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points

Returns numpy.ndarray of bool datatype indicating if points are inside definition domain.

class smuthi.fields.expansions.**PiecewiseFieldExpansion**

Manage a field that is expanded in different ways for different domains, i.e., an expansion of the kind

$$\mathbf{E}(\mathbf{r}) = \sum_i \mathbf{E}_i(\mathbf{r}),$$

where

$$\mathbf{E}_i(\mathbf{r}) = \begin{cases} \tilde{\mathbf{E}}_i(\mathbf{r}) & \text{if } \mathbf{r} \in D_i \\ 0 & \text{else} \end{cases}$$

and $\tilde{\mathbf{E}_\mathbf{i}}(\mathbf{r})$ is either a plane wave expansion or a spherical wave expansion, and D_i is its domain of validity.

compatible(*other*)

Returns always true, because any field expansion can be added to a piecewise field expansion.

diverging(*x*, *y*, *z*)

Test if points are in domain where expansion could diverge.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points

Returns numpy.ndarray of bool datatype indicating if points are inside divergence domain.

electric_field(*x*, *y*, *z*)

Evaluate electric field.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points

Returns Tuple of (E_x, E_y, E_z) numpy.ndarray objects with the Cartesian coordinates of complex electric field.

magnetic_field(*x*, *y*, *z*, *vacuum_wavelength*)

Evaluate magnetic field. Virtual method to be overwritten in child classes.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points
- **vacuum_wavelength** (*float*) – Vacuum wavelength in length units

Returns Tuple of (H_x, H_y, H_z) numpy.ndarray objects with the Cartesian coordinates of complex magnetic field.

valid(*x*, *y*, *z*)

Test if points are in definition range of the expansion.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points

- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points

Returns numpy.ndarray of bool datatype indicating if points are inside definition domain.

class smuthi.fields.expansions.**PlaneWaveExpansion**(*k*, *k_parallel*, *azimuthal_angles*, *kind=None*, *reference_point=None*, *lower_z=-inf*, *upper_z=inf*)

A class to manage plane wave expansions of the form

$$\mathbf{E}(\mathbf{r}) = \sum_{j=1}^{2} \iint \mathrm{d}^2\mathbf{k}_\parallel \, g_j(\kappa,\alpha) \mathbf{\Phi}_j^{\pm}(\kappa,\alpha;\mathbf{r}-\mathbf{r}_i)$$

for $\mathbf{r}$ located in a layer defined by $z \in [z_{min}, z_{max}]$ and $\mathrm{d}^2\mathbf{k}_\parallel = \kappa \, \mathrm{d}\alpha \, \mathrm{d}\kappa$.

The double integral runs over $\alpha \in [0, 2\pi]$ and $\kappa \in [0, \kappa_{\mathrm{max}}]$. Further, $\mathbf{\Phi}_j^{\pm}$ are the PVWFs, see `plane_vector_wave_function()`.

Internally, the expansion coefficients $g_{ij}^{\pm}(\kappa,\alpha)$ are stored as a 3-dimensional numpy ndarray.

If the attributes k_parallel and azimuthal_angles have only a single entry, a discrete distribution is assumed:

$$g_j^{\pm}(\kappa,\alpha) \sim \delta^2(\mathbf{k}_\parallel - \mathbf{k}_{\parallel,0})$$

Parameters

- **k** (*float*) – wavenumber in layer where expansion is valid
- **k_parallel** (*numpy ndarray*) – array of in-plane wavenumbers (can be float or complex)
- **azimuthal_angles** (*numpy ndarray*) – α, from 0 to 2π
- **kind** (*str*) – 'upgoing' for g^+ and 'downgoing' for g^- type expansions
- **reference_point** (*list or tuple*) – [x, y, z]-coordinates of point relative to which the plane waves are defined.
- **lower_z** (*float*) – the expansion is valid on and above that z-coordinate
- **upper_z** (*float*) – the expansion is valid below that z-coordinate

coefficients

coefficients[j, k, l] contains

Type numpy ndarray

:math:`g^pm_{j}`

Type kappa_{k}, alpha_{l}

class OptimizationMethodsForLinux

An enumeration.

evaluate_r_times_eikr

Attention! Sometimes this function can decrease speed on 1 core mode. Here foo_x, foo_y, foo_z are supposed to be 2dim arrays with [None, :, :]. This function can replace snippet

exp_j = np.exp(1j * exp_feed) foo_x_eikr = foo_x * exp_j foo_y_eikr = foo_y * exp_j foo_z_eikr = foo_z * exp_j

by

foo_x_eikr, foo_y_eikr, foo_z_eikr = numba_multiple_on_exp(foo_x, foo_y, foo_z, kr).

numba_3tensordots_1dim_times_2dim
This function can replace snippet

'foo = np.tensordot(x_float_1dim, x_complex_2dim, axes=0)

foo += np.tensordot(y_float_1dim, y_complex_2dim, axes=0)

foo += np.tensordot(z_float_1dim, z_complex_2dim, axes=0)'

by
'foo = get_3_tensordots(x_float_1dim, y_float_1dim, z_float_1dim, x_complex_2dim, y_complex_2dim, z_complex_2dim)'

numba_trapz_3dim_array
This function can replace snippet

'foo = np.trapz(y, x)'

by

'foo = numba_trapz_3dim_array(y, x)'

class OptimizationMethodsFor_Not_Linux
An enumeration.

evaluate_r_times_eikr

numba_3tensordots_1dim_times_2dim

numba_trapz_3dim_array

class RawSliceOfField(*axis*, *chunks*, *values*)

azimuthal_angle_grid()
Meshgrid of azimuthal_angles with respect to n_effective

compatible(*other*)
Check if two plane wave expansions are compatible in the sense that they can be added coefficient-wise

Parameters **other** (FieldExpansion) – expansion object to add to this object

Returns bool (true if compatible, false else)

diverging(*x*, *y*, *z*)
Test if points are in domain where expansion could diverge.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points

Returns numpy.ndarray of bool datatype indicating if points are inside divergence domain.

electric_field(*x*, *y*, *z*, *max_chunksize=50*, *cpu_precision='single precision'*)
Evaluate electric field.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points

- **max_chunksize** (*int*) – max number of field points that are simultaneously evaluated when running in CPU mode. In Windows/MacOS max_chunksize = chunksize, in Linux it can be decreased considering available CPU cores.
- **cpu_precision** (*string*) – set 'double precision' to use float64 and complex128 types instead of float32 and complex64

Returns Tuple of (E_x, E_y, E_z) numpy.ndarray objects with the Cartesian coordinates of complex electric field.

k_parallel_grid()
Meshgrid of n_effective with respect to azimuthal_angles

k_z()

k_z_grid()

magnetic_field(*x*, *y*, *z*, *vacuum_wavelength*, *max_chunksize=50*, *cpu_precision='single precision'*)
Evaluate magnetic field.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points
- **vacuum_wavelength** (*float*) – Vacuum wavelength in length units
- **chunksize** (*int*) – number of field points that are simultaneously evaluated when running in CPU mode

Returns Tuple of (H_x, H_y, H_z) numpy.ndarray objects with the Cartesian coordinates of complex magnetic field.

set_reference_point(*new_reference_point*)
Set a new reference point. This implies also a phase factor on the coefficients.

Parameters **new_reference_point** (*list or tuple*) – [x, y, z]-coordinates of point relative to which the plane waves are defined.

valid(*x*, *y*, *z*)
Test if points are in definition range of the expansion.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points

Returns numpy.ndarray of bool datatype indicating if points are inside definition domain.

class smuthi.fields.expansions.**SphericalWaveExpansion**(*k*, *l_max*, *m_max=None*, *kind=None*, *reference_point=None*, *lower_z=-inf*, *upper_z=inf*, *inner_r=0*, *outer_r=inf*)
A class to manage spherical wave expansions of the form

$$\mathbf{E}(\mathbf{r}) = \sum_{\tau=1}^{2}\sum_{l=1}^{\infty}\sum_{m=-l}^{l} a_{\tau lm}\mathbf{\Psi}^{(\nu)}_{\tau lm}(\mathbf{r}-\mathbf{r}_i)$$

for **r** located in a layer defined by $z \in [z_{min}, z_{max}]$ and where $\mathbf{\Psi}^{(\nu)}_{\tau lm}$ are the SVWFs, see `smuthi.vector_wave_functions.spherical_vector_wave_function()`.

Internally, the expansion coefficients $a_{\tau lm}$ are stored as a 1-dimensional array running over a multi index n subsumming over the SVWF indices (τ, l, m). The mapping from the SVWF indices to the multi index is organized by the function `multi_to_single_index()`.

Parameters

- **k** (*float*) – wavenumber in layer where expansion is valid
- **l_max** (*int*) – maximal multipole degree $l_{\max} \geq 1$ where to truncate the expansion.
- **m_max** (*int*) – maximal multipole order $0 \leq m_{\max} \leq l_{\max}$ where to truncate the expansion.
- **kind** (*str*) – 'regular' for $\nu = 1$ or 'outgoing' for $\nu = 3$
- **reference_point** (*list or tuple*) – [x, y, z]-coordinates of point relative to which the spherical waves are considered (e.g., particle center).
- **lower_z** (*float*) – the expansion is valid on and above that z-coordinate
- **upper_z** (*float*) – the expansion is valid below that z-coordinate
- **inner_r** (*float*) – radius inside which the expansion diverges (e.g. circumscribing sphere of particle)
- **outer_r** (*float*) – radius outside which the expansion diverges

coefficients
expansion coefficients $a_{\tau lm}$ ordered by multi index n

Type numpy ndarray

coefficients_tlm(*tau*, *l*, *m*)
SWE coefficient for given (tau, l, m)

Parameters

- **tau** (*int*) – SVWF polarization (0 for spherical TE, 1 for spherical TM)
- **l** (*int*) – SVWF degree
- **m** (*int*) – SVWF order

Returns SWE coefficient

compatible(*other*)
Check if two spherical wave expansions are compatible in the sense that they can be added coefficient-wise

Parameters **other** (FieldExpansion) – expansion object to add to this object

Returns bool (true if compatible, false else)

diverging(*x*, *y*, *z*)
Test if points are in domain where expansion could diverge.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points

Returns numpy.ndarray of bool datatype indicating if points are inside divergence domain.

electric_field(*x*, *y*, *z*)
Evaluate electric field.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points

Returns Tuple of (E_x, E_y, E_z) numpy.ndarray objects with the Cartesian coordinates of complex electric field.

magnetic_field(*x*, *y*, *z*, *vacuum_wavelength*)
Evaluate magnetic field.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points
- **vacuum_wavelength** (*float*) – Vacuum wavelength in length units

Returns Tuple of (H_x, H_y, H_z) numpy.ndarray objects with the Cartesian coordinates of complex electric field.

valid(*x*, *y*, *z*)
Test if points are in definition range of the expansion.

Parameters

- **x** (*numpy.ndarray*) – x-coordinates of query points
- **y** (*numpy.ndarray*) – y-coordinates of query points
- **z** (*numpy.ndarray*) – z-coordinates of query points

Returns numpy.ndarray of bool datatype indicating if points are inside definition domain.

5.2.3 fields.expansions_cuda

This module contains CUDA source code for the evaluation of the electric field from a VWF expansion.

5.2.4 fields.transformatinos

Functions for the transformation of plane and spherical vector wave functions as well as of plane and spherical wave fex.

smuthi.fields.transformations.**block_rotation_matrix_D_svwf**(*l_max*, *m_max*, *alpha*, *beta*, *gamma*, *wdsympy=False*)
Rotation matrix for the rotation of SVWFs between the labratory coordinate system (L) and a rotated coordinate system (R)

Parameters

- **l_max** (*int*) – Maximal multipole degree
- **m_max** (*int*) – Maximal multipole order

- **alpha** (*float*) – First Euler angle, rotation around z-axis, in rad
- **beta** (*float*) – Second Euler angle, rotation around y'-axis in rad
- **gamma** (*float*) – Third Euler angle, rotation around z"-axis in rad
- **wdsympy** (*bool*) – If True, Wigner-d-functions come from the sympy toolbox

Returns rotation matrix of dimension [blocksize, blocksize]

smuthi.fields.transformations.**pwe_to_swe_conversion**(*pwe*, *l_max*, *m_max*, *reference_point*)

Convert plane wave expansion object to a spherical wave expansion object.

