emg3d Documentation

Release 0.11.0

The emg3d Developers

19 June 2020

User Manual

Version: 0.11.0 ~ Date: 19 June 2020

A multigrid solver for 3D electromagnetic diffusion with tri-axial electrical anisotropy. The matrix-free solver can be used as main solver or as preconditioner for one of the Krylov subspace methods implemented in *scipy.sparse.linalg*, and the governing equations are discretized on a staggered Yee grid. The code is written completely in Python using the NumPy/SciPy-stack, where the most time- and memory-consuming parts are sped up through jitted numba-functions.

CHAPTER 1

More information

For more information regarding installation, usage, contributing, roadmap, bug reports, and much more, see

- **Website**: https://empymod.github.io,
- **Documentation**: https://emg3d.readthedocs.io,
- **Source Code**: https://github.com/empymod/emg3d,
- **Examples**: https://empymod.github.io/emg3d-gallery.

CHAPTER 2

Features

- Multigrid solver for 3D electromagnetic (EM) diffusion with regular grids (where source and receiver can be electric or magnetic).
- Calculate the 3D EM field in the complex frequency domain or in the real Laplace domain.
- Includes also routines to calculate the 3D EM field in the time domain.
- Can be used together with the SimPEG-framework.
- Can be used as a standalone solver or as a pre-conditioner for various Krylov subspace methods implemented in SciPy, e.g., BiCGSTAB (*scipy.sparse.linalg.bicgstab*) or CGS (*scipy.sparse.linalg.cgs*).
- Tri-axial electrical anisotropy.
- Isotropic magnetic permeability.
- Semicoarsening and line relaxation.
- Grid-size can be anything.
- As a multigrid method it scales with the number of unknowns *N* and has therefore optimal complexity *O(N)*.

CHAPTER 3

Installation

You can install emg3d either via `conda` (preferred):

```
conda install -c conda-forge emg3d
```

or via `pip`:

```
pip install emg3d
```

Required are Python version 3.7 or higher and the modules `NumPy`, `SciPy`, `numba`, and `empymod`; `discretize` (from SimPEG) is highly recommended. Consult the installation notes in the manual for more information regarding installation and requirements.

CHAPTER 4

Citation

If you publish results for which you used *emg3d*, please give credit by citing Werthmüller et al. (2019):

> Werthmüller, D., W. A. Mulder, and E. C. Slob, 2019, emg3d: A multigrid solver for 3D electromagnetic diffusion: Journal of Open Source Software, 4(39), 1463; DOI: 10.21105/joss.01463.

All releases have a Zenodo-DOI, which can be found on 10.5281/zenodo.3229006.

See *CREDITS* for the history of the code.

CHAPTER 5

License information

5.1 Getting started

The code `emg3d` ([WeMS19]) is a three-dimensional modeller for electromagnetic (EM) diffusion as used, for instance, in controlled-source EM (CSEM) surveys frequently applied in the search for, amongst other, groundwater, hydrocarbons, and minerals.

The core of the code is primarily based on [Muld06], [Muld07], and [Muld08]. You can read more about the background of the code in the chapter *Credits*. An introduction to the underlying theory of multigrid methods is given in the chapter *Theory*, and further literature is provided in the *References*.

5.1.1 Installation

You can install emg3d either via `conda`:

```
conda install -c conda-forge emg3d
```

or via `pip`:

```
pip install emg3d
```

Required are Python version 3.7 or higher and the modules `NumPy` and `SciPy`, `Numba`, and `empymod`; `discretize` (from SimPEG) is highly recommended.

If you are new to Python we recommend using a Python distribution, which will ensure that all dependencies are met, specifically properly compiled versions of `NumPy` and `SciPy`; we recommend using Anaconda. If you install Anaconda you can simply start the *Anaconda Navigator*, add the channel `conda-forge` and `emg3d` will appear in the package list and can be installed with a click.

You should ensure that you have `NumPy` and `SciPy` installed with the Intel Math Kernel Library `mkl`, as this makes quite a difference in terms of speed. You can check that by running

```
>>> import numpy as np
>>> np.show_config()
```

The output should contain a lot of references to `mkl`, and it should NOT contain references to `blas`, `lapack`, `openblas`, or similar.

5.1.2 Basic Example

Here we show a *very* basic example. To see some more realistic models have a look at the gallery. This particular example is also there, with some further explanations and examples to show how to plot the model and the data; see Minimum working example. It also contains an example without using `discretize`.

First, we load `emg3d` and `discretize` (to create a mesh), along with `numpy`:

```
>>> import emg3d
>>> import discretize
>>> import numpy as np
```

First, we define the mesh (see `discretize.TensorMesh` for more info). In reality, this task requires some careful considerations. E.g., to avoid edge effects, the mesh should be large enough in order for the fields to dissipate, yet fine enough around source and receiver to accurately model them. This grid is too small, but serves as a minimal example.

```
>>> grid = discretize.TensorMesh(
>>>          [[(25, 10, -1.04), (25, 28), (25, 10, 1.04)],
>>>           [(50, 8, -1.03), (50, 16), (50, 8, 1.03)],
>>>           [(30, 8, -1.05), (30, 16), (30, 8, 1.05)]],
>>>          x0='CCC')
>>> print(grid)

  TensorMesh: 49,152 cells

                       MESH EXTENT             CELL WIDTH      FACTOR
  dir    nC         min           max          min       max      max
  ---   ---  ---------------------------  ----------------  ------
   x     48       -662.16        662.16     25.00     37.01    1.04
   y     32       -857.96        857.96     50.00     63.34    1.03
   z     32       -540.80        540.80     30.00     44.32    1.05
```

Next we define a very simple fullspace model with $\rho_x = 1.5\,\Omega\,\mathrm{m}$, $\rho_y = 1.8\,\Omega\,\mathrm{m}$, and $\rho_z = 3.3\,\Omega\,\mathrm{m}$. The source is an x-directed dipole at the origin, with a 10 Hz signal of 1 A.

```
>>> model = emg3d.models.Model(grid, res_x=1.5, res_y=1.8, res_z=3.3)
>>> sfield = emg3d.fields.get_source_field(
>>>     grid, src=[0, 0, 0, 0, 0], freq=10.0)
```

Now we can calculate the electric field with `emg3d`:

```
>>> efield = emg3d.solve(grid, model, sfield, verb=3)

:: emg3d START :: 15:24:40 :: v0.9.1

   MG-cycle       : 'F'                 sslsolver : False
   semicoarsening : False [0]           tol       : 1e-06
   linerelaxation : False [0]           maxit     : 50
   nu_{i,1,c,2}   : 0, 2, 1, 2          verb      : 3
   Original grid  :  48 x  32 x  32     => 49,152 cells
   Coarsest grid  :   3 x   2 x   2     => 12 cells
   Coarsest level :   4 ;   4 ;   4
```

(continues on next page)

(continued from previous page)

```
   [hh:mm:ss]  rel. error                  [abs. error, last/prev]   l s

       h_
      2h_ \                  /
      4h_  \          /\    /
      8h_   \    /\  /  \  /
     16h_    \/\/  \/    \/

   [11:18:17]   2.623e-02  after   1 F-cycles   [1.464e-06, 0.026]   0 0
   [11:18:17]   2.253e-03  after   2 F-cycles   [1.258e-07, 0.086]   0 0
   [11:18:17]   3.051e-04  after   3 F-cycles   [1.704e-08, 0.135]   0 0
   [11:18:17]   5.500e-05  after   4 F-cycles   [3.071e-09, 0.180]   0 0
   [11:18:18]   1.170e-05  after   5 F-cycles   [6.531e-10, 0.213]   0 0
   [11:18:18]   2.745e-06  after   6 F-cycles   [1.532e-10, 0.235]   0 0
   [11:18:18]   6.873e-07  after   7 F-cycles   [3.837e-11, 0.250]   0 0

   > CONVERGED
   > MG cycles         : 7
   > Final rel. error : 6.873e-07

:: emg3d END   :: 15:24:42 :: runtime = 0:00:02
```

So the calculation required seven multigrid F-cycles and took just a bit more than 2 seconds. It was able to coarsen in each dimension four times, where the input grid had 49,152 cells, and the coarsest grid had 12 cells.

5.1.3 Related ecosystem

The hard dependencies for emg3d are with `numpy`, `scipy`, `numba`, and `empymod` comparably low. However, emg3d is, as such, "only" a solver. It does not contain fancy grid- nor model-creation routines or plotting functions. There exist other packages which do that much better.

To create advanced meshes it is recommended to use discretize from the SimPEG framework. It also comes with some neat plotting functionalities to plot model parameters and resulting fields. Furthermore, it can serve as a link to use PyVista to create nice 3D plots even within a notebook.

Projects which can be used to compare or validate the results are, e.g., empymod for layered models or SimPEG for 3D models. It is also possible to create a geological model with GemPy and, again via discretize, move it to emg3d to calculate CSEM responses for it.

Have a look at the gallery for many examples of how to use emg3d together with the mentioned projects and more!

5.1.4 Tipps and Tricks

The function `emg3d.solve()` is the main entry point, and it takes care whether multigrid is used as a solver or as a preconditioner (or not at all), while the actual multigrid solver is *`emg3d.solver.multigrid()`*. Most input parameters for `emg3d.solve()` are sufficiently described in its docstring. Here a few additional information.

- You can input any three-dimensional grid into *emg3d*. However, the implemented multigrid technique works with the existing nodes, meaning there are no new nodes created as coarsening is done by combining adjacent cells. The more times the grid dimension can be divided by two the better it is suited for MG. Ideally, the dimension of the coarsest grid should be a low prime number p, for which good sizes can then be calculated with $p2^n$. Good grid sizes (in each direction) up to 1024 are
  - $2^{2^{0,1,\ldots,9}}$: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024,
  - $3^{2^{0,1,\ldots,8}}$: 3, 6, 12, 24, 48, 96, 192, 384, 768,
  - $5^{2^{0,1,\ldots,7}}$: 5, 10, 20, 40, 80, 160, 320, 640,
  - $7^{2^{0,1,\ldots,7}}$: 7, 14, 28, 56, 112, 224, 448, 896,

and preference decreases from top to bottom row. Good grid sizes in sequential order: 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 20, 24, 28, 32, 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320, 384, 448, 512, 640, 768, 896, 1024.

- The multigrid method can be used as a solver or as a preconditioner, for instance for BiCGSTAB. Using multigrid as a preconditioner for BiCGSTAB together with semicoarsening and line relaxation is the most stable version, but expensive, and therefore only recommended on highly stretched grids. Which combination of solver is best (fastest) depends to a large extent on the grid stretching. As a rule of thumb:
  - No stretching: Multigrid (MG);
  - Moderate stretching (< 1.04): BiCGSTAB with MG as pre-conditioner;
  - Strong stretching (> 1.04): BicGSTAB with MG as preconditioner and line relaxation/semicoarsening.

5.1.5 Contributing and Roadmap

New contributions, bug reports, or any kind of feedback is always welcomed! Have a look at the Roadmap-project to get an idea of things that could be implemented. The GitHub issues and PR's are also a good starting point. The best way for interaction is at https://github.com/empymod or by joining the Slack channel «em-x-d» of SimPEG. If you prefer to get in touch outside of GitHub/Slack use the contact form on https://werthmuller.org.

To install emg3d from source, you can download the latest version from GitHub and install it in your python distribution via:

```
python setup.py install
```

Please make sure your code follows the pep8-guidelines by using, for instance, the python module `flake8`, and also that your code is covered with appropriate tests. Just get in touch if you have any doubts.

The structure of `emg3d` is:

- `solver`: These are the main routines, the flow of the multigrid method;
- `njited`: The expensive parts (computation, memory) are here in jitted functions; and
- `utils`: Some helper routines.

5.1.6 Tests and benchmarks

The modeller comes with a test suite using `pytest`. If you want to run the tests, just install `pytest` and run it within the `emg3d`-top-directory.

```
> pytest --cov=emg3d --flake8
```

It should run all tests successfully. Please let us know if not!

Note that installations of `em3gd` via conda or pip do not have the test-suite included. To run the test-suite you must download `emg3d` from GitHub.

There is also a benchmark suite using *airspeed velocity*, located in the empymod/emg3d-asv-repository. The results of my machine can be found in the empymod/emg3d-bench, its rendered version at empymod.github.io/emg3d-asv.

5.1.7 License

5.2 Multi-what?

If you have never heard of the multigrid method before you might ask yourself "*multi-what?*" The following is an intent to describe the multigrid method without the maths; just some keywords and some figures. **It is a heavily simplified intro, using a 2D grid for simplicity.** Have a look at the *Theory*-section for more details. A good, four-page intro with some maths is given by [Muld11]. More in-depth information can be found, e.g., in [BrHM00], [Hack85], and [Wess91].

The multigrid method ([Fedo64])

- is an iterative solver;
- scales almost linearly (CPU & RAM);
- can serve as a pre-conditioner or as a solver on its own.

The main driving motivation to use multigrid is the part about linear scaling.

5.2.1 Matrix-free solver

The implemented multigrid method is a matrix free solver, it never constructs the full matrix. This is how it achieves its relatively low memory consumption. To solve the system, it solves for all fields adjacent to one node, moves then to the next node, and so on until it reaches the last node, see Figure 5.1, where the red lines indicate the fields which are solved simultaneously per step (the fields on the boundaries are never calculated, as they are assumed to be 0).

Figure 5.1:: The multigrid solver solves by default on a node-by-node basis.

Normally, you would have to do this over and over again to achieve a good approximate solution. multigrid typically does it only a few times per grid, typically 2 times (one forward, one backward). This is why it is called **smoother**, as it only smoothes the error, it does not solve it. The implemented method for this is the *Gauss-Seidel* method.

Iterative solver which work in this matrix-free manner are typically **very fast at solving for the local problem**, hence at reducing the **high frequency error**, but **very slow at solving the global problem**, hence at reducing the **low frequency error**. High and low frequency errors are meant relatively to cell-size here.

5.2.2 Moving between different grids

The main thinking behind multigrid is now that we move to coarser grids. This has two advantages:

- Fewer cells means faster calculation and less memory.
- Coarser grid size transforms lower frequency error to higher frequency error, relatively to cell size, which means faster convergence.

The implemented multigrid method simply joins two adjacent cells to get from finer to coarser grids, see Figure 5.2 for an example coarsening starting with a 16 cells by 16 cells grid.

Figure 5.2:: Example of the implemented coarsening scheme.

There are different approaches how to cycle through different grid sizes, see Figures 5.7 to 5.9. The downsampling from a finer grid to a coarser grid is often termed **restriction**, whereas the interpolation from a coarser grid to a finer grid is termed **prolongation**.

5.2.3 Specialities

The convergence rate of the multigrid method suffers on severely stretched grids or by models with strong anisotropy. Two techniques are implemented, **semicoarsening** (Figure 5.3) and **line relaxation** (Figure 5.4). Both require more CPU and higher RAM per grid than the standard multigrid, but they can improve the convergence rate, which then in turn improves the overall CPU time.

Figure 5.3:: Example of semicoarsening: The cell size is kept constant in one direction. The direction can be alternated between iterations.

Figure 5.4:: Example of line relaxation: The system is solved for all fields adjacent to a whole line of nodes simultaneously in some direction. The direction can be alternated between iterations.

5.3 Theory

The following provides an introduction to the theoretical foundation of the solver *emg3d*. More specific theory is covered in the docstrings of many functions, have a look at the *Code*-section or follow the links to the corresponding functions here within the theory. If you just want to use the solver, but do not care much about the internal functionality, then the function `emg3d.solve()` is the only function you will ever need. It is the main entry point, and it takes care whether multigrid is used as a solver or as a preconditioner (or not at all), while the actual multigrid solver is *`emg3d.solver.multigrid()`*.

Note: This section is not an independent piece of work. Most things are taken from one of the following sources:

- [Muld06], pages 634-639:
  - The *Maxwell's equations* and *Discretisation* sections correspond with some adjustemens and additions to pages 634-636.
  - The start of *The Multigrid Method* corresponds roughly to page 637.
  - Pages 638 and 639 are in parts reproduced in the code-docstrings of the corresponding functions.
- [BrHM00]: This book is an excellent introduction to multigrid methods. Particularly the *Iterative Solvers* section is taken to a big extent from the book.

Please consult these original resources for more details, and refer to them for citation purposes and not to this manual. More in-depth information can also be found in, e.g., [Hack85] and [Wess91].

5.3.1 Maxwell's equations

Maxwell's equations in the presence of a current source $\mathbf{J}_\mathrm{s}$ are

$$\begin{aligned} \partial_t \mathbf{B}(\mathbf{x},t) + \nabla \times \mathbf{E}(\mathbf{x},t) &= 0, \\ \nabla \times \mathbf{H}(\mathbf{x},t) - \partial_t \mathbf{D}(\mathbf{x},t) &= \mathbf{J}_\mathrm{c}(\mathbf{x},t) + \mathbf{J}_\mathrm{s}(\mathbf{x},t), \end{aligned} \tag{5.1}$$

where the conduction current $\mathbf{J}_\mathrm{c}$ obeys Ohm's law,

$$\mathbf{J}_\mathrm{c}(\mathbf{x},t) = \sigma(\mathbf{x})\mathbf{E}(\mathbf{x},t). \tag{5.2}$$

Here, $\sigma(\mathbf{x})$ is the conductivity. $\mathbf{E}(\mathbf{x},t)$ is the electric field and $\mathbf{H}(\mathbf{x},t)$ is the magnetic field. The electric displacement $\mathbf{D}(\mathbf{x},t) = \varepsilon(\mathbf{x})\mathbf{E}(\mathbf{x},t)$ and the magnetic induction $\mathbf{B}(\mathbf{x},t) = \mu(\mathbf{x})\mathbf{H}(\mathbf{x},t)$. The dielectric constant or permittivity ε can be expressed as $\varepsilon = \varepsilon_r\varepsilon_0$, where ε_r is the relative permittivity and ε_0 is the vacuum value. Similarly, the magnetic permeability μ can be written as $\mu = \mu_r\mu_0$, where μ_r is the relative permeability and μ_0 is the vacuum value.

The magnetic field can be eliminated from Equation (5.1), yielding the second-order parabolic system of equations,

$$\varepsilon\partial_{tt}\mathbf{E} + \sigma\partial_t\mathbf{E} + \nabla\times\mu^{-1}\nabla\times\mathbf{E} = -\partial_t\mathbf{J}_\mathrm{s}. \tag{5.3}$$

To transform from the time domain to the frequency domain, we substitute

$$\mathbf{E}(\mathbf{x},t) = \frac{1}{2\pi}\int_{-\infty}^{\infty}\hat{\mathbf{E}}(\mathbf{x},\omega)e^{-\mathrm{i}\omega t}\,d\omega, \tag{5.4}$$

and use a similar representation for $\mathbf{H}(\mathbf{x},t)$. The resulting system of equations is

$$-s\mu_0(\sigma+s\varepsilon)\hat{\mathbf{E}} - \nabla\times\mu_r^{-1}\nabla\times\hat{\mathbf{E}} = s\mu_0\hat{\mathbf{J}}_s, \tag{5.5}$$

where $s=-\mathrm{i}\omega$. The multigrid method converges in the case of the diffusive approximation (with its smoothing and approximation properties), but not in the high-frequency range (at least not in the implemented form of the multigrid method in `emg3d`). The code `emg3d` assumes therefore the diffusive approximation, hence only low frequencies are considered that obey $|\omega\varepsilon| \ll \sigma$. In this case we can set $\varepsilon=0$, and Equation (5.5) simplifies to

$$-s\mu_0\sigma\hat{\mathbf{E}} - \nabla\times\mu_r^{-1}\nabla\times\hat{\mathbf{E}} = s\mu_0\hat{\mathbf{J}}_s, \tag{5.6}$$

From here on, the hats are omitted. We use the perfectly electrically conducting boundary

$$\mathbf{n}\times\mathbf{E} = 0 \quad\text{and}\quad \mathbf{n}\cdot\mathbf{H} = 0, \tag{5.7}$$

where $\mathbf{n}$ is the outward normal on the boundary of the domain.

The Maxwell's equations and Ohm's law are solved in the **frequency domain**. The **time-domain** solution can be obtained by taking the inverse Fourier transform.

Note: [Muld06] uses the time convention $e^{-\mathrm{i}\omega t}$, see Equation (5.4), with $s=-\mathrm{i}\omega$. However, the code *emg3d* uses the convention $e^{\mathrm{i}\omega t}$, hence $s=\mathrm{i}\omega$. This is the same convention as used in *empymod*, and commonly in CSEM.

Laplace domain

It is also possible to solve the problem in the **Laplace domain**, by using a real value for s in Equation (5.6), instead of the complex value $-\mathrm{i}\omega$. This simplifies the problem from complex numbers to real numbers, which accelerates the calculation. It also improves the convergence rate, as the solution is a smoother function. The solver `emg3d.solve()` is agnostic to the data type of the provided source field, and can solve for real and complex problems, hence frequency and Laplace domain. See the documentation of the functions *emg3d.fields.get_source_field()* and *emg3d.models.Model()* to see how you can use *emg3d* for Laplace-domain calculations.

5.3.2 Discretisation

Equation (5.6) can be discretised by the finite-integration technique ([Weil77], [ClWe01]). This scheme can be viewed as a finite-volume generalization of [Yee66]'s scheme for tensor-product Cartesian grids with variable grid spacings. An error analysis for the constant-coefficient case ([MoSu94]) showed that both the electric and magnetic field components have second-order accuracy.

Consider a tensor-product Cartesian grid with nodes at positions (x_k, y_l, z_m), where $k=0,\ldots,N_x, l=0,\ldots,N_y$ and $m=0,\ldots,N_z$. There are $N_x\times N_y\times N_z$ cells having these nodes as vertices. The cell centres are located at

$$\begin{aligned} x_{k+1/2} &= \tfrac{1}{2}\left(x_k + x_{k+1}\right), \\ y_{l+1/2} &= \tfrac{1}{2}\left(y_l + y_{l+1}\right), \\ z_{m+1/2} &= \tfrac{1}{2}\left(z_m + z_{m+1}\right). \end{aligned} \tag{5.8}$$

The material properties, σ and μ_r, are assumed to be given as cell-averaged values. The electric field components are positioned at the edges of the cells, as shown in Figure 5.5, in a manner similar to Yee's scheme. The first component of the electric field $E_{1,k+1/2,l,m}$ should approximate the average of $E_1(x, y_l, z_m)$ over the edge from x_k to x_{k+1} at given y_l and z_m. Here, the average is defined as the line integral divided by the length of the integration interval. The other components, $E_{2,k,l+1/2,m}$ and $E_{3,k,l,m+1/2}$, are defined in a similar way. Note that these averages may also be interpreted as point values at the midpoint of edges:

$$\begin{aligned} E_{1,k+1/2,l,m} &\simeq E_1\left(x_{k+1/2}, y_l, z_m\right), \\ E_{2,k,l+1/2,m} &\simeq E_2\left(x_k, y_{l+1/2}, z_m\right), \\ E_{3,k,l,m+1/2} &\simeq E_3\left(x_k, y_l, z_{m+1/2}\right). \end{aligned} \tag{5.9}$$

The averages and point-values are the same within second-order accuracy.

