# MacroMax

Tom Vettenburg

Apr 08, 2023

# CONTENTS:

CHAPTER

ONE

# INTRODUCTION

This Python 3 package enables solving the macroscopic Maxwell equations in complex dielectric materials.

The material properties are defined on a rectangular grid (1D, 2D, or 3D) for which each voxel defines an isotropic or anisotropic permittivity. Optionally, a heterogeneous (anisotropic) permeability as well as bi-anisotropic coupling factors may be specified (e.g. for chiral media). The source, such as an incident laser field, is specified as an oscillating current-density distribution.

The method iteratively corrects an estimated solution for the electric field (default: all zero). Its memory requirements are on the order of the storage requirements for the material properties and the electric field within the calculation volume. Full details can be found in the open-access manuscript "Calculating coherent light-wave propagation in large heterogeneous media". Automatic leveraging of detected GPU/Cloud is implemented using PyTorch (for further details follow this link).

Examples of usage can be found in the examples/ sub-folder. The Complete MacroMax Documentation can be found at https://macromax.readthedocs.io. All source code is available on GitHub under the **MIT License: https://opensource.org/licenses/MIT**

CHAPTER

TWO

# INSTALLATION

## 2.1 Prerequisites

This library requires Python 3 with the `numpy` and `scipy` packages for the main calculations. These modules will be automatically installed. The modules `multiprocessing`, `torch`, `pyfftw`, and `mkl-fft` (Intel(R) CPU specific) can significantly speed up the calculations.

The examples require `matplotlib` for displaying the results. In the creation of this package for distribution, the `pypandoc` package is used for translating this document to other formats. This is only necessary for software development.

The code has been tested on Python 3.7 and 3.10, though it is expected to work on versions 3.6 and above.

## 2.2 Installing

Installing the `macromax` package and its mandatory dependencies is as straightforward as running the following command in a terminal:

```
pip install macromax
```

While this is sufficient to get started, optional packages are useful to display the results and to speed-up the calculations.

### 2.2.1 Optimizing execution speed on a CPU

The calculation time can be reduced to a fraction by ensuring that you have the fastest libraries installed for your system. In particular the FFTW library can easily halve the calculation time on a CPU:

```
pip install macromax pyFFTW
```

On some systems the pyFFTW Python package requires the separate installation of the FFTW library; however, it is easy to install it using Anaconda with the commands: `conda install fftw`, or on Debian-based systems with `sudo apt-get install fftw`.

Alternatively, the mkl-fft package is available for Intel(R) CPUs, though it may require compilation or relying on the Anaconda or Intel Python distributions:

```
conda install -c intel intelpython
```

### 2.2.2 Leveraging a machine learning framework for GPU and cloud-based calculations

The calculation time can be reduced by several orders of magnitude using the PyTorch machine learning library. This can be as straightforward as using the appropriate runtime on Google Colab. The MacroMax library can here be installed by prepending the command with an exclamation mark as follows:

```
!pip install macromax
```

Local GPUs can also be used provided that PyTorch has a compatible implementation. At the time of writing, these includes NVidia's CUDA-enabled GPU as well AMD's ROCm-enabled GPUs (on Linux). Prior to installing the PyTorch module following the PyTorch Guide, install the appropriate CUDA or ROCm drivers for your GPU. Note that for PyTorch to work correctly, Nvidia drivers need to be up to date and match the installed CUDA version. At the time of writing, for CUDA version 11.6, PyTorch can be installed as follows using pip:

```
pip install torch --extra-index-url https://download.pytorch.org/whl/cu116
```

Specifics for your CUDA version and operating system are listed on PyTorch Guide.

When PyTorch and a compatible GPU are detected, these will be used by default. If not, FFTW and mkl-fft will be used if available. Otherwise, NumPy and SciPy will be used as a fallback. The default backend can be set at the start of your code, or by creating a text file named `backend_config.json` in the current working directory with contents as:

```
[
  {"type": "torch", "device": "cuda"},
  {"type": "numpy"}
]
```

to choose PyTorch when a compatible GPU is available, and NumPy otherwise. Although this machine learning library can be used without a hardware accelerator, we found that NumPy (with FFTW) can be faster when no GPU is available. This backend selection rule is used by default.

Experimental support exists for alternative backend implementations such as TensorFlow. Please refer to the source code for the latest work in progress. Pull requests are always welcome.

### 2.2.3 Additional packages

The package comes with a submodule containing example code that should run as-is on most desktop installations of Python. Some systems may require the installation of the ubiquitous `matplotlib` graphics library:

```
pip install matplotlib
```

The output logs can be colored by installing the coloredlogs packaged:

```
pip install coloredlogs
```

Building and distributing the library may require further packages as indicated below.

CHAPTER

# THREE

# USAGE

The basic calculation procedure consists of the following steps:

1. define the material
2. define the coherent light source
3. call `solution = macromax.solve(...)`
4. display the solution

The `macromax` package must be imported to be able to use the `solve` function. The package also contains several utility functions that may help in defining the property and source distributions.

Examples can be found in the examples package in the examples/ folder. Ensure that the entire `examples/` folder is downloaded, including the `__init__.py` file with general definitions. Run the examples from the parent folder using e.g. `python -m examples.air_glass_air_1D`.

The complete functionality is described in the Library Reference Documentation at https://macromax.readthedocs.io.

## 3.1 Loading the Python 3 package

The `macromax` package can be imported using:

```
import macromax
```

**Optional:** If the package is installed without a package manager, it may not be on Python's search path. If necessary, add the library to Python's search path, e.g. using:

```
import sys
import os
sys.path.append(os.path.dirname(os.getcwd()))
```

Reminder: this library requires Python 3, `numpy`, and `scipy`. Optionally, `pyfftw` can be used to speed up the calculations. The examples also require `matplotlib`.

## 3.2 Specifying the material

### 3.2.1 Defining the sampling grid

The material properties are sampled on a plaid uniform rectangular grid of voxels. The sample points are defined by one or more linearly increasing coordinate ranges, one range per dimensions. The coordinates must be specified in meters, e.g.:

```
import numpy as np
x_range = 50e-9 * np.arange(1000)
```

Ranges for multiple dimensions can be passed to `solve(...)` as a tuple of ranges: `ranges = (x_range, y_range)`, or the convenience object `Grid` in the `macromax.utils.array` sub-package. The latter can be used as follows:

```
data_shape = (200, 400)
sample_pitch = 50e-9  # or (50e-9, 50e-9)
grid = macromax.Grid(data_shape, sample_pitch)
```

This defines a uniformly spaced plaid grid, centered around the origin, unless specified otherwise.

### 3.2.2 Defining the material property distributions

The material properties are defined by ndarrays of 2+N dimensions, where N can be up to 3 for three-dimensional samples. In each sample point, or voxel, a complex 3x3 matrix defines the anisotropy at that point in the sample volume. The first two dimensions of the ndarray are used to store the 3x3 matrix, the following dimensions are the spatial indices x, y, and z. Optionally, four complex ndarrays can be specified: `epsilon`, `mu`, `xi`, and `zeta`. These ndarrays represent the permittivity, permeability, and the two coupling factors, respectively.

When the first two dimensions of a property are found to be both a singleton, i.e. 1x1, that property is assumed to be isotropic. Similarly, singleton spatial dimensions are interpreted as homogeneity in that property. The default permeability `mu` is 1, and the coupling constants are zero by default.

#### Boundary conditions

The underlying algorithm assumes periodic boundary conditions. Alternative boundary conditions can be implemented by surrounding the calculation area with absorbing (or reflective) layers. Back reflections can be suppressed by e.g. linearly increasing the imaginary part of the permittivity with depth into a boundary with a thickness of a few wavelengths.

## 3.3 Defining the source

The coherent source is defined by as a spatially-variant free current density. Although the current density may be non-zero in all of space, it is more common to define a source at one of the edges of the volume, to model e.g. an incident laser beam; or even as a single voxel, to simulate a dipole emitter. The source density can be specified as a complex number, indicating the phase and amplitude of the current at each point. If an extended source is defined, care should be taken so that the source currents constructively interfere in the desired direction. I.e. the current density at neighboring voxels should have a phase difference matching the k-vector in the background medium. Optionally, instead of a current density, the internally-used source distribution may be specified directly. It is related to the current density as follows: `S = i omega mu_0 J` with units of rad s^-1 H m^-1 A m^-2 = rad V m^-3, where `omega` is the angular frequency, and `mu_0` is the vacuum permeability, mu_0.

The source distribution is stored as a complex ndarray with 1+N dimensions. The first dimension contains the current 3D direction and amplitude for each voxel. The complex argument indicates the relative phase at each voxel.

## 3.4 Calculating the electromagnetic light field

Once the `macromax` module is imported, the solution satisfying the macroscopic Maxwell's equations is calculated by calling:

```
solution = macromax.solve(...)
```

The function arguments to `macromax.solve(...)` can be the following:

- `grid|x_range`: A Grid object, a vector (1D), or tuple of vectors (2D, or 3D) indicating the spatial coordinates of the sample points. Each vector must be a uniformly increasing array of coordinates, sufficiently dense to avoid aliasing artefacts.
- `vacuum_wavelength|wave_number|anguler_frequency`: The wavelength in vacuum of the coherent illumination in units of meters.
- `current_density` or `source_distribution`: An ndarray of complex values indicating the source value and direction at each sample point. The source values define the free current density in the sample. The first dimension contains the vector index, the following dimensions contain the spatial dimensions. If the source distribution is not specified, it is calculated as $-ick0mu_0 J$, where `i` is the imaginary constant, `c`, `k0`, and `mu_0`, the light-speed, wavenumber, and permeability in vacuum. Finally, `J` is the free current density (excluding the movement of bound charges in a dielectric), specified as the input argument current_density. These input arguments should be `numpy.ndarray`s with a shape as specified by the `grid` input argument, or have one extra dimension on the left to indicate the polarization. If polarization is not specified the solution to the *scalar* wave equation is calculated. However, when polarization is specified the *vectorial* problem is solved. The returned `macromax.Solution` object has the property `vectorial` to indicate whether polarization is accounted for or not.
- `refractive_index`: A complex `` `numpy.ndarray` `` of a shape as indicated by the *grid* argument. Each value indicates the refractive at the corresponding spatial grid point. Real values indicate a loss-less material. A positive imaginary part indicates the absorption coefficient, $\kappa$. This input argument is not required if the permittivity, `epsilon` is specified.
- `epsilon`: (optional, default: $n^2$) A complex `numpy.ndarray` of a shape as indicated by the `grid` argument for *isotropic* media, or a shape with two extra dimensions on the left to indicate *anisotropy/birefringence*. The array values indicate the relative permittivity at all sample points in space. The optional two first (left-most) dimensions may contain a 3x3 matrix at each spatial location to indicate the anisotropy/birefringence. By default the 3x3 identity matrix is assumed, scaled by the scalar value of the array without the first two dimensions. Real values indicate loss-less permittivity. This input argument is unit-less, it is relative to the vacuum permittivity.

Optionally one can also specify magnetic and coupling factors:

- `mu`: A complex ndarray that defines the 3x3 permeability matrix at all sample points. The first two dimensions contain the matrix indices, the following dimensions contain the spatial dimensions.
- `xi` and `zeta`: Complex ndarray that define the 3x3 coupling matrices at all sample points. This may be useful to model chiral materials. The first two dimensions contain the matrix indices, the following dimensions contain the spatial dimensions.

It is often useful to also specify a callback function that tracks progress. This can be done by defining the `callback`-argument as a function that takes an intermediate solution as argument. This user-defined callback function can display the intermediate solution and check if the convergence is adequate. The callback function should return `True` if more iterations are required, and `False` otherwise. E.g.:

```
callback=lambda s: s.residue > 0.01 and s.iteration < 1000
```

will iterate until the residue is at most 1% or until the number of iterations reaches 1,000.

The solution object (of the Solution class) fully defines the state of the iteration and the current solution as described below.

The `macromax.solve(...)` function returns a solution object. This object contains the electric field vector distribution as well as diagnostic information such as the number of iterations used and the magnitude of the correction applied in the last iteration. It can also calculate the displacement, magnetizing, and magnetic fields on demand. These fields can be queried as follows:

- `solution.E`: Returns the electric field distribution.
- `solution.H`: Returns the magnetizing field distribution.
- `solution.D`: Returns the electric displacement field distribution.
- `solution.B`: Returns the magnetic flux density distribution.
- `solution.S`: The Poynting vector distribution in the sample.

The field distributions are returned as complex `numpy` ndarrays in which the first dimensions is the polarization or direction index. The following dimensions are the spatial dimensions of the problem, e.g. x, y, and z, for three-dimensional problems.

The solution object also keeps track of the iteration itself. It has the following diagnostic properties:

- `solution.iteration`: The number of iterations performed.
- `solution.residue`: The relative magnitude of the correction during the previous iteration. and it can be used as a Python iterator.

Further information can be found in the examples and the signatures of each function and class.

## 3.5 Complete Example

The following code loads the library, defines the material and light source, calculates the result, and displays it. To keep this example as simple as possible, the calculation is limited to one dimension. Higher dimensional calculations simply require the definition of the material and light source in 2D or 3D.

The first section of the code loads the `macromax` library module as well as its `utils` submodule. More

```
import macromax

import numpy as np
import matplotlib.pyplot as plt
# %matplotlib notebook  # Uncomment this line in an iPython Jupyter notebook

#
# Define the material properties
#
wavelength = 500e-9  # [ m ] In SI units as everything else here
source_polarization = np.array([0, 1, 0])[:, np.newaxis]  # y-polarized

# Set the sampling grid
nb_samples = 1024
sample_pitch = wavelength / 10  # [ m ]  # Sub-sample for display
```

(continues on next page)

(continued from previous page)

```
boundary_thickness = 5e-6  # [ m ]
x_range = sample_pitch * np.arange(nb_samples) - boundary_thickness  # [ m ]

# Define the medium as a spatially-variant permittivity
# Don't forget absorbing boundary:
dist_in_boundary = np.maximum(0, np.maximum(-x_range,
                                            x_range - (x_range[-1] - boundary_thickness)
                                            ) / boundary_thickness)
permittivity = 1.0 + 0.25j * dist_in_boundary  # unit-less, relative to vacuum␣
↪permittivity
# glass has a refractive index of about 1.5
permittivity[(x_range >= 20e-6) & (x_range < 30e-6)] += 1.5**2
permittivity = permittivity[np.newaxis, np.newaxis, ...]  # Define an isotropic material

#
# Define the illumination source
#
# point source at x = 0
current_density = source_polarization * (np.abs(x_range) < sample_pitch/4)

#
# Solve Maxwell's equations
#
# (the actual work is done in this line)
solution = macromax.solve(x_range, vacuum_wavelength=wavelength,
                          current_density=current_density, epsilon=permittivity)

#
# Display the results
#
fig, ax = plt.subplots(2, 1, frameon=False, figsize=(8, 6))

x_range = solution.grid[0]  # coordinates
E = solution.E[1, :]  # Electric field in y
H = solution.H[2, :]  # Magnetizing field in z
S = solution.S_forw[0, :]  # Poynting vector in x
f = solution.f[0, :]  # Optical force in x
# Display the field for the polarization dimension
field_to_display = E
max_val_to_display = np.amax(np.abs(field_to_display))
poynting_normalization = np.amax(np.abs(S)) / max_val_to_display
ax[0].plot(x_range * 1e6,
           np.abs(field_to_display) ** 2 / max_val_to_display,
           color=[0, 0, 0])
ax[0].plot(x_range * 1e6, np.real(S) / poynting_normalization,
           color=[1, 0, 1])
ax[0].plot(x_range * 1e6, np.real(field_to_display),
           color=[0, 0.7, 0])
ax[0].plot(x_range * 1e6, np.imag(field_to_display),
           color=[1, 0, 0])
figure_title = "Iteration %d, " % solution.iteration
ax[0].set_title(figure_title)
```

(continues on next page)

(continued from previous page)

```
ax[0].set_xlabel("x  [$\mu$m]")
ax[0].set_ylabel("I, E  [a.u., V/m]")
ax[0].set_xlim(x_range[[0, -1]] * 1e6)

ax[1].plot(x_range[-1] * 2e6, 0,
           color=[0, 0, 0], label='I')
ax[1].plot(x_range[-1] * 2e6, 0,
           color=[1, 0, 1], label='$S_{real}$')
ax[1].plot(x_range[-1] * 2e6, 0,
           color=[0, 0.7, 0], label='$E_{real}$')
ax[1].plot(x_range[-1] * 2e6, 0,
           color=[1, 0, 0], label='$E_{imag}$')
ax[1].plot(x_range * 1e6, permittivity[0, 0].real,
           color=[0, 0, 1], label='$\epsilon_{real}$')
ax[1].plot(x_range * 1e6, permittivity[0, 0].imag,
           color=[0, 0.5, 0.5], label='$\epsilon_{imag}$')
ax[1].set_xlabel('x  [$\mu$m]')
ax[1].set_ylabel('$\epsilon$')
ax[1].set_xlim(x_range[[0, -1]] * 1e6)
ax[1].legend(loc='upper right')

plt.show(block=True)  # Not needed for iPython Jupyter notebook
```

## 3.6 Optimization of time and memory efficiency

Electromagnetic calculations tend to test the limits of the hardware. Two factors should be considered when optimizing the calculation: computation and memory. Naturally, the number of operations and the duration of each operation should be minimized. However, the latter is often dominated by memory accesses and copying of arrays. The memory usage therefore does not only affect the size of the problems that can be solved, it also tends to have an important impact on the total calculation time.

A straightforward method to reduce memory usage is to switch from 128-bit precision complex numbers to 64-bit. By default, the precision of the source_density is used, which is typically `np.complex128` or its real equivalent. The `Solution`'s default `dtype` can be overridden by specifying it as `solve(... dtype=np.complex64)`. Halving the storage requirements can eliminate additional copies between the main memory and CPU cache. In extreme cases it can also avoid swapping. Lower precision math also executes faster on many architectures.

While oversampling to less than 1/10th of the wavelength may aid visualization, it is often sufficient to sample at a quarter of the wavelength. The sample solution represents a sinc-interpolated continuous function. The final result can be visualized with arbitrary resolution using interpolation.

The number of operations can be kept to a minimum by:

- using non-magnetic and non-chiral materials,
- using isotropic materials,
- limiting the largest difference in permittivity (including the absorbing boundary), and
- using a scalar approximation whenever possible.

Optimization of the implementation is another route to consider. Potentially areas of improvement are:

- Profiling of memory usage and elimination of redundant temporary copies

- In-place fast-Fourier transforms. When available, the FFTW library is used; however, the drop-in fft and ifft replacements are used at the moment.
- Moving the calculations to a GPU or a cloud-computing environment. Since the copying-overheads may quickly become a bottleneck, it is important to consider the memory requirements for the problem you want to solve.

CHAPTER

# FOUR

# DEVELOPMENT

The Library API Documentation can be found at https://macromax.readthedocs.io.

## 4.1 Source code organization

The source code is organized as follows:

- / (root): Module description and distribution files.
- macromax/: The iterative solver.
  - macromax/utils/: Helper functionality used in the solver and to use the solver.
- examples/: Examples of how the solver can be used.
- tests/: Automated unit tests of the solver's functionality. Use this after making modifications to the solver and extend it if new functionality is added.

The library functions are contained in `macromax/`:

- solver: Defines the `solve(...)` function and the `Solution` class.
- backend: Defines linear algebra functions to work efficiently with large arrays of 3x3 matrices and 3-vectors.
- utils/: Defines utility functions that can be used to prepare and interpret function arguments.

The included examples in the examples/ folder are:

- notebook_example.ipynb: An iPython notebook demonstrating basic usage of the library.
- air_glass_air_1D.py: Calculation of the back reflection from an air-glass interface (one-dimensional calculation)
- air_glass_air_2D.py: Calculation of the refraction and reflection of light hitting a glass window at an angle (two-dimensional calculation)
- birefringent_crystal.py: Demonstration of how an anisotropic permittivity can split a diagonally polarized Gaussian beam into ordinary and extraordinary beams.
- polarizer.py: Calculation of light wave traversing a set of two and a set of three polarizers as a demonstration of anisotropic absorption (non-Hermitian permittivity)
- rutile.py: Scattering from disordered collection of birefringent rutile (TiO2) particles.
- benchmark.py: Timing of a simple two-dimensional calculation for comparison between versions.

## 4.2 Testing

Unit tests are contained in `macromax/tests`. The `BackEnd` class in `backend.py` is well covered and specific tests have been written for the `Solution` class in `solver.py`.

To run the tests, make sure that the `pytest` package is installed, and run the following commands from the `Macromax/python/` directory:

```
pip install pytest
pytest --ignore=tests/test_backend_tensorflow.py
```

The benchmark script in the `examples/` folder can be used to compare performance for different problems and architectures. Performance issues can be debugged using profilers as `pprofile` and `memory_profiler`, installed with:

```
pip install pprofile memory_profiler
```

## 4.3 Documentation

The `make` scripts in the `docs/` subdirectory automatically generate the documentation. This uses Sphinx and extensions that can be installed with

```
pip install sphinx==5.3.0 sphinx_autodoc_typehints sphinxcontrib_mermaid sphinx-rtd-
↪theme recommonmark
```

Examples of use can be found in the `examples/` and `tests/` folders. The former is more didactic, while the latter is more complete.

## 4.4 Building and Distributing

The source code consists of pure Python 3, hence only packaging is required for distribution. A package is generated by `setup.py`, which relies on the `m2r2` package to translate `.md` files to `.rst` format:

```
pip install m2r2==0.3.2  # for compatibility with sphinx-rtd-theme, which requires an␣
↪older version of docutils.
```

To prepare a package for distribution, increase the `__version__` number in macromax/__init__.py, and run:

```
pip install build --upgrade
python -m build -nsw
pip install . --upgrade
```

The second line installs the newly-forged `macromax` package for testing.

The package can then be uploaded to a test repository as follows:

```
pip install twine --upgrade
twine upload -r testpypi dist/*
```

Installing from the test repository is done as follows:

```
pip install -i https://test.pypi.org/simple/ macromax --upgrade
```

To facilitate importing the code, IntelliJ IDEA/PyCharm project files can be found in `MacroMax/python/`: `MacroMax/python/python.iml` and the folder `MacroMax/python/.idea`.

### 4.4.1 macromax package

The *solver* module calculates the solution to the wave equations. More specifically, the work is done in the iteration defined in the *Solution.__iter__()* method of the *Solution* class. The convenience function *solve()* is provided to construct a *Solution* object and iterate it to convergence using its *Solution.solve()* method.

Public attributes:

`__version__`: The MacroMax version number as a *str*.

*solve()*: The function to solve the wave problem.

*Solution*: The class that is used by the *solve()* function, which can be used for fine-control of the iteration or re-use.

*Grid*: A class representing uniformly spaced Cartesian grids and their Fourier Transforms.

`log`: The `logging` object of the *macromax* library. This can be used to make the messages more or less verbose.

*backend*: The sub-package with the back-end specifications.

`macromax.solve`(*grid*, *vectorial=None*, *wavenumber=1.0*, *angular_frequency=None*, *vacuum_wavelength=None*, *current_density=None*, *source_distribution=None*, *epsilon=None*, *xi=0.0*, *zeta=0.0*, *mu=1.0*, *refractive_index=None*, *bound=None*, *initial_field=0.0*, *dtype=None*, *callback=<function <lambda>>*)

Function to find a solution for Maxwell's equations in a media specified by the epsilon, xi, zeta, and mu distributions in the presence of a current source.

**Parameters**

- **grid** (Union[*Grid*, Sequence, ndarray]) – A Grid object or a Sequence of vectors with uniformly increasing values that indicate the positions in a plaid grid of sample points for the material and solution. In the one-dimensional case, a simple increasing Sequence of uniformly-spaced numbers may be provided as an alternative. The length of the ranges determines the data_shape, to which the source_distribution, epsilon, xi, zeta, mu, and initial_field must broadcast when specified as ndarrays.
- **vectorial** (Optional[bool]) – a boolean indicating if the source and solution are 3-vectors-fields (True) or scalar fields (False).
- **wavenumber** (Optional[Real]) – the wavenumber in vacuum = 2 pi / vacuum_wavelength. The wavelength in the same units as used for the other inputs/outputs.
- **angular_frequency** (Optional[Real]) – alternative argument to the wavenumber = angular_frequency / c
- **vacuum_wavelength** (Optional[Real]) – alternative argument to the wavenumber = 2 pi / vacuum_wavelength
- **current_density** (Union[Complex, Sequence, ndarray, None]) – (optional, instead of source_distribution) An array or function that returns the free (vectorial) current density input distribution, J. The free current density has units of $Am^{-}2$.
- **source_distribution** (Union[Complex, Sequence, ndarray, None]) – (optional, instead of current_density) An array or function that returns the (vectorial) source input wave distribution. The source values relate to the current density, J, as 1j * angular_frequency *

scipy.constants.mu_0 * J and has units of $rads^-1Hm^-1Am^-2 = radVm^-3$. More general, non-electro-magnetic wave problems can be solved using the source_distribution, as it does not rely on the vacuum permeability constant, $mu_0$.

- **epsilon** (Union[Complex, Sequence, ndarray, None]) – an array or function that returns the (tensor) epsilon that represents the permittivity at the points indicated by the grid specified as its input arguments.
- **xi** (Union[Complex, Sequence, ndarray]) – an array or function that returns the (tensor) xi for bi-(an)isotropy at the points indicated by the grid specified as its input arguments.
- **zeta** (Union[Complex, Sequence, ndarray]) – an array or function that returns the (tensor) zeta for bi-(an)isotropy at the points indicated by the grid specified as its input arguments.
- **mu** (Union[Complex, Sequence, ndarray]) – an array or function that returns the (tensor) permeability at the points indicated by the grid specified as its input arguments.
- **refractive_index** (Union[Complex, Sequence, ndarray, None]) – an array or function that returns the (complex) (tensor) refractive_index, as the square root of the permittivity, at the points indicated by the *grid* input argument.
- **bound** (Optional[*Bound*]) – An object representing the boundary of the calculation volume. Default: None, PeriodicBound(grid)
- **initial_field** (Union[Complex, Sequence, ndarray]) – optional start value for the E-field distribution (default: all zero E)
- **dtype** – optional numpy datatype for the internal operations and results. This must be a complex number type as numpy.complex128 or np.complex64.
- **callback** (Callable) – optional function that will be called with as argument this solver. This function can be used to check and display progress. It must return a boolean value of True to indicate that further iterations are required.

**Returns**

The Solution object that has the E and H fields, as well as iteration information.

**class** macromax.**Solution**(*grid*, *vectorial=None*, *wavenumber=1.0*, *angular_frequency=None*, *vacuum_wavelength=None*, *current_density=None*, *source_distribution=None*, *epsilon=None*, *xi=0.0*, *zeta=0.0*, *mu=1.0*, *refractive_index=None*, *bound=None*, *initial_field=0.0*, *dtype=None*)

Bases: object

**__init__**(*grid*, *vectorial=None*, *wavenumber=1.0*, *angular_frequency=None*, *vacuum_wavelength=None*, *current_density=None*, *source_distribution=None*, *epsilon=None*, *xi=0.0*, *zeta=0.0*, *mu=1.0*, *refractive_index=None*, *bound=None*, *initial_field=0.0*, *dtype=None*)

Class a solution that can be further iterated towards a solution for Maxwell's equations in a media specified by the epsilon, xi, zeta, and mu distributions.

**Parameters**

- **grid** (Union[*Grid*, Sequence, ndarray]) – A Grid object or a Sequence of vectors with uniformly increasing values that indicate the positions in a plaid grid of sample points for the material and solution. In the one-dimensional case, a simple increasing Sequence of uniformly-spaced numbers may be provided as an alternative. The length of the ranges determines the data_shape, to which the source_distribution, epsilon, xi, zeta, mu, and initial_field must broadcast when specified as `numpy.ndarray`s.
- **vectorial** (Optional[bool]) – a boolean indicating if the source and solution are 3-vectors-fields (True) or scalar fields (False). Default: True, when vectorial nor the source is

specified. Default: vectorial (True), unless the source field is scalar (False if first dimension is a singleton dimension).

- **wavenumber** (Optional[Real]) – the wavenumber in vacuum = 2pi / vacuum_wavelength. The wavelength in the same units as used for the other inputs/outputs.
- **angular_frequency** (Optional[Real]) – alternative argument to the wavenumber = angular_frequency / c
- **vacuum_wavelength** (Optional[Real]) – alternative argument to the wavenumber = 2 pi / vacuum_wavelength
- **current_density** (Union[Complex, Sequence, ndarray, None]) – (optional, instead of source_distribution) An array or function that returns the (vectorial) current density input distribution, J. The current density has units of $Am^-2$.
- **source_distribution** (Union[Complex, Sequence, ndarray, None]) – (optional, instead of current_density) An array or function that returns the (vectorial) source input wave distribution. The source values relate to the current density, J, as 1j * angular_frequency * scipy.constants.mu_0 * J and has units of $rads^-1Hm^-1Am^-2 = radVm^-3$. More general, non-electro-magnetic wave problems can be solved using the source_distribution, as it does not rely on the vacuum permeability constant, $mu_0$.
- **epsilon** (Union[Complex, Sequence, ndarray, None]) – an array or function that returns the (tensor) epsilon that represents the permittivity at the points indicated by the *grid* input argument.
- **xi** (Union[Complex, Sequence, ndarray]) – an array or function that returns the (tensor) xi for bi-(an)isotropy at the points indicated by the *grid* input argument.
- **zeta** (Union[Complex, Sequence, ndarray]) – an array or function that returns the (tensor) zeta for bi-(an)isotropy at the points indicated by the *grid* input argument.
- **mu** (Union[Complex, Sequence, ndarray]) – an array or function that returns the (tensor) permeability at the points indicated by the *grid* input argument.
- **refractive_index** (Union[Complex, Sequence, ndarray, None]) – an array or function that returns the (complex) (tensor) refractive_index, as the square root of the permittivity, at the points indicated by the *grid* input argument.
- **bound** (Optional[*Bound*]) – An object representing the boundary of the calculation volume. Default: None, PeriodicBound(grid)
- **initial_field** (Union[Complex, Sequence, ndarray]) – optional start value for the E-field distribution (default: all zero E)
- **dtype** – optional numpy datatype for the internal operations and results. This must be a complex number type as numpy.complex128 or np.complex64.

