empymod Documentation

Release 2.0.0

The empymod Developers

29 April 2020

User Manual

Version: 2.0.0 ~ Date: 29 April 2020

The electromagnetic modeller **empymod** can model electric or magnetic responses due to a three-dimensional electric or magnetic source in a layered-earth model with vertical transverse isotropic (VTI) resistivity, VTI electric permittivity, and VTI magnetic permeability, from very low frequencies (DC) to very high frequencies (GPR). The computation is carried out in the wavenumber-frequency domain, and various Hankel- and Fourier-transform methods are included to transform the responses into the space-frequency and space-time domains.

CHAPTER 1

More information

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

- **Website**: https://empymod.github.io
- **Documentation**: https://empymod.readthedocs.io
- **Source Code**: https://github.com/empymod
- **Examples**: https://empymod.readthedocs.io/en/stable/examples

CHAPTER 2

Features

- Computes the complete (diffusion and wave phenomena) 3D electromagnetic field in a layered-earth model including vertical transverse isotropic (VTI) resistivity, VTI electric permittivity, and VTI magnetic permeability, for electric and magnetic sources as well as electric and magnetic receivers.
- Modelling routines:
  - `bipole`: arbitrary oriented, finite length bipoles with given source strength; space-frequency and space-time domains.
  - `dipole`: infinitesimal small dipoles oriented along the principal axes, normalized field; space-frequency and space-time domains.
  - `loop`: arbitrary oriented loop source measured by arbitrary oriented, finite length electric or magnetic dipole or loop receivers; space-frequency and space-time domains.
  - `dipole_k`: as `dipole`, but returns the wavenumber-frequency domain response.
  - `gpr`: computes the ground-penetrating radar response for given central frequency, using a Ricker wavelet (experimental).
  - `analytical`: interface to the analytical, space-frequency and space-time domain solutions.
- Hankel transforms (wavenumber-frequency to space-frequency transform):
  - DLF: Digital Linear Filters (using included filters or providing own ones)
  - QWE: Quadrature with extrapolation
  - QUAD: Adaptive quadrature
- Fourier transforms (space-frequency to space-time transform):
  - DLF: Digital Linear Filters (using included filters or providing own ones)
  - QWE: Quadrature with extrapolation
  - FFTLog: Logarithmic Fast Fourier Transform
  - FFT: Fast Fourier Transform
- Analytical, space-frequency and space-time domain solutions:
  - Complete full-space (electric and magnetic sources and receivers); space-frequency domain
  - Diffusive half-space (electric sources and receivers); space-frequency and space-time domains:

        * Direct wave (= diffusive full-space solution)
        * Reflected wave
        * Airwave (semi-analytical in the case of step responses)

- Add-ons (`empymod.scripts`):

  The add-ons for empymod provide some very specific, additional functionalities:

  - `tmtemod`: Return up- and down-going TM/TE-mode contributions for x-directed electric sources and receivers, which are located in the same layer.
  - `fdesign`: Design digital linear filters for the Hankel and Fourier transforms.

- Hidden features (incomplete list, see manual for more):
  - Models with frequency-dependent resistivity (e.g., Cole-Cole IP).
  - Space-Laplace domain computation for the numerical and analytical solutions.

CHAPTER 3

Installation

You can install empymod either via **conda**:

```
conda install -c conda-forge empymod
```

or via **pip**:

```
pip install empymod
```

Required are Python version 3.6 or higher and the modules NumPy, SciPy, and Numba. 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 empymod, please give credit by citing Werthmüller (2017):

> Werthmüller, D., 2017, An open-source full 3D electromagnetic modeler for 1D VTI media in Python: empymod: Geophysics, 82(6), WB9-WB19; DOI: 10.1190/geo2016-0626.1.

All releases have a Zenodo-DOI, which can be found on 10.5281/zenodo.593094. Also consider citing Hunziker et al. (2015) and Key (2012), without which empymod would not exist.

CHAPTER 5

License information

5.1 Getting started

5.1.1 Installation

You can install empymod either via **conda**:

```
conda install -c conda-forge empymod
```

or via **pip**:

```
pip install empymod
```

Required are Python version 3.6 or higher and the modules NumPy, SciPy, and Numba.

The modeller empymod comes with add-ons (`empymod.scripts`). These add-ons provide some very specific, additional functionalities. Some of these add-ons have additional, optional dependencies such as matplotlib. See the *Add-ons*-section for their documentation.

If you are new to Python I recommend using a Python distribution, which will ensure that all dependencies are met, specifically properly compiled versions of NumPy and SciPy; I recommend using Anaconda. If you install Anaconda you can simply start the *Anaconda Navigator*, add the channel **conda-forge** and **empymod** 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.

The structure of empymod is:

- **model.py**: EM modelling; principal end-user facing routines.
- **utils.py**: Utilities such as checking input parameters.
- **kernel.py**: Kernel of empymod, computes the wavenumber-domain electromagnetic response. Plus analytical, frequency-domain full- and half-space solutions.
- **transform.py**: Methods to carry out the required Hankel transform from wavenumber to space domain and Fourier transform from frequency to time domain.
- **filters.py**: Filters for the *Digital Linear Filters* method DLF (Hankel and Fourier transforms).

Note: Until v2 empymod did not use Numba but instead optionally NumExpr. Use **v1.10.x** if you cannot use Numba or want to use NumExpr. However, no new features will land in v1, only bugfixes.

5.1.2 Usage

The main modelling routine is *empymod.model.bipole()*, which can compute the electromagnetic frequency- or time-domain field due to arbitrary finite electric or magnetic bipole sources, measured by arbitrary finite electric or magnetic bipole receivers. The model is defined by horizontal resistivity and anisotropy, horizontal and vertical electric permittivities and horizontal and vertical magnetic permeabilities. By default, the electromagnetic response is normalized to source and receiver of 1 m length, and source strength of 1 A.

A simple frequency-domain example, with most of the parameters left at the default value:

```
>>> import empymod
>>> import numpy as np
>>> # x-directed bipole source: x0, x1, y0, y1, z0, z1
>>> src = [-50, 50, 0, 0, -100, -100]
>>> # x-directed dipole source-array: x, y, z, azimuth, dip
>>> rec = [np.arange(1, 11)*500, np.zeros(10), -200, 0, 0]
>>> # layer boundaries
>>> depth = [0, -300, -1000, -1050]
>>> # layer resistivities
>>> res = [1e20, .3, 1, 50, 1]
>>> # Frequency
>>> freq = 1
>>> # Compute the electric field due to an electric source at 1 Hz.
>>> # [msrc = mrec = True (default)]
>>> EMfield = empymod.bipole(src, rec, depth, res, freq, verb=4)
~
:: empymod START  ::  v2.0.0
~
   depth       [m] :  -1050, -1000, -300, 0
   res     [Ohm.m] :  1 50 1 0.3 1E+20
   aniso       [-] :  1 1 1 1 1
   epermH      [-] :  1 1 1 1 1
   epermV      [-] :  1 1 1 1 1
   mpermH      [-] :  1 1 1 1 1
   mpermV      [-] :  1 1 1 1 1
   direct field    :  Comp. in wavenumber domain
   frequency  [Hz] :  1
   Hankel          :  DLF (Fast Hankel Transform)
     > Filter      :  Key 201 (2009)
     > DLF type    :  Standard
   Loop over       :  None (all vectorized)
   Source(s)       :  1 bipole(s)
     > intpts      :  1 (as dipole)
     > length  [m] :  100
     > strength[A] :  0
     > x_c     [m] :  0
```

(continues on next page)

(continued from previous page)

```
    > y_c      [m] :  0
    > z_c      [m] :  -100
    > azimuth  [°] :  0
    > dip      [°] :  0
  Receiver(s)      :  10 dipole(s)
    > x        [m] :  500 - 5000 : 10  [min-max; #]
                   :  500 1000 1500 2000 2500 3000 3500 4000 4500 5000
    > y        [m] :  0 - 0 : 10  [min-max; #]
                   :  0 0 0 0 0 0 0 0 0 0
    > z        [m] :  -200
    > azimuth  [°] :  0
    > dip      [°] :  0
  Required ab's    :  11
~
:: empymod END; runtime = 0:00:00.005536 :: 1 kernel call(s)
~
>>> print(EMfield)
[  1.68809346e-10 -3.08303130e-10j  -8.77189179e-12 -3.76920235e-11j
  -3.46654704e-12 -4.87133683e-12j  -3.60159726e-13 -1.12434417e-12j
   1.87807271e-13 -6.21669759e-13j   1.97200208e-13 -4.38210489e-13j
   1.44134842e-13 -3.17505260e-13j   9.92770406e-14 -2.33950871e-13j
   6.75287598e-14 -1.74922886e-13j   4.62724887e-14 -1.32266600e-13j]
```

A good starting point is the *Examples*-gallery or [Wert17b], and more detailed information can be found in [Wert17]. The description of all parameters can be found in the API documentation for `empymod.model.bipole()`.

5.1.3 Coordinate system

The used coordinate system is either a

- Left-Handed System (LHS), where Easting is the x-direction, Northing the y-direction, and positive z is pointing downwards;
- Right-Handed System (RHS), where Easting is the x-direction, Northing the y-direction, and positive z is pointing upwards.

Have a look at the example *Coordinate system* for further explanations.

5.1.4 Theory

The code is principally based on

- [HuTS15] for the wavenumber-domain computation (`kernel`),
- [Key12] for the DLF and QWE transforms,
- [SlHM10] for the analytical half-space solutions, and
- [Hami00] for the FFTLog.

See these publications and all the others given in the *References*, if you are interested in the theory on which empymod is based. Another good reference is [ZiSl19]. The book derives in great detail the equations for layered-Earth CSEM modelling.

5.1.5 Contributing

New contributions, bug reports, or any kind of feedback is always welcomed! Have a look at the Projects on GitHub to get an idea of things that could be implemented. The best way for interaction is at https://github.

com/empymod. If you prefer to contact me outside of GitHub use the contact form on my personal website, https://werthmuller.org.

To install empymod 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.

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 `empymod`-top-directory.

```
> pip install pytest coveralls pytest-flake8 pytest-mpl
> # and then
> cd to/the/empymod/folder  # Ensure you are in the right directory,
> ls -d */                  # your output should look the same.
docs/  empymod/  examples/  tests/
> # pytest will find the tests, which are located in the tests-folder.
> # simply run
> pytest --cov=empymod --flake8 --mpl
```

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

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

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

5.1.7 License

5.2 Transforms

Included **Hankel transforms**:

- DLF: Digital Linear Filters
- QWE: Quadrature with Extrapolation
- QUAD: Adaptive quadrature

Included **Fourier transforms**:

- DLF: Digital Linear Filters

- QWE: Quadrature with Extrapolation
- FFTLog: Logarithmic Fast Fourier Transform
- FFT: Fast Fourier Transform

5.2.1 Digital Linear Filters

The module *empymod.filters* comes with many DLFs for the Hankel and the Fourier transform. If you want to export one of these filters to plain ASCII files you can use the `tofile`-routine of each filter:

```
>>> import empymod
>>> # Load a filter
>>> filt = empymod.filters.wer_201_2018()
>>> # Save it to pure ASCII-files
>>> filt.tofile()
>>> # This will save the following three files:
>>> #    ./filters/wer_201_2018_base.txt
>>> #    ./filters/wer_201_2018_j0.txt
>>> #    ./filters/wer_201_2018_j1.txt
```

Similarly, if you want to use an own filter you can do that as well. The filter base and the filter coefficient have to be stored in separate files:

```
>>> import empymod
>>> # Create an empty filter;
>>> # Name has to be the base of the text files
>>> filt = empymod.filters.DigitalFilter('my-filter')
>>> # Load the ASCII-files
>>> filt.fromfile()
>>> # This will load the following three files:
>>> #    ./filters/my-filter_base.txt
>>> #    ./filters/my-filter_j0.txt
>>> #    ./filters/my-filter_j1.txt
>>> # and store them in filt.base, filt.j0, and filt.j1.
```

The path can be adjusted by providing `tofile` and `fromfile` with a `path`-argument.

5.2.2 FFTLog

FFTLog is the logarithmic analogue to the Fast Fourier Transform FFT originally proposed by [Talm78]. The code used by `empymod` was published in Appendix B of [Hami00] and is publicly available at casa.colorado.edu/~ajsh/FFTLog. From the `FFTLog`-website:

FFTLog is a set of fortran subroutines that compute the fast Fourier or Hankel (= Fourier-Bessel) transform of a periodic sequence of logarithmically spaced points.

FFTlog can be used for the Hankel as well as for the Fourier Transform, but currently `empymod` uses it only for the Fourier transform. It uses a simplified version of the python implementation of FFTLog, `pyfftlog` (github.com/prisae/pyfftlog).

[HaJo88] proposed a logarithmic Fourier transform (abbreviated by the authors as LFT) for electromagnetic geophysics, also based on [Talm78]. I do not know if Hamilton was aware of the work by Haines and Jones. The two publications share as reference only the original paper by Talman, and both cite a publication of Anderson; Hamilton cites [Ande82], and Haines and Jones cite [Ande79]. Hamilton probably never heard of Haines and Jones, as he works in astronomy, and Haines and Jones was published in the *Geophysical Journal*.

Logarithmic FFTs are not widely used in electromagnetics, as far as I know, probably because of the ease, speed, and generally sufficient precision of the digital filter methods with sine and cosine transforms ([Ande75]). However, comparisons show that FFTLog can be faster and more precise than digital filters, specifically for responses

with source and receiver at the interface between air and subsurface. Credit to use FFTLog in electromagnetics goes to David Taylor who, in the mid-2000s, implemented FFTLog into the forward modellers of the company Multi-Transient ElectroMagnetic (MTEM Ltd, later Petroleum Geo-Services PGS). The implementation was driven by land responses, where FFTLog can be much more precise than the filter method for very early times.

5.2.3 Notes on Fourier Transform

The Fourier transform to obtain the space-time domain impulse response from the complex-valued space-frequency response can be computed by either a cosine transform with the real values, or a sine transform with the imaginary part,

$$\begin{aligned} E(r,t)^{\text{Impulse}} &= \frac{2}{\pi}\int_0^\infty \Re[E(r,\omega)]\ \cos(\omega t)\ \mathrm{d}\omega\ , \\ &= -\frac{2}{\pi}\int_0^\infty \Im[E(r,\omega)]\ \sin(\omega t)\ \mathrm{d}\omega\ , \end{aligned} \tag{5.1}$$

see, e.g., [Ande75] or [Key12]. Quadrature-with-extrapolation, FFTLog, and obviously the sine/cosine-transform all make use of this split.

To obtain the step-on response the frequency-domain result is first divided by $\mathrm{i}\omega$, in the case of the step-off response it is additionally multiplied by -1. The impulse-response is the time-derivative of the step-response,

$$E(r,t)^{\text{Impulse}} = \frac{\partial\, E(r,t)^{\text{step}}}{\partial t}\ . \tag{5.2}$$

Using $\frac{\partial}{\partial t} \Leftrightarrow \mathrm{i}\omega$ and going the other way, from impulse to step, leads to the divison by $\mathrm{i}\omega$. This only holds because we define in accordance with the causality principle that $E(r, t \leq 0) = 0$.

With the sine/cosine transform (`ft='dlf'/'sin'/'cos'`) you can choose which one you want for the impulse responses. For the switch-on response, however, the sine-transform is enforced, and equally the cosine transform for the switch-off response. This is because these two do not need to now the field at time 0, $E(r, t = 0)$.

The Quadrature-with-extrapolation and FFTLog are hard-coded to use the cosine transform for step-off responses, and the sine transform for impulse and step-on responses. The FFT uses the full complex-valued response at the moment.

For completeness sake, the step-on response is given by

$$E(r,t)^{\text{Step-on}} = -\frac{2}{\pi}\int_0^\infty \Im\left[\frac{E(r,\omega)}{\mathrm{i}\omega}\right]\ \sin(\omega t)\ \mathrm{d}\omega\ , \tag{5.3}$$

and the step-off by

$$E(r,t)^{\text{Step-off}} = -\frac{2}{\pi}\int_0^\infty \Re\left[\frac{E(r,\omega)}{\mathrm{i}\omega}\right]\ \cos(\omega t)\ \mathrm{d}\omega\ . \tag{5.4}$$

5.2.4 Laplace domain

It is also possible to compute the response in the **Laplace domain**, by using a real value for s instead of the complex value $\mathrm{i}\omega$. This simplifies the problem from complex numbers to real numbers. However, the transform from Laplace-to-time domain is not as robust as the transform from frequency-to-time domain, and is currently not implemented in `empymod`. To compute Laplace-domain responses instead of frequency-domain responses simply provide negative frequency values. If all provided frequencies f are negative then s is set to $-f$ instead of the frequency-domain $s = 2\mathrm{i}\pi f$.

5.3 Tips and tricks

There is the usual trade-off between speed, memory, and accuracy. Very generally speaking we can say that the *DLF* is faster than *QWE*, but *QWE* is much easier on memory usage. *QWE* allows you to control the accuracy. A

standard quadrature in the form of *QUAD* is also provided. *QUAD* is generally orders of magnitudes slower, and more fragile depending on the input arguments. However, it can provide accurate results where *DLF* and *QWE* fail.

5.3.1 Memory

By default `empymod` will try to carry out the computation in one go, without looping. If your model has many offsets and many frequencies this can be heavy on memory usage. Even more so if you are computing time-domain responses for many times. If you are running out of memory, you should use either `loop='off'` or `loop='freq'` to loop over offsets or frequencies, respectively. Use `verb=3` to see how many offsets and how many frequencies are computed internally.

5.3.2 Depths, Rotation, and Bipole

Depths: Computation of many source and receiver positions is fastest if they remain at the same depth, as they can be computed in one kernel call. If depths do change, one has to loop over them. Note: Sources or receivers placed on a layer interface are considered in the upper layer.

Rotation: Sources and receivers aligned along the principal axes x, y, and z can be computed in one kernel call. For arbitrary oriented di- or bipoles, 3 kernel calls are required. If source and receiver are arbitrary oriented, 9 (3x3) kernel calls are required.

Bipole: Bipoles increase the computation time by the amount of integration points used. For a source and a receiver bipole with each 5 integration points you need 25 (5x5) kernel calls. You can compute it in 1 kernel call if you set both integration points to 1, and therefore compute the bipole as if they were dipoles at their centre.

Example: For 1 source and 10 receivers, all at the same depth, 1 kernel call is required. If all receivers are at different depths, 10 kernel calls are required. If you make source and receivers bipoles with 5 integration points, 250 kernel calls are required. If you rotate the source arbitrary horizontally, 500 kernel calls are required. If you rotate the receivers too, in the horizontal plane, 1‘000 kernel calls are required. If you rotate the receivers also vertically, 1‘500 kernel calls are required. If you rotate the source vertically too, 2‘250 kernel calls are required. So your computation will take 2‘250 times longer! No matter how fast the kernel is, this will take a long time. Therefore carefully plan how precise you want to define your source and receiver bipoles.

Table 1: Example as a table for comparison: 1 source, 10 receiver (one or many frequencies).

	source bipole			receiver bipole			
kernel calls	intpts	azimuth	dip	intpts	azimuth	dip	diff. z
1	1	0/90	0/90	1	0/90	0/90	1
10	1	0/90	0/90	1	0/90	0/90	10
250	5	0/90	0/90	5	0/90	0/90	10
500	5	arb.	0/90	5	0/90	0/90	10
1000	5	arb.	0/90	5	arb.	0/90	10
1500	5	arb.	0/90	5	arb.	arb.	10
2250	5	arb.	arb.	5	arb.	arb.	10

5.3.3 Lagged Convolution and Splined Transforms

Both Hankel and Fourier DLF have three options, which can be controlled via the `htarg['pts_per_dec']` and `ftarg['pts_per_dec']` parameters:

- `pts_per_dec=0` : *Standard DLF*;
- `pts_per_dec<0` : *Lagged Convolution DLF*: Spacing defined by filter base, interpolation is carried out in the input domain;
- `pts_per_dec>0` : *Splined DLF*: Spacing defined by `pts_per_dec`, interpolation is carried out in the output domain.

Similarly, interpolation can be used for `QWE` by setting `pts_per_dec` to a value bigger than 0.

The Lagged Convolution and Splined options should be used with caution, as they use interpolation and are therefore less precise than the standard version. However, they can significantly speed up *QWE*, and massively speed up *DLF*. Additionally, the interpolated versions minimizes memory requirements a lot. Speed-up is greater if all source-receiver angles are identical. Note that setting `pts_per_dec` to something else than 0 to compute only one offset (Hankel) or only one time (Fourier) will be slower than using the standard version.

QWE: Good speed-up is also achieved for *QWE* by setting `maxint` as low as possible. Also, the higher `nquad` is, the higher the speed-up will be.

DLF: Big improvements are achieved for long DLF-filters and for many offsets/frequencies (thousands).

Warning: Keep in mind that setting `pts_per_dec` to something else than 0 uses interpolation, and is therefore not as accurate as the standard version. Use with caution and always compare with the standard version to verify if you can apply interpolation to your problem at hand!

Be aware that *QUAD* (Hankel transform) *always* use the splined version and *always* loops over offsets. The Fourier transforms *FFTlog*, *QWE*, and *FFT* always use interpolation too, either in the frequency or in the time domain.

The splined versions of *QWE* check whether the ratio of any two adjacent intervals is above a certain threshold (steep end of the wavenumber or frequency spectrum). If it is, it carries out *QUAD* for this interval instead of *QWE*. The threshold is stored in `diff_quad`, which can be changed within the parameter `htarg` and `ftarg`.

For a graphical explanation of the differences between standard DLF, lagged convolution DLF, and splined DLF for the Hankel and the Fourier transforms see the example *Digital Linear Filters*.

5.3.4 Looping

By default, you can compute many offsets and many frequencies all in one go, vectorized (for the *DLF*), which is the default. The `loop` parameter gives you the possibility to force looping over frequencies or offsets. This parameter can have severe effects on both runtime and memory usage. Play around with this factor to find the fastest version for your problem at hand. It ALWAYS loops over frequencies if `ht = 'QWE'/'QUAD'` or if `ht = 'DLF'` and `pts_per_dec!=0` (Lagged Convolution or Splined Hankel DLF). All vectorized is very fast if there are few offsets or few frequencies. If there are many offsets and many frequencies, looping over the smaller of the two will be faster. Choosing the right looping can have a significant influence.

5.3.5 Vertical components and `xdirect`

Computing the direct field in the wavenumber-frequency domain (`xdirect=False`; the default) is generally faster than computing it in the frequency-space domain (`xdirect=True`).

However, using `xdirect = True` can improve the result (if source and receiver are in the same layer) to compute:

- the vertical electric field due to a vertical electric source,
- configurations that involve vertical magnetic components (source or receiver),
- all configurations when source and receiver depth are exactly the same.

The Hankel transforms methods are having sometimes difficulties transforming these functions.

5.3.6 Time-domain land CSEM

The derivation, as it stands, has a near-singular behaviour in the wavenumber-frequency domain when $\kappa^2 = \omega^2\epsilon\mu$. This can be a problem for land-domain CSEM computations if source and receiver are located at the surface between air and subsurface. Because most transforms do not sample the wavenumber-frequency domain sufficiently to catch this near-singular behaviour (hence not smooth), which then creates noise at early times where

the signal should be zero. To avoid the issue simply set the relative electric permittivity (`epermH`, `epermV`) of the air to zero. This trick obviously uses the diffusive approximation for the air-layer, it therefore will not work for very high frequencies (e.g., GPR computations). An example is given in *Improve land CSEM computation*.

This trick works fine for all horizontal components, but not so much for the vertical component. But then it is not feasible to have a vertical source or receiver *exactly* at the surface. A few tips for these cases: The receiver can be put pretty close to the surface (a few millimeters), but the source has to be put down a meter or two, more for the case of vertical source AND receiver, less for vertical source OR receiver. The results are generally better if the source is put deeper than the receiver. In either case, the best is to first test the survey layout against the analytical result (using `empymod.analytical` with `solution='dhs'`) for a half-space, and subsequently model more complex cases.

A common alternative to this trick is to apply a lowpass filter to filter out the unstable high frequencies.

5.3.7 Hook for user-defined computation of η and ζ

In principal it is always best to write your own modelling routine if you want to adjust something. Just copy `empymod.dipole` or `empymod.bipole` as a template, and modify it to your needs. Since `empymod v1.7.4`, however, there is a hook which allows you to modify η_h, η_v, ζ_h, and ζ_v quite easily.

The trick is to provide a dictionary (we name it `inp` here) instead of the resistivity vector in `res`. This dictionary, `inp`, has two mandatory plus optional entries:

- `res`: the resistivity vector you would have provided normally (mandatory).
- A function name, which has to be either or both of (mandatory)
  - `func_eta`: To adjust `etaH` and `etaV`, or
  - `func_zeta`: to adjust `zetaH` and `zetaV`.
- In addition, you have to provide all parameters you use in `func_eta`/`func_zeta` and are not already provided to `empymod`. All additional parameters must have #layers elements.

The functions `func_eta` and `func_zeta` must have the following characteristics:

- The signature is `func(inp, p_dict)`, where
  - `inp` is the dictionary you provide, and
  - `p_dict` is a dictionary that contains all parameters so far computed in empymod [`locals()`].
- It must return `etaH, etaV` if `func_eta`, or `zetaH, zetaV` if `func_zeta`.

Dummy example

```
def my_new_eta(inp, p_dict):
    # Your computations, using the parameters you provided
    # in `inp` and the parameters from empymod in `p_dict`.
    # In the example line below, we provide, e.g.,  inp['tau']
    return etaH, etaV
```

And then you call `empymod` with `res={'res': res-array, 'tau': tau, 'func_eta': my_new_eta}`.