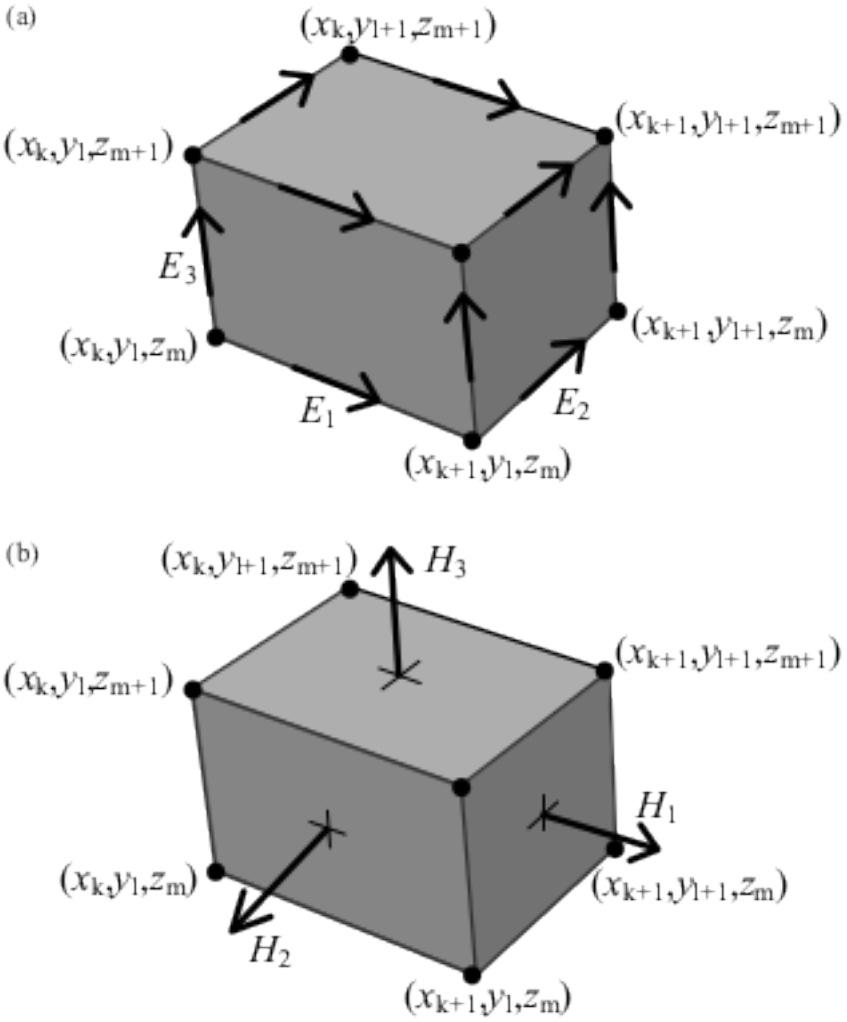

Figure 5.5:: (a) A grid cell with grid nodes and edge-averaged components of the electric field. (b) The face-averaged magnetic field components that are obtained by taking the curl of the electric field.

For the discretisation of the term $-s\mu_0\sigma\mathbf{E}$ related to Ohm's law, dual volumes related to edges are introduced. For a given edge, the dual volume is a quarter of the total volume of the four adjacent cells. An example for E_1 is shown in Figure 5.6(b). The vertices of the dual cell are located at the midpoints of the cell faces.

The volume of a normal cell is defined as

$$V_{k+1/2,l+1/2,m+1/2} = h^x_{k+1/2} h^y_{l+1/2} h^z_{m+1/2}, \tag{5.10}$$

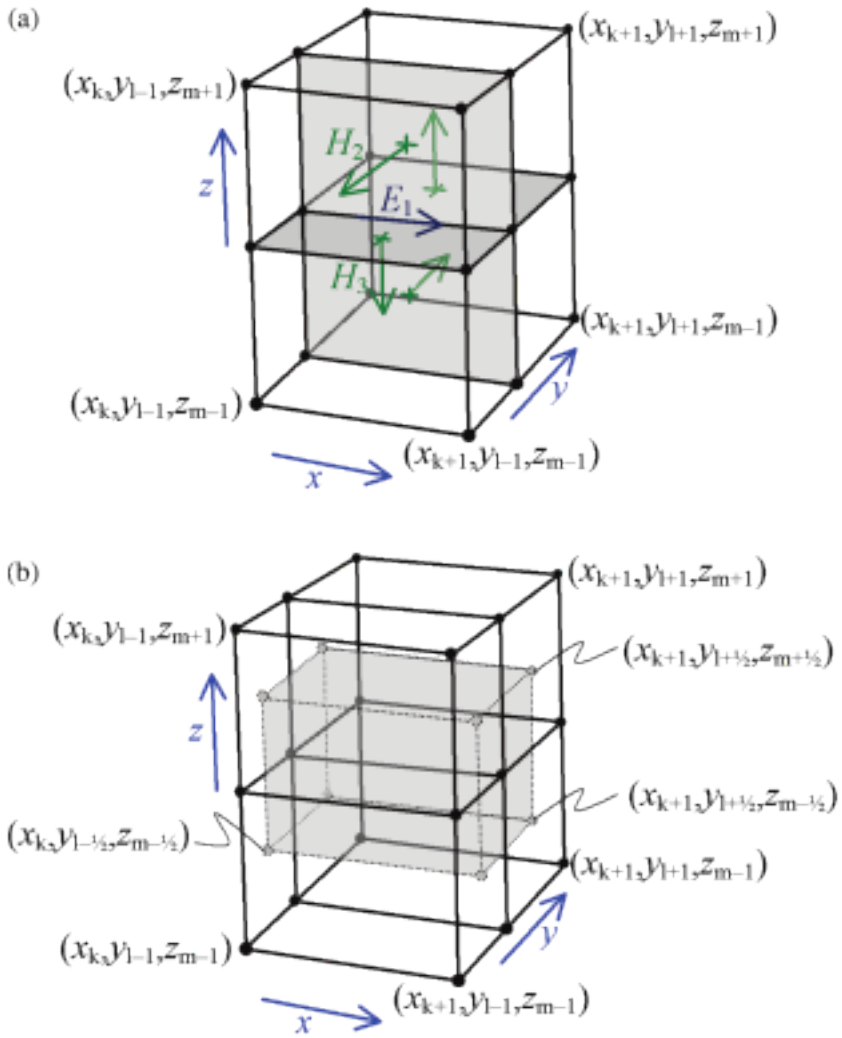

Figure 5.6:: The first electric field component $E_{1,k,l,m}$ is located at the intersection of the four cells shown in (a). Four faces of its dual volume are sketched in (b). The first component of the curl of the magnetic field should coincide with the edge on which E_1 is located. The four vectors that contribute to this curl are shown in (a). They are defined as normals to the four faces in (a). Before computing their curl, these vectors are interpreted as tangential components at the faces of the dual volume shown in (b). The curl is evaluated by taking the path integral over a rectangle of the dual volume that is obtained for constant x and by averaging over the interval $[x_k, x_{k+1}]$.

where

$$\begin{aligned} h^x_{k+1/2} &= x_{k+1} - x_k, \\ h^y_{l+1/2} &= y_{l+1} - y_l, \\ h^z_{m+1/2} &= z_{m+1} - z_m. \end{aligned} \tag{5.11}$$

For an edge parallel to the x-axis on which $E_{1,k+1/2,l,m}$ is located, the dual volume is

$$V_{k+1/2,l,m} = \tfrac{1}{4} h^x_{k+1/2} \sum_{m_2=0}^{1} \sum_{m_3=0}^{1} h^y_{l-1/2+m_2} h^z_{m-1/2+m_3}. \tag{5.12}$$

With the definitions,

$$\begin{aligned} d^x_k &= x_{k+1/2} - x_{k-1/2}, \\ d^y_l &= y_{l+1/2} - y_{l-1/2}, \\ d^z_m &= z_{m+1/2} - z_{m-1/2}, \end{aligned} \tag{5.13}$$

we obtain

$$\begin{aligned} V_{k+1/2,l,m} &= h^x_{k+1/2} d^y_l d^z_m, \\ V_{k,l+1/2,m} &= d^x_k h^y_{l+1/2} d^z_m, \\ V_{k,l,m+1/2} &= d^x_k d^y_l h^z_{m+1/2}. \end{aligned} \tag{5.14}$$

Note that Equation (5.13) does not define d^x_k, etc., at the boundaries. We may simply take $d^x_0 = h^x_{1/2}$ at $k = 0$, $d^x_{N_x} = h^x_{N_x-1/2}$ at $k = N_x$ and so on, or use half of these values as was done by [MoSu94].

The discrete form of the term $-s\mu_0\sigma\mathbf{E}$ in Equation (5.6), with each component multiplied by the corresponding dual volume, becomes $\mathcal{S}_{k+1/2,l,m}\, E_{1,k+1/2,l,m}$, $\mathcal{S}_{k,l+1/2,m}\, E_{2,k,l+1/2,m}$ and $\mathcal{S}_{k,l,m+1/2}\, E_{3,k,l,m+1/2}$ for the first, second and third components, respectively. Here $\mathcal{S} = -s\mu_0\sigma V$ is defined in terms of cell-averages. At the edges parallel to the x-axis, an averaging procedure similar to (5.12) gives

$$\begin{aligned} \mathcal{S}_{k+1/2,l,m} =& \tfrac{1}{4} \left(\mathcal{S}_{k+1/2,l-1/2,m-1/2} + \mathcal{S}_{k+1/2,l+1/2,m-1/2} \right. \\ &\left. + \mathcal{S}_{k+1/2,l-1/2,m+1/2} + \mathcal{S}_{k+1/2,l+1/2,m+1/2} \right). \end{aligned} \tag{5.15}$$

$\mathcal{S}_{k,l+1/2,m}$ and $\mathcal{S}_{k,l,m+1/2}$ are defined in a similar way.

The curl of $\mathbf{E}$ follows from path integrals around the edges that bound a face of a cell, drawn in Figure 5.5(a). After division by the area of the faces, the result is a face-averaged value that can be positioned at the centre of the face, as sketched in Figure 5.5(b). If this result is divided by $\mathrm{i}\omega\mu$, the component of the magnetic field that is normal to the face is obtained. In order to find the curl of the magnetic field, the magnetic field components that are normal to faces are interpreted as tangential components at the faces of the dual volumes. For E_1, this is shown in Figure 5.6. For the first component of Equation (5.6) on the edge $(k + 1/2, l, m)$ connecting (x_k, y_l, z_m) and (x_{k+1}, y_l, z_m), the corresponding dual volume comprises the set $[x_k, x_{k+1}] \times [y_{l-1/2}, y_{l+1/2}] \times [z_{m-1/2}, z_{m+1/2}]$ having volume $V_{k+1/2,l,m}$.

The scaling by μ_r^{-1} at the face requires another averaging step because the material properties are assumed to be given as cell-averaged values. We define $\mathcal{M} = V\mu_r^{-1}$, so

$$\mathcal{M}_{k+1/2,l+1/2,m+1/2} = h^x_{k+1/2} h^y_{l+1/2} h^z_{m+1/2} \mu^{-1}_{r,k+1/2,l+1/2,m+1/2} \tag{5.16}$$

for a given cell $(k + 1/2, l + 1/2, m + 1/2)$. An averaging step in, for instance, the z-direction gives

$$\mathcal{M}_{k+1/2,l+1/2,m} = \tfrac{1}{2} \left(\mathcal{M}_{k+1/2,l+1/2,m-1/2} + \mathcal{M}_{k+1/2,l+1/2,m+1/2} \right) \tag{5.17}$$

at the face $(k + 1/2, l + 1/2, m)$ between the cells $(k + 1/2, l + 1/2, m - 1/2)$ and $(k + 1/2, l + 1/2, m + 1/2)$.

Starting with $\mathbf{v} = \nabla \times \mathbf{E}$, we have

$$\begin{aligned} v_{1,k,l+1/2,m+1/2} =& e^y_{l+1/2} \left(E_{3,k,l+1,m+1/2} - E_{3,k,l,m+1/2} \right) \\ &- e^z_{m+1/2} \left(E_{2,k,l+1/2,m+1} - E_{2,k,l+1/2,m} \right), \\ v_{2,k+1/2,l,m+1/2} =& e^z_{m+1/2} \left(E_{1,k+1/2,l,m+1} - E_{1,k+1/2,l,m} \right) \\ &- e^x_{k+1/2} \left(E_{3,k+1,l,m+1/2} - E_{3,k,l,m+1/2} \right), \\ v_{3,k+1/2,l+1/2,m} =& e^x_{k+1/2} \left(E_{2,k+1/2,l+1,m} - E_{1,k+1/2,l,m} \right) \\ &- e^y_{l+1/2} \left(E_{1,k+1/2,l+1,m} - E_{1,k+1/2,l,m} \right). \end{aligned} \tag{5.18}$$

Here,

$$e^x_{k+1/2} = 1/h^x_{k+1/2}, \quad e^y_{l+1/2} = 1/h^y_{l+1/2}, \quad e^z_{m+1/2} = 1/h^z_{m+1/2}. \tag{5.19}$$

Next, we let

$$\begin{aligned} u_{1,k,l+1/2,m+1/2} &= \mathcal{M}_{k,l+1/2,m+1/2} v_{1,k,l+1/2,m+1/2}, \\ u_{2,k+1/2,l,m+1/2} &= \mathcal{M}_{k+1/2,l,m+1/2} v_{2,k+1/2,l+1/2,m}, \\ u_{3,k+1/2,l+1/2,m} &= \mathcal{M}_{k+1/2,l+1/2,m} v_{3,k+1/2,l+1/2,m}. \end{aligned} \tag{5.20}$$

Note that these components are related to the magnetic field components by

$$\begin{aligned} u_{1,k,l+1/2,m+1/2} &= \mathrm{i}\omega\mu_0 V_{k,l+1/2,m+1/2} H_{1,k+1/2,l,m+1/2}, \\ u_{2,k+1/2,l,m+1/2} &= \mathrm{i}\omega\mu_0 V_{k+1/2,l,m+1/2} H_{2,k+1/2,l,m+1/2}, \\ u_{3,k+1/2,l+1/2,m} &= \mathrm{i}\omega\mu_0 V_{k+1/2,l+1/2,m} H_{3,k+1/2,l+1/2,m}, \end{aligned} \tag{5.21}$$

where

$$\begin{aligned} V_{k,l+1/2,m+1/2} &= d^x_k h^y_{l+1/2} h^z_{m+1/2}, \\ V_{k+1/2,l,m+1/2} &= h^x_{k+1/2} d^y_l h^z_{m+1/2}, \\ V_{k+1/2,l+1/2,m} &= h^x_{k+1/2} h^y_{l+1/2} d^z_m. \end{aligned} \tag{5.22}$$

The discrete representation of the source term $\mathrm{i}\omega\mu_0\mathbf{J}_\mathrm{s}$, multiplied by the appropriate dual volume, is

$$\begin{aligned} s_{1,k+1/2,l,m} &= \mathrm{i}\omega\mu_0 V_{k+1/2,l,m} J_{1,k+1/2,l,m}, \\ s_{2,k,l+1/2,m} &= \mathrm{i}\omega\mu_0 V_{k,l+1/2,m} J_{2,k,l+1/2,m}, \\ s_{3,k,l,m+1/2} &= \mathrm{i}\omega\mu_0 V_{k,l,m+1/2} J_{3,k,l,m+1/2}. \end{aligned} \tag{5.23}$$

Let the residual for an arbitrary electric field that is not necessarily a solution to the problem be defined as

$$\mathbf{r} = V\left(\mathrm{i}\omega\mu_0\mathbf{J}_\mathrm{s} + -s\mu_0\sigma\mathbf{E} - \nabla\times\mu_\mathrm{r}^{-1}\nabla\times\mathbf{E}\right). \tag{5.24}$$

Its discretisation is

$$\begin{aligned} r_{1,k+1/2,l,m} &= s_{1,k+1/2,l,m} + \mathcal{S}_{k+1/2,l,m} E_{1,k+1/2,l,m} \\ &\quad - \left[e^y_{l+1/2} u_{3,k+1/2,l+1/2,m} - e^y_{l-1/2} u_{3,k+1/2,l-1/2,m]}\right. \\ &\quad + \left.\left[e^z_{m+1/2} u_{2,k+1/2,l,m+1/2} - e^z_{m-1/2} u_{2,k+1/2,l,m-1/2}\right\right], \\ r_{2,k,l+1/2,m} &= s_{2,k,l+1/2,m} + \mathcal{S}_{k,l+1/2,m} E_{2,k,l+1/2,m} \\ &\quad - \left[e^z_{m+1/2} u_{1,k,l+1/2,m+1/2} - e^z_{m-1/2} u_{1,k,l+1/2,m-1/2]}\right. \\ &\quad + \left.\left[e^x_{k+1/2} u_{3,k+1/2,l+1/2,m} - e^x_{k-1/2} u_{3,k-1/2,l+1/2,m]}\right\right], \\ r_{3,k,l,m+1/2} &= s_{3,k,l,m+1/2} + \mathcal{S}_{k,l,m+1/2} E_{3,k,l,m+1/2} \\ &\quad - \left[e^x_{k+1/2} u_{2,k+1/2,l,m+1/2} - e^x_{k-1/2} u_{2,k-1/2,m+1/2]}\right. \\ &\quad + \left.\left[e^y_{l+1/2} u_{1,k,l+1/2,m+1/2} - e^y_{l-1/2} u_{1,k,l-1/2,m+1/2}\right\right]. \end{aligned} \tag{5.25}$$

The weighting of the differences in u_1, etc., may appear strange. The reason is that the differences have been multiplied by the local dual volume. As already mentioned, the dual volume for $E_{1,k,l,m}$ is shown in Figure 5.6(b).

For further details of the discretisation see [Muld06] or [Yee66]. The actual meshing is done using discretize (part of the SimPEG-framework). The coordinate system of `discretize` uses a coordinate system were positive z is upwards.

The method is implemented in a matrix-free manner: the large sparse linear matrix that describes the discretised problem is never explicitly formed, only its action is evaluated on the latest estimate of the solution, thereby reducing storage requirements.

5.3.3 Iterative Solvers

The multigrid method is an iterative (or relaxation) method and shares as such the underlying idea of iterative solvers. We want to solve the linear equation system

$$A\mathbf{x} = \mathbf{b}, \tag{5.26}$$

where A is the $n \times n$ system matrix and x the unknown. If v is an approximation to x, then we can define two important measures. One is the error e

$$\mathbf{e} = \mathbf{x} - \mathbf{v}, \tag{5.27}$$

which magnitude can be measured by any standard vector norm, for instance the maximum norm and the Euclidean or 2-norm defined respectively, by

$$\|\mathbf{e}\|_\infty = \max_{1\leq j\leq n} |e_j| \quad \text{and} \quad \|\mathbf{e}\|_2 = \sqrt{\sum_{j=1}^{n} e_j^2}.$$

However, as the solution is not known the error cannot be calculated either. The second important measure, however, is a computable measure, the residual r (calculated in `emg3d.solver.residual()`)

$$\mathbf{r} = \mathbf{b} - A\mathbf{v}. \tag{5.28}$$

Using Equation (5.27) we can rewrite Equation (5.26) as

$$A\mathbf{e} = \mathbf{b} - A\mathbf{v},$$

from which we obtain with Equation (5.28) the *Residual Equation*

$$A\mathbf{e} = \mathbf{r}. \tag{5.29}$$

The *Residual Correction* is given by

$$\mathbf{x} = \mathbf{v} + \mathbf{e}. \tag{5.30}$$

5.3.4 The Multigrid Method

Note: If you have never heard of multigrid methods before you might want to read through the *Multi-what?*-section.

Multigrid is a numerical technique for solving large, often sparse, systems of equations, using several grids at the same time. An elementary introduction can be found in [BrHM00]. The motivation for this approach follows from the observation that it is fairly easy to determine the local, short-range behaviour of the solution, but more difficult to find its global, long-range components. The local behaviour is characterized by oscillatory or rough components of the solution. The slowly varying smooth components can be accurately represented on a coarser grid with fewer points. On coarser grids, some of the smooth components become oscillatory and again can be easily determined.

The following constituents are required to carry out multigrid. First, a sequence of grids is needed. If the finest grid on which the solution is to be found has a constant grid spacing h, then it is natural to define coarser grids with spacings of $2h$, $4h$, etc. Let the problem on the finest grid be defined by $A^h\mathbf{x}^h = \mathbf{b}^h$. The residual is $\mathbf{r}^h = \mathbf{b}^h - A^h\mathbf{x}^h$ (see the corresponding function `emg3d.solver.residual()`, and for more details also the function `emg3d.core.amat_x()`). To find the oscillatory components for this problem, a smoother or relaxation scheme is applied. Such a scheme is usually based on an approximation of A^h that is easy to invert. After one or more smoothing steps (see the corresponding function `emg3d.solver.smoothing()`), say ν_1 in total, convergence will slow down because it is generally difficult to find the smooth, long-range components of the solution. At this point, the problem is mapped to a coarser grid, using a restriction operator $\tilde{I}_h^{2h}$ (see

the corresponding function `emg3d.solver.restriction()`, and for more details, the functions `emg3d.core.restrict_weights()` and `emg3d.core.restrict()`. On the coarse-grid, $\mathbf{b}^{2h} = \tilde{I}_h^{2h}\mathbf{r}^h$. The problem $\mathbf{r}^{2h} = \mathbf{b}^{2h} - A^{2h}\mathbf{x}^{2h} = 0$ is now solved for $\mathbf{x}^{2h}$, either by a direct method if the number of points is sufficiently small or by recursively applying multigrid. The resulting approximate solution needs to be interpolated back to the fine grid and added to the solution. An interpolation operator I_{2h}^h, usually called prolongation in the context of multigrid, is used to update $\mathbf{x}^h := \mathbf{x}^h + I_{2h}^h\mathbf{x}^{2h}$ (see the corresponding function `emg3d.solver.prolongation()`). Here $I_{2h}^h\mathbf{x}^{2h}$ is called the coarse-grid correction. After prolongation, ν_2 additional smoothing steps can be applied. This constitutes one multigrid iteration.