**property grid:** *Grid*

The sample positions of the plaid sampling grid. This may be useful for displaying result axes.

**Returns**

A Grid object representing the sample points of the fields and material.

**property vectorial:** bool

Boolean to indicates whether calculations happen on vectorial (True) or scalar (False) fields.

**property dtype**

The numpy equivalent data type used in the calculation. This is either np.complex64 or np.complex128.

**property wavenumber:** Real

The vacuum wavenumber, $k_0$, used in the calculation.

**Returns**
A scalar indicating the wavenumber used in the calculation.

**property angular_frequency:** Real

The angular frequency, $\omega$, used in the calculation.

**Returns**
A scalar indicating the angular frequency used in the calculation.

**property wavelength:** Real

The vacuum wavelength, $\lambda_0$, used in the calculation.

**Returns**
A scalar indicating the vacuum wavelength used in the calculation.

**property magnetic:** bool

Indicates if this media is considered magnetic.

**Returns**
A boolean, True when magnetic, False otherwise.

**property bound:** *Bound*

The Bound object that defines the calculation boundaries.

**property source_distribution:** ndarray

The source distribution, i k0 mu_0 times the current density j.

**Returns**
A complex array indicating the amplitude and phase of the source vector field. The dimensions of the array are [1|3, self.grid.shape], where the first dimension is 1 in case of a scalar field, and 3 in case of a vector field.

**property j:** ndarray

The free current density, j, of the source vector field.

**Returns**
A complex array indicating the amplitude and phase of the current density vector field [A m^-2]. The dimensions of the array are [1|3, self.grid.shape], where the first dimension is 1 in case of a scalar field, and 3 in case of a vector field.

**property E:** ndarray

The electric field for every point in the sample space (SI units).

**Returns**
A vector array with the first dimension containing Ex, Ey, and Ez,

while the following dimensions are the spatial dimensions.

**property B:** ndarray

The magnetic field for every point in the sample space (SI units). This is calculated from H and E.

**Returns**
A vector array with the first dimension containing $B_x, B_y, and B_z$, while the following dimensions are the spatial dimensions.

**property D: ndarray**

The displacement field for every point in the sample space (SI units). This is calculated from E and H.

**Returns**

A vector array with the first dimension containing $D_x, D_y, and D_z$, while the following dimensions are the spatial dimensions.

**property H: ndarray**

The magnetizing field for every point in the sample space (SI units). This is calculated from E.

**Returns**

A vector array with the first dimension containing $H_x, H_y, and H_z$, while the following dimensions are the spatial dimensions.

**property S: ndarray**

The time-averaged Poynting vector for every point in space. :return: A vector array with the first dimension containing $S_x, S_y, and S_z$, while the following dimensions are the spatial dimensions.

**property energy_density: ndarray**

Returns the energy density, u.

**Returns**

A real array indicating the energy density in space.

**property stress_tensor: ndarray**

Maxwell's stress tensor for every point in space.

**Returns**

A real and symmetric matrix-array with the stress tensor for every point in space. The units are $N/m^2$.

**property f: ndarray**

The electromagnetic force density (force per SI unit volume, not per voxel).

**Returns**

A vector array representing the electro-magnetic force exerted per unit volume. The first dimension contains $f_x, f_y, and f_z$, while the following dimensions are the spatial dimensions. The units are $N/m^3$.

**property torque: ndarray**

The electromagnetic force density (force per SI unit volume, not per voxel).

**Returns**

A vector array representing the electro-magnetic torque exerted per unit volume. The first dimension contains torque_x, torque_y, and torque_z, while the following dimensions are the spatial dimensions. The units are $Nm/m^3 = Nm^{-2}$.

**property iteration: int**

The current iteration number.

**Returns**

An integer indicating how many iterations have been done.

**property previous_update_norm: Real**

The L2-norm of the last update, the difference between current and previous E-field.

**Returns**

A positive scalar indicating the norm of the last update.

**property residue:** Real

Returns the current relative residue of the inverse problem $E = H^{-1}S$. The relative residue is return as the l2-norm fraction $||E - H^{-1}S||/||E||$, where H represents the vectorial Helmholtz equation following Maxwell's equations and S the current density source. The solver searches for the electric field, E, that minimizes the preconditioned inverse problem.

**Returns**

A non-negative real scalar that indicates the change in E with the previous iteration normalized to the norm of the current E.

**__iter__**()

Returns an iterator that on __next__() yields this Solution after updating it with one cycle of the algorithm. Obtaining this iterator resets the iteration counter.

Usage:

```
for solution in Solution(...):
    if solution.iteration > 100:
        break
print(solution.residue)
```

**solve**(*callback=<function Solution.<lambda>>*)

Runs the algorithm until the convergence criterion is met or until the maximum number of iterations is reached.

**Parameters**

**callback** (Callable) – optional callback function that overrides the one set for the solver. E.g. callback=lambda s: s.iteration < 100

**Returns**

This Solution object, which can be used to query e.g. the final field E using Solution.E.

**class** macromax.**ScatteringMatrix**(*grid*, *vectorial=True*, *wavenumber=None*, *angular_frequency=None*, *vacuum_wavelength=None*, *epsilon=None*, *xi=0.0*, *zeta=0.0*, *mu=1.0*, *refractive_index=None*, *bound=None*, *dtype=None*, *callback=<function ScatteringMatrix.<lambda>>*, *caching=True*, *array=None*)

Bases: *LiteralScatteringMatrix*

A class representing scattering matrices.

**__init__**(*grid*, *vectorial=True*, *wavenumber=None*, *angular_frequency=None*, *vacuum_wavelength=None*, *epsilon=None*, *xi=0.0*, *zeta=0.0*, *mu=1.0*, *refractive_index=None*, *bound=None*, *dtype=None*, *callback=<function ScatteringMatrix.<lambda>>*, *caching=True*, *array=None*)

Construct a scattering matrix object for a medium specified by a refractive index distribution or the corresponding epsilon, xi, zeta, and mu distributions. Each electromagnetic field distribution entering the material is scattered into a certain electromagnetic field distributions propagating away from it from both sides. The complex matrix relates the amplitude and phase of all N propagating input modes to all N propagating output modes. No scattering, as in vacuum, is indicated by the NxN identity matrix. There are N/2 input and output modes on either side of the scattering material. Mode i and i+N/2 correspond to plane wave traveling in opposing directions. The mode directions are taken in raster-scan order and only propagating modes are included. When polarization is considered, the modes come in pairs corresponding to two orthogonal linear polarizations.

The modes are encoded as a vector of length N = 2xMxP for 2 sides, M angles, and P polarizations.

- First the N/2 modes propagating along the positive x-axis are considered, then those propagating in the reverse direction.

- In each direction, M different angles (k-vectors) can be considered. We choose propagating modes on a uniformly-spaced plaid grid that includes the origin (corresponding to the k-vector along the x-axis). Modes not propagating along the x-axis, i.e. in the y-z-plane are not considered. The angles are ordered in raster-scan order from negative k_y to positive k_y (slow) and from negative k_z to positive k_z (fast). The grid axes dimensions correspond to x(0), y(1), z(2).
- When polarization is considered, each angle has a pair of modes, one for each polarization. The first mode has the polarization oriented along the rotated y'-axis and the second mode along the rotated z'-axis. To avoid ambiguity for normal incidence, the Cartesian-coordinate system is rotated along the shortest possible path, i.e. along the axis that is normal to the original x-axis and the mode's k-vector. All rotations are around the origin of the coordinate system, incurring no phase shift there.

Vectors can be converted to field distributions on the complete grid using the methods:

*srcvec2freespace()* super-position of free-space plane waves in the whole volume (fast)

*srcvec2source()* super-position of free-space plane waves at the source planes at the front and back (fast)

*source2detfield()* calculate the field in the whole volume using the `solver.Solution` object (slow)

***detfield2detvec()* vector corresponding to the detected field at the detection planes (fast). The fields**
at those planes should only contain the outward propagating waves. Hence, inwards propagating waves should be subtracted before using this method!

***srcvec2detfield()* calculate the field in the whole volume and convert it to a detection vector (slow)**
The latter is used in the matrix multiplication method: `matmul`, @

**Parameters**

- **grid** (Union[*Grid*, Sequence, ndarray]) – A Grid object or a Sequence of vectors with uniformly increasing values that indicate the positions in a plaid grid of sample points for the material and solution. In the one-dimensional case, a simple increasing Sequence of uniformly-spaced numbers may be provided as an alternative. The length of the ranges determines the data_shape, to which the source_distribution, epsilon, xi, zeta, mu, and initial_field must broadcast when specified as ndarrays.
- **vectorial** (Optional[bool]) – a boolean indicating if the source and solution are 3-vectors-fields (True) or scalar fields (False).
- **wavenumber** (Optional[Real]) – the wavenumber in vacuum = 2 pi / vacuum_wavelength. The wavelength in the same units as used for the other inputs/outputs.
- **angular_frequency** (Optional[Real]) – alternative argument to the wavenumber = angular_frequency / c
- **vacuum_wavelength** (Optional[Real]) – alternative argument to the wavenumber = 2 pi / vacuum_wavelength
- **epsilon** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – an array or function that returns the (tensor) epsilon that represents the permittivity at the points indicated by the grid specified as its input arguments.
- **xi** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – an array or function that returns the (tensor) xi for bi-(an)isotropy at the points indicated by the grid specified as its input arguments.
- **zeta** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – an array or function that returns the (tensor) zeta for bi-(an)isotropy at the points indicated by the grid specified as its input arguments.

- **mu** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – an array or function that returns the (tensor) permeability at the points indicated by the grid specified as its input arguments.
- **refractive_index** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – an array or function that returns the (tensor) refractive_index = np.sqrt(permittivity) at the points indicated by the *grid* input argument.
- **bound** (Optional[*Bound*]) – An object representing the boundary of the calculation volume. Default: None, PeriodicBound(grid)
- **dtype** – optional numpy datatype for the internal operations and results. This must be a complex number type as numpy.complex128 or np.complex64.
- **callback** (Callable) – optional function that will be called with as argument this solver. This function can be used to check and display progress. It must return a boolean value of True to indicate that further iterations are required.
- **caching** (bool) – Cache field propagation calculations. By default, the results are cached for multiplications with basis vectors. Numerical errors might accumulate for certain superpositions. Setting this property to False will ensure that field propagations are always used and the constructor argument array is ignored.
- **array** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – Optional in case the matrix values have been calculated before and stored. If not specified, the matrix is calculated from the material properties. I specified, this must be a sequence or numpy.ndarray of complex numbers representing the matrix, or a function that returns one.

**Returns**

The Solution object that has the E and H fields, as well as iteration information.

**property grid:** *Grid*

The calculation grid.

**property vectorial:** bool

Boolean to indicates whether calculations happen on polarized (True) or scalar (False) fields.

**property caching:** bool

When set to True, this object uses cached values instead of propagating the field through the scatterer. Otherwise, field propagation is used for all matrix operations. This can help avoid the accumulation of numerical errors.

**srcvec2freespace**(*input_vector*)

Convert an input source vector to a superposition of plane waves at the origin and spreading over the whole volume. The input vector specifies the propagating modes in the far-field (the inverse Fourier transform of the fields at the sample origin). Incident waves at an angle will result in higher amplitudes to compensate for the reduction in propagation along the propagation axis through the entrance plane.

Used in `ScatteringMatrix.srcvec2source` to calculate the source field distribution before entering the scatterer.

Used in `ScatteringMatrix.__matmul__` to distinguish incoming from back-scattered light.

**Parameters**

**input_vector** (Union[Complex, Sequence, ndarray, LinearOperator]) – A source vector or array of shape [2, M, P], where the first axis indicates the side (front, back), the second axis indicates the propagation mode (direction, top-bottom-left-right), and the final axis indicates the polarization (1 for scalar, 2 for polarized: V-H).

**Return type**
ndarray

**Returns**
An nd-array with the field on the calculation grid. Its shape is (1, *self.grid.shape) for scalar calculations and (3, *self.grid.shape) for vectorial calculations with polarization.

**srcvec2source**(*input_vector*, *out=None*)

Converts a source vector into an (N+1)D-array with the source field at the front and back of the scatterer. The source field is such that it produces E-fields of unit intensity for unit vector inputs.

Used in `self.vector2field()` and `self.__matmul__()`.

**Parameters**

- **input_vector** (Union[Complex, Sequence, ndarray, LinearOperator]) – A source vector with `self.shape[1]` elements. One value per side, per independent polarization (2), and per mode (inwards propagating k-vectors only).
- **out** (Optional[ndarray]) – (optional) numpy array to store the result.

**Return type**
ndarray

**Returns**
The field distribution as an array of shape [nb_pol, *self.grid], where nb_pol = 3 for a vectorial calculation and 1 for a scalar calculation.

**source2detfield**(*source*, *out=None*)

Calculates the (N+1)D-input-field distribution throughout the scatterer for a given source field distribution.

**Parameters**

- **source** (Union[Complex, Sequence, ndarray, LinearOperator]) – The source field distribution in the whole space.
- **out** (Optional[ndarray]) – (optional) numpy array to store the result (shape: self.grid.shape, dtype: self.dtype).

**Return type**
ndarray

**Returns**
The field distribution as an array of shape [nb_pol, *self.grid], where nb_pol = 3 for a vectorial calculation and 1 for a scalar calculation.

**srcvec2detfield**(*input_vector*, *out=None*)

Calculates the (N+1)D-input-field distribution throughout the scatterer for a given input source vector.

Used in `self.__matmul__()` and external code.

**Parameters**

- **input_vector** (Union[Complex, Sequence, ndarray, LinearOperator]) – A source vector with `self.shape[1]` elements.
- **out** (Optional[ndarray]) – (optional) numpy array to store the result.

**Return type**
ndarray

**Returns**
The field distribution as an array of shape [nb_pol, *self.grid], where nb_pol = 3 for a vectorial calculation and 1 for a scalar calculation.

`detfield2detvec`(*field*)

Converts the (N+1)D-output-field defined at the front and back detection planes of the scatterer to a detection vector that describes the resulting far-field distribution (the inverse Fourier transform of the fields at the sample origin). The fields at those planes should only contain the outward propagating waves. Hence, inwards propagating waves should be subtracted before using this method!

This is used in `self.__matmul__()`.

**Parameters**
**field** (Union[Complex, Sequence, ndarray, LinearOperator]) – The detected field in all space (of which only the detection space is used).

**Return type**
ndarray

**Returns**
Detection vector.

`detvec2srcvec`(*vec*)

Convert forward propagating detection vector into a backward propagating (time-reversed) source vector.

**Parameters**
**vec** (Union[Complex, Sequence, ndarray, LinearOperator]) – detection vector obtained from solution or scattering matrix multiplication.

**Return type**
ndarray

**Returns**
Time-reversed vector, that can be used as a source vector.

`__setitem__`(*key*, *value*)

Updating this matrix is not possible. Use a `Matrix` object instead.

**Parameters**

- **key** – Index or slice.
- **value** – The new value.

`__array__`(*out=None*)

Lazily calculates the scattering matrix as a regular `numpy.ndarray`

`property H`

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

**property T**

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**__add__**(*x*)

**__call__**(*x*)

Call self as a function.

**__len__**()

The number of rows in the matrix as an integer.

**Return type**
int

**__matmul__**(*other*)

**__mul__**(*x*)

**__neg__**()

**static __new__**(*cls*, **args*, ***kwargs*)

**__pow__**(*p*)

**__rmatmul__**(*other*)

**__rmul__**(*x*)

**__sub__**(*x*)

**adjoint**()

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

**property back_reflection:** *BackReflectionMatrix*

Select the quarter of the scattering matrix corresponding to the light that is reflected of the back. It indicates how the light coming from positive infinity is back reflected to positive infinity.

**Returns**
The back-reflection matrix of shape `self.shape // 2`.

**property backward_transmission:** *BackwardTransmissionMatrix*

Select the backward-transmitted quarter of the scattering matrix. It indicates how the light coming from positive infinity is transmitted to negative infinity.

**Returns**
The backward-transmission matrix of shape `self.shape // 2`.

**dot**(*x*)

Matrix-matrix or matrix-vector multiplication.

**Parameters**
**x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

**Returns**
**Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

**Return type**
array

**property forward_transmission:** *ForwardTransmissionMatrix*

Select the forward-transmitted quarter of the scattering matrix. It indicates how the light coming from negative infinity is transmitted to positive infinity.

**Returns**
The forward-transmission matrix of shape `self.shape // 2`.

**property front_reflection:** *FrontReflectionMatrix*

Select the quarter of the scattering matrix corresponding to the light that is reflected of the front. It indicates how the light coming from negative infinity is back reflected to negative infinity.

**Returns**
The front-reflection matrix of shape `self.shape // 2`.

**inv**(*noise_level=0.0*)

The (Tikhonov regularized) inverse of the scattering matrix.

**Parameters**
**noise_level** (`float`) – (optional) argument to regularize the inversion of a (near-)singular matrix.

**Return type**
`ndarray`

**Returns**
An nd-array with the inverted matrix so that `self @ self.inv` approximates the identity.

**matmat**(*X*)

Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

**Parameters**
**X** (*{matrix, ndarray}*) – An array with shape (N,K).

**Returns**
**Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

**Return type**
{matrix, ndarray}

### Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

`matvec(x)`

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

**Returns**
**y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

`ndim = 2`

`rmatmat(X)`

Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

**Parameters**
**X** (*{matrix, ndarray}*) – A matrix or 2D array.

**Returns**
**Y** – A matrix or 2D array depending on the type of the input.

**Return type**
{matrix, ndarray}

### Notes

This rmatmat wraps the user-specified rmatmat routine.

`rmatvec(x)`

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

**Returns**
**y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

**property side:** int

**transfer**(*noise_level=0.0*)

Calculates the transfer matrix, relating one side of the scatterer to the other side (top, bottom). Each side can have incoming and outgoing waves. This is in contrast to the scattering matrix, `self.__array__`, which relates incoming waves from both sides to outgoing waves from both sides. One can be calculated from the other using the `matrix.convert()` function, though this calculation may be ill-conditioned (sensitive to noise). Therefore, the optional argument `noise_level` should be used to indicate the root-mean-square expectation value of the measurement error. This avoids divisions by near-zero values and obtains a best estimate using Tikhonov regularization.

**Parameters**
**noise_level** (float) – (optional) argument to regularize the inversion of a (near) singular backwards transmission matrix.

**Return type**
ndarray

**Returns**
An nd-array with the transfer matrix relating top-to-bottom instead of in-to-out. This can be converted back into a scattering matrix using the `matrix.convert()` function.

The first half of the vector inputs and outputs to the scattering and transfer matrices represent fields propagating forward along the positive propagation axis (0) and the second half represents fields propagating backward along the negative direction.

**Notation:**

**p**
positive propagation direction along propagation axis 0

**n**
negative propagation direction along propagation axis 0

**i**
inwards propagating (from source on either side)

**o**
outwards propagating (backscattered or transmitted)

Scattering matrix equation (in -> out):

[po] = [A, B] [pi]

[no] = [C, D] [ni]

Transfer matrix equation (top -> bottom):

[po] = [A - B inv(D) C, B inv(D)] [pi]

[ni] = [ - inv(D) C inv(D)] [no],

where inv(D) is the (regularized) inverse of D.

**transpose**()

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

class macromax.**Grid**(*shape=None*, *step=None*, *extent=None*, *first=None*, *center=None*, *last=None*, *include_last=False*, *ndim=None*, *flat=False*, *origin_at_center=True*, *center_at_index=True*)

Bases: Sequence

A class representing an immutable uniformly-spaced plaid Cartesian grid and its Fourier Transform.

See also MutableGrid

**__init__**(*shape=None*, *step=None*, *extent=None*, *first=None*, *center=None*, *last=None*, *include_last=False*, *ndim=None*, *flat=False*, *origin_at_center=True*, *center_at_index=True*)

Construct an immutable Grid object.

**Parameters**

- **shape** – An integer vector array with the shape of the sampling grid.
- **step** – A vector array with the spacing of the sampling grid.
- **extent** – The extent of the sampling grid as shape * step
- **first** – A vector array with the first element for each dimension. The first element is the smallest element if step is positive, and the largest when step is negative.
- **center** – A vector array with the center element for each dimension. The center position in the grid is rounded to the next integer index unless center_at_index is set to False for that partical axis.
- **last** – A vector array with the last element for each dimension. Unless include_last is set to True for the associated dimension, all but the last element is returned when calling self[axis].
- **include_last** – A boolean vector array indicating whether the returned vectors, self[axis], should include the last element (True) or all-but-the-last (False)
- **ndim** (Optional[int]) – A scalar integer indicating the number of dimensions of the sampling space.
- **flat** (Union[bool, Sequence, ndarray]) – A boolean vector array indicating whether the returned vectors, self[axis], should be flattened (True) or returned as an open grid (False)
- **origin_at_center** (Union[bool, Sequence, ndarray]) – A boolean vector array indicating whether the origin should be fft-shifted (True) or be ifftshifted to the front (False) of the returned vectors for self[axis].
- **center_at_index** (Union[bool, Sequence, ndarray]) – A boolean vector array indicating whether the center of the grid should be rounded to an integer index for each dimension. If False and the shape has an even number of elements, the next index is used as the center, (self.shape / 2).astype(int).

**static from_ranges**(**ranges*)

Converts one or more ranges of numbers to a single Grid object representation. The ranges can be specified as separate parameters or as a tuple.

**Parameters**

**ranges** (Union[int, float, complex, Sequence, ndarray]) – one or more ranges of uniformly spaced numbers.

**Return type**
*Grid*

**Returns**
A Grid object that represents the same ranges.

**property ndim: int**

The number of dimensions of the space this grid spans.

**property shape: array**

The number of sample points along each axis of the grid.

**property step: ndarray**

The sample spacing along each axis of the grid.

**property center: ndarray**

The central coordinate of the grid.

**property center_at_index: array**

Boolean vector indicating whether the central coordinate is aligned with a grid point when the number of points is even along the associated axis. This has no effect when the the number of sample points is odd.

**property flat: array**

Boolean vector indicating whether self[axis] returns flattened (raveled) vectors (True) or not (False).

**property origin_at_center: array**

Boolean vector indicating whether self[axis] returns ranges that are monotonous (True) or ifftshifted so that the central index is the first element of the sequence (False).

**property as_flat: *Grid***

return: A new Grid object where all the ranges are 1d-vectors (flattened or raveled)

**property as_non_flat: *Grid***

return: A new Grid object where all the ranges are 1d-vectors (flattened or raveled)

**property as_origin_at_0: *Grid***

return: A new Grid object where all the ranges are ifftshifted so that the origin as at index 0.

**property as_origin_at_center: *Grid***

return: A new Grid object where all the ranges have the origin at the center index, even when the number of elements is odd.

**swapaxes**(*axes*)

Reverses the order of the specified axes.

**Return type**
*Grid*

**transpose**(*axes=None*)

Reverses the order of all axes.

**Return type**
*Grid*

**project**(*axes_to_keep=None*, *axes_to_remove=None*)

Removes all but the specified axes and reduces the dimensions to the number of specified axes.

**Parameters**

- **axes_to_keep** (Union[int, slice, Sequence, array, None]) – The indices of the axes to keep.
- **axes_to_remove** (Union[int, slice, Sequence, array, None]) – The indices of the axes to remove. Default: None

**Return type**
*Grid*

**Returns**
A Grid object with ndim == len(axes) and shape == shape[axes].

**property first: ndarray**

return: A vector with the first element of each range

**property extent: ndarray**

The spatial extent of the sampling grid.

**property size: int**

The total number of sampling points as an integer scalar.

**property dtype**

The numeric data type for the coordinates.

**property f: *Grid***

The equivalent frequency Grid.

**property k: *Grid***

The equivalent k-space Grid.

**__add__**(*term*)

Add a (scalar) offset to the Grid coordinates.

**Return type**
*Grid*

**__mul__**(*factor*)

Scales all ranges with a factor.

**Parameters**
**factor** (Union[int, float, complex, Sequence, array]) – A scalar factor for all dimensions, or a vector of factors, one for each dimension.

**Return type**
*Grid*

**Returns**
A new scaled Grid object.

**__matmul__**(*other*)

Determines the Grid spanning the tensor space, with ndim equal to the sum of both ndims.

**Parameters**
**other** (*Grid*) – The Grid with the right-hand dimensions.

**Return type**
*Grid*

**Returns**
A new Grid with ndim == self.ndim + other.ndim.

**__sub__**(*term*)

Subtract a (scalar) value from all Grid coordinates.

**Return type**
*Grid*

**__truediv__**(*denominator*)

Divide the grid coordinates by a value.

**Parameters**
**denominator** (Union[int, float, complex, Sequence, ndarray]) – The denominator to divide by.

**Return type**
*Grid*

**Returns**
A new Grid with the divided coordinates.

**__neg__**()

Invert the coordinate values and the direction of the axes.

**__len__**()

The number of axes in this sampling grid. Or, the number of elements when this object is not multi-dimensional.

**Return type**
int

**__iter__**()

**property immutable:** *Grid*

Return a new immutable Grid object.

**property mutable:** *MutableGrid*

return: A new MutableGrid object.

**__eq__**(*other*)

Compares two Grid objects.

**Return type**
bool

**property multidimensional:** bool

Single-dimensional grids behave as Sequences, multi-dimensional behave as a Sequence of vectors.

**classmethod __init_subclass__**(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

**static __new__**(*cls*, **args*, ***kwds*)

**count**(*value*) → integer -- return number of occurrences of value

**index**(*value*[, *start*[, *stop*]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

### Subpackages

#### macromax.backend package

This package defines a general interface to do parallel operations efficiently on different architectures, and it provides specific implementations that are used by the *macromax.Solution* class.

The specific implementation that is used can be:

- selected automatically based on availability and in order of deemed efficiency, or
- specified when loading the backend with the *load()* function, or
- configured using *config()* function.

macromax.backend.**config**(**args*, ***kwargs*)

Configure a specific back-end, overriding the automatic detection. E.g. use *from macromax import backend* followed by:

```
backend.config(type='numpy')
backend.config(dict(type='numpy'))
backend.config(type='torch', device='cpu')
backend.config(dict(type='torch', device='cpu'))
backend.config(dict(type='torch', device='cuda'))
backend.config(dict(type='torch', device='cuda'))
backend.config([dict(type='torch', device='cuda'), dict(type='numpy')])
```

Default back-ends can be specified in the file *backend_config.json* in the current folder. This configuration file should contain a list of potential back-ends in [JSON format](https://en.wikipedia.org/wiki/JSON), e.g. ```json [

{“type”: “torch”, “device”: “cuda”}, {“type”: “numpy”}, {“type”: “torch”, “device”: “cpu”}

]

The first back-end that loads correctly will be used.

**type args**
Sequence[Dict]

**param args**
A dictionary or a sequence of dictionaries with at least the ‘type’ key. Key-word arguments are interpreted as a dictionary.

macromax.backend.**load**(*nb_pol_dims*, *grid*, *dtype*, *config_list=None*)

Load the default or the backend specified using backend.config(). This configuration file should contain a list of potential back-ends in [JSON format](https://en.wikipedia.org/wiki/JSON), e.g. ```json [

{“type”: “torch”, “device”: “cuda”}, {“type”: “numpy”}, {“type”: “torch”, “device”: “cpu”}

]

The first back-end that loads correctly will be used.

**type nb_pol_dims**
int

**param nb_pol_dims**
The number of polarization dimensions: 1 for scalar, 3 for vectorial calculations.

**type grid**
*Grid*

**param grid**
The uniformly-spaced Cartesian calculation grid as a Grid object.

**type dtype**
**param dtype**
The scalar data type. E.g. np.complex64 or np.complex128.

**type config_list**
Optional[List[Dict]]

**param config_list**
List of alternative backend configurations.

**rtype**
*BackEnd*

**return**
A BackEnd object to start the calculation with.

**class** macromax.backend.**BackEnd**(*nb_dims*, *grid*, *hardware_dtype=<class 'numpy.complex128'>*)

Bases: ABC

A class that provides methods to work with arrays of matrices or block-diagonal matrices, represented as ndarrays, where the first two dimensions are those of the matrix, and the final dimensions are the coordinates over which the operations are parallelized and the Fourier transforms are applied.

**__init__**(*nb_dims*, *grid*, *hardware_dtype=<class 'numpy.complex128'>*)

Construct object to handle parallel operations on square matrices of nb_rows x nb_rows elements. The matrices refer to points in space on a uniform plaid grid.

**Parameters**

- **nb_dims** (int) – The number of rows and columns in each matrix. 1 for scalar operations, 3 for polarization
- **grid** (*Grid*) – The grid that defines the position of the matrices.
- **hardware_dtype** – (optional) The datatype to use for operations.

**property vector_length:** int

The shape of the square matrix that transforms a single vector in the set. This is a pair of identical integer numbers.

**property ft_axes:** tuple

The integer indices of the spatial dimensions, i.e. on which to Fourier Transform.

**property grid:** *Grid*

A Grid object representing the sample points in the spatial dimensions.

**property vectorial: bool**

A boolean indicating if this object represents a vector space (as opposed to a scalar space).

**property hardware_dtype**

The scalar data type that is processed by this back-end.

**property numpy_dtype**

The equivalent hardware data type in numpy

**astype**(*arr*, *dtype=None*)

**Parameters**

- **arr** (Union[Complex, Sequence, ndarray]) – An object that is, or can be converted to, an ndarray.
- **dtype** – (optional) scalar data type of the returned array elements

**Return type**
ndarray

**Returns**
torch.Tensor type

**asnumpy**(*arr*)

Convert the internal array (or tensor) presentation to a numpy.ndarray.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The to-be-converted array.