Have a look at the corresponding example in the Gallery, where this hook is exploited in the low-frequency range to use the Cole-Cole model for IP computation. It could also be used in the high-frequency range to model dielectricity.

5.4 Examples

5.4.1 Coordinate system

Short version

The used coordinate system is either a

- Left-Handed System (LHS), where Easting is the x-direction, Northing the y-direction, and positive z is pointing downwards;
- Right-Handed System (RHS), where Easting is the x-direction, Northing the y-direction, and positive z is pointing upwards.

In more detail

The derivation on which `empymod` is based ([HuTS15]) uses a right-handed system with x to the East, y to the South, and z downwards (ESD). In the actual original implementation of `empymod` this was changed to a left-handed system with x to the East, y to the North, and z downwards (END). However, `empymod` can equally well be used for a coordinate system where positive z is pointing up by just flipping z, resulting in x to the East, y to the North, and z upwards (ENU).

	Left-handed system	Right-handed system
x	Easting	Easting
y	Northing	Northing
z	Down	Up
θ	Angle E-N	Angle E-N
φ	Angle down	Angle up

There are a few other important points to keep in mind when switching between coordinate systems:

- The interfaces (`depth`) have to be defined continuously increasing or decreasing, either from lowest to highest or the other way around. E.g., a simple five-layer model with the sea-surface at 0 m, a 100 m water column, and a target of 50 m 900 m below the seafloor can be defined in four ways:
  - `[0, 100, 1000, 1050]` -> LHS low to high
  - `[0, -100, -1000, -1050]` -> RHS high to low
  - `[1050, 1000, 100, 0]` -> LHS high to low
  - `[-1050, -1000, -100, 0]` -> RHS low to high
- The above point affects also all model parameters (`res`, `aniso`, `{e;m}perm{H;V}`). E.g., for the above five-layer example this would be
  - `res = [1e12, 0.3, 1, 50, 1]` -> LHS low to high
  - `res = [1e12, 0.3, 1, 50, 1]` -> RHS high to low
  - `res = [1, 50, 1, 0.3, 1e12]` -> LHS high to low
  - `res = [1, 50, 1, 0.3, 1e12]` -> RHS low to high

  Note that in a two-layer scenario the values are always taken as low-to-high (as it is not possible to detect the direction from only one interface).
- A source or a receiver *exactly on* a boundary is taken as being in the lower layer. Hence, if $z_{\mathrm{rec=z_0}}$, where z_0 is the surface, then the receiver is taken as in the air in the LHS, but as in the subsurface in the RHS. Similarly, if $z_{\mathrm{rec=z_{seafloor}}}$, then the receiver is taken as in the sea in the LHS, but as in the subsurface in the RHS. This can be avoided by never placing it exactly on a boundary, but slightly (e.g., 1 mm) in the layer where you want to have it.
- In `bipole`, the `dip` switches sign. Correspondingly in `dipole`, the `ab`'s containing vertical directions switch the sign for each vertical component.
- Sign switches also occur for magnetic sources or receivers.

In this example we first create a sketch of the LHS and RHS for visualization, followed by a few examples using `dipole` and `bipole` to demonstrate the two possibilities.

```
import empymod
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d import proj3d
from matplotlib.patches import FancyArrowPatch
plt.style.use('ggplot')
```

RHS vs LHS

Comparison of the right-handed system with positive z downwards and the left-handed system with positive z upwards. Easting is always x, and Northing is y.

```
class Arrow3D(FancyArrowPatch):
    """https://stackoverflow.com/a/29188796"""

    def __init__(self, xs, ys, zs):
        FancyArrowPatch.__init__(
                self, (0, 0), (0, 0), mutation_scale=20, lw=1.5,
                arrowstyle='-|>', color='.2', zorder=100)
        self._verts3d = xs, ys, zs

    def draw(self, renderer):
        xs3d, ys3d, zs3d = self._verts3d
        xs, ys, _ = proj3d.proj_transform(xs3d, ys3d, zs3d, renderer.M)
        self.set_positions((xs[0], ys[0]), (xs[1], ys[1]))
        FancyArrowPatch.draw(self, renderer)
```

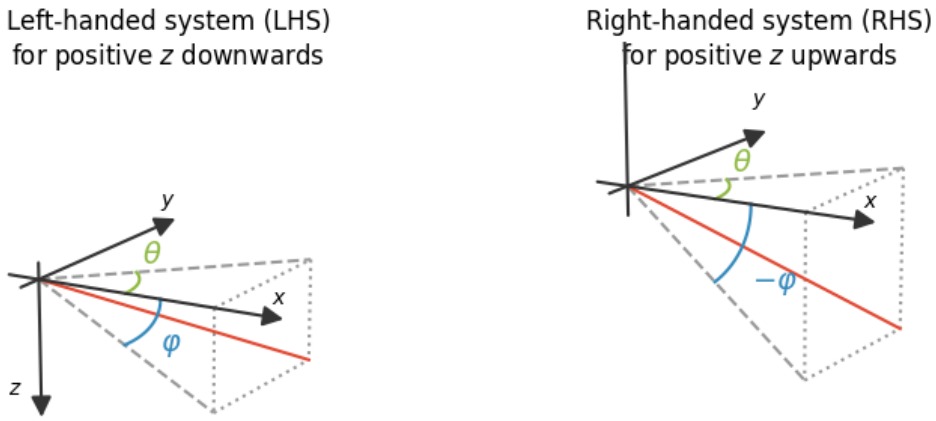
```
def repeated(ax, pm):
    """These are all repeated for the two subplots."""

    # Coordinate system
    # The first three are not visible, but for the aspect ratio of the plot.
    ax.plot([-2, 12], [0, 0], [0, 0], c='w')
    ax.plot([0, 0], [-2, 12], [0, 0], c='w')
    ax.plot([0, 0], [0, 0], [-pm*2, pm*12], c='w')
    ax.add_artist(Arrow3D([-2, 14], [0, 0], [0, 0]))
    ax.add_artist(Arrow3D([0, 0], [-2, 14], [0, 0]))
    ax.add_artist(Arrow3D([0, 0], [0, 0], [-pm*2, pm*14]))

    # Annotate it
    ax.text(12, 2, 0, r'$x$')
    ax.text(0, 12, 2, r'$y$')
    ax.text(-2, 0, pm*12, r'$z$')

    # Helper lines
    ax.plot([0, 10], [0, 10], [0, 0], '--', c='.6')
    ax.plot([0, 10], [0, 0], [0, -10], '--', c='.6')
    ax.plot([10, 10], [0, 10], [0, 0], ':', c='.6')
    ax.plot([10, 10], [10, 10], [0, -10], ':', c='.6')
    ax.plot([10, 10], [0, 0], [0, -10], ':', c='.6')
    ax.plot([10, 10], [0, 10], [-10, -10], ':', c='.6')

    # Resulting trajectory
    ax.plot([0, 10], [0, 10], [0, -10], 'C0')

    # Theta
    azimuth = np.linspace(np.pi/4, np.pi/2, 31)
```

(continues on next page)

(continued from previous page)

```
    ax.plot(np.sin(azimuth)*5, np.cos(azimuth)*5, 0, c='C5')
    ax.text(3, 5, 0, r"$\theta$", color='C5', fontsize=14)

    # Phi
    ax.plot(np.sin(azimuth)*7, azimuth*0, -np.cos(azimuth)*7, c='C1')

    ax.view_init(azim=-60, elev=20)
```

```
# Create figure
fig = plt.figure(figsize=(8, 3.5))

# Left-handed system
ax1 = fig.add_subplot(121, projection=Axes3D.name, facecolor='w')
ax1.axis('off')
plt.title('Left-handed system (LHS)\nfor positive $z$ downwards', fontsize=12)
ax1.text(7, 0, -5, r"$\varphi$", color='C1', fontsize=14)

repeated(ax1, -1)

# Right-handed  system
ax2 = fig.add_subplot(122, projection='3d', facecolor='w', sharez=ax1)
ax2.axis('off')
plt.title('Right-handed system (RHS)\nfor positive $z$ upwards', fontsize=12)
ax2.text(7, 0, -5, r"$-\varphi$", color='C1', fontsize=14)

repeated(ax2, 1)

plt.tight_layout()
plt.show()
```



Dipole

A simple example using `dipole`. It is a marine case with 300 meter water depth and a 50 m thick target 700 m below the seafloor.

```
off = np.linspace(500, 10000, 301)
```

LHS

In the left-handed system positive z is downwards. So we have to define our model by beginning with the air layer, followed by water, background, target, and background again. This means that all our depth-values are positive, as the air-interface z_0 is at 0 m.

```
lhs = empymod.dipole(
        src=[0, 0, 100],
        rec=[off, np.zeros(off.size), 200],
        depth=[0, 300, 1000, 1050],
        res=[1e20, 0.3, 1, 50, 1],
        # depth=[1050, 1000, 300, 0],  # Alternative way, LHS high to low.
        # res=[1, 50, 1, 0.3, 1e20],   # " " "
        freqtime=1,
        verb=0
)
```

RHS

In the right-handed system positive z is upwards. So we have to define our model by beginning with the background, followed by the target, background again, water, and air. This means that all our depth-values are negative.

```
rhs = empymod.dipole(
        src=[0, 0, -100],
        rec=[off, np.zeros(off.size), -200],
        depth=[0, -300, -1000, -1050],
        res=[1e20, 0.3, 1, 50, 1],
        # depth=[-1050, -1000, -300, 0],  # Alternative way, RHS low to high.
        # res=[1, 50, 1, 0.3, 1e20],      # " " "
        freqtime=1,
        verb=0
)
```

Compare

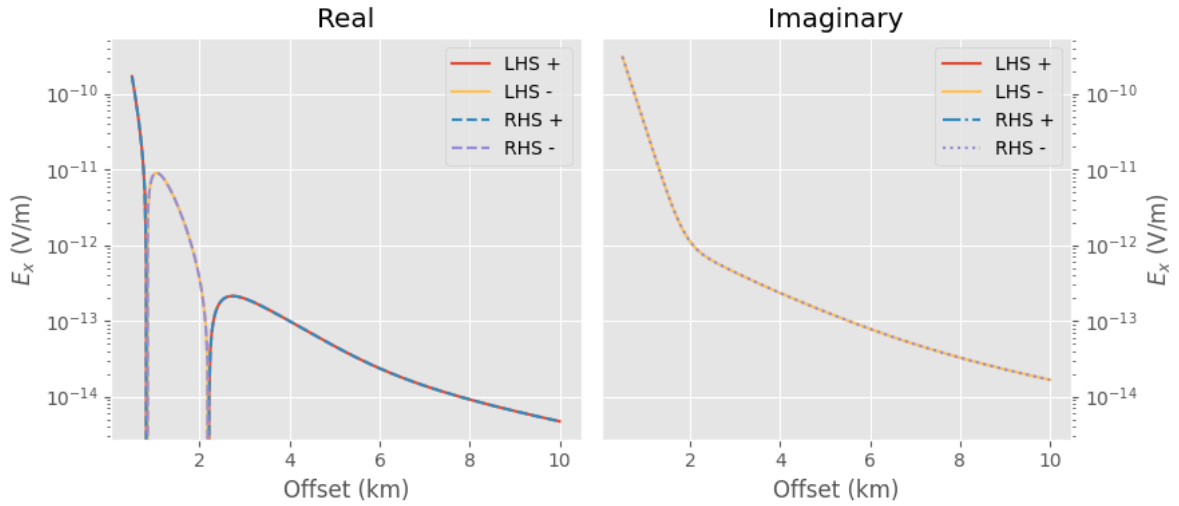
Plotting the two confirms that the results agree, no matter if we use the LHS or the RHS definition.

```
plt.figure(figsize=(9, 4))

ax1 = plt.subplot(121)
plt.title('Real')
plt.plot(off/1e3, lhs.real, 'C0', label='LHS +')
plt.plot(off/1e3, -lhs.real, 'C4', label='LHS -')
plt.plot(off/1e3, rhs.real, 'C1--', label='RHS +')
plt.plot(off/1e3, -rhs.real, 'C2--', label='RHS -')
plt.yscale('log')
plt.xlabel('Offset (km)')
plt.ylabel('$E_x$ (V/m)')
plt.legend()

ax2 = plt.subplot(122, sharey=ax1)
plt.title('Imaginary')
plt.plot(off/1e3, lhs.imag, 'C0', label='LHS +')
plt.plot(off/1e3, -lhs.imag, 'C4', label='LHS -')
plt.plot(off/1e3, rhs.imag, 'C1-.', label='RHS +')
plt.plot(off/1e3, -rhs.imag, 'C2:', label='RHS -')
plt.yscale('log')
plt.xlabel('Offset (km)')
```

(continues on next page)

(continued from previous page)

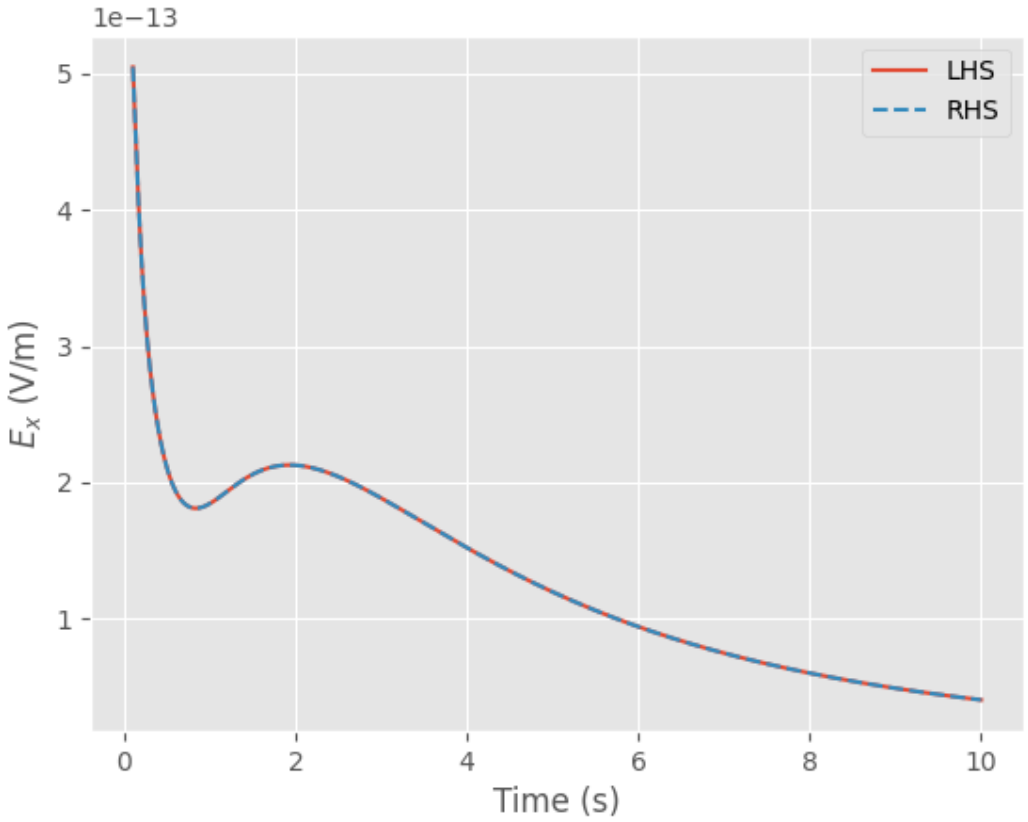
```
plt.ylabel('$E_x$ (V/m)')
plt.legend()
ax2.yaxis.set_label_position("right")
ax2.yaxis.tick_right()

plt.tight_layout()
plt.show()
```



Bipole [x1, x2, y1, y2, z1, z2]

A time-domain example using rotated bipoles, where we define them as $[x_1, x_2, y_1, y_2, z_1, z_2]$.

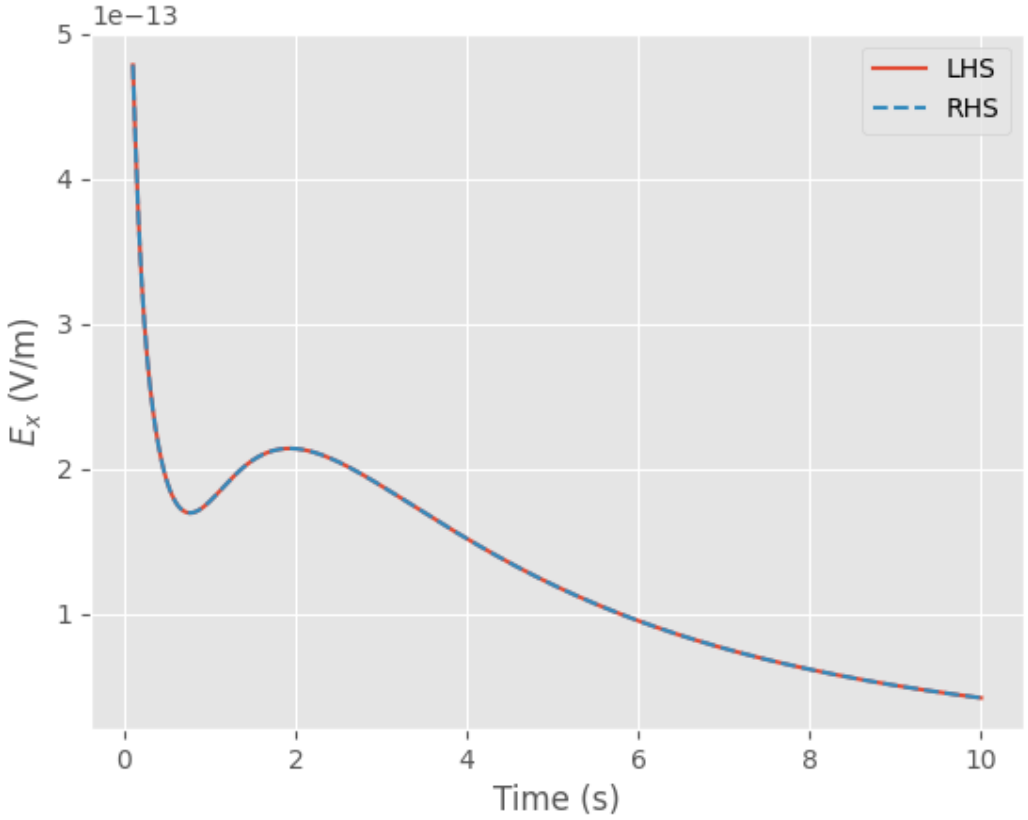
```
times = np.linspace(0.1, 10, 301)
inp = {'freqtime': times, 'signal': 0, 'verb': 0}

lhs = empymod.bipole(
        src=[-50, 50, -10, 10, 100, 110],
        rec=[6000, 6100, 20, -20, 220, 200],
        depth=[0, 300, 1000, 1050],
        res=[1e20, 0.3, 1, 50, 1],
        **inp
)

rhs = empymod.bipole(
        src=[-50, 50, -10, 10, -100, -110],
        rec=[6000, 6100, 20, -20, -220, -200],
        depth=[0, -300, -1000, -1050],
        res=[1e20, 0.3, 1, 50, 1],
        **inp
)

plt.figure()

plt.plot(times, lhs, 'C0', label='LHS')
plt.plot(times, rhs, 'C1--', label='RHS')
plt.xlabel('Time (s)')
plt.ylabel('$E_x$ (V/m)')
plt.legend()

plt.show()
```



Bipole [x, y, z, azimuth, dip]

A very similar time-domain example using rotated bipoles, but this time defining them as $[x, y, z, \theta, \varphi]$. Note that φ has to change the sign, while θ does not.

```
lhs = empymod.bipole(
        src=[0, 0, 100, 10, 20],
        rec=[6000, 0, 200, -5, 15],
        depth=[0, 300, 1000, 1050],
        res=[1e20, 0.3, 1, 50, 1],
        **inp
)

rhs = empymod.bipole(
        src=[0, 0, -100, 10, -20],
        rec=[6000, 0, -200, -5, -15],
        depth=[0, -300, -1000, -1050],
        res=[1e20, 0.3, 1, 50, 1],
        **inp
)

plt.figure()

plt.plot(times, lhs, 'C0', label='LHS')
plt.plot(times, rhs, 'C1--', label='RHS')
plt.xlabel('Time (s)')
plt.ylabel('$E_x$ (V/m)')
plt.legend()

plt.show()
```




```
empymod.Report()
```

Total running time of the script: (0 minutes 33.372 seconds)

Estimated memory usage: 189 MB

5.4.2 Frequency Domain

Basic examples in the frequency domain.

Point dipole vs finite length dipole

Comparison of a 800 m long bipole with a dipole at its centre.

```
import empymod
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

Define models

```
name = 'Example Model'          # Model name
depth = [0, 300, 1000, 1200]    # Layer boundaries
res = [2e14, 0.3, 1, 50, 1]     # Anomaly resistivities
resBG = [2e14, 0.3, 1, 1, 1]    # Background resistivities
aniso = [1, 1, 1.5, 1.5, 1.5]   # Layer anis. (same for anomaly and background)
```

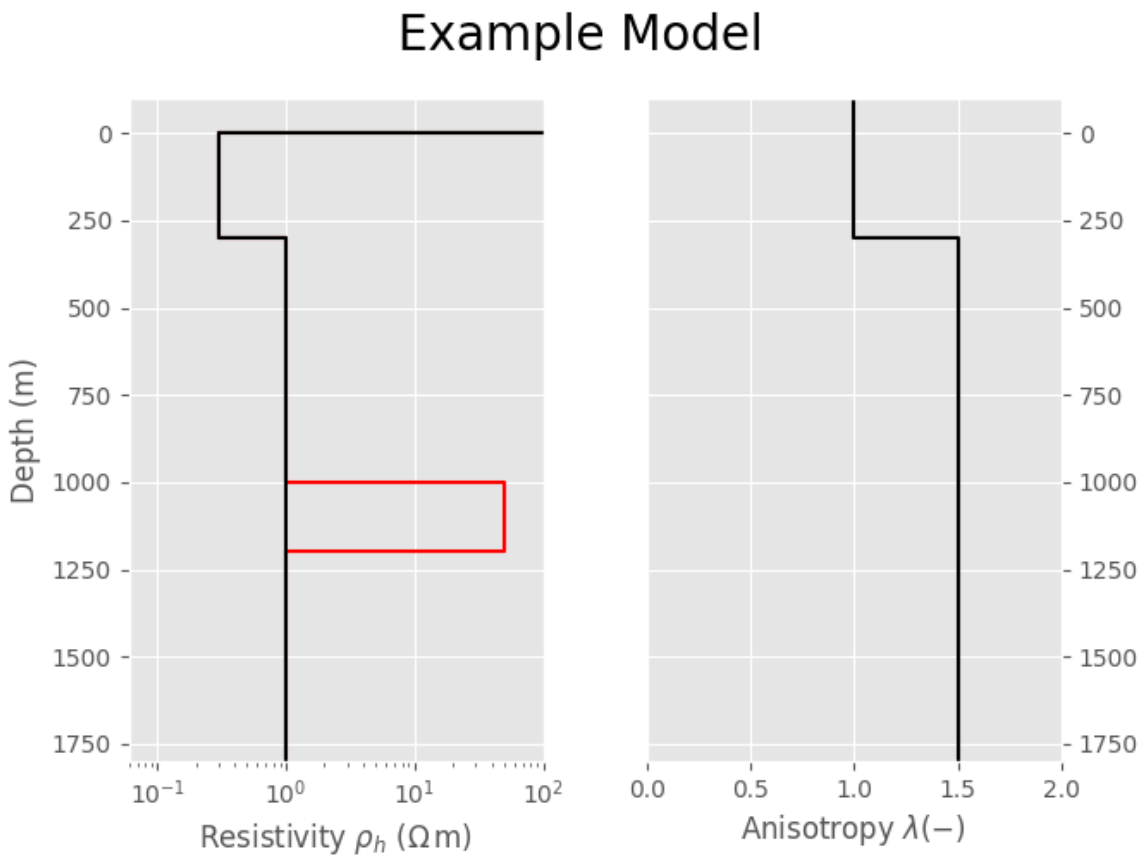
(continues on next page)

(continued from previous page)

```
# Modelling parameters
verb = 2

# Spatial parameters
zsrc = 250                    # Src-depth
zrec = 300                    # Rec-depth
fx = np.arange(5, 101)*100    # Offsets
fy = np.zeros(fx.size)        # 0s
```

Plot models

```
pdepth = np.repeat(np.r_[-100, depth], 2)
pdepth[:-1] = pdepth[1:]
pdepth[-1] = 2*depth[-1]
pres = np.repeat(res, 2)
presBG = np.repeat(resBG, 2)
pani = np.repeat(aniso, 2)

# Create figure
fig = plt.figure(figsize=(7, 5), facecolor='w')
fig.subplots_adjust(wspace=.25, hspace=.4)
plt.suptitle(name, fontsize=20)

# Plot Resistivities
ax1 = plt.subplot(1, 2, 1)
plt.plot(pres, pdepth, 'r')
plt.plot(presBG, pdepth, 'k')
plt.xscale('log')
plt.xlim([.2*np.array(res).min(), 2*np.array(res)[1:].max()])
plt.ylim([1.5*depth[-1], -100])
plt.ylabel('Depth (m)')
plt.xlabel(r'Resistivity $\rho_h\ (\Omega\,\rm{m})$')