So far, we have not specified the coarse-grid operator A^{2h}. It can be formed by using the same discretisation scheme as that applied on the fine grid. Another popular choice, $A^{2h} = \tilde{I}_h^{2h}A^hI_{2h}^h$, has not been considered here. Note that the tilde is used to distinguish restriction of the residual from operations on the solution, because these act on elements of different function spaces.

If multigrid is applied recursively, a strategy is required for moving through the various grids. The simplest approach is the V-cycle shown in Figure 5.7 for the case of four grids. Here, the same number of pre- and post-smoothing steps is used on each grid, except perhaps on the coarsest. In many cases, the V-cycle does not solve the coarse-grid equations sufficiently well. The W-cycle, shown in Figure 5.8, will perform better in that case. In a W-cycle, the number of coarse-grid corrections is doubled on subsequent coarser grids, starting with one coarse-grid correction on the finest grid. Because of its cost, it is often replaced by the F-cycle (Figure 5.9). In the F-cycle, the number of coarse-grid corrections increases by one on each subsequent coarser grid.

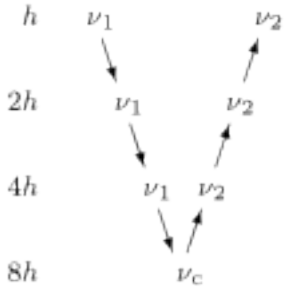

Figure 5.7:: V-cycle with ν_1 pre-smoothing steps and ν_2 post-smoothing steps. On the coarsest grid, ν_c smoothing steps are applied or an exact solver is used. The finest grid has a grid spacing h and the coarsest $8h$. A single coarse-grid correction is computed for all grids but the coarsest.

Figure 5.8:: W-cycle with ν_1 pre-smoothing steps and ν_2 post-smoothing steps. On each grid except the coarsest, the number of coarse-grid corrections is twice that of the underlying finer grid.

One reason why multigrid methods may fail to reach convergence is strong anisotropy in the coefficients of the governing partial differential equation or severely stretched grids (which has the same effect as anisotropy). In that case, more sophisticated smoothers or coarsening strategies may be required. Two strategies are currently implemented, *semicoarsening* and *line relaxation*, which can be used on their own or combined. Semicoarsening is when the grid is only coarsened in some directions. Line relaxation is when in some directions the whole gridlines of values are found simultaneously. If slow convergence is caused by just a few components of the solution, a Krylov subspace method can be used to remove them. In this way, multigrid is accelerated by a Krylov method. Alternatively, multigrid might be viewed as a preconditioner for a Krylov method.

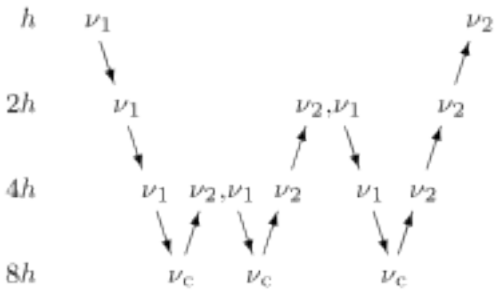

Figure 5.9:: F-cycle with ν_1 pre-smoothing steps and ν_2 post-smoothing steps. On each grid except the coarsest, the number of coarse-grid corrections increases by one compared to the underlying finer grid.

Gauss-Seidel

The smoother implemented in `emg3d` is a Gauss-Seidel smoother. The Gauss-Seidel method solves the linear equation system $A\mathbf{x} = \mathbf{b}$ iteratively using the following method:

$$\mathbf{x}^{(k+1)} = L_*^{-1}\left(\mathbf{b} - U\mathbf{x}^{(k)}\right) \ , \tag{5.31}$$

where L_* is the lower triangular component, and U the strictly upper triangular component, $A = L_* + U$. On the coarsest grid it acts as direct solver, whereas on the finer grid it acts as a smoother with only few iterations.

See the function *emg3d.solver.smoothing()*, and for more details, the functions *emg3d.core.gauss_seidel()*, *emg3d.core.gauss_seidel_x()*, *emg3d.core.gauss_seidel_y()*, *emg3d.core.gauss_seidel_z()*, and also *emg3d.core.blocks_to_amat()*.

Choleski factorisation

The actual solver of the system $A\mathbf{x} = \mathbf{b}$ is a non-standard Cholesky factorisation without pivoting for a symmetric, complex matrix A tailored to the problem of the multigrid solver, using only the main diagonal and five lower off-diagonals of the banded matrix A. The result is the same as simply using, e.g., `numpy.linalg.solve()`, but faster for the particular use-case of this code.

See *emg3d.core.solve()* for more details.

5.4 CPU & RAM

The multigrid method is attractive because it shows optimal scaling for both runtime and memory consumption. In the following are a few notes regarding memory and runtime requirements. It also contains information about what has been tried and what still could be tried in order to improve the current code.

5.4.1 Runtime

The gallery contains a script to do some testing with regards to runtime, see the Tools Section. An example output of that script is shown in Figure 5.10.

Figure 5.10:: Runtime as a function of cell size, which shows nicely the linear scaling of multigrid solvers (using a single thread).

The costliest functions (for big models) are:

- >90 %: *emg3d.solver.smoothing()* (*emg3d.core.gauss_seidel()*)
- <5 % each, in decreasing importance:

- *emg3d.solver.prolongation()* (*emg3d.solver.RegularGridProlongator*)
- *emg3d.solver.residual()* (*emg3d.core.amat_x()*)
- *emg3d.solver.restriction()*

Example with 262,144 / 2,097,152 cells (`nu_{i,1,c,2}=0,2,1,2`; `sslsolver=False`; `semicoarsening=True`; `linerelaxation=True`):

- 93.7 / 95.8 % `smoothing`
- 3.6 / 2.0 % `prolongation`
- 1.9 / 1.9 % `residual`
- 0.6 / 0.4 % `restriction`

The rest can be ignored. For small models, the percentage of `smoothing` goes down and of `prolongation` and `restriction` go up. But then the modeller is fast anyway.

emg3d.core.gauss_seidel() and *emg3d.core.amat_x()* are written in `numba`; jitting *emg3d.solver.RegularGridProlongator* turned out to not improve things, and many functions used in the restriction are jitted too. The costliest functions (RAM- and CPU-wise) are therefore already written in `numba`.

Any serious attempt to improve the speed will have to tackle the smoothing itself.

Things which could be tried

- Not much has been tested with the `numba`-options `parallel`; `prange`; and `nogil`.
- There might be an additional gain by making *emg3d.meshes.TensorMesh*, *emg3d.models.Model*, and *emg3d.fields.Field* instances jitted classes.

Things which have been tried

- One important aspect of the smoothing part is the memory layout. *emg3d.core.gauss_seidel()* and *emg3d.core.gauss_seidel_x()* are ideal for F-arrays (loop z-y-x, hence slowest to fastest axis). *emg3d.core.gauss_seidel_y()* and *emg3d.core.gauss_seidel_z()*, however, would be optimal for C-arrays. But copying the arrays to C-order and afterwards back is costlier in most cases for both CPU and RAM. The one possible and therefore implemented solution was to swap the loop-order in *emg3d.core.gauss_seidel_y()*.
- Restriction and prolongation information could be saved in a dictionary instead of recalculating it every time. Turns out to be not worth the trouble.
- Rewrite `emg3d.RegularGridInterpolator` as jitted function, but the iterator approach seems to be better for large grids.

5.4.2 Memory

Most of the memory requirement comes from storing the data itself, mainly the fields (source field, electric field, and residual field) and the model parameters (resistivity, eta, mu). For a big model, they some up; e.g., almost 3 GB for an isotropic model with 256x256x256 cells.

The gallery contains a script to do some testing with regards to the RAM usage, see the Tools Section. An example output of that script is shown in Figure 5.11.

Figure 5.11:: RAM usage, showing the optimal behaviour of multigrid methods. "Data RAM" is the memory required by the fields (source field, electric field, residual field) and by the model parameters (resistivity; and eta, mu). "MG Base" is for solving one Gauss-Seidel iteration on the original grid. "MG full RAM" is for solving one multigrid F-Cycle.

The theory of multigrid says that in an ideal scenario, multigrid requires 8/7 (a bit over 1.14) the memory requirement of carrying out one Gauss-Seidel step on the finest grid. As can be seen in the figure, for models up to 2 million cells that holds pretty much, afterwards it becomes a bit worse.

However, for this estimation one has to run the model first. Another way to estimate the requirement is by starting from the RAM used to store the fields and parameters. As can be seen in the figure, for big models one is on the save side estimating the required RAM as 1.35 times the storage required for the fields and model parameters.

The figure also shows nicely the linear behaviour of multigrid; for twice the number of cells twice the memory is required (from a certain size onwards).

Attempts at improving memory usage should focus on the difference between the red line (actual usage) and the dashed black line (1.14 x base usage).

5.5 Gallery

The gallery with many examples can be found at empymod.github.io/emg3d-gallery.

5.6 References

5.7 Credits

This project was started by **Dieter Werthmüller**. Every contributor will be listed here and is considered to be part of «The emg3d Developers»:

- Dieter Werthmüller

Various bits got improved through discussions on Slack at SWUNG and at SimPEG, thanks to both communities. Special thanks to @jokva (general), @banesullivan (visualization), @joferkington (interpolation), and @jcapriot (volume averaging).

5.7.1 Historical credits

The core of *emg3d* is a complete rewrite and redesign of the multigrid code by **Wim A. Mulder** ([Muld06], [Muld07], [Muld08], [MuWS08]), developed at Shell and at TU Delft. Various authors contributed to the original code, amongst others, **Tom Jönsthövel** ([JoOM06]; improved solver for strongly stretched grids), **Marwan Wirianto** ([WiMS10], [WiMS11]; computation of time-domain data), and **Evert C. Slob** ([SlHM10]; analytical solutions). The original code was written in Matlab, where the most time- and memory-consuming parts were sped up through mex-files (written in C). It included multigrid with or without BiCGSTAB, VTI resistivity, semi-coarsening, and line relaxation; the number of cells had to be powers of two, and coarsening was done only until the first dimension was at two cells. As such it corresponded roughly to *emg3d v0.3.0*.

See the *References* in the manual for the full citations and a more extensive list.

Note: This software was initially (till 05/2021) developed at *Delft University of Technology* (https://www.tudelft.nl) within the **Gitaro.JIM** project funded through MarTERA as part of Horizon 2020, a funding scheme of the European Research Area (ERA-NET Cofund, https://www.martera.eu).

5.8 Changelog

5.8.1 *v0.11.0* : Refactor

2020-05-05

Grand refactor with new internal layout. Mainly splitting-up *utils* into smaller bits. Most functionalities (old names) are currently retained in *utils* and it should be mostly backwards compatible for now, but they are deprecated and will eventually be removed. Some previously deprecated functions were removed, however.

- Removed deprecated functions:
  - *emg3d.solver.solver* (use *emg3d.solver.solve* instead).
  - Aliases of *emg3d.io.data_write* and *emg3d.io.data_read* in *emg3d.utils*.
- Changes:
  - *SourceField* has now the same signature as *Field* (this might break your code if you called *SourceField* directly, with positional arguments, and not through *get_source_field*).
  - More functions and classes in the top namespace.
  - Replaced *core.l2norm* with *scipy.linalg.norm*, as SciPy 1.4 got the following PR: https://github.com/scipy/scipy/pull/10397 (reason to raise minimum SciPy to 1.4).
  - Increased minimum required versions of dependencies to
    - *scipy>=1.4.0* (raised from 1.1, see note above)
    - *empymod>=2.0.0* (no min requirement before)
    - *numba>=0.45.0* (raised from 0.40)
- New layout
  - *njitted -> core*.
  - *utils* split in *fields*, *meshes*, *models*, *maps*, and *utils*.
- Bugfixes:
  - Fixed *to_dict*, *from_dict*, and *copy* for the *SourceField*.
  - Fixed *io* for *SourceField*, that was not implemented properly.

5.8.2 *v0.10.1* : Zero Source

2020-04-29

- Bug fixes:
  - Checks now if provided source-field is zero, and exists gracefully if so, returning a zero electric field. Until now it failed with a division-by-zero error.
- Improvements:
  - Warnings: If `verb=1` it prints a warning in case it did not converge (it finished silently until now).
  - Improvements to docs (figures-scaling; intersphinx).
  - Adjust `Fields.pha` and `Fields.amp` in accordance with `empymod v2`: `.pha` and `.amp` are now methods; uses directly `empymod.utils.EMArray`.
  - Adjust tests for `empymod v2` (Fields, Fourier).

5.8.3 *v0.10.0* : Data persistence

2020-03-25

- New:
  - New functions `emg3d.save` and `emg3d.load` to save and load all sort of `emg3d` instances. The currently implemented backends are `h5py` for `.h5`-files (default, but requires `h5py` to be installed) and `numpy` for `.npz`-files.
  - Classes `emg3d.utils.Field`, `emg3d.utils.Model`, and `emg3d.utils.TensorMesh` have new methods `.copy()`, `.to_dict()`, and `.from_dict()`.

  - `emg3d.utils.Model`: Possible to create new models by adding or subtracting existing models, and comparing two models (+, −, == and !=). New attributes `shape` and `size`.
  - `emg3d.utils.Model` does not store the volume any longer (just `vnC`).
- Deprecations:
  - Deprecated `data_write` and `data_read`.
- Internal and bug fixes:
  - All I/O-related stuff moved to its own file *io.py*.
  - Change from `NUMBA_DISABLE_JIT` to use `py_func` for testing and coverage.
  - Bugfix: `emg3d.njitted.restrict` did not store the {x;y;z}-field if `sc_dir` was {4;5;6}, respectively.

5.8.4 *v0.9.3* : Sphinx gallery

2020-02-11

- Rename `solver.solver` to `solver.solve`; load `solve` also into the main namespace as `emg3d.solve`.
- Adjustment to termination criterion for *STAGNATION*: The current error is now compared to the last error of the same cycle type. Together with this the workaround for sslsolver when called with an initial efield introduced in v0.8.0 was removed.
- Adjustment to `utils.get_hx_h0` (this might change your boundaries): The calculation domain is now calculated so that the distance for the signal travelling from the source to the boundary and back to the most remote receiver is at least two wavelengths away. If this is within the provided domain, then now extra buffer is added around the domain. Additionally, the function has a new parameter `max_domain`, which is the maximum distance from the center to the boundary; defaults to 100 km.
- New parameter `log` for `utils.grid2grid`; if `True`, then the interpolation is carried out on a log10-scale.
- Change from the notebook-based `emg3d-examples`-repo to the `sphinx`-based `emg3d-gallery`-repo.

5.8.5 *v0.9.2* : Complex sources

2019-12-26

- Strength input for `get_source_field` can now be complex; it also stores now the source location and its strength and moment.
- `get_receiver` can now take entire `Field` instances, and returns in that case (`fx`, `fy`, `fz`) at receiver locations.
- Krylov subspace solvers:
  - Solver now finishes in the middle of preconditioning cycles if tolerance is reached.
  - Solver now aborts if solution diverges or stagnates also for the SSL solvers; it fails and returns a zero field.
  - Removed `gmres` and `lgmres` from the supported SSL solvers; they do not work nice for this problem. Supported remain `bicgstab` (default), `cgs`, and `gcrotmk`.
- Various small things:
  - New attribute `Field.is_electric`, so the field knows if it is electric or magnetic.
  - New `verb`-possibility: `verb=-1` is a continuously updated one-liner, ideal to monitor large sets of calculations or in inversions.

  - The returned `info` dictionary contains new keys:
    * `runtime_at_cycle`: accumulated total runtime at each cycle;
    * `error_at_cycle`: absolute error at each cycle.
  - Simple `__repr__` for `TensorMesh`, `Model`, `Fourier`, `Time`.
- Bugfixes:
  - Related to `get_hx_h0`, `data_write`, printing in `Fourier`.

5.8.6 *v0.9.1* : VolumeModel

2019-11-13

- New class `VolumeModel`; changes in `Model`:
  - `Model` now only contains resistivity, magnetic permeability, and electric permittivity.
  - `VolumeModel` contains the volume-averaged values eta and zeta; called from within `emg3d.solver.solver`.
  - Full wave equation is enabled again, via `epsilon_r`; by default it is set to None, hence diffusive approximation.
  - Model parameters are now internally stored as 1D arrays.
  - An {isotropic, VTI, HTI} initiated model can be changed by providing the missing resistivities.
- Bugfix: Up and till version 0.8.1 there was a bug. If resistivity was set with slices, e.g., `model.res[:, :, :5]=1e10`, it DID NOT update the corresponding eta. This bug was unintentionally fixed in 0.9.0, but only realised now.
- Various:
  - The log now lists the version of emg3d.
  - PEP8: internal imports now use absolute paths instead of relative ones.
  - Move from conda-channel `prisae` to `conda-forge`.
  - Automatic deploy for PyPi and conda-forge.

5.8.7 *v0.9.0* : Fourier

2019-11-07

- New routine:
  - `emg3d.utils.Fourier`, a class to handle Fourier-transform related stuff for time-domain modelling. See the example notebooks for its usage.
- Utilities:
  - `Fields` and returned receiver-arrays (`EMArray`) both have amplitude (`.amp`) and phase (`.pha`) attributes.
  - `Fields` have attributes containing frequency-information (`freq`, `smu0`).
  - New class `SourceField`; a subclass of `Field`, adding `vector` and `v{x,y,z}` attributes for the real valued source vectors.
  - The `Model` is not frequency-dependent any longer and does NOT take a `freq`-parameter any more (currently it still takes it, but it is deprecated and will be removed in the future).
  - `data_write` automatically removes `_vol` from `TensorMesh` instances and `_eta_{x,y,z}`, `_zeta` from `Model` instances. This makes the archives smaller, and they are not required, as they are simply reconstructed if needed.

- Internal changes:
  - The multigrid method, as implemented, only works for the diffusive approximation. Nevertheless, we always used `\sigma-i\omega\epsilon`, hence a complex number. This is now changed and `\epsilon` set to 0, leaving only `\sigma`.
  - Change time convention from `exp(-iwt)` to `exp(iwt)`, as used in `empymod` and commonly in CSEM. Removed the parameter `conjugate` from the solver, to simplify.
  - Change own private class variables from ___ to _.
  - `res` and `mu_r` are now checked to ensure they are >0; `freq` is checked to ensure !=0.
- New dependencies and maintenance:
  - `empymod` is a new dependency.
  - Travis now checks all the url's in the documentation, so there should be no broken links down the road. (Check is allowed to fail, it is visual QC.)
- Bugfixes:
  - Fixes to the `setuptools_scm`-implementation (`MANIFEST.in`).

5.8.8 *v0.8.1* : setuptools_scm

2019-10-22

- Implement `setuptools_scm` for versioning (adds git hashes for dev-versions).

5.8.9 *v0.8.0* : Laplace

2019-10-04

- Laplace-domain calculation: By providing a negative `freq`-value to `utils.get_source_field` and `utils.Model`, the calculation is carried out in the real Laplace domain `s = freq` instead of the complex frequency domain `s = 2i*pi*freq`.
- New meshing helper routines (particularly useful for transient modelling where frequency-dependent/adaptive meshes are inevitable):
  - `utils.get_hx_h0` to get cell widths and origin for given parameters including a few fixed interfaces (center plus two, e.g. top anomaly, sea-floor, and sea-surface).
  - `utils.get_cell_numbers` to get good values of number of cells for given primes.
- Speed-up `njitted.volume_average` significantly thanks to @jcapriot.
- Bugfixes and other minor things:
  - Abort if l2-norm is NaN (only works for MG).
  - Workaround for the case where a `sslsolver` is used together with a provided initial `efield`.
  - Changed parameter `rho` to `res` for consistency reasons in `utils.get_domain`.
  - Changed parameter `h_min` to `min_width` for consistency reasons in `utils.get_stretched_h`.

5.8.10 *v0.7.1* : JOSS article

2019-07-17

- Version of the JOSS article, https://doi.org/10.21105/joss.01463 .

- New function `utils.grid2grid` to move from one grid to another. Both functions (`utils.get_receiver` and `utils.grid2grid`) can be used for fields and model parameters (with or without extrapolation). They are very similar, the former taking coordinates (x, y, z) as new points, the latter one another TensorMesh instance.
- New jitted function `njitted.volume_average` for interpolation using the volume-average technique.
- New parameter `conjugate` in `solver.solver` to permit both Fourier transform conventions.
- Added `exit_status` and `exit_message` to `info_dict`.
- Add section `Related ecosystem` to documentation.

5.8.11 *v0.7.0* : H-field

2019-07-05

- New routines:
  - `utils.get_h_field`: Small routine to calculate the magnetic field from the electric field using Faraday's law.
  - `utils.get_receiver`: Small wrapper to interpolate a field at receiver positions. Added 3D spline interpolation; is the new default.
- Re-implemented the possibility to define isotropic magnetic permeabilities in `utils.Model`. Magnetic permeability is not tri-axially included in the solver currently; however, it would not be too difficult to include if there is a need.
- CPU-graph added on top of RAM-graph.
- Expand `utils.Field` to work with pickle/shelve.
- Jit `np.linalg.norm` (`njitted.l2norm`).
- Use `scooby` (soft dependency) for versioning, rename `Version` to `Report` (backwards incompatible).
- Bug fixes:
  - Small bugfix introduced in ebd2c9d5: `sc_cycle` and `lr_cycle` was not updated any longer at the end of a cycle (only affected `sslsolver=True`.
  - Small bugfix in `utils.get_hx`.

5.8.12 *v0.6.2* : CPU & RAM

2019-06-03

Further speed and memory improvements:

- Add *CPU & RAM*-page to documentation.
- Change loop-order from x-z-y to z-x-y in Gauss-Seidel smoothing with line relaxation in y-direction. Hence reversed lexicographical order. This results in a significant speed-up, as x is the fastest changing axis.
- Move total residual calculation from `solver.residual` into `njitted.amat_x`.
- Simplifications in `utils`:
  - Simplify `utils.get_source_field`.
  - Simplify `utils.Model`.
  - Removed unused timing-stuff from early development.