**Return type**
ndarray

**Returns**
The corresponding numpy ndarray.

**property eps: float**

The precision of the data type (self.dtype) of this back-end.

**abstract allocate_array**(*shape=None*, *dtype=None*, *fill_value=None*)

Allocates a new vector array of shape grid.shape and word-aligned for efficient calculations.

**Parameters**

- **shape** (Union[Complex, Sequence, ndarray, None]) – (optional) The shape of the returned array.
- **dtype** (Union[Type[float64], Type[float32], None]) – (optional) The scalar data type of the elements in the array. This may be converted to an equivalent, back-end, specific data type.
- **fill_value** (Optional[Complex]) – (optional) A scalar value to pre-populate the array. The default (None) does not pre-populate the array, leaving it in a random state.

**Return type**
ndarray

**Returns**
A reference to the array.

**assign**(*arr*, *out*)

Assign the values of one array to those of another. The dtype and shape is broadcasted to match that of the output array.

**Parameters**

- **arr** (Union[Complex, Sequence, ndarray]) – The values that need to be assigned to another array.
- **out** (ndarray) – (optional) The target array, which must be of the same shape.

**Return type**
ndarray

**Returns**
The target array or a newly allocated array with the same values as those in arr.

**assign_exact**(*arr*, *out*)

Assign the values of one array to those of another. The dtype and shape must match that of the output array.

**Parameters**

- **arr** (Union[Complex, Sequence, ndarray]) – The values that need to be assigned to another array.
- **out** (ndarray) – (optional) The target array, which must be of the same shape.

**Return type**
ndarray

**Returns**
The target array or a newly allocated array with the same values as those in arr.

**copy**(*arr*)

Makes an independent copy of an ndarray.

**Return type**
ndarray

**ravel**(*arr*)

Returns a flattened view of the array.

**Return type**
ndarray

**sign**(*arr*)

Returns an array with values of -1 where arr is negative, 0 where arr is 0, and 1 where arr is positive.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The array to check.

**Return type**
ndarray

**Returns**
np.sign(arr) or equivalent.

**first**(*arr*)

Returns the first element of the flattened array.

**Return type**
Complex

**static expand_dims**(*arr*, *axis*)

Inserts a new singleton axis at the indicated position, thus increasing ndim by 1.

**Return type**
ndarray

**property eye: ndarray**

Returns an identity tensor that can be multiplied using singleton expansion. This can be useful for scalar additions or subtractions.

**Returns**
an array with the number of dimensions matching that of the BackEnds's data set.

**any**(*arr*)

Returns True if all elements of the array are True.

**Return type**
bool

**allclose**(*arr*, *other=0.0*)

Returns True if all elements in arr are close to other.

**Return type**
bool

**amax**(*arr*)

Returns the maximum of the flattened array.

**Return type**
float

**sort**(*arr*)

Sorts array elements along the first (left-most) axis.

**Return type**
ndarray

**abstract ft**(*arr*)

Calculates the discrete Fourier transform over the spatial dimensions of E. The computational complexity is that of a Fast Fourier Transform: `O(N.log(N))`.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – An ndarray representing a vector field.

**Return type**
ndarray

**Returns**
An ndarray holding the Fourier transform of the vector field E.

**abstract ift**(*arr*)

Calculates the inverse Fourier transform over the spatial dimensions of E. The computational complexity is that of a Fast Fourier Transform: `O(N.log(N))`. The scaling is so that `E == self.ift(self.ft(E))`

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – An ndarray representing a Fourier-transformed vector field.

**Return type**
ndarray

**Returns**
An ndarray holding the inverse Fourier transform of the vector field E.

**property array_ft_input: ndarray**

The pre-allocate array for Fourier Transform inputs

**property array_ft_output: ndarray**

The pre-allocate array for Fourier Transform outputs

**property array_ift_input: ndarray**

The pre-allocate array for Inverse Fourier Transform inputs

**property array_ift_output: ndarray**

The pre-allocate array for Inverse Fourier Transform outputs

**conj**(*arr*)

Returns the conjugate of the elements in the input.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The input array.

**Return type**
ndarray

**Returns**
arr.conj or equivalent

**abs**(*arr*)

Returns the absolute value (magnitude) of the elements in the input.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The input array.

**Return type**
ndarray

**Returns**
np.abs(arr) or equivalent

**real**(*arr*)

**Return type**
ndarray

**convolve**(*operation_ft*, *arr*)

Apply an operator in Fourier space. This is used to perform a cyclic FFT convolution which overwrites its input argument *arr*.

**Parameters**

- **operation_ft** (Callable[[Union[Complex, Sequence, ndarray]], Union[Complex, Sequence, ndarray]]) – The function that acts on the Fourier-transformed input.
- **arr** (Union[Complex, Sequence, ndarray]) – The to-be-convolved argument array.

**Return type**
ndarray

**Returns**
the convolved input array.

`is_scalar`(*arr*)

Tests if A represents a scalar field (as opposed to a vector field).

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The ndarray to be tested.

**Return type**
bool

**Returns**
A boolean indicating whether A represents a scalar field (True) or not (False).

`is_vector`(*arr*)

Tests if A represents a vector field.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The ndarray to be tested.

**Return type**
bool

**Returns**
A boolean indicating whether A represents a vector field (True) or not (False).

`is_matrix`(*arr*)

Checks if an ndarray is a matrix as defined by this parallel_ops_column object.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The matrix to be tested.

**Return type**
bool

**Returns**
boolean value, indicating if A is a matrix.

`swapaxes`(*arr*, *ax_from*, *ax_to*)

Transpose (permute) two axes of an ndarray.

**Return type**
ndarray

`to_matrix_field`(*arr*)

Converts the input to an array of the full number of dimensions: len(self.matrix_shape) + len(self.grid.shape), and dtype. The size of each dimensions must match that of that set for the *BackEnd* or be 1 (assumes broadcasting). For electric fields in 3-space, self.matrix_shape == (N, N) == (3, 3) The first (left-most) dimensions of the output are either

- 1x1: The identity matrix for a scalar field, as sound waves or isotropic permittivity.
- Nx1: A vector for a vector field, as the electric field.
- NxN: A matrix for a matrix field, as anisotropic permittivity

None is interpreted as 0. Singleton dimensions are added on the left so that all dimensions are present. Inputs with 1xN are transposed (not conjugate) to Nx1 vectors.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The input can be scalar, which assumes that its value is assumed to be repeated for all space. The value can be a one-dimensional vector, in which case the vector is assumed to be repeated for all space.

**Return type**
ndarray

**Returns**
An array with *ndim == len(self.matrix_shape) + len(self.grid.shape)* and with each non-singleton dimension matching those of the nb_rows and data_shape.

`adjoint`(*mat*)

Transposes the elements of individual matrices with complex conjugation.

**Parameters**
**mat** (Union[Complex, Sequence, ndarray]) – The ndarray with the matrices in the first two dimensions.

**Return type**
ndarray

**Returns**
An ndarray with the complex conjugate transposed matrices.

`subtract`(*left_term*, *right_term*)

Point-wise difference of A and B.

**Parameters**

- **left_term** (Union[Complex, Sequence, ndarray]) – The left matrix array, must start with dimensions n x m
- **right_term** (Union[Complex, Sequence, ndarray]) – The right matrix array, must have matching or singleton dimensions to those of A. In case of missing dimensions, singletons are assumed.

**Return type**
ndarray

**Returns**
The point-wise difference of both sets of matrices. Singleton dimensions are expanded.

`mul`(*left_factor*, *right_factor*, *out=None*)

Point-wise matrix multiplication of A and B. Overwrites right_factor!

**Parameters**

- **left_factor** (Union[Complex, Sequence, ndarray]) – The left matrix array, must start with dimensions n x m
- **right_factor** (Union[Complex, Sequence, ndarray]) – The right matrix array, must have matching or singleton dimensions to those of A, bar the first two dimensions. In case of missing dimensions, singletons are assumed. The first dimensions must be m x p. Where the m matches that of the left hand matrix unless both m and p are 1 or both n and m are 1, in which case the scaled identity is assumed.
- **out** (Optional[ndarray]) – (optional) The destination array for the results.

**Return type**
ndarray

**Returns**
An array of matrix products with all but the first two dimensions broadcast as needed.

**abstract ldivide**(*denominator*, *numerator=1.0*)

Parallel matrix left division, A^{-1}B, on the final two dimensions of A and B result_lm = A_kl B_km

A and B must have have all but the final dimension identical or singletons. B defaults to the identity matrix.

**Parameters**

- **denominator** (Union[Complex, Sequence, ndarray]) – The set of denominator matrices.
- **numerator** (Union[Complex, Sequence, ndarray]) – The set of numerator matrices.

**Return type**
ndarray

**Returns**
The set of divided matrices.

**inv**(*mat*)

Inverts the set of input matrices M.

**Parameters**
**mat** (Union[Complex, Sequence, ndarray]) – The set of input matrices.

**Return type**
ndarray

**Returns**
The set of inverted matrices.

**curl**(*field_array*)

Calculates the curl of a vector E with the final dimension the vector dimension. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The set of input matrices.

**Return type**
ndarray

**Returns**
The curl of E.

**curl_ft**(*field_array_ft*)

Calculates the Fourier transform of the curl of a Fourier transformed E with the final dimension the vector dimension. The final dimension of the output will always be of length 3; however, the input length may be shorter, in which case the missing values are assumed to be zero. The first dimension of the input array corresponds to the first element in the final dimension, if it exists, the second dimension corresponds to the second element etc.

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray]) – The input vector array of dimensions `[vector_length, 1, *data_shape]`.

**Return type**
ndarray

**Returns**
The Fourier transform of the curl of F.

**cross**(*A*, *B*)

Calculates the cross product of vector arrays A and B.

**Parameters**

- **A** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`
- **B** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`

**Return type**
ndarray

**Returns**
A vector array of dimensions `[vector_length, 1, *data_shape]` containing the cross product A x B in the first dimension and the other dimensions remain on the same axes.

**outer**(*A*, *B*)

Calculates the Dyadic product of vector arrays A and B.

**Parameters**

- **A** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`
- **B** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`

**Return type**
ndarray

**Returns**
A matrix array of dimensions `[vector_length, vector_length, *data_shape]` containing the dyadic product $A \otimes B$ in the first two dimensions and the other dimensions remain on the same axes.

**div**(*field_array*)

Calculate the divergence of the input vector or tensor field E. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The input array representing vector or tensor field. The input is of the shape `[m, n, x, y, z, ...]`

**Return type**
ndarray

**Returns**
The divergence of the vector or tensor field in the shape `[n, 1, x, y, z]`.

**div_ft**(*field_array_ft*)

Calculate the Fourier transform of the divergence of the pre-Fourier transformed input electric field.

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray]) – The input array representing the field pre-Fourier-transformed in all spatial dimensions. The input is of the shape `[m, n, x, y, z, ...]`

**Return type**
ndarray

**Returns**
The Fourier transform of the divergence of the field in the shape `[n, 1, x, y, z]`.

**transversal_projection**(*field_array*)

Projects vector arrays onto their transverse component. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The input vector E array.

**Return type**
ndarray

**Returns**
The transversal projection.

**longitudinal_projection**(*field_array*)

Projects vector arrays onto their longitudinal component. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The input vector E array.

**Return type**
ndarray

**Returns**
The longitudinal projection.

**transversal_projection_ft**(*field_array_ft*)

Projects the Fourier transform of a vector E array onto its transversal component.

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray]) – The Fourier transform of the input vector E array.

**Return type**
ndarray

**Returns**
The Fourier transform of the transversal projection.

**longitudinal_projection_ft**(*field_array_ft*)

Projects the Fourier transform of a vector array onto its longitudinal component. Overwrites self.array_ft_input!

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray]) – The Fourier transform of the input vector E array.

**Return type**
ndarray

**Returns**
The Fourier transform of the longitudinal projection.

**property k:** Sequence[ndarray]

A list of the k-vector components along each axis.

**property k2:** ndarray

Helper def for calculation of the Fourier transform of the Green def

**Returns**
$|k|^2$ for the specified sample grid and output shape

`mat3_eigh`(*arr*)

Calculates the eigenvalues of the 3x3 Hermitian matrices represented by A and returns a new array of 3-vectors, one for each matrix in A and of the same dimensions, baring the second dimension. When the first two dimensions are 3x1 or 1x3, a diagonal matrix is assumed. When the first two dimensions are singletons (1x1), a constant diagonal matrix is assumed and only one eigenvalue is returned. Returns an array of one dimension less: 3 x data_shape. With the exception of the first dimension, the shape is maintained.

Before substituting this for `numpy.linalg.eigvalsh`, note that this implementation is about twice as fast as the current numpy implementation for 3x3 Hermitian matrix fields. The difference is even greater for the PyTorch implementation. The implementation below can be made more efficient by limiting it to Hermitian matrices and thus real eigenvalues. In the end, only the maximal eigenvalue is required, so an iterative power iteration may be even faster, perhaps after applying a Given's rotation or Householder reflection to make Hermitian tridiagonal.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The set of 3x3 input matrices for which the eigenvalues are requested. This must be an ndarray with the first two dimensions of size 3.

**Return type**
ndarray

**Returns**
The set of eigenvalue-triples contained in an ndarray with its first dimension of size 3, and the remaining dimensions equal to all but the first two input dimensions.

`calc_roots_of_low_order_polynomial`(*C*)

Calculates the (complex) roots of polynomials up to order 3 in parallel. The coefficients of the polynomials are in the first dimension of C and repeated in the following dimensions, one for each polynomial to determine the roots of. The coefficients are input for low to high order in each column. In other words, the polynomial is: `np.sum(C * (x**range(C.size))) == 0`

This method is used by mat3_eigh, but only for polynomials with real roots.

**Parameters**
**C** (Union[Complex, Sequence, ndarray]) – The coefficients of the polynomial, per polynomial.

**Return type**
ndarray

**Returns**
The zeros in the complex plane, per polynomial.

`static evaluate_polynomial`(*C*, *X*)

Evaluates the polynomial P at X for testing.

**Parameters**

- **C** (Union[Complex, Sequence, ndarray]) – The coefficients of the polynomial, for each polynomial.
- **X** (Union[Complex, Sequence, ndarray]) – The argument of the polynomial, per polynomial.

**Return type**
ndarray

**Returns**
The values of the polynomials for the arguments X.

**static clear_cache**()

**norm**(*arr*)

Returns the l2-norm of a vectorized array.

**Return type**
float

macromax.backend.**tensor_type**

alias of ndarray

### Submodules

### macromax.backend.numpy module

The module providing the pure-python NumPy back-end implementation.

**class** macromax.backend.numpy.**BackEndNumpy**(*nb_dims*, *grid*, *hardware_dtype=<class 'numpy.complex128'>*)

Bases: BackEnd

A class that provides methods to work with arrays of matrices or block-diagonal matrices, represented as ndarrays, where the first two dimensions are those of the matrix, and the final dimensions are the coordinates over which the operations are parallelized and the Fourier transforms are applied.

**__init__**(*nb_dims*, *grid*, *hardware_dtype=<class 'numpy.complex128'>*)

Construct object to handle parallel operations on square matrices of nb_rows x nb_rows elements. The matrices refer to points in space on a uniform plaid grid.

**Parameters**

- **nb_dims** (int) – The number of rows and columns in each matrix. 1 for scalar operations, 3 for polarization
- **grid** (*Grid*) – The grid that defines the position of the matrices.
- **hardware_dtype** – (optional) The data type to use for operations.

**allocate_array**(*shape=None*, *dtype=None*, *fill_value=None*)

Allocates a new vector array of shape grid.shape and word-aligned for efficient calculations.

**Return type**
ndarray

**ft**(*arr*)

Calculates the discrete Fourier transform over the spatial dimensions of E. The computational complexity is that of a Fast Fourier Transform: O(N\log(N)).

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – An ndarray representing a vector field.

**Return type**
ndarray

**Returns**
An ndarray holding the Fourier transform of the vector field E.

**ift**(*arr*)

Calculates the inverse Fourier transform over the spatial dimensions of E. The computational complexity is that of a Fast Fourier Transform: `O(N\log(N))`. The scaling is so that `E == self.ift(self.ft(E))`

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – An ndarray representing a Fourier-transformed vector field.

**Return type**
ndarray

**Returns**
An ndarray holding the inverse Fourier transform of the vector field E.

**abs**(*arr*)

Returns the absolute value (magnitude) of the elements in the input.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The input array.

**Return type**
ndarray

**Returns**
np.abs(arr) or equivalent

**adjoint**(*mat*)

Transposes the elements of individual matrices with complex conjugation.

**Parameters**
**mat** (Union[Complex, Sequence, ndarray]) – The ndarray with the matrices in the first two dimensions.

**Return type**
ndarray

**Returns**
An ndarray with the complex conjugate transposed matrices.

**allclose**(*arr*, *other=0.0*)

Returns True if all elements in arr are close to other.

**Return type**
bool

**amax**(*arr*)

Returns the maximum of the flattened array.

**Return type**
float

**any**(*arr*)

Returns True if all elements of the array are True.

**Return type**
bool

**property array_ft_input: ndarray**

The pre-allocate array for Fourier Transform inputs

**property array_ft_output: ndarray**

The pre-allocate array for Fourier Transform outputs

**property array_ift_input: ndarray**

The pre-allocate array for Inverse Fourier Transform inputs

**property array_ift_output: ndarray**

The pre-allocate array for Inverse Fourier Transform outputs

**asnumpy**(*arr*)

Convert the internal array (or tensor) presentation to a numpy.ndarray.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The to-be-converted array.

**Return type**
ndarray

**Returns**
The corresponding numpy ndarray.

**assign**(*arr*, *out*)

Assign the values of one array to those of another. The dtype and shape is broadcasted to match that of the output array.

**Parameters**

- **arr** (Union[Complex, Sequence, ndarray]) – The values that need to be assigned to another array.
- **out** (ndarray) – (optional) The target array, which must be of the same shape.

**Return type**
ndarray

**Returns**
The target array or a newly allocated array with the same values as those in arr.

**assign_exact**(*arr*, *out*)

Assign the values of one array to those of another. The dtype and shape must match that of the output array.

**Parameters**

- **arr** (Union[Complex, Sequence, ndarray]) – The values that need to be assigned to another array.
- **out** (ndarray) – (optional) The target array, which must be of the same shape.

**Return type**
ndarray

**Returns**
The target array or a newly allocated array with the same values as those in arr.

**astype**(*arr*, *dtype=None*)

**Parameters**

- **arr** (Union[Complex, Sequence, ndarray]) – An object that is, or can be converted to, an ndarray.
- **dtype** – (optional) scalar data type of the returned array elements

**Return type**
ndarray

**Returns**
torch.Tensor type

`calc_roots_of_low_order_polynomial`(*C*)

Calculates the (complex) roots of polynomials up to order 3 in parallel. The coefficients of the polynomials are in the first dimension of C and repeated in the following dimensions, one for each polynomial to determine the roots of. The coefficients are input for low to high order in each column. In other words, the polynomial is: `np.sum(C * (x**range(C.size))) == 0`

This method is used by mat3_eigh, but only for polynomials with real roots.

**Parameters**
**C** (Union[Complex, Sequence, ndarray]) – The coefficients of the polynomial, per polynomial.

**Return type**
ndarray

**Returns**
The zeros in the complex plane, per polynomial.

`static clear_cache`()

`conj`(*arr*)

Returns the conjugate of the elements in the input.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The input array.

**Return type**
ndarray

**Returns**
arr.conj or equivalent

`convolve`(*operation_ft*, *arr*)

Apply an operator in Fourier space. This is used to perform a cyclic FFT convolution which overwrites its input argument *arr*.

**Parameters**

- **operation_ft** (Callable[[Union[Complex, Sequence, ndarray]], Union[Complex, Sequence, ndarray]]) – The function that acts on the Fourier-transformed input.
- **arr** (Union[Complex, Sequence, ndarray]) – The to-be-convolved argument array.

**Return type**
ndarray

**Returns**
the convolved input array.

`copy`(*arr*)

Makes an independent copy of an ndarray.

**Return type**
ndarray

**cross**(*A*, *B*)

Calculates the cross product of vector arrays A and B.

**Parameters**

- **A** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions [vector_length, 1, *data_shape]
- **B** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions [vector_length, 1, *data_shape]

**Return type**
ndarray

**Returns**
A vector array of dimensions [vector_length, 1, *data_shape] containing the cross product A x B in the first dimension and the other dimensions remain on the same axes.

**curl**(*field_array*)

Calculates the curl of a vector E with the final dimension the vector dimension. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The set of input matrices.

**Return type**
ndarray

**Returns**
The curl of E.

**curl_ft**(*field_array_ft*)

Calculates the Fourier transform of the curl of a Fourier transformed E with the final dimension the vector dimension. The final dimension of the output will always be of length 3; however, the input length may be shorter, in which case the missing values are assumed to be zero. The first dimension of the input array corresponds to the first element in the final dimension, if it exists, the second dimension corresponds to the second element etc.

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray]) – The input vector array of dimensions [vector_length, 1, *data_shape].

**Return type**
ndarray

**Returns**
The Fourier transform of the curl of F.

**div**(*field_array*)

Calculate the divergence of the input vector or tensor field E. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The input array representing vector or tensor field. The input is of the shape [m, n, x, y, z, ...]

**Return type**
ndarray

**Returns**
The divergence of the vector or tensor field in the shape [n, 1, x, y, z].

**div_ft**(*field_array_ft*)

Calculate the Fourier transform of the divergence of the pre-Fourier transformed input electric field.

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray]) – The input array representing the field pre-Fourier-transformed in all spatial dimensions. The input is of the shape `[m, n, x, y, z, ...]`

**Return type**
ndarray

**Returns**
The Fourier transform of the divergence of the field in the shape `[n, 1, x, y, z]`.

**property eps: float**

The precision of the data type (self.dtype) of this back-end.

**static evaluate_polynomial**(*C*, *X*)

Evaluates the polynomial P at X for testing.

**Parameters**

- **C** (Union[Complex, Sequence, ndarray]) – The coefficients of the polynomial, for each polynomial.
- **X** (Union[Complex, Sequence, ndarray]) – The argument of the polynomial, per polynomial.

**Return type**
ndarray

**Returns**
The values of the polynomials for the arguments X.

**static expand_dims**(*arr*, *axis*)

Inserts a new singleton axis at the indicated position, thus increasing ndim by 1.

**Return type**
ndarray

**property eye: ndarray**

Returns an identity tensor that can be multiplied using singleton expansion. This can be useful for scalar additions or subtractions.

**Returns**
an array with the number of dimensions matching that of the BackEnds's data set.

**first**(*arr*)

Returns the first element of the flattened array.

**Return type**
Complex

**property ft_axes: tuple**

The integer indices of the spatial dimensions, i.e. on which to Fourier Transform.

**property grid: *Grid***

A Grid object representing the sample points in the spatial dimensions.

**property hardware_dtype**

The scalar data type that is processed by this back-end.

**inv**(*mat*)

Inverts the set of input matrices M.

**Parameters**
**mat** (Union[Complex, Sequence, ndarray]) – The set of input matrices.

**Return type**
ndarray

**Returns**
The set of inverted matrices.

**is_matrix**(*arr*)

Checks if an ndarray is a matrix as defined by this parallel_ops_column object.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The matrix to be tested.

**Return type**
bool

**Returns**
boolean value, indicating if A is a matrix.

**is_scalar**(*arr*)

Tests if A represents a scalar field (as opposed to a vector field).

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The ndarray to be tested.

**Return type**
bool

**Returns**
A boolean indicating whether A represents a scalar field (True) or not (False).

**is_vector**(*arr*)

Tests if A represents a vector field.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The ndarray to be tested.

**Return type**
bool

**Returns**
A boolean indicating whether A represents a vector field (True) or not (False).

**property k:** Sequence[ndarray]

A list of the k-vector components along each axis.

**property k2:** ndarray

Helper def for calculation of the Fourier transform of the Green def

**Returns**
$|k|^2$ for the specified sample grid and output shape

**ldivide**(*denominator*, *numerator=1.0*)

Parallel matrix left division, A^{-1}B, on the final two dimensions of A and B result_lm = A_kl B_km

A and B must have have all but the final dimension identical or singletons. B defaults to the identity matrix.

**Parameters**

- **denominator** (Union[Complex, Sequence, ndarray]) – The set of denominator matrices.
- **numerator** (Union[Complex, Sequence, ndarray]) – The set of numerator matrices.

**Return type**
ndarray

**Returns**
The set of divided matrices.

**longitudinal_projection**(*field_array*)

Projects vector arrays onto their longitudinal component. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The input vector E array.

**Return type**
ndarray

**Returns**
The longitudinal projection.

**longitudinal_projection_ft**(*field_array_ft*)

Projects the Fourier transform of a vector array onto its longitudinal component. Overwrites self.array_ft_input!

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray]) – The Fourier transform of the input vector E array.

**Return type**
ndarray

**Returns**
The Fourier transform of the longitudinal projection.

**mat3_eigh**(*arr*)

Calculates the eigenvalues of the 3x3 Hermitian matrices represented by A and returns a new array of 3-vectors, one for each matrix in A and of the same dimensions, baring the second dimension. When the first two dimensions are 3x1 or 1x3, a diagonal matrix is assumed. When the first two dimensions are singletons (1x1), a constant diagonal matrix is assumed and only one eigenvalue is returned. Returns an array of one dimension less: 3 x data_shape. With the exception of the first dimension, the shape is maintained.

Before substituting this for `numpy.linalg.eigvalsh`, note that this implementation is about twice as fast as the current numpy implementation for 3x3 Hermitian matrix fields. The difference is even greater for the PyTorch implementation. The implementation below can be made more efficient by limiting it to Hermitian matrices and thus real eigenvalues. In the end, only the maximal eigenvalue is required, so an iterative power iteration may be even faster, perhaps after applying a Given's rotation or Householder reflection to make Hermitian tridiagonal.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The set of 3x3 input matrices for which the eigenvalues are requested. This must be an ndarray with the first two dimensions of size 3.

**Return type**
ndarray

**Returns**
The set of eigenvalue-triples contained in an ndarray with its first dimension of size 3, and the remaining dimensions equal to all but the first two input dimensions.

**mul**(*left_factor*, *right_factor*, *out=None*)

Point-wise matrix multiplication of A and B. Overwrites right_factor!

**Parameters**

- **left_factor** (Union[Complex, Sequence, ndarray]) – The left matrix array, must start with dimensions n x m
- **right_factor** (Union[Complex, Sequence, ndarray]) – The right matrix array, must have matching or singleton dimensions to those of A, bar the first two dimensions. In case of missing dimensions, singletons are assumed. The first dimensions must be m x p. Where the m matches that of the left hand matrix unless both m and p are 1 or both n and m are 1, in which case the scaled identity is assumed.
- **out** (Optional[ndarray]) – (optional) The destination array for the results.

**Return type**
ndarray

**Returns**
An array of matrix products with all but the first two dimensions broadcast as needed.

**norm**(*arr*)

Returns the l2-norm of a vectorized array.

**Return type**
float

**property numpy_dtype**

The equivalent hardware data type in numpy

**outer**(*A*, *B*)

Calculates the Dyadic product of vector arrays A and B.

**Parameters**

- **A** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`
- **B** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`

**Return type**
ndarray

**Returns**
A matrix array of dimensions `[vector_length, vector_length, *data_shape]` containing the dyadic product $A \otimes B$ in the first two dimensions and the other dimensions remain on the same axes.

**ravel**(*arr*)

Returns a flattened view of the array.

**Return type**
ndarray

**real**(*arr*)

**Return type**
ndarray

**sign**(*arr*)

Returns an array with values of -1 where arr is negative, 0 where arr is 0, and 1 where arr is positive.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The array to check.

**Return type**
ndarray

**Returns**
np.sign(arr) or equivalent.

**sort**(*arr*)

Sorts array elements along the first (left-most) axis.

**Return type**
ndarray

**subtract**(*left_term*, *right_term*)

Point-wise difference of A and B.

**Parameters**

- **left_term** (Union[Complex, Sequence, ndarray]) – The left matrix array, must start with dimensions n x m
- **right_term** (Union[Complex, Sequence, ndarray]) – The right matrix array, must have matching or singleton dimensions to those of A. In case of missing dimensions, singletons are assumed.

**Return type**
ndarray

**Returns**
The point-wise difference of both sets of matrices. Singleton dimensions are expanded.

**swapaxes**(*arr*, *ax_from*, *ax_to*)

Transpose (permute) two axes of an ndarray.

**Return type**
ndarray

**to_matrix_field**(*arr*)

Converts the input to an array of the full number of dimensions: len(self.matrix_shape) + len(self.grid.shape), and dtype. The size of each dimensions must match that of that set for the BackEnd or be 1 (assumes broadcasting). For electric fields in 3-space, self.matrix_shape == (N, N) == (3, 3) The first (left-most) dimensions of the output are either

- 1x1: The identity matrix for a scalar field, as sound waves or isotropic permittivity.
- Nx1: A vector for a vector field, as the electric field.
- NxN: A matrix for a matrix field, as anisotropic permittivity

None is interpreted as 0. Singleton dimensions are added on the left so that all dimensions are present. Inputs with 1xN are transposed (not conjugate) to Nx1 vectors.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The input can be scalar, which assumes that its value is assumed to be repeated for all space. The value can be a one-dimensional vector, in which case the vector is assumed to be repeated for all space.

**Return type**
ndarray

**Returns**
An array with *ndim == len(self.matrix_shape) + len(self.grid.shape)* and with each non-singleton dimension matching those of the nb_rows and data_shape.

`transversal_projection`(*field_array*)

Projects vector arrays onto their transverse component. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The input vector E array.

**Return type**
ndarray

**Returns**
The transversal projection.

`transversal_projection_ft`(*field_array_ft*)

Projects the Fourier transform of a vector E array onto its transversal component.

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray]) – The Fourier transform of the input vector E array.