# Plot anisotropies
ax2 = plt.subplot(1, 2, 2)
plt.plot(pani, pdepth, 'k')
plt.xlim([0, 2])
plt.ylim([1.5*depth[-1], -100])
plt.xlabel(r'Anisotropy $\lambda (-)$')
ax2.yaxis.tick_right()

plt.show()
```



Frequency response for f = 1 Hz

Compute

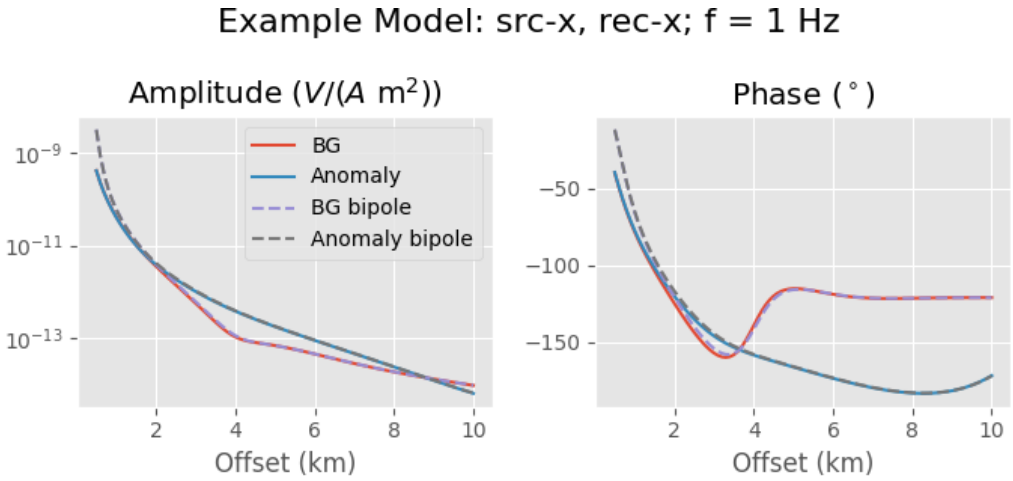
```
# Dipole
inpdat = {'src': [0, 0, zsrc, 0, 0], 'rec': [fx, fy, zrec, 0, 0],
          'depth': depth, 'freqtime': 1, 'aniso': aniso,
          'htarg': {'pts_per_dec': -1}, 'verb': verb}
fEM = empymod.bipole(**inpdat, res=res)
fEMBG = empymod.bipole(**inpdat, res=resBG)

# Bipole
inpdat['src'] = [-400, 400, 0, 0, zsrc, zsrc]
inpdat['srcpts'] = 10
fEMbp = empymod.bipole(**inpdat, res=res)
fEMBGbp = empymod.bipole(**inpdat, res=resBG)
```

Out:

```
:: empymod END; runtime = 0:00:00.007365 :: 1 kernel call(s)


:: empymod END; runtime = 0:00:00.004097 :: 1 kernel call(s)


:: empymod END; runtime = 0:00:00.030403 :: 10 kernel call(s)


:: empymod END; runtime = 0:00:00.022424 :: 10 kernel call(s)
```

Plot

```
fig = plt.figure(figsize=(8, 6), facecolor='w')
fig.subplots_adjust(wspace=.25, hspace=.4)
fig.suptitle(name+': src-x, rec-x; f = 1 Hz', fontsize=16, y=1)

# Plot Amplitude
ax1 = plt.subplot(2, 2, 1)
plt.semilogy(fx/1000, fEMBG.amp(), label='BG')
plt.semilogy(fx/1000, fEM.amp(), label='Anomaly')
plt.semilogy(fx/1000, fEMBGbp.amp(), '--', label='BG bipole')
plt.semilogy(fx/1000, fEMbp.amp(), '--', label='Anomaly bipole')
plt.legend(loc='best')
plt.title(r'Amplitude ($V/(A\ $m$^2$))')
plt.xlabel('Offset (km)')

# Plot Phase
ax2 = plt.subplot(2, 2, 2)
plt.title(r'Phase ($^\circ$)')
plt.plot(fx/1000, fEMBG.pha(deg=True), label='BG')
plt.plot(fx/1000, fEM.pha(deg=True), label='Anomaly')
plt.plot(fx/1000, fEMBGbp.pha(deg=True), '--', label='BG bipole')
plt.plot(fx/1000, fEMbp.pha(deg=True), '--', label='Anomaly bipole')
plt.xlabel('Offset (km)')

plt.show()
```




```
empymod.Report()
```

Total running time of the script: (0 minutes 1.765 seconds)

Estimated memory usage: 9 MB

A simple frequency-domain example

For a single frequency and a crossplot frequency vs offset.

```
import empymod
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

Define models

```
name = 'Example Model'          # Model name
depth = [0, 300, 1000, 1200]    # Layer boundaries
res = [2e14, 0.3, 1, 50, 1]     # Anomaly resistivities
resBG = [2e14, 0.3, 1, 1, 1]    # Background resistivities
aniso = [1, 1, 1.5, 1.5, 1.5]   # Layer anis. (same for anomaly & backg.)

# Modelling parameters
verb = 0
ab = 11    # source and receiver x-directed

# Spatial parameters
zsrc = 250                      # Src-depth
zrec = 300                      # Rec-depth
fx = np.arange(20, 101)*100     # Offsets
fy = np.zeros(fx.size)          # 0s
```

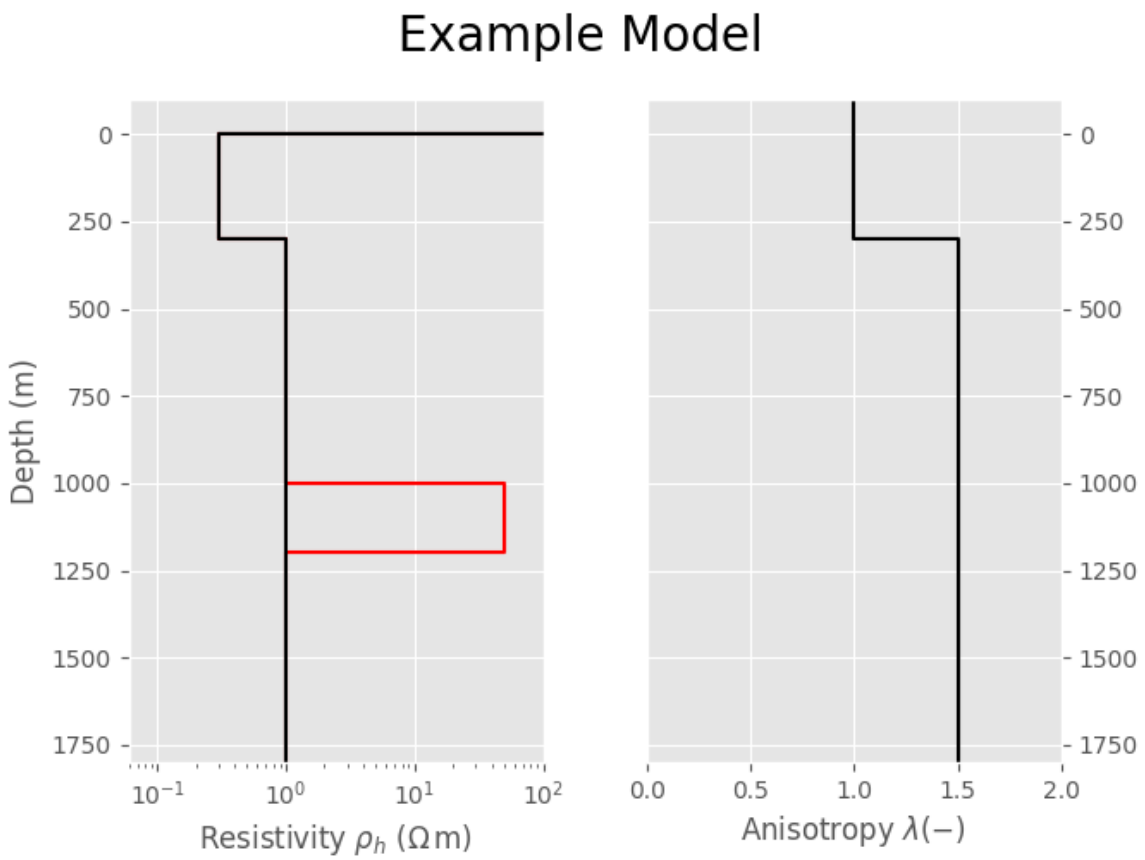
Plot models

```
pdepth = np.repeat(np.r_[-100, depth], 2)
pdepth[:-1] = pdepth[1:]
pdepth[-1] = 2*depth[-1]
pres = np.repeat(res, 2)
presBG = np.repeat(resBG, 2)
pani = np.repeat(aniso, 2)

# Create figure
fig = plt.figure(figsize=(7, 5), facecolor='w')
fig.subplots_adjust(wspace=.25, hspace=.4)
plt.suptitle(name, fontsize=20)

# Plot Resistivities
ax1 = plt.subplot(1, 2, 1)
plt.plot(pres, pdepth, 'r')
plt.plot(presBG, pdepth, 'k')
plt.xscale('log')
plt.xlim([.2*np.array(res).min(), 2*np.array(res)[1:].max()])
plt.ylim([1.5*depth[-1], -100])
plt.ylabel('Depth (m)')
plt.xlabel(r'Resistivity $\rho_h\ (\Omega\,\rm{m})$')

# Plot anisotropies
ax2 = plt.subplot(1, 2, 2)
plt.plot(pani, pdepth, 'k')
plt.xlim([0, 2])
```

(continues on next page)

(continued from previous page)

```
plt.ylim([1.5*depth[-1], -100])
plt.xlabel(r'Anisotropy $\lambda$ (-)$')
ax2.yaxis.tick_right()

plt.show()
```



1. Frequency response for f = 1 Hz

Compute

```
inpdat = {'src': [0, 0, zsrc], 'rec': [fx, fy, zrec], 'depth': depth,
          'freqtime': 1, 'aniso': aniso, 'ab': ab,
          'htarg': {'pts_per_dec': -1}, 'verb': verb}

fEM = empymod.dipole(**inpdat, res=res)
fEMBG = empymod.dipole(**inpdat, res=resBG)
```

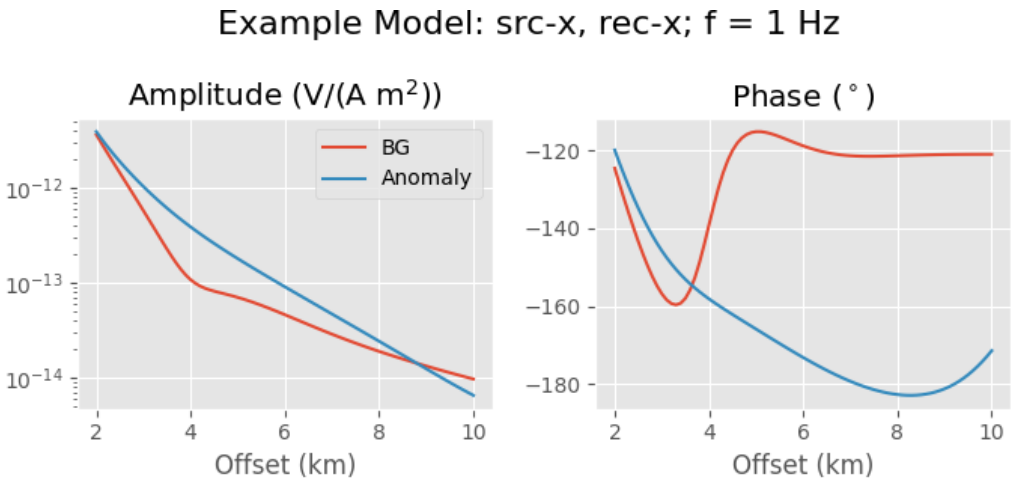
Plot

```
fig = plt.figure(figsize=(8, 6), facecolor='w')
fig.subplots_adjust(wspace=.25, hspace=.4)
fig.suptitle(name+': src-x, rec-x; f = 1 Hz', fontsize=16, y=1)

# Plot Amplitude
ax1 = plt.subplot(2, 2, 1)
plt.semilogy(fx/1000, fEMBG.amp(), label='BG')
```

(continues on next page)

(continued from previous page)

```
plt.semilogy(fx/1000, fEM.amp(), label='Anomaly')
plt.legend(loc='best')
plt.title(r'Amplitude (V/(A m$^2$))')
plt.xlabel('Offset (km)')

# Plot Phase
ax2 = plt.subplot(2, 2, 2)
plt.title(r'Phase ($^\circ$)')
plt.plot(fx/1000, fEMBG.pha(deg=True), label='BG')
plt.plot(fx/1000, fEM.pha(deg=True), label='Anomaly')
plt.xlabel('Offset (km)')

plt.show()
```



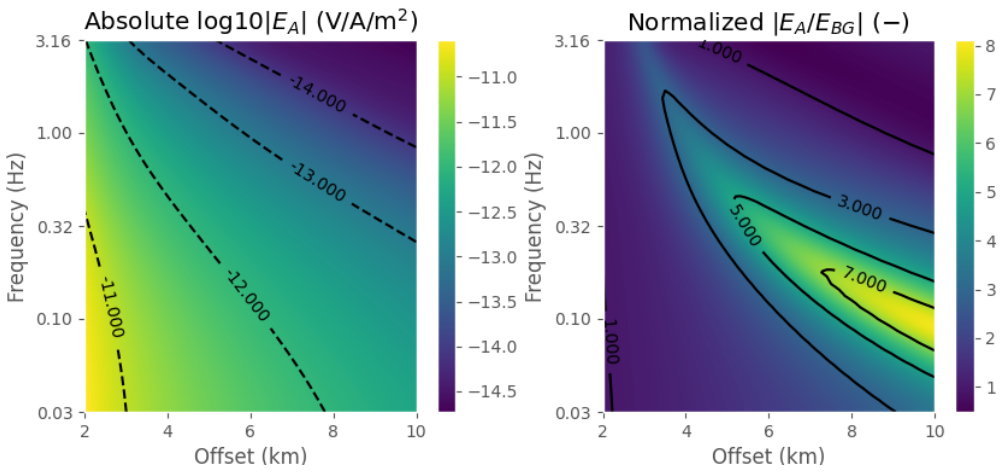
2. Crossplot

Compute

```
# Compute responses
freq = np.logspace(-1.5, .5, 33)  # 33 frequencies from -1.5 to 0.5 (logspace)
inpdat = {'src': [0, 0, zsrc], 'rec': [fx, fy, zrec], 'depth': depth,
          'freqtime': freq, 'aniso': aniso, 'ab': ab,
          'htarg': {'pts_per_dec': -1}, 'verb': verb}

xfEM = empymod.dipole(**inpdat, res=res)
xfEMBG = empymod.dipole(**inpdat, res=resBG)
```

Plot

```
lfreq = np.log10(freq)

# Create figure
fig = plt.figure(figsize=(10, 4), facecolor='w')
fig.subplots_adjust(wspace=.25, hspace=.4)

# Plot absolute (amplitude) in log10
ax1 = plt.subplot(1, 2, 2)
plt.title(r'Normalized $|E_A/E_{BG}|\ (-)$')
plt.imshow(np.abs(xfEM/xfEMBG), interpolation='bicubic',
           extent=[fx[0]/1000, fx[-1]/1000, lfreq[0], lfreq[-1]],
           origin='lower', aspect='auto')
plt.grid(False)
plt.colorbar()
CS = plt.contour(fx/1000, lfreq, np.abs(xfEM/xfEMBG), [1, 3, 5, 7], colors='k')
plt.clabel(CS, inline=1, fontsize=10)
plt.ylim([lfreq[0], lfreq[-1]])
plt.xlim([fx[0]/1000, fx[-1]/1000])
plt.xlabel('Offset (km)')
plt.ylabel('Frequency (Hz)')
plt.yticks([-1.5, -1, -.5, 0, .5], ('0.03', '0.10', '0.32', '1.00', '3.16'))

# Plot normalized
ax2 = plt.subplot(1, 2, 1)
plt.title(r'Absolute log10$|E_A|$ (V/A/m$^2$)')
plt.imshow(np.log10(np.abs(xfEM)), interpolation='bicubic',
           extent=[fx[0]/1000, fx[-1]/1000, lfreq[0], lfreq[-1]],
           origin='lower', aspect='auto')
plt.grid(False)
plt.colorbar()
CS = plt.contour(fx/1000, lfreq, np.log10(np.abs(xfEM)),
                 [-14, -13, -12, -11], colors='k')
plt.clabel(CS, inline=1, fontsize=10)
plt.ylim([lfreq[0], lfreq[-1]])
plt.xlim([fx[0]/1000, fx[-1]/1000])
plt.xlabel('Offset (km)')
plt.ylabel('Frequency (Hz)')
plt.yticks([-1.5, -1, -.5, 0, .5], ('0.03', '0.10', '0.32', '1.00', '3.16'))

fig.suptitle(name+': src-x, rec-x', fontsize=18, y=1.05)

plt.show()
```



```
empymod.Report()
```

Total running time of the script: (0 minutes 2.744 seconds)

Estimated memory usage: 9 MB

Comparison of all src-rec combinations

Comparison of all source-receiver combinations; electric and magnetic

We compute the secondary field for a simple model of a 1 Ohm.m halfspace below air. Source is 50 m above the surface in the air, receivers are on the surface, frequency is 1 Hz.

```
import empymod
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

Define Model

```
x = np.linspace(-10, 10, 101)*1000
rx = np.repeat([x, ], np.size(x), axis=0)
ry = rx.transpose()
inp = {'src': [0, 0, -50],
       'rec': [rx.ravel(), ry.ravel(), 0],
       'depth': 0,
       'res': [2e14, 1],
       'freqtime': 1,
       'xdirect': None,  # Secondary field comp., req. empymod >= v1.6.1.
       'htarg': {'pts_per_dec': -1},  # To speed-up the calculation
       'verb': 0}
```

Compute

```
# All possible combinations
pab = [11, 12, 13, 14, 15, 16, 21, 22, 23, 24, 25, 26,
       31, 32, 33, 34, 35, 36, 41, 42, 43, 44, 45, 46,
       51, 52, 53, 54, 55, 56, 61, 62, 63, 64, 65, 66]

# Compute and store them in fs
fs = dict()
for ab in pab:
    fs[str(ab)] = empymod.dipole(ab=ab, **inp).reshape(np.shape(rx))
```

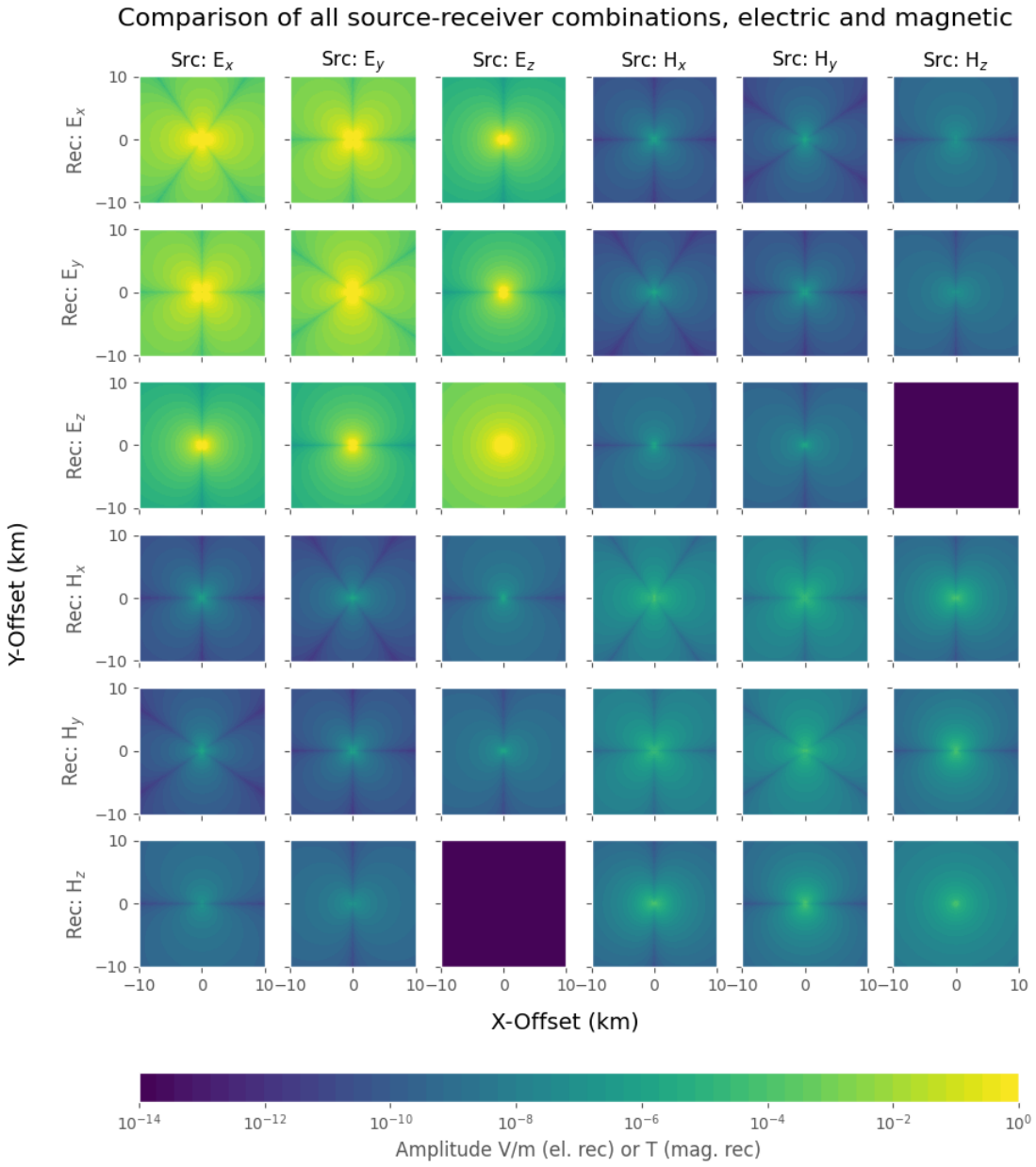
Plot

```
fig, axs = plt.subplots(figsize=(10, 12), nrows=6, ncols=6)
axs = axs.ravel()

# Labels
label1 = ['x', 'y', 'z']
label2 = ['E', 'H']

# Colour settings
vmin = 1e-14
```

(continues on next page)

(continued from previous page)

```
vmax = 1e-0
props = {'levels': np.logspace(np.log10(vmin), np.log10(vmax), 50),
         'locator': plt.matplotlib.ticker.LogLocator()}

# Loop over combinations
for i, val in enumerate(pab):
    plt.sca(axs[i])

    # Axis settings
    plt.xlim(min(x)/1000, max(x)/1000)
    plt.ylim(min(x)/1000, max(x)/1000)
    plt.axis('equal')

    # Plot the contour
    cf = plt.contourf(
            rx/1000, ry/1000, np.abs(fs[str(val)]).clip(vmin, vmax), **props)

    # Add titels
    if i < 6:
        label = r'Src: '
        label += label2[0] if i < 3 else label2[1]
        label += '$_' + label1[i % 3] + '$'
        plt.title(label, fontsize=12)

    # Remove unnecessary x-tick labels
    if i < 30:
        plt.xticks([-10, 0, 10], ())

    # Remove unnecessary y-tick labels; add y-labels
    if i % 6 != 0:
        plt.yticks([-10, 0, 10], ())
    else:
        label = r'Rec: '
        label += label2[0] if i < 18 else label2[1]
        label += '$_' + label1[(i // 6) % 3] + '$'
        plt.ylabel(label, fontsize=12)

# Colour bar
cax, kw = plt.matplotlib.colorbar.make_axes(
        axs, location='bottom', fraction=.05, pad=0.1, aspect=30)
cb = plt.colorbar(
        cf, cax=cax, ticks=np.logspace(np.log10(vmin), np.log10(vmax), 8),
        **kw)
cb.set_label(r'Amplitude V/m (el. rec) or T (mag. rec)')

# Annotate
plt.suptitle('Comparison of all source-receiver combinations, electric ' +
             'and magnetic', y=0.93, fontsize=16)
fig.text(0.5, 0.18, 'X-Offset (km)', ha='center', fontsize=14)
fig.text(0.01, 0.5, 'Y-Offset (km)', va='center', rotation='vertical',
         fontsize=14)

plt.show()
```




```
empymod.Report()
```

Total running time of the script: (0 minutes 4.066 seconds)

Estimated memory usage: 8 MB

5.4.3 Time Domain

Basic examples in the time domain.

Cole-Cole

There are various different definitions of a Cole-Cole model, see for instance Tarasov and Titov (2013). We try a few different ones here, but you can supply your preferred version.

The original Cole-Cole (1940) model was formulated for the complex dielectric permittivity. It is reformulated to conductivity to use it for IP,

$$\sigma(\omega) = \sigma_\infty + \frac{\sigma_0 - \sigma_\infty}{1 + (i\omega\tau)^C} \,. \qquad (1)$$

Another, similar model is given by Pelton et al. (1978),

$$\rho(\omega) = \rho_\infty + \frac{\rho_0 - \rho_\infty}{1 + (i\omega\tau)^C} \,. \qquad (2)$$

Equation (2) is just like equation (1), but replaces σ by ρ. However, mathematically they are not the same. Substituting $\rho = 1/\sigma$ in the latter and resolving it for σ will not yield the former. Equation (2) is usually written in the following form, using the chargeability $m = (\rho_0 - \rho_\infty)/\rho_0$,

$$\rho(\omega) = \rho_0 \left[1 - m\left(1 - \frac{1}{1 + (i\omega\tau)^C}\right)\right] \,. \qquad (3)$$

In all cases we add the part coming from the dielectric permittivity (displacement currents), even tough it usually doesn't matter in the frequency range of IP.

References

- **Cole, K.S., and R.H. Cole, 1941**, Dispersion and adsorption in dielectrics. I. Alternating current characteristics; *Journal of Chemical Physics*, Volume 9, Pages 341-351, doi: 10.1063/1.1750906.
- **Pelton, W.H., S.H. Ward, P.G. Hallof, W.R. Sill, and P.H. Nelson, 1978**, Mineral discrimination and removal of inductive coupling with multifrequency IP, *Geophysics*, Volume 43, Pages 588-609, doi: 10.1190/1.1440839.
- **Tarasov, A., and K. Titov, 2013**, On the use of the Cole–Cole equations in spectral induced polarization; *Geophysical Journal International*, Volume 195, Issue 1, Pages 352-356, doi: 10.1093/gji/ggt251.