5.8.13 *v0.6.1* : Memory

2019-05-28

Memory and speed improvements:

- Only calculate residual and l2-norm when absolutely necessary.
- Inplace calculations for `np.conjugate` in `solver.solver` and `np.subtract` in `solver.residual`.

5.8.14 *v0.6.0* : RegularGridInterpolator

2019-05-26

- Replace `scipy.interpolate.RegularGridInterpolator` with a custom tailored version of it (*solver.RegularGridProlongator*); results in twice as fast prolongation.
- Simplify the fine-grid calculation in `prolongation` without using `gridE*`; memory friendlier.
- Submission to JOSS.
- Add *Multi-what?*-page to documentation.
- Some major refactoring, particularly in `solver`.
- Removed `discretize` as hard dependency.
- Rename `rdir` and `ldir` (and related `p*dir`; `*cycle`) to the more descriptive `sc_dir` and `lr_dir`.

5.8.15 v0.5.0 : Accept any grid size

2019-05-01

- First open-source version.
- Include RTD, Travis, Coveralls, Codacy, and Zenodo. No benchmarks yet.
- Accepts now *any* grid size (warns if a bad grid size for MG is provided).
- Coarsens now to the lowest level of each dimension, not only to the coarsest level of the smallest dimension.
- Combined `restrict_rx`, `restrict_ry`, and `restrict_rz` to `restrict`.
- Improve speed by passing pre-allocated arrays to jitted functions.
- Store `res_y`, `res_z` and corresponding `eta_y`, `eta_z` only if `res_y`, `res_z` were provided in initial call to `utils.model`.
- Change `zeta` to `v_mu_r`.
- Include rudimentary `TensorMesh`-class in `utils`; removes hard dependency on `discretize`.
- Bugfix: Take a provided `efield` into account; don't return if provided.

5.8.16 v0.4.0 : Cholesky

2019-03-29

- Use `solve_chol` for everything, remove `solve_zlin`.
- Moved `mesh.py` and some functionalities from `solver.py` into `utils.py`.
- New mesh-tools. Should move to `discretize` eventually.
- Improved source generation tool. Might also move to `discretize`.
- `printversion` is now included in `utils`.

- Many bug fixes.
- Lots of improvements to tests.
- Lots of improvements to documentation. Amongst other, moved docs from `__init__.py` into the docs rst.

5.8.17 v0.3.0 : Semicoarsening

2019-01-18

- Semicoarsening option.
- Number of cells must still be 2^n, but n can be different in the x-, y-, and z-directions.
- Many other iterative solvers from `scipy.sparse.linalg` can be used. It seems to work fine with the following methods:
  - `scipy.sparse.linalg.bicgstab()`: BIConjugate Gradient STABilize;
  - `scipy.sparse.linalg.cgs()`: Conjugate Gradient Squared;
  - `scipy.sparse.linalg.gmres()`: Generalized Minimal RESidual;
  - `scipy.sparse.linalg.lgmres()`: Improvement of GMRES using alternating residual vectors;
  - `scipy.sparse.linalg.gcrotmk()`: GCROT: Generalized Conjugate Residual with inner Orthogonalization and Outer Truncation.
- The SciPy-solver or MG can be used all in combination or on its own, hence only MG, SciPy-solver with MG preconditioning, only SciPy-solver.

5.8.18 v0.2.0 : Line relaxation

2019-01-14

- Line relaxation option.

5.8.19 v0.1.0 : Initial

2018-12-28

- Standard multigrid with or without BiCGSTAB.
- Tri-axial anisotropy.
- Number of cells must be 2^n, and n has to be the same in the x-, y-, and z-directions.

5.9 Maintainers Guide

5.9.1 Making a release

1. Update `CHANGELOG.rst`.
2. Push it to GitHub, create a release tagging it.
3. Tagging it on GitHub will automatically deploy it to PyPi, which in turn will create a PR for the conda-forge feedstock. Merge that PR.
4. Check that:

- PyPi deployed;

- conda-forge deployed;
- Zenodo minted a DOI;
- emg3d.rtfd.io created a tagged version.

5.9.2 Useful things

- If there were changes to README, check it with:

```
python setup.py --long-description | rst2html.py --no-raw > index.html
```

- If unsure, test it first on testpypi (requires ~/.pypirc):

```
~/anaconda3/bin/twine upload dist/* -r testpypi
```

- If unsure, test the test-pypi for conda if the skeleton builds:

```
conda skeleton pypi --pypi-url https://test.pypi.io/pypi/ emg3d
```

- If it fails, you might have to install `python3-setuptools`:

```
sudo apt install python3-setuptools
```

5.9.3 CI

Automatic bits

- Testing on Travis, includes:
  - Tests using `pytest`
  - Linting / code style with `pytest-flake8`
  - Ensure all http(s)-links work (`sphinx linkcheck`)
- Line-coverage with `pytest-cov` on Coveralls
- Code-quality on Codacy
- Manual on ReadTheDocs
- DOI minting on Zenodo

Manual things

- Benchmarks with Airspeed Velocity (`asv`)
- Gallery in emg3d-gallery (`sphinx-gallery`)

Automatically deploys if tagged

- PyPi
- conda -c conda-forge

5.10 Main solver routine

emg3d.solver.**solve**(*grid*, *model*, *sfield*, *efield=None*, *cycle='F'*, *sslsolver=False*, *semicoarsening=False*, *linerelaxation=False*, *verb=2*, ***kwargs*)

Solver for 3D CSEM data with tri-axial electrical anisotropy.

The principal solver of *emg3d* is using the multigrid method as presented in [Muld06]. Multigrid can be used as a standalone solver, or as a preconditioner for an iterative solver from the `scipy.sparse.linalg`-library, e.g., `scipy.sparse.linalg.bicgstab()`. Alternatively, these Krylov subspace solvers can also be used without multigrid at all. See the *cycle* and *sslsolver* parameters.

Implemented are the *F*-, *V*-, and *W*-cycle schemes for multigrid (*cycle* parameter), and the amount of smoothing steps (initial smoothing, pre-smoothing, coarsest-grid smoothing, and post-smoothing) can be set individually (*nu_init*, *nu_pre*, *nu_coarse*, and *nu_post*, respectively). The maximum level of coarsening can be restricted with the *clevel* parameter.

Semicoarsening and line relaxation, as presented in [Muld07], are implemented, see the *semicoarsening* and *linerelaxation* parameters. Using the BiCGSTAB solver together with multigrid preconditioning with semicoarsening and line relaxation is slow but generally the most robust. Not using BiCGSTAB nor semicoarsening nor line relaxation is fast but may fail on stretched grids.

Parameters

grid [*emg3d.meshes.TensorMesh*] The grid. See *emg3d.meshes.TensorMesh*.

model [*emg3d.models.Model*] The model. See *emg3d.models.Model*.

sfield [*emg3d.fields.SourceField*] The source field. See *emg3d.fields.get_source_field()*.

efield [*emg3d.fields.Field*, optional] Initial electric field. It is initiated with zeroes if not provided. A provided efield MUST have frequency information (initiated with `emg3d.fields.Field(..., freq)`).

If an initial efield is provided nothing is returned, but the final efield is directly put into the provided efield.

If an initial field is provided and a sslsolver is used, then it first carries out one multigrid cycle without semicoarsening nor line relaxation. The sslsolver is at times unstable with an initial guess, carrying out one MG cycle helps to stabilize it.

cycle [str; optional.] Type of multigrid cycle. Default is 'F'.

- 'V': V-cycle, simplest version;
- 'W': W-cycle, most expensive version;
- 'F': F-cycle, sort of a compromise between 'V' and 'W';
- None: Does not use multigrid, only *sslsolver*.

If None, *sslsolver* must be provided, and the *sslsolver* will be used without multigrid pre-conditioning.

Comparison of V (left), F (middle), and W (right) cycles for the case of four grids (three relaxation and prolongation steps):

```
 h_
2h_   \    /   \          /   \            /
4h_    \  /     \    /\  /     \    /\    /
8h_     \/       \/\/  \/       \/\/  \/\/
```

sslsolver [str, optional] A `scipy.sparse.linalg`-solver, to use with MG as preconditioner or on its own (if `cycle=None`). Default is False.

Current possibilities:

- True or 'bicgstab': BIConjugate Gradient STABilized `scipy.sparse.linalg.bicgstab()`;
- 'cgs': Conjugate Gradient Squared `scipy.sparse.linalg.cgs()`;
- 'gcrotmk': GCROT: Generalized Conjugate Residual with inner Orthogonalization and Outer Truncation `scipy.sparse.linalg.gcrotmk()`.

It does currently not work with 'cg', 'bicg', 'qmr', and 'minres' for various reasons (e.g., some require *rmatvec* in addition to *matvec*).

semicoarsening [int; optional] Semicoarsening. Default is False.

- True: Cycling over 1, 2, 3.
- 0 or False: No semicoarsening.
- 1: Semicoarsening in x direction.
- 2: Semicoarsening in y direction.
- 3: Semicoarsening in z direction.
- Multi-digit number containing digits from 0 to 3. Multigrid will cycle over these values, e.g., `semicoarsening=1213` will cycle over [1, 2, 1, 3].

linerelaxation [int; optional] Line relaxation. Default is False.

This parameter is not respected on the coarsest grid, except if it is set to 0. If it is bigger than zero line relaxation on the coarsest grid is carried out along all dimensions which have more than 2 cells.

- True: Cycling over [4, 5, 6].
- 0 or False: No line relaxation.
- 1: line relaxation in x direction.
- 2: line relaxation in y direction.
- 3: line relaxation in z direction.
- 4: line relaxation in y and z directions.
- 5: line relaxation in x and z directions.
- 6: line relaxation in x and y directions.
- 7: line relaxation in x, y, and z directions.
- Multi-digit number containing digits from 0 to 7. Multigrid will cycle over these values, e.g., `linerelaxation=1213` will cycle over [1, 2, 1, 3].

Note: Smoothing is generally done in lexicographical order, except for line relaxation in y direction; the reason is speed (memory access).

verb [int; optional] Level of verbosity (the higher the more verbose). Default is 2.

- 0: Print nothing.
- 1: Print warnings.
- 2: Print runtime and information about the method.
- 3: Print additional information for each MG-cycle.
- 4: Print everything (slower due to additional error calculations).
- -1: Print one-liner (dynamically updated).

****kwargs** [Optional solver options:]

- *tol* : float

  Convergence tolerance. Default is 1e-6.

  Iterations stop as soon as the norm of the residual has decreased by this factor, relative to the residual norm obtained for a zero electric field.

- *maxit* : int

  Maximum number of multigrid iterations. Default is 50.

  If *sslsolver* is used, this applies to the *sslsolver*.

  In the case that multigrid is used as a pre-conditioner for the *sslsolver*, the maximum iteration for multigrid is defined by the maximum length of the *linerelaxation* and *semicoarsening*-cycles.

- *nu_init* : int

  Number of initial smoothing steps, before MG cycle. Default is 0.

- *nu_pre* : int

  Number of pre-smoothing steps. Default is 2.

- *nu_coarse* : int

  Number of smoothing steps on coarsest grid. Default is 1.

- *nu_post* : int

  Number of post-smoothing steps. Default is 2.

- *clevel* : int

  The maximum coarsening level can be different for each dimension and is, by default, automatically determined (`clevel=-1`). The parameter *clevel* can be used to restrict the maximum coarsening level in any direction by its value. Default is -1.

- *return_info* : bool

  If True, a dictionary is returned with runtime info (final norm and number of iterations of MG and the sslsolver).

Returns

efield [*emg3d.fields.Field*] Resulting electric field. Is not returned but replaced in-place if an initial efield was provided.

info_dict [dict] Dictionary with runtime info; only if `return_info=True`.

Keys:

- *exit*: Exit status, 0=Success, 1=Failure;
- *exit_message*: Exit message, check this if `exit=1`;
- *abs_error*: Absolute error;
- *rel_error*: Relative error;
- *ref_error*: Reference error [norm(sfield)];
- *tol*: Tolerance (abs_error<ref_error*tol);
- *it_mg*: Number of multigrid iterations;
- *it_ssl*: Number of SSL iterations;
- *time*: Runtime (s).
- *runtime_at_cycle*: Runtime after each cycle (s).
- *error_at_cycle*: Absolute error after each cycle.

Examples

```
>>> import emg3d
>>> import numpy as np
>>> # Create a simple grid, 8 cells of length 1 in each direction,
>>> # starting at the origin.
>>> grid = emg3d.meshes.TensorMesh(
>>>         [np.ones(8), np.ones(8), np.ones(8)],
>>>         x0=np.array([0, 0, 0]))
>>> # The model is a fullspace with tri-axial anisotropy.
>>> model = emg3d.models.Model(grid, res_x=1.5, res_y=1.8, res_z=3.3)
>>> # The source is a x-directed, horizontal dipole at (4, 4, 4)
>>> # with a frequency of 10 Hz.
>>> sfield = emg3d.fields.get_source_field(
>>>         grid, src=[4, 4, 4, 0, 0], freq=10)
>>> # Calculate the electric signal.
>>> efield = emg3d.solve(grid, model, sfield, verb=3)
>>> # Get the corresponding magnetic signal.
>>> hfield = emg3d.fields.get_h_field(grid, model, efield)
.
:: emg3d START :: 10:27:25 :: v0.9.1
.
   MG-cycle       : 'F'                 sslsolver : False
   semicoarsening : False [0]           tol       : 1e-06
   linerelaxation : False [0]           maxit     : 50
   nu_{i,1,c,2}   : 0, 2, 1, 2          verb      : 3
   Original grid  :   8 x   8 x   8     => 512 cells
   Coarsest grid  :   2 x   2 x   2     => 8 cells
   Coarsest level :   2 ;   2 ;   2
.
   [hh:mm:ss]  rel. error                  [abs. error, last/prev]   l s
.
       h_
      2h_ \    /
      4h_  \/\/
.
   [10:27:25]   2.284e-02  after   1 F-cycles   [1.275e-06, 0.023]   0 0
   [10:27:25]   1.565e-03  after   2 F-cycles   [8.739e-08, 0.069]   0 0
   [10:27:25]   1.295e-04  after   3 F-cycles   [7.232e-09, 0.083]   0 0
   [10:27:25]   1.197e-05  after   4 F-cycles   [6.685e-10, 0.092]   0 0
   [10:27:25]   1.233e-06  after   5 F-cycles   [6.886e-11, 0.103]   0 0
   [10:27:25]   1.415e-07  after   6 F-cycles   [7.899e-12, 0.115]   0 0
.
   > CONVERGED
   > MG cycles        : 6
   > Final rel. error : 1.415e-07
.
:: emg3d END   :: 10:27:25 :: runtime = 0:00:00
```

5.11 Code

Electromagnetic modeller in the diffusive limit (low frequencies) for 3D media with tri-axial electrical anisotropy. The matrix-free multigrid solver can be used as main solver or as preconditioner for one of the Krylov subspace methods implemented in `scipy.sparse.linalg`, and the governing equations are discretized on a staggered Yee grid. The code is written completely in Python using the `numpy`/`scipy`-stack, where the most time-consuming parts are sped-up through jitted `numba`-functions.

5.11.1 `solver` – Multigrid solver

The actual solver routines. The most computationally intensive parts, however, are in the *emg3d.core* as numba-jitted functions.

emg3d.solver.**multigrid**(*grid*, *model*, *sfield*, *efield*, *var*, ***kwargs*)
Multigrid solver for 3D controlled-source electromagnetic (CSEM) data.

Multigrid solver as presented in [Muld06], including semicoarsening and line relaxation as presented in and [Muld07].

- The electric field is stored in-place in *efield*.
- The number of multigrid cycles is stored in *var.it*.
- The current error (l2-norm) is stored in *var.l2*.
- The reference error (l2-norm of sfield) is stored in *var.l2_refe*.

This function is called by *solve()*.

Parameters

grid [*emg3d.meshes.TensorMesh*] The grid. See *emg3d.meshes.TensorMesh*.

model [*emg3d.models.VolumeModel*] The Model. See *emg3d.models.VolumeModel*.

sfield [*emg3d.fields.SourceField*] The source field. See *emg3d.fields.get_source_field()*.

efield [*emg3d.fields.Field*] The electric field. See *emg3d.fields.Field*.

var [*MGParameters* instance] As returned by *multigrid()*.

****kwargs** [Recursion parameters.] Do not use; only used internally by recursion; *level* (current coarsening level) and *new_cycmax* (new maximum of MG cycles, takes care of V/W/F-cycling).

emg3d.solver.**smoothing**(*grid*, *model*, *sfield*, *efield*, *nu*, *lr_dir*)
Reducing high-frequency error by smoothing.

Solves the linear equation system $Ax = b$ iteratively using the Gauss-Seidel method. This acts as smoother or, on the coarsest grid, as a direct solver.

This is a simple wrapper for the jitted calculation in *emg3d.core.gauss_seidel()*, *emg3d.core.gauss_seidel_x()*, *emg3d.core.gauss_seidel_y()*, and *emg3d.core.gauss_seidel_z()* (*@njit* can not [yet] access class attributes). See these functions for more details and corresponding theory.

The electric fields are updated in-place.

This function is called by *multigrid()*.

Parameters

grid [*emg3d.meshes.TensorMesh*] Input grid.

model [*emg3d.models.VolumeModel*] Input model.

sfield [*emg3d.fields.SourceField*] Input source field.

efield [*emg3d.fields.Field*] Input electric field.

nu [int] Number of Gauss-Seidel steps; odd numbers are forward, even numbers are reversed. E.g., `nu=2` is one symmetric Gauss-Seidel iteration, with a forward and a backward step.

lr_dir [int] Direction of line relaxation {0, 1, 2, 3, 4, 5, 6, 7}.

emg3d.solver.**restriction**(*grid*, *model*, *sfield*, *residual*, *sc_dir*)
Downsampling of grid, model, and fields to a coarser grid.

The restriction of the residual is used as source term for the coarse grid.

Corresponds to Equations 8 and 9 in [Muld06] and surrounding text. In the case of the restriction of the residual, this function is a wrapper for the jitted functions *emg3d.core.restrict_weights()* and *emg3d.core.restrict()* (@*njit* can not [yet] access class attributes). See these functions for more details and corresponding theory.

This function is called by *multigrid()*.

Parameters

grid [*emg3d.meshes.TensorMesh*] Input grid.

model [*emg3d.models.VolumeModel*] Input model.

sfield [*emg3d.fields.SourceField*] Input source field.

sc_dir [int] Direction of semicoarsening (0, 1, 2, or 3).

Returns

cgrid [*emg3d.meshes.TensorMesh*] Coarse grid.

cmodel [*emg3d.models.VolumeModel*] Coarse model.

csfield [*emg3d.fields.SourceField*] Coarse source field. Corresponds to restriction of fine-grid residual.

cefield [*emg3d.fields.Field*] Coarse electric field, complex zeroes.

emg3d.solver.**prolongation**(*grid*, *efield*, *cgrid*, *cefield*, *sc_dir*)
Interpolating the electric field from coarse grid to fine grid.

The prolongation from a coarser to a finer grid is the inverse process of the restriction (*restriction()*) from a finer to a coarser grid. The interpolated values of the coarse grid electric field are added to the fine grid electric field, in-place. Piecewise constant interpolation is used in the direction of the field, and bilinear interpolation in the other two directions.

See Equation 10 in [Muld06] and surrounding text.

This function is called by *multigrid()*.

Parameters

grid, cgrid [*emg3d.meshes.TensorMesh*] Fine and coarse grids.

efield, cefield [*emg3d.fields.Field*] Fine and coarse grid electric fields.

sc_dir [int] Direction of semicoarsening (0, 1, 2, or 3).

emg3d.solver.**residual**(*grid*, *model*, *sfield*, *efield*, *norm=False*)
Calculating the residual.

Returns the complete residual as given in [Muld06], page 636, middle of the right column:

$$\mathbf{r} = V\left(\mathrm{i}\omega\mu_0\mathbf{J_s} + \mathrm{i}\omega\mu_0\tilde{\sigma}\mathbf{E} - \nabla\times\mu_\mathrm{r}^{-1}\nabla\times\mathbf{E}\right).$$

This is a simple wrapper for the jitted calculation in *emg3d.core.amat_x()* (@*njit* can not [yet] access class attributes). See *emg3d.core.amat_x()* for more details and corresponding theory.

This function is called by *multigrid()*.

Parameters

grid [*emg3d.meshes.TensorMesh*] Input grid.

model [*emg3d.models.VolumeModel*] Input model.

sfield [*emg3d.fields.SourceField*] Input source field.

efield [*emg3d.fields.Field*] Input electric field.

norm [bool] If True, the error (l2-norm) of the residual is returned, not the residual.

Returns

residual [Field] Returned if `norm=False`. The residual field; *emg3d.fields.Field* instance.

norm [float] Returned if `norm=True`. The error (l2-norm) of the residual

emg3d.solver.**krylov**(*grid*, *model*, *sfield*, *efield*, *var*)

Krylov Subspace iterative solver for 3D CSEM data.

Using a Krylov subspace iterative solver (defined in *var.sslsolver*) implemented in SciPy with or without multigrid as a pre-conditioner ([Muld06]).

- The electric field is stored in-place in *efield*.
- The current error (l2-norm) is stored in *var.l2*.
- The reference error (l2-norm of sfield) is stored in *var.l2_refe*.

This function is called by *solve()*.

Parameters

grid [*emg3d.meshes.TensorMesh*] The grid. See *emg3d.meshes.TensorMesh*.

model [*emg3d.models.VolumeModel*] The Model. See *emg3d.models.VolumeModel*.

sfield [*emg3d.fields.SourceField*] The source field. See *emg3d.fields.get_source_field()*.

efield [*emg3d.fields.Field*] The electric field. See *emg3d.fields.Field*.

var [*MGParameters* instance] As returned by *multigrid()*.

class emg3d.solver.**MGParameters**(*verb: int*, *cycle: str*, *sslsolver: str*, *linerelaxation: int*, *semi-coarsening: int*, *vnC: tuple*, *tol: float = 1e-06*, *maxit: int = 50*, *nu_init: int = 0*, *nu_pre: int = 2*, *nu_coarse: int = 1*, *nu_post: int = 2*, *clevel: int = -1*, *return_info: bool = False*)

Collect multigrid solver settings.