Parameters

- **pwe** (PlaneWaveExpansion) – Plane wave expansion to be converted
- **l_max** (*int*) – Maximal multipole degree of spherical wave expansion
- **m_max** (*int*) – Maximal multipole order of spherical wave expansion
- **reference_point** (*list*) – Coordinates of reference point in the format [x, y, z]

Returns SphericalWaveExpansion object.

smuthi.fields.transformations.**swe_to_pwe_conversion**(*swe*, *k_parallel*, *azimuthal_angles*, *layer_system=None*, *layer_number=None*, *layer_system_mediated=False*, *only_l=None*, *only_m=None*, *only_pol=None*, *only_tau=None*)

Convert SphericalWaveExpansion object to a PlaneWaveExpansion object.

Parameters

- **swe** (SphericalWaveExpansion) – Spherical wave expansion to be converted
- **k_parallel** (*numpy array or str*) – In-plane wavenumbers for the pwe object.
- **azimuthal_angles** (*numpy array or str*) – Azimuthal angles for the pwe object
- **layer_system** (smuthi.layers.LayerSystem) – Stratified medium in which the origin of the SWE is located
- **layer_number** (*int*) – Layer number in which the PWE should be valid.
- **layer_system_mediated** (*bool*) – If True, the PWE refers to the layer system response of the SWE, otherwise it is the direct transform.
- **only_pol** (*int*) – if set to 0 or 1, only this plane wave polarization (0=TE, 1=TM) is considered
- **only_tau** (*int*) – if set to 0 or 1, only this spherical vector wave polarization (0 — magnetic, 1 — electric) is considered
- **only_l** (*int*) – if set to positive number, only this multipole degree is considered
- **only_m** (*int*) – if set to non-negative number, only this multipole order is considered

Returns Tuple of two PlaneWaveExpansion objects, first upgoing, second downgoing.

smuthi.fields.transformations.**transformation_coefficients_vwf**(*tau*, *l*, *m*, *pol*, *kp=None*, *kz=None*, *pilm_list=None*, *taulm_list=None*, *dagger=False*)

Transformation coefficients B to expand SVWF in PVWF and vice versa:

$$B_{\tau lm,j}(x) = -\frac{1}{\mathrm{i}^{l+1}}\frac{1}{\sqrt{2l(l+1)}}(\mathrm{i}\delta_{j1}+\delta_{j2})(\delta_{\tau j}\tau_l^{|m|}(x)+(1-\delta_{\tau j}m\pi_l^{|m|}(x))$$

For the definition of the τ_l^m and π_l^m functions, see A. Doicu, T. Wriedt, and Y. A. Eremin: "Light Scattering by Systems of Particles", Springer-Verlag, 2006

Compare also section 2.3.3 of [Egel 2018 diss].

Parameters

- **tau** (*int*) – SVWF polarization, 0 for spherical TE, 1 for spherical TM
- **l** (*int*) – l=1,... SVWF multipole degree
- **m** (*int*) – m=-l,...,l SVWF multipole order
- **pol** (*int*) – PVWF polarization, 0 for TE, 1 for TM
- **kp** (*numpy array*) – PVWF in-plane wavenumbers
- **kz** (*numpy array*) – complex numpy-array: PVWF out-of-plane wavenumbers
- **pilm_list** (*list*) – 2D list numpy-arrays: alternatively to kp and kz, pilm and taulm as generated with legendre_normalized can directly be handed
- **taulm_list** (*list*) – 2D list numpy-arrays: alternatively to kp and kz, pilm and taulm as generated with legendre_normalized can directly be handed
- **dagger** (*bool*) – switch on when expanding PVWF in SVWF and off when expanding SVWF in PVWF

Returns Transformation coefficient as array (size like kp).

smuthi.fields.transformations.**translation_block**(*vacuum_wavelength*, *receiving_particle*, *emitting_particle*, *layer_system*, *kind*)

Direct particle translation matrix W for two particles that do not have intersecting circumscribing spheres.

This routine is explicit.

To reduce computation time, this routine relies on two internal accelerations. First, in most cases the number of unique maximum multipole indicies, (τ, l_{max}, m_{max}), is much less than the number of unique particles. Therefore, all calculations that depend only on multipole indicies are stored in an intermediate hash table. Second, Cython acceleration is used by default to leverage fast looping. If the Cython files are not supported, this routine will fall back on equivalent Python looping.

Cython acceleration can be between 10-1,000x faster compared to the Python equivalent. Speed variability depends on the number of unique multipoles indicies, the size of the largest multipole order, and if particles share the same z coordinate.

Parameters

- **vacuum_wavelength** (*float*) – Vacuum wavelength λ (length unit)
- **receiving_particle** (smuthi.particles.Particle) – Particle that receives the scattered field

- **emitting_particle** (smuthi.particles.Particle) – Particle that emits the scattered field
- **layer_system** (smuthi.layers.LayerSystem) – Stratified medium in which the coupling takes place

Returns Direct coupling matrix block as numpy array.

smuthi.fields.transformations.**translation_coefficients_svwf**(*tau1*, *l1*, *m1*, *tau2*, *l2*, *m2*, *k*, *d*, *sph_hankel=None*, *legendre=None*, *exp_immphi=None*)

Coefficients of the translation operator for the expansion of an outgoing spherical wave in terms of regular spherical waves with respect to a different origin:

$$\mathbf{\Psi}^{(3)}_{\tau lm}(\mathbf{r}+\mathbf{d} = \sum_{\tau'}\sum_{l'}\sum_{m'} A_{\tau lm,\tau' l' m'}(\mathbf{d})\mathbf{\Psi}^{(1)}_{\tau' l' m'}(\mathbf{r})$$

for $|\mathbf{r}| < |\mathbf{d}|$.

See also section 2.3.3 and appendix B of [Egel 2018 diss].

Parameters

- **tau1** (*int*) – tau1=0,1: Original wave's spherical polarization
- **l1** (*int*) – l=1,...: Original wave's SVWF multipole degree
- **m1** (*int*) – m=-l,...,l: Original wave's SVWF multipole order
- **tau2** (*int*) – tau2=0,1: Partial wave's spherical polarization
- **l2** (*int*) – l=1,...: Partial wave's SVWF multipole degree
- **m2** (*int*) – m=-l,...,l: Partial wave's SVWF multipole order
- **k** (*float or complex*) – wavenumber (inverse length unit)
- **d** (*list*) – translation vectors in format [dx, dy, dz] (length unit) dx, dy, dz can be scalars or ndarrays
- **sph_hankel** (*list*) – Optional. sph_hankel[i] contains the spherical hankel funciton of degree i, evaluated at k*d where d is the norm of the distance vector(s)
- **legendre** (*list*) – Optional. legendre[l][m] contains the legendre function of order l and degree m, evaluated at cos(theta) where theta is the polar angle(s) of the distance vector(s)

Returns translation coefficient A (complex)

smuthi.fields.transformations.**translation_coefficients_svwf_out_to_out**(*tau1*, *l1*, *m1*, *tau2*, *l2*, *m2*, *k*, *d*, *sph_bessel=None*, *legendre=None*, *exp_immphi=None*)

Coefficients of the translation operator for the expansion of an outgoing spherical wave in terms of outgoing

spherical waves with respect to a different origin:

$$\mathbf{\Psi}^{(3)}_{\tau lm}(\mathbf{r}+\mathbf{d} = \sum_{\tau'}\sum_{l'}\sum_{m'} A_{\tau lm,\tau' l' m'}(\mathbf{d})\mathbf{\Psi}^{(3)}_{\tau' l' m'}(\mathbf{r})$$

for $|\mathbf{r}| > |\mathbf{d}|$.

Parameters

- **tau1** (*int*) – tau1=0,1: Original wave's spherical polarization
- **l1** (*int*) – l=1,...: Original wave's SVWF multipole degree
- **m1** (*int*) – m=-l,...,l: Original wave's SVWF multipole order
- **tau2** (*int*) – tau2=0,1: Partial wave's spherical polarization
- **l2** (*int*) – l=1,...: Partial wave's SVWF multipole degree
- **m2** (*int*) – m=-l,...,l: Partial wave's SVWF multipole order
- **k** (*float or complex*) – wavenumber (inverse length unit)
- **d** (*list*) – translation vectors in format [dx, dy, dz] (length unit) dx, dy, dz can be scalars or ndarrays
- **sph_bessel** (*list*) – Optional. sph_bessel[i] contains the spherical Bessel funciton of degree i, evaluated at k*d where d is the norm of the distance vector(s)
- **legendre** (*list*) – Optional. legendre[l][m] contains the legendre function of order l and degree m, evaluated at cos(theta) where theta is the polar angle(s) of the distance vector(s)

Returns translation coefficient A (complex)

5.2.5 fields.vector_wave_functions

This module contains the vector wave functions and their transformations.

smuthi.fields.vector_wave_functions.**plane_vector_wave_function**(*x*, *y*, *z*, *kp*, *alpha*, *kz*, *pol*)

Electric field components of plane wave (PVWF), see section 2.3.1 of [Egel 2018 diss].

$$\mathbf{\Phi}_j = \exp(\mathrm{i}\mathbf{k}\cdot\mathbf{r})\hat{\mathbf{e}}_j$$

with $\hat{\mathbf{e}}_0$ denoting the unit vector in azimuthal direction ('TE' or 's' polarization), and $\hat{\mathbf{e}}_1$ denoting the unit vector in polar direction ('TM' or 'p' polarization).

The input arrays should have one of the following dimensions:

- x,y,z: (N x 1) matrix
- kp,alpha,kz: (1 x M) matrix
- Ex, Ey, Ez: (M x N) matrix

or

- x,y,z: (M x N) matrix
- kp,alpha,kz: scalar
- Ex, Ey, Ez: (M x N) matrix

Parameters

- **x** (*numpy.ndarray*) – x-coordinate of position where to test the field (length unit)
- **y** (*numpy.ndarray*) – y-coordinate of position where to test the field
- **z** (*numpy.ndarray*) – z-coordinate of position where to test the field
- **kp** (*numpy.ndarray*) – parallel component of k-vector (inverse length unit)
- **alpha** (*numpy.ndarray*) – azimthal angle of k-vector (rad)
- **kz** (*numpy.ndarray*) – z-component of k-vector (inverse length unit)
- **pol** (*int*) – Polarization (0=TE, 1=TM)

Returns

- x-coordinate of PVWF electric field (numpy.ndarray)
- y-coordinate of PVWF electric field (numpy.ndarray)
- z-coordinate of PVWF electric field (numpy.ndarray)

smuthi.fields.vector_wave_functions.**spherical_vector_wave_function**(*x*, *y*, *z*, *k*, *nu*, *tau*, *l*, *m*)

Electric field components of spherical vector wave function (SVWF). The conventions are chosen according to A. Doicu, T. Wriedt, and Y. A. Eremin: "Light Scattering by Systems of Particles", Springer-Verlag, 2006 See also section 2.3.2 of [Egel 2018 diss].

Parameters

- **x** (*numpy.ndarray*) – x-coordinate of position where to test the field (length unit)
- **y** (*numpy.ndarray*) – y-coordinate of position where to test the field
- **z** (*numpy.ndarray*) – z-coordinate of position where to test the field
- **k** (*float or complex*) – wavenumber (inverse length unit)
- **nu** (*int*) – 1 for regular waves, 3 for outgoing waves
- **tau** (*int*) – spherical polarization, 0 for spherical TE and 1 for spherical TM
- **l** (*int*) – l=1,... multipole degree (polar quantum number)
- **m** (*int*) – m=-l,...,l multipole order (azimuthal quantum number)

Returns

- x-coordinate of SVWF electric field (numpy.ndarray)
- y-coordinate of SVWF electric field (numpy.ndarray)
- z-coordinate of SVWF electric field (numpy.ndarray)

5.3 The smuthi.linearsystem package

5.3.1 linearsystem

This package contains functionality that is related to the assembly or solution of the system of linear equations that yield the solution of the scattering problem.