```
import empymod
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

Use empymod with user-def. func. to adjust η and ζ

In principal it is always best to write your own modelling routine if you want to adjust something. Just copy `empymod.dipole` or `empymod.bipole` as a template, and modify it to your needs. Since `empymod v1.7.4`, however, there is a hook which allows you to modify η_h, η_v, ζ_h, and ζ_v quite easily.

The trick is to provide a dictionary (we name it `inp` here) instead of the resistivity vector in `res`. This dictionary, `inp`, has two mandatory plus optional entries: - `res`: the resistivity vector you would have provided normally (mandatory).

- A function name, which has to be either or both of (mandatory):
  - `func_eta`: To adjust `etaH` and `etaV`, or
  - `func_zeta`: to adjust `zetaH` and `zetaV`.
- In addition, you have to provide all parameters you use in `func_eta`/`func_zeta` and are not already provided to `empymod`. All additional parameters must have #layers elements.

The functions `func_eta` and `func_zeta` must have the following characteristics:

- The signature is `func(inp, p_dict)`, where

  - `inp` is the dictionary you provide, and
  - `p_dict` is a dictionary that contains all parameters so far computed in empymod [`locals()`].
- It must return `etaH, etaV` if `func_eta`, or `zetaH, zetaV` if `func_zeta`.

Dummy example

```
def my_new_eta(inp, p_dict):
    # Your computations, using the parameters you provided
    # in ``inp`` and the parameters from empymod in ``p_dict``.
    # In the example below, we provide, e.g., inp['tau']
    return etaH, etaV
```

And then you call `empymod` with `res = {'res': res-array, 'tau': tau, 'func_eta': my_new_eta}`.

Define the Cole-Cole model

In this notebook we exploit this hook in empymod to compute η_h and η_v with the Cole-Cole model. By default, η_h and η_v are computed like this:

$$\eta_h = \frac{1}{\rho} + j\omega\varepsilon_{r;h}\varepsilon_0 \,, \qquad (4)$$

$$\eta_v = \frac{1}{\rho\lambda^2} + j\omega\varepsilon_{r;v}\varepsilon_0 \,. \qquad (5)$$

With this function we recompute it. We replace the real part, the resistivity ρ, in equations (4) and (5) by the complex, frequency-dependent Cole-Cole resistivity [$\rho(\omega)$], as given, for instance, in equations (1)-(3). Then we add back the imaginary part coming from thet dielectric permittivity (basically zero for low frequencies).

Note that in this notebook we use this hook to model relaxation in the low frequency spectrum for IP measurements, replacing ρ by a frequency-dependent model $\rho(\omega)$. However, this could also be used to model dielectric phenomena in the high frequency spectrum, replacing ε_r by a frequency-dependent formula $\varepsilon_r(\omega)$.

```
def cole_cole(inp, p_dict):
    """Cole and Cole (1941)."""

    # Compute complex conductivity from Cole-Cole
    iotc = np.outer(2j*np.pi*p_dict['freq'], inp['tau'])**inp['c']
    condH = inp['cond_8'] + (inp['cond_0']-inp['cond_8'])/(1+iotc)
    condV = condH/p_dict['aniso']**2

    # Add electric permittivity contribution
    etaH = condH + 1j*p_dict['etaH'].imag
    etaV = condV + 1j*p_dict['etaV'].imag

    return etaH, etaV


def pelton_et_al(inp, p_dict):
    """ Pelton et al. (1978)."""

    # Compute complex resistivity from Pelton et al.
    iotc = np.outer(2j*np.pi*p_dict['freq'], inp['tau'])**inp['c']
    rhoH = inp['rho_0']*(1 - inp['m']*(1 - 1/(1 + iotc)))
    rhoV = rhoH*p_dict['aniso']**2

    # Add electric permittivity contribution
    etaH = 1/rhoH + 1j*p_dict['etaH'].imag
```

(continues on next page)

(continued from previous page)

```
    etaV = 1/rhoV + 1j*p_dict['etaV'].imag

    return etaH, etaV
```

Example

Two half-space model, air above earth:

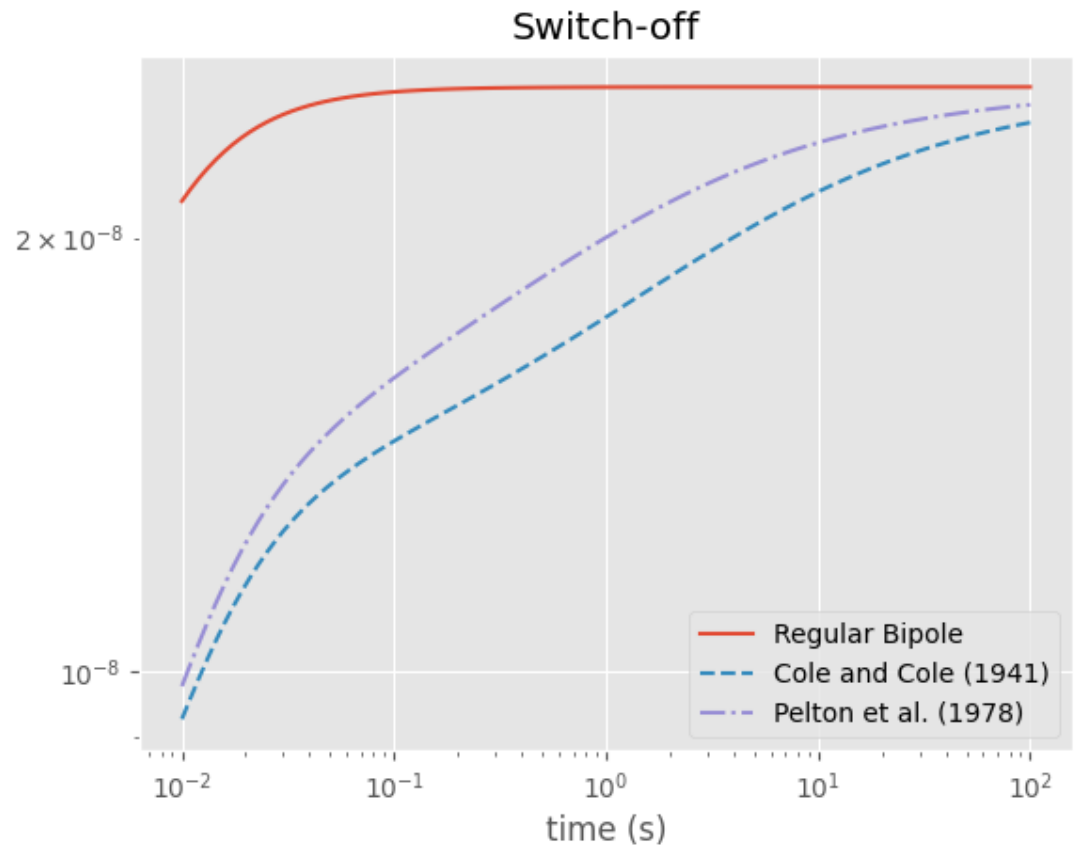
- x-directed sourcer at the surface
- x-directed receiver, also at the surface, inline at an offset of 500 m.
- Switch-on time-domain response
- Isotropic
- Model [air, subsurface]
  - $\rho_\infty = 1/\sigma_\infty = [2e14, 10]$
  - $\rho_0 = 1/\sigma_0 = [2e14, 5]$
  - $\tau = [0, 1]$
  - $c = [0, 0.5]$

```
# Times
times = np.logspace(-2, 2, 101)

# Model parameter which apply for all
model = {
    'src': [0, 0, 1e-5, 0, 0],
    'rec': [500, 0, 1e-5, 0, 0],
    'depth': 0,
    'freqtime': times,
    'signal': 1,
    'verb': 1
}

# Collect Cole-Cole models
res_0 = np.array([2e14, 10])
res_8 = np.array([2e14, 5])
tau = [0, 1]
c = [0, 0.5]
m = (res_0-res_8)/res_0

cole_model = {'res': res_0, 'cond_0': 1/res_0, 'cond_8': 1/res_8,
              'tau': tau, 'c': c, 'func_eta': cole_cole}
pelton_model = {'res': res_0, 'rho_0': res_0, 'm': m,
                'tau': tau, 'c': c, 'func_eta': pelton_et_al}

# Compute
out_bipole = empymod.bipole(res=res_0, **model)
out_cole = empymod.bipole(res=cole_model, **model)
out_pelton = empymod.bipole(res=pelton_model, **model)

# Plot
plt.figure()
plt.title('Switch-off')
plt.plot(times, out_bipole, label='Regular Bipole')
plt.plot(times, out_cole, '--', label='Cole and Cole (1941)')
plt.plot(times, out_pelton, '-.', label='Pelton et al. (1978)')
plt.legend()
```

(continues on next page)

(continued from previous page)

```
plt.yscale('log')
plt.xscale('log')
plt.xlabel('time (s)')
plt.show()
```




```
empymod.Report()
```

Total running time of the script: (0 minutes 1.386 seconds)

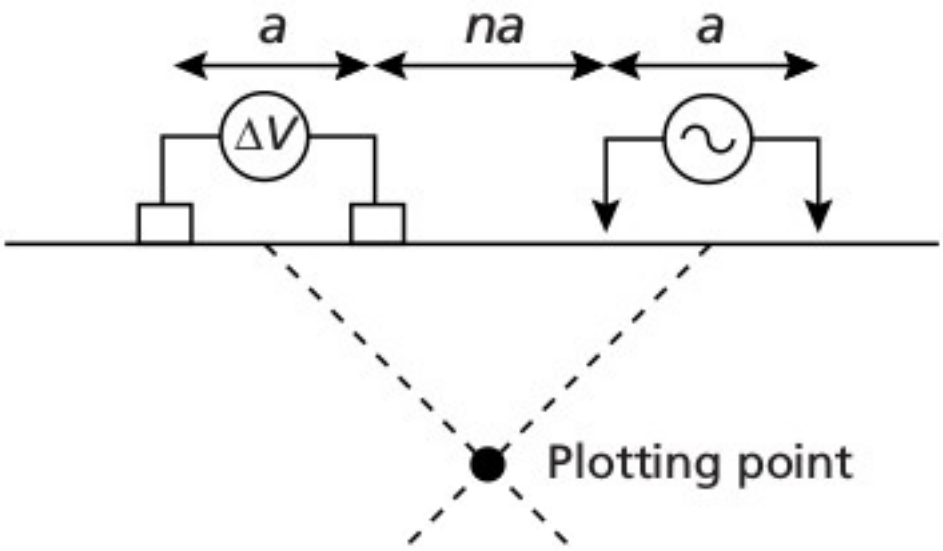
Estimated memory usage: 8 MB

DC apparent resistivity

DC apparent resistivity, dipole-dipole configuration.

There are various DC sounding layouts, the most common ones being Schlumberger, Wenner, pole-pole, pole-dipole, and **dipole-dipole**, at which we have a look here.

Dipole-dipole layout as shown in figure 8.32 in Kearey et al. (2002):



The apparent resistivity ρ_a of the *plotting point* is then computed with

$$\rho_a = \frac{V}{I}\pi n a(n+1)(n+2)\ ,$$

where V is measured Voltage, I is source strength, a is dipole length, and n is the factor of source-receiver separation.

References

Kearey, P., M. Brooks, and I. Hill, 2002, An introduction to geophysical exploration, 3rd ed.: Blackwell Scientific Publications, ISBN: 0 632 04929 4.

```
import empymod
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

Compute ρ_a

First we define a function to compute apparent resistivity for a given model and given source and receivers.

```
def comp_appres(depth, res, a, n, srcpts=1, recpts=1, verb=1):
    """Return apparent resistivity for dipole-dipole DC measurement

        rho_a = V/I pi a n (n+1) (n+2).

    Returns die apparent resistivity due to:
    - Electric source, inline (y = 0 m).
    - Source of 1 A strength.
    - Source and receiver are located at the air-interface.
    - Source is centered at x = 0 m.

    Note: DC response can be obtained by either t->infinity s or f->0 Hz. f = 0
          Hz is much faster, as there is no Fourier transform involved and only
          a single frequency has to be computed. By default, the minimum
          frequency in empymod is 1e-20 Hz. The difference between the signals
          for 1e-20 Hz and 0 Hz is very small.
```

(continues on next page)

(continued from previous page)

```
    For more explanation regarding input parameters see `empymod.model`.

    Parameters
    ----------
    depth : Absolute depths of layer interfaces, without air-interface.
    res : Resistivities of the layers, one more than depths (lower HS).
    a : Dipole length.
    n : Separation factors.
    srcpts, recpts : If < 3, bipoles are approximated as dipoles.
    verb : Verbosity.

    Returns
    -------
    rho_a : Apparent resistivity.
    AB2 : Src/rec-midpoints

    """

    # Get offsets between src-midpoint and rec-midpoint, AB
    AB = (n+1)*a

    # Collect model, putting source and receiver slightly (1e-3 m) into the
    # ground.
    model = {
        'src': [-a/2, a/2, 0, 0, 1e-3, 1e-3],
        'rec': [AB-a/2, AB+a/2, AB*0, AB*0, 1e-3, 1e-3],
        'depth': np.r_[0, np.array(depth, ndmin=1)],
        'freqtime': 1e-20,  # Smaller f would be set to 1e-20 be empymod.
        'verb': verb,       # Setting it to 1e-20 avoids warning-message.
        'res': np.r_[2e14, np.array(res, ndmin=1)],
        'strength': 1,      # So it is NOT normalized to 1 m src/rec.
        'htarg': {'pts_per_dec': -1},
    }

    return np.real(empymod.bipole(**model))*np.pi*a*n*(n+1)*(n+2), AB/2
```

Plot-function

Second we create a plot-function, which includes the call to *comp_appres*, to use for a couple of different models.

```
def plotit(depth, a, n, res1, res2, res3, title):
    """Call `comp_appres` and plot result."""

    # Compute the three different models
    rho1, AB2 = comp_appres(depth, res1, a, n)
    rho2, _ = comp_appres(depth, res2, a, n)
    rho3, _ = comp_appres(depth, res3, a, n)

    # Create figure
    plt.figure()

    # Plot curves
    plt.loglog(AB2, rho1, label='Case 1')
    plt.plot(AB2, rho2, label='Case 2')
    plt.plot(AB2, rho3, label='Case 3')

    # Legend, labels
    plt.legend(loc='best')
    plt.title(title)
```

(continues on next page)

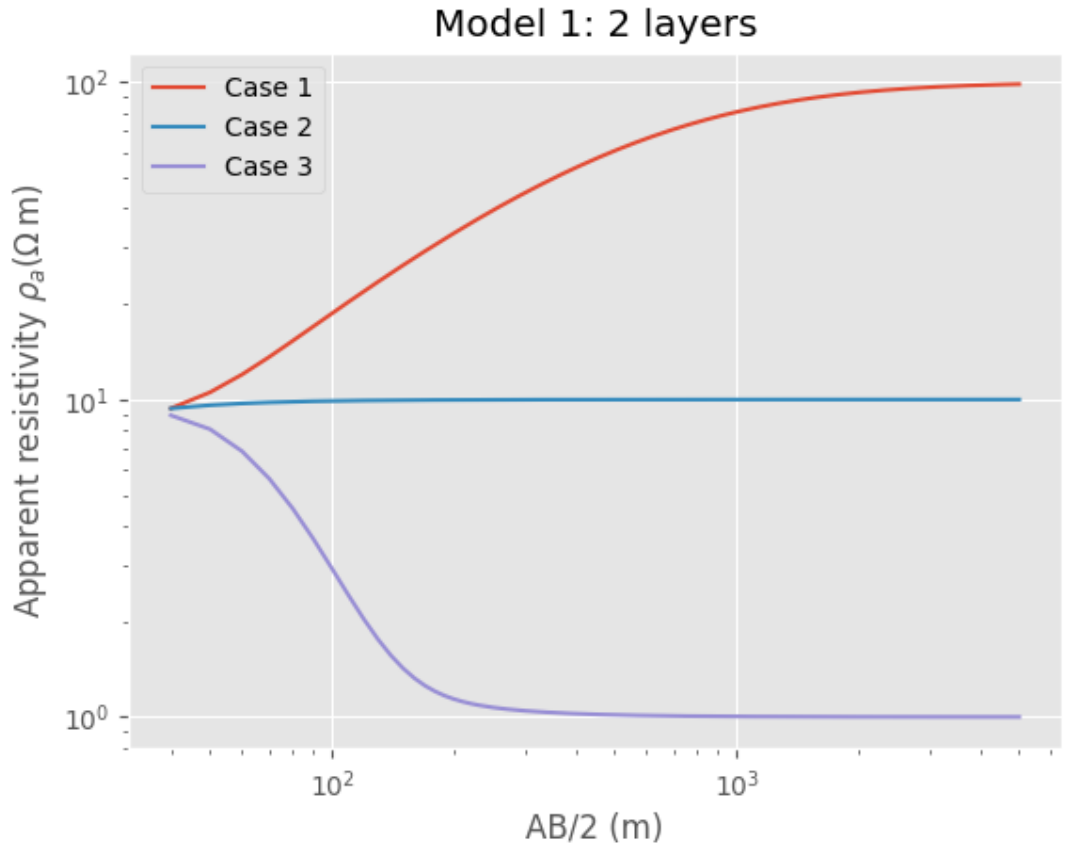
(continued from previous page)

```
    plt.xlabel('AB/2 (m)')
    plt.ylabel(r'Apparent resistivity $\rho_a (\Omega\,$m)')

    plt.show()
```

Model 1: 2 layers

layer	depth (m)	resistivity (Ohm m)
air	$-\infty$ - 0	2e14
layer 1	0 - 50	10
layer 2	50 - ∞	100 / 10 / 1

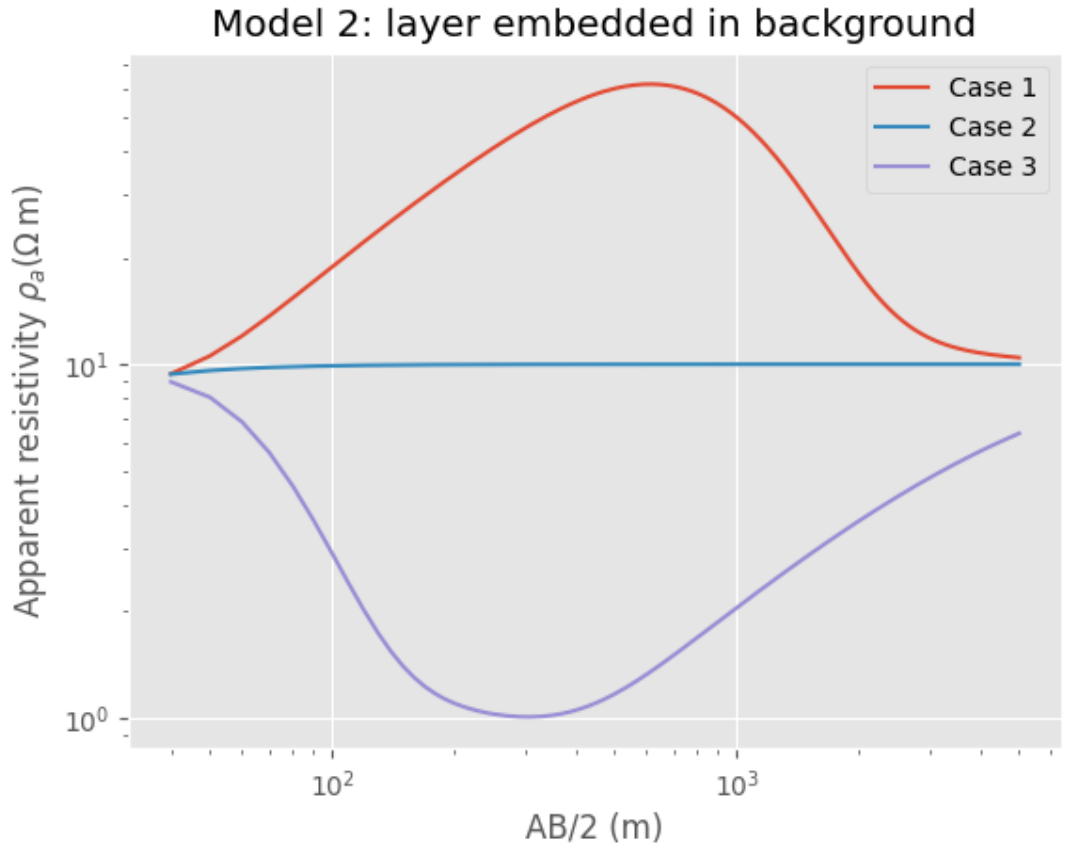
```
plotit(
    50,                 # Depth
    20,                 # a (src- and rec-lengths)
    np.arange(3, 500),  # n
    [10, 100],          # Case 1
    [10,  10],          # Case 2
    [10,   1],          # Case 3
    'Model 1: 2 layers')
```



Model 2: layer embedded in background

layer	depth (m)	resistivity (Ohm m)
air	$-\infty$ - 0	2e14
layer 1	0 - 50	10
layer 2	50 - 500	100 / 10 / 1
layer 3	500 - ∞	10

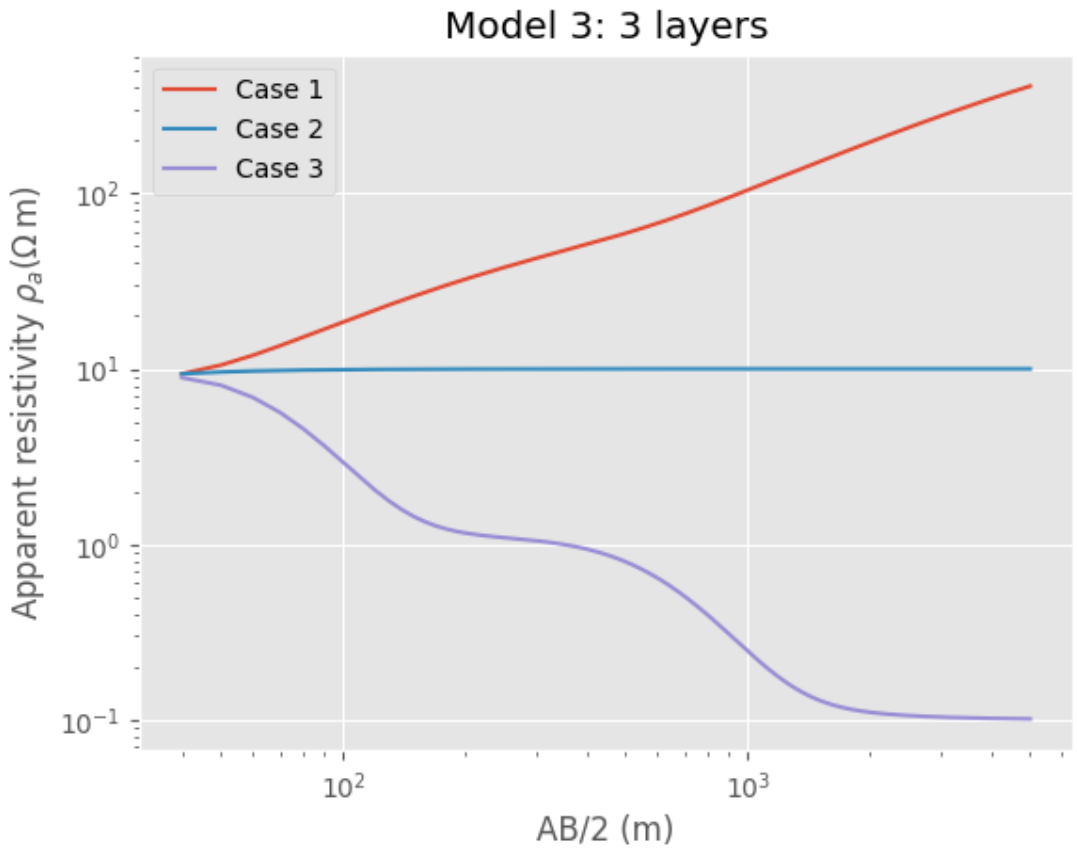
```
plotit(
    [50, 500],           # Depth
    20,                  # a (src- and rec-lengths)
    np.arange(3, 500),   # n
    [10, 100, 10],       # Case 1
    [10,  10, 10],       # Case 2
    [10,   1, 10],       # Case 3
    'Model 2: layer embedded in background')
```



Model 3: 3 layers

layer	depth (m)	resistivity (Ohm m)
air	$-\infty$ - 0	2e14
layer 1	0 - 50	10
layer 2	50 - 500	100 / 10 / 1
layer 3	500 - ∞ | 1000 / 10 / 0.1	

```
plotit(
    [50, 500],           # Depth
    20,                  # a (src- and rec-lengths)
    np.arange(3, 500),   # n
    [10, 100, 1000],     # Case 1
    [10,  10,   10],     # Case 2
    [10,   1,    0.1],   # Case 3
    'Model 3: 3 layers')
```




```
empymod.Report()
```

Total running time of the script: (0 minutes 2.655 seconds)

Estimated memory usage: 8 MB

Transform utilities within empymod for other modellers

This is an example how you can use the Fourier-transform tools implemented in `empymod` with other modellers. You could achieve the same for the Hankel transform.

`empymod` has various Fourier transforms implemented:

- Digital Linear Filters DLF (Sine/Cosine)
- Quadrature with Extrapolation QWE
- Logarithmic Fast Fourier Transform FFTLog
- Fast Fourier Transform FFT

For details of all the parameters see the `empymod`-docs or the function's docstrings.