**Return type**
ndarray

**Returns**
The Fourier transform of the transversal projection.

`property vector_length: int`

The shape of the square matrix that transforms a single vector in the set. This is a pair of identical integer numbers.

`property vectorial: bool`

A boolean indicating if this object represents a vector space (as opposed to a scalar space).

### macromax.backend.tensorflow module

The module providing the TensorFlow back-end implementation.

`class macromax.backend.tensorflow.BackEndTensorFlow`(*nb_dims*, *grid*, *hardware_dtype=tensorflow.complex128*, *device=None*, *address=None*)

Bases: BackEnd

A class that provides methods to work with arrays of matrices or block-diagonal matrices, represented as ndarrays, where the first two dimensions are those of the matrix, and the final dimensions are the coordinates over which the operations are parallelized and the Fourier transforms are applied.

`__init__`(*nb_dims*, *grid*, *hardware_dtype=tensorflow.complex128*, *device=None*, *address=None*)

Construct object to handle parallel operations on square matrices of nb_rows x nb_rows elements. The matrices refer to points in space on a uniform plaid grid.

**Parameters**

- **nb_dims** (int) – The number of rows and columns in each matrix. 1 for scalar operations, 3 for polarization

- **grid** (*Grid*) – The grid that defines the position of the matrices.
- **hardware_dtype** – (optional) The data type to use for operations.
- **device** (Optional[str]) – (optional) 'cpu', 'gpu', or 'tpu' to indicate where the calculation will happen.
- **address** (Optional[str]) – (optional)

**property numpy_dtype**

The equivalent hardware data type in numpy

**property eps: float**

The precision of the data type (self.dtype) of this back-end.

**astype**(*arr*, *dtype=None*)

As necessary, convert the ndarray arr to the type dtype.

**Return type**
Tensor

**asnumpy**(*arr*)

Convert the internal array (or tensor) presentation to a numpy.ndarray.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray, Tensor]) – The to-be-converted array.

**Return type**
ndarray

**Returns**
The corresponding numpy ndarray.

**assign**(*arr*, *out*)

Assign the values of one array to those of another. The dtype and shape is broadcasted to match that of the output array.

**Parameters**

- **arr** – The values that need to be assigned to another array.
- **out** – (optional) The target array, which must be of the same shape.

**Return type**
Tensor

**Returns**
The target array or a newly allocated array with the same values as those in arr.

**assign_exact**(*arr*, *out*)

Assign the values of one array to those of another. The dtype and shape must match that of the output array.

**Parameters**

- **arr** – The values that need to be assigned to another array.
- **out** (Tensor) – (optional) The target array, which must be of the same shape.

**Return type**
Tensor

**Returns**
The target array or a newly allocated array with the same values as those in arr.

**allocate_array**(*shape=None*, *dtype=None*, *fill_value=None*)

Allocates a new vector array of shape grid.shape and word-aligned for efficient calculations.

**Return type**
Tensor

**copy**(*arr*)

Makes an independent copy of an ndarray.

**Return type**
Tensor

**ravel**(*arr*)

Returns a flattened view of the array.

**Return type**
Tensor

**sign**(*arr*)

Returns an array with values of -1 where arr is negative, 0 where arr is 0, and 1 where arr is positive.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray, Tensor]) – The array to check.

**Return type**
Tensor

**Returns**
np.sign(arr) or equivalent.

**swapaxes**(*arr*, *ax_from*, *ax_to*)

Transpose (permute) two axes of an ndarray.

**Return type**
Tensor

**static expand_dims**(*arr*, *axis*)

Inserts a new singleton axis at the indicated position, thus increasing ndim by 1.

**Return type**
Tensor

**abs**(*arr*)

Returns the absolute value (magnitude) of the elements in the input.

**Parameters**
**arr** – The input array.

**Return type**
Tensor

**Returns**
np.abs(arr) or equivalent

**real**(*arr*)

**Return type**
Tensor

**conj**(*arr*)

Returns the conjugate of the elements in the input.

**Parameters**
**arr** – The input array.

**Return type**
Tensor

**Returns**
arr.conj or equivalent

**any**(*arr*)

Returns True if all elements of the array are True.

**allclose**(*arr*, *other=0.0*)

Returns True if all elements in arr are close to other.

**Return type**
bool

**amax**(*arr*)

Returns the maximum of the flattened array.

**sort**(*arr*)

Sorts array elements along the first (left-most) axis.

**Return type**
Tensor

**ft**(*arr*)

Calculates the discrete Fourier transform over the spatial dimensions of E. The computational complexity is that of a Fast Fourier Transform: `O(N\log(N))`.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray, Tensor]) – An ndarray representing a vector field.

**Return type**
Tensor

**Returns**
An ndarray holding the Fourier transform of the vector field E.

**ift**(*arr*)

Calculates the inverse Fourier transform over the spatial dimensions of E. The computational complexity is that of a Fast Fourier Transform: `O(N\log(N))`. The scaling is so that `E == self.ift(self.ft(E))`

**Parameters**
**arr** (Union[Complex, Sequence, ndarray, Tensor]) – An ndarray representing a Fourier-transformed vector field.

**Return type**
Tensor

**Returns**
An ndarray holding the inverse Fourier transform of the vector field E.

**convolve**(*operation_ft*, *arr*)

Apply an operator in Fourier space. This is used to perform a cyclic FFT convolution which overwrites its input argument *arr*.

**Parameters**

- **operation_ft** (Callable[[Union[Complex, Sequence, ndarray, Tensor]], Union[Complex, Sequence, ndarray, Tensor]]) – The function that acts on the Fourier-transformed input.
- **arr** (Union[Complex, Sequence, ndarray, Tensor]) – The to-be-convolved argument array.

**Return type**
Tensor

**Returns**
the convolved input array.

**adjoint**(*mat*)

Transposes the elements of individual matrices with complex conjugation.

**Parameters**
**mat** (Union[Complex, Sequence, ndarray, Tensor]) – The ndarray with the matrices in the first two dimensions.

**Return type**
Tensor

**Returns**
An ndarray with the complex conjugate transposed matrices.

**subtract**(*left_term*, *right_term*)

Point-wise difference of A and B.

**Parameters**

- **left_term** (Union[Complex, Sequence, ndarray, Tensor]) – The left matrix array, must start with dimensions n x m
- **right_term** (Union[Complex, Sequence, ndarray, Tensor]) – The right matrix array, must have matching or singleton dimensions to those of A. In case of missing dimensions, singletons are assumed.

**Return type**
ndarray

**Returns**
The point-wise difference of both sets of matrices. Singleton dimensions are expanded.

**is_scalar**(*arr*)

Tests if A represents a scalar field (as opposed to a vector field).

**Parameters**
**arr** (Union[Complex, Sequence, ndarray, Tensor]) – The ndarray to be tested.

**Return type**
bool

**Returns**
A boolean indicating whether A represents a scalar field (True) or not (False).

**mul**(*left_factor*, *right_factor*, *out=None*)

Point-wise matrix multiplication of A and B. Overwrites right_factor!

**Parameters**

- **left_factor** (Union[Complex, Sequence, ndarray, Tensor]) – The left matrix array, must start with dimensions n x m

- **right_factor** (Union[Complex, Sequence, ndarray, Tensor]) – The right matrix array, must have matching or singleton dimensions to those of A, bar the first two dimensions. In case of missing dimensions, singletons are assumed. The first dimensions must be m x p. Where the m matches that of the left hand matrix unless both m and p are 1 or both n and m are 1, in which case the scaled identity is assumed.
- **out** (Optional[Tensor]) – (optional) The destination array for the results.

**Return type**
Tensor

**Returns**
An array of matrix products with all but the first two dimensions broadcast as needed.

**ldivide**(*denominator*, *numerator=1.0*)

Parallel matrix left division, A^{-1}B, on the final two dimensions of A and B result_lm = A_kl B_km

A and B must have have all but the final dimension identical or singletons. B defaults to the identity matrix.

**Parameters**

- **denominator** (Union[Complex, Sequence, ndarray, Tensor]) – The set of denominator matrices.
- **numerator** (Union[Complex, Sequence, ndarray, Tensor]) – The set of numerator matrices.

**Return type**
Tensor

**Returns**
The set of divided matrices.

**norm**(*arr*)

Returns the l2-norm of a vectorized array.

**Return type**
float

**longitudinal_projection_ft**(*field_array_ft*)

Projects the Fourier transform of a vector array onto its longitudinal component. Overwrites self.array_ft_input!

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray, Tensor]) – The Fourier transform of the input vector E array.

**Return type**
ndarray

**Returns**
The Fourier transform of the longitudinal projection.

**transversal_projection_ft**(*field_array_ft*)

Projects the Fourier transform of a vector E array onto its transversal component.

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray, Tensor]) – The Fourier transform of the input vector E array.

**Return type**
Tensor

**Returns**
The Fourier transform of the transversal projection.

**div**(*field_array*)

Calculates the divergence of input field_array.

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray, Tensor]) – The input array representing the field in all spatial dimensions. The input is of the shape `[m, n, x, y, z, ...]`

**Return type**
Tensor

**Returns**
The divergence of the field in the shape `[n, 1, x, y, z]`.

**curl_ft**(*field_array_ft*)

Calculates the Fourier transform of the curl of a Fourier transformed E with the final dimension the vector dimension. The final dimension of the output will always be of length 3; however, the input length may be shorter, in which case the missing values are assumed to be zero. The first dimension of the input array corresponds to the first element in the final dimension, if it exists, the second dimension corresponds to the second element etc.

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray, Tensor]) – The input vector array of dimensions `[vector_length, 1, *data_shape]`.

**Return type**
Tensor

**Returns**
The Fourier transform of the curl of F.

**property array_ft_input: ndarray**

The pre-allocate array for Fourier Transform inputs

**property array_ft_output: ndarray**

The pre-allocate array for Fourier Transform outputs

**property array_ift_input: ndarray**

The pre-allocate array for Inverse Fourier Transform inputs

**property array_ift_output: ndarray**

The pre-allocate array for Inverse Fourier Transform outputs

**calc_roots_of_low_order_polynomial**(*C*)

Calculates the (complex) roots of polynomials up to order 3 in parallel. The coefficients of the polynomials are in the first dimension of C and repeated in the following dimensions, one for each polynomial to determine the roots of. The coefficients are input for low to high order in each column. In other words, the polynomial is: `np.sum(C * (x**range(C.size))) == 0`

This method is used by mat3_eigh, but only for polynomials with real roots.

**Parameters**
**C** (Union[Complex, Sequence, ndarray]) – The coefficients of the polynomial, per polynomial.

**Return type**
ndarray

**Returns**
The zeros in the complex plane, per polynomial.

`static clear_cache()`

`cross(A, B)`

Calculates the cross product of vector arrays A and B.

**Parameters**

- **A** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`
- **B** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`

**Return type**
ndarray

**Returns**
A vector array of dimensions `[vector_length, 1, *data_shape]` containing the cross product A x B in the first dimension and the other dimensions remain on the same axes.

`curl(field_array)`

Calculates the curl of a vector E with the final dimension the vector dimension. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The set of input matrices.

**Return type**
ndarray

**Returns**
The curl of E.

`div_ft(field_array_ft)`

Calculate the Fourier transform of the divergence of the pre-Fourier transformed input electric field.

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray]) – The input array representing the field pre-Fourier-transformed in all spatial dimensions. The input is of the shape `[m, n, x, y, z, ...]`

**Return type**
ndarray

**Returns**
The Fourier transform of the divergence of the field in the shape `[n, 1, x, y, z]`.

`static evaluate_polynomial(C, X)`

Evaluates the polynomial P at X for testing.

**Parameters**

- **C** (Union[Complex, Sequence, ndarray]) – The coefficients of the polynomial, for each polynomial.
- **X** (Union[Complex, Sequence, ndarray]) – The argument of the polynomial, per polynomial.

**Return type**
ndarray

**Returns**
The values of the polynomials for the arguments X.

**property eye: ndarray**

Returns an identity tensor that can be multiplied using singleton expansion. This can be useful for scalar additions or subtractions.

**Returns**
an array with the number of dimensions matching that of the BackEnds's data set.

**first**(*arr*)

Returns the first element of the flattened array.

**Return type**
Complex

**property ft_axes: tuple**

The integer indices of the spatial dimensions, i.e. on which to Fourier Transform.

**property grid: *Grid***

A Grid object representing the sample points in the spatial dimensions.

**property hardware_dtype**

The scalar data type that is processed by this back-end.

**inv**(*mat*)

Inverts the set of input matrices M.

**Parameters**
**mat** (Union[Complex, Sequence, ndarray]) – The set of input matrices.

**Return type**
ndarray

**Returns**
The set of inverted matrices.

**is_matrix**(*arr*)

Checks if an ndarray is a matrix as defined by this parallel_ops_column object.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The matrix to be tested.

**Return type**
bool

**Returns**
boolean value, indicating if A is a matrix.

**is_vector**(*arr*)

Tests if A represents a vector field.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The ndarray to be tested.

**Return type**
bool

**Returns**
A boolean indicating whether A represents a vector field (True) or not (False).

**property k:** Sequence[ndarray]

A list of the k-vector components along each axis.

**property k2:** ndarray

Helper def for calculation of the Fourier transform of the Green def

**Returns**
$|k|^2$ for the specified sample grid and output shape

**longitudinal_projection**(*field_array*)

Projects vector arrays onto their longitudinal component. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The input vector E array.

**Return type**
ndarray

**Returns**
The longitudinal projection.

**mat3_eigh**(*arr*)

Calculates the eigenvalues of the 3x3 Hermitian matrices represented by A and returns a new array of 3-vectors, one for each matrix in A and of the same dimensions, baring the second dimension. When the first two dimensions are 3x1 or 1x3, a diagonal matrix is assumed. When the first two dimensions are singletons (1x1), a constant diagonal matrix is assumed and only one eigenvalue is returned. Returns an array of one dimension less: 3 x data_shape. With the exception of the first dimension, the shape is maintained.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray, Tensor]) – The set of 3x3 input matrices for which the eigenvalues are requested. This must be an ndarray with the first two dimensions of size 3.

**Return type**
Tensor

**Returns**
The set of eigenvalue-triples contained in an ndarray with its first dimension of size 3, and the remaining dimensions equal to all but the first two input dimensions.

**outer**(*A*, *B*)

Calculates the Dyadic product of vector arrays A and B.

**Parameters**

- **A** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`
- **B** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`

**Return type**
ndarray

**Returns**
A matrix array of dimensions `[vector_length, vector_length, *data_shape]` containing the dyadic product $A \otimes B$ in the first two dimensions and the other dimensions remain on the same axes.

`to_matrix_field`(*arr*)

Converts the input to an array of the full number of dimensions: len(self.matrix_shape) + len(self.grid.shape), and dtype. The size of each dimensions must match that of that set for the `BackEnd` or be 1 (assumes broadcasting). For electric fields in 3-space, self.matrix_shape == (N, N) == (3, 3) The first (left-most) dimensions of the output are either

- 1x1: The identity matrix for a scalar field, as sound waves or isotropic permittivity.
- Nx1: A vector for a vector field, as the electric field.
- NxN: A matrix for a matrix field, as anisotropic permittivity

None is interpreted as 0. Singleton dimensions are added on the left so that all dimensions are present. Inputs with 1xN are transposed (not conjugate) to Nx1 vectors.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The input can be scalar, which assumes that its value is assumed to be repeated for all space. The value can be a one-dimensional vector, in which case the vector is assumed to be repeated for all space.

**Return type**
ndarray

**Returns**
An array with *ndim == len(self.matrix_shape) + len(self.grid.shape)* and with each non-singleton dimension matching those of the nb_rows and data_shape.

`transversal_projection`(*field_array*)

Projects vector arrays onto their transverse component. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The input vector E array.

**Return type**
ndarray

**Returns**
The transversal projection.

`property vector_length: int`

The shape of the square matrix that transforms a single vector in the set. This is a pair of identical integer numbers.

`property vectorial: bool`

A boolean indicating if this object represents a vector space (as opposed to a scalar space).

### macromax.backend.torch module

The module providing the PyTorch back-end implementation.

`class macromax.backend.torch.BackEndTorch`(*nb_dims*, *grid*, *hardware_dtype=torch.complex128*, *device=None*)

Bases: `BackEnd`

A class that provides methods to work with arrays of matrices or block-diagonal matrices, represented as ndarrays, where the first two dimensions are those of the matrix, and the final dimensions are the coordinates over which the operations are parallelized and the Fourier transforms are applied.

**__init__**(*nb_dims*, *grid*, *hardware_dtype=torch.complex128*, *device=None*)

Construct object to handle parallel operations on square matrices of nb_rows x nb_rows elements. The matrices refer to points in space on a uniform plaid grid.

**Parameters**

- **nb_dims** (int) – The number of rows and columns in each matrix. 1 for scalar operations, 3 for polarization
- **grid** (*Grid*) – The grid that defines the position of the matrices.
- **hardware_dtype** – (optional) The data type to use for operations.
- **device** (Optional[str]) – (optional) 'cuda' or 'cpu', to indicate where the calculation will happen.

**property numpy_dtype**

The equivalent hardware data type in numpy

**property eps: float**

The precision of the data type (self.dtype) of this back-end.

**astype**(*arr*, *dtype=None*)

As necessary, convert the ndarray arr to the type dtype.

**Return type**
Tensor

**asnumpy**(*arr*)

Convert the internal array (or tensor) presentation to a numpy.ndarray.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray, Tensor]) – The to-be-converted array.

**Return type**
ndarray

**Returns**
The corresponding numpy ndarray.

**assign**(*arr*, *out*)

Assign the values of one array to those of another. The dtype and shape is broadcasted to match that of the output array.

**Parameters**

- **arr** – The values that need to be assigned to another array.
- **out** – (optional) The target array, which must be of the same shape.

**Return type**
Tensor

**Returns**
The target array or a newly allocated array with the same values as those in arr.

**assign_exact**(*arr*, *out*)

Assign the values of one array to those of another. The dtype and shape must match that of the output array.

**Parameters**

- **arr** – The values that need to be assigned to another array.
- **out** – (optional) The target array, which must be of the same shape.

**Return type**
Tensor

**Returns**
The target array or a newly allocated array with the same values as those in arr.

**allocate_array**(*shape=None*, *dtype=None*, *fill_value=None*)

Allocates a new vector array of shape grid.shape and word-aligned for efficient calculations.

**Return type**
Tensor

**copy**(*arr*)

Makes an independent copy of an ndarray.

**Return type**
Tensor

**ravel**(*arr*)

Returns a flattened view of the array.

**Return type**
Tensor

**sign**(*arr*)

Returns an array with values of -1 where arr is negative, 0 where arr is 0, and 1 where arr is positive.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray, Tensor]) – The array to check.

**Return type**
Tensor

**Returns**
np.sign(arr) or equivalent.

**swapaxes**(*arr*, *ax_from*, *ax_to*)

Transpose (permute) two axes of an ndarray.

**Return type**
Tensor

**static expand_dims**(*arr*, *axis*)

Inserts a new singleton axis at the indicated position, thus increasing ndim by 1.

**Return type**
Tensor

**abs**(*arr*)

Returns the absolute value (magnitude) of the elements in the input.

**Parameters**
**arr** – The input array.

**Return type**
Tensor

**Returns**
np.abs(arr) or equivalent

`conj`(*arr*)

Returns the conjugate of the elements in the input.

**Parameters**
**arr** – The input array.

**Return type**
Tensor

**Returns**
arr.conj or equivalent

`any`(*arr*)

Returns True if all elements of the array are True.

`allclose`(*arr*, *other=0.0*)

Returns True if all elements in arr are close to other.

**Return type**
bool

`amax`(*arr*)

Returns the maximum of the flattened array.

`sort`(*arr*)

Sorts array elements along the first (left-most) axis.

**Return type**
Tensor

`ft`(*arr*)

Calculates the discrete Fourier transform over the spatial dimensions of E. The computational complexity is that of a Fast Fourier Transform: `O(N\log(N))`.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray, Tensor]) – An ndarray representing a vector field.

**Return type**
Tensor

**Returns**
An ndarray holding the Fourier transform of the vector field E.

`ift`(*arr*)

Calculates the inverse Fourier transform over the spatial dimensions of E. The computational complexity is that of a Fast Fourier Transform: `O(N\log(N))`. The scaling is so that `E == self.ift(self.ft(E))`

**Parameters**
**arr** (Union[Complex, Sequence, ndarray, Tensor]) – An ndarray representing a Fourier-transformed vector field.

**Return type**
Tensor

**Returns**
An ndarray holding the inverse Fourier transform of the vector field E.

`adjoint`(*mat*)

Transposes the elements of individual matrices with complex conjugation.

**Parameters**
**mat** (Union[Complex, Sequence, ndarray, Tensor]) – The ndarray with the matrices in the first two dimensions.

**Return type**
Tensor

**Returns**
An ndarray with the complex conjugate transposed matrices.

**real**(*arr*)

**Return type**
Tensor

**mul**(*left_factor*, *right_factor*, *out=None*)

Point-wise matrix multiplication of A and B. Overwrites right_factor!

**Parameters**

- **left_factor** (Union[Complex, Sequence, ndarray, Tensor]) – The left matrix array, must start with dimensions n x m
- **right_factor** (Union[Complex, Sequence, ndarray, Tensor]) – The right matrix array, must have matching or singleton dimensions to those of A, bar the first two dimensions. In case of missing dimensions, singletons are assumed. The first dimensions must be m x p. Where the m matches that of the left hand matrix unless both m and p are 1 or both n and m are 1, in which case the scaled identity is assumed.
- **out** (Optional[Tensor]) – (optional) The destination array for the results.

**Return type**
Tensor

**Returns**
An array of matrix products with all but the first two dimensions broadcast as needed.

**ldivide**(*denominator*, *numerator=1.0*)

Parallel matrix left division, A^{-1}B, on the final two dimensions of A and B result_lm = A_kl B_km

A and B must have have all but the final dimension identical or singletons. B defaults to the identity matrix.

**Parameters**

- **denominator** (Union[Complex, Sequence, ndarray, Tensor]) – The set of denominator matrices.
- **numerator** (Union[Complex, Sequence, ndarray, Tensor]) – The set of numerator matrices.

**Return type**
Tensor

**Returns**
The set of divided matrices.

**static clear_cache**()

**property array_ft_input: ndarray**

The pre-allocate array for Fourier Transform inputs

**property array_ft_output: ndarray**

The pre-allocate array for Fourier Transform outputs

**property array_ift_input: ndarray**

The pre-allocate array for Inverse Fourier Transform inputs

**property array_ift_output: ndarray**

The pre-allocate array for Inverse Fourier Transform outputs

**calc_roots_of_low_order_polynomial**(*C*)

Calculates the (complex) roots of polynomials up to order 3 in parallel. The coefficients of the polynomials are in the first dimension of C and repeated in the following dimensions, one for each polynomial to determine the roots of. The coefficients are input for low to high order in each column. In other words, the polynomial is: `np.sum(C * (x**range(C.size))) == 0`

This method is used by mat3_eigh, but only for polynomials with real roots.

**Parameters**
**C** (Union[Complex, Sequence, ndarray]) – The coefficients of the polynomial, per polynomial.

**Return type**
ndarray

**Returns**
The zeros in the complex plane, per polynomial.

**convolve**(*operation_ft*, *arr*)

Apply an operator in Fourier space. This is used to perform a cyclic FFT convolution which overwrites its input argument *arr*.

**Parameters**

- **operation_ft** (Callable[[Union[Complex, Sequence, ndarray]], Union[Complex, Sequence, ndarray]]) – The function that acts on the Fourier-transformed input.
- **arr** (Union[Complex, Sequence, ndarray]) – The to-be-convolved argument array.

**Return type**
ndarray

**Returns**
the convolved input array.

**cross**(*A*, *B*)

Calculates the cross product of vector arrays A and B.

**Parameters**

- **A** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`
- **B** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`

**Return type**
ndarray

**Returns**
A vector array of dimensions `[vector_length, 1, *data_shape]` containing the cross product A x B in the first dimension and the other dimensions remain on the same axes.

**curl**(*field_array*)

Calculates the curl of a vector E with the final dimension the vector dimension. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The set of input matrices.

**Return type**
ndarray

**Returns**
The curl of E.

**curl_ft**(*field_array_ft*)

Calculates the Fourier transform of the curl of a Fourier transformed E with the final dimension the vector dimension. The final dimension of the output will always be of length 3; however, the input length may be shorter, in which case the missing values are assumed to be zero. The first dimension of the input array corresponds to the first element in the final dimension, if it exists, the second dimension corresponds to the second element etc.

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray]) – The input vector array of dimensions `[vector_length, 1, *data_shape]`.

**Return type**
ndarray

**Returns**
The Fourier transform of the curl of F.

**div**(*field_array*)

Calculate the divergence of the input vector or tensor field E. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The input array representing vector or tensor field. The input is of the shape `[m, n, x, y, z, ...]`

**Return type**
ndarray

**Returns**
The divergence of the vector or tensor field in the shape `[n, 1, x, y, z]`.

**div_ft**(*field_array_ft*)

Calculate the Fourier transform of the divergence of the pre-Fourier transformed input electric field.

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray]) – The input array representing the field pre-Fourier-transformed in all spatial dimensions. The input is of the shape `[m, n, x, y, z, ...]`

**Return type**
ndarray

**Returns**
The Fourier transform of the divergence of the field in the shape `[n, 1, x, y, z]`.

**static evaluate_polynomial**(*C*, *X*)

Evaluates the polynomial P at X for testing.

**Parameters**

- **C** (Union[Complex, Sequence, ndarray]) – The coefficients of the polynomial, for each polynomial.

- **X** (Union[Complex, Sequence, ndarray]) – The argument of the polynomial, per polynomial.

**Return type**
ndarray

**Returns**
The values of the polynomials for the arguments X.

**property eye: ndarray**

Returns an identity tensor that can be multiplied using singleton expansion. This can be useful for scalar additions or subtractions.

**Returns**
an array with the number of dimensions matching that of the BackEnds's data set.

**first**(*arr*)

Returns the first element of the flattened array.

**Return type**
Complex

**property ft_axes: tuple**

The integer indices of the spatial dimensions, i.e. on which to Fourier Transform.

**property grid: *Grid***

A Grid object representing the sample points in the spatial dimensions.

**property hardware_dtype**

The scalar data type that is processed by this back-end.

**inv**(*mat*)

Inverts the set of input matrices M.

**Parameters**
**mat** (Union[Complex, Sequence, ndarray]) – The set of input matrices.

**Return type**
ndarray

**Returns**
The set of inverted matrices.

**is_matrix**(*arr*)

Checks if an ndarray is a matrix as defined by this parallel_ops_column object.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The matrix to be tested.

**Return type**
bool

**Returns**
boolean value, indicating if A is a matrix.

**is_scalar**(*arr*)

Tests if A represents a scalar field (as opposed to a vector field).

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The ndarray to be tested.

**Return type**
bool

**Returns**
A boolean indicating whether A represents a scalar field (True) or not (False).

**is_vector**(*arr*)

Tests if A represents a vector field.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The ndarray to be tested.

**Return type**
bool

**Returns**
A boolean indicating whether A represents a vector field (True) or not (False).

**property k:** Sequence[ndarray]

A list of the k-vector components along each axis.

**property k2:** ndarray

Helper def for calculation of the Fourier transform of the Green def

**Returns**
$|k|^2$ for the specified sample grid and output shape

**longitudinal_projection**(*field_array*)

Projects vector arrays onto their longitudinal component. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The input vector E array.

**Return type**
ndarray

**Returns**
The longitudinal projection.

**longitudinal_projection_ft**(*field_array_ft*)

Projects the Fourier transform of a vector array onto its longitudinal component. Overwrites self.array_ft_input!

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray]) – The Fourier transform of the input vector E array.

**Return type**
ndarray

**Returns**
The Fourier transform of the longitudinal projection.

**mat3_eigh**(*arr*)

Calculates the eigenvalues of the 3x3 Hermitian matrices represented by A and returns a new array of 3-vectors, one for each matrix in A and of the same dimensions, baring the second dimension. When the first two dimensions are 3x1 or 1x3, a diagonal matrix is assumed. When the first two dimensions are singletons (1x1), a constant diagonal matrix is assumed and only one eigenvalue is returned. Returns an array of one dimension less: 3 x data_shape. With the exception of the first dimension, the shape is maintained.

Before substituting this for `numpy.linalg.eigvalsh`, note that this implementation is about twice as fast as the current numpy implementation for 3x3 Hermitian matrix fields. The difference is even greater

for the PyTorch implementation. The implementation below can be made more efficient by limiting it to Hermitian matrices and thus real eigenvalues. In the end, only the maximal eigenvalue is required, so an iterative power iteration may be even faster, perhaps after applying a Given's rotation or Householder reflection to make Hermitian tridiagonal.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The set of 3x3 input matrices for which the eigenvalues are requested. This must be an ndarray with the first two dimensions of size 3.

**Return type**
ndarray

**Returns**
The set of eigenvalue-triples contained in an ndarray with its first dimension of size 3, and the remaining dimensions equal to all but the first two input dimensions.

**norm**(*arr*)

Returns the l2-norm of a vectorized array.

**Return type**
float

**outer**(*A*, *B*)

Calculates the Dyadic product of vector arrays A and B.

**Parameters**

- **A** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`
- **B** (Union[Complex, Sequence, ndarray]) – A vector array of dimensions `[vector_length, 1, *data_shape]`

**Return type**
ndarray

**Returns**
A matrix array of dimensions `[vector_length, vector_length, *data_shape]` containing the dyadic product $A \otimes B$ in the first two dimensions and the other dimensions remain on the same axes.

**subtract**(*left_term*, *right_term*)

Point-wise difference of A and B.

**Parameters**

- **left_term** (Union[Complex, Sequence, ndarray]) – The left matrix array, must start with dimensions n x m
- **right_term** (Union[Complex, Sequence, ndarray]) – The right matrix array, must have matching or singleton dimensions to those of A. In case of missing dimensions, singletons are assumed.