5.3.2 linearsystem.linear_system

This package contains classes and functions to represent the system of linear equations that needs to be solved in order to solve the scattering problem, see section 3.7 of [Egel 2018 dissertation].

Symbolically, the linear system can be written like

$$(1 - TW)b = Ta,$$

where T is the transition matrices of the particles, W is the particle coupling matrix, b are the (unknown) coefficients of the scattered field in terms of an outgoing spherical wave expansion and a are the coefficients of the initial field in terms of a regular spherical wave expansion.

class smuthi.linearsystem.linear_system.**CouplingMatrixExplicit**(*vacuum_wavelength*, *particle_list*, *layer_system*, *k_parallel='default'*, *use_pvwf_coupling=False*, *pvwf_coupling_k_parallel=None*)

Class for an explicit representation of the coupling matrix. Recommended for small particle numbers.

Parameters

- **vacuum_wavelength** (*float*) – Vacuum wavelength in length units
- **particle_list** (*list*) – List of smuthi.particles.Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – Stratified medium
- **k_parallell** (*numpy.ndarray or str*) – In-plane wavenumber. If 'default', use smuthi.fields.default_Sommerfeld_k_parallel_array

class smuthi.linearsystem.linear_system.**CouplingMatrixPeriodicGridNumba**(*initial_field*, *particle_list*, *layer_system*, *periodicity*, *ewald_sum_separation_par*, *num_threads='default'*)

Class for an explicit representation of the coupling matrix of periodic particle arrangements.

Computation supports Numba.

Parameters

- **initial_field** (smuthi.initial_field.PlaneWave) – initial plane wave object
- **particle_list** (*list*) – list of smuthi.particles.Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium
- **periodicity** (*tuple*) – (a1, a2) lattice vector 1 and 2 in carthesian coordinates
- **ewald_sum_separation_parameter** (*float*) – Ewald sum separation parameter

- **num_threads** (*int or str*) – if 'default' all available CPU cores are used if negative, all but num_threads are used

class smuthi.linearsystem.linear_system.**CouplingMatrixRadialLookup**(*vacuum_wavelength*, *particle_list*, *layer_system*, *k_parallel='default'*, *resolution=None*)

Base class for radial lookup based coupling matrix either on CPU or on GPU (CUDA).

Parameters

- **vacuum_wavelength** (*float*) – vacuum wavelength in length units
- **particle_list** (*list*) – list of sumthi.particles.Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium
- **k_parallel** (*numpy.ndarray or str*) – in-plane wavenumber. If 'default', use smuthi.fields.default_Sommerfeld_k_parallel_array
- **resolution** (*float or None*) – spatial resolution of the lookup in the radial direction

class smuthi.linearsystem.linear_system.**CouplingMatrixRadialLookupCPU**(*vacuum_wavelength*, *particle_list*, *layer_system*, *k_parallel='default'*, *resolution=None*, *interpolator_kind='linear'*)

Class for radial lookup based coupling matrix running on CPU. This is used when no suitable GPU device is detected or when PyCuda is not installed.

Parameters

- **vacuum_wavelength** (*float*) – vacuum wavelength in length units
- **particle_list** (*list*) – list of sumthi.particles.Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium
- **k_parallel** (*numpy.ndarray or str*) – in-plane wavenumber. If 'default', use smuthi.fields.default_Sommerfeld_k_parallel_array
- **resolution** (*float or None*) – spatial resolution of the lookup in the radial direction
- **kind** (*str*) – interpolation order, e.g. 'linear' or 'cubic'

class smuthi.linearsystem.linear_system.**CouplingMatrixRadialLookupCUDA**(*vacuum_wavelength*, *particle_list*, *layer_system*, *k_parallel='default'*, *resolution=None*, *cuda_blocksize=None*, *interpolator_kind='linear'*)

Radial lookup based coupling matrix either on GPU (CUDA).

Parameters

- **vacuum_wavelength** (*float*) – vacuum wavelength in length units
- **particle_list** (*list*) – list of sumthi.particles.Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium
- **k_parallel** (*numpy.ndarray or str*) – in-plane wavenumber. If 'default', use smuthi.fields.default_Sommerfeld_k_parallel_array
- **resolution** (*float or None*) – spatial resolution of the lookup in the radial direction
- **cuda_blocksize** (*int*) – threads per block when calling CUDA kernel

class smuthi.linearsystem.linear_system.**CouplingMatrixVolumeLookup**(*vacuum_wavelength*, *particle_list*, *layer_system*, *k_parallel='default'*, *resolution=None*)

Base class for 3D lookup based coupling matrix either on CPU or on GPU (CUDA).

Parameters

- **vacuum_wavelength** (*float*) – vacuum wavelength in length units
- **particle_list** (*list*) – list of sumthi.particles.Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium
- **k_parallel** (*numpy.ndarray or str*) – in-plane wavenumber. If 'default', use smuthi.fields.default_Sommerfeld_k_parallel_array
- **resolution** (*float or None*) – spatial resolution of the lookup in the radial direction

class smuthi.linearsystem.linear_system.**CouplingMatrixVolumeLookupCPU**(*vacuum_wavelength*, *particle_list*, *layer_system*, *k_parallel='default'*, *resolution=None*, *interpolator_kind='cubic'*)

Class for 3D lookup based coupling matrix running on CPU. This is used when no suitable GPU device is detected or when PyCuda is not installed.

Parameters

- **vacuum_wavelength** (*float*) – vacuum wavelength in length units
- **particle_list** (*list*) – list of sumthi.particles.Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium
- **k_parallel** (*numpy.ndarray or str*) – in-plane wavenumber. If 'default', use smuthi.fields.default_Sommerfeld_k_parallel_array
- **resolution** (*float or None*) – spatial resolution of the lookup in the radial direction
- **interpolator_kind** (*str*) – 'linear' or 'cubic' interpolation

class smuthi.linearsystem.linear_system.**CouplingMatrixVolumeLookupCUDA**(*vacuum_wavelength*, *particle_list*, *layer_system*, *k_parallel='default'*, *resolution=None*, *cuda_blocksize=None*, *interpolator_kind='linear'*)

Class for 3D lookup based coupling matrix running on GPU.

Parameters

- **vacuum_wavelength** (*float*) – vacuum wavelength in length units
- **particle_list** (*list*) – list of sumthi.particles.Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium
- **k_parallel** (*numpy.ndarray or str*) – in-plane wavenumber. If 'default', use smuthi.fields.default_Sommerfeld_k_parallel_array
- **resolution** (*float or None*) – spatial resolution of the lookup in the radial direction
- **cuda_blocksize** (*int*) – threads per block for cuda call
- **interpolator_kind** (*str*) – 'linear' (default) or 'cubic' interpolation

class smuthi.linearsystem.linear_system.**LinearSystem**(*particle_list*, *initial_field*, *layer_system*, *k_parallel='default'*, *solver_type='LU'*, *solver_tolerance=0.0001*, *store_coupling_matrix=True*, *coupling_matrix_lookup_resolution=None*, *interpolator_kind='cubic'*, *cuda_blocksize=None*, *periodicity=None*, *ewald_sum_separation_parameter='default'*, *number_of_threads_periodic='default'*, *use_pvwf_coupling=False*, *pvwf_coupling_k_parallel=None*)

Manage the assembly and solution of the linear system of equations.

Parameters

- **particle_list** (*list*) – List of smuthi.particles.Particle objects
- **initial_field** (smuthi.initial_field.InitialField) – Initial field object
- **layer_system** (smuthi.layers.LayerSystem) – Stratified medium
- **k_parallel** (*numpy.ndarray or str*) – in-plane wavenumber. If 'default', use smuthi.fields.default_Sommerfeld_k_parallel_array
- **solver_type** (*str*) – What solver to use? Options: 'LU' for LU factorization, 'gmres' for GMRES iterative solver
- **store_coupling_matrix** (*bool*) – If True (default), the coupling matrix is stored. Otherwise it is recomputed on the fly during each iteration of the solver.
- **coupling_matrix_lookup_resolution** (*float or None*) – If type float, compute particle coupling by interpolation of a lookup table with that spacial resolution. A smaller number implies higher accuracy and memory footprint. If None (default), don't use a lookup table but compute the coupling directly. This is more suitable for a small particle number.
- **interpolator_kind** (*str*) – interpolation order to be used, e.g. 'linear' or 'cubic'. This argument is ignored if coupling_matrix_lookup_resolution is None. In general, cubic interpolation is more accurate but a bit slower than linear.
- **periodicity** (*tuple*) – tuple (a1, a2) specifying two 3-dimensional lattice vectors in Carthesian coordinates with a1, a2 (numpy.ndarrays)
- **ewald_sum_separation_parameter** (*float*) – Used to separate the real and reciprocal lattice sums to evaluate particle coupling in periodic lattices.
- **number_of_threads_periodic** (*int or str*) – sets the number of threats used in a simulation with periodic particle arrangements if 'default', all available CPU cores are used if negative, all but number_of_threads_periodic are used
- **use_pvwf_coupling** (*bool*) – If set to True, plane wave coupling is used to calculate the direct. Currently only possible in combination with direct solver strategy.
- **pvwf_coupling_k_parallel** (*array*) – k-parallel for PVWF coupling

compute_coupling_matrix()

Initialize coupling matrix object.

compute_initial_field_coefficients()
Evaluate initial field coefficients.

compute_t_matrix()
Initialize T-matrix object.

prepare()

solve()
Compute scattered field coefficients and store them in the particles' spherical wave expansion objects.

class smuthi.linearsystem.linear_system.**MasterMatrix**(*t_matrix*, *coupling_matrix*)
Represent the master matrix $M = 1 - TW$ as a linear operator.

Parameters

- **t_matrix** (SystemMatrix) – System T-matrix
- **coupling_matrix** (SystemMatrix) – System coupling matrix

class smuthi.linearsystem.linear_system.**SystemMatrix**(*particle_list*)
A system matrix is an abstract linear operator that operates on a system coefficient vector, i.e. a vector $c = c^i_{\tau,l,m}$, where (τ, l, m) are the multipole indices and i indicates the particle number. In other words, if we have a spherical wave expansion for each particle, and write all the expansion coefficients of these expansions into one (long) array, what we get is a system vector.

index(*i*, *tau*, *l*, *m*)

Parameters

- **i** (*int*) – particle number
- **tau** (*int*) – spherical polarization index
- **l** (*int*) – multipole degree
- **m** (*int*) – multipole order

Returns Position in a system vector that corresponds to the (τ, l, m) coefficient of the i-th particle.

index_block(*i*)

Parameters **i** (*int*) – number of particle

Returns indices that correspond to the coefficients for that particle

class smuthi.linearsystem.linear_system.**TMatrix**(*particle_list*)
Collect the particle T-matrices in a global lienear operator.

Parameters **particle_list** (*list*) – List of smuthi.particles.Particle objects containing a t_matrix attribute.

right_hand_side()
The right hand side of the linear system is given by $\sum_{\tau lm} T^i_{\tau lm} a^i_{\tau lm}$

Returns right hand side as a complex numpy.ndarray

5.3.3 linearsystem.linear_system_cuda

This module contains CUDA source code for the evaluation of the coupling matrix from lookups.

5.4 The smuthi.linearsystem.tmatrix package

5.4.1 tmatrix

5.4.2 tmatrix.t_matrix

smuthi.linearsystem.tmatrix.t_matrix.**internal_mie_coefficient**(*tau*, *l*, *k_medium*, *k_particle*, *radius*)

Return the Mie coefficients to compute the internal field of a sphere.

Parameters

- **integer** (*l*) – spherical polarization, 0 for spherical TE and 1 for spherical TM
- **integer** – l=1,... multipole degree (polar quantum number)
- **float or complex** (*k_particle*) – wavenumber in surrounding medium (inverse length unit)
- **float or complex** – wavenumber inside sphere (inverse length unit)
- **float** (*radius*) – radius of sphere (length unit)

Returns Internal Mie coefficients as complex

smuthi.linearsystem.tmatrix.t_matrix.**mie_coefficient**(*tau*, *l*, *k_medium*, *k_particle*, *radius*)

Return the Mie coefficients of a sphere.