```
import empymod
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

Model and transform parameters

The model actually doesn't matter for our purpose, but we need some model to show how it works.

```
# Define model, a halfspace
model = {
    'src': [0, 0, 0.001],      # Source at origin, slightly below interface
    'rec': [6000, 0, 0.001],   # Receivers in-line, 0.5m below interface
    'depth': [0],              # Air interface
    'res': [2e14, 1],          # Resistivity: [air, half-space]
    'epermH': [0, 1],          # Set el. perm. of air to 0 because of num. noise
}

# Specify desired times
time = np.linspace(0.1, 30, 301)

# Desired time-domain signal (0: impulse; 1: step-on; -1: step-off)
signal = 1

# Get required frequencies to model this time-domain result
# => we later need ``ft`` and ``ftarg`` for the Fourier transform.
# => See the docstrings (e.g., empymod.model.dipole) for available transforms
#    and their arguments.
time, freq, ft, ftarg = empymod.utils.check_time(
        time=time, signal=signal, ft='dlf', ftarg={}, verb=3)
```

Out:

```
time        [s] :  0.1 - 30 : 301  [min-max; #]
Fourier         :  DLF (Sine-Filter)
  > Filter      :  Key 201 CosSin (2012)
  > DLF type    :  Lagged Convolution
```

Frequency-domain computation

=> Here we compute the frequency-domain result with 'empymod', but you could compute it with any other modeller.

```
fresp = empymod.dipole(freqtime=freq, **model)
```

Out:

```
:: empymod END; runtime = 0:00:00.035593 :: 1 kernel call(s)
```

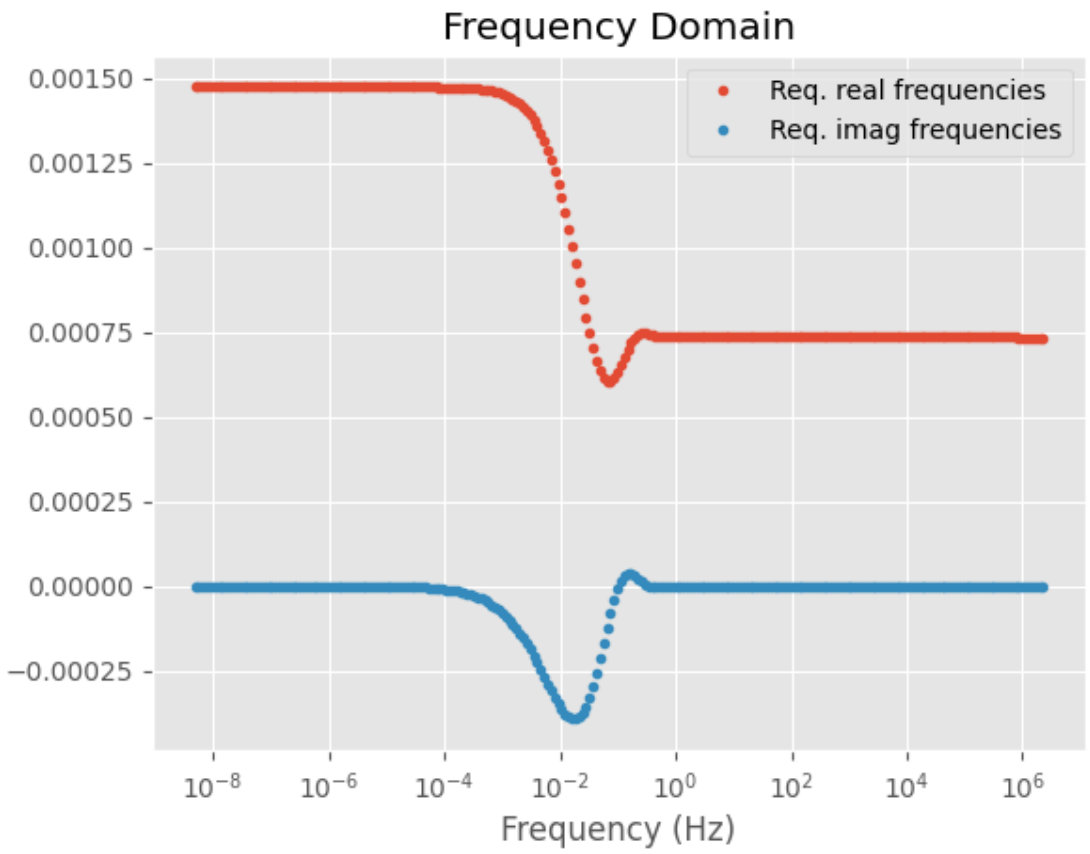
Plot frequency-domain result

```
plt.figure()

plt.title('Frequency Domain')
plt.plot(freq, 1e9*fresp.real, 'C0.', label='Req. real frequencies')
plt.plot(freq, 1e9*fresp.imag, 'C1.', label='Req. imag frequencies')
plt.legend()
plt.xscale('log')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Frequency (Hz)')
plt.ylabel('$E_x$ (nV/m)')

plt.show()
```



Fourier transform

```
# Compute corresponding time-domain signal.
tresp, _ = empymod.model.tem(
    fEM=fresp[:, None],
    off=np.array(model['rec'][0]),
    freq=freq,
    time=time,
    signal=signal,
    ft=ft,
    ftarg=ftarg)

tresp = np.squeeze(tresp)

# Time-domain result just using empymod
tresp2 = empymod.dipole(freqtime=time, signal=signal, verb=1, **model)
```

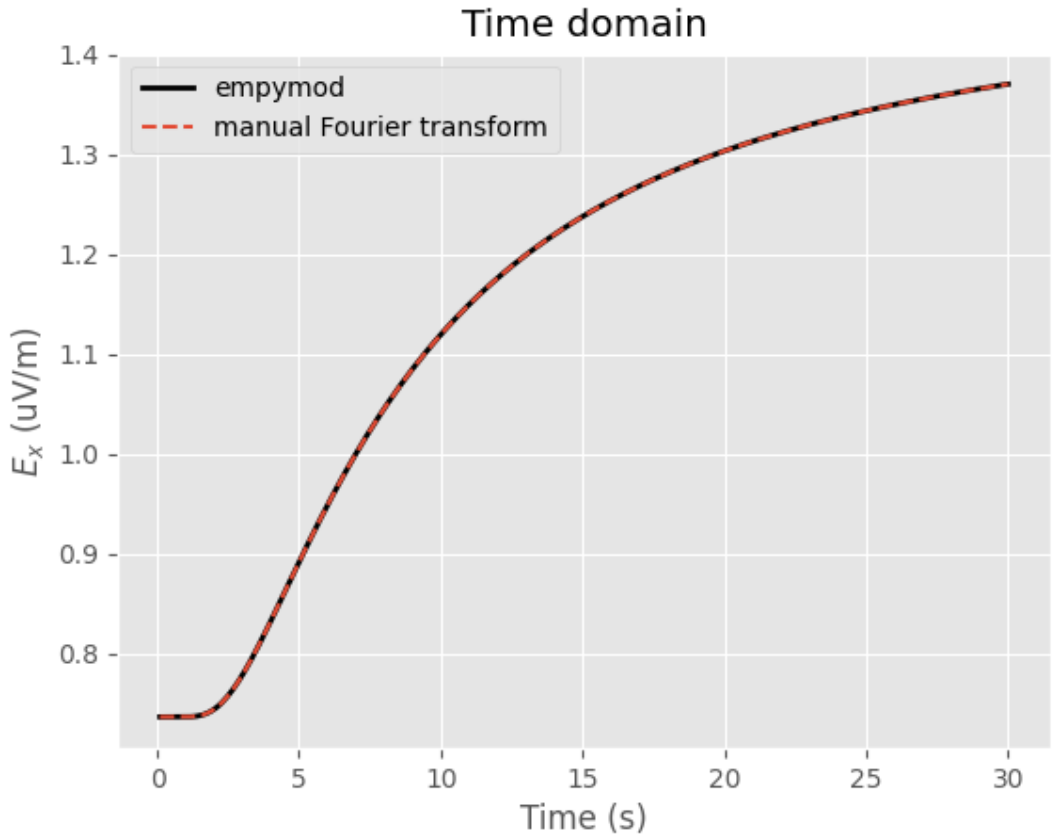
Plot time-domain result

```
fig = plt.figure()

plt.title('Time domain')
```

(continues on next page)

(continued from previous page)

```
plt.plot(time, tresp2*1e12, 'k', lw=2, label='empymod')
plt.plot(time, tresp*1e12, 'C0--', label='manual Fourier transform')
plt.legend()
plt.xlabel('Time (s)')
plt.ylabel('$E_x$ (uV/m)')

plt.show()
```




```
empymod.Report()
```

Total running time of the script: (0 minutes 2.010 seconds)

Estimated memory usage: 10 MB

Improve land CSEM computation

The problem

There exists a numerical singularity in the wavenumber-frequency domain. This singularity is always present, but it is usually neglectible except when the resistivity is very high (like air; hence conductivity goes to zero and therefore the real part of η_H, η_V goes to zero) and source and receiver are close to this boundary. So unfortunately exactly in the case of a land CSEM survey.

This singularity leads to noise at very high frequencies and therefore at very early times because the Hankel transform cannot handle the singularity correctly (or, if you would choose a sufficiently precise quadrature, it would take literally forever to compute it).

The "solution"

Electric permittivity (ε_H, ε_V) are set to 1 by default. They are not important for the frequency range of CSEM. By setting the electric permittivity of the air-layer to 0, the singularity disapears, which subsquently improves a lot the time-domain result for land CSEM. It therefore uses the diffusive approximation for the air layer, but again, that doesn't matter for the frequencies required for CSEM.

```
import empymod
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

Define model

```
# Times (s)
t = np.logspace(-2, 1, 301)

# Model
model = {
    'src': [0, 0, 0.001],           # Src at origin, slightly below interface
    'rec': [6000, 0, 0.0001],       # 6 km off., in-line, slightly bel. interf.
    'depth': [0, 2000, 2100],       # Target of 100 m at 2 km depth
    'res': [2e14, 10, 100, 10],     # Res: [air, overb., target, half-space]
    'epermH': [1, 1, 1, 1],         # El. permittivity: default values
    'freqtime': t,                  # Times
    'signal': 0,                    # 0: Impulse response
    'ftarg': {'dlf': 'key_81_CosSin_2009'},  # Shorter filter then the default
    'verb': 1,                      # Verbosity; set to 3 to see all parameters
}
```

Compute

```
# Compute with default eperm_air = 1
res_1 = empymod.dipole(**model)

# Set horizontal and vertical electric permittivity of air to 0
model['epermH'][0] = 0
# Note that for empymod < v2.0.0 you have to set `epermH` AND `epermV`. From
# v2.0.0 onwards `eperm` is assumed isotropic if `epermV` is not provided, and
# `epermV` is therefore internally a copy of `epermH`.

# Compute with eperm_air = 0
res_0 = empymod.dipole(**model)
```

Plot result

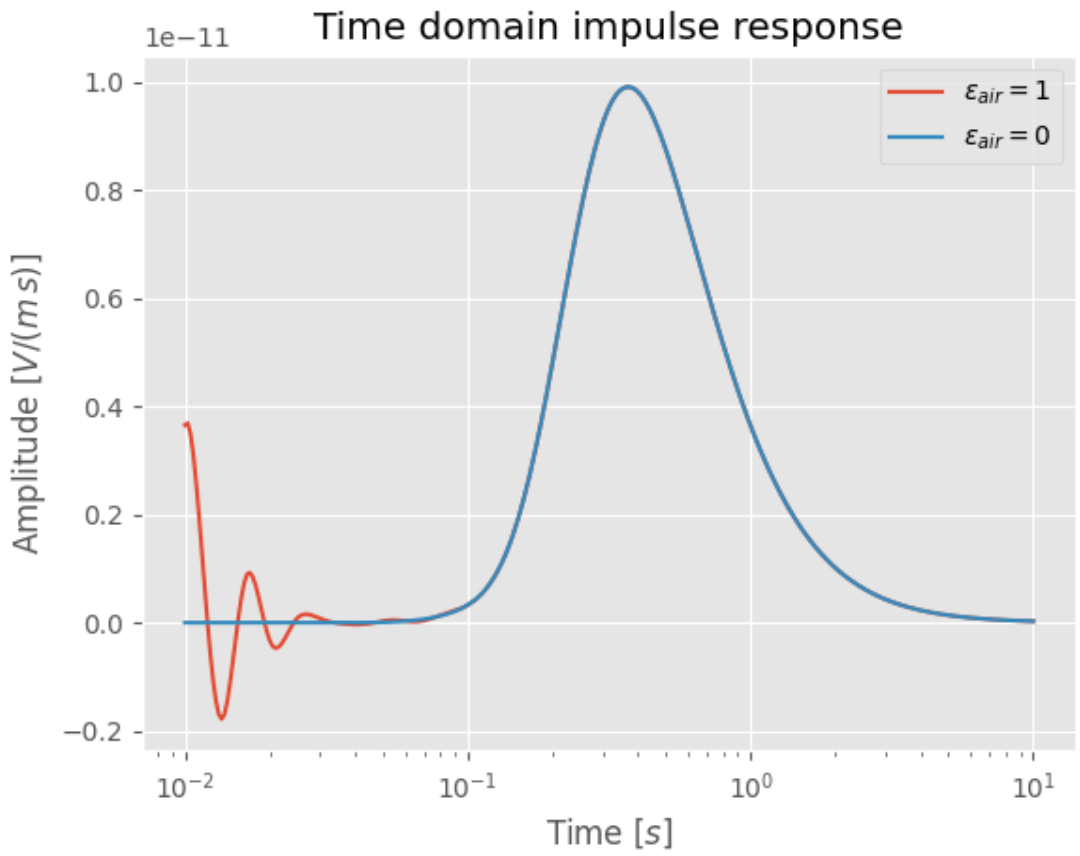
As it can be seen, setting $\varepsilon_{air} = 0$ improves the land CSEM result significantly for earlier times, where the signal should be zero.

```
plt.figure()

plt.title('Time domain impulse response')
plt.semilogx(t, res_1, label=r'$\varepsilon_{air}=1$')
plt.semilogx(t, res_0, label=r'$\varepsilon_{air}=0$')
plt.xlabel(r'Time $[s]$')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel(r'Amplitude $[V/(m\,s)]$')
plt.legend()

plt.show()
```



Version 1.7.0 and older

If you use a version of *empymod* that is smaller than *1.7.1* and set ε_H, ε_V to zero you will see the following warning,

```
* WARNING :: Parameter epermH < 1e-20  are set to 1e-20 !
* WARNING :: Parameter epermV < 1e-20  are set to 1e-20 !
```

and the values will be re-set to the defined minimum value, which is by default 1e-20. Using a value of 1e-20 for *epermH*/*epermV* is also working just fine for land CSEM.

It is possible to change the minimum to zero for these old versions of *empymod*. However, there is a caveat: In *empymod v1.7.0* and older, the *min_param*-parameter also checks frequency and anisotropy. If you set *min_param=0*, and provide *empymod* with resistivities or anisotropies equals to zero, *empymod* will crash due to division by zero errors (avoiding division by zero is the purpose behind the *min_param*-parameter).

To change the *min_param*-parameter do:

```
import empymod
empymod.set_minimum(min_param=0)
```

```
empymod.Report()
```

Total running time of the script: (0 minutes 1.599 seconds)

Estimated memory usage: 8 MB

Step and impulse responses

These examples compare the analytical solution with *empymod* for time-domain step and impulse responses for inline, x-directed source and receivers, for the four different frequency-to-time methods **QWE**, **DLF**, **FFTLog**, and **FFT**. Which method is faster and which is more precise depends on the model (land or marine, source/receiver at air-interface or not) and the response (step or impulse).

```
import empymod
import numpy as np
from scipy.special import erf
import matplotlib.pyplot as plt
from scipy.constants import mu_0        # Permeability of free space  [H/m]
plt.style.use('ggplot')
colors = [color['color'] for color in list(plt.rcParams['axes.prop_cycle'])]
```

Analytical solutions

Analytical solution for source and receiver at the interface between two half-spaces

The time-domain step and impulse responses for a source at the origin ($x_s = y_s = z_s = 0$ m) and an in-line receiver at the surface ($y_r = z_r = 0$ m), is given by the following equations, where ρ_h is horizontal resistivity (Ω m), λ is anisotropy (-), with $\lambda = \sqrt{\rho_v/\rho_h}$, r is offset (m), t is time (s), and $\tau_h = \sqrt{\mu_0 r^2/(\rho_h t)}$; μ_0 is the magnetic permeability of free space (H/m).

Time Domain: Step Response $\mathcal{H}(\mathrm{t})$

$$E_x(\rho_h, \lambda, r, t) = \frac{\rho_h}{2\pi r^3}\left[2\lambda + \mathrm{erf}\left(\frac{\tau_\mathrm{h}}{2}\right) - 2\lambda\mathrm{erf}\left(\frac{\tau_\mathrm{h}}{2\lambda}\right) + \frac{\tau_\mathrm{h}}{\sqrt{\pi}}\exp\left(-\frac{\tau_\mathrm{h}^2}{4\lambda^2}\right)\right]$$

Time Domain: Impulse Response $\delta(\mathrm{t})$

$$\dot{E}_x(\rho_h, \lambda, r, t) = \frac{\rho_h}{2\pi r^3}\left[\delta(t) + \frac{\tau_h}{2t\sqrt{\pi}}\left\{-\exp\left(-\frac{\tau_h^2}{4}\right) + \left(\frac{\tau_h^2}{2\lambda^2} + 1\right)\exp\left(-\frac{\tau_h^2}{4\lambda^2}\right)\right\}\right]$$

Reference

Equations 3.2 and 3.3 in Werthmüller, D., 2009, Inversion of multi-transient EM data from anisotropic media: M.S. thesis, TU Delft, ETH Zürich, RWTH Aachen; UUID: f4b071c1-8e55-4ec5-86c6-a2d54c3eda5a.

Analytical functions

```
def ee_xx_impulse(res, aniso, off, time):
    """VTI-Halfspace impulse response, xx, inline.

    res   : horizontal resistivity [Ohm.m]
    aniso : anisotropy [-]
```

(continues on next page)

(continued from previous page)

```
    off   : offset [m]
    time  : time(s) [s]
    """
    tau_h = np.sqrt(mu_0*off**2/(res*time))
    t0 = tau_h/(2*time*np.sqrt(np.pi))
    t1 = np.exp(-tau_h**2/4)
    t2 = tau_h**2/(2*aniso**2) + 1
    t3 = np.exp(-tau_h**2/(4*aniso**2))
    Exx = res/(2*np.pi*off**3)*t0*(-t1 + t2*t3)
    Exx[time == 0] = res/(2*np.pi*off**3)  # Delta dirac part
    return Exx


def ee_xx_step(res, aniso, off, time):
    """VTI-Halfspace step response, xx, inline.

    res   : horizontal resistivity [Ohm.m]
    aniso : anisotropy [-]
    off   : offset [m]
    time  : time(s) [s]
    """
    tau_h = np.sqrt(mu_0*off**2/(res*time))
    t0 = erf(tau_h/2)
    t1 = 2*aniso*erf(tau_h/(2*aniso))
    t2 = tau_h/np.sqrt(np.pi)*np.exp(-tau_h**2/(4*aniso**2))
    Exx = res/(2*np.pi*off**3)*(2*aniso + t0 - t1 + t2)
    return Exx
```

Example 1: Source and receiver at z=0m

Comparison with analytical solution; put 1 mm below the interface, as they would be regarded as in the air by *emmod* otherwise.

```
src = [0, 0, 0.001]           # Source at origin, slightly below interface
rec = [6000, 0, 0.001]        # Receivers in-line, 0.5m below interface
res = [2e14, 10]              # Resistivity: [air, half-space]
aniso = [1, 2]                # Anisotropy: [air, half-space]
eperm = [0, 1]                # Set el. perm. of air to 0 because of num. noise
t = np.logspace(-2, 1, 301)   # Desired times (s)

# Collect parameters
inparg = {'src': src, 'rec': rec, 'depth': 0, 'freqtime': t, 'res': res,
          'aniso': aniso, 'epermH': eperm, 'ht': 'dlf', 'verb': 2}
```

Impulse response

```
ex = ee_xx_impulse(res[1], aniso[1], rec[0], t)

inparg['signal'] = 0  # signal 0 = impulse
print('QWE')
qwe = empymod.dipole(**inparg, ft='qwe')
print('DLF (Sine)')
sin = empymod.dipole(**inparg, ft='dlf', ftarg={'dlf': 'key_81_CosSin_2009'})
print('FFTLog')
ftl = empymod.dipole(**inparg, ft='fftlog')
print('FFT')
fft = empymod.dipole(
```

(continues on next page)

(continued from previous page)

```
        **inparg, ft='fft',
        ftarg={'dfreq': .0005, 'nfreq': 2**20, 'pts_per_dec': 10})
```

Out:

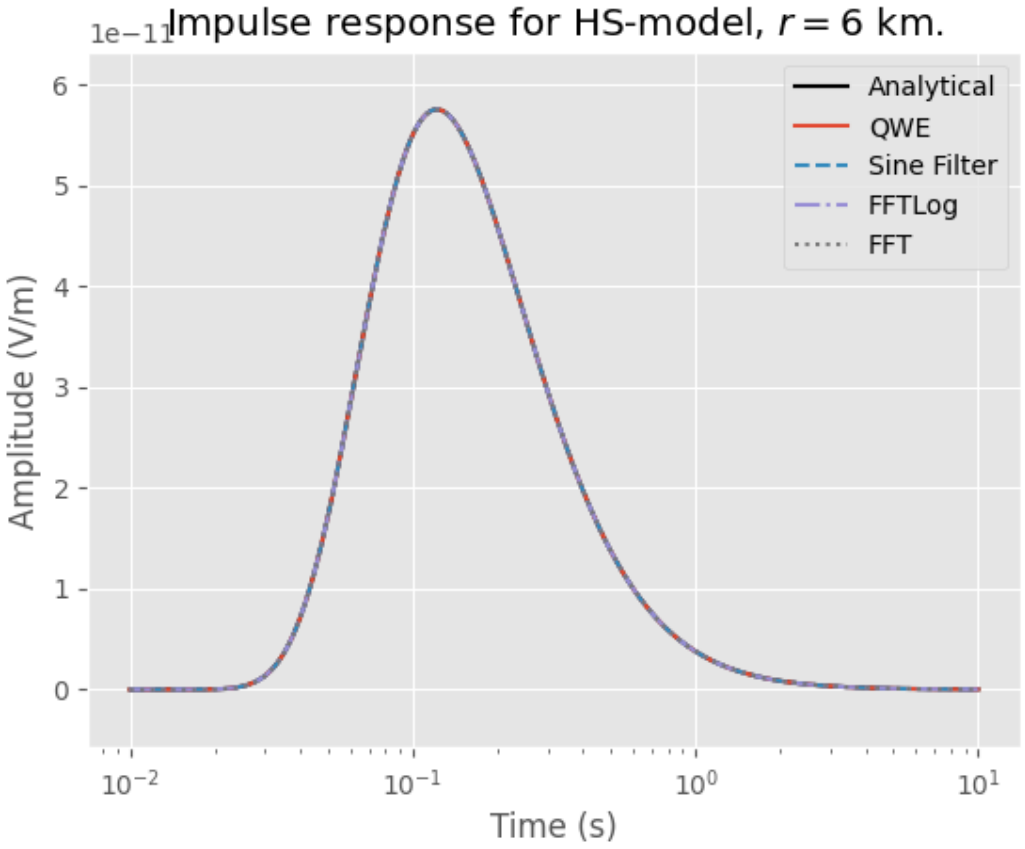
```
QWE
* WARNING :: Fourier-quadrature did not converge at least once;
             => desired `atol` and `rtol` might not be achieved.