**Return type**
ndarray

**Returns**
The point-wise difference of both sets of matrices. Singleton dimensions are expanded.

**to_matrix_field**(*arr*)

Converts the input to an array of the full number of dimensions: len(self.matrix_shape) + len(self.grid.shape), and dtype. The size of each dimensions must match that of that set for the `BackEnd`

or be 1 (assumes broadcasting). For electric fields in 3-space, self.matrix_shape == (N, N) == (3, 3) The first (left-most) dimensions of the output are either

- 1x1: The identity matrix for a scalar field, as sound waves or isotropic permittivity.
- Nx1: A vector for a vector field, as the electric field.
- NxN: A matrix for a matrix field, as anisotropic permittivity

None is interpreted as 0. Singleton dimensions are added on the left so that all dimensions are present. Inputs with 1xN are transposed (not conjugate) to Nx1 vectors.

**Parameters**
**arr** (Union[Complex, Sequence, ndarray]) – The input can be scalar, which assumes that its value is assumed to be repeated for all space. The value can be a one-dimensional vector, in which case the vector is assumed to be repeated for all space.

**Return type**
ndarray

**Returns**
An array with *ndim == len(self.matrix_shape) + len(self.grid.shape)* and with each non-singleton dimension matching those of the nb_rows and data_shape.

**transversal_projection**(*field_array*)

Projects vector arrays onto their transverse component. The input argument may be overwritten!

**Parameters**
**field_array** (Union[Complex, Sequence, ndarray]) – The input vector E array.

**Return type**
ndarray

**Returns**
The transversal projection.

**transversal_projection_ft**(*field_array_ft*)

Projects the Fourier transform of a vector E array onto its transversal component.

**Parameters**
**field_array_ft** (Union[Complex, Sequence, ndarray]) – The Fourier transform of the input vector E array.

**Return type**
ndarray

**Returns**
The Fourier transform of the transversal projection.

**property vector_length: int**

The shape of the square matrix that transforms a single vector in the set. This is a pair of identical integer numbers.

**property vectorial: bool**

A boolean indicating if this object represents a vector space (as opposed to a scalar space).

### macromax.utils package

This module contains functionality that is not directly related to the rest of the library and can be used independently.

### Subpackages

### macromax.utils.array package

This module provides additional functionality to work with nd-arrays.

### Submodules

### macromax.utils.array.add_dims_on_right module

macromax.utils.array.add_dims_on_right.**add_dims_on_right**(*arr*, *new_axes_on_right=0*, *ndim=None*)

A function that returns a view with additional singleton dimensions on the right. This is useful to broadcast and expand on the left-hand side with an array of an arbitrary number of dimensions on the right-hand side.

**Parameters**

- **arr** (Union[Complex, Sequence, ndarray]) – The original array or sequence of numbers that can be converted to an array.
- **new_axes_on_right** (int) – (optional) The number of axes to add on the right hand side. Default: ndim - arr.ndim or 0 if the latter is not specified. If negative, singleton dimensions are removed from the right. This will fail if those on the right are not singleton dimensions.
- **ndim** (Optional[int]) – (optional) The total number of axes of the returned view. Default: arr.ndim + new_axes_on_right

**Returns**

A view with ndim == arr.ndim + new_axes_on_right dimensions.

### macromax.utils.array.vector_to_axis module

macromax.utils.array.vector_to_axis.**vector_to_axis**(*vec*, *axis=0*, *ndim=None*)

Adds singleton dimensions to a 1D vector up to dimension n and orients the vector in dimension axis (default 0)

**Parameters**

- **vec** – the input vector
- **axis** (int) – the target axis (default: 0)
- **ndim** (Optional[int]) – the number of desired dimensions. Default: axis

**Returns**

an n-dimensional array with all-but-one singleton dimension

### macromax.utils.array.word_align module

macromax.utils.array.word_align.**word_align**(*input_array*, *word_length=32*)

Returns a new array that is byte-aligned to words of length word_length bytes. This may be required for libraries such as pyfftw.

**Parameters**

- **input_array** – The input array to align.
- **word_length** (int) – The word length to align to. This must be an integer multiple of the dtype size.

**Returns**

A word-aligned array with the same contents and shape as input_array.

### macromax.utils.display package

This package contains functionality to simplify the display of complex matrices.

### Submodules

### macromax.utils.display.colormap module

**class** macromax.utils.display.colormap.**InterpolatedColorMap**(*name*, *colors*, *points=None*)

Bases: LinearSegmentedColormap

A custom colormap for use with imshow and colorbar.

Example usage:

::

cmap = colormap.InterpolatedColorMap('hsv', [(0, 0, 0), (1, 0, 0), (1, 1, 0), (0, 1, 0), (0, 1, 1), (0, 0, 1), (1, 0, 1), (1, 0, 0), (1, 1, 1)]) cmap = colormap.InterpolatedColorMap('rainbow', [(0, 0, 0), (1, 0, 0), (0.75, 0.75, 0), (0, 1, 0), (0, 0.75, 0.75), (0, 0, 1), (0.75, 0, 0.75), (1, 1, 1)],

points=[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 1])

fig, ax = subplots(1, 1) ax.imshow(intensity_array, cmap=cmap) from matplotlib import cm fig.colorbar(cm.ScalarMappable(norm=None, cmap=cmap), ax=ax)

**__init__**(*name*, *colors*, *points=None*)

Create colormap from linear mapping segments

segmentdata argument is a dictionary with a red, green and blue entries. Each entry should be a list of *x*, *y0*, *y1* tuples, forming rows in a table. Entries for alpha are optional.

Example: suppose you want red to increase from 0 to 1 over the bottom half, green to do the same over the middle half, and blue over the top half. Then you would use:

```
cdict = {'red':   [(0.0,  0.0, 0.0),
                   (0.5,  1.0, 1.0),
                   (1.0,  1.0, 1.0)],

         'green': [(0.0,  0.0, 0.0),
                   (0.25, 0.0, 0.0),
```

(continues on next page)

(continued from previous page)

```
                   (0.75, 1.0, 1.0),
                   (1.0,  1.0, 1.0)],

         'blue':  [(0.0,  0.0, 0.0),
                   (0.5,  0.0, 0.0),
                   (1.0,  1.0, 1.0)]}
```

Each row in the table for a given color is a sequence of *x*, *y0*, *y1* tuples. In each sequence, *x* must increase monotonically from 0 to 1. For any input value *z* falling between *x[i]* and *x[i+1]*, the output value of a given color will be linearly interpolated between *y1[i]* and *y0[i+1]*:

```
row i:   x  y0  y1
               /
              /
row i+1: x  y0  y1
```

Hence y0 in the first row and y1 in the last row are never used.

**See also:**

`LinearSegmentedColormap.from_list`
Static method; factory function for generating a smoothly-varying LinearSegmentedColormap.

`__call__`(*X*, *alpha=None*, *bytes=False*)

**Parameters**

- **X** (*float or int, ndarray or scalar*) – The data value(s) to convert to RGBA. For floats, X should be in the interval `[0.0, 1.0]` to return the RGBA values `X*100` percent along the Colormap line. For integers, X should be in the interval `[0, Colormap.N)` to return RGBA values *indexed* from the Colormap with index `X`.
- **alpha** (*float or array-like or None*) – Alpha must be a scalar between 0 and 1, a sequence of such floats with shape matching X, or None.
- **bytes** (*bool*) – If False (default), the returned RGBA values will be floats in the interval `[0, 1]` otherwise they will be uint8s in the interval `[0, 255]`.

**Returns**

- *Tuple of RGBA values if X is scalar, otherwise an array of*
- RGBA values with a shape of `X.shape + (4, )`.

`__copy__`()

`__eq__`(*other*)
Return self==value.

`copy`()
Return a copy of the colormap.

`static from_list`(*name*, *colors*, *N=256*, *gamma=1.0*)
Create a *LinearSegmentedColormap* from a list of colors.

**Parameters**

- **name** (*str*) – The name of the colormap.

- **colors** (*array-like of colors or array-like of (value, color)*) – If only colors are given, they are equidistantly mapped from the range $[0, 1]$; i.e. 0 maps to `colors[0]` and 1 maps to `colors[-1]`. If (value, color) pairs are given, the mapping is from *value* to *color*. This can be used to divide the range unevenly.
- **N** (*int*) – The number of rgb quantization levels.
- **gamma** (*float*)

`get_bad()`

Get the color for masked values.

`get_over()`

Get the color for high out-of-range values.

`get_under()`

Get the color for low out-of-range values.

`is_gray()`

Return whether the colormap is grayscale.

`reversed`(*name=None*)

Return a reversed instance of the Colormap.

**Parameters**
**name** (*str, optional*) – The name for the reversed colormap. If it's None the name will be the name of the parent colormap + "_r".

**Returns**
The reversed colormap.

**Return type**
LinearSegmentedColormap

`set_bad`(*color='k'*, *alpha=None*)

Set the color for masked values.

`set_extremes`(*, *bad=None*, *under=None*, *over=None*)

Set the colors for masked (*bad*) values and, when `norm.clip = False`, low (*under*) and high (*over*) out-of-range values.

`set_gamma`(*gamma*)

Set a new gamma value and regenerate colormap.

`set_over`(*color='k'*, *alpha=None*)

Set the color for high out-of-range values.

`set_under`(*color='k'*, *alpha=None*)

Set the color for low out-of-range values.

`with_extremes`(*, *bad=None*, *under=None*, *over=None*)

Return a copy of the colormap, for which the colors for masked (*bad*) values and, when `norm.clip = False`, low (*under*) and high (*over*) out-of-range values, have been set accordingly.

`colorbar_extend`

When this colormap exists on a scalar mappable and colorbar_extend is not False, colorbar creation will pick up `colorbar_extend` as the default value for the `extend` keyword in the *matplotlib.colorbar.Colorbar* constructor.

### macromax.utils.display.complex2rgb module

macromax.utils.display.complex2rgb.**complex2rgb**(*complex_image*, *normalization=None*, *inverted=False*, *alpha=None*, *dtype=<class 'float'>*, *axis=-1*)

Converts a complex image to an RGB image.

**Parameters**

- **complex_image** (Union[complex, Sequence, array]) – A 2D array
- **normalization** (Union[bool, float, int, None]) – An optional multidimensional to indicate the target magnitude of the maximum value (1.0 is saturation).
- **inverted** (bool) – By default 0 is shown as black and amplitudes of 1 as the brightest hues. Setting this input argument to True could be useful for printing on a white background.
- **alpha** (Optional[float]) – The maximum alpha value (0 = transparent, 1 is opaque). When specified, each pixel's alpha value is proportional to the intensity times this value. Default: None = no alpha channel.
- **dtype** – The output data type. The value is scaled to the maximum positive numeric range for integers (np.iinfo(dtype).max). Floating point numbers are within [0, 1]. (Default: float)
- **axis** (int) – (default: -1) The channel axis of the output array, and also the input array if neither saturation, nor value are provided.

**Return type**
ndarray

**Returns**
A real 3d-array with values between 0 and 1 if the the dtype is a float and covering the dynamic range when the dtype is an integer.

### macromax.utils.display.grid2extent module

macromax.utils.display.grid2extent.**grid2extent**(**args*, *origin_lower=False*)

Utility function to determine extent values for matplotlib.pyploy.imshow

**Parameters**

- **args** – A Grid object or monotonically increasing ranges, one per dimension (vertical, horizontal)
- **origin_lower** (bool) – (default: False) Set this to True when imshow has the origin set to 'lower' to have the vertical axis increasing upwards.

**Returns**
An nd-array with 4 numbers indicating the extent of the displayed data.

### macromax.utils.display.hsv module

macromax.utils.display.hsv.**hsv2rgb**(*hue=None*, *saturation=None*, *value=None*, *alpha=None*, *axis=-1*)

Converts a hue-saturation-intensity value image to a red-green-blue image.

**Parameters**

- **hue** (Union[float, Sequence, ndarray, None]) – A 2D numpy.array with the hue. If the saturation is not provided, the first argument will be interpreted as a 3D numpy.array with the HSV image, the channel must be in the final right-hand dimension.
- **saturation** (Union[float, Sequence, ndarray, None]) – (optional) a 2D numpy.array with the saturation.
- **value** (Union[float, Sequence, ndarray, None]) – (optional) a 2D numpy.array with the intensity value.
- **alpha** (Union[float, Sequence, ndarray, None]) – (optional) a 2D numpy.array with the opaqueness alpha value.
- **axis** (int) – (default: -1) The channel axis of the output array, and also the input array if neither saturation, nor value are provided.

**Return type**
ndarray

**Returns**
rgb_image: a 3D numpy.array with the RGB image, the channel in the final right-hand dimension, or axis if provided.

macromax.utils.display.hsv.**rgb2hsv**(*red*, *green=None*, *blue=None*, *alpha=None*, *axis=-1*)

Converts a red-green-blue value image to a hue-saturation-intensity image.

**Parameters**

- **red** (Union[float, Sequence, ndarray]) – An 2D numpy.array with the red channel. If neither green and blue are provided, then this will be interpreted as a stack of the red, green, and blue channels.
- **green** (Union[float, Sequence, ndarray, None]) – (optional) a 2D numpy.array with the green channel.
- **blue** (Union[float, Sequence, ndarray, None]) – (optional) a 2D numpy.array with the blue channel.
- **alpha** (Union[float, Sequence, ndarray, None]) – (optional) a 2D numpy.array with the opaqueness alpha value.
- **axis** – (default: -1) The channel axis of the output array, and also the input array if neither saturation, nor value are provided.

**Return type**
ndarray

**Returns**
hsv_image: a 3D numpy.array with the HSV image

### macromax.utils.ft package

### Submodules

### macromax.utils.ft.ft_implementation module

The *ft* module provides direct access to *numpy.fft*, *scipy.fftpack*, *pyfftw*, or *mkl_fft*, depending on availability. The back-end that is deemed most efficient is automatically imported. Independently of the back-end, the *ft* module provides at least the following functions:

- *ft.fft(a: array_like, axis: int)*: The 1-dimensional fast Fourier transform.
- *ft.ifft(a: array_like, axis: int)*: The 1-dimensional fast Fourier transform.
- *ft.fftn(a: array_like, axes: Sequence)*: The n-dimensional fast Fourier transform.
- *ft.ifftn(a: array_like, axes: Sequence)*: The n-dimensional inverse fast Fourier transform.
- *ft.fftshift(a: array_like, axes: Sequence)*: This fftshifts the input array.
- *ft.ifftshift(a: array_like, axes: Sequence)*: This ifftshifts the input array (only different from *ft.fftshift* for odd-shapes).

All functions return a numpy.ndarray of the same shape as the input array or array_like, *a*. With the exception of *ft.fft()* and *ft.ifft()*, all functions take the *axes* argument to limit the action to specific axes of the numpy.ndarray.

Note that axis indices should be unique and non-negative. **Negative or repeated axis indices are not compatible with all back-end implementations!**

`macromax.utils.ft.ft_implementation.`**`fftshift`**`(`*x*, *axes=None*`)`

Shift the zero-frequency component to the center of the spectrum.

This function swaps half-spaces for all axes listed (defaults to all). Note that `y[0]` is the Nyquist component only if `len(x)` is even.

**Parameters**

- **x** (*array_like*) – Input array.
- **axes** (*int or shape tuple, optional*) – Axes over which to shift. Default is None, which shifts all axes.

**Returns**

**y** – The shifted array.

**Return type**

ndarray

**See also:**

*`ifftshift`*

The inverse of *fftshift*.

### Examples

```
>>> freqs = np.fft.fftfreq(10, 0.1)
>>> freqs
array([ 0.,  1.,  2., ..., -3., -2., -1.])
>>> np.fft.fftshift(freqs)
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
```

Shift the zero-frequency component only along the second axis:

```
>>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
>>> np.fft.fftshift(freqs, axes=(1,))
array([[ 2.,  0.,  1.],
       [-4.,  3.,  4.],
       [-1., -3., -2.]])
```

macromax.utils.ft.ft_implementation.**ifftshift**(*x*, *axes=None*)

The inverse of *fftshift*. Although identical for even-length *x*, the functions differ by one sample for odd-length *x*.

**Parameters**

- **x** (*array_like*) – Input array.
- **axes** (*int or shape tuple, optional*) – Axes over which to calculate. Defaults to None, which shifts all axes.

**Returns**

**y** – The shifted array.

**Return type**

ndarray

**See also:**

*fftshift*

Shift zero-frequency component to the center of the spectrum.

### Examples

```
>>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
>>> np.fft.ifftshift(np.fft.fftshift(freqs))
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
```

macromax.utils.ft.ft_implementation.**fft**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*)

Return discrete Fourier transform of real or complex sequence.

The returned complex array contains `y(0), y(1),..., y(n-1)`, where

`y(j) = (x * exp(-2*pi*sqrt(-1)*j*np.arange(n)/n)).sum()`.

**Parameters**

- **x** (*array_like*) – Array to Fourier transform.
- **n** (*int, optional*) – Length of the Fourier transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.
- **axis** (*int, optional*) – Axis along which the fft's are computed; the default is over the last axis (i.e., `axis=-1`).
- **overwrite_x** (*bool, optional*) – If True, the contents of *x* can be destroyed; the default is False.

**Returns**

**z** –

with the elements:

```
[y(0),y(1),..,y(n/2),y(1-n/2),...,y(-1)]        if n is even
[y(0),y(1),..,y((n-1)/2),y(-(n-1)/2),...,y(-1)]  if n is odd
```

where:

```
y(j) = sum[k=0..n-1] x[k] * exp(-sqrt(-1)*j*k* 2*pi/n), j = 0..n-1
```

**Return type**
complex ndarray

**See also:**

***ifft***
Inverse FFT

**rfft**
FFT of a real sequence

## Notes

The packing of the result is "standard": If `A = fft(a, n)`, then `A[0]` contains the zero-frequency term, `A[1:n/2]` contains the positive-frequency terms, and `A[n/2:]` contains the negative-frequency terms, in order of decreasingly negative frequency. So ,for an 8-point transform, the frequencies of the result are [0, 1, 2, 3, -4, -3, -2, -1]. To rearrange the fft output so that the zero-frequency component is centered, like [-4, -3, -2, -1, 0, 1, 2, 3], use *fftshift*.

Both single and double precision routines are implemented. Half precision inputs will be converted to single precision. Non-floating-point inputs will be converted to double precision. Long-double precision inputs are not supported.

This function is most efficient when *n* is a power of two, and least efficient when *n* is prime.

Note that if `x` is real-valued, then `A[j] == A[n-j].conjugate()`. If `x` is real-valued and `n` is even, then `A[n/2]` is real.

If the data type of *x* is real, a "real FFT" algorithm is automatically used, which roughly halves the computation time. To increase efficiency a little further, use *rfft*, which does the same calculation, but only outputs half of

the symmetrical spectrum. If the data is both real and symmetrical, the *dct* can again double the efficiency by generating half of the spectrum from half of the signal.

### Examples

```
>>> from scipy.fftpack import fft, ifft
>>> x = np.arange(5)
>>> np.allclose(fft(ifft(x)), x, atol=1e-15)  # within numerical accuracy.
True
```

macromax.utils.ft.ft_implementation.**ifft**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*)

Return discrete inverse Fourier transform of real or complex sequence.

The returned complex array contains `y(0), y(1),..., y(n-1)`, where

`y(j) = (x * exp(2*pi*sqrt(-1)*j*np.arange(n)/n)).mean()`.

**Parameters**

- **x** (*array_like*) – Transformed data to invert.
- **n** (*int, optional*) – Length of the inverse Fourier transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.
- **axis** (*int, optional*) – Axis along which the ifft's are computed; the default is over the last axis (i.e., `axis=-1`).
- **overwrite_x** (*bool, optional*) – If True, the contents of *x* can be destroyed; the default is False.

**Returns**

**ifft** – The inverse discrete Fourier transform.

**Return type**

ndarray of floats

**See also:**

*fft*

Forward FFT

### Notes

Both single and double precision routines are implemented. Half precision inputs will be converted to single precision. Non-floating-point inputs will be converted to double precision. Long-double precision inputs are not supported.

This function is most efficient when *n* is a power of two, and least efficient when *n* is prime.

If the data type of *x* is real, a "real IFFT" algorithm is automatically used, which roughly halves the computation time.

### Examples

```
>>> from scipy.fftpack import fft, ifft
>>> import numpy as np
>>> x = np.arange(5)
>>> np.allclose(ifft(fft(x)), x, atol=1e-15)  # within numerical accuracy.
True
```

macromax.utils.ft.ft_implementation.**fftn**(*x*, *shape=None*, *axes=None*, *overwrite_x=False*)

Return multidimensional discrete Fourier transform.

The returned array contains:

```
y[j_1,..,j_d] = sum[k_1=0..n_1-1, ..., k_d=0..n_d-1]
   x[k_1,..,k_d] * prod[i=1..d] exp(-sqrt(-1)*2*pi/n_i * j_i * k_i)
```

where d = len(x.shape) and n = x.shape.

**Parameters**

- **x** (*array_like*) – The (N-D) array to transform.
- **shape** (*int or array_like of ints or None, optional*) – The shape of the result. If both *shape* and *axes* (see below) are None, *shape* is `x.shape`; if *shape* is None but *axes* is not None, then *shape* is `numpy.take(x.shape, axes, axis=0)`. If `shape[i] > x.shape[i]`, the ith dimension is padded with zeros. If `shape[i] < x.shape[i]`, the ith dimension is truncated to length `shape[i]`. If any element of *shape* is -1, the size of the corresponding dimension of *x* is used.
- **axes** (*int or array_like of ints or None, optional*) – The axes of *x* (*y* if *shape* is not None) along which the transform is applied. The default is over all axes.
- **overwrite_x** (*bool, optional*) – If True, the contents of *x* can be destroyed. Default is False.

**Returns**

**y** – The (N-D) DFT of the input array.

**Return type**

complex-valued N-D NumPy array

**See also:**

*ifftn*

### Notes

If `x` is real-valued, then `y[..., j_i, ...] == y[..., n_i-j_i, ...].conjugate()`.

Both single and double precision routines are implemented. Half precision inputs will be converted to single precision. Non-floating-point inputs will be converted to double precision. Long-double precision inputs are not supported.

### Examples

```
>>> from scipy.fftpack import fftn, ifftn
>>> y = (-np.arange(16), 8 - np.arange(16), np.arange(16))
>>> np.allclose(y, fftn(ifftn(y)))
True
```

macromax.utils.ft.ft_implementation.**ifftn**(*x*, *shape=None*, *axes=None*, *overwrite_x=False*)

Return inverse multidimensional discrete Fourier transform.

The sequence can be of an arbitrary type.

The returned array contains:

```
y[j_1,..,j_d] = 1/p * sum[k_1=0..n_1-1, ..., k_d=0..n_d-1]
   x[k_1,..,k_d] * prod[i=1..d] exp(sqrt(-1)*2*pi/n_i * j_i * k_i)
```

where `d = len(x.shape)`, `n = x.shape`, and `p = prod[i=1..d] n_i`.

For description of parameters see *fftn*.

**See also:**

*fftn*
: for detailed information.

### Examples

```
>>> from scipy.fftpack import fftn, ifftn
>>> import numpy as np
>>> y = (-np.arange(16), 8 - np.arange(16), np.arange(16))
>>> np.allclose(y, ifftn(fftn(y)))
True
```

## macromax.utils.ft.grid module

**class** macromax.utils.ft.grid.**Grid**(*shape=None*, *step=None*, *extent=None*, *first=None*, *center=None*, *last=None*, *include_last=False*, *ndim=None*, *flat=False*, *origin_at_center=True*, *center_at_index=True*)

Bases: `Sequence`

A class representing an immutable uniformly-spaced plaid Cartesian grid and its Fourier Transform.

See also *MutableGrid*

**__init__**(*shape=None*, *step=None*, *extent=None*, *first=None*, *center=None*, *last=None*, *include_last=False*, *ndim=None*, *flat=False*, *origin_at_center=True*, *center_at_index=True*)

Construct an immutable Grid object.

**Parameters**

- **shape** – An integer vector array with the shape of the sampling grid.
- **step** – A vector array with the spacing of the sampling grid.
- **extent** – The extent of the sampling grid as shape * step

- **first** – A vector array with the first element for each dimension. The first element is the smallest element if step is positive, and the largest when step is negative.
- **center** – A vector array with the center element for each dimension. The center position in the grid is rounded to the next integer index unless center_at_index is set to False for that partical axis.
- **last** – A vector array with the last element for each dimension. Unless include_last is set to True for the associated dimension, all but the last element is returned when calling self[axis].
- **include_last** – A boolean vector array indicating whether the returned vectors, self[axis], should include the last element (True) or all-but-the-last (False)
- **ndim** (Optional[int]) – A scalar integer indicating the number of dimensions of the sampling space.
- **flat** (Union[bool, Sequence, ndarray]) – A boolean vector array indicating whether the returned vectors, self[axis], should be flattened (True) or returned as an open grid (False)
- **origin_at_center** (Union[bool, Sequence, ndarray]) – A boolean vector array indicating whether the origin should be fft-shifted (True) or be ifftshifted to the front (False) of the returned vectors for self[axis].
- **center_at_index** (Union[bool, Sequence, ndarray]) – A boolean vector array indicating whether the center of the grid should be rounded to an integer index for each dimension. If False and the shape has an even number of elements, the next index is used as the center, (self.shape / 2).astype(int).

**static from_ranges**(**ranges*)

Converts one or more ranges of numbers to a single Grid object representation. The ranges can be specified as separate parameters or as a tuple.

**Parameters**
**ranges** (Union[int, float, complex, Sequence, ndarray]) – one or more ranges of uniformly spaced numbers.

**Return type**
*Grid*

**Returns**
A Grid object that represents the same ranges.

**property ndim:** int

The number of dimensions of the space this grid spans.

**property shape:** array

The number of sample points along each axis of the grid.

**property step:** ndarray

The sample spacing along each axis of the grid.

**property center:** ndarray

The central coordinate of the grid.

**property center_at_index:** array

Boolean vector indicating whether the central coordinate is aligned with a grid point when the number of points is even along the associated axis. This has no effect when the the number of sample points is odd.

**property flat: array**

Boolean vector indicating whether self[axis] returns flattened (raveled) vectors (True) or not (False).

**property origin_at_center: array**

Boolean vector indicating whether self[axis] returns ranges that are monotonous (True) or ifftshifted so that the central index is the first element of the sequence (False).

**property as_flat: *Grid***

return: A new Grid object where all the ranges are 1d-vectors (flattened or raveled)

**property as_non_flat: *Grid***

return: A new Grid object where all the ranges are 1d-vectors (flattened or raveled)

**property as_origin_at_0: *Grid***

return: A new Grid object where all the ranges are ifftshifted so that the origin as at index 0.

**property as_origin_at_center: *Grid***

return: A new Grid object where all the ranges have the origin at the center index, even when the number of elements is odd.

**swapaxes**(*axes*)

Reverses the order of the specified axes.

**Return type**
*Grid*

**transpose**(*axes=None*)

Reverses the order of all axes.

**Return type**
*Grid*

**project**(*axes_to_keep=None*, *axes_to_remove=None*)

Removes all but the specified axes and reduces the dimensions to the number of specified axes.

**Parameters**

- **axes_to_keep** (Union[int, slice, Sequence, array, None]) – The indices of the axes to keep.
- **axes_to_remove** (Union[int, slice, Sequence, array, None]) – The indices of the axes to remove. Default: None

**Return type**
*Grid*

**Returns**
A Grid object with ndim == len(axes) and shape == shape[axes].

**property first: ndarray**

return: A vector with the first element of each range

**property extent: ndarray**

The spatial extent of the sampling grid.

**property size: int**

The total number of sampling points as an integer scalar.

**property dtype**

The numeric data type for the coordinates.

**property f:** *Grid*

The equivalent frequency Grid.

**property k:** *Grid*

The equivalent k-space Grid.

**__add__**(*term*)

Add a (scalar) offset to the Grid coordinates.

**Return type**
*Grid*

**__mul__**(*factor*)

Scales all ranges with a factor.

**Parameters**
**factor** (Union[int, float, complex, Sequence, array]) – A scalar factor for all dimensions, or a vector of factors, one for each dimension.

**Return type**
*Grid*

**Returns**
A new scaled Grid object.

**__matmul__**(*other*)

Determines the Grid spanning the tensor space, with ndim equal to the sum of both ndims.

**Parameters**
**other** (*Grid*) – The Grid with the right-hand dimensions.

**Return type**
*Grid*

**Returns**
A new Grid with ndim == self.ndim + other.ndim.

**__sub__**(*term*)

Subtract a (scalar) value from all Grid coordinates.

**Return type**
*Grid*

**__truediv__**(*denominator*)

Divide the grid coordinates by a value.

**Parameters**
**denominator** (Union[int, float, complex, Sequence, ndarray]) – The denominator to divide by.

**Return type**
*Grid*

**Returns**
A new Grid with the divided coordinates.

**__neg__**()

Invert the coordinate values and the direction of the axes.

**__len__**()

The number of axes in this sampling grid. Or, the number of elements when this object is not multi-dimensional.

**Return type**
int

**__iter__**()

**property immutable:** *Grid*

Return a new immutable Grid object.

**property mutable:** *MutableGrid*

return: A new MutableGrid object.

**__eq__**(*other*)

Compares two Grid objects.

**Return type**
bool

**property multidimensional:** bool

Single-dimensional grids behave as Sequences, multi-dimensional behave as a Sequence of vectors.

**classmethod __init_subclass__**(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

**static __new__**(*cls*, **args*, ***kwds*)

**count**(*value*) → integer -- return number of occurrences of value

**index**(*value*[, *start*[, *stop* ]])  → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

**class** macromax.utils.ft.grid.**MutableGrid**(*shape=None*, *step=None*, *extent=None*, *first=None*, *center=None*, *last=None*, *include_last=False*, *ndim=None*, *flat=False*, *origin_at_center=True*, *center_at_index=True*)

Bases: *Grid*

A class representing a mutable uniformly-spaced plaid Cartesian grid and its Fourier Transform.