This dataclass is used by the main *solve()*-routine. See *solve()* for a description of the mandatory and optional input parameters and more information .

Returns

var [class:*MGParameters*] As required by *multigrid()*.

cprint(*self*, *info*, *verbosity*, ***kwargs*)

Conditional printing.

Prints *info* if *self.verb* > *verbosity*.

Parameters

info [str] String to be printed.

verbosity [int] Verbosity of info.

kwargs [optional] Arguments passed to *print*.

max_level

Sets dimension-dependent level variable *clevel*.

Requires at least two cells in each direction (for *nCx*, *nCy*, and *nCz*).

one_liner(*self*, *l2_last*, *last=False*)

Print continuously updated one-liner.

Parameters

l2_last [float] Current error.

last [bool] If True, adds *exit_message* and finishes line.

class emg3d.solver.**RegularGridProlongator**(*x*, *y*, *cxy*)

Prolongate field from coarse to fine grid.

This is a heavily modified and adapted version of `scipy.interpolate.RegularGridInterpolator`.

The main difference (besides the pre-sets) is that this version allows to initiate an instance with the coarse and fine grids. This initialize will calculate the required weights, and it has therefore only to be done once.

After this, interpolating values from the coarse to the fine grid can be carried out much faster.

Simplifications in comparison to `scipy.interpolate.RegularGridInterpolator`:

- No sanity checks what-so-ever.
- Only 2D data;
- `method='linear'`;
- `bounds_error=False`;
- `fill_value=None`.

It results in a speed-up factor of about 2, independent of grid size, for this particular case. The prolongation is the second-most expensive part of multigrid after the smoothing. Trying to improve this further might therefore be useful.

Parameters

x, y [ndarray] The x/y-coordinates defining the coarse grid.

cxy [ndarray of shape (..., 2)] The ([[x], [y]]).T-coordinates defining the fine grid.

5.11.2 `core` – Number crunching

The core functionalities, the most computationally demanding parts, of the *emg3d.solver* as just-in-time (jit) compiled functions using `numba`.

emg3d.core.**amat_x**(*rx*, *ry*, *rz*, *ex*, *ey*, *ez*, *eta_x*, *eta_y*, *eta_z*, *zeta*, *hx*, *hy*, *hz*)

Residual without or with source term.

Calculate the residual as given in [Muld06] in middle of the right column on page 636, but without the source term:

$$\mathbf{r} = V\left(\mathrm{i}\omega\mu_0\tilde{\sigma}\mathbf{E} - \nabla\times\mu_\mathrm{r}^{-1}\nabla\times\mathbf{E}\right).$$

The calculation is carried out in a matrix-free manner; on said page 636 (or in the *Theory*) are the various steps laid out to discretise the different parts, for instance involved curls. This can also be understood as the left-hand-side of $Ax = b$, as given in Equation 2 in [Muld06] (here without the cell volumes V),

$$\mathrm{i}\omega\mu_0\tilde{\sigma}\mathrm{E} - \nabla\times\zeta^{-1}\nabla\times\mathrm{E} = -\mathrm{i}\omega\mu_0\mathrm{J_s}.$$

It can therefore be used as *matvec* to create a *LinearOperator*, which can be passed to a solver.

It is assumed that ex, ey, and ez have PEC boundaries; otherwise the output will not have PEC boundaries.

The residuals are subtracted in-place from *rx*, *ry*, and *rz*. That means that if *rx*, *ry*, and *rz* contain the source field, they will contain the total residual afterwards; if they are empty fields, they will contain the negative partial residual afterwards.

Parameters

rx, ry, rz [ndarray] Source field or pre-allocated zero residual field in x-, y-, and z-directions.

ex, ey, ez [ndarray] Electric fields in x-, y-, and z-directions, as obtained from *emg3d.fields.Field*.

eta_x, eta_y, eta_z, zeta [ndarray] VolumeModel parameters (multiplied by volumes) as obtained from *emg3d.models.VolumeModel()*.

hx, hy, hz [ndarray] Cell widths in x-, y-, and z-directions.

emg3d.core.**blocks_to_amat**(*amat*, *bvec*, *middle*, *left*, *rhs*, *im*, *nC*)

Insert middle, left, and rhs into main arrays amat and bvec.

The banded matrix amat contains the main diagonal and the first five lower off-diagonals. They are stored one column after the other, in a 6*n ndarray.

The complete main matrix *amat* and the *middle* and *left* blocks are given by:

```
.-0
|X|\   0
0-.-0       left:  middle:  right:
 \|X|\                      (not used)
  0-.-0      0-     .-      0
   \|X|\      \     |X      |\
    0-.-0
 0   \|X|
      0-.

. 1*1, - 4*1, | 1*4, X 4*4, \ 4*4 upper or lower
```

Both, *middle* and *left*, are 5x5 matrices. The corresponding right-hand-side *rhs* is filled into *bvec*. The matrices *left* and *middle* provided in a single call are horizontally aligned (not vertically). The sorting of amat (banded matrix) and bvec are given by:

```
 amat (66,)             example: n = 11                   bvec (11,)
 --------------                                            --
|01            |                    FIRST CALL             01
|02 07         |                    Only `middle` and `rhs`  02
|03 08 13      |                    are used, not `left`.    03
|04 09 14 19   |                                             04
|05 10 15 20 25|                                             05
 -------------- --------------                              --
| 0 11 16 21 26|31            |     SUBSEQUENT CALLS         06
|   12 17 22 27|32 37         |     (normal case)            07
|      18 23 28|33 38 43      |     Complete `left`,         08
|         24 29|34 39 44 49   |     `middle` and `rhs`       09
|            30|35 40 45 50 55|     are used.                10
 -------------- -------------- ---                          --
               | 0 41 46 51 56|61   LAST CALL                11
               |    0  0  0  0| 0   Only top row of `left`
               |       0  0  0| 0   and the first elements
               |          0  0| 0   of `middle` and `rhs`
               |             0| 0   are used.
                --------------- ---
                               | 0

Single zeros (0) show elements in amat which are 0, hence not used.
Their location in amat can be deduced from their neighbours.
```

Parameters

amat [ndarray] Main banded matrix (stored as array) of length 6*n.

bvec [ndarray] Main right-hand-side of length n.

middle [ndarray] Middle block of size 5x5, as ndarray of length 25. Only the diagonal and the lower triangular part are used.

left [ndarray] Left block of size 5x5, as ndarray of length 25. Only the diagonal and the first row are used.

rhs [ndarray] Corresponding right-hand-side of length 5.

im [int] Current minus-index of direction of line relaxation, {ixm, iym, izm}.

nC [int] Total number of cells in direction of line relaxation, {nCx, nCy, nCz}.

emg3d.core.**gauss_seidel**(*ex*, *ey*, *ez*, *sx*, *sy*, *sz*, *eta_x*, *eta_y*, *eta_z*, *zeta*, *hx*, *hy*, *hz*, *nu*)
Gauss-Seidel method.

Solves the linear equation system $Ax = b$ iteratively using the following method:

$$\mathbf{x}^{(k+1)} = L_*^{-1}\left(\mathbf{b} - U\mathbf{x}^{(k)}\right) \ ,$$

where L_* is the lower triangular component, and U the strictly upper triangular component, $A = L_* + U$:

$$L_* = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \ , \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \ .$$

On the coarsest grid it acts as direct solver, whereas on the fine grid it acts as a smoother with only few iterations, defined by ν (*nu*). Odd numbers of *nu* use forward ordering, even numbers use backwards ordering. `nu=2` is therefore one symmetric Gauss-Seidel iteration, one forward ordered iteration followed by one backward ordered iteration.

From [Muld06]: The method proposed by [ArFW00] is chosen as a smoother. It selects one node of the grid and simultaneously solves for the six degrees of freedom on the six edges attached to the node. If node (x_k, y_l, z_m) is selected, the six equations, $r_{x;k\pm1/2,l,m} = 0$, $r_{y;k,l\pm1/2,m} = 0$ and $r_{z;k,l,m\pm1/2} = 0$, are solved for $e_{x;k\pm1/2,l,m}$, $e_{y;k,l\pm1/2,m}$ and $e_{z;k,l,m\pm1/2}$. Here, this smoother is applied in a symmetric Gauss-Seidel fashion, following the lexicographical ordering of the nodes (x_k, y_l, z_m), with fastest index $k = 1, \ldots, N_x - 1$, intermediate index $l = 1, \ldots, N_y - 1$, and slowest index $m = 1, \ldots, N_z - 1$.

To actually solve the system of six equations a non-standard Cholesky factorisation is used, *solve()*.

Tangential components at the boundaries are assumed to be zero (PEC boundaries).

The result is stored in the provided electric fields *ex*, *ey*, and *ez*.

Parameters

ex, ey, ez [ndarray] Electric fields in x-, y-, and z-directions, as obtained from *emg3d.fields.Field*.

sx, sy, sz : Source fields in x-, y-, and z-directions, as obtained from *emg3d.fields.Field*.

eta_x, eta_y, eta_z, zeta : VolumeModel parameters (multiplied by volumes) as obtained from *emg3d.models.VolumeModel()*.

hx, hy, hz [ndarray] Cell widths in x-, y-, and z-directions.

nu [int] Number of Gauss-Seidel iterations.

emg3d.core.**gauss_seidel_x**(*ex*, *ey*, *ez*, *sx*, *sy*, *sz*, *eta_x*, *eta_y*, *eta_z*, *zeta*, *hx*, *hy*, *hz*, *nu*)
Gauss-Seidel method with line relaxation in x-direction.

This is the equivalent to *gauss_seidel()*, but with line relaxation in the x-direction. See *gauss_seidel()* for more details.

The resulting system A x = b to solve consists of n unknowns (x-vector), and the corresponding matrix A is a banded matrix with the main diagonal and five upper and lower diagonals:

```
.-0
|X|\   0
0-.-0       left:  middle:  right:
 \|X|\                      (not used)
  0-.-0      0-     .-      0
   \|X|\      \     |X      |\
    0-.-0
 0   \|X|
      0-.

. 1*1, - 4*1, | 1*4, X 4*4, \ 4*4 upper or lower
```

The matrix A is complex and symmetric (A = A^T), and therefore only the main diagonal and the lower five off-diagonals are required.

- The right-hand-side b has length 5*nCx-4 (nCx even).
- The matrix A has length of b and 1+2*5 diagonals; we use for it an array of length 6*len(b).

The values are calculated in rows of 5 lines, with the indicated middle and left matrices as indicated in the above scheme. These blocks are filled into the main matrix A and vector b, and subsequently solved with a non-standard Cholesky factorisation, *solve()*.

Tangential components at the boundaries are assumed to be 0 (PEC boundaries).

The result is stored in the provided electric fields *ex*, *ey*, and *ez*.

Parameters

ex, ey, ez [ndarray] Electric fields in x-, y-, and z-directions, as obtained from *emg3d.fields.Field*.

sx, sy, sz : Source fields in x-, y-, and z-directions, as obtained from *emg3d.fields.Field*.

eta_x, eta_y, eta_z, zeta : VolumeModel parameters (multiplied by volumes) as obtained from *emg3d.models.VolumeModel()*.

hx, hy, hz [ndarray] Cell widths in x-, y-, and z-directions.

nu [int] Number of Gauss-Seidel iterations.

emg3d.core.**gauss_seidel_y**(*ex*, *ey*, *ez*, *sx*, *sy*, *sz*, *eta_x*, *eta_y*, *eta_z*, *zeta*, *hx*, *hy*, *hz*, *nu*)

Gauss-Seidel method with line relaxation in y-direction.

This is the equivalent to *gauss_seidel()*, but with line relaxation in the y-direction. See *gauss_seidel()* for more details.

The resulting system A x = b to solve consists of n unknowns (x-vector), and the corresponding matrix A is a banded matrix with the main diagonal and five upper and lower diagonals:

```
.-0
|X|\   0
0-.-0       left:  middle:  right:
 \|X|\                      (not used)
  0-.-0      0-     .-      0
   \|X|\      \     |X      |\
    0-.-0
 0   \|X|
      0-.

. 1*1, - 4*1, | 1*4, X 4*4, \ 4*4 upper or lower
```

The matrix A is complex and symmetric (A = A^T), and therefore only the main diagonal and the lower five off-diagonals are required.

- The right-hand-side b has length 5*nCy-4 (nCy even).

- The matrix A has length of b and 1+2*5 diagonals; we use for it an array of length 6*len(b).

The values are calculated in rows of 5 lines, with the indicated middle and left matrices as indicated in the above scheme. These blocks are filled into the main matrix A and vector b, and subsequently solved with a non-standard Cholesky factorisation, *solve()*.

Note: The smoothing with linerelaxation in y-direction is carried out in reversed lexicographical order, in order to improve speed (memory access). All other smoothers (*gauss_seidel()*, *gauss_seidel_x()*, and *gauss_seidel_z()*) use lexicographical order.

Tangential components at the boundaries are assumed to be 0 (PEC boundaries).

The result is stored in the provided electric fields *ex*, *ey*, and *ez*.

Parameters

ex, ey, ez [ndarray] Electric fields in x-, y-, and z-directions, as obtained from *emg3d.fields.Field*.

sx, sy, sz : Source fields in x-, y-, and z-directions, as obtained from *emg3d.fields.Field*.

eta_x, eta_y, eta_z, zeta : VolumeModel parameters (multiplied by volumes) as obtained from *emg3d.models.VolumeModel()*.

hx, hy, hz [ndarray] Cell widths in x-, y-, and z-directions.

nu [int] Number of Gauss-Seidel iterations.

emg3d.core.**gauss_seidel_z**(*ex*, *ey*, *ez*, *sx*, *sy*, *sz*, *eta_x*, *eta_y*, *eta_z*, *zeta*, *hx*, *hy*, *hz*, *nu*)

Gauss-Seidel method with line relaxation in z-direction.

This is the equivalent to *gauss_seidel()*, but with line relaxation in the z-direction. See *gauss_seidel()* for more details.

The resulting system A x = b to solve consists of n unknowns (x-vector), and the corresponding matrix A is a banded matrix with the main diagonal and five upper and lower diagonals:

```
.-0
|X|\   0
0-.-0       left:  middle:  right:
 \|X|\                      (not used)
  0-.-0      0-     .-      0
   \|X|\      \     |X      |\
    0-.-0
 0   \|X|
      0-.

. 1*1, - 4*1, | 1*4, X 4*4, \ 4*4 upper or lower
```

The matrix A is complex and symmetric (A = A^T), and therefore only the main diagonal and the lower five off-diagonals are required.

- The right-hand-side b has length 5*nCz-4 (nCz even).
- The matrix A has length of b and 1+2*5 diagonals; we use for it an array of length 6*len(b).

The values are calculated in rows of 5 lines, with the indicated middle and left matrices as indicated in the above scheme. These blocks are filled into the main matrix A and vector b, and subsequently solved with a non-standard Cholesky factorisation, *solve()*.

Tangential components at the boundaries are assumed to be 0 (PEC boundaries).

The result is stored in the provided electric fields *ex*, *ey*, and *ez*.

Parameters

ex, ey, ez [ndarray] Electric fields in x-, y-, and z-directions, as obtained from *emg3d.fields.Field*.

sx, sy, sz : Source fields in x-, y-, and z-directions, as obtained from *emg3d.fields.Field*.

eta_x, eta_y, eta_z, zeta : VolumeModel parameters (multiplied by volumes) as obtained from *emg3d.models.VolumeModel()*.

hx, hy, hz [ndarray] Cell widths in x-, y-, and z-directions.

nu [int] Number of Gauss-Seidel iterations.

emg3d.core.**restrict**(*crx*, *cry*, *crz*, *rx*, *ry*, *rz*, *wx*, *wy*, *wz*, *sc_dir*)

Restriction of residual from fine to coarse grid.

Corresponds to Equation 8 in [Muld06]. The equation for the x-direction, using the notation $\{x, y, z\}$ instead of $\{1, 2, 3\}$, is given by

$$r^{2h}_{x,K+1/2,L,M} = \sum_{j_y=-1}^{1}\sum_{j_z=-1}^{1} w^y_{L,j_y} w^z_{M,j_z} \times \left(r^h_{x,k+1/2,l+j_y,m+j_z} + r^h_{x,k+3/2,l+j_y,m+j_z}\right).$$

The superscripts $h, 2h$ indicate quantities defined on the coarse grid and on the fine grid, respectively. The indices $\{K, L, M\}$ on the coarse grid correspond to $\{k, l, m\} = 2\{K, L, M\}$ on the fine grid. The weights w are obtained from *restrict_weights()*.

The restrictions of *rx*, *ry*, and *rz* are stored directly in *crx*, *cry*, and *crz*.

Parameters

crx, cry, crz [ndarray] Coarse grid {x,y,z}-directed residual (pre-allocated empty arrays).

rx, ry, rz [ndarray] Fine grid {x,y,z}-directed residual.

wx, wy, wz: tuple Tuples containing the weights (wl, w0, wr) as returned from *restrict_weights()* for the x-, y-, and z-directions.

sc_dir [int] Direction of semicoarsening; 0 for no semicoarsening.

emg3d.core.**restrict_weights**(*vectorN*, *vectorCC*, *h*, *cvectorN*, *cvectorCC*, *ch*)

Restriction weights for the coarse-grid correction operator.

Corresponds to Equation 9 in [Muld06]. A generalized version of that equation is given by

$$\begin{aligned} w^v_{Q,-1} &= \left(v^h_{q-1/2} - v^{2h}_{Q-1/2}\right)/d^v_{q-1}, \\ w^v_{Q,0} &= 1, \\ w^v_{Q,1} &= \left(v^{2h}_{Q+1/2} - v^h_{q+1/2}\right)/d^v_{q+1}, \end{aligned}$$

where d are the dual grid cell widths, v is one of $\{x, y, z\}$, and Q, q the corresponding entries of $\{K, L, M\}, \{k, l, m\}$. The superscripts $h, 2h$ indicate quantities defined on the coarse grid and on the fine grid, respectively. The indices $\{K, L, M\}$ on the coarse grid correspond to $\{k, l, m\} = 2\{K, L, M\}$ on the fine grid.

For the dual volume cell widths at the boundaries the scheme of [MoSu94] is applied, where $d^x_0 = h^x_{1/2}/2$ at $k = 0$, $d^x_{N_x} = h^x_{N_x-1/2}$ at $k = N_x$, and so on.

The following parameters must all be in the same direction, hence, all must be either for the x, the y, or the z direction. The returned weights are for this direction.

Parameters

vectorN, cvectorN [ndarray] Cell edges of the fine (vectorN) and coarse (cvectorN) grids.

vectorCC, cvectorCC [ndarray] Cell centers of the fine (vectorCC) and coarse (cvectorCC) grids.

h, ch [ndarray] Cell widths of the fine (h) and coarse (ch) grids.

Returns

wl, w0, wr [ndarray] Left, central, and right weights in the direction provided in the input.

emg3d.core.**solve**(*amat*, *bvec*)

Solve A x = b using a non-standard Cholesky factorisation.

Solve the system A x = b using a non-standard Cholesky factorisation without pivoting for a symmetric, complex matrix A tailored to the problem of the multigrid solver. The matrix A (amat) is an array of length 6*n, containing the main diagonal and the first five lower off-diagonals (ordered so that the first element of the main diagonal is followed by the first elements of the off diagonals, then the second elements and so on). The vector bvec has length b.

The solution is placed in b (bvec), and A (amat) is replaced by its decomposition.

1. Non-standard Cholesky factorisation.

   From [Muld07]: We use a non-standard Cholesky factorisation. The standard factorisation factors a hermitian matrix A into L L^H, where L is a lower triangular matrix and L^H its complex conjugate transpose. In our case, the discretisation is based on the Finite Integration Technique ([Weil77]) and provides a matrix A that is complex-valued and symmetric: A = A^T, where the superscript T denotes the transpose. The line relaxation scheme takes a matrix B that is a subset of A along the line. B is a complex symmetric band matrix with eleven diagonals. The non-standard Cholesky factorisation factors the matrix B into L L^T. Because of the symmetry, only the main diagonal and five lower diagonal elements of B need to be computed. The Cholesky factorisation replaces this matrix by L, containing six diagonals, after which the line relaxation can be carried out by simple back-substitution.

   $A = LDL^T$ factorisation without pivoting:

   $$D(j) = A(j,j) - \sum_{k=1}^{j-1} L(j,k)^2 D(k),\ j = 1,..,n;$$

   $$L(i,j) = \frac{1}{D(j)} \left[A(i,j) - \sum_{k=1}^{j-1} L(i,k)L(j,k)D(k) \right],\ i = j+1,..,n.$$

   A and L are in this case arrays, where $A(i,j) \to A(i+5j)$.

2. Solve A x = b.

   Solve A x = b, given L which is the result from the factorisation in the first step (and stored in A), hence, solve L x = b, where x is stored in b:

   $$b(j) = b(j) - \sum_{k=1}^{j-1} L(j,k)x(k), j = 2,..,n.$$

The result is equivalent with simply using `numpy.linalg.solve()`, but faster for the particular use-case of this code.

Note that in this custom solver there is no pivoting, and the diagonals of the matrix cannot be zero.

Parameters

amat [ndarray] Banded matrix A provided as a vector of length 6*n, containing main diagonal plus first five lower diagonals.

bvec [ndarray] Right-hand-side vector b of length n.

5.11.3 `utils` – Utilities

Utility functions for the multigrid solver.

class emg3d.utils.**Fourier**(*time*, *fmin*, *fmax*, *signal=0*, *ft='dlf'*, *ftarg=None*, ***kwargs*)

Time-domain CSEM calculation.

Class to carry out time-domain modelling with the frequency-domain code *emg3d*. Instances of the class take care of calculating the required frequencies, the interpolation from coarse, limited-band frequencies to the required frequencies, and carrying out the actual transform.

Everything related to the Fourier transform is done by utilising the capabilities of the 1D modeller `empymod`. The input parameters *time*, *signal*, *ft*, and *ftarg* are passed to the function `empymod.utils.check_time()` to obtain the required frequencies. The actual transform is subsequently carried out by calling `empymod.model.tem()`. See these functions for more details about the exact implementations of the Fourier transforms and its parameters. Note that also the *verb*-argument follows the definition in *empymod*.