Parameters

- **integer** (*l*) – spherical polarization, 0 for spherical TE and 1 for spherical TM
- **integer** – l=1,... multipole degree (polar quantum number)
- **float or complex** (*k_particle*) – wavenumber in surrounding medium (inverse length unit)
- **float or complex** – wavenumber inside sphere (inverse length unit)
- **float** (*radius*) – radius of sphere (length unit)

Returns Mie coefficients as complex

smuthi.linearsystem.tmatrix.t_matrix.**rotate_t_matrix**(*T*, *l_max*, *m_max*, *euler_angles*, *wdsympy=False*)

T-matrix of a rotated particle.

Parameters

- **T** (*numpy.array*) – T-matrix
- **l_max** (*int*) – Maximal multipole degree
- **m_max** (*int*) – Maximal multipole order
- **euler_angles** (*list*) – Euler angles [alpha, beta, gamma] of rotated particle in (zy'z"-convention) in radian

Returns rotated T-matrix (numpy.array)

smuthi.linearsystem.tmatrix.t_matrix.**t_matrix_sphere**(*k_medium*, *k_particle*, *radius*, *l_max*, *m_max*)

T-matrix of a spherical scattering object.

Parameters

- **k_medium** (*float or complex*) – Wavenumber in surrounding medium (inverse length unit)
- **k_particle** (*float or complex*) – Wavenumber inside sphere (inverse length unit)
- **radius** (*float*) – Radius of sphere (length unit)
- **l_max** (*int*) – Maximal multipole degree
- **m_max** (*int*) – Maximal multipole order

Returns T-matrix as ndarray

5.5 The smuthi.linearsystem.tmatrix.nfmds package

5.5.1 nfmds.indexconverter

smuthi.linearsystem.tmatrix.nfmds.indexconverter.**multi_index_to_single_nfmds**(*tau*, *l*, *m*, *Nrank*, *Mrank*)

Converts a (tau,l,m) index to single index in NFMDS convention.

Parameters

- **tau** (*int*) – SVWF polarization (0 for spherical TE, 1 for spherical TM)
- **l** (*int*) – SVWF degree
- **m** (*int*) – SVWF order
- **Nrank** (*int*) – NFMDS Nrank parameter
- **Mrank** (*int*) – NFMDS Mrank parameter

Returns single index in NFMDS convention

Return type index (int)

smuthi.linearsystem.tmatrix.nfmds.indexconverter.**nfmds_to_smuthi_matrix**

Converts a T-matrix obtained with NFMDS to SMUTHI compatible format.

Parameters

- **T** (*array*) – T-matrix in NFMDS convention
- **Nrank** (*int*) – NFMDS Nrank parameter
- **Mrank** (*int*) – NFMDS Mrank parameter
- **l_max** (*int*) – Maximal multipole degree used for the spherical wave expansion of incoming and scattered field
- **m_max** (*int*) – Maximal multipole order used for the spherical wave expansion of incoming and scattered field

Returns T-matrix in SMUTHI convention

Return type Tsm (array)

smuthi.linearsystem.tmatrix.nfmds.indexconverter.**python_to_smuthi_matrix**
Converts a T-matrix obtained with Alan's code to SMUTHI compatible format.

Parameters

- **T** (*array*) – T-matrix in NFMDS convention
- **Nrank** (*int*) – Alan's lmax parameter
- **Mrank** (*int*) – Alan's lmax parameter
- **l_max** (*int*) – Maximal multipole degree used for the spherical wave expansion of incoming and scattered field
- **m_max** (*int*) – Maximal multipole order used for the spherical wave expansion of incoming and scattered field

Returns T-matrix in SMUTHI convention

Return type Tsm (array)

smuthi.linearsystem.tmatrix.nfmds.indexconverter.**single_index_to_multi_nfmds**
Converts single index to (tau,l,m) tuple in NFMDS convention.

Parameters

- **index** (*int*) – single index in NFMDS convention
- **Nrank** (*int*) – NFMDS Nrank parameter
- **Mrank** (*int*) – NFMDS Mrank parameter

Returns SVWF polarization (0 for spherical TE, 1 for spherical TM) l (int): SVWF degree m (int): SVWF order

Return type tau (int)

5.5.2 nfmds.stlmanager

smuthi.linearsystem.tmatrix.nfmds.stlmanager.**convert_stl_to_fem**(*stlname*, *femname*)
Converts STL to FEM file :param stlname: name of STL file :type stlname: string :param femname: name of FEM file :type femname: string

smuthi.linearsystem.tmatrix.nfmds.stlmanager.**readstl**(*stlname*)
Reads surface information from STL file :param stlname: name of STL file :type stlname: string

Returns A list of dictionaries with information about faces of scatterer geometry.

smuthi.linearsystem.tmatrix.nfmds.stlmanager.**writefem**(*femname*, *surfaces*)
Writes information about particle geometry to FEM file. :param femname: name of FEM file :type femname: string :param surfaces: information about faces of scatterer geometry :type surfaces: list

5.6 The smuthi.postprocessing package

5.6.1 postprocessing

5.6.2 postprocessing.far_field

Manage post processing steps to evaluate the scattered far field

class smuthi.postprocessing.far_field.**FarField**(*polar_angles='default'*, *azimuthal_angles='default'*, *angular_resolution=None*, *signal_type='intensity'*, *reference_point=None*)

Represent the far field amplitude and far field intensity of an electromagnetic field.

The electric field amplitude $\mathbf{A}(\theta, \phi)$ is defined by

$$\mathbf{E}(\mathbf{r}) \approx \frac{e^{ikr}}{-ikr}\mathbf{A}(\theta, \phi)$$

for $kr \to \infty$, compare equation (3.10) of Bohren and Huffman's textbook on light scattering.

In the above, $\mathbf{A}(\theta, \phi)$ is a complex, vector valued function of polar and azimuthal angle. It contains information on the amplitude and phase of the scattered electric field in far field domain.

The intensity $I_{\Omega,j}(\beta, \alpha)$ is defined by

$$P = \sum_{j=1}^{2} \iint \mathrm{d}^2\Omega \, I_{\Omega,j}(\beta, \alpha),$$

where P is the radiative power, j indicates the polarization and $\mathrm{d}^2\Omega = \mathrm{d}\alpha \sin\beta \mathrm{d}\beta$ denotes the infinitesimal solid angle.

Parameters

- **polar_angles** (*numpy.ndarray*) – array of polar angles for plane wave expansions. If 'default', use smuthi.fields.default_polar_angles
- **azimuthal_angles** (*ndarray or str*) – array of azimuthal angles for plane wave expansions. If 'default', use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range
- **signal_type** (*str*) – Use this field to describe the physical meaning of the power related signal (e.g., 'intensity' for standard power flux far fields).
- **reference_point** (*list or tuple*) – [x, y, z]-coordinates of point relative to which the far field is defined

alpha_grid()

Returns Meshgrid with α values.

append(*other*)

Combine two FarField objects with disjoint angular ranges. The other far field is appended to this one.

Parameters **other** (FarField) – far field to append to this one.

azimuthal_integral()

Far field intensity as a function of the polar angle cosine only.

$$P = \sum_{j=1}^{2} \int \mathrm{d}\cos\beta \, I_{\cos\beta,j}(\beta),$$

with

$$I_{\beta,j}(\beta) = \int \mathrm{d}\alpha \, I_j(\beta, \alpha),$$

Returns $I_{\cos\beta,j}(\beta)$ as numpy ndarray. First index is polarization, second is polar angle.

azimuthal_integral_times_sin_beta()
Far field intensity as a function of polar angle only.

$$P = \sum_{j=1}^{2} \int \mathrm{d}\beta \, I_{\beta,j}(\beta),$$

with

$$I_{\beta,j}(\beta) = \int \mathrm{d}\alpha \, \sin\beta I_j(\beta,\alpha),$$

Returns $I_{\beta,j}(\beta)$ as numpy ndarray. First index is polarization, second is polar angle.

beta_grid()

Returns Meshgrid with β values.

bottom()
Split far field into top and bottom part.

Returns FarField object with only the intensity for bottom hemisphere ($\beta \geq \pi/2$)

electric_field_amplitude()
Evaluate electric field amplitude vector

Returns Tuple of (A_x, A_y, A_z) numpy.ndarray objects with the Cartesian coordinates of complex electric field amplitude.

integral()
Integrate intensity to obtain total power P.

Returns P_j as numpy 1D-array with length 2, the index referring to polarization.

top()
Split far field into top and bottom part.

Returns FarField object with only the intensity for top hemisphere ($\beta \leq \pi/2$)

smuthi.postprocessing.far_field.**extinction_cross_section**(*simulation=None*, *initial_field=None*, *particle_list=None*, *layer_system=None*, *only_l=None*, *only_m=None*, *only_pol=None*, *only_tau=None*, *extinction_direction='both'*)
Evaluate the extinction cross section.

Parameters

- **simulation** (*smuthi.Simulation.simulation*) – Simulation object (optional)
- **initial_field** (smuthi.initial_field.PlaneWave) – Plane wave object (optional)
- **particle_list** (*list*) – List of smuthi.particles.Particle objects (optional)
- **layer_system** (smuthi.layers.LayerSystem) – Representing the stratified medium

- **only_pol** (*int*) – if set to 0 or 1, only this plane wave polarization (0=TE, 1=TM) is considered
- **only_tau** (*int*) – if set to 0 or 1, only this spherical vector wave polarization (0 — magnetic, 1 — electric) is considered
- **only_l** (*int*) – if set to positive number, only this multipole degree is considered
- **only_m** (*int*) – if set non-negative number, only this multipole order is considered
- **extinction_direction** (*string*) – if set to 'both': return full excinction, if to 'reflection': extinction of reflected wave, if to 'transmission': extinction of transmitted wave. See section on *Extinction cross section* for details.

Returns Extinction cross section.

smuthi.postprocessing.far_field.**pwe_to_ff_conversion**(*vacuum_wavelength*, *plane_wave_expansion*)

Compute the far field of a plane wave expansion object.

Parameters

- **vacuum_wavelength** (*float*) – Vacuum wavelength in length units.
- **plane_wave_expansion** (PlaneWaveExpansion) – Plane wave expansion to convert into far field object.

Returns A FarField object containing the far field intensity.

smuthi.postprocessing.far_field.**scattered_far_field**(*vacuum_wavelength*, *particle_list*, *layer_system*, *polar_angles='default'*, *azimuthal_angles='default'*, *angular_resolution=None*, *reference_point=None*)

Evaluate the scattered far field.

Parameters

- **vacuum_wavelength** (*float*) – in length units
- **particle_list** (*list*) – list of smuthi.Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – represents the stratified medium
- **polar_angles** (*numpy.ndarray or str*) – polar angles values (radian). if 'default', use smuthi.fields.default_polar_angles
- **azimuthal_angles** (*numpy.ndarray or str*) – azimuthal angle values (radian) if 'default', use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range
- **reference_point** (*list or tuple*) – If set to a value other than None, the far field will be calculated with this as its reference point.

Returns A smuthi.field_expansion.FarField object of the scattered field.

smuthi.postprocessing.far_field.**scattering_cross_section**(*initial_field*, *particle_list*, *layer_system*, *polar_angles='default'*, *azimuthal_angles='default'*, *angular_resolution=None*)

Evaluate and display the differential scattering cross section as a function of solid angle.

Parameters

- **initial_field** (*smuthi.initial.PlaneWave*) – Initial Plane wave
- **particle_list** (*list*) – scattering particles
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium
- **polar_angles** (*numpy.ndarray or str*) – polar angles values (radian). if 'default', use smuthi.fields.default_polar_angles
- **azimuthal_angles** (*numpy.ndarray or str*) – azimuthal angle values (radian) if 'default', use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range

Returns A smuthi.field_expansion.FarField object.

smuthi.postprocessing.far_field.**total_far_field**(*initial_field*, *particle_list*, *layer_system*, *polar_angles='default'*, *azimuthal_angles='default'*, *angular_resolution=None*, *reference_point=None*)

Evaluate the total far field, the initial far field and the scattered far field. Cannot be used if initial field is a plane wave.