See also *Grid*

**__init__**(*shape=None*, *step=None*, *extent=None*, *first=None*, *center=None*, *last=None*, *include_last=False*, *ndim=None*, *flat=False*, *origin_at_center=True*, *center_at_index=True*)

Construct a mutable Grid object.

**Parameters**

- **shape** – An integer vector array with the shape of the sampling grid.
- **step** – A vector array with the spacing of the sampling grid.
- **extent** – The extent of the sampling grid as shape * step
- **first** – A vector array with the first element for each dimension. The first element is the smallest element if step is positive, and the largest when step is negative.

- **center** – A vector array with the center element for each dimension. The center position in the grid is rounded to the next integer index unless center_at_index is set to False for that partical axis.
- **last** – A vector array with the last element for each dimension. Unless include_last is set to True for the associated dimension, all but the last element is returned when calling self[axis].
- **include_last** – A boolean vector array indicating whether the returned vectors, self[axis], should include the last element (True) or all-but-the-last (False)
- **ndim** (Optional[int]) – A scalar integer indicating the number of dimensions of the sampling space.
- **flat** (Union[bool, Sequence, ndarray]) – A boolean vector array indicating whether the returned vectors, self[axis], should be flattened (True) or returned as an open grid (False)
- **origin_at_center** (Union[bool, Sequence, ndarray]) – A boolean vector array indicating whether the origin should be fft-shifted (True) or be ifftshifted to the front (False) of the returned vectors for self[axis].
- **center_at_index** (Union[bool, Sequence, ndarray]) – A boolean vector array indicating whether the center of the grid should be rounded to an integer index for each dimension. If False and the shape has an even number of elements, the next index is used as the center, (self.shape / 2).astype(int).

**property shape: array**

The number of sample points along each axis of the grid.

**property step: ndarray**

The sample spacing along each axis of the grid.

**property center: ndarray**

The central coordinate of the grid.

**property flat: array**

Boolean vector indicating whether self[axis] returns flattened (raveled) vectors (True) or not (False).

**property origin_at_center: array**

Boolean vector indicating whether self[axis] returns ranges that are monotonous (True) or ifftshifted so that the central index is the first element of the sequence (False).

**__add__**(*term*)

Add a (scalar) offset to the Grid coordinates.

**Return type**
*Grid*

**__eq__**(*other*)

Compares two Grid objects.

**Return type**
bool

**classmethod __init_subclass__**(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

**__iter__**()

__len__()

The number of axes in this sampling grid. Or, the number of elements when this object is not multi-dimensional.

**Return type**
int

__matmul__(*other*)

Determines the Grid spanning the tensor space, with ndim equal to the sum of both ndims.

**Parameters**
**other** (*Grid*) – The Grid with the right-hand dimensions.

**Return type**
*Grid*

**Returns**
A new Grid with ndim == self.ndim + other.ndim.

__mul__(*factor*)

Scales all ranges with a factor.

**Parameters**
**factor** (Union[int, float, complex, Sequence, array]) – A scalar factor for all dimensions, or a vector of factors, one for each dimension.

**Return type**
*Grid*

**Returns**
A new scaled Grid object.

__neg__()

Invert the coordinate values and the direction of the axes.

static __new__(*cls*, **args*, ***kwds*)

__sub__(*term*)

Subtract a (scalar) value from all Grid coordinates.

**Return type**
*Grid*

__truediv__(*denominator*)

Divide the grid coordinates by a value.

**Parameters**
**denominator** (Union[int, float, complex, Sequence, ndarray]) – The denominator to divide by.

**Return type**
*Grid*

**Returns**
A new Grid with the divided coordinates.

property as_flat: *Grid*

return: A new Grid object where all the ranges are 1d-vectors (flattened or raveled)

property as_non_flat: *Grid*

return: A new Grid object where all the ranges are 1d-vectors (flattened or raveled)

**property as_origin_at_0:** *Grid*

return: A new Grid object where all the ranges are ifftshifted so that the origin as at index 0.

**property as_origin_at_center:** *Grid*

return: A new Grid object where all the ranges have the origin at the center index, even when the number of elements is odd.

**property center_at_index:** array

Boolean vector indicating whether the central coordinate is aligned with a grid point when the number of points is even along the associated axis. This has no effect when the the number of sample points is odd.

**count**(*value*) → integer -- return number of occurrences of value

**property extent:** ndarray

The spatial extent of the sampling grid.

**property f:** *Grid*

The equivalent frequency Grid.

**property first:** ndarray

return: A vector with the first element of each range

**static from_ranges**(**ranges*)

Converts one or more ranges of numbers to a single Grid object representation. The ranges can be specified as separate parameters or as a tuple.

**Parameters**
**ranges** (Union[int, float, complex, Sequence, ndarray]) – one or more ranges of uniformly spaced numbers.

**Return type**
*Grid*

**Returns**
A Grid object that represents the same ranges.

**property immutable:** *Grid*

Return a new immutable Grid object.

**index**(*value*[, *start*[, *stop* ]])  → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

**property k:** *Grid*

The equivalent k-space Grid.

**property multidimensional:** bool

Single-dimensional grids behave as Sequences, multi-dimensional behave as a Sequence of vectors.

**property mutable:** *MutableGrid*

return: A new MutableGrid object.

**property ndim:** int

The number of dimensions of the space this grid spans.

**project**(*axes_to_keep=None*, *axes_to_remove=None*)

Removes all but the specified axes and reduces the dimensions to the number of specified axes.

**Parameters**

- **axes_to_keep** (Union[int, slice, Sequence, array, None]) – The indices of the axes to keep.
- **axes_to_remove** (Union[int, slice, Sequence, array, None]) – The indices of the axes to remove. Default: None

**Return type**
*Grid*

**Returns**
A Grid object with ndim == len(axes) and shape == shape[axes].

**property size: int**

The total number of sampling points as an integer scalar.

**swapaxes**(*axes*)

Reverses the order of the specified axes.

**Return type**
*Grid*

**transpose**(*axes=None*)

Reverses the order of all axes.

**Return type**
*Grid*

**property dtype**

The numeric data type for the coordinates.

**__iadd__**(*number*)

**__imul__**(*number*)

**__isub__**(*number*)

**__idiv__**(*number*)

### macromax.utils.ft.subpixel module

Classes and functions to register a subject array to a reference array with subpixel precision. This is based on the algorithm described in: Manuel Guizar-Sicairos, Samuel T. Thurman, and James R. Fienup, "Efficient subpixel image registration algorithms," Optics Letters. 33, 156-158 (2008).

macromax.utils.ft.subpixel.**register**(*subject=None*, *reference=None*, *precision=None*)

Registers a subject array to a reference array with subpixel precision. This is based on the algorithm described in Manuel Guizar-Sicairos, Samuel T. Thurman, and James R. Fienup, "Efficient subpixel image registration algorithms," Optics Letters. 33, 156-158 (2008).

**Parameters**

- **subject** (Union[int, float, Sequence, ndarray, None]) – The subject image array (n-dimensional) .
- **reference** (Union[int, float, Sequence, ndarray, None]) – The optional reference array (n-dimensional).
- **precision** (Optional[float]) – The registration precision in units of fracional pixels (default).

**Return type**
*Registration*

**Returns**
A Registration instance describing the registered image, the shift, and the amplitude.

macromax.utils.ft.subpixel.**roll**(*subject*, *shift*, *axis=None*)

Rolls (shifting with wrapping around) and nd-array with sub-pixel precision.

**Parameters**

- **subject** (ndarray) – The to-be-shifted nd-array. Default: all zeros except first element.
- **shift** (Union[int, float, Sequence, ndarray, None]) – The (fractional) shift. Default: all zeros except first element.
- **axis** (Union[int, Sequence, ndarray, None]) – Optional int or sequence of ints. The axis or axes along which elements are shifted. Unlike numpy's roll function, by default, the left-most axes are used.

**Returns**
The shifted nd-array.

macromax.utils.ft.subpixel.**roll_ft**(*subject_ft*, *shift*, *axes=None*)

Rolls (shifting with wrapping around) and nd-array with sub-pixel precision. The input and output array are Fourier transformed.

**Parameters**

- **subject_ft** (ndarray) – The Fourier transform of the to-be-shifted nd-array.
- **shift** (Union[int, float, Sequence, ndarray, None]) – The (fractional) shift.
- **axes** (Union[int, Sequence, ndarray, None]) – Optional int or sequence of ints. The axis or axes along which elements are shifted. Unlike numpy's roll function, by default, the left-most axes are used.

**Returns**
The Fourier transform of the shifted nd-array.

**class** macromax.utils.ft.subpixel.**Registration**(*shift*, *factor=1.0*, *error=0.0*, *original_ft=None*, *original=None*, *registered_ft=None*, *registered=None*)

Bases: object

A class to represent the result of a registration of reference class.

**__init__**(*shift*, *factor=1.0*, *error=0.0*, *original_ft=None*, *original=None*, *registered_ft=None*, *registered=None*)

Constructs a registration result.

**Parameters**

- **shift** (Union[int, float, Sequence, ndarray]) – translation in pixels of the registered with respect to original reference
- **factor** (complex) – scaling factor between registered and original reference image
- **error** (float) – The root-mean-square difference after registration. See `register` for more details.
- **original_ft** (Union[int, float, Sequence, ndarray, None]) – The Fourier transform of the original image, prior to registration.

- **original** (Union[int, float, Sequence, ndarray, None]) – The original image, prior to registration.
- **registered_ft** (Union[int, float, Sequence, ndarray, None]) – The Fourier transform of the registered image, identical to the original but with a sub-pixel shift.
- **registered** (Union[int, float, Sequence, ndarray, None]) – The registered image, identical to the original but with a sub-pixel shift.

**property shift: ndarray**

Vector indicating subpixel shift between the original and registration image.

**property ndim: int**

The number of registration dimensions.

**property factor**

(Complex) scaling factor indicating the ratio between original and registered image.

**property error: float**

The rms difference between the registered and the original image including rescaling factor.

**property image_ft: ndarray**

Fourier transform of the registered and renormalized array.

**property image: ndarray**

Registered and renormalized array. It is shifted and scaled so that it is as close as possible to the reference. I.e. it minimizes the l2-norm of the difference.

**__array__()**

Return the registered and renormalized image as an array.

**Return type**
ndarray

**property original_ft: ndarray**

Fourier transform of the original reference array.

**property original: ndarray**

Original reference array.

**class** macromax.utils.ft.subpixel.**Reference**(*reference_image=None*, *precision=None*, *axes=None*, *ndim=None*, *reference_image_ft=None*)

Bases: object

Represents a reference 'image' (which can be n-dimensional) to be used to register against. Multi-channel arrays should be handled iteratively or by averaging over the channels.

**__init__**(*reference_image=None*, *precision=None*, *axes=None*, *ndim=None*, *reference_image_ft=None*)

Construct a reference 'image' object. If neither *reference_image* or its Fourier transform *reference_image_ft* are specified, a point-reference is assumed, where the point is in the first element of the nd-array (top-left corner).

**Parameters**

- **reference_image** (Union[int, float, Sequence, ndarray, None]) – The reference image nd-array. As an alternative, its unshifted Fourier transform can be specified as reference_image_ft.
- **precision** (Optional[float]) – (optional) The default sub-pixel precision (default: 1/128).

- **axes** (Union[int, float, Sequence, ndarray, None]) – (optional) The axes to operate on. If not specified, all dimensions of the reference image or its Fourier transform are used.
- **ndim** (Optional[int]) – The number of dimensions is usually determined from . If neither is specified, ndim determines the number of dimensions to operate on.
- **reference_image_ft** (Union[int, float, Sequence, ndarray, None]) –

**property ndim:** int

**property shape**

**register**(*subject=None*, *subject_ft=None*, *precision=None*)

Register an nd-image to sub-pixel precision.

Algorithm based on the 2-d implementation of: Manuel Guizar-Sicairos, Samuel T. Thurman, and James R. Fienup, "Efficient subpixel image registration algorithms," Opt. Lett. 33, 156-158 (2008).

**Parameters**

- **subject** (Optional[array]) – The subject image as an nd-array. If this is not specified, its Fourier transform should be.
- **subject_ft** (Optional[array]) – (optional) The (non-fftshifted) Fourier transform of the subject image.
- **precision** (Optional[float]) – (optional) The sub-pixel precision in units of pixel (default: 1/128).

**Return type**
*Registration*

**Returns**
A Registration object representing the registered image as well as the shift, global phase change, and error.

## Submodules

### macromax.utils.beam module

**class** macromax.utils.beam.**Beam**(*grid*, *propagation_axis=None*, *vacuum_wavenumber=1.0*, *vacuum_wavelength=None*, *background_permittivity=1.0*, *background_refractive_index=None*, *field=None*, *field_ft=None*)

Bases: object

**__init__**(*grid*, *propagation_axis=None*, *vacuum_wavenumber=1.0*, *vacuum_wavelength=None*, *background_permittivity=1.0*, *background_refractive_index=None*, *field=None*, *field_ft=None*)

Represents the wave equation solution that is propagated using the beam-propagation method. The grid must be one dimension higher than that of the beam sections. The field or field_ft argument specify sections as for the associated BeamSection object.

**Parameters**

- **grid** (*Grid*) – The regularly-spaced grid at which to calculate the field values. This grid includes the propagation dimension (indicated the propagation_axis argument).
- **propagation_axis** (Optional[int]) – The propagation axes. Default: -grid.ndim.
- **vacuum_wavenumber** (Optional[Complex]) – (optional) The vacuum wavenumber in rad/m, can also be specified as wavelength.

- **vacuum_wavelength** (Optional[Real]) – (optional) Vacuum wavelength, alternative for wavenumber in units of meter.
- **background_permittivity** (Optional[Complex]) – (optional) The homogeneous background permittivity as a scalar, default 1. Heterogeneous distributions are specified when querying the results from this object.
- **background_refractive_index** (Optional[Complex]) – (optional) Alternative to the above, the square of the permittivity.
- **field** (Union[Complex, Sequence, array, Callable[[Union[Complex, Sequence, array], Union[Complex, Sequence, array]], Union[Complex, Sequence, array]], None]) – (optional) The field specification at propagation distance 0 as specified on the transverse dimensions of the grid.
- **field_ft** (Union[Complex, Sequence, array, Callable[[Union[Complex, Sequence, array], Union[Complex, Sequence, array]], Union[Complex, Sequence, array]], None]) – (optional) Alternative field specification as its fft (not fftshifted).

**property grid:** *Grid*

The grid of sample points at which the beam's field is calculated and at which the material properties are defined.

**property propagation_axis:** int

The propagation axis index in the polarization vector, i.e. the longitudinal polarization.

**property vectorial:** bool

Returns True if this is a vectorial beam propagation, and False if it is scalar.

**property shape:** ndarray

The shape of the returned `field` or `field_ft` array.

**property dtype**

The data type of the field return in the iterator and by the `Beam.field` method.

**beam_section_at_exit**(*permittivity=None*, *refractive_index=None*)

Propagates the field through the calculation volume and returns the beam section at the exit surface.

**Parameters**

- **permittivity** – (optional) The permittivity distribution within the calculation volume. Default: the background permittivity, unless refractive_index is specified.
- **refractive_index** – (optional) The refractive index distribution within the calculation volume. Default: the background refractive index, unless permittivity is specified.

**Return type**
*BeamSection*

**Returns**
The BeamSection object after propagating through the calculation volume.

**__iter__**(*permittivity=None*, *refractive_index=None*)

Iterator returning BeamSections for propagation in an arbitrary medium (by default homogeneous) A BeamSection object is generated _after_ propagating through each section of material.

**Parameters**

- **permittivity** (Union[Complex, Sequence, array, None]) – (optional) The (complex) permittivity (distribution) that the beam traverses prior to yielding each field-section. This should be a sequence/generator of permittivity slices or None.

- **refractive_index** (Union[Complex, Sequence, array, None]) – (optional) Alternative specification of the permittivity squared. This should be a sequence/generator of refractive index slices or None.

**Return type**
Generator[*BeamSection*, None, None]

**Returns**
A Generator object producing BeamSections.

**field**(*permittivity=None*, *refractive_index=None*, *out=None*)

Calculates the field at the grid points self.grid for the specified (or default) permittivity / refractive index.

**Parameters**

- **permittivity** (Union[Sequence, Iterable, None]) – (optional) The 3D permittivity distribution as a sequence or iterable.
- **refractive_index** (Union[Sequence, Iterable, None]) – (optional) The 3D refractive index distribution as a sequence or iterable.
- **out** (Union[Complex, Sequence, array, None]) – (optional) The output array where to store the result.

**Return type**
ndarray

**Returns**
A `numpy.ndarray` of which the final four dimensions are polarization, z, y, and x.

**__array__**()

Returns all field values. This is the same as Beam.field().

**Return type**
ndarray

**class** macromax.utils.beam.**BeamSection**(*grid*, *propagation_axis=-3*, *vacuum_wavenumber=1.0*, *vacuum_wavelength=None*, *background_permittivity=1.0*, *background_refractive_index=None*, *field=None*, *field_ft=None*, *dtype=None*, *vectorial=None*)

Bases: object

**__init__**(*grid*, *propagation_axis=-3*, *vacuum_wavenumber=1.0*, *vacuum_wavelength=None*, *background_permittivity=1.0*, *background_refractive_index=None*, *field=None*, *field_ft=None*, *dtype=None*, *vectorial=None*)

Represents a single transversal section of the wave equation solution that is propagated using the beam-propagation method. Its main purpose is to propagate the field from one plane to the next using the `propagate(...)` method, for use in the `Beam` class. The field can be scalar or vectorial. In the latter case, the polarization must be represented by axis -4 in the field and field_ft arrays. Singleton dimensions must be added for lower-dimensional calculations.

**Parameters**

- **grid** (*Grid*) – The regularly-spaced grid at which the field values are defined. This grid can have up to three dimensions. By default, the third dimension from the right is the propagation dimension. If the grid has less than 3 dimensions, new dimensions will be added on the left.
- **propagation_axis** (int) – The grid index of the propagation axis and that of the longitudinal polarization. Default: -3

- **vacuum_wavenumber** (Optional[Complex]) – (optional) The vacuum wavenumber in rad/m, can also be specified as wavelength.
- **vacuum_wavelength** (Optional[Complex]) – (optional) Vacuum wavelength, alternative for wavenumber in units of meter.
- **background_permittivity** (Optional[Complex]) – (optional) The homogeneous background permittivity as a scalar, default 1. Heterogeneous distributions are specified when querying the results from this object.
- **background_refractive_index** (Optional[Complex]) – (optional) Alternative to the above, the square of the permittivity.
- **field** (Union[Complex, Sequence, array, Callable[[Union[Complex, Sequence, array], Union[Complex, Sequence, array]], Union[Complex, Sequence, array]], None]) – (optional) The field specification (at propagation distance 0). If it is vectorial, the polarization axis should be 4th from the right (-4).
- **field_ft** (Union[Complex, Sequence, array, Callable[[Union[Complex, Sequence, array], Union[Complex, Sequence, array]], Union[Complex, Sequence, array]], None]) – (optional) Alternative field specification as its Fourier transform (not fftshifted). If the number of dimensions of field or field_ft equals that of the grid, a singleton dimension is prepended on the left to indicate a scalar field. Otherwise, the polarization_axis indicates whether this is a vectorial calculation or not. If it is vectorial, the polarization axis should be 4th from the right (-4).
- **dtype** – (optional) The complex dtype of the returned results. Default: that of field or field_ft.
- **vectorial** (Optional[bool]) – (optional) Indicates scalar (False) or vectorial = polarized (True) calculations. Default: consisted with the size of the polarization axis of the field or field_ft array.

**property grid:** *Grid*

The real space sampling grid.

**property transverse_grid:** *Grid*

The real space sampling grid of the transverse slice at the center. Its 3D shape is the same as the full 3D grid except for the propagation axis which has shape 1.

**property propagation_axis:** int

The propagation axis as a negative index in the inputs and outputs. This also corresponds to the longitudinal polarization.

**property polarization_axis:** int

The polarization axis as a negative index in the inputs and outputs. This is currently fixed to -4.

**property vectorial:** bool

Returns True if this is a vectorial beam propagation, and False if it is scalar.

**property shape:** ndarray

The shape of field (`BeamSection.field` or `BeamSection.field_ft`) of a single slice, including the polarisation dimension. I.e.: [3, 1, y, x], [3, 1, 1, x], [3, 1, 1, 1], [1, 1, y, x], [1, 1, 1, x], [1, 1, 1, 1], [3, z, 1, x], or … Dimensions on the left are added as broadcast.

**property ndim:** int

The number of dimensions of the values returned by the `BeamSection.field` and `BeamSection.field_ft` methods.

**property dtype**

The data type of the field return by the `BeamSection.field` and `BeamSection.field_ft` methods.

**property vacuum_wavenumber: Real**

The vacuum wavenumber of the wave being propagated.

**property wavenumber: Real**

The wavenumber of the wave being propagated in the background material.

**property vacuum_wavelength: Real**

The vacuum wavelength of the wave being propagated.

**property wavelength: Real**

The wavelength of the wave being propagated in the background material.

**property background_permittivity: Real**

The permittivity of the background medium. This is the permittivity that is assumed between scattering events.

**property background_refractive_index: Real**

The refractive index of the background medium. This is the refractive index that is assumed between scattering events.

**property field: array**

The electric field in the current section at the sample points are given by `BeamSection.transverse_grid`. The dimension to the left of that indicates the polarization. Scalar propagation was used if it is a singleton, vectorial propagation otherwise.

**__array__()**

Returns the current field values. This is the same as BeamSection.field

**Return type**
ndarray

**property field_ft: array**

The electric field in k-space, sampled at the grid specified by `BeamSection.k` (i.e. not fftshifted). The shape of the polarisation dimension on the left (axis -self.grid.ndim-1) will be 3 when doing a vectorial calculation and 1 when doing a scalar calculation.

**propagate**(*distance*, *permittivity=None*, *refractive_index=None*)

Propagates the beam section forward by a distance *distance*, optionally through a heterogeneous material with refractive index or permittivity as specified. If no material properties are specified, the homogeneous background medium is assumed. If the propagation distance is negative, time-reversal is assumed. I.e. the phases changes in the opposite direction and attenuation becomes gain.

The current BeamSection is updated and returned.

**Parameters**

- **distance** (float) – The distance (in meters) to propagate forward.
- **permittivity** (Union[Complex, Sequence, array, Callable[[Union[Complex, Sequence, array], Union[Complex, Sequence, array]], Union[Complex, Sequence, array]], None]) – The permittivity at the spatial grid points or as a function of (y, x).
- **refractive_index** (Union[Complex, Sequence, array, Callable[[Union[Complex, Sequence, array], Union[Complex, Sequence, array]], Union[Complex, Sequence, array]], None]) – The refractive index at the spatial grid points or as a function of (y, x).

**Return type**
*BeamSection*

**Returns**
The BeamSection propagated to the new position.

## Submodules

### macromax.bound module

The module provides the abstract *Bound* to represent the boundary of the simulation, e.g. periodic, or gradually more absorbing. Specific boundaries are implemented as subclasses and can be used directly as the *bound* argument to *macromax.solve()* or *macromax.Solution*. The precludes the inclusion of boundaries in the material description. It is sufficient to leave some space for the boundaries.

**class** macromax.bound.**Electric**

Bases: object

Mixin for Bound to indicate that the electric susceptibility is non-zero.

**property background_permittivity:** Complex

A complex scalar indicating the permittivity of the background.

**property electric_susceptibility:** ndarray

**property permittivity:** ndarray

The electric permittivity, epsilon, at every sample point. Note that the returned array may have singleton dimensions that must be broadcast!

**class** macromax.bound.**Magnetic**

Bases: object

Mixin for Bound to indicate that the magnetic susceptibility is non-zero.

**property magnetic_susceptibility:** ndarray

**property permeability:** ndarray

The magnetic permeability, mu, at every sample point. Note that the returned array may have singleton dimensions that must be broadcast!

**class** macromax.bound.**Bound**(*grid=None*, *thickness=0.0*, *background_permittivity=1.0*)

Bases: object

A base class to represent calculation-volume-boundaries. Use the subclasses for practical implementations.

**__init__**(*grid=None*, *thickness=0.0*, *background_permittivity=1.0*)

**Parameters**

- **grid** (Union[*Grid*, Sequence, ndarray, None]) – The Grid to which to the boundaries will be applied.
- **thickness** (Union[Real, Sequence, ndarray]) – The thickness as a scalar, vector, or 2d-array (axes x side). Broadcasting is used as necessary.
- **background_permittivity** (Complex) – The background permittivity of the boundary (default: 1.0 for vacuum). This is only used when the absolute permittivity is requested.

**property grid:** *Grid*

The Cartesian grid that indicates the sample positions of this bound and the volume it encompasses.

**property thickness:** ndarray

The thickness as a 2D-array *thickness[axis, front_back]* in meters.

**property background_permittivity:** Complex

A complex scalar indicating the permittivity of the background.

**property electric_susceptibility:** ndarray

The electric susceptibility, chi_E, at every sample point. Note that the returned array may have singleton dimensions that must be broadcast!

**property magnetic_susceptibility:** ndarray

The magnetic susceptibility, chi_H, at every sample point. Note that the returned array may have singleton dimensions that must be broadcast!

**property between:** ndarray

Returns a boolean array indicating True for the voxels between the boundaries where the calculation happens. The inner edge is considered in between the boundaries.

**property beyond:** ndarray

Returns a boolean array indicating True for the voxels beyond the boundary edge, i.e. inside the boundaries, where the calculation should be ignored.

**class** macromax.bound.**PeriodicBound**(*grid*)

Bases: *Bound*

**__init__**(*grid*)

Constructs an object that represents periodic boundaries.

**Parameters**
**grid** (Union[*Grid*, Sequence, ndarray]) – The Grid to which to the boundaries will be applied.

**property background_permittivity:** Complex

A complex scalar indicating the permittivity of the background.

**property between:** ndarray

Returns a boolean array indicating True for the voxels between the boundaries where the calculation happens. The inner edge is considered in between the boundaries.

**property beyond:** ndarray

Returns a boolean array indicating True for the voxels beyond the boundary edge, i.e. inside the boundaries, where the calculation should be ignored.

**property electric_susceptibility:** ndarray

The electric susceptibility, chi_E, at every sample point. Note that the returned array may have singleton dimensions that must be broadcast!

**property grid:** *Grid*

The Cartesian grid that indicates the sample positions of this bound and the volume it encompasses.

**property magnetic_susceptibility:** ndarray

The magnetic susceptibility, chi_H, at every sample point. Note that the returned array may have singleton dimensions that must be broadcast!

**property thickness:** ndarray

The thickness as a 2D-array *thickness[axis, front_back]* in meters.

class macromax.bound.**AbsorbingBound**(*grid*, *thickness=0.0*, *extinction_coefficient_function=<function AbsorbingBound.<lambda>>*, *background_permittivity=1.0*)

Bases: *Bound*, *Electric*

**__init__**(*grid*, *thickness=0.0*, *extinction_coefficient_function=<function AbsorbingBound.<lambda>>*, *background_permittivity=1.0*)

Constructs a boundary with depth-dependent extinction coefficient, kappa(rel_depth).

**Parameters**

- **grid** (Union[*Grid*, Sequence, ndarray]) – The Grid to which to the boundaries will be applied.
- **thickness** (Union[Real, Sequence, ndarray]) – The boundary thickness(es) in meters. This can be specified as a 2d-array [axis, side]. Singleton dimensions are broadcast.
- **extinction_coefficient_function** (Union[Callable, Sequence, ndarray]) – A function that returns the extinction coefficient as function of the depth in the boundary relative to the total thickness of the boundary.
- **background_permittivity** (Complex) – (default: 1.0 for vacuum)

**property is_electric:** bool

**property extinction:** ndarray

Determines the extinction coefficient, kappa, of the boundary on a plaid grid. The only non-zero values are found in the boundaries. At the corners, the maximum extinction value of the overlapping dimensions is returned.

Note that the returned array may have singleton dimensions that must be broadcast!

**Returns**

An nd-array with the extinction coefficient, kappa.

**property electric_susceptibility:** ndarray

The electric susceptibility, chi_E, at every sample point. Note that the returned array may have singleton dimensions that must be broadcast!

**property background_permittivity:** Complex

A complex scalar indicating the permittivity of the background.

**property between:** ndarray

Returns a boolean array indicating True for the voxels between the boundaries where the calculation happens. The inner edge is considered in between the boundaries.

**property beyond:** ndarray

Returns a boolean array indicating True for the voxels beyond the boundary edge, i.e. inside the boundaries, where the calculation should be ignored.

**property grid:** *Grid*

The Cartesian grid that indicates the sample positions of this bound and the volume it encompasses.

**property magnetic_susceptibility:** ndarray

The magnetic susceptibility, chi_H, at every sample point. Note that the returned array may have singleton dimensions that must be broadcast!

**property permittivity:** ndarray

The electric permittivity, epsilon, at every sample point. Note that the returned array may have singleton dimensions that must be broadcast!

**property thickness:** ndarray

The thickness as a 2D-array *thickness[axis, front_back]* in meters.

**class** macromax.bound.**LinearBound**(*grid*, *thickness=0.0*, *max_extinction_coefficient=0.25*, *background_permittivity=1.0*)

Bases: *AbsorbingBound*

**__init__**(*grid*, *thickness=0.0*, *max_extinction_coefficient=0.25*, *background_permittivity=1.0*)

Constructs a boundary with linearly increasing extinction coefficient, kappa.

**Parameters**

- **grid** (Union[*Grid*, Sequence, ndarray]) – The Grid to which to the boundaries will be applied.
- **thickness** (Union[Real, Sequence, ndarray]) – The boundary thickness(es) in meters. This can be specified as a 2d-array [axis, side]. Singleton dimensions are broadcast.
- **max_extinction_coefficient** (Union[Real, Sequence, ndarray]) – The maximum extinction coefficient, reached at the deepest point of the boundary at the edge of the calculation volume.
- **background_permittivity** (Complex) – (default: 1.0 for vacuum)

**property background_permittivity:** Complex

A complex scalar indicating the permittivity of the background.