The mapping from calculated frequencies to the frequencies required for the Fourier transform is done in three steps:

- Data for $f > f_{\max}$ is set to 0+0j.
- Data for $f < f_{\min}$ is interpolated by adding an additional data point at a frequency of 1e-100 Hz. The data for this point is `data.real[0]+0j`, hence the real part of the lowest calculated frequency and zero imaginary part. Interpolation is carried out using PCHIP `scipy.interpolate.pchip_interpolate()`.
- Data for $f_{\min} \leq f \leq f_{\max}$ is calculated with cubic spline interpolation (on a log-scale) `scipy.interpolate.InterpolatedUnivariateSpline`.

Note that *fmin* and *fmax* should be chosen wide enough such that the mapping for $f > f_{\max}$ $f < f_{\min}$ does not matter that much.

Parameters

time [ndarray] Desired times (s).

fmin, fmax [float] Minimum and maximum frequencies (Hz) to calculate:

- Data for freq > fmax is set to 0+0j.
- Data for freq < fmin is interpolated, using an extra data-point at f = 1e-100 Hz, with value data.real[0]+0j. (Hence zero imaginary part, and the lowest calculated real value.)

signal [{0, 1, -1}, optional]

Source signal, default is 0:

- None: Frequency-domain response
- -1 : Switch-off time-domain response
- 0 : Impulse time-domain response
- +1 : Switch-on time-domain response

ft [{'sin', 'cos', 'fftlog'}, optional] Flag to choose either the Digital Linear Filter method (Sine- or Cosine-Filter) or the FFTLog for the Fourier transform. Defaults to 'sin'.

ftarg [dict, optional] Depends on the value for *ft*:

- If *ft='dlf'*:
  - *dlf*: string of filter name in `empymod.filters` or the filter method itself. (Default: `empymod.filters.key_201_CosSin_2012()`)
  - *pts_per_dec*: points per decade; (default: -1)
    - If 0: Standard DLF.
    - If < 0: Lagged Convolution DLF.
    - If > 0: Splined DLF

- If *ft='fftlog'*:
  - *pts_per_dec*: sampels per decade (default: 10)
  - *add_dec*: additional decades [left, right] (default: [-2, 1])
  - *q*: exponent of power law bias (default: 0); -1 <= q <= 1

freq_inp [array] Frequencies to use for calculation. Mutually exclusive with *every_x_freq*.

every_x_freq [int] Every *every_x_freq*-th frequency of the required frequency-range is used for calculation. Mutually exclusive with *freq_calc*.

every_x_freq
If set, freq_coarse is every_x_freq-frequency of freq_req.

fmax
Maximum frequency (Hz) to calculate.

fmin
Minimum frequency (Hz) to calculate.

fourier_arguments(*self*, *ft*, *ftarg*)
Set Fourier type and its arguments.

freq2time(*self*, *fdata*, *off*)
Calculate corresponding time-domain signal.

Carry out the actual Fourier transform.

Parameters

fdata [ndarray] Frequency-domain data corresponding to *freq_calc*.

off [float] Corresponding offset (m).

Returns

tdata [ndarray] Time-domain data corresponding to Fourier.time.

freq_calc
Frequencies at which the model has to be calculated.

freq_calc_i
Indices of *freq_coarse* which have to be calculated.

freq_coarse
Coarse frequency range, can be different from *freq_req*.

freq_extrapolate
These are the frequencies to extrapolate.

In fact, it is dow via interpolation, using an extra data-point at f = 1e-100 Hz, with value data.real[0]+0j. (Hence zero imaginary part, and the lowest calculated real value.)

freq_extrapolate_i
Indices of the frequencies to extrapolate.

freq_inp
If set, freq_coarse is set to freq_inp.

freq_interpolate
These are the frequencies to interpolate.

If freq_req is equal freq_coarse, then this is eual to freq_calc.

freq_interpolate_i
Indices of the frequencies to interpolate.

If freq_req is equal freq_coarse, then this is eual to freq_calc_i.

freq_req
Frequencies required to carry out the Fourier transform.

ft
Type of Fourier transform. Set via `fourier_arguments(ft, ftarg)`.

ftarg
Fourier transform arguments. Set via `fourier_arguments(ft, ftarg)`.

interpolate(*self*, *fdata*)
Interpolate from calculated data to required data.

Parameters

fdata [ndarray] Frequency-domain data corresponding to *freq_calc*.

Returns

full_data [ndarray] Frequency-domain data corresponding to *freq_req*.

signal
Signal in time domain {0, 1, -1}.

time
Desired times (s).

class emg3d.utils.**Time**
Class for timing (now; runtime).

elapsed
Return runtime in seconds since time zero.

now
Return string of current time.

runtime
Return string of runtime since time zero.

t0
Return time zero of this class instance.

class emg3d.utils.**Report**(*add_pckg=None*, *ncol=3*, *text_width=80*, *sort=False*)
Print date, time, and version information.

Use *scooby* to print date, time, and package version information in any environment (Jupyter notebook, IPython console, Python console, QT console), either as html-table (notebook) or as plain text (anywhere).

Always shown are the OS, number of CPU(s), *numpy*, *scipy*, *emg3d*, *numba*, *sys.version*, and time/date.

Additionally shown are, if they can be imported, *IPython* and *matplotlib*. It also shows MKL information, if available.

All modules provided in *add_pckg* are also shown.

Note: The package *scooby* has to be installed in order to use *Report*: `pip install scooby`.

Parameters

add_pckg [packages, optional] Package or list of packages to add to output information (must be imported beforehand).

ncol [int, optional] Number of package-columns in html table (no effect in text-version); Defaults to 3.

text_width [int, optional] The text width for non-HTML display modes

sort [bool, optional] Sort the packages when the report is shown

Examples

```
>>> import pytest
>>> import dateutil
>>> from emg3d import Report
>>> Report()                                # Default values
>>> Report(pytest)                          # Provide additional package
>>> Report([pytest, dateutil], ncol=5)      # Set nr of columns
```

class emg3d.utils.**Field**

Create a Field instance with x-, y-, and z-views of the field.

A *Field* is an *ndarray* with additional views of the x-, y-, and z-directed fields as attributes, stored as *fx*, *fy*, and *fz*. The default array contains the whole field, which can be the electric field, the source field, or the residual field, in a 1D array. A *Field* instance has additionally the property *ensure_pec* which, if called, ensures Perfect Electric Conductor (PEC) boundary condition. It also has the two attributes *amp* and *pha* for the amplitude and phase, as common in frequency-domain CSEM.

A *Field* can be initiated in three ways:

1. `Field(grid, dtype=complex)`: Calling it with a *TensorMesh* instance returns a *Field* instance of correct dimensions initiated with zeroes of data type *dtype*.
2. `Field(grid, field)`: Calling it with a *TensorMesh* instance and an *ndarray* returns a *Field* instance of the provided *ndarray*, of same data type.
3. `Field(fx, fy, fz)`: Calling it with three *ndarray*'s which represent the field in x-, y-, and z-direction returns a *Field* instance with these views, of same data type.

Sort-order is 'F'.

Parameters

fx_or_grid [*TensorMesh* or ndarray] Either a TensorMesh instance or an ndarray of shape grid.nEx or grid.vnEx. See explanations above. Only mandatory parameter; if the only one provided, it will initiate a zero-field of *dtype*.

fy_or_field [*Field* or ndarray, optional] Either a Field instance or an ndarray of shape grid.nEy or grid.vnEy. See explanations above.

fz [ndarray, optional] An ndarray of shape grid.nEz or grid.vnEz. See explanations above.

dtype [dtype, optional] Only used if `fy_or_field=None` and `fz=None`; the initiated zero-field for the provided TensorMesh has data type *dtype*. Default: complex.

freq [float, optional] Source frequency (Hz), used to calculate the Laplace parameter s. Either positive or negative:

- *freq* > 0: Frequency domain, hence $s = -\mathrm{i}\omega = -2\mathrm{i}\pi f$ (complex);
- *freq* < 0: Laplace domain, hence $s = f$ (real).

Just added as info if provided.

amp (*self*)

Amplitude of the electromagnetic field.

copy (*self*)

Return a copy of the Field.

ensure_pec

Set Perfect Electric Conductor (PEC) boundary condition.

field

Entire field, 1D [fx, fy, fz].

freq

Return frequency.

classmethod from_dict(*inp*)
Convert dictionary into *Field* instance.

Parameters

inp [dict] Dictionary as obtained from *Field.to_dict()*. The dictionary needs the keys *field*, *freq*, *vnEx*, *vnEy*, and *vnEz*.

Returns

obj [*Field* instance]

fx
View of the x-directed field in the x-direction (nCx, nNy, nNz).

fy
View of the field in the y-direction (nNx, nCy, nNz).

fz
View of the field in the z-direction (nNx, nNy, nCz).

is_electric
Returns True if Field is electric, False if it is magnetic.

pha(*self*, *deg=False*, *unwrap=True*, *lag=True*)
Phase of the electromagnetic field.

Parameters

deg [bool] If True the returned phase is in degrees, else in radians. Default is False (radians).

unwrap [bool] If True the returned phase is unwrapped. Default is True (unwrapped).

lag [bool] If True the returned phase is lag, else lead defined. Default is True (lag defined).

smu0
Return s*mu_0; mu_0 = Magn. permeability of free space [H/m].

sval
Return s; s=iw in frequency domain; s=freq in Laplace domain.

to_dict(*self*, *copy=False*)
Store the necessary information of the Field in a dict.

class emg3d.utils.**SourceField**
Create a Source-Field instance with x-, y-, and z-views of the field.

A subclass of *Field*. Additional properties are the real-valued source vector (*vector*, *vx*, *vy*, *vz*), which sum is always one. For a *SourceField* frequency is a mandatory parameter, unlike for a *Field* (recommended also for *Field* though),

Parameters

fx_or_grid [*TensorMesh* or ndarray] Either a TensorMesh instance or an ndarray of shape grid.nEx or grid.vnEx. See explanations above. Only mandatory parameter; if the only one provided, it will initiate a zero-field of *dtype*.

fy_or_field [*Field* or ndarray, optional] Either a Field instance or an ndarray of shape grid.nEy or grid.vnEy. See explanations above.

fz [ndarray, optional] An ndarray of shape grid.nEz or grid.vnEz. See explanations above.

dtype [dtype, optional] Only used if `fy_or_field=None` and `fz=None`; the initiated zero-field for the provided TensorMesh has data type *dtype*. Default: complex.

freq [float] Source frequency (Hz), used to calculate the Laplace parameter *s*. Either positive or negative:

- *freq* > 0: Frequency domain, hence $s = -\mathrm{i}\omega = -2\mathrm{i}\pi f$ (complex);
- *freq* < 0: Laplace domain, hence $s = f$ (real).

In difference to *Field*, the frequency has to be provided for a *SourceField*.

copy(*self*)
Return a copy of the SourceField.

classmethod from_dict(*inp*)
Convert dictionary into *SourceField* instance.

Parameters

inp [dict] Dictionary as obtained from `SourceField.to_dict()`. The dictionary needs the keys *field*, *freq*, *vnEx*, *vnEy*, and *vnEz*.

Returns

obj [*SourceField* instance]

vector
Entire vector, 1D [vx, vy, vz].

vx
View of the x-directed vector in the x-direction (nCx, nNy, nNz).

vy
View of the vector in the y-direction (nNx, nCy, nNz).

vz
View of the vector in the z-direction (nNx, nNy, nCz).

emg3d.utils.**get_source_field**(*grid*, *src*, *freq*, *strength=0*)
Return the source field.

The source field is given in Equation 2 in [Muld06],

$$s\mu_0\mathbf{J}_\mathrm{s},$$

where $s = \mathrm{i}\omega$. Either finite length dipoles or infinitesimal small point dipoles can be defined, whereas the return source field corresponds to a normalized (1 Am) source distributed within the cell(s) it resides (can be changed with the *strength*-parameter).

The adjoint of the trilinear interpolation is used to distribute the point(s) to the grid edges, which corresponds to the discretization of a Dirac ([PlDM07]).

Parameters

grid [TensorMesh] Model grid; a *TensorMesh* instance.

src [list of floats] Source coordinates (m). There are two formats:

- Finite length dipole: `[x0, x1, y0, y1, z0, z1]`.
- Point dipole: `[x, y, z, azimuth, dip]`.

freq [float] Source frequency (Hz), used to calculate the Laplace parameter s. Either positive or negative:

- *freq* > 0: Frequency domain, hence $s = -\mathrm{i}\omega = -2\mathrm{i}\pi f$ (complex);
- *freq* < 0: Laplace domain, hence $s = f$ (real).

strength [float or complex, optional] Source strength (A):

- If 0, output is normalized to a source of 1 m length, and source strength of 1 A.
- If != 0, output is returned for given source length and strength.

Default is 0.

Returns

sfield [*SourceField()* instance] Source field, normalized to 1 A m.

emg3d.utils.**get_receiver**(*grid*, *values*, *coordinates*, *method='cubic'*, *extrapolate=False*)

Return values corresponding to grid at coordinates.

Works for electric fields as well as magnetic fields obtained with *get_h_field()*, and for model parameters.

Parameters

grid [TensorMesh] Model grid; a *TensorMesh* instance.

values [ndarray] Field instance, or a particular field (e.g. field.fx); Model parameters.

coordinates [tuple (x, y, z)] Coordinates (x, y, z) where to interpolate *values*; e.g. receiver locations.

method [str, optional] The method of interpolation to perform, 'linear' or 'cubic'. Default is 'cubic' (forced to 'linear' if there are less than 3 points in any direction).

extrapolate [bool] If True, points on *new_grid* which are outside of *grid* are filled by the nearest value (if `method='cubic'`) or by extrapolation (if `method='linear'`). If False, points outside are set to zero.

Default is False.

Returns

new_values [ndarray or `empymod.utils.EMArray`] Values at *coordinates*.

If input was a field it returns an EMArray, which is a subclassed ndarray with `.pha` and `.amp` attributes.

If input was an entire Field instance, output is a tuple (fx, fy, fz).

See also:

grid2grid Interpolation of model parameters or fields to a new grid.

emg3d.utils.**get_h_field**(*grid*, *model*, *field*)

Return magnetic field corresponding to provided electric field.

Retrieve the magnetic field $\mathbf{H}$ from the electric field $\mathbf{E}$ using Farady's law, given by

$$\nabla \times \mathbf{E} = \mathrm{i}\omega\mu\mathbf{H}.$$

Note that the magnetic field in x-direction is defined in the center of the face defined by the electric field in y- and z-directions, and similar for the other field directions. This means that the provided electric field and the returned magnetic field have different dimensions:

```
E-field:  x: [grid.vectorCCx,  grid.vectorNy,  grid.vectorNz]
          y: [ grid.vectorNx, grid.vectorCCy,  grid.vectorNz]
          z: [ grid.vectorNx,  grid.vectorNy, grid.vectorCCz]

H-field:  x: [ grid.vectorNx, grid.vectorCCy, grid.vectorCCz]
          y: [grid.vectorCCx,  grid.vectorNy, grid.vectorCCz]
          z: [grid.vectorCCx, grid.vectorCCy,  grid.vectorNz]
```

Parameters

grid [TensorMesh] Model grid; *TensorMesh* instance.

model [Model] Model; *Model* instance.

field [Field] Electric field; *Field* instance.

Returns

hfield [Field] Magnetic field; *Field* instance.

class emg3d.utils.**Model**(*grid*, *res_x=1.0*, *res_y=None*, *res_z=None*, *mu_r=None*, *epsilon_r=None*)

Create a model instance.

Class to provide model parameters (x-, y-, and z-directed resistivities, electric permittivity and magnetic permeability) to the solver. Relative magnetic permeability μ_r is by default set to one and electric permittivity ε_r is by default set to zero, but they can also be provided (isotropically). Keep in mind that the multigrid method as implemented in *emg3d* only works for the diffusive approximation. As soon as the displacement-part in the Maxwell's equations becomes too dominant it will fail (high frequencies or very high electric permittivity).

Parameters

grid [TensorMesh] Grid on which to apply model.

res_x, res_y, res_z [float or ndarray; default to 1.] Resistivity in x-, y-, and z-directions. If ndarray, they must have the shape of grid.vnC (F-ordered) or grid.nC. Resistivities have to be bigger than zero and smaller than infinity.

mu_r [None, float, or ndarray] Relative magnetic permeability (isotropic). If ndarray it must have the shape of grid.vnC (F-ordered) or grid.nC. Default is None, which corresponds to 1., but avoids the calculation of zeta. Magnetic permeability has to be bigger than zero and smaller than infinity.

epsilon_r [None, float, or ndarray] Relative electric permittivity (isotropic). If ndarray it must have the shape of grid.vnC (F-ordered) or grid.nC. The displacement part is completely neglected (diffusive approximation) if set to None, which is the default. Electric permittivity has to be bigger than zero and smaller than infinity.

copy(*self*)
Return a copy of the Model.

epsilon_r
Electric permittivity.

classmethod from_dict(*inp*)
Convert the dictionary into a Model instance.

Parameters

inp [dict] Dictionary as obtained from *Model.to_dict()*. The dictionary needs the keys *res_x*, *res_y*, *res_z*, *mu_r*, *epsilon_r*, and *vnC*.

Returns

obj [*Model* instance]

mu_r
Magnetic permeability.

res_x
Resistivity in x-direction.

res_y
Resistivity in y-direction.

res_z
Resistivity in z-direction.

to_dict(*self*, *copy=False*)
Store the necessary information of the Model in a dict.

class emg3d.utils.**VolumeModel**(*grid*, *model*, *sfield*)
Return a volume-averaged version of provided model.

Takes a Model instance and returns the volume averaged values. This is used by the solver internally.

$$\eta_{\{x,y,z\}} = -V \mathrm{i}\omega\mu_0 \left(\rho^{-1}_{\{x,y,z\}} + \mathrm{i}\omega\varepsilon\right)$$

$$\zeta = V \mu_\mathrm{r}^{-1}$$

Parameters

grid [TensorMesh] Grid on which to apply model.

model [Model] Model to transform to volume-averaged values.

sfield [SourceField] A VolumeModel is frequency-dependent. The frequency-information is taken from the provided source filed.

static calculate_eta(*name*, *grid*, *model*, *field*)
eta: volume divided by resistivity.

static calculate_zeta(*name*, *grid*, *model*)
zeta: volume divided by mu_r.

eta_x
eta in x-direction.

eta_y
eta in y-direction.

eta_z
eta in z-direction.

zeta
zeta.

emg3d.utils.**grid2grid**(*grid*, *values*, *new_grid*, *method='linear'*, *extrapolate=True*, *log=False*)
Interpolate *values* located on *grid* to *new_grid*.

Note 1: The default method is 'linear', because it works with fields and model parameters. However, recommended are 'volume' for model parameters and 'cubic' for fields.

Note 2: For model parameters with *method='volume'* the result is quite different if you provide resistivity, conductivity, or the logarithm of any of the two. The recommended way is to provide the logarithm of resistivity or conductivity, in which case the output of one is indeed the inverse of the output of the other.

Parameters

grid, new_grid [TensorMesh] Input and output model grids; *TensorMesh* instances.

values [ndarray] Model parameters; *emg3d.fields.Field* instance, or a particular field (e.g. field.fx). For fields the method cannot be 'volume'.

method [{<'linear'>, 'volume', 'cubic'}, optional] The method of interpolation to perform. The volume averaging method ensures that the total sum of the property stays constant.

Volume averaging is only implemented for model parameters, not for fields. The method 'cubic' requires at least three points in any direction, otherwise it will fall back to 'linear'.

Default is 'linear', because it works with fields and model parameters. However, recommended are 'volume' for model parameters and 'cubic' for fields.

extrapolate [bool] If True, points on *new_grid* which are outside of *grid* are filled by the nearest value (if `method='cubic'`) or by extrapolation (if `method='linear'`). If False, points outside are set to zero.

For `method='volume'` it always uses the nearest value for points outside of *grid*.

Default is True.

log [bool] If True, the interpolation is carried out on a log10-scale; hence the same as `10**grid2grid(grid, np.log10(values), ...)`. Default is False.

Returns

new_values [ndarray] Values corresponding to *new_grid*.

See also:

get_receiver Interpolation of model parameters or fields to (x, y, z).

emg3d.utils.**interp3d**(*points*, *values*, *new_points*, *method*, *fill_value*, *mode*)

Interpolate values in 3D either linearly or with a cubic spline.

Return *values* corresponding to a regular 3D grid defined by *points* on *new_points*.

This is a modified version of `scipy.interpolate.interpn()`, using `scipy.interpolate.RegularGridInterpolator` if `method='linear'` and a custom-wrapped version of `scipy.ndimage.map_coordinates()` if `method='cubic'`. If speed is important then choose 'linear', as it can be significantly faster.

Parameters

points [tuple of ndarray of float, with shapes ((nx,), (ny,) (nz,))] The points defining the regular grid in three dimensions.

values [array_like, shape (nx, ny, nz)] The data on the regular grid in three dimensions.

new_points [tuple (rec_x, rec_y, rec_z)] Coordinates (x, y, z) of new points.

method [{'cubic', 'linear'}, optional] The method of interpolation to perform, 'linear' or 'cubic'. Default is 'cubic' (forced to 'linear' if there are less than 3 points in any direction).

fill_value [float or None] Passed to `scipy.interpolate.RegularGridInterpolator` if `method='linear'`: The value to use for points outside of the interpolation domain. If None, values outside the domain are extrapolated.

mode [{'constant', 'nearest', 'mirror', 'reflect', 'wrap'}] Passed to `scipy.ndimage.map_coordinates()` if `method='cubic'`: Determines how the input array is extended beyond its boundaries.

Returns

new_values [ndarray] Values corresponding to *new_points*.

class emg3d.utils.**TensorMesh**(*h*, *x0*)

Rudimentary mesh for multigrid calculation.

The tensor-mesh `discretize.TensorMesh` is a powerful tool, including sophisticated mesh-generation possibilities in 1D, 2D, and 3D, plotting routines, and much more. However, in the multigrid solver we have to generate a mesh at each level, many times over and over again, and we only need a very limited set of attributes. This tensor-mesh class provides all required attributes. All attributes here are the same as their counterparts in `discretize.TensorMesh` (both in name and value).

Warning: This is a slimmed-down version of `discretize.TensorMesh`, meant principally for internal use by the multigrid modeller. It is highly recommended to use `discretize.TensorMesh` to create the input meshes instead of this class. There are no input-checks carried out here, and there is only one accepted input format for *h* and *x0*.