Parameters

- **initial_field** (smuthi.initial_field.InitialField) – represents the initial field
- **particle_list** (*list*) – list of smuthi.Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – represents the stratified medium
- **polar_angles** (*numpy.ndarray or str*) – polar angles values (radian). if 'default', use smuthi.fields.default_polar_angles
- **azimuthal_angles** (*numpy.ndarray or str*) – azimuthal angle values (radian) if 'default', use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range
- **reference_point** (*list or tuple*) – If set to a value other than None, the far field will be calculated with this as its reference point.

Returns A tuple of three smuthi.field_expansion.FarField objects for total, initial and scattered far field. Mind that the scattered far field has no physical meaning and is for illustration purposes only.

smuthi.postprocessing.far_field.**total_scattering_cross_section**(*simulation=None*, *initial_field=None*, *particle_list=None*, *layer_system=None*, *polar_angles='default'*, *azimuthal_angles='default'*, *angular_resolution=None*)

Evaluate the total scattering cross section.

Parameters

- **simulation** (*smuthi.Simulation.simulation*) – Simulation object (optional)
- **initial_field** (smuthi.initial_field.PlaneWave) – Initial Plane wave (optional)
- **particle_list** (*list*) – scattering particles (optional)
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium (optional)
- **polar_angles** (*numpy.ndarray or str*) – polar angles values (radian, default None). If None, use smuthi.fields.default_polar_angles
- **azimuthal_angles** (*numpy.ndarray or str*) – azimuthal angle values (radian, default None). If None, use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range

Returns A tuple of smuthi.field_expansion.FarField objects, one for forward scattering (i.e., into the top hemisphere) and one for backward scattering (bottom hemisphere).

5.6.3 postprocessing.graphical_output

Functions to generate plots and animations.

smuthi.postprocessing.graphical_output.**compute_near_field**(*simulation=None*, *X=None*, *Y=None*, *Z=None*, *type='scatt'*, *chunksize=None*, *k_parallel='default'*, *azimuthal_angles='default'*, *angular_resolution=None*)

Compute a certain component of the electric near field

smuthi.postprocessing.graphical_output.**plot_layer_interfaces**(*dim1min*, *dim1max*, *layer_system*)

Add lines to plot to display layer system interfaces

Parameters

- **dim1min** (*float*) – From what x-value plot line
- **dim1max** (*float*) – To what x-value plot line

- **layer_system** (smuthi.layers.LayerSystem) – Stratified medium

smuthi.postprocessing.graphical_output.**plot_particles**(*xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*, *particle_list*, *draw_circumscribing_sphere*, *fill_particle=True*)

Add circles, ellipses and rectangles to plot to display spheres, spheroids and cylinders.

Parameters

- **xmin** (*float*) – Minimal x-value of plot
- **xmax** (*float*) – Maximal x-value of plot
- **ymin** (*float*) – Minimal y-value of plot
- **ymax** (*float*) – Maximal y-value of plot
- **zmin** (*float*) – Minimal z-value of plot
- **zmax** (*float*) – Maximal z-value of plot
- **particle_list** (*list*) – List of smuthi.particles.Particle objects
- **draw_circumscribing_sphere** (*bool*) – If true (default), draw a circle indicating the circumscribing sphere of particles.
- **fill_particle** (*bool*) – If true, draw opaque particles.

smuthi.postprocessing.graphical_output.**show_far_field**(*far_field, show_plots=True, show_opts=[{'label': 'far_field'}], save_plots=False, save_opts=None, save_data=False, data_format='hdf5', output-dir='.', flip_downward=True, split=True, log_scale=False*)

Display and export the far field.

Parameters

- **far_field** (*smuthi.field_expansion.FarField*) – Far-field object to show and export
- **show_plots** (*bool*) – Display plots if True
- **show_opts** (*dict list*) – List of dictionaries containing options to be passed to pcolormesh for plotting. If save_plots=True, a 1:1 correspondence between show_opts and save_opts dictionaries is assumed. For simplicity, one can also provide a single show_opts entry that will be applied to all save_opts. The following keys are available (see matplotlib.pyplot.pcolormesh documentation): 'alpha' (None) 'cmap' ('inferno') 'norm' (None), is set to matplotlib.colors.LogNorm() if log_scale is True 'vmin' (None), applies only to 2D plots 'vmax' (None), applies only to 2D plots 'shading' ('nearest'), applies only to 2D plots. 'gouraud' is also available 'linewidth' (None), applies only to 1D plots 'linestyle' (None), applies only to 1D plots 'marker' (None), applies only to 1D plots An optional extra key called 'label' of type string is shown in the plot title and appended to the associated file if save_plots is True Finally, an optional 'figsize' key is available to set the width and height of the figure window (see matplotlib.pyplot.figure documentation)
- **save_plots** (*bool*) – If True, plots are exported to file.

- **save_opts** (*dict list*) – List of dictionaries containing options to be passed to savefig. A 1:1 correspondence between save_opts and show_opts dictionaries is assumed. For simplicity, one can also provide a single save_opts entry that will be applied to all show_opts. The following keys are made available (see matplotlib.pyplot.savefig documentation): 'dpi' (None) 'orientation' (None) 'format' ('png'), also available: eps, jpeg, jpg, pdf, ps, svg, tif, tiff ... 'transparent' (False) 'bbox_inches' ('tight') 'pad_inches' (0.1)
- **save_data** (*bool*) – If True, raw data are exported to file
- **data_format** (*str*) – Output data format string, 'hdf5' and 'ascii' formats are available
- **outputdir** (*str*) – Path to the directory where files are to be saved
- **flip_downward** (*bool*) – If True, represent downward directions as 0-90 deg instead of 90-180
- **split** (*bool*) – If True, show two different plots for upward and downward directions
- **log_scale** (*bool*) – If True, set a logarithmic scale

smuthi.postprocessing.graphical_output.**show_near_field**(*simulation=None*, *quantities_to_plot=None*, *show_plots=True*, *show_opts=None*, *save_plots=False*, *save_opts=None*, *save_data=False*, *data_format='hdf5'*, *outputdir='.'*, *xmin=0*, *xmax=0*, *ymin=0*, *ymax=0*, *zmin=0*, *zmax=0*, *resolution_step=25*, *k_parallel='default'*, *azimuthal_angles='default'*, *angular_resolution=None*, *draw_circumscribing_sphere=True*, *show_internal_field=False*)

Plot the electric near field along a plane. To plot along the xy-plane, specify zmin=zmax and so on.

Parameters

- **simulation** (smuthi.simulation.Simulation) – Simulation object
- **quantities_to_plot** – List of strings that specify what to plot. Select from 'E_x', 'E_y', 'E_z', 'norm(E)' The list may contain one or more of the following strings:

  'E_x' real part of x-component of complex total electric field 'E_y' real part of y-component of complex total electric field 'E_z' real part of z-component of complex total electric field 'norm(E)' norm of complex total electric field

  'E_scat_x' real part of x-component of complex scattered electric field 'E_scat_y' real part of y-component of complex scattered electric field 'E_scat_z' real part of z-component of complex scattered electric field 'norm(E_scat)' norm of complex scattered electric field

  'E_init_x' real part of x-component of complex initial electric field 'E_init_y' real part of y-component of complex initial electric field 'E_init_z' real part of z-component of complex initial electric field 'norm(E_init)' norm of complex initial electric field
- **show_plots** (*logical*) – If True, plots are shown
- **show_opts** (*dict list*) – List of dictionaries containing options to be passed to imshow for plotting. For each entry in quantities_to_plot, all show_opts dictionaries will

be applied. If save_plots=True, a 1:1 correspondence between show_opts and save_opts dictionaries is assumed. For simplicity, one can also provide a single show_opts entry that will be applied to all save_opts. The following keys are made available (see matplotlib.pyplot.imshow documentation): 'cmap' defaults to 'inferno' for norm quantities and 'RdYlBu' otherwise 'norm' (None). If a norm is provided, its vmin and vmax take precedence 'aspect' ('equal') 'interpolation' (None), also available: bilinear, bicubic, spline16, quadric, ... 'alpha' (None) 'vmin' (None), will be set to 0 for norm quantities and -vmax otherwise 'vmax' initialized with the max of the quantity to plot 'origin' ('lower') 'extent' calculated automatically based on plotting coordinate limits An optional extra key called 'label' of type string is shown in the plot title and appended to the associated file if save_plots is True Finally, an optional 'figsize' key is available to set the width and height of the figure window (see matplotlib.pyplot.figure documentation)

- **save_plots** (*logical*) – If True, plots are exported to file.
- **save_opts** (*dict list*) – List of dictionaries containing options to be passed to savefig. For each entry in quantities_to_plot, all save_opts dictionaries will be applied. A 1:1 correspondence between save_opts and show_opts dictionaries is assumed. For simplicity, one can also provide a single save_opts entry that will be applied to all show_opts. The following keys are made available (see matplotlib.pyplot.savefig documentation): 'dpi' (None) 'orientation' (None) 'format' ('png'), also available: eps, jpeg, jpg, pdf, ps, svg, tif, tiff ... 'transparent' (False) 'bbox_inches' ('tight') 'pad_inches' (0.1) Passing 'gif' as one of the format values will result in an animation if the quantity to plot is of non-norm type
- **save_data** (*logical*) – If True, raw data are exported to file
- **data_format** (*str*) – Output data format string, 'hdf5' and 'ascii' formats are available
- **outputdir** (*str*) – Path to directory where to save the export files
- **xmin** (*float*) – Plot from that x (length unit)
- **xmax** (*float*) – Plot up to that x (length unit)
- **ymin** (*float*) – Plot from that y (length unit)
- **ymax** (*float*) – Plot up to that y (length unit)
- **zmin** (*float*) – Plot from that z (length unit)
- **zmax** (*float*) – Plot up to that z (length unit)
- **resolution_step** (*float*) – Compute the field with that spatial resolution (length unit, distance between computed points), can be a tuple for [resx, resy, resz]
- **k_parallel** (*numpy.ndarray or str*) – in-plane wavenumbers for the plane wave expansion if 'default', use smuthi.fields.default_Sommerfeld_k_parallel_array
- **azimuthal_angles** (*numpy.ndarray or str*) – azimuthal angles for the plane wave expansion if 'default', use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range
- **draw_circumscribing_sphere** (*bool*) – If true (default), draw a circle indicating the circumscribing sphere of particles.
- **show_internal_field** (*bool*) – If true, compute also the field inside the particles (only for spheres)

smuthi.postprocessing.graphical_output.**show_scattered_far_field**(*simulation, show_plots=True, show_opts=[{'label': 'scattered_far_field'}], save_plots=False, save_opts=None, save_data=False, data_format='hdf5', outputdir='.', flip_downward=True, split=True, log_scale=False, polar_angles='default', azimuthal_angles='default', angular_resolution=None*)

Display and export the scattered far field.