**property between:** ndarray

Returns a boolean array indicating True for the voxels between the boundaries where the calculation happens. The inner edge is considered in between the boundaries.

**property beyond:** ndarray

Returns a boolean array indicating True for the voxels beyond the boundary edge, i.e. inside the boundaries, where the calculation should be ignored.

**property electric_susceptibility:** ndarray

The electric susceptibility, chi_E, at every sample point. Note that the returned array may have singleton dimensions that must be broadcast!

**property extinction:** ndarray

Determines the extinction coefficient, kappa, of the boundary on a plaid grid. The only non-zero values are found in the boundaries. At the corners, the maximum extinction value of the overlapping dimensions is returned.

Note that the returned array may have singleton dimensions that must be broadcast!

**Returns**

An nd-array with the extinction coefficient, kappa.

**property grid:** *Grid*

The Cartesian grid that indicates the sample positions of this bound and the volume it encompasses.

**property is_electric:** bool

**property magnetic_susceptibility:** ndarray

The magnetic susceptibility, chi_H, at every sample point. Note that the returned array may have singleton dimensions that must be broadcast!

**property permittivity:** ndarray

The electric permittivity, epsilon, at every sample point. Note that the returned array may have singleton dimensions that must be broadcast!

**property thickness:** ndarray

The thickness as a 2D-array *thickness[axis, front_back]* in meters.

## macromax.matrix module

**class** macromax.matrix.**CachingMatrix**(*caching=True*)

Bases: object

**__init__**(*caching=True*)

A mixin for Matrices that can cache the output.

**Parameters**

**caching** (bool) – Cache field propagation calculations. By default, the results are cached for multiplications with basis vectors. Numerical errors might accumulate for certain superpositions. Setting this property to Setting this to *False* disables the cache so that field propagations are always used and the constructor argument array is ignored.

**property caching:** bool

When set to True, this object uses cached values instead of propagating the field through the scatterer. Otherwise, field propagation is used for all matrix operations. This can help avoid the accumulation of numerical errors.

**class** macromax.matrix.**Matrix**(*array=None*, *shape=None*, *dtype=<class 'numpy.complex128'>*)

Bases: LinearOperator

A class to represent rectangular or square matrices that can be multiplied from the left or right, and pseudo-inverted.

**__init__**(*array=None*, *shape=None*, *dtype=<class 'numpy.complex128'>*)

Constructs a matrix from a rectangular numpy.ndarray, array-like object, or a function or method that returns one.

**Parameters**

- **array** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – A sequence or numpy.ndarray of complex numbers representing the matrix, or a function that returns one. If not specified, shape must be specified and _matmul and _rmatmul can be implemented in a subclass.
- **shape** (Optional[Sequence[int]]) – The optional shape of the matrix, i.e. the number of rows and columns.
- **dtype** – The optional dtype of the matrix elements.

**__len__**()

The number of rows in the matrix as an integer.

**Return type**

int

`__setitem__`(*key*, *value*)

Update (part of) the matrix.

**Parameters**

- **key** – Index or slice.
- **value** – The new value.

`__array__`()

The array values represented by this matrix.

**Return type**
ndarray

`inv`(*noise_level=0.0*)

The (Tikhonov regularized) inverse of the scattering matrix.

**Parameters**
**noise_level** (float) – (optional) argument to regularize the inversion of a (near-)singular matrix.

**Return type**
ndarray

**Returns**
An nd-array with the inverted matrix so that `self @ self.inv` approximates the identity.

`property H`

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

`property T`

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

`__add__`(*x*)

`__call__`(*x*)

Call self as a function.

`__matmul__`(*other*)

`__mul__`(*x*)

`__neg__`()

`static __new__`(*cls*, **args*, ***kwargs*)

`__pow__`(*p*)

`__rmatmul__`(*other*)

`__rmul__`(*x*)

`__sub__`(*x*)

`adjoint`()

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

`dot`(*x*)

Matrix-matrix or matrix-vector multiplication.

**Parameters**
**x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

**Returns**
**Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

**Return type**
array

`matmat`(*X*)

Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

**Parameters**
**X** (*{matrix, ndarray}*) – An array with shape (N,K).

**Returns**
**Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

**Return type**
{matrix, ndarray}

### Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

`matvec`(*x*)

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

**Returns**
**y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

**ndim = 2**

**rmatmat**(*X*)

Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

**Parameters**
**X** (*{matrix, ndarray}*) – A matrix or 2D array.

**Returns**
**Y** – A matrix or 2D array depending on the type of the input.

**Return type**
{matrix, ndarray}

### Notes

This rmatmat wraps the user-specified rmatmat routine.

**rmatvec**(*x*)

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

**Returns**
**y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

**transpose**()

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**class** macromax.matrix.**SquareMatrix**(*array=None*, *side=None*, *dtype=<class 'numpy.complex128'>*)

Bases: *Matrix*

A class to represent square matrices that can be inverted with or without regularization.

**__init__**(*array=None*, *side=None*, *dtype=<class 'numpy.complex128'>*)

Constructs a matrix from a square array, array-like object, or a function or method that returns one.

**Parameters**

- **array** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – A sequence or numpy.ndarray of complex numbers representing the matrix, or a function that returns one. If not specified, the identity matrix is assumed.
- **side** (Optional[int]) – The side of the matrix, i.e. the number of rows or columns.
- **dtype** – The optional dtype of the matrix elements.

**property side: int**

**property H**

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

**property T**

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**__add__**(*x*)

**__array__**()

The array values represented by this matrix.

**Return type**
ndarray

**__call__**(*x*)

Call self as a function.

__len__()

The number of rows in the matrix as an integer.

**Return type**
int

__matmul__(*other*)

__mul__(*x*)

__neg__()

static __new__(*cls*, **args*, ***kwargs*)

__pow__(*p*)

__rmatmul__(*other*)

__rmul__(*x*)

__setitem__(*key*, *value*)

Update (part of) the matrix.

**Parameters**

- **key** – Index or slice.
- **value** – The new value.

__sub__(*x*)

adjoint()

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

dot(*x*)

Matrix-matrix or matrix-vector multiplication.

**Parameters**
**x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

**Returns**
**Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

**Return type**
array

inv(*noise_level=0.0*)

The (Tikhonov regularized) inverse of the scattering matrix.

**Parameters**
**noise_level** (float) – (optional) argument to regularize the inversion of a (near-)singular matrix.

**Return type**
ndarray

**Returns**
An nd-array with the inverted matrix so that `self @ self.inv` approximates the identity.

**matmat**(*X*)

Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

**Parameters**
**X** (*{matrix, ndarray}*) – An array with shape (N,K).

**Returns**
**Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

**Return type**
{matrix, ndarray}

### Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

**matvec**(*x*)

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

**Returns**
**y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

**ndim = 2**

**rmatmat**(*X*)

Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

**Parameters**
**X** (*{matrix, ndarray}*) – A matrix or 2D array.

**Returns**
**Y** – A matrix or 2D array depending on the type of the input.

**Return type**
{matrix, ndarray}

### Notes

This rmatmat wraps the user-specified rmatmat routine.

`rmatvec`(*x*)

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

**Returns**
**y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

`transpose`()

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

`class` macromax.matrix.`LiteralScatteringMatrix`(*array=None*, *side=None*, *dtype=<class 'numpy.complex128'>*)

Bases: *SquareMatrix*

A class to represent scattering matrices constructed from an array of complex numbers.

`__init__`(*array=None*, *side=None*, *dtype=<class 'numpy.complex128'>*)

Constructs a scattering matrix from a square array, array-like object, or a function or method that returns one.

**Parameters**

- **array** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – A sequence or numpy.ndarray of complex numbers representing the matrix, or a function that returns one. If not specified, the identity matrix is assumed.
- **side** (Optional[int]) – The side of the matrix, i.e. the number of rows or columns.
- **dtype** – The optional dtype of the matrix elements.

**transfer**(*noise_level=0.0*)

Calculates the transfer matrix, relating one side of the scatterer to the other side (top, bottom). Each side can have incoming and outgoing waves. This is in contrast to the scattering matrix, `self.__array__`, which relates incoming waves from both sides to outgoing waves from both sides. One can be calculated from the other using the `matrix.convert()` function, though this calculation may be ill-conditioned (sensitive to noise). Therefore, the optional argument `noise_level` should be used to indicate the root-mean-square expectation value of the measurement error. This avoids divisions by near-zero values and obtains a best estimate using Tikhonov regularization.

**Parameters**
**noise_level** (float) – (optional) argument to regularize the inversion of a (near) singular backwards transmission matrix.

**Return type**
ndarray

**Returns**
An nd-array with the transfer matrix relating top-to-bottom instead of in-to-out. This can be converted back into a scattering matrix using the `matrix.convert()` function.

The first half of the vector inputs and outputs to the scattering and transfer matrices represent fields propagating forward along the positive propagation axis (0) and the second half represents fields propagating backward along the negative direction.

**Notation:**

**p**
positive propagation direction along propagation axis 0

**n**
negative propagation direction along propagation axis 0

**i**
inwards propagating (from source on either side)

**o**
outwards propagating (backscattered or transmitted)

Scattering matrix equation (in -> out):

[po] = [A, B] [pi]

[no] = [C, D] [ni]

Transfer matrix equation (top -> bottom):

[po] = [A - B inv(D) C, B inv(D)] [pi]

[ni] = [ - inv(D) C inv(D)] [no],

where inv(D) is the (regularized) inverse of D.

**property forward_transmission:** *ForwardTransmissionMatrix*

Select the forward-transmitted quarter of the scattering matrix. It indicates how the light coming from negative infinity is transmitted to positive infinity.

**Returns**
The forward-transmission matrix of shape `self.shape // 2`.

**property front_reflection:** *FrontReflectionMatrix*

Select the quarter of the scattering matrix corresponding to the light that is reflected of the front. It indicates how the light coming from negative infinity is back reflected to negative infinity.

**Returns**
The front-reflection matrix of shape `self.shape // 2`.

**property back_reflection:** *BackReflectionMatrix*

Select the quarter of the scattering matrix corresponding to the light that is reflected of the back. It indicates how the light coming from positive infinity is back reflected to positive infinity.

**Returns**
The back-reflection matrix of shape `self.shape // 2`.

**property backward_transmission:** *BackwardTransmissionMatrix*

Select the backward-transmitted quarter of the scattering matrix. It indicates how the light coming from positive infinity is transmitted to negative infinity.

**Returns**
The backward-transmission matrix of shape `self.shape // 2`.

**property H**

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

**property T**

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**__add__**(*x*)

**__array__**()

The array values represented by this matrix.

**Return type**
ndarray

**__call__**(*x*)

Call self as a function.

**__len__**()

The number of rows in the matrix as an integer.

**Return type**
int

**__matmul__**(*other*)

**__mul__**(*x*)

**__neg__**()

**static __new__**(*cls*, **args*, ***kwargs*)

**__pow__**(*p*)

**__rmatmul__**(*other*)

**__rmul__**(*x*)

**__setitem__**(*key*, *value*)

Update (part of) the matrix.

**Parameters**

- **key** – Index or slice.
- **value** – The new value.

**__sub__**(*x*)

**adjoint**()

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

**dot**(*x*)

Matrix-matrix or matrix-vector multiplication.

**Parameters**
**x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

**Returns**
**Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

**Return type**
array

**inv**(*noise_level=0.0*)

The (Tikhonov regularized) inverse of the scattering matrix.

**Parameters**
**noise_level** (float) – (optional) argument to regularize the inversion of a (near-)singular matrix.

**Return type**
ndarray

**Returns**
An nd-array with the inverted matrix so that `self @ self.inv` approximates the identity.

**matmat**(*X*)

Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

**Parameters**
**X** (*{matrix, ndarray}*) – An array with shape (N,K).

**Returns**
**Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

**Return type**
{matrix, ndarray}

### Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

`matvec(x)`

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

**Returns**
**y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

`ndim = 2`

`rmatmat(X)`

Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

**Parameters**
**X** (*{matrix, ndarray}*) – A matrix or 2D array.

**Returns**
**Y** – A matrix or 2D array depending on the type of the input.

**Return type**
{matrix, ndarray}

### Notes

This rmatmat wraps the user-specified rmatmat routine.

**rmatvec**(*x*)

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

**Returns**
**y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

**property side:** int

**transpose**()

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**class** macromax.matrix.**ScatteringMatrix**(*grid*, *vectorial=True*, *wavenumber=None*, *angular_frequency=None*, *vacuum_wavelength=None*, *epsilon=None*, *xi=0.0*, *zeta=0.0*, *mu=1.0*, *refractive_index=None*, *bound=None*, *dtype=None*, *callback=<function ScatteringMatrix.<lambda>>*, *caching=True*, *array=None*)

Bases: *LiteralScatteringMatrix*

A class representing scattering matrices.

**__init__**(*grid*, *vectorial=True*, *wavenumber=None*, *angular_frequency=None*, *vacuum_wavelength=None*, *epsilon=None*, *xi=0.0*, *zeta=0.0*, *mu=1.0*, *refractive_index=None*, *bound=None*, *dtype=None*, *callback=<function ScatteringMatrix.<lambda>>*, *caching=True*, *array=None*)

Construct a scattering matrix object for a medium specified by a refractive index distribution or the corresponding epsilon, xi, zeta, and mu distributions. Each electromagnetic field distribution entering the material is scattered into a certain electromagnetic field distributions propagating away from it from both sides. The complex matrix relates the amplitude and phase of all N propagating input modes to all N propagating output modes. No scattering, as in vacuum, is indicated by the NxN identity matrix. There are N/2 input and output modes on either side of the scattering material. Mode i and i+N/2 correspond to plane wave traveling in opposing directions. The mode directions are taken in raster-scan order and only propagating modes are included. When polarization is considered, the modes come in pairs corresponding to two orthogonal linear polarizations.

The modes are encoded as a vector of length N = 2xMxP for 2 sides, M angles, and P polarizations.

- First the N/2 modes propagating along the positive x-axis are considered, then those propagating in the reverse direction.

- In each direction, M different angles (k-vectors) can be considered. We choose propagating modes on a uniformly-spaced plaid grid that includes the origin (corresponding to the k-vector along the x-axis). Modes not propagating along the x-axis, i.e. in the y-z-plane are not considered. The angles are ordered in raster-scan order from negative k_y to positive k_y (slow) and from negative k_z to positive k_z (fast). The grid axes dimensions correspond to x(0), y(1), z(2).
- When polarization is considered, each angle has a pair of modes, one for each polarization. The first mode has the polarization oriented along the rotated y'-axis and the second mode along the rotated z'-axis. To avoid ambiguity for normal incidence, the Cartesian-coordinate system is rotated along the shortest possible path, i.e. along the axis that is normal to the original x-axis and the mode's k-vector. All rotations are around the origin of the coordinate system, incurring no phase shift there.

Vectors can be converted to field distributions on the complete grid using the methods:

*srcvec2freespace()* super-position of free-space plane waves in the whole volume (fast)

*srcvec2source()* super-position of free-space plane waves at the source planes at the front and back (fast)

*source2detfield()* calculate the field in the whole volume using the `solver.Solution` object (slow)

***detfield2detvec()* vector corresponding to the detected field at the detection planes (fast). The fields**
at those planes should only contain the outward propagating waves. Hence, inwards propagating waves should be subtracted before using this method!

***srcvec2detfield()* calculate the field in the whole volume and convert it to a detection vector (slow)**
The latter is used in the matrix multiplication method: `matmul`, @

**Parameters**

- **grid** (Union[*Grid*, Sequence, ndarray]) – A Grid object or a Sequence of vectors with uniformly increasing values that indicate the positions in a plaid grid of sample points for the material and solution. In the one-dimensional case, a simple increasing Sequence of uniformly-spaced numbers may be provided as an alternative. The length of the ranges determines the data_shape, to which the source_distribution, epsilon, xi, zeta, mu, and initial_field must broadcast when specified as ndarrays.
- **vectorial** (Optional[bool]) – a boolean indicating if the source and solution are 3-vectors-fields (True) or scalar fields (False).
- **wavenumber** (Optional[Real]) – the wavenumber in vacuum = 2 pi / vacuum_wavelength. The wavelength in the same units as used for the other inputs/outputs.
- **angular_frequency** (Optional[Real]) – alternative argument to the wavenumber = angular_frequency / c
- **vacuum_wavelength** (Optional[Real]) – alternative argument to the wavenumber = 2 pi / vacuum_wavelength
- **epsilon** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – an array or function that returns the (tensor) epsilon that represents the permittivity at the points indicated by the grid specified as its input arguments.
- **xi** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – an array or function that returns the (tensor) xi for bi-(an)isotropy at the points indicated by the grid specified as its input arguments.
- **zeta** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – an array or function that returns the (tensor) zeta for bi-(an)isotropy at the points indicated by the grid specified as its input arguments.

- **mu** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – an array or function that returns the (tensor) permeability at the points indicated by the grid specified as its input arguments.
- **refractive_index** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – an array or function that returns the (tensor) refractive_index = np.sqrt(permittivity) at the points indicated by the *grid* input argument.
- **bound** (Optional[*Bound*]) – An object representing the boundary of the calculation volume. Default: None, PeriodicBound(grid)
- **dtype** – optional numpy datatype for the internal operations and results. This must be a complex number type as numpy.complex128 or np.complex64.
- **callback** (Callable) – optional function that will be called with as argument this solver. This function can be used to check and display progress. It must return a boolean value of True to indicate that further iterations are required.
- **caching** (bool) – Cache field propagation calculations. By default, the results are cached for multiplications with basis vectors. Numerical errors might accumulate for certain superpositions. Setting this property to False will ensure that field propagations are always used and the constructor argument array is ignored.
- **array** (Union[Complex, Sequence, ndarray, LinearOperator, None]) – Optional in case the matrix values have been calculated before and stored. If not specified, the matrix is calculated from the material properties. I specified, this must be a sequence or numpy.ndarray of complex numbers representing the matrix, or a function that returns one.

**Returns**

The Solution object that has the E and H fields, as well as iteration information.

**property grid:** *Grid*

The calculation grid.

**property vectorial:** bool

Boolean to indicates whether calculations happen on polarized (True) or scalar (False) fields.

**property caching:** bool

When set to True, this object uses cached values instead of propagating the field through the scatterer. Otherwise, field propagation is used for all matrix operations. This can help avoid the accumulation of numerical errors.

**srcvec2freespace**(*input_vector*)

Convert an input source vector to a superposition of plane waves at the origin and spreading over the whole volume. The input vector specifies the propagating modes in the far-field (the inverse Fourier transform of the fields at the sample origin). Incident waves at an angle will result in higher amplitudes to compensate for the reduction in propagation along the propagation axis through the entrance plane.

Used in `ScatteringMatrix.srcvec2source` to calculate the source field distribution before entering the scatterer.

Used in `ScatteringMatrix.__matmul__` to distinguish incoming from back-scattered light.

**Parameters**

**input_vector** (Union[Complex, Sequence, ndarray, LinearOperator]) – A source vector or array of shape [2, M, P], where the first axis indicates the side (front, back), the second axis indicates the propagation mode (direction, top-bottom-left-right), and the final axis indicates the polarization (1 for scalar, 2 for polarized: V-H).

**Return type**
ndarray

**Returns**
An nd-array with the field on the calculation grid. Its shape is (1, *self.grid.shape) for scalar calculations and (3, *self.grid.shape) for vectorial calculations with polarization.

**srcvec2source**(*input_vector*, *out=None*)

Converts a source vector into an (N+1)D-array with the source field at the front and back of the scatterer. The source field is such that it produces E-fields of unit intensity for unit vector inputs.

Used in `self.vector2field()` and `self.__matmul__()`.

**Parameters**

- **input_vector** (Union[Complex, Sequence, ndarray, LinearOperator]) – A source vector with `self.shape[1]` elements. One value per side, per independent polarization (2), and per mode (inwards propagating k-vectors only).
- **out** (Optional[ndarray]) – (optional) numpy array to store the result.

**Return type**
ndarray

**Returns**
The field distribution as an array of shape [nb_pol, *self.grid], where nb_pol = 3 for a vectorial calculation and 1 for a scalar calculation.

**source2detfield**(*source*, *out=None*)

Calculates the (N+1)D-input-field distribution throughout the scatterer for a given source field distribution.

**Parameters**

- **source** (Union[Complex, Sequence, ndarray, LinearOperator]) – The source field distribution in the whole space.
- **out** (Optional[ndarray]) – (optional) numpy array to store the result (shape: self.grid.shape, dtype: self.dtype).

**Return type**
ndarray

**Returns**
The field distribution as an array of shape [nb_pol, *self.grid], where nb_pol = 3 for a vectorial calculation and 1 for a scalar calculation.

**srcvec2detfield**(*input_vector*, *out=None*)

Calculates the (N+1)D-input-field distribution throughout the scatterer for a given input source vector.

Used in `self.__matmul__()` and external code.

**Parameters**

- **input_vector** (Union[Complex, Sequence, ndarray, LinearOperator]) – A source vector with `self.shape[1]` elements.
- **out** (Optional[ndarray]) – (optional) numpy array to store the result.

**Return type**
ndarray

**Returns**
The field distribution as an array of shape [nb_pol, *self.grid], where nb_pol = 3 for a vectorial calculation and 1 for a scalar calculation.

`detfield2detvec`(*field*)

Converts the (N+1)D-output-field defined at the front and back detection planes of the scatterer to a detection vector that describes the resulting far-field distribution (the inverse Fourier transform of the fields at the sample origin). The fields at those planes should only contain the outward propagating waves. Hence, inwards propagating waves should be subtracted before using this method!

This is used in `self.__matmul__()`.

**Parameters**
**field** (Union[Complex, Sequence, ndarray, LinearOperator]) – The detected field in all space (of which only the detection space is used).

**Return type**
ndarray

**Returns**
Detection vector.

`detvec2srcvec`(*vec*)

Convert forward propagating detection vector into a backward propagating (time-reversed) source vector.

**Parameters**
**vec** (Union[Complex, Sequence, ndarray, LinearOperator]) – detection vector obtained from solution or scattering matrix multiplication.

**Return type**
ndarray

**Returns**
Time-reversed vector, that can be used as a source vector.

`__setitem__`(*key*, *value*)

Updating this matrix is not possible. Use a `Matrix` object instead.

**Parameters**

- **key** – Index or slice.
- **value** – The new value.

`__array__`(*out=None*)

Lazily calculates the scattering matrix as a regular numpy.ndarray

`property H`

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

property T

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

__add__(*x*)

__call__(*x*)

Call self as a function.

__len__()

The number of rows in the matrix as an integer.

**Return type**
int

__matmul__(*other*)

__mul__(*x*)

__neg__()

static __new__(*cls*, **args*, ***kwargs*)

__pow__(*p*)

__rmatmul__(*other*)

__rmul__(*x*)

__sub__(*x*)

adjoint()

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

property back_reflection: *BackReflectionMatrix*

Select the quarter of the scattering matrix corresponding to the light that is reflected of the back. It indicates how the light coming from positive infinity is back reflected to positive infinity.

**Returns**
The back-reflection matrix of shape `self.shape // 2`.

property backward_transmission: *BackwardTransmissionMatrix*

Select the backward-transmitted quarter of the scattering matrix. It indicates how the light coming from positive infinity is transmitted to negative infinity.

**Returns**
The backward-transmission matrix of shape `self.shape // 2`.

**dot**(*x*)

Matrix-matrix or matrix-vector multiplication.

**Parameters**
**x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

**Returns**
**Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

**Return type**
array

**property forward_transmission:** *ForwardTransmissionMatrix*

Select the forward-transmitted quarter of the scattering matrix. It indicates how the light coming from negative infinity is transmitted to positive infinity.

**Returns**
The forward-transmission matrix of shape `self.shape // 2`.

**property front_reflection:** *FrontReflectionMatrix*

Select the quarter of the scattering matrix corresponding to the light that is reflected of the front. It indicates how the light coming from negative infinity is back reflected to negative infinity.

**Returns**
The front-reflection matrix of shape `self.shape // 2`.

**inv**(*noise_level=0.0*)

The (Tikhonov regularized) inverse of the scattering matrix.

**Parameters**
**noise_level** (`float`) – (optional) argument to regularize the inversion of a (near-)singular matrix.

**Return type**
`ndarray`

**Returns**
An nd-array with the inverted matrix so that `self @ self.inv` approximates the identity.

**matmat**(*X*)

Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

**Parameters**
**X** (*{matrix, ndarray}*) – An array with shape (N,K).

**Returns**
**Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

**Return type**
{matrix, ndarray}

### Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

**matvec**(*x*)

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

**Returns**
**y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

**ndim = 2**

**rmatmat**(*X*)

Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

**Parameters**
**X** (*{matrix, ndarray}*) – A matrix or 2D array.

**Returns**
**Y** – A matrix or 2D array depending on the type of the input.

**Return type**
{matrix, ndarray}

### Notes

This rmatmat wraps the user-specified rmatmat routine.

**rmatvec**(*x*)

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

**Returns**
**y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

**property side:** `int`

**transfer**(*noise_level=0.0*)

Calculates the transfer matrix, relating one side of the scatterer to the other side (top, bottom). Each side can have incoming and outgoing waves. This is in contrast to the scattering matrix, `self.__array__`, which relates incoming waves from both sides to outgoing waves from both sides. One can be calculated from the other using the `matrix.convert()` function, though this calculation may be ill-conditioned (sensitive to noise). Therefore, the optional argument `noise_level` should be used to indicate the root-mean-square expectation value of the measurement error. This avoids divisions by near-zero values and obtains a best estimate using Tikhonov regularization.

**Parameters**
**noise_level** (`float`) – (optional) argument to regularize the inversion of a (near) singular backwards transmission matrix.

**Return type**
`ndarray`

**Returns**
An nd-array with the transfer matrix relating top-to-bottom instead of in-to-out. This can be converted back into a scattering matrix using the `matrix.convert()` function.

The first half of the vector inputs and outputs to the scattering and transfer matrices represent fields propagating forward along the positive propagation axis (0) and the second half represents fields propagating backward along the negative direction.

**Notation:**

**p**
positive propagation direction along propagation axis 0

**n**
negative propagation direction along propagation axis 0

**i**
inwards propagating (from source on either side)

**o**
outwards propagating (backscattered or transmitted)

Scattering matrix equation (in -> out):

[po] = [A, B] [pi]

[no] = [C, D] [ni]

Transfer matrix equation (top -> bottom):

[po] = [A - B inv(D) C, B inv(D)] [pi]

[ni] = [ - inv(D) C inv(D)] [no],

where inv(D) is the (regularized) inverse of D.

**transpose**()

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**class** macromax.matrix.**QuarterMatrix**(*matrix*, *backwards_output*, *backwards_input*)

Bases: *SquareMatrix*

A base class representing a quarter of a scattering matrix.

**__init__**(*matrix*, *backwards_output*, *backwards_input*)

Construct an object refering to a quarter of a scattering matrix. Any Scattering matrix should have an even number of rows and columns.

**Parameters**

- **matrix** (*SquareMatrix*) – The underlying scattering matrix.
- **backwards_output** (bool) – When True, select the bottom half of the matrix, i.e. the quadrants corresponding to the output modes exiting the front side of the scatterer (back-to-front propagation).
- **backwards_input** (bool) – When True, select the right half of the matrix, i.e. the quadrants corresponding to the input modes entering from the back side of the scatterer (back-to-front propagation).

**property full_matrix:** *SquareMatrix*

The underlying (scattering) matrix of size *2 * self.shape*.

**property H**

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

**property T**

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**__add__**(*x*)

**__array__**()

The array values represented by this matrix.

**Return type**
ndarray

**__call__**(*x*)

Call self as a function.

__len__()

The number of rows in the matrix as an integer.

**Return type**
int

__matmul__(*other*)

__mul__(*x*)

__neg__()

static __new__(*cls*, **args*, ***kwargs*)

__pow__(*p*)

__rmatmul__(*other*)

__rmul__(*x*)

__setitem__(*key*, *value*)

Update (part of) the matrix.

**Parameters**

- **key** – Index or slice.
- **value** – The new value.

__sub__(*x*)

adjoint()

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

dot(*x*)

Matrix-matrix or matrix-vector multiplication.

**Parameters**
**x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

**Returns**
**Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

**Return type**
array

inv(*noise_level=0.0*)

The (Tikhonov regularized) inverse of the scattering matrix.

**Parameters**
**noise_level** (float) – (optional) argument to regularize the inversion of a (near-)singular matrix.

**Return type**
ndarray

**Returns**
An nd-array with the inverted matrix so that `self @ self.inv` approximates the identity.

**matmat**(*X*)

Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

**Parameters**
**X** (*{matrix, ndarray}*) – An array with shape (N,K).

**Returns**
**Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

**Return type**
{matrix, ndarray}

### Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

**matvec**(*x*)

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

**Returns**
**y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

**ndim = 2**

**rmatmat**(*X*)

Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

**Parameters**
**X** (*{matrix, ndarray}*) – A matrix or 2D array.

**Returns**
**Y** – A matrix or 2D array depending on the type of the input.

**Return type**
{matrix, ndarray}

### Notes

This rmatmat wraps the user-specified rmatmat routine.

**rmatvec**(*x*)

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

**Returns**
**y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

**property side:** int

**transpose**()

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**class** macromax.matrix.**ForwardTransmissionMatrix**(*scattering_matrix*)

Bases: *QuarterMatrix*

The forward transmission matrix of the specified scattering matrix. It indicates how the light coming from negative infinity is transmitted to positive infinity.

**__init__**(*scattering_matrix*)

Construct an object refering to a quarter of a scattering matrix. Any Scattering matrix should have an even number of rows and columns.

**Parameters**

- **matrix** – The underlying scattering matrix.
- **backwards_output** – When True, select the bottom half of the matrix, i.e. the quadrants corresponding to the output modes exiting the front side of the scatterer (back-to-front propagation).
- **backwards_input** – When True, select the right half of the matrix, i.e. the quadrants corresponding to the input modes entering from the back side of the scatterer (back-to-front propagation).

property H

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

property T

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

__add__(*x*)

__array__()

The array values represented by this matrix.