:: empymod END; runtime = 0:00:04.216296 :: 1 kernel call(s)

DLF (Sine)

:: empymod END; runtime = 0:00:00.012680 :: 1 kernel call(s)

FFTLog

:: empymod END; runtime = 0:00:00.006852 :: 1 kernel call(s)

FFT

:: empymod END; runtime = 0:00:00.792403 :: 1 kernel call(s)
```

=> *FFTLog* is the fastest by quite a margin, followed by the *Sine*-filter. What cannot see from the output (set *verb* to something bigger than 2 to see it) is how many frequencies each method used:

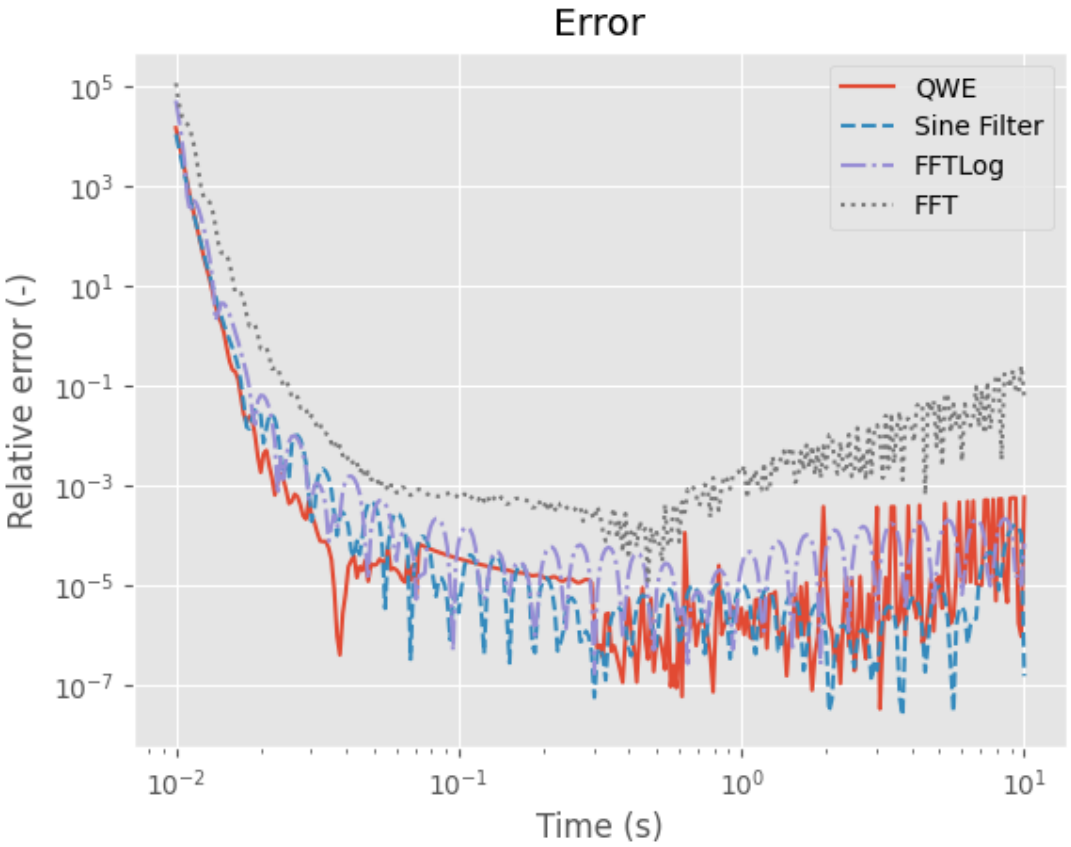
- QWE: 159 (0.000794328 - 63095.7 Hz)
- Sine: 116 (5.33905E-06 - 52028 Hz)
- FFTLog: 60 (0.000178575 - 141.847 Hz)
- FFT: 61 (0.0005 - 524.288 Hz)

Note that for the actual transform, *FFT* used 2^20 = 1‘048‘576 frequencies! It only computed 60 frequencies, and then interpolated the rest, as it requires regularly spaced data.

```
plt.figure()
plt.title(r'Impulse response for HS-model, $r=$' +
          str(int(rec[0]/1000)) + ' km.')
plt.xlabel('Time (s)')
plt.ylabel(r'Amplitude (V/m)')
plt.semilogx(t, ex, 'k-', label='Analytical')
plt.semilogx(t, qwe, 'C0-', label='QWE')
plt.semilogx(t, sin, 'C1--', label='Sine Filter')
plt.semilogx(t, ftl, 'C2-.', label='FFTLog')
plt.semilogx(t, fft, 'C3:', label='FFT')
plt.legend(loc='best')
plt.ylim([-.1*np.max(ex), 1.1*np.max(ex)])
plt.show()
```




```
plt.figure()
plt.title('Error')
plt.xlabel('Time (s)')
plt.ylabel('Relative error (-)')
plt.loglog(t, abs(qwe-ex)/ex, 'C0-', label='QWE')
plt.plot(t, abs(sin-ex)/ex, 'C1--', label='Sine Filter')
plt.plot(t, abs(ftl-ex)/ex, 'C2-.', label='FFTLog')
plt.plot(t, abs(fft-ex)/ex, 'C3:', label='FFT')
plt.legend(loc='best')
plt.show()
```



=> The error is comparable in all cases. *FFT* is not too good at later times. This could be improved by computing lower frequencies. But because FFT needs regularly spaced data, our vector would soon explode (and you would need a lot of memory). In the current case we are already using 2^20 samples!

Step response

Step responses are almost impossible with *FFT*. We can either try to model late times with lots of low frequencies, or the step with lots of high frequencies. I do not use *FFT* in the step-response examples.

Switch-on

```
ex = ee_xx_step(res[1], aniso[1], rec[0], t)

inparg['signal'] = 1  # signal 1 = switch-on
print('QWE')
qwe = empymod.dipole(**inparg, ft='qwe')
print('DLF (Sine)')
sin = empymod.dipole(**inparg, ft='dlf', ftarg={'dlf': 'key_81_CosSin_2009'})
print('FFTLog')
ftl = empymod.dipole(**inparg, ft='fftlog', ftarg={'q': -0.6})
```

Out:

```
QWE
* WARNING :: Fourier-quadrature did not converge at least once;
             => desired `atol` and `rtol` might not be achieved.
```

(continues on next page)

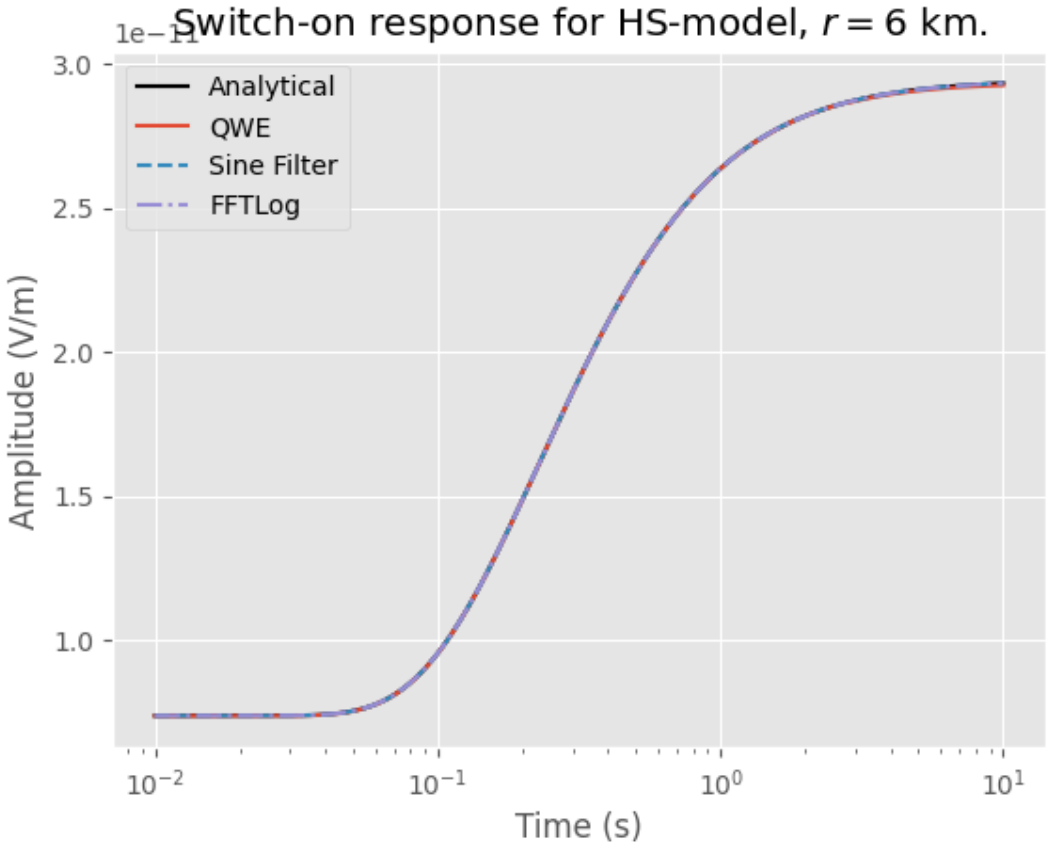
(continued from previous page)

```
:: empymod END; runtime = 0:00:17.664912 :: 1 kernel call(s)

DLF (Sine)

:: empymod END; runtime = 0:00:00.013355 :: 1 kernel call(s)

FFTLog

:: empymod END; runtime = 0:00:00.007041 :: 1 kernel call(s)
```

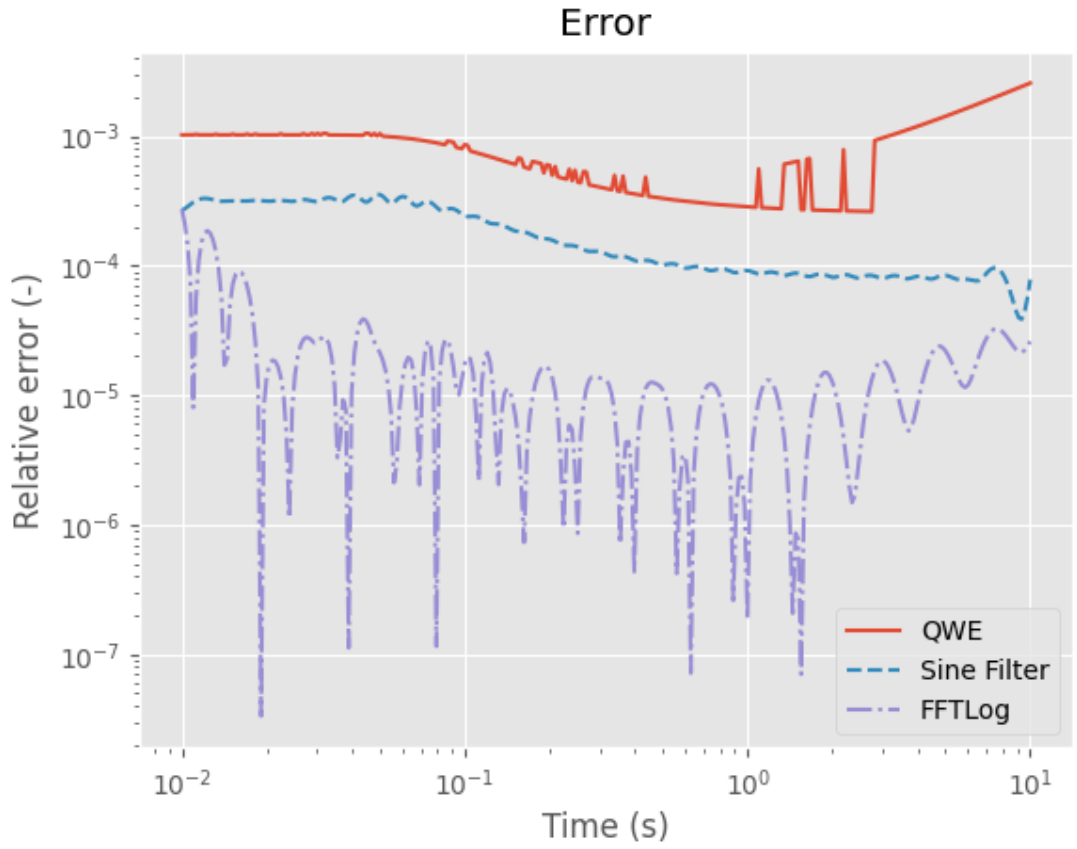
Used number of frequencies:

- QWE: 159 (0.000794328 - 63095.7 Hz)
- Sine: 116 (5.33905E-06 - 52028 Hz)
- FFTLog: 60 (0.000178575 - 141.847 Hz)

```
plt.figure()
plt.title(r'Switch-on response for HS-model, $r=$' +
          str(int(rec[0]/1000)) + ' km.')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude (V/m)')
plt.semilogx(t, ex, 'k-', label='Analytical')
plt.semilogx(t, qwe, 'C0-', label='QWE')
plt.semilogx(t, sin, 'C1--', label='Sine Filter')
plt.semilogx(t, ftl, 'C2-.', label='FFTLog')
plt.legend(loc='best')
plt.show()
```



```
plt.figure()
plt.title('Error')
plt.xlabel('Time (s)')
plt.ylabel('Relative error (-)')
plt.loglog(t, abs(qwe-ex)/ex, 'C0-', label='QWE')
plt.plot(t, abs(sin-ex)/ex, 'C1--', label='Sine Filter')
plt.plot(t, abs(ftl-ex)/ex, 'C2-.', label='FFTLog')
plt.legend(loc='best')
plt.show()
```



Switch-off

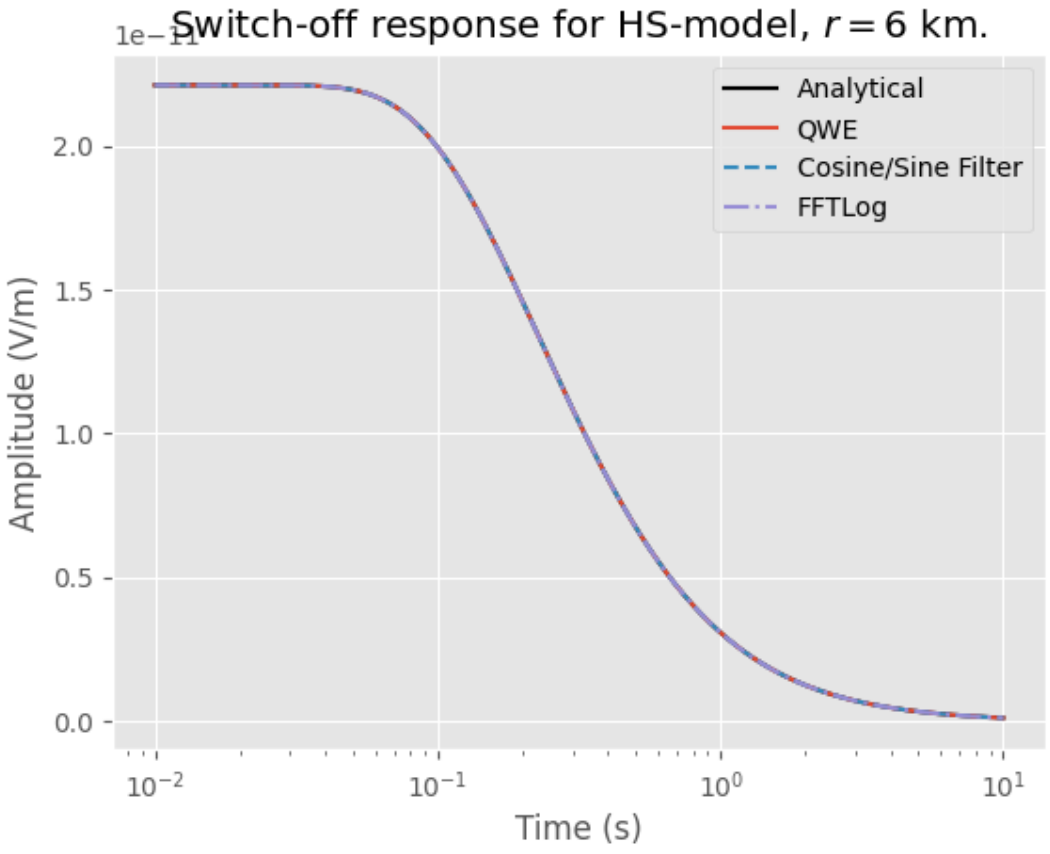
For switch-off to work properly you need *empymod*-version bigger than 1.3.0! You can do it with previous releases too, but you will have to do the DC-computation and subtraction manually, as is done here for *ee_xx_step*.

```
exDC = ee_xx_step(res[1], aniso[1], rec[0], 60*60)
ex = exDC - ee_xx_step(res[1], aniso[1], rec[0], t)

inparg['signal'] = -1  # signal -1 = switch-off
print('QWE')
qwe = empymod.dipole(**inparg, ft='qwe')
print('DLF (Cosine/Sine)')
sin = empymod.dipole(**inparg, ft='dlf', ftarg={'dlf': 'key_81_CosSin_2009'})
print('FFTLog')
ftl = empymod.dipole(**inparg, ft='fftlog', ftarg={'add_dec': [-5, 3]})
```

Out:

```
QWE
* WARNING :: Fourier-quadrature did not converge at least once;
             => desired `atol` and `rtol` might not be achieved.

:: empymod END; runtime = 0:00:13.112963 :: 1 kernel call(s)

DLF (Cosine/Sine)

:: empymod END; runtime = 0:00:00.013442 :: 1 kernel call(s)

FFTLog

:: empymod END; runtime = 0:00:00.011433 :: 1 kernel call(s)
```

```
plt.figure()
plt.title(r'Switch-off response for HS-model, $r=$' +
          str(int(rec[0]/1000)) + ' km.')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude (V/m)')
plt.semilogx(t, ex, 'k-', label='Analytical')
plt.semilogx(t, qwe, 'C0-', label='QWE')
plt.semilogx(t, sin, 'C1--', label='Cosine/Sine Filter')
plt.semilogx(t, ftl, 'C2-.', label='FFTLog')
plt.legend(loc='best')
plt.show()
```

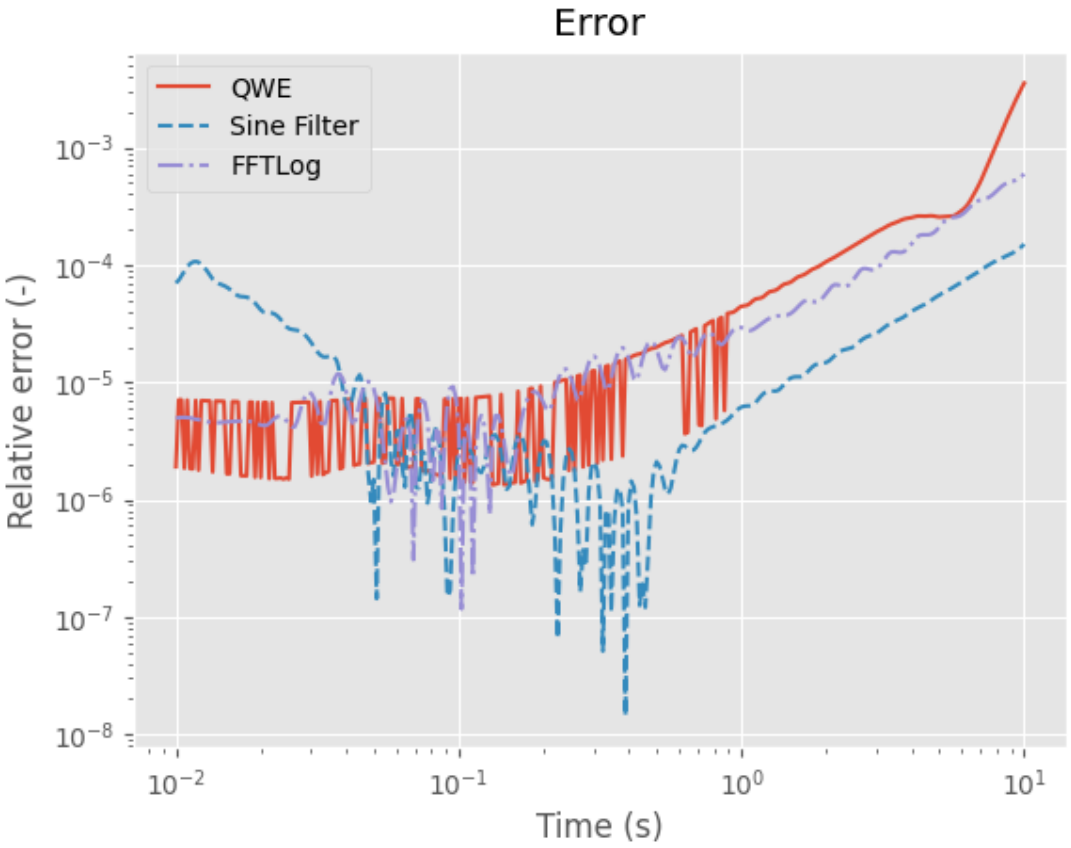



```
plt.figure()
plt.title('Error')
plt.xlabel('Time (s)')
plt.ylabel('Relative error (-)')
```

(continues on next page)

(continued from previous page)

```
plt.loglog(t, abs(qwe-ex)/ex, 'C0-', label='QWE')
plt.plot(t, abs(sin-ex)/ex, 'C1--', label='Sine Filter')
plt.plot(t, abs(ftl-ex)/ex, 'C2-.', label='FFTLog')
plt.legend(loc='best')
plt.show()
```



Example 2: Air-seawater-halfspace

In seawater the transformation is generally much easier, as we do not have the step or the impules at zero time.

```
src = [0, 0, 950]              # Source 50 m above seabottom
rec = [6000, 0, 1000]          # Receivers in-line, at seabottom
res = [1e23, 1/3, 10]          # Resistivity: [air, water, half-space]
aniso = [1, 1, 2]              # Anisotropy: [air, water, half-space]
t = np.logspace(-2, 1, 301)  # Desired times (s)

# Collect parameters
inparg = {'src': src, 'rec': rec, 'depth': [0, 1000], 'freqtime': t,
          'res': res, 'aniso': aniso, 'ht': 'dlf', 'verb': 2}
```

Impulse response

```
inparg['signal'] = 0  # signal 0 = impulse
print('QWE')
qwe = empymod.dipole(**inparg, ft='qwe', ftarg={'maxint': 500})
print('DLF (Sine)')
```

(continues on next page)

(continued from previous page)

```
sin = empymod.dipole(**inparg, ft='dlf', ftarg={'dlf': 'key_81_CosSin_2009'})
print('FFTLog')
ftl = empymod.dipole(**inparg, ft='fftlog')
print('FFT')
fft = empymod.dipole(
        **inparg, ft='fft',
        ftarg={'dfreq': .001, 'nfreq': 2**15, 'ntot': 2**16, 'pts_per_dec': 10}
        )
```

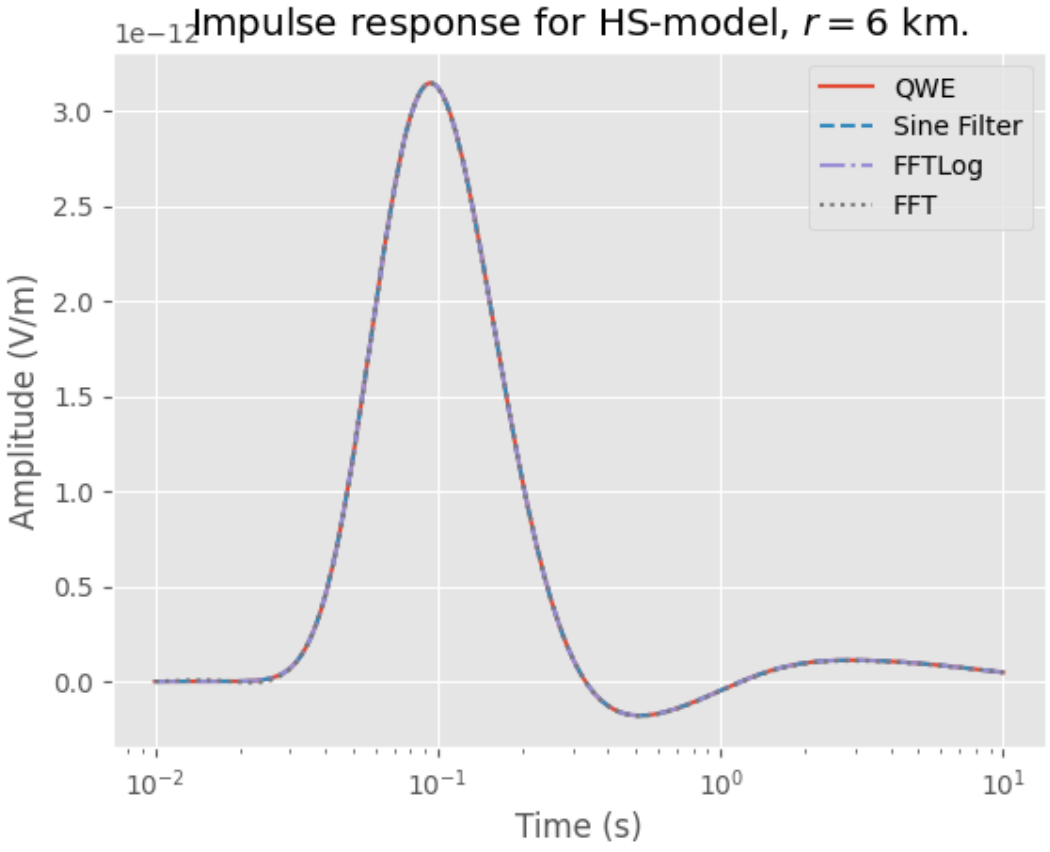
Out:

```
QWE

:: empymod END; runtime = 0:00:18.979857 :: 1 kernel call(s)

DLF (Sine)

:: empymod END; runtime = 0:00:00.042693 :: 1 kernel call(s)

FFTLog

:: empymod END; runtime = 0:00:00.020483 :: 1 kernel call(s)

FFT

:: empymod END; runtime = 0:00:00.050903 :: 1 kernel call(s)
```

Used number of frequencies:

- QWE: 167 (0.000794328 - 158489 Hz)
- Sine: 116 (5.33905E-06 - 52028 Hz)
- FFTLog: 60 (0.000178575 - 141.847 Hz)
- FFT: 46 (0.001 - 32.768 Hz)

```
plt.figure()
plt.title(r'Impulse response for HS-model, $r=$' +
          str(int(rec[0]/1000)) + ' km.')
plt.xlabel('Time (s)')
plt.ylabel(r'Amplitude (V/m)')
plt.semilogx(t, qwe, 'C0-', label='QWE')
plt.semilogx(t, sin, 'C1--', label='Sine Filter')
plt.semilogx(t, ftl, 'C2-.', label='FFTLog')
plt.semilogx(t, fft, 'C3:', label='FFT')
plt.legend(loc='best')
plt.show()
```



Step response

```
inparg['signal'] = 1  # signal 1 = step
print('QWE')
qwe = empymod.dipole(**inparg, ft='qwe', ftarg={'nquad': 31, 'maxint': 500})
print('DLF (Sine)')
sin = empymod.dipole(**inparg, ft='dlf', ftarg={'dlf': 'key_81_CosSin_2009'})
print('FFTLog')
ftl = empymod.dipole(**inparg, ft='fftlog', ftarg={'add_dec': [-2, 4]})
```

Out:

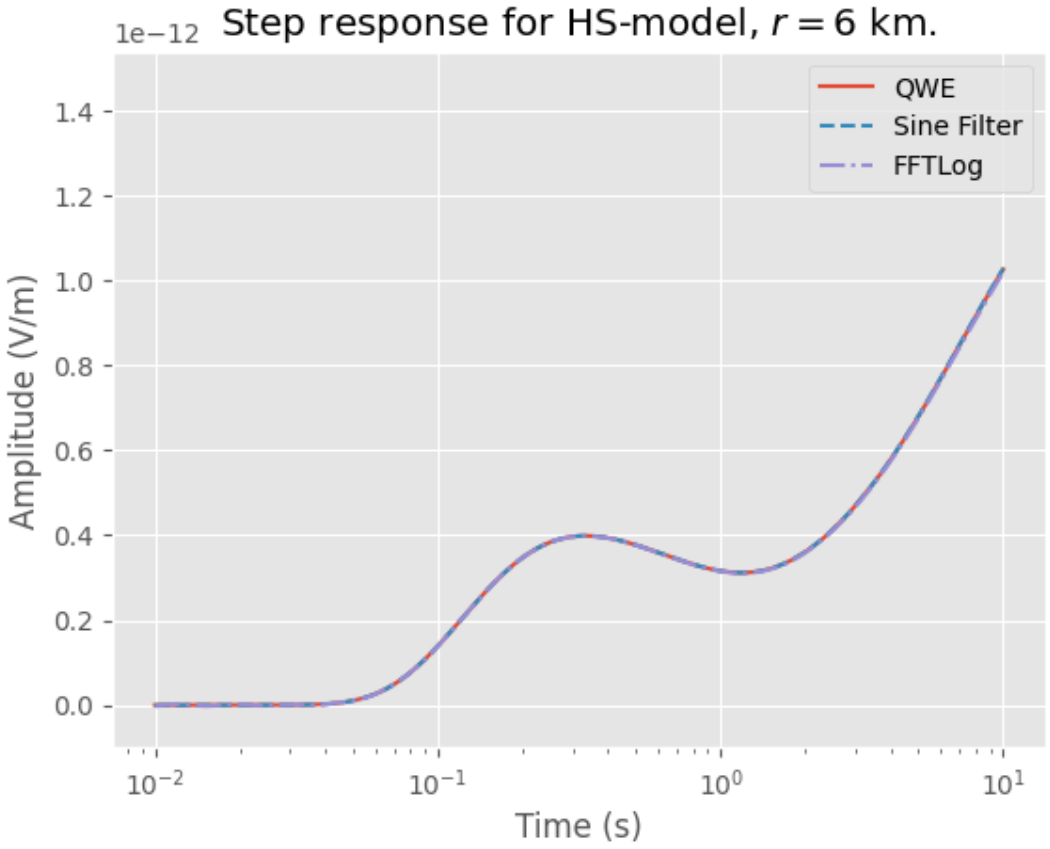
```
QWE
* WARNING :: Fourier-quadrature did not converge at least once;
             => desired `atol` and `rtol` might not be achieved.