Parameters

- **simulation** (smuthi.simulation.Simulation) – Simulation object
- **show_plots** (*bool*) – Display plots if True
- **show_opts** (*dict list*) – List of dictionaries containing options to be passed to pcolormesh for plotting. If save_plots=True, a 1:1 correspondence between show_opts and save_opts dictionaries is assumed. For simplicity, one can also provide a single show_opts entry that will be applied to all save_opts. The following keys are available (see matplotlib.pyplot.pcolormesh documentation): 'alpha' (None) 'cmap' ('inferno') 'norm' (None), is set to matplotlib.colors.LogNorm() if log_scale is True 'vmin' (None), applies only to 2D plots 'vmax' (None), applies only to 2D plots 'shading' ('nearest'), applies only to 2D plots. 'gouraud' is also available 'linewidth' (None), applies only to 1D plots 'linestyle' (None), applies only to 1D plots 'marker' (None), applies only to 1D plots An optional extra key called 'label' of type string is shown in the plot title and appended to the associated file if save_plots is True
- **save_plots** (*bool*) – If True, plots are exported to file.
- **save_opts** (*dict list*) – List of dictionaries containing options to be passed to savefig. A 1:1 correspondence between save_opts and show_opts dictionaries is assumed. For simplicity, one can also provide a single save_opts entry that will be applied to all show_opts. The following keys are made available (see matplotlib.pyplot.savefig documentation): 'dpi' (None) 'orientation' (None) 'format' ('png'), also available: eps, jpeg, jpg, pdf, ps, svg, tif, tiff ... 'transparent' (False) 'bbox_inches' ('tight') 'pad_inches' (0.1)
- **save_data** (*bool*) – If True, raw data are exported to file
- **data_format** (*str*) – Output data format string, 'hdf5' and 'ascii' formats are available
- **outputdir** (*str*) – Path to the directory where files are to be saved
- **flip_downward** (*bool*) – If True, represent downward directions as 0-90 deg instead of 90-180
- **split** (*bool*) – If True, show two different plots for upward and downward directions
- **log_scale** (*bool*) – If True, set a logarithmic scale

- **polar_angles** (*numpy.ndarray or str*) – Polar angles values (radian). If 'default', use smuthi.fields.default_polar_angles
- **azimuthal_angles** (*numpy.ndarray or str*) – Azimuthal angle values (radian). If 'default', use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range

smuthi.postprocessing.graphical_output.**show_scattering_cross_section**(*simulation, show_plots=True, show_opts=[{'label': 'scattering_cross_section'}], save_plots=False, save_opts=None, save_data=False, data_format='hdf5', output_dir='.', flip_downward=True, split=True, log_scale=False, polar_angles='default', azimuthal_angles='default', angular_resolution=None*)

Display and export the differential scattering cross section.

Parameters

- **simulation** (smuthi.simulation.Simulation) – Simulation object
- **show_plots** (*bool*) – Display plots if True
- **show_opts** (*dict list*) – List of dictionaries containing options to be passed to pcolormesh for plotting. If save_plots=True, a 1:1 correspondence between show_opts and save_opts dictionaries is assumed. For simplicity, one can also provide a single show_opts entry that will be applied to all save_opts. The following keys are available (see matplotlib.pyplot.pcolormesh documentation): 'alpha' (None) 'cmap' ('inferno') 'norm' (None), is set to matplotlib.colors.LogNorm() if log_scale is True 'vmin' (None), applies only to 2D plots 'vmax' (None), applies only to 2D plots 'shading' ('nearest'), applies only to 2D plots. 'gouraud' is also available 'linewidth' (None), applies only to 1D plots 'linestyle' (None), applies only to 1D plots 'marker' (None), applies only to 1D plots An optional extra key called 'label' of type string is shown in the plot title and appended to the associated file if save_plots is True
- **save_plots** (*bool*) – If True, plots are exported to file.
- **save_opts** (*dict list*) – List of dictionaries containing options to be passed to savefig. A 1:1 correspondence between save_opts and show_opts dictionaries is assumed. For simplicity, one can also provide a single save_opts entry that will be applied to all show_opts. The following keys are made available (see matplotlib.pyplot.savefig documentation): 'dpi' (None) 'orientation' (None) 'format' ('png'), also available: eps, jpeg, jpg, pdf, ps, svg, tif, tiff ... 'transparent' (False) 'bbox_inches' ('tight') 'pad_inches' (0.1)

- **save_data** (*bool*) – If True, raw data are exported to file
- **data_format** (*str*) – Output data format string, 'hdf5' and 'ascii' formats are available
- **outputdir** (*str*) – Path to the directory where files are to be saved
- **flip_downward** (*bool*) – If True, represent downward directions as 0-90 deg instead of 90-180
- **split** (*bool*) – If True, show two different plots for upward and downward directions
- **log_scale** (*bool*) – If True, set a logarithmic scale
- **polar_angles** (*numpy.ndarray or str*) – Polar angles values (radian). If 'default', use smuthi.fields.default_polar_angles
- **azimuthal_angles** (*numpy.ndarray or str*) – Azimuthal angle values (radian). If 'default', use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range

smuthi.postprocessing.graphical_output.**show_total_far_field**(*simulation, show_plots=True, show_opts=[{'label': 'total_far_field'}], save_plots=False, save_opts=None, save_data=False, data_format='hdf5', outputdir='.', flip_downward=True, split=True, log_scale=False, polar_angles='default', azimuthal_angles='default', angular_resolution=None*)

Display and export the total far field. This function cannot be used if the inital field is a plane wave.

Parameters

- **simulation** (smuthi.simulation.Simulation) – Simulation object
- **show_plots** (*bool*) – Display plots if True
- **show_opts** (*dict list*) – List of dictionaries containing options to be passed to pcolormesh for plotting. If save_plots=True, a 1:1 correspondence between show_opts and save_opts dictionaries is assumed. For simplicity, one can also provide a single show_opts entry that will be applied to all save_opts. The following keys are available (see matplotlib.pyplot.pcolormesh documentation): 'alpha' (None) 'cmap' ('inferno') 'norm' (None), is set to matplotlib.colors.LogNorm() if log_scale is True 'vmin' (None), applies only to 2D plots 'vmax' (None), applies only to 2D plots 'shading' ('nearest'), applies only to 2D plots. 'gouraud' is also available 'linewidth' (None), applies only to 1D plots 'linestyle' (None), applies only to 1D plots 'marker' (None), applies only to 1D plots An optional extra key called 'label' of type string is shown in the plot title and appended to the associated file if save_plots is True
- **save_plots** (*bool*) – If True, plots are exported to file.

- **save_opts** (*dict list*) – List of dictionaries containing options to be passed to savefig. A 1:1 correspondence between save_opts and show_opts dictionaries is assumed. For simplicity, one can also provide a single save_opts entry that will be applied to all show_opts. The following keys are made available (see matplotlib.pyplot.savefig documentation): 'dpi' (None) 'orientation' (None) 'format' ('png'), also available: eps, jpeg, jpg, pdf, ps, svg, tif, tiff ... 'transparent' (False) 'bbox_inches' ('tight') 'pad_inches' (0.1)
- **save_data** (*bool*) – If True, raw data are exported to file
- **data_format** (*str*) – Output data format string, 'hdf5' and 'ascii' formats are available
- **outputdir** (*str*) – Path to the directory where files are to be saved
- **flip_downward** (*bool*) – If True, represent downward directions as 0-90 deg instead of 90-180
- **split** (*bool*) – If True, show two different plots for upward and downward directions
- **log_scale** (*bool*) – If True, set a logarithmic scale
- **polar_angles** (*numpy.ndarray or str*) – Polar angles values (radian). If 'default', use smuthi.fields.default_polar_angles
- **azimuthal_angles** (*numpy.ndarray or str*) – Azimuthal angle values (radian). If 'default', use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range

postprocessing.internal_field

Manage post processing steps to evaluate the electric field inside a sphere

smuthi.postprocessing.internal_field.**internal_field_piecewise_expansion**(*vacuum_wavelength*, *particle_list*, *layer_system*)

Compute a piecewise field expansion of the internal field of spheres.

Parameters

- **vacuum_wavelength** (*float*) – vacuum wavelength
- **particle_list** (*list*) – list of smuthi.particles.Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium

Returns internal field as smuthi.field_expansion.PiecewiseFieldExpansion object

5.6.4 postprocessing.scattered_field

Manage post processing steps to evaluate the scattered electric field

smuthi.postprocessing.scattered_field.**evaluate_scattered_field_stat_phase_approx**(*x*, *y*, *z*, *vacuum_wavelength*, *particle_list*, *layer_system*)

Evaluate the scattered electric field for N particles on a substrate. The substrate reflection is evaluated by means of the stationary phase approximation, as presented in "A quick way to approximate a Sommerfeld-Weyl_type Sommerfeld integral" by W.C. Chew (1988).

See also the technical note "Usage of the stationary phase approximation in SMUTHI" by A. Egel (2020)

The stationary phase approximation is expected to yield good results for field points far away from the particles.

Note: This function assumes that the particles are located in the upper layer of a two-layer system (particles on substrate). For other cases, this function does not apply.
****

Parameters

- **x** (*float or numpy.ndarray*) – x-coordinates of query points
- **y** (*float or numpy.ndarray*) – y-coordinates of query points
- **z** (*float or numpy.ndarray*) – z-coordinates of query points
- **vacuum_wavelength** (*float*) – Vacuum wavelength λ (length unit)
- **particle_list** (*list*) – List of Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – Stratified medium

Returns Tuple of (E_x, E_y, E_z) numpy.ndarray objects with the Cartesian coordinates of complex electric field.

smuthi.postprocessing.scattered_field.**scattered_field_piecewise_expansion**(*vacuum_wavelength*, *particle_list*, *layer_system*, *k_parallel='default'*, *azimuthal_angles='default'*, *angular_resolution=None*, *layer_numbers=None*)

Compute a piecewise field expansion of the scattered field.

Parameters

- **vacuum_wavelength** (*float*) – vacuum wavelength
- **particle_list** (*list*) – list of smuthi.particles.Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – stratified medium
- **k_parallel** (*numpy.ndarray or str*) – in-plane wavenumbers array. if 'default', use smuthi.fields.default_Sommerfeld_k_parallel_array

- **azimuthal_angles** (*numpy.ndarray or str*) – azimuthal angles array if 'default', use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range
- **layer_numbers** (*list*) – if specified, append only plane wave expansions for these layers

Returns scattered field as smuthi.field_expansion.PiecewiseFieldExpansion object

smuthi.postprocessing.scattered_field.**scattered_field_pwe**(*vacuum_wavelength*, *particle_list*, *layer_system*, *layer_number*, *k_parallel='default'*, *azimuthal_angles='default'*, *angular_resolution=None*, *include_direct=True*, *include_layer_response=True*, *only_l=None*, *only_m=None*, *only_pol=None*, *only_tau=None*)

Calculate the plane wave expansion of the scattered field of a set of particles.

Parameters

- **vacuum_wavelength** (*float*) – Vacuum wavelength (length unit)
- **particle_list** (*list*) – List of Particle objects
- **layer_system** (smuthi.layers.LayerSystem) – Stratified medium
- **layer_number** (*int*) – Layer number in which the plane wave expansion should be valid
- **k_parallel** (*numpy.ndarray or str*) – in-plane wavenumbers array. if 'default', use smuthi.fields.default_Sommerfeld_k_parallel_array
- **azimuthal_angles** (*numpy.ndarray or str*) – azimuthal angles array if 'default', use smuthi.fields.default_azimuthal_angles
- **angular_resolution** (*float*) – If provided, angular arrays are generated with this angular resolution over the default angular range
- **include_direct** (*bool*) – If True, include the direct scattered field
- **include_layer_response** (*bool*) – If True, include the layer system response
- **only_pol** (*int*) – if set to 0 or 1, only this plane wave polarization (0=TE, 1=TM) is considered
- **only_tau** (*int*) – if set to 0 or 1, only this spherical vector wave polarization (0 — magnetic, 1 — electric) is considered
- **only_l** (*int*) – if set to positive number, only this multipole degree is considered
- **only_m** (*int*) – if set to non-negative number, only this multipole order is considered

Returns A tuple of PlaneWaveExpansion objects for upgoing and downgoing waves.

5.6.5 postprocessing.power_flux

Manage post processing steps to evaluate power flux

smuthi.postprocessing.power_flux.**power_flux_through_zplane**(*vacuum_wavelength*, *z*, *upgoing_pwe=None*, *downgoing_pwe=None*)

Evaluate time averaged power flux though a plane of z=const.

Parameters

- **vacuum_wavelength** (*float*) – Vacuum wavelength in length units.
- **z** (*float*) – plane height z
- **upgoing_pwe** (PlaneWaveExpansion) – of kind "upgoing"
- **downgoing_pwe** (PlaneWaveExpansion) – of kind "downgoing"

Returns Time averaged energy flux.

5.7 The smuthi.utility package

5.7.1 utility

5.7.2 utility.automatic_parameter_selection

Functions that assist the user in the choice of suitable numerical simulation parameters.

smuthi.utility.automatic_parameter_selection.**converge_angular_resolution**(*simulation*, *detector='extinction cross section'*, *tolerance=0.001*, *max_iter=30*, *ax=None*)

Find a suitable discretization step size for the default angular arrays used for plane wave expansions.