Parameters

h [list of three ndarrays] Cell widths in [x, y, z] directions.

x0 [ndarray of dimension (3,)] Origin (x, y, z).

copy(*self*)
Return a copy of the TensorMesh.

classmethod from_dict(*inp*)
Convert dictionary into `TensorMesh` instance.

Parameters

inp [dict] Dictionary as obtained from `TensorMesh.to_dict()`. The dictionary needs the keys *hx*, *hy*, *hz*, and *x0*.

Returns

obj [`TensorMesh` instance]

to_dict(*self*, *copy=False*)
Store the necessary information of the TensorMesh in a dict.

vol
Construct cell volumes of the 3D model as 1D array.

emg3d.utils.**get_hx_h0**(*freq*, *res*, *domain*, *fixed=0.0*, *possible_nx=None*, *min_width=None*, *pps=3*, *alpha=None*, *max_domain=100000.0*, *raise_error=True*, *verb=1*, *return_info=False*)
Return cell widths and origin for given parameters.

Returns cell widths for the provided frequency, resistivity, domain extent, and other parameters using a flexible amount of cells. See input parameters for more details. A maximum of three hard/fixed boundaries can be provided (one of which is the grid center).

The minimum cell width is calculated through δ/pps, where the skin depth is given by $\delta = 503.3\sqrt{\rho/f}$, and the parameter *pps* stands for 'points-per-skindepth'. The minimum cell width can be restricted with the parameter *min_width*.

The actual calculation domain adds a buffer zone around the (survey) domain. The thickness of the buffer is six times the skin depth. The field is basically zero after two wavelengths. A wavelength is $2\pi\delta$, hence roughly 6 times the skin depth. Taking a factor 6 gives therefore almost two wavelengths, as the field travels to the boundary and back. The actual buffer thickness can be steered with the *res* parameter.

One has to take into account that the air is very resistive, which has to be considered not just in the vertical direction, but also in the horizontal directions, as the airwave will bounce back from the sides otherwise. In the marine case this issue reduces with increasing water depth.

Parameters

freq [float] Frequency (Hz) to calculate the skin depth. The skin depth is a concept defined in the frequency domain. If a negative frequency is provided, it is assumed that the calculation is carried out in the Laplace domain. To calculate the skin depth, the value of *freq* is then multiplied by -2π, to simulate the closest frequency-equivalent.

res [float or list] Resistivity (Ohm m) to calculate the skin depth. The skin depth is used to calculate the minimum cell width and the boundary thicknesses. Up to three resistivities can be provided:

- float: Same resistivity for everything;
- [min_width, boundaries];
- [min_width, left boundary, right boundary].

domain [list] Contains the survey-domain limits [min, max]. The actual calculation domain consists of this domain plus a buffer zone around it, which depends on frequency and resistivity.

fixed [list, optional] Fixed boundaries, one, two, or maximum three values. The grid is centered around the first value. Hence it is the center location with the smallest cell. Two more fixed boundaries can be added, at most one on each side of the first one. Default is 0.

possible_nx [list, optional] List of possible numbers of cells. See *get_cell_numbers()*. Default is `get_cell_numbers(500, 5, 3)`, which corresponds to [16, 24, 32, 40, 48, 64, 80, 96, 128, 160, 192, 256, 320, 384].

min_width [float, list or None, optional] Minimum cell width restriction:

- None : No restriction;
- float : Fixed to this value, ignoring skin depth and *pps*.
- list [min, max] : Lower and upper bounds.

Default is None.

pps [int, optional] Points per skindepth; minimum cell width is calculated via *dmin* = *skindepth/pps*. Default = 3.

alpha [list, optional] Maximum alpha and step size to find a good alpha. The first value is the maximum alpha of the survey domain, the second value is the maximum alpha for the buffer zone, and the third value is the step size. Default = [1, 1.5, .01], hence no stretching within the survey domain and a maximum stretching of 1.5 in the buffer zone; step size is 0.01.

max_domain [float, optional] Maximum calculation domain from fixed[0] (usually source position). Default is 100,000.

raise_error [bool, optional] If True, an error is raised if no suitable grid is found. Otherwise it just prints a message and returns None's. Default is True.

verb [int, optional] Verbosity, 0 or 1. Default = 1.

return_info [bool] If True, a dictionary is returned with some grid info (min and max cell width and alpha).

Returns

hx [ndarray] Cell widths of mesh.

x0 [float] Origin of the mesh.

info [dict] Dictionary with mesh info; only if `return_info=True`.

Keys:

- *dmin*: Minimum cell width;
- *dmax*: Maximum cell width;
- *amin*: Minimum alpha;
- *amax*: Maximum alpha.

See also:

get_stretched_h Get *hx* for a fixed number *nx* and within a fixed domain.

emg3d.utils.**get_cell_numbers**(*max_nr*, *max_prime=5*, *min_div=3*)

Returns 'good' cell numbers for the multigrid method.

'Good' cell numbers are numbers which can be divided by 2 as many times as possible. At the end there will be a low prime number.

The function adds all numbers $p2^n \leq M$ for $p = 2, 3, ..., p_{\max}$ and $n = n_{\min}, n_{\min}+1, ..., \infty$; $M, p_{\max}, n_{\min}$ correspond to *max_nr*, *max_prime*, and *min_div*, respectively.

Parameters

max_nr [int] Maximum number of cells.

max_prime [int] Highest permitted prime number p for p*2^n. {2, 3, 5, 7} are good upper limits in order to avoid too big lowest grids in the multigrid method. Default is 5.

min_div [int] Minimum times the number can be divided by two. Default is 3.

Returns

numbers [array] Array containing all possible cell numbers from lowest to highest.

emg3d.utils.**get_stretched_h**(*min_width*, *domain*, *nx*, *x0=0*, *x1=None*, *resp_domain=False*)

Return cell widths for a stretched grid within the domain.

Returns *nx* cell widths within *domain*, where the minimum cell width is *min_width*. The cells are not stretched within *x0* and *x1*, and outside uses a power-law stretching. The actual stretching factor and the number of cells left and right of *x0* and *x1* are find in a minimization process.

The domain is not completely respected. The starting point of the domain is, but the endpoint of the domain might slightly shift (this is more likely the case for small *nx*, for big *nx* the shift should be small). The new endpoint can be obtained with `domain[0]+np.sum(hx)`. If you want the domain to be respected absolutely, set `resp_domain=True`. However, be aware that this will introduce one stretch-factor which is different from the other stretch factors, to accommodate the restriction. This one-off factor is between the left- and right-side of *x0*, or, if *x1* is provided, just after *x1*.

Parameters

min_width [float] Minimum cell width. If x1 is provided, the actual minimum cell width might be smaller than min_width.

domain [list] [start, end] of model domain.

nx [int] Number of cells.

x0 [float] Center of the grid. *x0* is restricted to *domain*. Default is 0.

x1 [float] If provided, then no stretching is applied between *x0* and *x1*. The non-stretched part starts at *x0* and stops at the first possible location at or after *x1*. *x1* is restricted to *domain*. This will min_width so that an integer number of cells fit within x0 and x1.

resp_domain [bool] If False (default), then the domain-end might shift slightly to assure that the same stretching factor is applied throughout. If set to True, however, the domain is respected absolutely. This will introduce one stretch-factor which is different from the other stretch factors, to accommodate the restriction. This one-off factor is between the left- and right-side of *x0*, or, if *x1* is provided, just after *x1*.

Returns

hx [ndarray] Cell widths of mesh.

See also:

get_hx_x0 Get *hx* and *x0* for a flexible number of *nx* with given bounds.

emg3d.utils.**get_domain**(*x0=0*, *freq=1*, *res=0.3*, *limits=None*, *min_width=None*, *fact_min=0.2*, *fact_neg=5*, *fact_pos=None*)

Get domain extent and minimum cell width as a function of skin depth.

Returns the extent of the calculation domain and the minimum cell width as a multiple of the skin depth, with possible user restrictions on minimum calculation domain and range of possible minimum cell widths.

$$\delta = 503.3\sqrt{\frac{\rho}{f}},$$
$$x_{\text{start}} = x_0 - k_{\text{neg}}\delta,$$
$$x_{\text{end}} = x_0 + k_{\text{pos}}\delta,$$
$$h_{\text{min}} = k_{\text{min}}\delta.$$

Parameters

x0 [float] Center of the calculation domain. Normally the source location. Default is 0.

freq [float] Frequency (Hz) to calculate the skin depth. The skin depth is a concept defined in the frequency domain. If a negative frequency is provided, it is assumed that the calculation is carried out in the Laplace domain. To calculate the skin depth, the value of *freq* is then multiplied by -2π, to simulate the closest frequency-equivalent.

Default is 1 Hz.

res [float, optional] Resistivity (Ohm m) to calculate skin depth. Default is 0.3 Ohm m (sea water).

limits [None or list] [start, end] of model domain. This extent represents the minimum extent of the domain. The domain is therefore only adjusted if it has to reach outside of [start, end]. Default is None.

min_width [None, float, or list of two floats] Minimum cell width is calculated as a function of skin depth: fact_min*sd. If *min_width* is a float, this is used. If a list of two values [min, max] are provided, they are used to restrain min_width. Default is None.

fact_min, fact_neg, fact_pos [floats] The skin depth is multiplied with these factors to estimate:

- Minimum cell width (*fact_min*, default 0.2)
- Domain-start (*fact_neg*, default 5), and
- Domain-end (*fact_pos*, defaults to *fact_neg*).

Returns

h_min [float] Minimum cell width.

domain [list] Start- and end-points of calculation domain.

emg3d.utils.**get_hx**(*alpha*, *domain*, *nx*, *x0*, *resp_domain=True*)

Return cell widths for given input.

Find the number of cells left and right of *x0*, *nl* and *nr* respectively, for the provided alpha. For this, we solve

$$\frac{x_{\max} - x_0}{x_0 - x_{\min}} = \frac{a^{nr} - 1}{a^{nl} - 1}$$

where $a = 1 + \alpha$.

Parameters

alpha [float] Stretching factor *a* is given by `a=1+alpha`.

domain [list] [start, end] of model domain.

nx [int] Number of cells.

x0 [float] Center of the grid. *x0* is restricted to *domain*.

resp_domain [bool] If False (default), then the domain-end might shift slightly to assure that the same stretching factor is applied throughout. If set to True, however, the domain is respected absolutely. This will introduce one stretch-factor which is different from the other stretch factors, to accommodate the restriction. This one-off factor is between the left- and right-side of *x0*, or, if *x1* is provided, just after *x1*.

Returns

hx [ndarray] Cell widths of mesh.

5.11.4 `meshes` – Discretization

Everything related to meshes appropriate for the multigrid solver.

class emg3d.meshes.**TensorMesh**(*h*, *x0*)
Rudimentary mesh for multigrid calculation.

The tensor-mesh `discretize.TensorMesh` is a powerful tool, including sophisticated mesh-generation possibilities in 1D, 2D, and 3D, plotting routines, and much more. However, in the multigrid solver we have to generate a mesh at each level, many times over and over again, and we only need a very limited set of attributes. This tensor-mesh class provides all required attributes. All attributes here are the same as their counterparts in `discretize.TensorMesh` (both in name and value).

Warning: This is a slimmed-down version of `discretize.TensorMesh`, meant principally for internal use by the multigrid modeller. It is highly recommended to use `discretize.TensorMesh` to create the input meshes instead of this class. There are no input-checks carried out here, and there is only one accepted input format for *h* and *x0*.

Parameters

h [list of three ndarrays] Cell widths in [x, y, z] directions.

x0 [ndarray of dimension (3,)] Origin (x, y, z).

copy(*self*)
Return a copy of the TensorMesh.

classmethod from_dict(*inp*)
Convert dictionary into *TensorMesh* instance.

Parameters

inp [dict] Dictionary as obtained from *TensorMesh.to_dict()*. The dictionary needs the keys *hx*, *hy*, *hz*, and *x0*.

Returns

obj [*TensorMesh* instance]

to_dict(*self*, *copy=False*)
Store the necessary information of the TensorMesh in a dict.

vol
Construct cell volumes of the 3D model as 1D array.

emg3d.meshes.**get_hx_h0**(*freq*, *res*, *domain*, *fixed=0.0*, *possible_nx=None*, *min_width=None*, *pps=3*, *alpha=None*, *max_domain=100000.0*, *raise_error=True*, *verb=1*, *return_info=False*)
Return cell widths and origin for given parameters.

Returns cell widths for the provided frequency, resistivity, domain extent, and other parameters using a flexible amount of cells. See input parameters for more details. A maximum of three hard/fixed boundaries can be provided (one of which is the grid center).

The minimum cell width is calculated through δ/pps, where the skin depth is given by $\delta = 503.3\sqrt{\rho/f}$, and the parameter *pps* stands for 'points-per-skindepth'. The minimum cell width can be restricted with the parameter *min_width*.

The actual calculation domain adds a buffer zone around the (survey) domain. The thickness of the buffer is six times the skin depth. The field is basically zero after two wavelengths. A wavelength is $2\pi\delta$, hence roughly 6 times the skin depth. Taking a factor 6 gives therefore almost two wavelengths, as the field travels to the boundary and back. The actual buffer thickness can be steered with the *res* parameter.

One has to take into account that the air is very resistive, which has to be considered not just in the vertical direction, but also in the horizontal directions, as the airwave will bounce back from the sides otherwise. In the marine case this issue reduces with increasing water depth.

Parameters

freq [float] Frequency (Hz) to calculate the skin depth. The skin depth is a concept defined in the frequency domain. If a negative frequency is provided, it is assumed that the calculation is carried out in the Laplace domain. To calculate the skin depth, the value of *freq* is then multiplied by -2π, to simulate the closest frequency-equivalent.

res [float or list] Resistivity (Ohm m) to calculate the skin depth. The skin depth is used to calculate the minimum cell width and the boundary thicknesses. Up to three resistivities can be provided:

- float: Same resistivity for everything;
- [min_width, boundaries];
- [min_width, left boundary, right boundary].

domain [list] Contains the survey-domain limits [min, max]. The actual calculation domain consists of this domain plus a buffer zone around it, which depends on frequency and resistivity.

fixed [list, optional] Fixed boundaries, one, two, or maximum three values. The grid is centered around the first value. Hence it is the center location with the smallest cell. Two more fixed boundaries can be added, at most one on each side of the first one. Default is 0.

possible_nx [list, optional] List of possible numbers of cells. See *get_cell_numbers()*. Default is `get_cell_numbers(500, 5, 3)`, which corresponds to [16, 24, 32, 40, 48, 64, 80, 96, 128, 160, 192, 256, 320, 384].

min_width [float, list or None, optional] Minimum cell width restriction:

- None : No restriction;
- float : Fixed to this value, ignoring skin depth and *pps*.
- list [min, max] : Lower and upper bounds.

Default is None.

pps [int, optional] Points per skindepth; minimum cell width is calculated via *dmin* = *skindepth/pps*. Default = 3.

alpha [list, optional] Maximum alpha and step size to find a good alpha. The first value is the maximum alpha of the survey domain, the second value is the maximum alpha for the buffer zone, and the third value is the step size. Default = [1, 1.5, .01], hence no stretching within the survey domain and a maximum stretching of 1.5 in the buffer zone; step size is 0.01.

max_domain [float, optional] Maximum calculation domain from fixed[0] (usually source position). Default is 100,000.

raise_error [bool, optional] If True, an error is raised if no suitable grid is found. Otherwise it just prints a message and returns None's. Default is True.

verb [int, optional] Verbosity, 0 or 1. Default = 1.

return_info [bool] If True, a dictionary is returned with some grid info (min and max cell width and alpha).

Returns

hx [ndarray] Cell widths of mesh.

x0 [float] Origin of the mesh.

info [dict] Dictionary with mesh info; only if `return_info=True`.

Keys:

- *dmin*: Minimum cell width;
- *dmax*: Maximum cell width;
- *amin*: Minimum alpha;
- *amax*: Maximum alpha.

See also:

get_stretched_h Get *hx* for a fixed number *nx* and within a fixed domain.

emg3d.meshes.**get_cell_numbers**(*max_nr*, *max_prime=5*, *min_div=3*)
Returns 'good' cell numbers for the multigrid method.

'Good' cell numbers are numbers which can be divided by 2 as many times as possible. At the end there will be a low prime number.

The function adds all numbers $p2^n \leq M$ for $p = 2, 3, ..., p_{\max}$ and $n = n_{\min}, n_{\min}+1, ..., \infty$; $M, p_{\max}, n_{\min}$ correspond to *max_nr*, *max_prime*, and *min_div*, respectively.

Parameters

max_nr [int] Maximum number of cells.

max_prime [int] Highest permitted prime number p for p*2^n. {2, 3, 5, 7} are good upper limits in order to avoid too big lowest grids in the multigrid method. Default is 5.

min_div [int] Minimum times the number can be divided by two. Default is 3.

Returns

numbers [array] Array containing all possible cell numbers from lowest to highest.

emg3d.meshes.**get_stretched_h**(*min_width*, *domain*, *nx*, *x0=0*, *x1=None*, *resp_domain=False*)
Return cell widths for a stretched grid within the domain.

Returns *nx* cell widths within *domain*, where the minimum cell width is *min_width*. The cells are not stretched within *x0* and *x1*, and outside uses a power-law stretching. The actual stretching factor and the number of cells left and right of *x0* and *x1* are find in a minimization process.

The domain is not completely respected. The starting point of the domain is, but the endpoint of the domain might slightly shift (this is more likely the case for small *nx*, for big *nx* the shift should be small). The new endpoint can be obtained with `domain[0]+np.sum(hx)`. If you want the domain to be respected absolutely, set `resp_domain=True`. However, be aware that this will introduce one stretch-factor which is different from the other stretch factors, to accommodate the restriction. This one-off factor is between the left- and right-side of *x0*, or, if *x1* is provided, just after *x1*.

Parameters

min_width [float] Minimum cell width. If x1 is provided, the actual minimum cell width might be smaller than min_width.

domain [list] [start, end] of model domain.

nx [int] Number of cells.

x0 [float] Center of the grid. *x0* is restricted to *domain*. Default is 0.

x1 [float] If provided, then no stretching is applied between *x0* and *x1*. The non-stretched part starts at *x0* and stops at the first possible location at or after *x1*. *x1* is restricted to *domain*. This will min_width so that an integer number of cells fit within x0 and x1.

resp_domain [bool] If False (default), then the domain-end might shift slightly to assure that the same stretching factor is applied throughout. If set to True, however, the domain is respected absolutely. This will introduce one stretch-factor which is different from the other stretch factors, to accommodate the restriction. This one-off factor is between the left- and right-side of *x0*, or, if *x1* is provided, just after *x1*.

Returns

hx [ndarray] Cell widths of mesh.

See also:

get_hx_x0 Get *hx* and *x0* for a flexible number of *nx* with given bounds.

emg3d.meshes.**get_domain**(*x0=0*, *freq=1*, *res=0.3*, *limits=None*, *min_width=None*, *fact_min=0.2*, *fact_neg=5*, *fact_pos=None*)

Get domain extent and minimum cell width as a function of skin depth.

Returns the extent of the calculation domain and the minimum cell width as a multiple of the skin depth, with possible user restrictions on minimum calculation domain and range of possible minimum cell widths.

$$\delta = 503.3\sqrt{\frac{\rho}{f}},$$
$$x_{\text{start}} = x_0 - k_{\text{neg}}\delta,$$
$$x_{\text{end}} = x_0 + k_{\text{pos}}\delta,$$
$$h_{\text{min}} = k_{\text{min}}\delta.$$

Parameters

x0 [float] Center of the calculation domain. Normally the source location. Default is 0.

freq [float] Frequency (Hz) to calculate the skin depth. The skin depth is a concept defined in the frequency domain. If a negative frequency is provided, it is assumed that the calculation is carried out in the Laplace domain. To calculate the skin depth, the value of *freq* is then multiplied by -2π, to simulate the closest frequency-equivalent.

Default is 1 Hz.

res [float, optional] Resistivity (Ohm m) to calculate skin depth. Default is 0.3 Ohm m (sea water).

limits [None or list] [start, end] of model domain. This extent represents the minimum extent of the domain. The domain is therefore only adjusted if it has to reach outside of [start, end]. Default is None.

min_width [None, float, or list of two floats] Minimum cell width is calculated as a function of skin depth: fact_min*sd. If *min_width* is a float, this is used. If a list of two values [min, max] are provided, they are used to restrain min_width. Default is None.

fact_min, fact_neg, fact_pos [floats] The skin depth is multiplied with these factors to estimate:

- Minimum cell width (*fact_min*, default 0.2)
- Domain-start (*fact_neg*, default 5), and
- Domain-end (*fact_pos*, defaults to *fact_neg*).

Returns

h_min [float] Minimum cell width.

domain [list] Start- and end-points of calculation domain.

emg3d.meshes.**get_hx**(*alpha*, *domain*, *nx*, *x0*, *resp_domain=True*)

Return cell widths for given input.

Find the number of cells left and right of *x0*, *nl* and *nr* respectively, for the provided alpha. For this, we solve

$$\frac{x_{\max} - x_0}{x_0 - x_{\min}} = \frac{a^{nr} - 1}{a^{nl} - 1}$$

where $a = 1 + \alpha$.

Parameters

alpha [float] Stretching factor *a* is given by `a=1+alpha`.

domain [list] [start, end] of model domain.

nx [int] Number of cells.

x0 [float] Center of the grid. *x0* is restricted to *domain*.

resp_domain [bool] If False (default), then the domain-end might shift slightly to assure that the same stretching factor is applied throughout. If set to True, however, the domain is respected absolutely. This will introduce one stretch-factor which is different from the other stretch factors, to accommodate the restriction. This one-off factor is between the left- and right-side of *x0*, or, if *x1* is provided, just after *x1*.

Returns

hx [ndarray] Cell widths of mesh.

5.11.5 `models` – Earth properties

Everything to create model-properties for the multigrid solver.

class emg3d.models.**Model**(*grid*, *res_x=1.0*, *res_y=None*, *res_z=None*, *mu_r=None*, *epsilon_r=None*)

Create a model instance.