:: empymod END; runtime = 0:00:15.977281 :: 1 kernel call(s)

DLF (Sine)

:: empymod END; runtime = 0:00:00.037453 :: 1 kernel call(s)

FFTLog

:: empymod END; runtime = 0:00:00.028312 :: 1 kernel call(s)
```

Used number of frequencies:

- QWE: 173 (0.000398107 - 158489 Hz)

- Sine: 116 (5.33905E-06 - 52028 Hz)
- FFTLog: 90 (0.000178575 - 141847 Hz)

```
plt.figure()
plt.title(r'Step response for HS-model, $r=$' + str(int(rec[0]/1000)) + ' km.')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude (V/m)')
plt.semilogx(t, qwe, 'C0-', label='QWE')
plt.semilogx(t, sin, 'C1--', label='Sine Filter')
plt.semilogx(t, ftl, 'C2-.', label='FFTLog')
plt.ylim([-.1e-12, 1.5*qwe.max()])
plt.legend(loc='best')
plt.show()
```




```
empymod.Report()
```

Total running time of the script: (1 minutes 16.612 seconds)

Estimated memory usage: 311 MB

TEM: ABEM WalkTEM

The modeller `empymod` models the electromagnetic (EM) full wavefield Greens function for electric and magnetic point sources and receivers. As such, it can model any EM method from DC to GPR. However, how to actually implement a particular EM method and survey layout can be tricky, as there are many more things involved than just computing the EM Greens function.

In this example we are going to compute a TEM response, in particular from the system WalkTEM, and compare it with data obtained from AarhusInv. However, you can use and adapt this example to model other TEM systems, such as skyTEM, SIROTEM, TEM-FAST, or any other system.

What is not included in `empymod` at this moment (but hopefully in the future), but is required to model TEM data, is to **account for arbitrary source waveform**, and to apply a **lowpass filter**. So we generate these two things here, and create our own wrapper to model TEM data.

The incentive for this example came from Leon Foks (@leonfoks) for GeoBIPy, and it was created with his help and also the help of Seogi Kang (@sgkang) from simpegEM1D; the waveform function is based on work from Kerry Key (@kerrykey) from emlab.

```
import empymod
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import LogLocator, NullFormatter
from scipy.integrate.quadrature import _cached_roots_legendre
from scipy.interpolate import InterpolatedUnivariateSpline as iuSpline
plt.style.use('ggplot')
```

1. AarhusInv data

The comparison data was created by Leon Foks using AarhusInv.

Off times (when measurement happens)

```
# Low moment
lm_off_time = np.array([
    1.149E-05, 1.350E-05, 1.549E-05, 1.750E-05, 2.000E-05, 2.299E-05,
    2.649E-05, 3.099E-05, 3.700E-05, 4.450E-05, 5.350E-05, 6.499E-05,
    7.949E-05, 9.799E-05, 1.215E-04, 1.505E-04, 1.875E-04, 2.340E-04,
    2.920E-04, 3.655E-04, 4.580E-04, 5.745E-04, 7.210E-04
])

# High moment
hm_off_time = np.array([
    9.810e-05, 1.216e-04, 1.506e-04, 1.876e-04, 2.341e-04, 2.921e-04,
    3.656e-04, 4.581e-04, 5.746e-04, 7.211e-04, 9.056e-04, 1.138e-03,
    1.431e-03, 1.799e-03, 2.262e-03, 2.846e-03, 3.580e-03, 4.505e-03,
    5.670e-03, 7.135e-03
])
```

Data resistive model

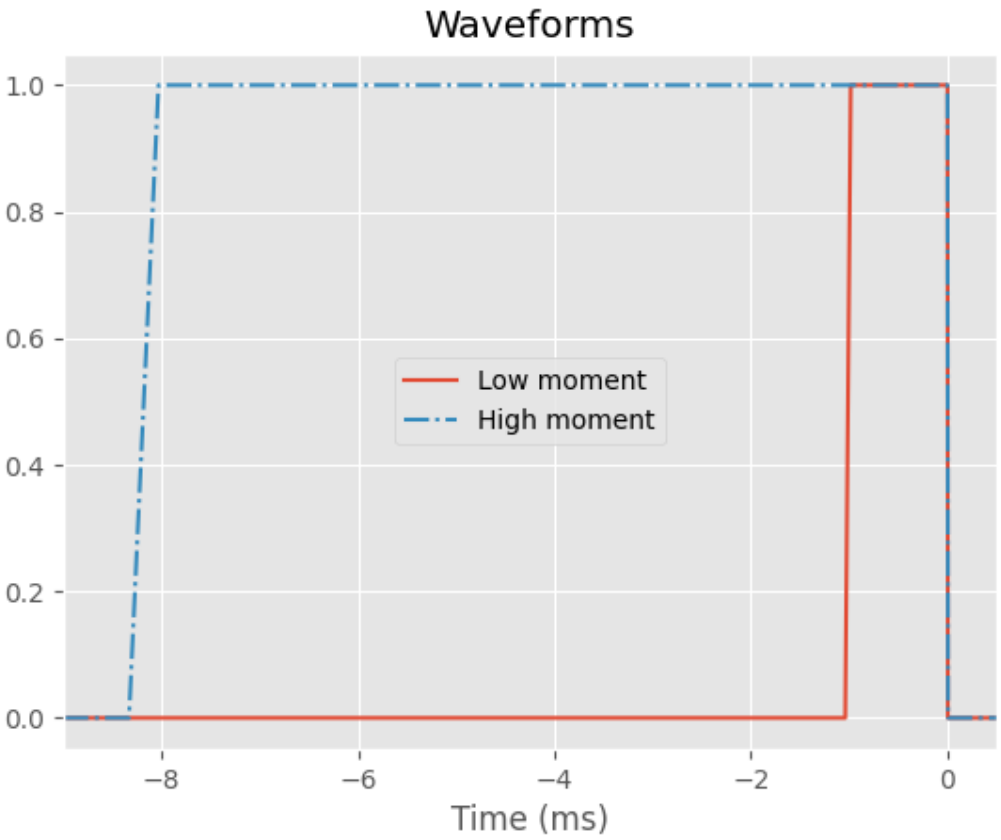
```
# Low moment
lm_aarhus_res = np.array([
    7.980836E-06, 4.459270E-06, 2.909954E-06, 2.116353E-06, 1.571503E-06,
    1.205928E-06, 9.537814E-07, 7.538660E-07, 5.879494E-07, 4.572059E-07,
    3.561824E-07, 2.727531E-07, 2.058368E-07, 1.524225E-07, 1.107586E-07,
    7.963634E-08, 5.598970E-08, 3.867087E-08, 2.628711E-08, 1.746382E-08,
    1.136561E-08, 7.234771E-09, 4.503902E-09
])

# High moment
hm_aarhus_res = np.array([
    1.563517e-07, 1.139461e-07, 8.231679e-08, 5.829438e-08, 4.068236e-08,
    2.804896e-08, 1.899818e-08, 1.268473e-08, 8.347439e-09, 5.420791e-09,
    3.473876e-09, 2.196246e-09, 1.372012e-09, 8.465165e-10, 5.155328e-10,
    3.099162e-10, 1.836829e-10, 1.072522e-10, 6.161256e-11, 3.478720e-11
])
```

Data conductive model

```
# Low moment
lm_aarhus_con = np.array([
    1.046719E-03, 7.712241E-04, 5.831951E-04, 4.517059E-04, 3.378510E-04,
    2.468364E-04, 1.777187E-04, 1.219521E-04, 7.839379E-05, 4.861241E-05,
    2.983254E-05, 1.778658E-05, 1.056006E-05, 6.370305E-06, 3.968808E-06,
    2.603794E-06, 1.764719E-06, 1.218968E-06, 8.483796E-07, 5.861686E-07,
    3.996331E-07, 2.678636E-07, 1.759663E-07
])

# High moment
hm_aarhus_con = np.array([
    6.586261e-06, 4.122115e-06, 2.724062e-06, 1.869149e-06, 1.309683e-06,
    9.300854e-07, 6.588088e-07, 4.634354e-07, 3.228131e-07, 2.222540e-07,
    1.509422e-07, 1.010134e-07, 6.662953e-08, 4.327995e-08, 2.765871e-08,
    1.738750e-08, 1.073843e-08, 6.512053e-09, 3.872709e-09, 2.256841e-09
])
```

WalkTEM Waveform and other characteristics

```
# Low moment
lm_waveform_times = np.r_[-1.041E-03, -9.850E-04, 0.000E+00, 4.000E-06]
lm_waveform_current = np.r_[0.0, 1.0, 1.0, 0.0]

# High moment
hm_waveform_times = np.r_[-8.333E-03, -8.033E-03, 0.000E+00, 5.600E-06]
hm_waveform_current = np.r_[0.0, 1.0, 1.0, 0.0]

plt.figure()
plt.title('Waveforms')
plt.plot(np.r_[-9, lm_waveform_times*1e3, 2], np.r_[0, lm_waveform_current, 0],
         label='Low moment')
plt.plot(np.r_[-9, hm_waveform_times*1e3, 2], np.r_[0, hm_waveform_current, 0],
         '-.', label='High moment')
plt.xlabel('Time (ms)')
plt.xlim([-9, 0.5])
plt.legend()
plt.show()
```



2. `empymod` implementation

```
def waveform(times, resp, times_wanted, wave_time, wave_amp, nquad=3):
    """Apply a source waveform to the signal.

    Parameters
    ----------
    times : ndarray
        Times of computed input response; should start before and end after
        `times_wanted`.

    resp : ndarray
        EM-response corresponding to `times`.

    times_wanted : ndarray
        Wanted times.

    wave_time : ndarray
        Time steps of the wave.

    wave_amp : ndarray
        Amplitudes of the wave corresponding to `wave_time`, usually
        in the range of [0, 1].

    nquad : int
        Number of Gauss-Legendre points for the integration. Default is 3.

    Returns
    -------
```

(continues on next page)

(continued from previous page)

```
    resp_wanted : ndarray
        EM field for `times_wanted`.

    """

    # Interpolate on log.
    PP = iuSpline(np.log10(times), resp)

    # Wave time steps.
    dt = np.diff(wave_time)
    dI = np.diff(wave_amp)
    dIdt = dI/dt

    # Gauss-Legendre Quadrature; 3 is generally good enough.
    g_x, g_w = _cached_roots_legendre(nquad)

    # Pre-allocate output.
    resp_wanted = np.zeros_like(times_wanted)

    # Loop over wave segments.
    for i, cdIdt in enumerate(dIdt):

        # We only have to consider segments with a change of current.
        if cdIdt == 0.0:
            continue

        # If wanted time is before a wave element, ignore it.
        ind_a = wave_time[i] < times_wanted
        if ind_a.sum() == 0:
            continue

        # If wanted time is within a wave element, we cut the element.
        ind_b = wave_time[i+1] > times_wanted[ind_a]

        # Start and end for this wave-segment for all times.
        ta = times_wanted[ind_a]-wave_time[i]
        tb = times_wanted[ind_a]-wave_time[i+1]
        tb[ind_b] = 0.0  # Cut elements

        # Gauss-Legendre for this wave segment. See
        # https://en.wikipedia.org/wiki/Gaussian_quadrature#Change_of_interval
        # for the change of interval, which makes this a bit more complex.
        logt = np.log10(np.outer((tb-ta)/2, g_x)+(ta+tb)[:, None]/2)
        fact = (tb-ta)/2*cdIdt
        resp_wanted[ind_a] += fact*np.sum(np.array(PP(logt)*g_w), axis=1)

    return resp_wanted
```

```
def get_time(time, r_time):
    """Additional time for ramp.

    Because of the arbitrary waveform, we need to compute some times before and
    after the actually wanted times for interpolation of the waveform.

    Some implementation details: The actual times here don't really matter. We
    create a vector of time.size+2, so it is similar to the input times and
    accounts that it will require a bit earlier and a bit later times. Really
    important are only the minimum and maximum times. The Fourier DLF, with
    `pts_per_dec=-1`, computes times from minimum to at least the maximum,
    where the actual spacing is defined by the filter spacing. It subsequently
    interpolates to the wanted times. Afterwards, we interpolate those again to
```

(continues on next page)

(continued from previous page)

```
    compute the actual waveform response.

    Note: We could first call `waveform`, and get the actually required times
          from there. This would make this function obsolete. It would also
          avoid the double interpolation, first in `empymod.model.time` for the
          Fourier DLF with `pts_per_dec=-1`, and second in `waveform`. Doable.
          Probably not or marginally faster. And the code would become much
          less readable.

    Parameters
    ----------
    time : ndarray
        Desired times

    r_time : ndarray
        Waveform times

    Returns
    -------
    time_req : ndarray
        Required times
    """
    tmin = np.log10(max(time.min()-r_time.max(), 1e-10))
    tmax = np.log10(time.max()-r_time.min())
    return np.logspace(tmin, tmax, time.size+2)
```

```
def walktem(moment, depth, res):
    """Custom wrapper of empymod.model.bipole.

    Here, we compute WalkTEM data using the ``empymod.model.bipole`` routine as
    an example. We could achieve the same using ``empymod.model.dipole`` or
    ``empymod.model.loop``.

    We model the big source square loop by computing only half of one side of
    the electric square loop and approximating the finite length dipole with 3
    point dipole sources. The result is then multiplied by 8, to account for
    all eight half-sides of the square loop.

    The implementation here assumes a central loop configuration, where the
    receiver (1 m2 area) is at the origin, and the source is a 40x40 m electric
    loop, centered around the origin.

    Note: This approximation of only using half of one of the four sides
          obviously only works for central, horizontal square loops. If your
          loop is arbitrary rotated, then you have to model all four sides of
          the loop and sum it up.


    Parameters
    ----------
    moment : str {'lm', 'hm'}
        Moment. If 'lm', above defined ``lm_off_time``, ``lm_waveform_times``,
        and ``lm_waveform_current`` are used. Else, the corresponding
        ``hm_``-parameters.

    depth : ndarray
        Depths of the resistivity model (see ``empymod.model.bipole`` for more
        info.)

    res : ndarray
        Resistivities of the resistivity model (see ``empymod.model.bipole``
```

(continues on next page)

(continued from previous page)

```
    for more info.)

Returns
-------
WalkTEM : EMArray
    WalkTEM response (dB/dt).

"""

# Get the measurement time and the waveform corresponding to the provided
# moment.
if moment == 'lm':
    off_time = lm_off_time
    waveform_times = lm_waveform_times
    waveform_current = lm_waveform_current
elif moment == 'hm':
    off_time = hm_off_time
    waveform_times = hm_waveform_times
    waveform_current = hm_waveform_current
else:
    raise ValueError("Moment must be either 'lm' or 'hm'!")

# === GET REQUIRED TIMES ===
time = get_time(off_time, waveform_times)

# === GET REQUIRED FREQUENCIES ===
time, freq, ft, ftarg = empymod.utils.check_time(
    time=time,          # Required times
    signal=1,           # Switch-on response
    ft='dlf',           # Use DLF
    ftarg={'dlf': 'key_81_CosSin_2009'},  # Short, fast filter; if you
    verb=2,                 # need higher accuracy choose a longer filter.
)

# === COMPUTE FREQUENCY-DOMAIN RESPONSE ===
# We only define a few parameters here. You could extend this for any
# parameter possible to provide to empymod.model.bipole.
EM = empymod.model.bipole(
    src=[20, 20,   0, 20, 0, 0],  # El. bipole source; half of one side.
    rec=[0, 0, 0, 0, 90],         # Receiver at the origin, vertical.
    depth=np.r_[0, depth],        # Depth-model, adding air-interface.
    res=np.r_[2e14, res],         # Provided resistivity model, adding air.
    # aniso=aniso,                # Here you could implement anisotropy...
    #                             # ...or any parameter accepted by bipole.
    freqtime=freq,                # Required frequencies.
    mrec=True,                    # It is an el. source, but a magn. rec.
    strength=8,                   # To account for 4 sides of square loop.
    srcpts=3,                     # Approx. the finite dip. with 3 points.
    htarg={'dlf': 'key_101_2009'},  # Short filter, so fast.
)

# Multiply the frequecny-domain result with
# \mu for H->B, and i\omega for B->dB/dt.
EM *= 2j*np.pi*freq*4e-7*np.pi

# === Butterworth-type filter (implemented from simpegEM1D.Waveforms.py)===
# Note: Here we just apply one filter. But it seems that WalkTEM can apply
#       two filters, one before and one after the so-called front gate
#       (which might be related to ``delay_rst``, I am not sure about that
#       part.)
cutofffreq = 4.5e5              # As stated in the WalkTEM manual
```

(continues on next page)

(continued from previous page)

```
    h = (1+1j*freq/cutofffreq)**-1    # First order type
    h *= (1+1j*freq/3e5)**-1
    EM *= h

    # === CONVERT TO TIME DOMAIN ===
    delay_rst = 1.8e-7                # As stated in the WalkTEM manual
    EM, _ = np.squeeze(empymod.model.tem(EM[:, None], np.array([1]),
                       freq, time+delay_rst, 1, ft, ftarg))

    # === APPLY WAVEFORM ===
    return waveform(time, EM, off_time, waveform_times, waveform_current)
```

3. Computation

```
# Compute resistive model
lm_empymod_res = walktem('lm', depth=[75], res=[500, 20])
hm_empymod_res = walktem('hm', depth=[75], res=[500, 20])

# Compute conductive model
lm_empymod_con = walktem('lm', depth=[30], res=[10, 1])
hm_empymod_con = walktem('hm', depth=[30], res=[10, 1])
```

Out:

```
:: empymod END; runtime = 0:00:00.016766 :: 3 kernel call(s)


:: empymod END; runtime = 0:00:00.012212 :: 3 kernel call(s)


:: empymod END; runtime = 0:00:00.011708 :: 3 kernel call(s)


:: empymod END; runtime = 0:00:00.011348 :: 3 kernel call(s)
```

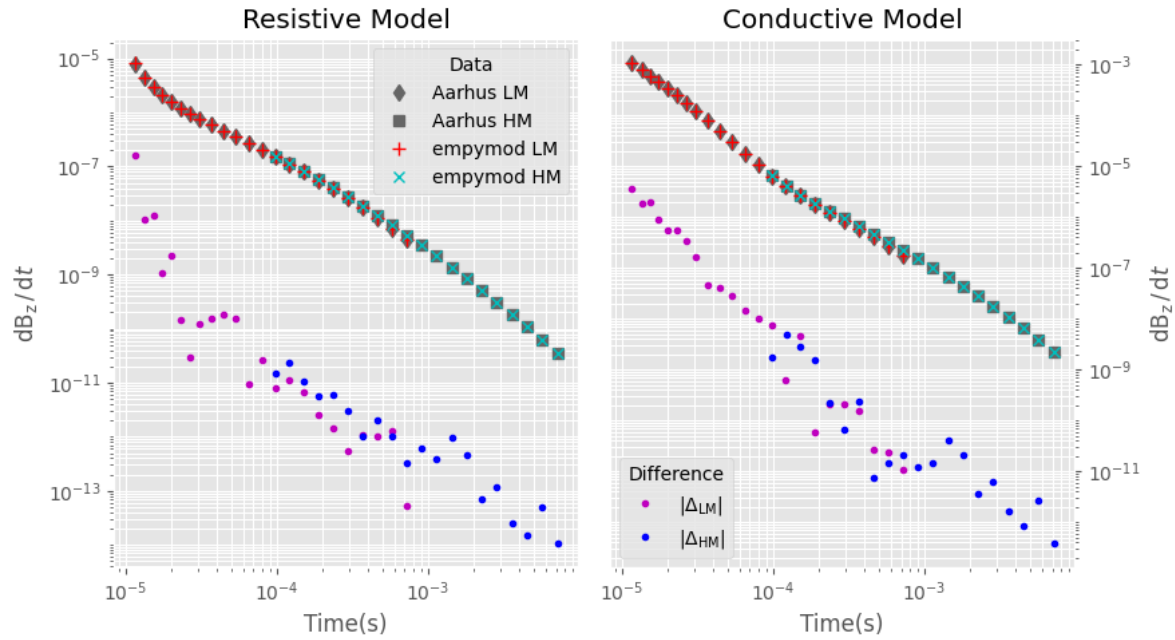
4. Comparison

```
plt.figure(figsize=(9, 5))

# Plot result resistive model
ax1 = plt.subplot(121)
plt.title('Resistive Model')

# AarhusInv
plt.plot(lm_off_time, lm_aarhus_res, 'd', mfc='.4', mec='.4',
         label="Aarhus LM")
plt.plot(hm_off_time, hm_aarhus_res, 's', mfc='.4', mec='.4',
         label="Aarhus HM")

# empymod
plt.plot(lm_off_time, lm_empymod_res, 'r+', ms=7, label="empymod LM")
plt.plot(hm_off_time, hm_empymod_res, 'cx', label="empymod HM")

# Difference
plt.plot(lm_off_time, np.abs((lm_aarhus_res - lm_empymod_res)), 'm.')
plt.plot(hm_off_time, np.abs((hm_aarhus_res - hm_empymod_res)), 'b.')
```

(continues on next page)

(continued from previous page)

```
# Plot settings
plt.xscale('log')
plt.yscale('log')
plt.xlabel("Time(s)")
plt.ylabel(r"$\mathrm{d}\mathrm{B}_\mathrm{z}\,/\,\mathrm{d}t$")
plt.grid(which='both', c='w')
plt.legend(title='Data', loc=1)

# Plot result conductive model
ax2 = plt.subplot(122)
plt.title('Conductive Model')
ax2.yaxis.set_label_position("right")
ax2.yaxis.tick_right()

# AarhusInv
plt.plot(lm_off_time, lm_aarhus_con, 'd', mfc='.4', mec='.4')
plt.plot(hm_off_time, hm_aarhus_con, 's', mfc='.4', mec='.4')

# empymod
plt.plot(lm_off_time, lm_empymod_con, 'r+', ms=7)
plt.plot(hm_off_time, hm_empymod_con, 'cx')

# Difference
plt.plot(lm_off_time, np.abs((lm_aarhus_con - lm_empymod_con)), 'm.',
         label=r"$|\Delta_\mathrm{LM}|$")
plt.plot(hm_off_time, np.abs((hm_aarhus_con - hm_empymod_con)), 'b.',
         label=r"$|\Delta_\mathrm{HM}|$")

# Plot settings
plt.xscale('log')
plt.yscale('log')
plt.xlabel("Time(s)")
plt.ylabel(r"$\mathrm{d}\mathrm{B}_\mathrm{z}\,/\,\mathrm{d}t$")
plt.legend(title='Difference', loc=3)

# Force minor ticks on logscale
ax1.yaxis.set_minor_locator(LogLocator(subs='all', numticks=20))
ax2.yaxis.set_minor_locator(LogLocator(subs='all', numticks=20))
ax1.yaxis.set_minor_formatter(NullFormatter())
ax2.yaxis.set_minor_formatter(NullFormatter())
plt.grid(which='both', c='w')

# Finish off
plt.tight_layout()
plt.show()
```




```
empymod.Report()
```

Total running time of the script: (0 minutes 3.502 seconds)

Estimated memory usage: 8 MB

5.4.4 Comparisons to analytical solution

Full wavefield vs diffusive approx. for a fullspace

Play around to see that the difference is getting bigger for

- higher frequencies,
- higher eperm/mperm.