Parameters

- **simulation** (smuthi.simulation.Simulation) – Simulation object
- **detector** (*function or string*) – Function that accepts a simulation object and returns a detector value the change of which is used to define convergence. Alternatively, use "extinction cross section" (default) to have the extinction cross section as the detector value.
- **tolerance** (*float*) – Relative tolerance for the detector value change.
- **max_iter** (*int*) – Break convergence loops after that number of iterations, even if no convergence has been achieved.

- **ax** (*np.array of AxesSubplot*) – Array of AxesSubplots where to live-plot convergence output

Returns Detector value for converged settings.

smuthi.utility.automatic_parameter_selection.**converge_l_max**(*simulation*, *detector='extinction cross section'*, *tolerance=0.001*, *tolerance_steps=2*, *max_iter=100*, *start_from_1=True*, *ax=None*)

Find suitable multipole cutoff degree *l_max* for a given particle and simulation. The routine starts with the current *l_max* of the particle. The value of *l_max* is successively incremented in a loop until the resulting relative change in the detector value is smaller than the specified tolerance. The method updates the input particle object with the *l_max* value for which convergence has been achieved.

Parameters

- **simulation** (smuthi.simulation.Simulation) – Simulation object containing the particle
- **detector** (*function or string*) – Function that accepts a simulation object and returns a detector value the change of which is used to define convergence. Alternatively, use "extinction cross section" (default) to have the extinction cross section as the detector value.
- **tolerance** (*float*) – Relative tolerance for the detector value change.
- **tolerance_steps** (*int*) – Number of consecutive steps at which the tolerance must be met during multipole truncation convergence. Default: 2
- **max_iter** (*int*) – Break convergence loop after that number of iterations, even if no convergence has been achieved.
- **start_from_1** (*logical*) – If true (default), start from *l_max=1*. Otherwise, start from the current particle *l_max*.
- **ax** (*np.array of AxesSubplot*) – Array of AxesSubplots where to live-plot convergence output

Returns

A 3-tuple containing

- detector value of converged or break-off parameter settings.
- series of lmax values
- the detector values for the given lmax values

smuthi.utility.automatic_parameter_selection.**converge_m_max**(*simulation*, *detector='extinction cross section'*, *tolerance=0.001*, *target_value=None*, *ax=None*)

Find suitable multipole cutoff order *m_max* for a given particle and simulation. The routine starts with the current *l_max* of the particle, i.e. with *m_max=l_max*. The value of *m_max* is successively decremented in a loop until the resulting relative change in the detector value is larger than the specified tolerance. The method updates the input particle object with the so determined *m_max*.

Parameters

- **simulation** (smuthi.simulation.Simulation) – Simulation object containing the particle
- **detector** (*function or string*) – Function that accepts a simulation object and returns a detector value the change of which is used to define convergence. Alternatively, use "extinction cross section" (default) to have the extinction cross section as the detector value.
- **tolerance** (*float*) – Relative tolerance for the detector value change.
- **max_iter** (*int*) – Break convergence loop after that number of iterations, even if no convergence has been achieved.
- **target_value** (*float*) – If available (typically from preceding neff selection procedure), use as target detector value
- **ax** (*np.array of AxesSubplot*) – Array of AxesSubplots where to live-plot convergence output

Returns Detector value of converged or break-off parameter settings.

smuthi.utility.automatic_parameter_selection.**converge_multipole_cutoff**(*simulation*, *detector='extinction cross section'*, *tolerance=0.001*, *tolerance_steps=2*, *max_iter=100*, *current_value=None*, *l_max_list=None*, *detector_value_list=None*, *converge_m=True*, *ax=None*)

Find suitable multipole cutoff degree *l_max* and order *m_max* for all particles in a given simulation object. The method updates the input simulation object with the so determined multipole truncation values.

Parameters

- **simulation** (smuthi.simulation.Simulation) – Simulation object
- **detector** (*function or string*) – Function that accepts a simulation object and returns a detector value the change of which is used to define convergence Alternatively, use "extinction cross section" (default) to have the extinction cross section as the detector value
- **tolerance** (*float*) – Relative tolerance for the detector value change
- **tolerance_steps** (*int*) – Number of consecutive steps at which the tolerance must be met during multipole truncation convergence. Default: 2

- **max_iter** (*int*) – Break convergence loops after that number of iterations, even if no convergence has been achieved
- **current_value** (*float*) – If specified, skip l_max run and use this value for the resulting detector value. Otherwise, start with l_max run.
- **l_max_list** (*list*) – If current_value was specified, the l_max run is skipped. Then, this list is returned as the second item in the returned tuple.
- **detector_value_list** (*list*) – If current_value was specified, the l_max run is skipped. Then, this list is returned as the third item in the returned tuple.
- **converge_m** (*logical*) – If false, only converge *l_max*, but keep *m_max=l_max*. Default is true
- **ax** (*np.array of AxesSubplot*) – Array of AxesSubplots where to live-plot convergence output

Returns

A 3-tuple containing

- detector value of converged or break-off parameter settings.
- series of lmax values
- the detector values for the given lmax values

smuthi.utility.automatic_parameter_selection.**converge_neff_max**(*simulation*, *detector='extinction cross section'*, *tolerance=0.001*, *tolerance_factor=0.1*, *tolerance_steps=2*, *max_iter=30*, *neff_imag=0.01*, *neff_resolution=0.002*, *neff_max_increment=0.5*, *neff_max_offset=0*, *converge_lm=True*, *ax=None*)

Find a suitable truncation value for the multiple scattering Sommerfeld integral contour and update the simulation object accordingly.

Parameters

- **simulation** (smuthi.simulation.Simulation) – Simulation object
- **detector** (*function or string*) – Function that accepts a simulation object and returns a detector value the change of which is used to define convergence. Alternatively, use "extinction cross section" (default) to have the extinction cross section as the detector value.
- **tolerance** (*float*) – Relative tolerance for the detector value change.

- **tolerance_factor** (*float*) – During neff selection, a smaller tolerance should be allowed to avoid fluctuations of the order of ~tolerance which would compromise convergence. Default: 0.1
- **tolerance_steps** (*int*) – Number of consecutive steps at which the tolerance must be met during multipole truncation convergence. Default: 2
- **max_iter** (*int*) – Break convergence loops after that number of iterations, even if no convergence has been achieved.
- **neff_imag** (*float*) – Extent of the contour into the negative imaginary direction (in terms of effective refractive index, n_eff=kappa/omega).
- **neff_resolution** (*float*) – Discretization of the contour (in terms of eff. refractive index).
- **neff_max_increment** (*float*) – Increment the neff_max parameter with that step size
- **neff_max_offset** (*float*) – Start neff_max selection from the last estimated singularity location plus this value (in terms of effective refractive index)
- **converge_lm** (*logical*) – If set to true, update multipole truncation during each step (this takes longer time, but is necessary for critical use cases like flat particles on a substrate)
- **ax** (*np.array of AxesSubplot*) – Array of AxesSubplots where to live-plot convergence output

Returns Detector value for converged settings.

smuthi.utility.automatic_parameter_selection.**converge_neff_resolution**(*simulation*, *detector='extinction cross section'*, *tolerance=0.001*, *max_iter=30*, *neff_imag=0.01*, *neff_max=None*, *neff_resolution=0.01*, *ax=None*)

Find a suitable discretization step size for the multiple scattering Sommerfeld integral contour and update the simulation object accordingly.

Parameters

- **simulation** (smuthi.simulation.Simulation) – Simulation object
- **detector** (*function or string*) – Function that accepts a simulation object and returns a detector value the change of which is used to define convergence. Alternatively, use "extinction cross section" (default) to have the extinction cross section as the detector value.
- **tolerance** (*float*) – Relative tolerance for the detector value change.
- **max_iter** (*int*) – Break convergence loops after that number of iterations, even if no convergence has been achieved.
- **neff_imag** (*float*) – Extent of the contour into the negative imaginary direction (in terms of effective refractive index, n_eff=kappa/omega).
- **neff_max** (*float*) – Truncation value of contour (in terms of effective refractive index).

- **neff_resolution** (*float*) – Discretization of the contour (in terms of eff. refractive index) - start value for iteration
- **ax** (*np.array of AxesSubplot*) – Array of AxesSubplots where to live-plot convergence output

Returns Detector value for converged settings.

smuthi.utility.automatic_parameter_selection.**evaluate**(*simulation*, *detector*)

Run a simulation and evaluate the detector. :param simulation: simulation object :type simulation: smuthi.simulation.Simulation :param detector: Specify a method that accepts a simulation as input and returns a float. Otherwise, type "extinction cross section" to use the extinction cross section as a detector.

Returns The detector value (float)

smuthi.utility.automatic_parameter_selection.**select_numerical_parameters**(*simulation*, *detector='extinction cross section'*, *tolerance=0.001*, *tolerance_factor=0.1*, *tolerance_steps=2*, *max_iter=30*, *neff_imag=0.01*, *neff_resolution=0.01*, *select_neff_max=True*, *neff_max_increment=0.5*, *neff_max_offset=0*, *neff_max=None*, *select_neff_resolution=True* *select_angular_resolution=* *select_multipole_cutoff=Tru* *relative_convergence=True*, *show_plot=True*)

Trigger automatic selection routines for various numerical parameters.

Parameters

- **simulation** (smuthi.simulation.Simulation) – Simulation object from which parameters are read and into which results are stored.

- **detector** (*function or string*) – Function that accepts a simulation object and returns a detector value the change of which is used to define convergence. Alternatively, use "extinction cross section" (default) to have the extinction cross section as the detector value.
- **tolerance** (*float*) – Relative tolerance for the detector value change.
- **tolerance_factor** (*float*) – During neff selection, a smaller tolerance should be allowed to avoid fluctuations of the order of ~tolerance which would compromise convergence. Default: 0.1
- **tolerance_steps** (*int*) – Number of consecutive steps at which the tolerance must be met during multipole truncation convergence. Default: 2
- **max_iter** (*int*) – Break convergence loops after that number of iterations, even if no convergence has been achieved.
- **neff_imag** (*float*) – Extent of the contour into the negative imaginary direction (in terms of effective refractive index, n_eff=kappa/omega)
- **neff_resolution** (*float*) – Discretization of the contour (in terms of eff. refractive index) - if *select_neff_resolution* is true, this value will be eventually overwritten. However, it is required in any case. Default: 1e-2
- **select_neff_max** (*logical*) – If set to true (default), the Sommerfeld integral truncation parameter *neff_max* is determined automatically with the help of a Cauchy convergence criterion.
- **neff_max_increment** (*float*) – Only needed if *select_neff_max* is true. Step size with which *neff_max* is incremented.
- **neff_max_offset** (*float*) – Only needed if *select_neff_max* is true. Start n_eff selection from the last estimated singularity location plus this value (in terms of effective refractive index)
- **neff_max** (*float*) – Only needed if *select_neff_max* is false. Truncation value of contour (in terms of effective refractive index).
- **select_neff_resolution** (*logical*) – If set to true (default), the Sommerfeld integral discretization parameter *neff_resolution* is determined automatically with the help of a Cauchy convergence criterion.
- **select_angular_resolution** (*logical*) – If set to true, the angular resolution step for the default polar and azimuthal angles is determined automatically according to a Cauchy convergenge criterion.
- **select_multipole_cutoff** (*logical*) – If set to true (default), the multipole expansion cutoff parameters *l_max* and *m_max* are determined automatically with the help of a Cauchy convergence criterion.
- **relative_convergence** (*logical*) – If set to true (default), the *neff_max* convergence and the *l_max* and *m_max* convergence routine are performed in the spirit of relative convergence, i.e., the multipole expansion convergence is checked again for each value of the Sommerfeld integral truncation. This takes more time, but is required at least in the case of flat particles near interfaces.

smuthi.utility.automatic_parameter_selection.**update_contour**(*simulation*, *neff_imag=0.005*, *neff_max=None*, *neff_max_offset=0.5*, *neff_resolution=0.002*)

Update the default *k_parallel* arrays in smuthi.fields with a newly constructed Sommerfeld integral contour, and

set the simulation object to use the default contour for particle coupling.