Class to provide model parameters (x-, y-, and z-directed resistivities, electric permittivity and magnetic permeability) to the solver. Relative magnetic permeability μ_r is by default set to one and electric permittivity ε_r is by default set to zero, but they can also be provided (isotropically). Keep in mind that the multigrid method as implemented in *emg3d* only works for the diffusive approximation. As soon as the displacement-part in the Maxwell's equations becomes too dominant it will fail (high frequencies or very high electric permittivity).

Parameters

grid [TensorMesh] Grid on which to apply model.

res_x, res_y, res_z [float or ndarray; default to 1.] Resistivity in x-, y-, and z-directions. If ndarray, they must have the shape of grid.vnC (F-ordered) or grid.nC. Resistivities have to be bigger than zero and smaller than infinity.

mu_r [None, float, or ndarray] Relative magnetic permeability (isotropic). If ndarray it must have the shape of grid.vnC (F-ordered) or grid.nC. Default is None, which corresponds to 1., but avoids the calculation of zeta. Magnetic permeability has to be bigger than zero and smaller than infinity.

epsilon_r [None, float, or ndarray] Relative electric permittivity (isotropic). If ndarray it must have the shape of grid.vnC (F-ordered) or grid.nC. The displacement part is completely neglected (diffusive approximation) if set to None, which is the default. Electric permittivity has to be bigger than zero and smaller than infinity.

copy(*self*)

Return a copy of the Model.

epsilon_r
Electric permittivity.

classmethod from_dict(*inp*)
Convert the dictionary into a Model instance.

Parameters

inp [dict] Dictionary as obtained from `Model.to_dict()`. The dictionary needs the keys *res_x*, *res_y*, *res_z*, *mu_r*, *epsilon_r*, and *vnC*.

Returns

obj [`Model` instance]

mu_r
Magnetic permeability.

res_x
Resistivity in x-direction.

res_y
Resistivity in y-direction.

res_z
Resistivity in z-direction.

to_dict(*self*, *copy=False*)
Store the necessary information of the Model in a dict.

class emg3d.models.**VolumeModel**(*grid*, *model*, *sfield*)
Return a volume-averaged version of provided model.

Takes a Model instance and returns the volume averaged values. This is used by the solver internally.

$$\eta_{\{x,y,z\}} = -V \mathrm{i}\omega\mu_0 \left(\rho^{-1}_{\{x,y,z\}} + \mathrm{i}\omega\varepsilon\right)$$

$$\zeta = V \mu_\mathrm{r}^{-1}$$

Parameters

grid [TensorMesh] Grid on which to apply model.

model [Model] Model to transform to volume-averaged values.

sfield [SourceField] A VolumeModel is frequency-dependent. The frequency-information is taken from the provided source filed.

static calculate_eta(*name*, *grid*, *model*, *field*)
eta: volume divided by resistivity.

static calculate_zeta(*name*, *grid*, *model*)
zeta: volume divided by mu_r.

eta_x
eta in x-direction.

eta_y
eta in y-direction.

eta_z
eta in z-direction.

zeta
zeta.

5.11.6 `maps` – Interpolation routines

Interpolation routines mapping grids to grids, grids to fields, and fields to grids.

emg3d.maps.**grid2grid**(*grid*, *values*, *new_grid*, *method='linear'*, *extrapolate=True*, *log=False*)
Interpolate *values* located on *grid* to *new_grid*.

Note 1: The default method is 'linear', because it works with fields and model parameters. However, recommended are 'volume' for model parameters and 'cubic' for fields.

Note 2: For model parameters with *method='volume'* the result is quite different if you provide resistivity, conductivity, or the logarithm of any of the two. The recommended way is to provide the logarithm of resistivity or conductivity, in which case the output of one is indeed the inverse of the output of the other.

Parameters

grid, new_grid [TensorMesh] Input and output model grids; `TensorMesh` instances.

values [ndarray] Model parameters; *emg3d.fields.Field* instance, or a particular field (e.g. field.fx). For fields the method cannot be 'volume'.

method [{<'linear'>, 'volume', 'cubic'}, optional] The method of interpolation to perform. The volume averaging method ensures that the total sum of the property stays constant.

Volume averaging is only implemented for model parameters, not for fields. The method 'cubic' requires at least three points in any direction, otherwise it will fall back to 'linear'.

Default is 'linear', because it works with fields and model parameters. However, recommended are 'volume' for model parameters and 'cubic' for fields.

extrapolate [bool] If True, points on *new_grid* which are outside of *grid* are filled by the nearest value (if `method='cubic'`) or by extrapolation (if `method='linear'`). If False, points outside are set to zero.

For `method='volume'` it always uses the nearest value for points outside of *grid*.

Default is True.

log [bool] If True, the interpolation is carried out on a log10-scale; hence the same as `10**grid2grid(grid, np.log10(values), ...)`. Default is False.

Returns

new_values [ndarray] Values corresponding to *new_grid*.

See also:

`get_receiver` Interpolation of model parameters or fields to (x, y, z).

emg3d.maps.**interp3d**(*points*, *values*, *new_points*, *method*, *fill_value*, *mode*)
Interpolate values in 3D either linearly or with a cubic spline.

Return *values* corresponding to a regular 3D grid defined by *points* on *new_points*.

This is a modified version of `scipy.interpolate.interpn()`, using `scipy.interpolate.RegularGridInterpolator` if `method='linear'` and a custom-wrapped version of `scipy.ndimage.map_coordinates()` if `method='cubic'`. If speed is important then choose 'linear', as it can be significantly faster.

Parameters

points [tuple of ndarray of float, with shapes ((nx,), (ny,) (nz,))] The points defining the regular grid in three dimensions.

values [array_like, shape (nx, ny, nz)] The data on the regular grid in three dimensions.

new_points [tuple (rec_x, rec_y, rec_z)] Coordinates (x, y, z) of new points.

method [{'cubic', 'linear'}, optional] The method of interpolation to perform, 'linear' or 'cubic'. Default is 'cubic' (forced to 'linear' if there are less than 3 points in any direction).

fill_value [float or None] Passed to `scipy.interpolate.RegularGridInterpolator` if `method='linear'`: The value to use for points outside of the interpolation domain. If None, values outside the domain are extrapolated.

mode [{'constant', 'nearest', 'mirror', 'reflect', 'wrap'}] Passed to `scipy.ndimage.map_coordinates()` if `method='cubic'`: Determines how the input array is extended beyond its boundaries.

Returns

new_values [ndarray] Values corresponding to *new_points*.

5.11.7 `fields` – Electric and magnetic fields

Everything related to the multigrid solver that is a field: source field, electric and magnetic fields, and fields at receivers.

class emg3d.fields.**Field**

Create a Field instance with x-, y-, and z-views of the field.

A *Field* is an *ndarray* with additional views of the x-, y-, and z-directed fields as attributes, stored as *fx*, *fy*, and *fz*. The default array contains the whole field, which can be the electric field, the source field, or the residual field, in a 1D array. A *Field* instance has additionally the property *ensure_pec* which, if called, ensures Perfect Electric Conductor (PEC) boundary condition. It also has the two attributes *amp* and *pha* for the amplitude and phase, as common in frequency-domain CSEM.

A *Field* can be initiated in three ways:

1. `Field(grid, dtype=complex)`: Calling it with a `TensorMesh` instance returns a *Field* instance of correct dimensions initiated with zeroes of data type *dtype*.
2. `Field(grid, field)`: Calling it with a `TensorMesh` instance and an *ndarray* returns a *Field* instance of the provided *ndarray*, of same data type.
3. `Field(fx, fy, fz)`: Calling it with three *ndarray*'s which represent the field in x-, y-, and z-direction returns a *Field* instance with these views, of same data type.

Sort-order is 'F'.

Parameters

fx_or_grid [`TensorMesh` or ndarray] Either a TensorMesh instance or an ndarray of shape grid.nEx or grid.vnEx. See explanations above. Only mandatory parameter; if the only one provided, it will initiate a zero-field of *dtype*.

fy_or_field [*Field* or ndarray, optional] Either a Field instance or an ndarray of shape grid.nEy or grid.vnEy. See explanations above.

fz [ndarray, optional] An ndarray of shape grid.nEz or grid.vnEz. See explanations above.

dtype [dtype, optional] Only used if `fy_or_field=None` and `fz=None`; the initiated zero-field for the provided TensorMesh has data type *dtype*. Default: complex.

freq [float, optional] Source frequency (Hz), used to calculate the Laplace parameter s. Either positive or negative:

- *freq* > 0: Frequency domain, hence $s = -\mathrm{i}\omega = -2\mathrm{i}\pi f$ (complex);
- *freq* < 0: Laplace domain, hence $s = f$ (real).

Just added as info if provided.

amp (*self*)
Amplitude of the electromagnetic field.

copy (*self*)
Return a copy of the Field.

ensure_pec
Set Perfect Electric Conductor (PEC) boundary condition.

field
Entire field, 1D [fx, fy, fz].

freq
Return frequency.

classmethod from_dict (*inp*)
Convert dictionary into *Field* instance.

Parameters

inp [dict] Dictionary as obtained from *Field.to_dict()*. The dictionary needs the keys *field*, *freq*, *vnEx*, *vnEy*, and *vnEz*.

Returns

obj [*Field* instance]

fx
View of the x-directed field in the x-direction (nCx, nNy, nNz).

fy
View of the field in the y-direction (nNx, nCy, nNz).

fz
View of the field in the z-direction (nNx, nNy, nCz).

is_electric
Returns True if Field is electric, False if it is magnetic.

pha (*self*, *deg=False*, *unwrap=True*, *lag=True*)
Phase of the electromagnetic field.

Parameters

deg [bool] If True the returned phase is in degrees, else in radians. Default is False (radians).

unwrap [bool] If True the returned phase is unwrapped. Default is True (unwrapped).

lag [bool] If True the returned phase is lag, else lead defined. Default is True (lag defined).

smu0
Return s*mu_0; mu_0 = Magn. permeability of free space [H/m].

sval
Return s; s=iw in frequency domain; s=freq in Laplace domain.

to_dict (*self*, *copy=False*)
Store the necessary information of the Field in a dict.

class emg3d.fields.**SourceField**
Create a Source-Field instance with x-, y-, and z-views of the field.

A subclass of *Field*. Additional properties are the real-valued source vector (*vector*, *vx*, *vy*, *vz*), which sum is always one. For a *SourceField* frequency is a mandatory parameter, unlike for a *Field* (recommended also for *Field* though),

Parameters

fx_or_grid [TensorMesh or ndarray] Either a TensorMesh instance or an ndarray of shape grid.nEx or grid.vnEx. See explanations above. Only mandatory parameter; if the only one provided, it will initiate a zero-field of *dtype*.

fy_or_field [*Field* or ndarray, optional] Either a Field instance or an ndarray of shape grid.nEy or grid.vnEy. See explanations above.

fz [ndarray, optional] An ndarray of shape grid.nEz or grid.vnEz. See explanations above.

dtype [dtype, optional] Only used if `fy_or_field=None` and `fz=None`; the initiated zero-field for the provided TensorMesh has data type *dtype*. Default: complex.

freq [float] Source frequency (Hz), used to calculate the Laplace parameter s. Either positive or negative:

- *freq* > 0: Frequency domain, hence $s = -\mathrm{i}\omega = -2\mathrm{i}\pi f$ (complex);
- *freq* < 0: Laplace domain, hence $s = f$ (real).

In difference to *Field*, the frequency has to be provided for a *SourceField*.

copy(*self*)
Return a copy of the SourceField.

classmethod from_dict(*inp*)
Convert dictionary into *SourceField* instance.

Parameters

inp [dict] Dictionary as obtained from `SourceField.to_dict()`. The dictionary needs the keys *field*, *freq*, *vnEx*, *vnEy*, and *vnEz*.

Returns

obj [*SourceField* instance]

vector
Entire vector, 1D [vx, vy, vz].

vx
View of the x-directed vector in the x-direction (nCx, nNy, nNz).

vy
View of the vector in the y-direction (nNx, nCy, nNz).

vz
View of the vector in the z-direction (nNx, nNy, nCz).

emg3d.fields.**get_source_field**(*grid*, *src*, *freq*, *strength=0*)
Return the source field.

The source field is given in Equation 2 in [Muld06],

$$s\mu_0 \mathbf{J}_\mathrm{s},$$

where $s = \mathrm{i}\omega$. Either finite length dipoles or infinitesimal small point dipoles can be defined, whereas the return source field corresponds to a normalized (1 Am) source distributed within the cell(s) it resides (can be changed with the *strength*-parameter).

The adjoint of the trilinear interpolation is used to distribute the point(s) to the grid edges, which corresponds to the discretization of a Dirac ([PlDM07]).

Parameters

grid [TensorMesh] Model grid; a `TensorMesh` instance.

src [list of floats] Source coordinates (m). There are two formats:

- Finite length dipole: `[x0, x1, y0, y1, z0, z1]`.
- Point dipole: `[x, y, z, azimuth, dip]`.

freq [float] Source frequency (Hz), used to calculate the Laplace parameter s. Either positive or negative:

- *freq* > 0: Frequency domain, hence $s = -\mathrm{i}\omega = -2\mathrm{i}\pi f$ (complex);
- *freq* < 0: Laplace domain, hence $s = f$ (real).

strength [float or complex, optional] Source strength (A):

- If 0, output is normalized to a source of 1 m length, and source strength of 1 A.
- If != 0, output is returned for given source length and strength.

Default is 0.

Returns

sfield [*SourceField()* instance] Source field, normalized to 1 A m.

emg3d.fields.**get_receiver**(*grid*, *values*, *coordinates*, *method='cubic'*, *extrapolate=False*)

Return values corresponding to grid at coordinates.

Works for electric fields as well as magnetic fields obtained with *get_h_field()*, and for model parameters.

Parameters

grid [TensorMesh] Model grid; a `TensorMesh` instance.

values [ndarray] Field instance, or a particular field (e.g. field.fx); Model parameters.

coordinates [tuple (x, y, z)] Coordinates (x, y, z) where to interpolate *values*; e.g. receiver locations.

method [str, optional] The method of interpolation to perform, 'linear' or 'cubic'. Default is 'cubic' (forced to 'linear' if there are less than 3 points in any direction).

extrapolate [bool] If True, points on *new_grid* which are outside of *grid* are filled by the nearest value (if `method='cubic'`) or by extrapolation (if `method='linear'`). If False, points outside are set to zero.

Default is False.

Returns

new_values [ndarray or `empymod.utils.EMArray`] Values at *coordinates*.

If input was a field it returns an EMArray, which is a subclassed ndarray with `.pha` and `.amp` attributes.

If input was an entire Field instance, output is a tuple (fx, fy, fz).

See also:

grid2grid Interpolation of model parameters or fields to a new grid.

emg3d.fields.**get_h_field**(*grid*, *model*, *field*)

Return magnetic field corresponding to provided electric field.

Retrieve the magnetic field $\mathbf{H}$ from the electric field $\mathbf{E}$ using Farady's law, given by

$$\nabla \times \mathbf{E} = \mathrm{i}\omega\mu\mathbf{H}.$$

Note that the magnetic field in x-direction is defined in the center of the face defined by the electric field in y- and z-directions, and similar for the other field directions. This means that the provided electric field and the returned magnetic field have different dimensions:

```
E-field:  x: [grid.vectorCCx,  grid.vectorNy,  grid.vectorNz]
          y: [ grid.vectorNx, grid.vectorCCy,  grid.vectorNz]
          z: [ grid.vectorNx,  grid.vectorNy, grid.vectorCCz]

H-field:  x: [ grid.vectorNx, grid.vectorCCy, grid.vectorCCz]
          y: [grid.vectorCCx,  grid.vectorNy, grid.vectorCCz]
          z: [grid.vectorCCx, grid.vectorCCy,  grid.vectorNz]
```

Parameters

grid [TensorMesh] Model grid; `TensorMesh` instance.

model [Model] Model; `Model` instance.

field [Field] Electric field; *`Field`* instance.

Returns

hfield [Field] Magnetic field; *`Field`* instance.

5.11.8 `io` – I/O utilities

Utility functions for writing and reading data.

`emg3d.io.`**`save`**(*fname*, *backend='h5py'*, *compression='gzip'*, ***kwargs*)

Save meshes, models, fields, and other data to disk.

Serialize and save *`emg3d.meshes.TensorMesh`*, *`emg3d.fields.Field`*, and *`emg3d.models.Model`* instances and add arbitrary other data, where instances of the same type are grouped together.

The serialized instances will be de-serialized if loaded with *`load()`*.

Parameters

fname [str] File name.

backend [str, optional] Backend to use. Implemented are currently:

- *h5py* (default): Uses *h5py* to store inputs to a hierarchical, compressed binary hdf5 file with the extension '.h5'. Recommended and default backend, but requires the module *h5py*. Use *numpy* if you don't want to install *h5py*.
- *numpy*: Uses *numpy* to store inputs to a flat, compressed binary file with the extension '.npz'.

compression [int or str, optional] Passed through to h5py, default is 'gzip'.

kwargs [Keyword arguments, optional] Data to save using its key as name. The following instances will be properly serialized: *`emg3d.meshes.TensorMesh`*, *`emg3d.fields.Field`*, and *`emg3d.models.Model`* and serialized again if loaded with *`load()`*. These instances are collected in their own group if h5py is used.

`emg3d.io.`**`load`**(*fname*, ***kwargs*)

Load meshes, models, fields, and other data from disk.

Load and de-serialize *`emg3d.meshes.TensorMesh`*, *`emg3d.fields.Field`*, and *`emg3d.models.Model`* instances and add arbitrary other data that were saved with *`save()`*.

Parameters

fname [str] File name including extension. Used backend depends on the file extensions:

- '.npz': numpy-binary
- '.h5': h5py-binary (needs *h5py*)

verb [int] If 1 (default) verbose, if 0 silent.

Returns

out [dict] A dictionary containing the data stored in fname; *emg3d.meshes.TensorMesh*, *emg3d.fields.Field*, and *emg3d.models.Model* instances are de-serialized and returned as instances.

Bibliography

[ArFW00] Arnold, D. N., R. S. Falk, and R. Winther, 2000, Multigrid in H(div) and H(curl): Numerische Mathematik, 85, 197–217; DOI: 10.1007/PL00005386.

[BrHM00] Briggs, W., V. Henson, and S. McCormick, 2000, A Multigrid Tutorial, Second Edition: Society for Industrial and Applied Mathematics; DOI: 10.1137/1.9780898719505.

[ClWe01] Clemens, M., and T. Weiland, 2001, Discrete electromagnetism with the finite integration technique: PIER, 32, 65–87; DOI: 10.2528/PIER00080103.

[Fedo64] Fedorenko, R. P., 1964, The speed of convergence of one iterative process: USSR Computational Mathematics and Mathematical Physics, 4, 227–235; DOI 10.1016/0041-5553(64)90253-8.

[JoOM06] Jönsthövel, T. B., C. W. Oosterlee, and W. A. Mulder, 2006, Improving multigrid for 3-D electromagnetic diffusion on stretched grids: European Conference on Computational Fluid Dynamics; UUID: df65da5c-e43f-47ab-b80d-2f8ee7f35464.

[Hack85] Hackbusch, W., 1985, Multi-grid methods and applications: Springer, Berlin, Heidelberg, Volume 4 of Springer Series in Computational Mathematics; DOI: 10.1007/978- 3-662-02427-0.

[MoSu94] Monk, P., and E. Süli, 1994, A convergence analysis of Yee's scheme on nonuniform grids: SIAM Journal on Numerical Analysis, 31, 393–412; DOI 10.1137/0731021.

[Muld06] Mulder, W. A., 2006, A multigrid solver for 3D electromagnetic diffusion: Geophysical Prospecting, 54, 633–649; DOI: 10.1111/j.1365-2478.2006.00558.x.

[Muld07] Mulder, W. A., 2007, A robust solver for CSEM modelling on stretched grids: EAGE Technical Program Expanded Abstracts, D036; DOI 10.3997/2214-4609.201401567.

[Muld08] Mulder, W. A., 2008, Geophysical modelling of 3D electromagnetic diffusion with multigrid: Computing and Visualization in Science, 11, 29–138; DOI: 10.1007/s00791-007-0064-y.

[Muld11] Mulder, W. A., 2011, *in* Numerical Methods, Multigrid: Springer Netherlands, 895–900; DOI 10.1007/978-90-481-8702-7_153.

[MuWS08] Mulder, W. A., M. Wirianto, and E. C. Slob, 2008, Time-domain modeling of electromagnetic diffusion with a frequency-domain code: Geophysics, 73, F1–F8; DOI: 10.1190/1.2799093.

[PlDM07] Plessix, R.-E., M. Darnet, and W. A. Mulder, 2007, An approach for 3D multisource, multifrequency CSEM modeling: Geophysics, 72, SM177–SM184; DOI: 10.1190/1.2744234.

[SlHM10] Slob, E., J. Hunziker, and W. A. Mulder, 2010, Green's tensors for the diffusive electric field in a VTI half-space: PIER, 107, 1–20: DOI: 10.2528/PIER10052807.

[Weil77] Weiland, T., 1977, Eine Methode zur Lösung der Maxwellschen Gleichungen für sechskomponentige Felder auf diskreter Basis: Archiv für Elektronik und Übertragungstechnik, 31, 116–120; pdf: leibniz-publik.de/de/fs1/object/display/bsb00064886_00001.html.

[WeMS19] Werthmüller, D., W. A. Mulder, and E. C. Slob, 2019, emg3d: A multigrid solver for 3D electromagnetic diffusion: Journal of Open Source Software, 4(39), 1463; DOI: 10.21105/joss.01463.

[Wess91] Wesseling, P., 1991, An introduction to multigrid methods: John Wiley & Sons. Pure and Applied Mathematics; ISBN: 0-471-93083-0.

[WiMS10] Wirianto, M., W. A. Mulder, and E. C. Slob, 2010, A feasibility study of land CSEM reservoir monitoring in a complex 3-D model: Geophysical Journal International, **181**, 741–755; DOI: 10.1111/j.1365-246X.2010.04544.x.

[WiMS11] Wirianto, M., W. A. Mulder, and E. C. Slob, 2011, Applying essentially non-oscillatory interpolation to controlled-source electromagnetic modelling: Geophysical Prospecting, 59, 161–175; DOI: 10.1111/j.1365-2478.2010.00899.x.

[Yee66] Yee, K., 1966, Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media: IEEE Transactions on Antennas and Propagation, 14, 302–307; DOI: 10.1109/TAP.1966.1138693.

Python Module Index

e

Index

G

I

K

L

M

N

O

P

R

S

T

V

Z