Macroscopic Maxwell Solver

Introduction

This Python 3 module 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 anistropic 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 manuscript "Calculating coherent light-wave propagation in large heterogeneous media."

License: LGPL-3.0

Installation

Prerequisites

This library requires Python 3 with the modules `numpy` and `scipy` for the main calculations. From the main library, the modules `sys`, `io`, `os`, and `multiprocessing` are imported; as well as the modules `logging` and `time` for diagnostics. The `pyfftw` module can help speed up the calculations.

The examples require `matplotlib` for displaying the results. The `pypandoc` module is required for translating this document to other formats.

The code has been tested on Python 3.6.

Installing

Installing the `macromax` module and its dependencies can be done by running the following command in a terminal:

```
pip install macromax
```

The module comes with a submodule containing example code.

The `pypandoc` module requires the separate installation of `pandoc`. Please refer to: https://pypi.org/project/pypandoc/ for instructions on its installation for your operating system of choice.

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

Loading the Python 3 module

The `macromax` module can be imported using:

```
import macromax
```

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

Specifying the material

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.:

```
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 function `utils.calc_ranges` can be used as follows:

```
from macromax import utils
data_shape = (200, 400)
sample_pitch = 50e-9  # or (50e-9, 50e-9)
ranges = utils.calc_ranges(data_shape, sample_pitch)
```

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. 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 contants are zero by default.

Defining the source

The coherent source is defined by an oscillating current density, to model e.g. an incident laser beam. It is sufficient to define its phase, amplitude, and the direction as a function the spatial coordinates; alongside the angular frequency, omega, of the coherent source. To avoid issues with numerical precision, the current density is multiplied by the angular frequency, omega, and the vacuum permeability, mu_0. The source values is proportional to the current density, J, and related as follows: S = i omega mu_0 J with units of rad s^-1 H m^-1 A m^-2 = rad V m^-3.

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.

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:

- `x_range|ranges`: 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.
- `source_distribution`: An ndarray of complex values indicating the source value and direction at each sample point. The source values define the

current density in the sample. The first dimension contains the vector index, the following dimensions contain the spatial dimensions.

- `epsilon`: A complex ndarray that defines the 3x3 permittivity matrix at all sample points. The first two dimensions contain the matrix indices, the following dimensions contain the spatial dimensions.

Anisotropic material properties such as permittivity can be defined as a square 3x3 matrix at each sample point. Isotropic materials may be represented by 1x1 scalars instead (the first two dimensions are singletons). Homogeneous materials may be specified with spatial singleton dimensions.

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.iteration < 1e4 and s.residue > 1e-4
```

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 function and class signature documentation. The examples can be imported using:

```
from macromax import examples
```

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 scipy.constants as const
import matplotlib.pyplot as plt
%matplotlib notebook

#
# Define the material properties
#
wavelength = 500e-9
angular_frequency = 2 * const.pi * const.c / wavelength
source_amplitude = 1j * angular_frequency * const.mu_0
p_source = np.array([0, 1, 0])  # y-polarized

# Set the sampling grid
nb_samples = 1024
sample_pitch = wavelength / 16
x_range = sample_pitch * np.arange(nb_samples) - 4e-6

# define the medium
permittivity = np.ones((1, 1, len(x_range)), dtype=np.complex64)
# Don't forget absorbing boundary:
dist_in_boundary = np.maximum(-(x_range - -1e-6), x_range - 26e-6) / 4e-6
permittivity[:, :, (x_range < -1e-6) | (x_range > 26e-6)] = \
    1.0 + (0.8j * dist_in_boundary[(x_range < -1e-6) | (x_range > 26e-6)])
```

```
# glass has a refractive index of about 1.5
permittivity[:, :, (x_range >= 10e-6) & (x_range < 20e-6)] = 1.5 ** 2

#
# Define the illumination source
#
# point source at x = 0
source = -source_amplitude * sample_pitch * (np.abs(x_range) < sample_pitch/4)
source = p_source[:, np.newaxis] * source[np.newaxis, :]

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

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

x_range = solution.ranges[0]  # coordinates
E = solution.E[1, :]  # Electric field
H = solution.H[2, :]  # Magnetizing field
S = solution.S[0, :]  # Poynting vector
f = solution.f[0, :]  # Optical force
# Display the field for the polarization dimension
field_to_display = angular_frequency * E
max_val_to_display = np.maximum(np.max(np.abs(field_to_display)),
                                np.finfo(field_to_display.dtype).eps)
poynting_normalization = np.max(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])[0]
ax[0].plot(x_range * 1e6, np.real(S) / poynting_normalization,
           color=[1, 0, 1])[0]
ax[0].plot(x_range * 1e6, np.real(field_to_display),
           color=[0, 0.7, 0])[0]
ax[0].plot(x_range * 1e6, np.imag(field_to_display),
           color=[1, 0, 0])[0]
figure_title = "Iteration %d, " % solution.iteration
ax[0].set_title(figure_title)
ax[0].set_xlabel("x  [$\mu$m]")
ax[0].set_ylabel("I, E  [a.u.]")
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('$\epsilon$, $\mu$')
ax[1].set_xlim(x_range[[0, -1]] * 1e6)
ax[1].legend(loc='upper right')
```

Development

Source code organization

The source code is organized as follows:

- `/` (root): Module description and distribution files.
- `/macromax`: The iterative solver.
- `/macromax/examples`: Examples of how the solver can be used.
- `/macromax/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.
- `parallel_ops_column`: 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 `/macromax/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.

Testing

Unit tests are contained in `macromax/tests`. The `ParallelOperations` class in `parallel_ops_column.pi` is pretty well covered and some specific tests have been written for the `Solution` class in `solver.py`. However, the `utils` module does not have any tests at present.

To run the tests:

```
pip install nose
python setup.py test
```

Building and Distributing

The code consists of pure Python 3, hence only packaging is required for distribution. To prepare a package for distribution, run:

```
python setup.py sdist bdist_wheel
pip install . --upgrade
```

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

```
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

Installing from the test repository is done as follows:

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