Parameters

- **simulation** (smuthi.simulation.Simulation) – Simulation object
- **neff_imag** (*float*) – Extent of the contour into the negative imaginary direction (in terms of effective refractive index, n_eff=kappa/omega).
- **neff_max** (*float*) – Truncation value of contour (in terms of effective refractive index).
- **neff_max_offset** (*float*) – If no value for *neff_max* is specified, use the last estimated singularity location plus this value (in terms of effective refractive index).
- **neff_resolution** (*float*) – Discretization of the contour (in terms of effective refractive index).

smuthi.utility.automatic_parameter_selection.**update_lmax_mmax**(*simulation*, *l_max*)

Assign the same l_max and m_max = l_max to all particles in simulation

5.7.3 utility.cuda

smuthi.utility.cuda.**enable_gpu**(*enable=True*)

Sets the use_gpu flag to enable/disable the use of CUDA kernels.

Parameters **enable** (*bool*) – Set use_gpu flag to this value (default=True).

5.7.4 utility.logging

class smuthi.utility.logging.**Logger**(*log_filename=None*, *log_to_file=True*, *log_to_terminal=True*)

Allows to prompt messages both to terminal and to log file simultaneously. It also allows to print with indentation or to temporally mute the Logger.

fileno()

flush()

write(*message*)

class smuthi.utility.logging.**LoggerIndented**(*indendatation=' '*)

class smuthi.utility.logging.**LoggerMuted**

mute_logger = <smuthi.utility.logging.Logger object>

class smuthi.utility.logging.**bcolors**

BOLD = '\x1b[1m'

ENDC = '\x1b[0m'

FAIL = '\x1b[91m'

HEADER = '\x1b[95m'

OKBLUE = '\x1b[94m'

OKGREEN = '\x1b[92m'

UNDERLINE = '\x1b[4m'

WARNING = '\x1b[93m'

smuthi.utility.logging.**write_blue**(*message*)

smuthi.utility.logging.**write_green**(*message*)

smuthi.utility.logging.**write_header**(*message*)

smuthi.utility.logging.**write_red**(*message*)

5.7.5 utility.math

This module contains several mathematical functions.

smuthi.utility.math.**dx_xh**(*n*, *x*)
Derivative of $xh_n(x)$, where $h_n(x)$ is the spherical Hankel function.

Parameters

- **n** (*int*) – (n>0): Order of spherical Bessel function
- **x** (*array, complex or float*) – Argument for spherical Hankel function

Returns Derivative $\partial_x(xh_n(x))$ as array.

smuthi.utility.math.**dx_xj**(*n*, *x*)
Derivative of $xj_n(x)$, where $j_n(x)$ is the spherical Bessel function.

Parameters

- **n** (*int*) – (n>0): Order of spherical Bessel function
- **x** (*array, complex or float*) – Argument for spherical Bessel function

Returns Derivative $\partial_x(xj_n(x))$ as array.

smuthi.utility.math.**inverse_vector_rotation**(*r*, *alpha=None*, *beta=None*, *gamma=None*, *euler_angles=None*)

smuthi.utility.math.**legendre_normalized**(*ct*, *st*, *lmax*)
Return the normalized associated Legendre function $P_l^m(\cos\theta)$ and the angular functions $\pi_l^m(\cos\theta)$ and $\tau_l^m(\cos\theta)$, as defined in A. Doicu, T. Wriedt, and Y. A. Eremin: "Light Scattering by Systems of Particles", Springer-Verlag, 2006. Two arguments (ct and st) are passed such that the function is valid for general complex arguments, while the branch cuts are defined by the user already in the definition of st.

Parameters

- **ct** (*ndarray*) – cosine of theta (or kz/k)
- **st** (*ndarray*) – sine of theta (or kp/k), need to have same dimension as ct, and st**2+ct**2=1 is assumed
- **lmax** (*int*) – maximal multipole order

Returns

- ndarray plm[l, m, *ct.shape] contains $P_l^m(\cos\theta)$. The entries of the list have same dimension as ct (and st)
- ndarray pilm[l, m, *ct.shape] contains $\pi_l^m(\cos\theta)$.
- ndarray taulm[l, m, *ct.shape] contains $\tau_l^m(\cos\theta)$.

smuthi.utility.math.**legendre_normalized_numbed**

smuthi.utility.math.**nb_wig3jj**(*jj_1*, *jj_2*, *jj_3*, *mm_1*, *mm_2*, *mm_3*)

smuthi.utility.math.**rotation_matrix**(*alpha=None*, *beta=None*, *gamma=None*, *euler_angles=None*)

smuthi.utility.math.**spherical_hankel**(*n*, *x*)

smuthi.utility.math.**vector_rotation**(*r*, *alpha=None*, *beta=None*, *gamma=None*, *euler_angles=None*)

smuthi.utility.math.**wigner_D**(*l*, *m*, *m_prime*, *alpha*, *beta*, *gamma*, *wdsympy=False*)
Computation of Wigner-D-functions for the rotation of a T-matrix

Parameters

- **l** (*int*) – Degree l (1, . . . , lmax)
- **m** (*int*) – Order m (-min(l,mmax),. . . ,min(l,mmax))
- **m_prime** (*int*) – Order m_prime (-min(l,mmax),. . . ,min(l,mmax))
- **alpha** (*float*) – First Euler angle in rad
- **beta** (*float*) – Second Euler angle in rad
- **gamma** (*float*) – Third Euler angle in rad
- **wdsympy** (*bool*) – If True, Wigner-d-functions come from the sympy toolbox

Returns single complex value of Wigner-D-function

smuthi.utility.math.**wigner_d**(*l*, *m*, *m_prime*, *beta*, *wdsympy=False*)
Computation of Wigner-d-functions for the rotation of a T-matrix

Parameters

- **l** (*int*) – Degree l (1, . . . , lmax)
- **m** (*int*) – Order m (-min(l,mmax),. . . ,min(l,mmax))
- **m_prime** (*int*) – Order m_prime (-min(l,mmax),. . . ,min(l,mmax))
- **beta** (*float*) – Second Euler angle in rad
- **wdsympy** (*bool*) – If True, Wigner-d-functions come from the sympy toolbox

Returns real value of Wigner-d-function

5.7.6 utility.memoizing

Provide functionality to store intermediate results in lookup tables (memoize)

class smuthi.utility.memoizing.**Memoize**(*fn*)
To be used as a decorator for functions that are memoized.

5.7.7 utility.optical_constants

Provide functionality to read optical constants in format provided by refractiveindex.info website

smuthi.utility.optical_constants.**read_refractive_index_from_yaml**(*filename*, *vacuum_wavelength*, *units='mkm'*, *kind=1*)
Read optical constants in format provided by refractiveindex.info website.

Parameters

- **filename** (*str*) – path and file name for yaml data downloaded from refractiveindex.info
- **vacuum_wavelength** (*float or np.array*) – wavelengths where refractive index data is needed
- **units** (*str*) – units for wavelength. currently, microns ('mkm' or 'um') and nanometers ('nm') can be selected
- **kind** (*int*) – order of interpolation

Returns A pair (or np.array of pairs) of wavelength and corresponding refractive index (complex)

CHAPTER 6

Literature

Main publication describing SMUTHI (if you use the software for a scientific publication, please cite this):

[Egel et al. 2021] Amos Egel, Krzysztof M Czajkowski, Dominik Theobald, Konstantin Ladutenko, Alexey S Kuznetsov, Lorenzo Pattelli: "SMUTHI: A python package for the simulation of light scattering by multiple particles near or between planar interfaces", Journal of Quantitative Spectroscopy and Radiative Transfer, 273, 2021, 107846, DOI: 10.1016/j.jqsrt.2021.107846

Publications that describe the theory behind Smuthi:

[Theobald 2017] Dominik Theobald, Amos Egel, Guillaume Gomard, Uli Lemmer: "Plane-wave coupling formalism for T-matrix simulations of light scattering by nonspherical particles." Physical Review A 96.3 (2017): 033822.

[Egel 2018] Amos Egel: "Accurate optical simulation of disordered scattering layers for light extractionfrom organic light emitting diodes", Dissertation, Karlsruhe (2018), DOI: 10.5445/IR/1000093961

[Egel and Lemmer 2014] Amos Egel, Uli Lemmer: "Dipole emission in stratified media with multiple spherical scatterers: Enhanced outcoupling from OLEDs", Journal of Quantitative Spectroscopy and Radiative Transfer, 148, 2014, 165-176,DOI: 10.1016/j.jqsrt.2014.06.022

[Egel et al. 2016a] Amos Egel, Siegfried W. Kettlitz, Uli Lemmer. "Efficient evaluation of Sommerfeld integrals for the optical simulation of many scattering particles in planarly layered media." JOSA A 33.4 (2016): 698-706.

[Egel et al. 2016b] Amos Egel, Dominik Theobald, Yidenekachew Donie, Uli Lemmer, Guillaume Gomard, G: "Light scattering by oblate particles near planar interfaces: on the validity of the T-matrix approach." Optics express 24.22 (2016): 25154-25168.

[Egel et al. 2017b] Egel, A., Eremin, Y., Wriedt, T., Theobald, D., Lemmer, U., & Gomard, G. (2017). Extending the applicability of the T-matrix method to light scattering by flat particles on a substrate via truncation of sommerfeld integrals. Journal of Quantitative Spectroscopy and Radiative Transfer, 202, 279-285.

This book describes the Null-Field Method with Discrete Sources (NFM-DS):

[Doicu et al. 2006] Doicu, Adrian, Thomas Wriedt, and Yuri A. Eremin. Light scattering by systems of particles: null-field method with discrete sources: theory and programs. Vol. 124. Springer, 2006.

Other publications to which we refer in this user manual:

[Wiscombe 1980] W.J. Wiscombe: "Improved Mie scattering algorithms", Appl. Opt. 19, 1505-1509 (1980)

[Neves 2012] Antonio A. R. Neves and Dario Pisignano: "Effect of finite terms on the truncation error of Mie series." Optics letters 37.12 (2012): 2418-2420.

Publications that use Smuthi:

[Egel et al. 2017a] Egel, A., Gomard, G., Kettlitz, S. W., & Lemmer, U. (2017). Accurate optical simulation of nano-particle based internal scattering layers for light outcoupling from organic light emitting diodes. Journal of Optics, 19(2), 025605.

[Theobald et al. 2017] Theobald, D., Egel, A., Gomard, G., & Lemmer, U. (2017). Plane-wave coupling formalism for T-matrix simulations of light scattering by nonspherical particles. Physical Review A, 96(3), 033822.

[Warren et al. 2020] Aran Warren, M. Alkaisi and C. Moore, "Design of 2D Plasmonic Diffraction Gratings for Sensing and Super-Resolution Imaging Applications," 2020 IEEE International Instrumentation and Measurement Technology Conference (I2MTC), Dubrovnik, Croatia, 2020, pp. 1-6, doi: 10.1109/I2MTC43012.2020.9129161.

[Theobald et al. 2020] Theobald, D., Yu, S., Gomard, G., & Lemmer, U. (2020). Design of Selective Reflectors Utilizing Multiple Scattering by Core–Shell Nanoparticles for Color Conversion Films. ACS Photonics.

[Czajkowski et al. 2020] Czajkowski, Krzysztof M., Maria Bancerek, and Tomasz J. Antosiewicz. "Multipole analysis of substrate-supported dielectric nanoresonator arrays with T-matrix method." arXiv preprint arXiv:2006.09137 (2020).

[Pidgayko et al. 2020] Pidgayko, D. A., Sadrieva, Z. F., Ladutenko, K. S., & Bogdanov, A. A. (2020). Polarization-controlled selective excitation of Mie resonances of dielectric nanoparticle on a coated substrate. arXiv preprint arXiv:2011.06494.

[Warren et al. 2021] Aran Warren, Maan M. Alkaisi, and Ciaran P. Moore. "Finite-size and disorder effects on 1D unipartite and bipartite surface lattice resonances", Opt. Express 30, 3302-3315 (2022), DOI: 10.1364/OE.445414

Python Module Index

S

Index

D

E

F

G

H

I

K

L

R

S

T

U

V

W