**Return type**
ndarray

__call__(*x*)

Call self as a function.

__len__()

The number of rows in the matrix as an integer.

**Return type**
int

__matmul__(*other*)

__mul__(*x*)

__neg__()

static __new__(*cls*, **args*, ***kwargs*)

__pow__(*p*)

__rmatmul__(*other*)

__rmul__(*x*)

__setitem__(*key*, *value*)

Update (part of) the matrix.

**Parameters**

- **key** – Index or slice.
- **value** – The new value.

__sub__(*x*)

`adjoint()`

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

`dot(x)`

Matrix-matrix or matrix-vector multiplication.

**Parameters**
**x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

**Returns**
**Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

**Return type**
array

`property full_matrix: SquareMatrix`

The underlying (scattering) matrix of size *2 * self.shape*.

`inv(noise_level=0.0)`

The (Tikhonov regularized) inverse of the scattering matrix.

**Parameters**
**noise_level** (`float`) – (optional) argument to regularize the inversion of a (near-)singular matrix.

**Return type**
`ndarray`

**Returns**
An nd-array with the inverted matrix so that `self @ self.inv` approximates the identity.

`matmat(X)`

Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

**Parameters**
**X** (*{matrix, ndarray}*) – An array with shape (N,K).

**Returns**
**Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

**Return type**
{matrix, ndarray}

### Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

**matvec**(*x*)

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

**Returns**
**y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

**ndim = 2**

**rmatmat**(*X*)

Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

**Parameters**
**X** (*{matrix, ndarray}*) – A matrix or 2D array.

**Returns**
**Y** – A matrix or 2D array depending on the type of the input.

**Return type**
{matrix, ndarray}

### Notes

This rmatmat wraps the user-specified rmatmat routine.

**rmatvec**(*x*)

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

**Returns**
**y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

**property side:** int

**transpose**()

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**class** macromax.matrix.**FrontReflectionMatrix**(*scattering_matrix*)

Bases: *QuarterMatrix*

The forward reflection matrix of the specified scattering matrix. It indicates how the light coming from positive infinity is back reflected to positive infinity.

**__init__**(*scattering_matrix*)

Construct an object refering to a quarter of a scattering matrix. Any Scattering matrix should have an even number of rows and columns.

**Parameters**

- **matrix** – The underlying scattering matrix.
- **backwards_output** – When True, select the bottom half of the matrix, i.e. the quadrants corresponding to the output modes exiting the front side of the scatterer (back-to-front propagation).
- **backwards_input** – When True, select the right half of the matrix, i.e. the quadrants corresponding to the input modes entering from the back side of the scatterer (back-to-front propagation).

**property H**

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

**property T**

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**__add__**(*x*)

__array__()

The array values represented by this matrix.

**Return type**
ndarray

__call__(*x*)

Call self as a function.

__len__()

The number of rows in the matrix as an integer.

**Return type**
int

__matmul__(*other*)

__mul__(*x*)

__neg__()

static __new__(*cls*, **args*, ***kwargs*)

__pow__(*p*)

__rmatmul__(*other*)

__rmul__(*x*)

__setitem__(*key*, *value*)

Update (part of) the matrix.

**Parameters**

- **key** – Index or slice.
- **value** – The new value.

__sub__(*x*)

adjoint()

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

dot(*x*)

Matrix-matrix or matrix-vector multiplication.

**Parameters**
**x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

**Returns**
**Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

**Return type**
array

**property full_matrix:** *SquareMatrix*

The underlying (scattering) matrix of size *2 * self.shape*.

**inv**(*noise_level=0.0*)

The (Tikhonov regularized) inverse of the scattering matrix.

**Parameters**
**noise_level** (float) – (optional) argument to regularize the inversion of a (near-)singular matrix.

**Return type**
ndarray

**Returns**
An nd-array with the inverted matrix so that `self @ self.inv` approximates the identity.

**matmat**(*X*)

Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

**Parameters**
**X** (*{matrix, ndarray}*) – An array with shape (N,K).

**Returns**
**Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

**Return type**
{matrix, ndarray}

### Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

**matvec**(*x*)

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

**Returns**
**y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

**ndim = 2**

**rmatmat**(*X*)

Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

**Parameters**
**X** (*{matrix, ndarray}*) – A matrix or 2D array.

**Returns**
**Y** – A matrix or 2D array depending on the type of the input.

**Return type**
{matrix, ndarray}

### Notes

This rmatmat wraps the user-specified rmatmat routine.

**rmatvec**(*x*)

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

**Returns**
**y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

**property side:** int

**transpose**()

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**class** macromax.matrix.**BackReflectionMatrix**(*scattering_matrix*)

Bases: *QuarterMatrix*

The backward reflection matrix of the specified scattering matrix. It indicates how the light coming from negative infinity is back reflected to negative infinity.

**__init__**(*scattering_matrix*)

Construct an object refering to a quarter of a scattering matrix. Any Scattering matrix should have an even number of rows and columns.

**Parameters**

- **matrix** – The underlying scattering matrix.
- **backwards_output** – When True, select the bottom half of the matrix, i.e. the quadrants corresponding to the output modes exiting the front side of the scatterer (back-to-front propagation).
- **backwards_input** – When True, select the right half of the matrix, i.e. the quadrants corresponding to the input modes entering from the back side of the scatterer (back-to-front propagation).

**property H**

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

**property T**

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**__add__**(*x*)

**__array__**()

The array values represented by this matrix.

**Return type**
ndarray

**__call__**(*x*)

Call self as a function.

**__len__**()

The number of rows in the matrix as an integer.

**Return type**
int

**__matmul__**(*other*)

__mul__(*x*)

__neg__()

static __new__(*cls*, **args*, ***kwargs*)

__pow__(*p*)

__rmatmul__(*other*)

__rmul__(*x*)

__setitem__(*key*, *value*)

Update (part of) the matrix.

**Parameters**

- **key** – Index or slice.
- **value** – The new value.

__sub__(*x*)

adjoint()

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

dot(*x*)

Matrix-matrix or matrix-vector multiplication.

**Parameters**
**x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

**Returns**
**Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

**Return type**
array

property full_matrix: *SquareMatrix*

The underlying (scattering) matrix of size *2 * self.shape*.

inv(*noise_level=0.0*)

The (Tikhonov regularized) inverse of the scattering matrix.

**Parameters**
**noise_level** (float) – (optional) argument to regularize the inversion of a (near-)singular matrix.

**Return type**
ndarray

**Returns**
An nd-array with the inverted matrix so that `self @ self.inv` approximates the identity.

`matmat`(*X*)

Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

**Parameters**
**X** (*{matrix, ndarray}*) – An array with shape (N,K).

**Returns**
**Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

**Return type**
{matrix, ndarray}

### Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

`matvec`(*x*)

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

**Returns**
**y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

`ndim = 2`

`rmatmat`(*X*)

Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

**Parameters**
**X** (*{matrix, ndarray}*) – A matrix or 2D array.

**Returns**
**Y** – A matrix or 2D array depending on the type of the input.

**Return type**
{matrix, ndarray}

### Notes

This rmatmat wraps the user-specified rmatmat routine.

**rmatvec**(*x*)

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

**Returns**
**y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

**property side:** int

**transpose**()

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**class** macromax.matrix.**BackwardTransmissionMatrix**(*scattering_matrix*)

Bases: *QuarterMatrix*

The backward transmission matrix of the specified scattering matrix. It indicates how the light coming from positive infinity is transmitted to negative infinity.

**__init__**(*scattering_matrix*)

Construct an object refering to a quarter of a scattering matrix. Any Scattering matrix should have an even number of rows and columns.

**Parameters**

- **matrix** – The underlying scattering matrix.
- **backwards_output** – When True, select the bottom half of the matrix, i.e. the quadrants corresponding to the output modes exiting the front side of the scatterer (back-to-front propagation).
- **backwards_input** – When True, select the right half of the matrix, i.e. the quadrants corresponding to the input modes entering from the back side of the scatterer (back-to-front propagation).

**property H**

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

**property T**

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**__add__**(*x*)

**__array__**()

The array values represented by this matrix.

**Return type**
ndarray

**__call__**(*x*)

Call self as a function.

**__len__**()

The number of rows in the matrix as an integer.

**Return type**
int

**__matmul__**(*other*)

**__mul__**(*x*)

**__neg__**()

**static __new__**(*cls*, **args*, ***kwargs*)

**__pow__**(*p*)

**__rmatmul__**(*other*)

**__rmul__**(*x*)

**__setitem__**(*key*, *value*)

Update (part of) the matrix.

**Parameters**

- **key** – Index or slice.
- **value** – The new value.

**__sub__**(*x*)

**adjoint**()

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

`dot`(*x*)

Matrix-matrix or matrix-vector multiplication.

**Parameters**
**x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

**Returns**
**Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

**Return type**
array

`property full_matrix:` *SquareMatrix*

The underlying (scattering) matrix of size *2 * self.shape*.

`inv`(*noise_level=0.0*)

The (Tikhonov regularized) inverse of the scattering matrix.

**Parameters**
**noise_level** (float) – (optional) argument to regularize the inversion of a (near-)singular matrix.

**Return type**
ndarray

**Returns**
An nd-array with the inverted matrix so that `self @ self.inv` approximates the identity.

`matmat`(*X*)

Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

**Parameters**
**X** (*{matrix, ndarray}*) – An array with shape (N,K).

**Returns**
**Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

**Return type**
{matrix, ndarray}

### Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

`matvec`(*x*)

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

**Returns**
**y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

**ndim = 2**

**rmatmat**(*X*)

Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

**Parameters**
**X** (*{matrix, ndarray}*) – A matrix or 2D array.

**Returns**
**Y** – A matrix or 2D array depending on the type of the input.

**Return type**
{matrix, ndarray}

### Notes

This rmatmat wraps the user-specified rmatmat routine.

**rmatvec**(*x*)

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

**Returns**
**y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

**property side: int**

**transpose()**

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

**class** macromax.matrix.**DepositionMatrix**(*scattering_matrix*, *input_operator=None*, *output_operator=None*, *caching=True*)

Bases: *Matrix*, *CachingMatrix*

A rectangular matrix that relates free-space input vectors to an arbitrary subset of fields in the interior.

**__init__**(*scattering_matrix*, *input_operator=None*, *output_operator=None*, *caching=True*)

Creates a matrix based on the internally scattered fields of a ScatteringMatrix.

**Parameters**

- **scattering_matrix** (*ScatteringMatrix*) – The base ScatteringMatrix
- **input_operator** (Optional[LinearOperator]) – The optional field projector, a LinearOperator object that multiplies any input vector to produce a linear combination of source fields. Default: the srcvec2source method of the ScatteringMatrix.
- **output_operator** (Union[LinearOperator, Callable[[Union[Complex, Sequence, ndarray, LinearOperator]], ndarray], None]) – The optional detection field projector, a LinearOperator object that multiplies the raveled full-field distribution and returns a projection vector or a function that does the same. Default: the detfield2detvec method of the ScatteringMatrix.
- **caching** (bool) – Cache field propagation calculations. By default, the results are cached for multiplications with basis vectors. Numerical errors might accumulate for certain superpositions. Setting this property to False will ensure that field propagations are always used and the constructor argument array is ignored.

**property H**

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

**property T**

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

__add__(*x*)

__array__()

The array values represented by this matrix.

**Return type**
ndarray

__call__(*x*)

Call self as a function.

__len__()

The number of rows in the matrix as an integer.

**Return type**
int

__matmul__(*other*)

__mul__(*x*)

__neg__()

static __new__(*cls*, **args*, ***kwargs*)

__pow__(*p*)

__rmatmul__(*other*)

__rmul__(*x*)

__setitem__(*key*, *value*)

Update (part of) the matrix.

**Parameters**

- **key** – Index or slice.
- **value** – The new value.

__sub__(*x*)

adjoint()

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

**Returns**
**A_H** – Hermitian adjoint of self.

**Return type**
LinearOperator

property caching: bool

When set to True, this object uses cached values instead of propagating the field through the scatterer. Otherwise, field propagation is used for all matrix operations. This can help avoid the accumulation of numerical errors.

`dot`(*x*)

Matrix-matrix or matrix-vector multiplication.

**Parameters**
**x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

**Returns**
**Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

**Return type**
array

`inv`(*noise_level=0.0*)

The (Tikhonov regularized) inverse of the scattering matrix.

**Parameters**
**`noise_level`** (`float`) – (optional) argument to regularize the inversion of a (near-)singular matrix.

**Return type**
`ndarray`

**Returns**
An nd-array with the inverted matrix so that `self @ self.inv` approximates the identity.

`matmat`(*X*)

Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

**Parameters**
**X** (*{matrix, ndarray}*) – An array with shape (N,K).

**Returns**
**Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

**Return type**
{matrix, ndarray}

### Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

`matvec`(*x*)

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

**Returns**
**y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

**ndim = 2**

**rmatmat**(*X*)

Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

**Parameters**
**X** (*{matrix, ndarray}*) – A matrix or 2D array.

**Returns**
**Y** – A matrix or 2D array depending on the type of the input.

**Return type**
{matrix, ndarray}

### Notes

This rmatmat wraps the user-specified rmatmat routine.

**rmatvec**(*x*)

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

**Parameters**
**x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

**Returns**
**y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

**Return type**
{matrix, ndarray}

### Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

**transpose**()

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

macromax.matrix.**inv**(*mat*, *noise_level=0.0*)

ScatteringMatrix inversion with optional Tikhonov regularization.

**Return type**
ndarray

macromax.matrix.**convert**(*s*, *noise_level=0.0*)

Converts a scattering matrix into a transfer matrix and vice-versa.

**Notation:**
p: positive propagation direction along propagation axis 0, n: negative propagation direction along propagation axis 0, i: inwards propagating (from source on either side), o: outwards propagating (backscattered or transmitted).

**Scattering matrix equation (in -> out):**
[po] = [A, B] [pi], [no] [C, D] [ni].

**Transfer matrix equation (top -> bottom):**
[po] = [A - B inv(D) C, B inv(D)] [pi], [ni] = [ - inv(D) C inv(D)] [no],

where inv(D) is the (regularized) inverse of D.

The first half of the vector inputs and outputs to the scattering and transfer matrices represent fields propagating forward along the positive propagation axis (0) and the second half represents fields propagating backward along the negative direction.

**Parameters**

- **s** (Union[Complex, Sequence, ndarray, LinearOperator]) – The scattering (transfer) matrix as a 2D np.ndarray of shape [N, N]. Alternatively, each dimension can be split in two halves for downward and upward propagating fields. In that case the shape would be [2, N//2, 2, N//2].
- **noise_level** (float) – The noise level of the measured matrix elements (singular values). If greater than 0, (Tikhonov) regularization will be used.

**Return type**
ndarray

**Returns**
The transfer (scattering) matrix of the same shape as the scattering matrix.

## macromax.solver module

This module calculates the solution to the wave equations. More specifically, the work is done in the iteration defined in the *Solution.__iter__()* method of the *Solution* class. The convenience function *solve()* is provided to construct a *Solution* object and iterate it to convergence using its *Solution.solve()* method.

macromax.solver.**solve**(*grid*, *vectorial=None*, *wavenumber=1.0*, *angular_frequency=None*, *vacuum_wavelength=None*, *current_density=None*, *source_distribution=None*, *epsilon=None*, *xi=0.0*, *zeta=0.0*, *mu=1.0*, *refractive_index=None*, *bound=None*, *initial_field=0.0*, *dtype=None*, *callback=<function <lambda>>*)

Function to find a solution for Maxwell's equations in a media specified by the epsilon, xi, zeta, and mu distributions in the presence of a current source.

**Parameters**

- **grid** (Union[*Grid*, Sequence, ndarray]) – A Grid object or a Sequence of vectors with uniformly increasing values that indicate the positions in a plaid grid of sample points for the material and solution. In the one-dimensional case, a simple increasing Sequence of uniformly-spaced numbers may be provided as an alternative. The length of the ranges determines the data_shape, to which the source_distribution, epsilon, xi, zeta, mu, and initial_field must broadcast when specified as ndarrays.
- **vectorial** (Optional[bool]) – a boolean indicating if the source and solution are 3-vectors-fields (True) or scalar fields (False).

- **wavenumber** (Optional[Real]) – the wavenumber in vacuum = 2 pi / vacuum_wavelength. The wavelength in the same units as used for the other inputs/outputs.
- **angular_frequency** (Optional[Real]) – alternative argument to the wavenumber = angular_frequency / c
- **vacuum_wavelength** (Optional[Real]) – alternative argument to the wavenumber = 2 pi / vacuum_wavelength
- **current_density** (Union[Complex, Sequence, ndarray, None]) – (optional, instead of source_distribution) An array or function that returns the free (vectorial) current density input distribution, J. The free current density has units of $Am^-2$.
- **source_distribution** (Union[Complex, Sequence, ndarray, None]) – (optional, instead of current_density) An array or function that returns the (vectorial) source input wave distribution. The source values relate to the current density, J, as 1j * angular_frequency * scipy.constants.mu_0 * J and has units of $rads^-1Hm^-1Am^-2 = radVm^-3$. More general, non-electro-magnetic wave problems can be solved using the source_distribution, as it does not rely on the vacuum permeability constant, $mu_0$.
- **epsilon** (Union[Complex, Sequence, ndarray, None]) – an array or function that returns the (tensor) epsilon that represents the permittivity at the points indicated by the grid specified as its input arguments.
- **xi** (Union[Complex, Sequence, ndarray]) – an array or function that returns the (tensor) xi for bi-(an)isotropy at the points indicated by the grid specified as its input arguments.
- **zeta** (Union[Complex, Sequence, ndarray]) – an array or function that returns the (tensor) zeta for bi-(an)isotropy at the points indicated by the grid specified as its input arguments.
- **mu** (Union[Complex, Sequence, ndarray]) – an array or function that returns the (tensor) permeability at the points indicated by the grid specified as its input arguments.
- **refractive_index** (Union[Complex, Sequence, ndarray, None]) – an array or function that returns the (complex) (tensor) refractive_index, as the square root of the permittivity, at the points indicated by the *grid* input argument.
- **bound** (Optional[*Bound*]) – An object representing the boundary of the calculation volume. Default: None, PeriodicBound(grid)
- **initial_field** (Union[Complex, Sequence, ndarray]) – optional start value for the E-field distribution (default: all zero E)
- **dtype** – optional numpy datatype for the internal operations and results. This must be a complex number type as numpy.complex128 or np.complex64.
- **callback** (Callable) – optional function that will be called with as argument this solver. This function can be used to check and display progress. It must return a boolean value of True to indicate that further iterations are required.

**Returns**

The Solution object that has the E and H fields, as well as iteration information.

class macromax.solver.**Solution**(*grid*, *vectorial=None*, *wavenumber=1.0*, *angular_frequency=None*, *vacuum_wavelength=None*, *current_density=None*, *source_distribution=None*, *epsilon=None*, *xi=0.0*, *zeta=0.0*, *mu=1.0*, *refractive_index=None*, *bound=None*, *initial_field=0.0*, *dtype=None*)

Bases: object

**__init__**(*grid*, *vectorial=None*, *wavenumber=1.0*, *angular_frequency=None*, *vacuum_wavelength=None*, *current_density=None*, *source_distribution=None*, *epsilon=None*, *xi=0.0*, *zeta=0.0*, *mu=1.0*, *refractive_index=None*, *bound=None*, *initial_field=0.0*, *dtype=None*)

Class a solution that can be further iterated towards a solution for Maxwell's equations in a media specified by the epsilon, xi, zeta, and mu distributions.

**Parameters**

- **grid** (Union[*Grid*, Sequence, ndarray]) – A Grid object or a Sequence of vectors with uniformly increasing values that indicate the positions in a plaid grid of sample points for the material and solution. In the one-dimensional case, a simple increasing Sequence of uniformly-spaced numbers may be provided as an alternative. The length of the ranges determines the data_shape, to which the source_distribution, epsilon, xi, zeta, mu, and initial_field must broadcast when specified as `numpy.ndarray`s.
- **vectorial** (Optional[bool]) – a boolean indicating if the source and solution are 3-vectors-fields (True) or scalar fields (False). Default: True, when vectorial nor the source is specified. Default: vectorial (True), unless the source field is scalar (False if first dimension is a singleton dimension).
- **wavenumber** (Optional[Real]) – the wavenumber in vacuum = 2pi / vacuum_wavelength. The wavelength in the same units as used for the other inputs/outputs.
- **angular_frequency** (Optional[Real]) – alternative argument to the wavenumber = angular_frequency / c
- **vacuum_wavelength** (Optional[Real]) – alternative argument to the wavenumber = 2 pi / vacuum_wavelength
- **current_density** (Union[Complex, Sequence, ndarray, None]) – (optional, instead of source_distribution) An array or function that returns the (vectorial) current density input distribution, J. The current density has units of $Am^-2$.
- **source_distribution** (Union[Complex, Sequence, ndarray, None]) – (optional, instead of current_density) An array or function that returns the (vectorial) source input wave distribution. The source values relate to the current density, J, as 1j * angular_frequency * scipy.constants.mu_0 * J and has units of $rads^-1Hm^-1Am^-2 = radVm^-3$. More general, non-electro-magnetic wave problems can be solved using the source_distribution, as it does not rely on the vacuum permeability constant, $mu_0$.
- **epsilon** (Union[Complex, Sequence, ndarray, None]) – an array or function that returns the (tensor) epsilon that represents the permittivity at the points indicated by the *grid* input argument.
- **xi** (Union[Complex, Sequence, ndarray]) – an array or function that returns the (tensor) xi for bi-(an)isotropy at the points indicated by the *grid* input argument.
- **zeta** (Union[Complex, Sequence, ndarray]) – an array or function that returns the (tensor) zeta for bi-(an)isotropy at the points indicated by the *grid* input argument.
- **mu** (Union[Complex, Sequence, ndarray]) – an array or function that returns the (tensor) permeability at the points indicated by the *grid* input argument.
- **refractive_index** (Union[Complex, Sequence, ndarray, None]) – an array or function that returns the (complex) (tensor) refractive_index, as the square root of the permittivity, at the points indicated by the *grid* input argument.
- **bound** (Optional[*Bound*]) – An object representing the boundary of the calculation volume. Default: None, PeriodicBound(grid)
- **initial_field** (Union[Complex, Sequence, ndarray]) – optional start value for the E-field distribution (default: all zero E)

- **dtype** – optional numpy datatype for the internal operations and results. This must be a complex number type as numpy.complex128 or np.complex64.

**property grid:** *Grid*

The sample positions of the plaid sampling grid. This may be useful for displaying result axes.

**Returns**
A Grid object representing the sample points of the fields and material.

**property vectorial:** bool

Boolean to indicates whether calculations happen on vectorial (True) or scalar (False) fields.

**property dtype**

The numpy equivalent data type used in the calculation. This is either np.complex64 or np.complex128.

**property wavenumber:** Real

The vacuum wavenumber, $k_0$, used in the calculation.

**Returns**
A scalar indicating the wavenumber used in the calculation.

**property angular_frequency:** Real

The angular frequency, $\omega$, used in the calculation.

**Returns**
A scalar indicating the angular frequency used in the calculation.

**property wavelength:** Real

The vacuum wavelength, $\lambda_0$, used in the calculation.

**Returns**
A scalar indicating the vacuum wavelength used in the calculation.

**property magnetic:** bool

Indicates if this media is considered magnetic.

**Returns**
A boolean, True when magnetic, False otherwise.

**property bound:** *Bound*

The Bound object that defines the calculation boundaries.

**property source_distribution:** ndarray

The source distribution, i k0 mu_0 times the current density j.

**Returns**
A complex array indicating the amplitude and phase of the source vector field. The dimensions of the array are [1|3, self.grid.shape], where the first dimension is 1 in case of a scalar field, and 3 in case of a vector field.

**property j:** ndarray

The free current density, j, of the source vector field.

**Returns**
A complex array indicating the amplitude and phase of the current density vector field [A m^-2]. The dimensions of the array are [1|3, self.grid.shape], where the first dimension is 1 in case of a scalar field, and 3 in case of a vector field.

**property E: ndarray**

The electric field for every point in the sample space (SI units).

**Returns**

A vector array with the first dimension containing Ex, Ey, and Ez,

while the following dimensions are the spatial dimensions.

**property B: ndarray**

The magnetic field for every point in the sample space (SI units). This is calculated from H and E.

**Returns**

A vector array with the first dimension containing $B_x, B_y, and B_z$, while the following dimensions are the spatial dimensions.

**property D: ndarray**

The displacement field for every point in the sample space (SI units). This is calculated from E and H.

**Returns**

A vector array with the first dimension containing $D_x, D_y, and D_z$, while the following dimensions are the spatial dimensions.

**property H: ndarray**

The magnetizing field for every point in the sample space (SI units). This is calculated from E.

**Returns**

A vector array with the first dimension containing $H_x, H_y, and H_z$, while the following dimensions are the spatial dimensions.

**property S: ndarray**

The time-averaged Poynting vector for every point in space. :return: A vector array with the first dimension containing $S_x, S_y, and S_z$, while the following dimensions are the spatial dimensions.

**property energy_density: ndarray**

Returns the energy density, u.

**Returns**

A real array indicating the energy density in space.

**property stress_tensor: ndarray**

Maxwell's stress tensor for every point in space.

**Returns**

A real and symmetric matrix-array with the stress tensor for every point in space. The units are $N/m^2$.

**property f: ndarray**

The electromagnetic force density (force per SI unit volume, not per voxel).

**Returns**

A vector array representing the electro-magnetic force exerted per unit volume. The first dimension contains $f_x, f_y, and f_z$, while the following dimensions are the spatial dimensions. The units are $N/m^3$.

**property torque: ndarray**

The electromagnetic force density (force per SI unit volume, not per voxel).

**Returns**

A vector array representing the electro-magnetic torque exerted per unit volume. The first dimension contains torque_x, torque_y, and torque_z, while the following dimensions are the spatial dimensions. The units are $Nm/m^3 = Nm^{-2}$.

**property iteration:** *int*

The current iteration number.

**Returns**
An integer indicating how many iterations have been done.

**property previous_update_norm:** *Real*

The L2-norm of the last update, the difference between current and previous E-field.

**Returns**
A positive scalar indicating the norm of the last update.

**property residue:** *Real*

Returns the current relative residue of the inverse problem $E = H^{-1}S$. The relative residue is return as the l2-norm fraction $||E - H^{-1}S||/||E||$, where H represents the vectorial Helmholtz equation following Maxwell's equations and S the current density source. The solver searches for the electric field, E, that minimizes the preconditioned inverse problem.

**Returns**
A non-negative real scalar that indicates the change in E with the previous iteration normalized to the norm of the current E.

**__iter__**()

Returns an iterator that on __next__() yields this Solution after updating it with one cycle of the algorithm. Obtaining this iterator resets the iteration counter.

Usage:

```
for solution in Solution(...):
    if solution.iteration > 100:
        break
print(solution.residue)
```

**solve**(*callback=<function Solution.<lambda>>*)

Runs the algorithm until the convergence criterion is met or until the maximum number of iterations is reached.

**Parameters**
**callback** (*Callable*) – optional callback function that overrides the one set for the solver. E.g. callback=lambda s: s.iteration < 100

**Returns**
This Solution object, which can be used to query e.g. the final field E using Solution.E.

### 4.4.2 Version History

#### Version 0.2.1

- Fixed GPU memory leaks when using PyTorch back-end, thus enabling to solve larger problems.
- Improved documentation and automated deployment.

### Version 0.2.0

- Implemented the ScatteringMatrix class, which allows calculation of scattering, reflection,and transmission matrices for any material for which the MacroMax solver can calculate the electro-magnetic field.
- Added TensorFlow back-end, allowing access to, for instance, Google Colab's Tensor Processing Units.

### Version 0.1.5

- Added PyTorch back-end, thus enabling the use of a GPU (NVidia CUDA).
- Ensured backwards compatibility with Python 3.6.

### Version 0.1.4

- Streamlined iteration: 3x speed improvement of benchmark with PyFFTW on Intel Core i7-6700.
- Reorganized parallel_ops backend.
- Expanded unit testing.

### Version 0.1.3

- Input arguments for isotropic materials or scalar calculations do not require singleton dimensions on the left anymore.
- `macromax.solve(...)` and `macromax.Solution(...)` now take the optional input argument `refractive_index` as an alternative to the permittivity and permeability.
- The `macromax.bound` module provides the `Bound` class and subclasses to more conveniently specify arbitrary absorbing or periodic boundaries. Custom implementations can be specified as a subclass of `Bound`.
- Convenience class `macromax.Grid` provides an easy method to construct uniformly plaid sample grids and their Fourier-space counterparts.

### Version 0.1.2

- The solve function and Solution constructor now take `dtype` as an argument. By setting dtype=np.complex64 instead of the default dtype=np.complex128, all calculations will be done in single precision. This only requires half the memory, and typically also reduces the calculation time by about 30%.

### Version 0.1.1

- A Grid object can now be used to specify the sampling grid spacing and extent.

### Version 0.1.0

- The current density can now be specified directly for `solve(...)` instead of the `source_density`.
- Reorganized the file structure.
- Removed the `k0^2 = (2pi/wavelength)^2` factor again after the calculations. This normalization factor was introduced to avoid underflow in the iteration. Now it is removed as intended, after the iteration, or when an intermediate solution is queried.

### Version 0.0.9

- Made the dependency on the multiprocessing module optional.

### Version 0.0.8

- Extended unit tests, including the ```utils"' sub-module.
- Improved logging.
- Removed unused ```conductive"' indicator property from solver.
- Brought naming in line with Python conventions.

### Version 0.0.6

- Initial version on production PyPI repository.

# PYTHON MODULE INDEX

## m

# INDEX

## Symbols

## A

## B

## C



## D

## E

## F



## G

## H



## I

## J



## K



## L



## M

## N



## O



## P



## Q



## R

## S

## T

## V



## W