```
import empymod
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

Define model

```
x = (np.arange(526))*20-500
rx = np.repeat([x, ], np.size(x), axis=0)
ry = rx.transpose()
zsrc = 150
zrec = 200
res = 1/3
freq = 0.5
ab = 11
aniso = np.sqrt(3/.3)
perm = 1
inp = {
    'src': [0, 0, zsrc],
    'rec': [rx.ravel(), ry.ravel(), zrec],
```

(continues on next page)

(continued from previous page)

```
    'res': res,
    'freqtime': freq,
    'aniso': aniso,
    'ab': ab,
    'epermH': perm,
    'mpermH': perm,
    'verb': 0
}
```

Computation

```
# Halfspace
hs = empymod.analytical(**inp, solution='dfs')
hs = hs.reshape(np.shape(rx))

# Fullspace
fs = empymod.analytical(**inp)
fs = fs.reshape(np.shape(rx))

# Relative error (%)
amperr = np.abs((fs.amp() - hs.amp())/fs.amp())*100
phaerr = np.abs((fs.pha(unwrap=False) - hs.pha(unwrap=False)) /
                fs.pha(unwrap=False))*100
```

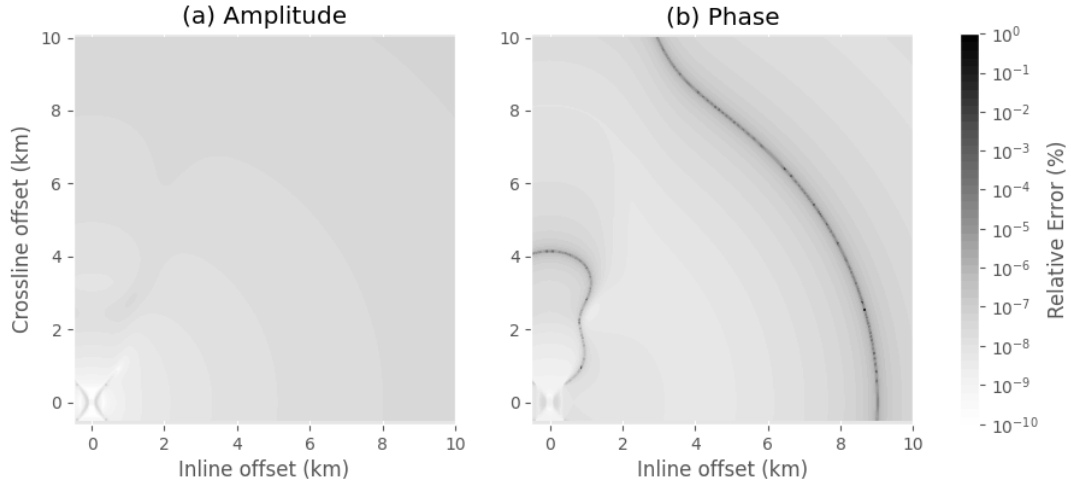
Plot

```
fig, axs = plt.subplots(figsize=(10, 4.2), nrows=1, ncols=2)

# Min and max, properties
vmin = 1e-10
vmax = 1e0
props = {'levels': np.logspace(np.log10(vmin), np.log10(vmax), 50),
         'locator': plt.matplotlib.ticker.LogLocator(), 'cmap': 'Greys'}

# Plot amplitude error
plt.sca(axs[0])
plt.title(r'(a) Amplitude')
cf1 = plt.contourf(rx/1000, ry/1000, amperr.clip(vmin, vmax), **props)
plt.ylabel('Crossline offset (km)')
plt.xlabel('Inline offset (km)')
plt.xlim(min(x)/1000, max(x)/1000)
plt.ylim(min(x)/1000, max(x)/1000)
plt.axis('equal')

# Plot phase error
plt.sca(axs[1])
plt.title(r'(b) Phase')
cf2 = plt.contourf(rx/1000, ry/1000, phaerr.clip(vmin, vmax), **props)
plt.xlabel('Inline offset (km)')
plt.xlim(min(x)/1000, max(x)/1000)
plt.ylim(min(x)/1000, max(x)/1000)
plt.axis('equal')

# Title
plt.suptitle('Analytical fullspace solution\nDifference between full ' +
             'wavefield and diffusive approximation.', y=1.1)
```

(continues on next page)

(continued from previous page)

```
# Plot colorbar
cax, kw = plt.matplotlib.colorbar.make_axes(
        [axs[0], axs[1]], location='right', fraction=.05, pad=0.05, aspect=20)
cb = plt.colorbar(cf2, cax=cax, ticks=10**(-(np.arange(13.)[::-1])+2), **kw)
cb.set_label(r'Relative Error $(\%)$')

# Show
plt.show()
```




```
empymod.Report()
```

Total running time of the script: (0 minutes 3.798 seconds)

Estimated memory usage: 186 MB

Comparison of half-space solutions

Comparing of the functions `analytical` with `dipole` for a half-space and a fullspace-solution, where `dipole` internally uses `kernel.fullspace` for the fullspace solution (`xdirect=True`), and `analytical` uses internally `kernel.halfspace`. Both in the frequency and in the time domain.

```
import empymod
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

Time Domain

Define models

```
src = [0, 0, 100]
rec = [2000, 500, 200]
res = [2e14, 2.5]
aniso = [1, 2]
time = np.logspace(-2, 3, 301)

# Collect parameters
inpEM = {'src': src, 'rec': rec, 'freqtime': time, 'verb': 0}
inpEMdip = inpEM.copy()
```

(continues on next page)

(continued from previous page)

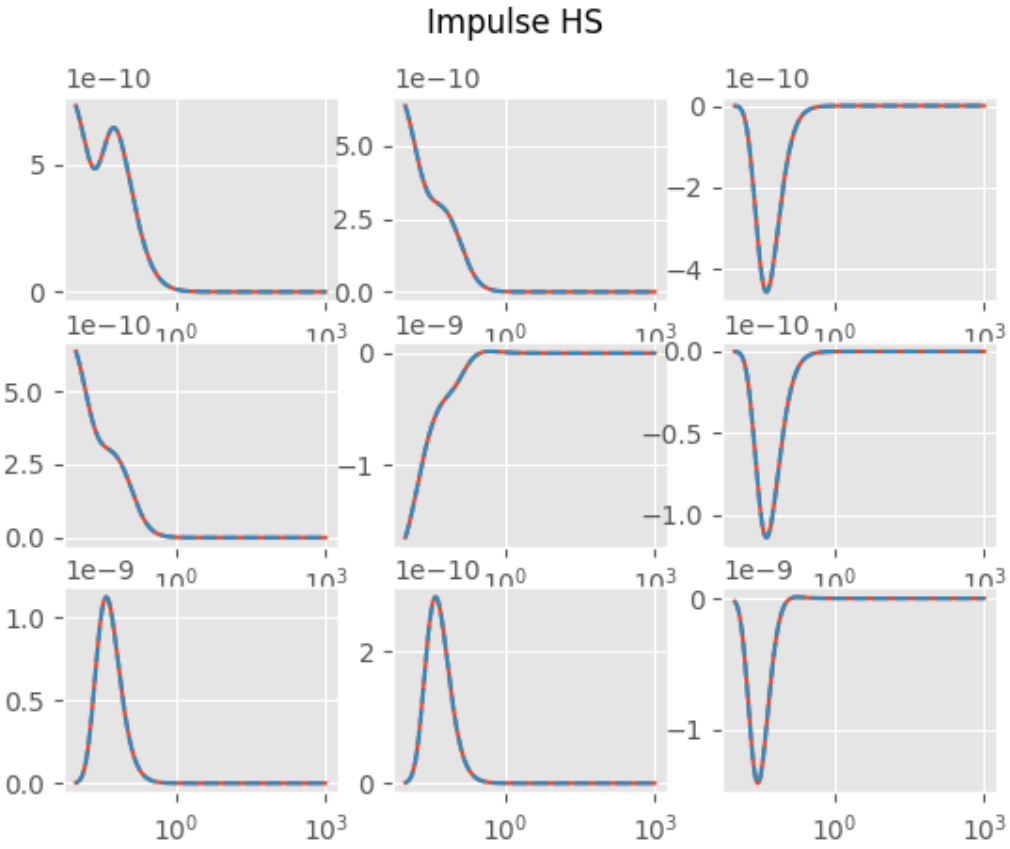
```
inpEMdip['htarg'] = {'pts_per_dec': -1}
modHS = {'res': res, 'aniso': aniso}
modFS = {'res': res[1], 'aniso': aniso[1]}

all_abs = [11, 12, 13, 21, 22, 23, 31, 32, 33]
```

Plot result

```
def plot_t(EM, HS, title, i):
    plt.figure(title, figsize=(10, 8))
    plt.subplot(i)
    plt.semilogx(time, EM)
    plt.semilogx(time, HS, '--')
```

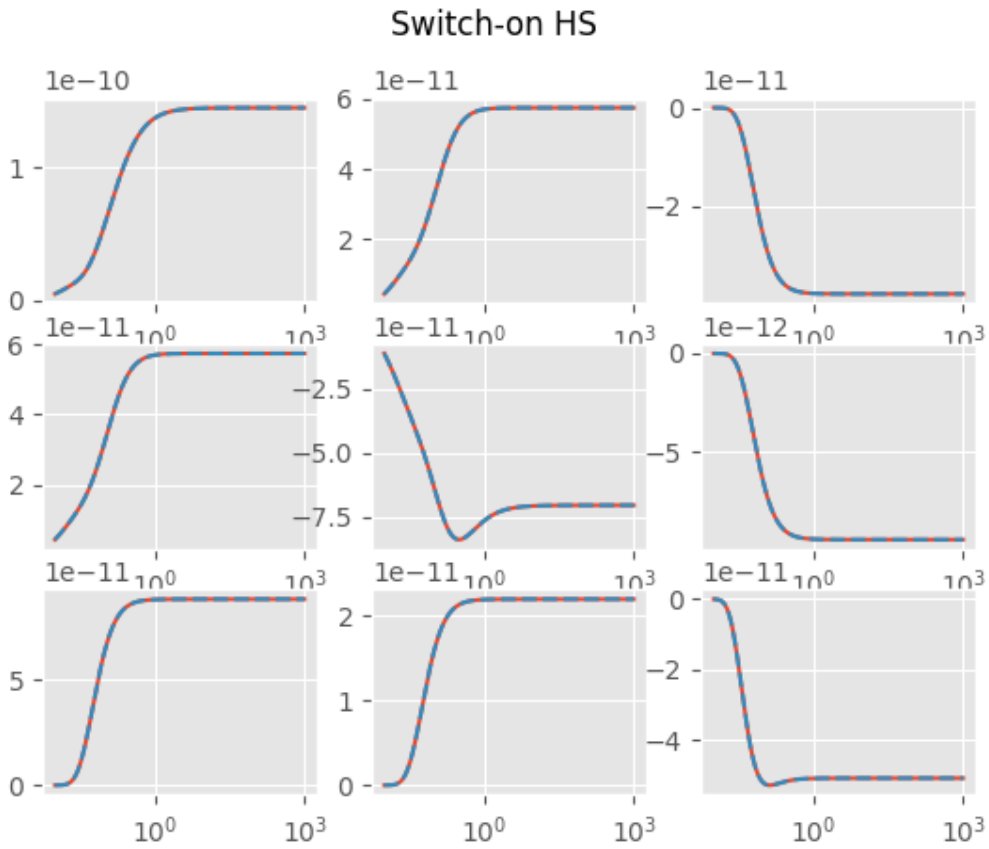
Impulse HS

```
plt.figure('Impulse HS')
i = 330
for ab in all_abs:
    i += 1
    EM = empymod.dipole(**inpEMdip, **modHS, ab=ab, signal=0, depth=0)
    HS = empymod.analytical(**inpEM, **modFS, solution='dhs', ab=ab, signal=0)
    plot_t(EM, HS, 'Impulse HS', i)
plt.suptitle('Impulse HS')
plt.show()
```



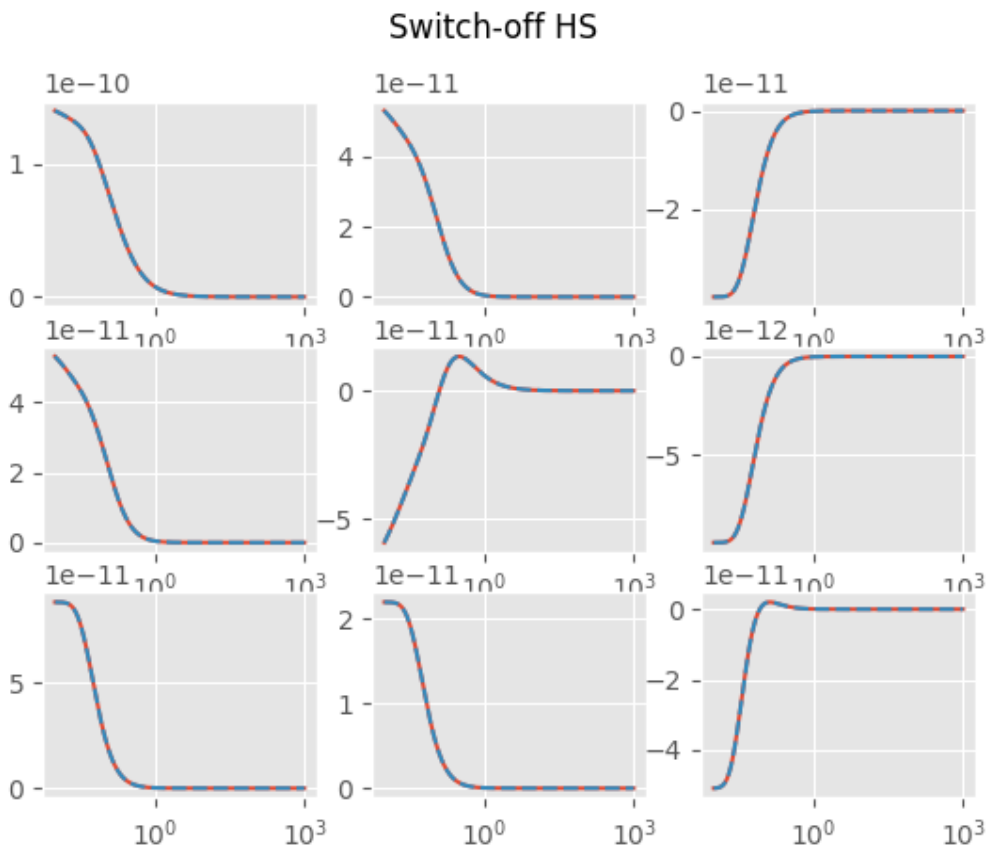
Switch-on HS

```
plt.figure('Switch-on HS')
i = 330
for ab in all_abs:
    i += 1
    EM = empymod.dipole(**inpEMdip, **modHS, ab=ab, signal=1, depth=0)
    HS = empymod.analytical(**inpEM, **modFS, solution='dhs', ab=ab, signal=1)
    plot_t(EM, HS, 'Switch-on HS', i)
plt.suptitle('Switch-on HS')
plt.show()
```



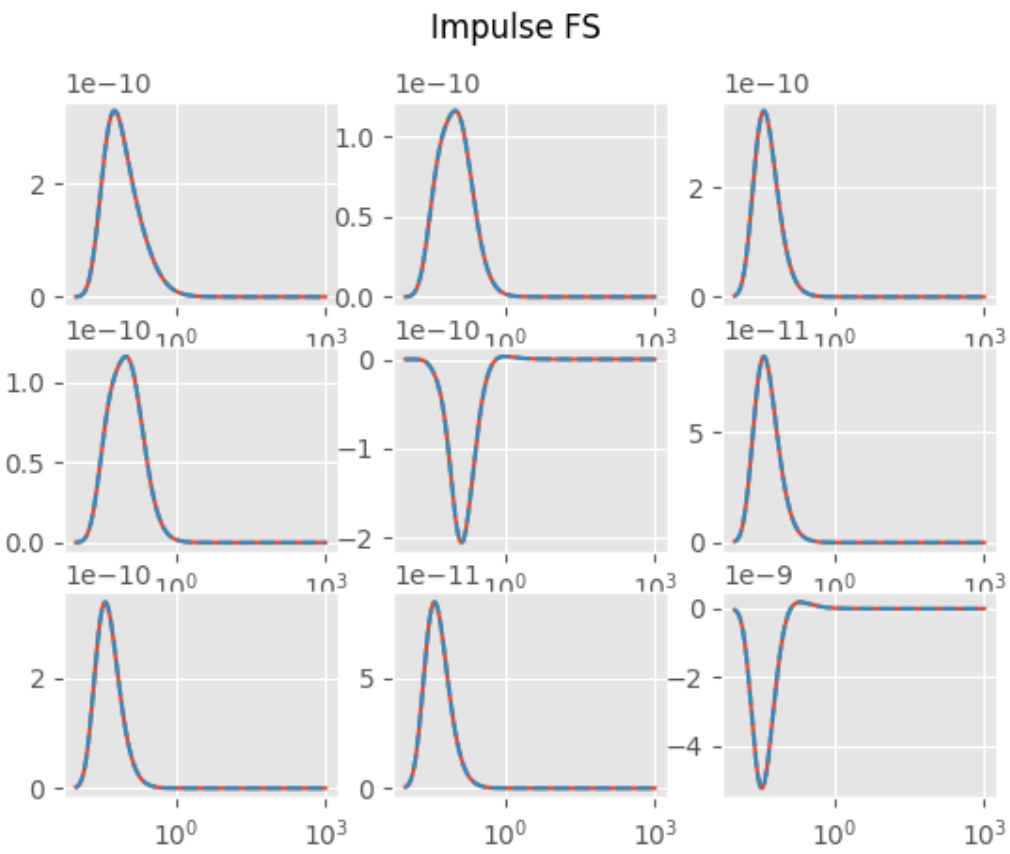
Switch-off HS

```
plt.figure('Switch-off HS')
i = 330
for ab in all_abs:
    i += 1
    EM = empymod.dipole(**inpEMdip, **modHS, ab=ab, signal=-1, depth=0)
    HS = empymod.analytical(**inpEM, **modFS, solution='dhs', ab=ab, signal=-1)
    plot_t(EM, HS, 'Switch-off HS', i)
plt.suptitle('Switch-off HS')
plt.show()
```



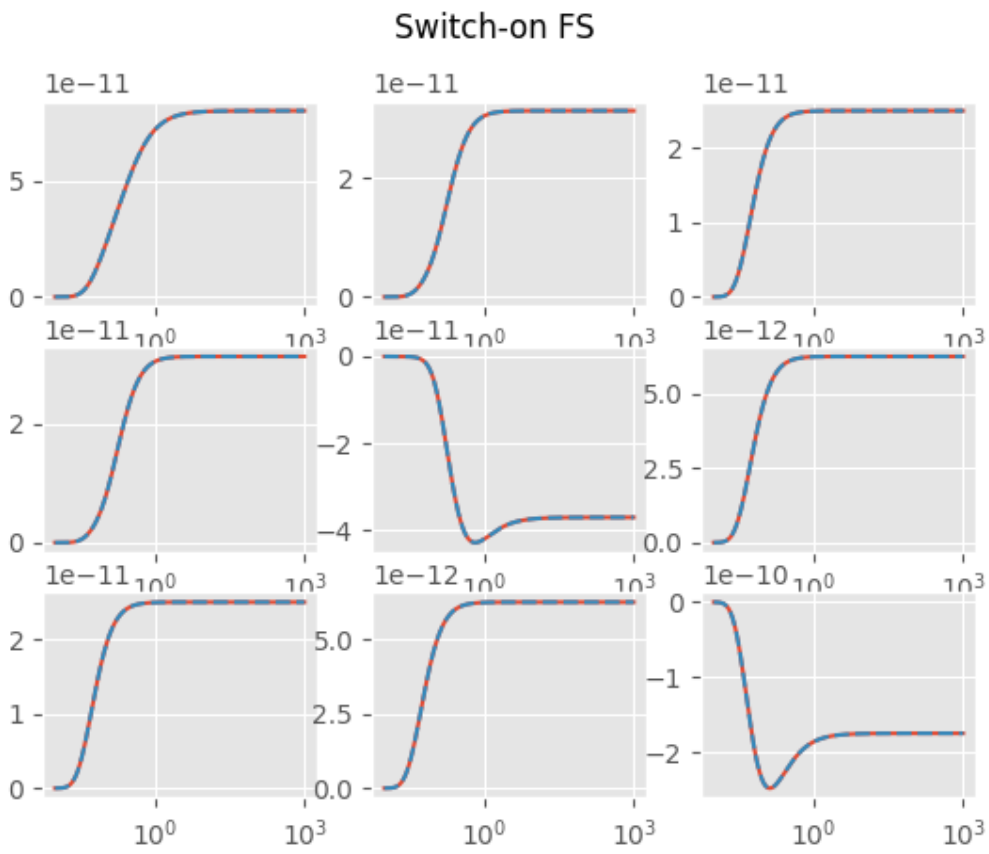
Impulse FS

```
plt.figure('Impulse FS')
i = 330
for ab in all_abs:
    i += 1
    EM = empymod.dipole(**inpEMdip, **modFS, ab=ab, signal=0, depth=[])
    HS = empymod.analytical(**inpEM, **modFS, solution='dfs', ab=ab, signal=0)
    plot_t(EM, HS, 'Impulse FS', i)
plt.suptitle('Impulse FS')
plt.show()
```



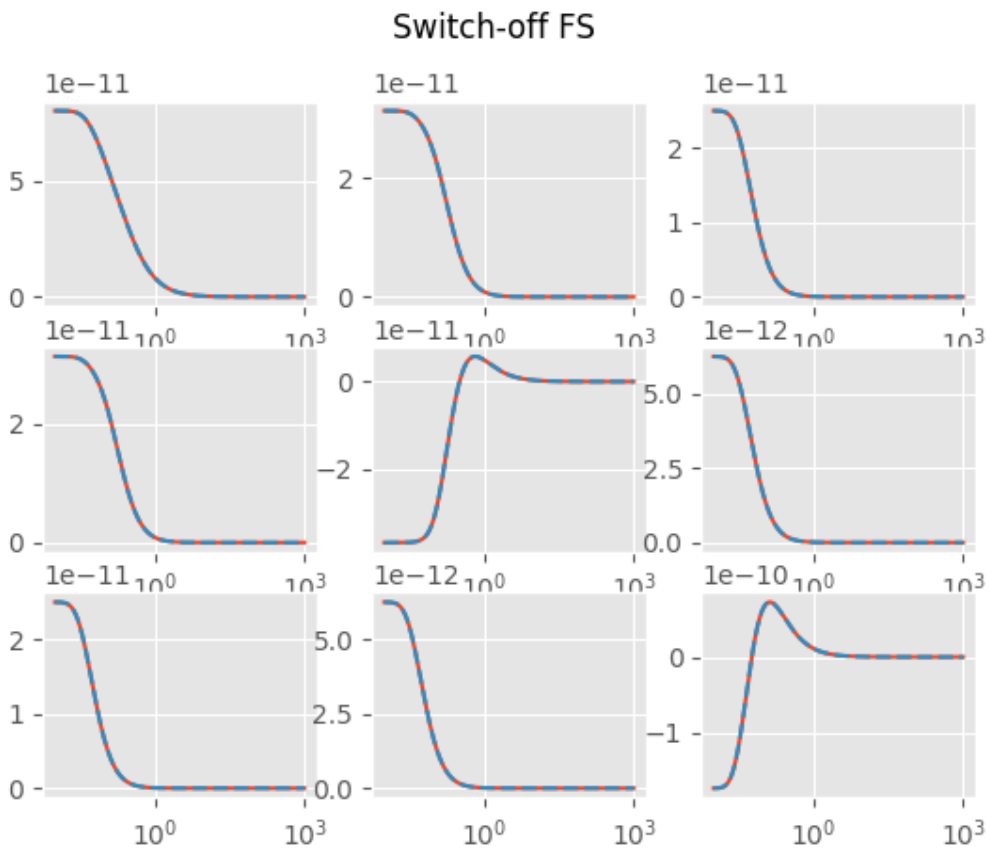
Switch-on FS

```
plt.figure('Switch-on FS')
i = 330
for ab in all_abs:
    i += 1
    EM = empymod.dipole(**inpEMdip, **modFS, ab=ab, signal=1, depth=[])
    HS = empymod.analytical(**inpEM, **modFS, solution='dfs', ab=ab, signal=1)
    plot_t(EM, HS, 'Switch-on FS', i)
plt.suptitle('Switch-on FS')
plt.show()
```



Switch-off FS

```
plt.figure('Switch-off FS')
i = 330
for ab in all_abs:
    i += 1

    # Switch-off
    EM = empymod.dipole(**inpEMdip, **modFS, ab=ab, signal=-1, depth=[])
    HS = empymod.analytical(**inpEM, **modFS, solution='dfs', ab=ab, signal=-1)
    plot_t(EM, HS, 'Switch-off FS', i)
plt.suptitle('Switch-off FS')
plt.show()
```

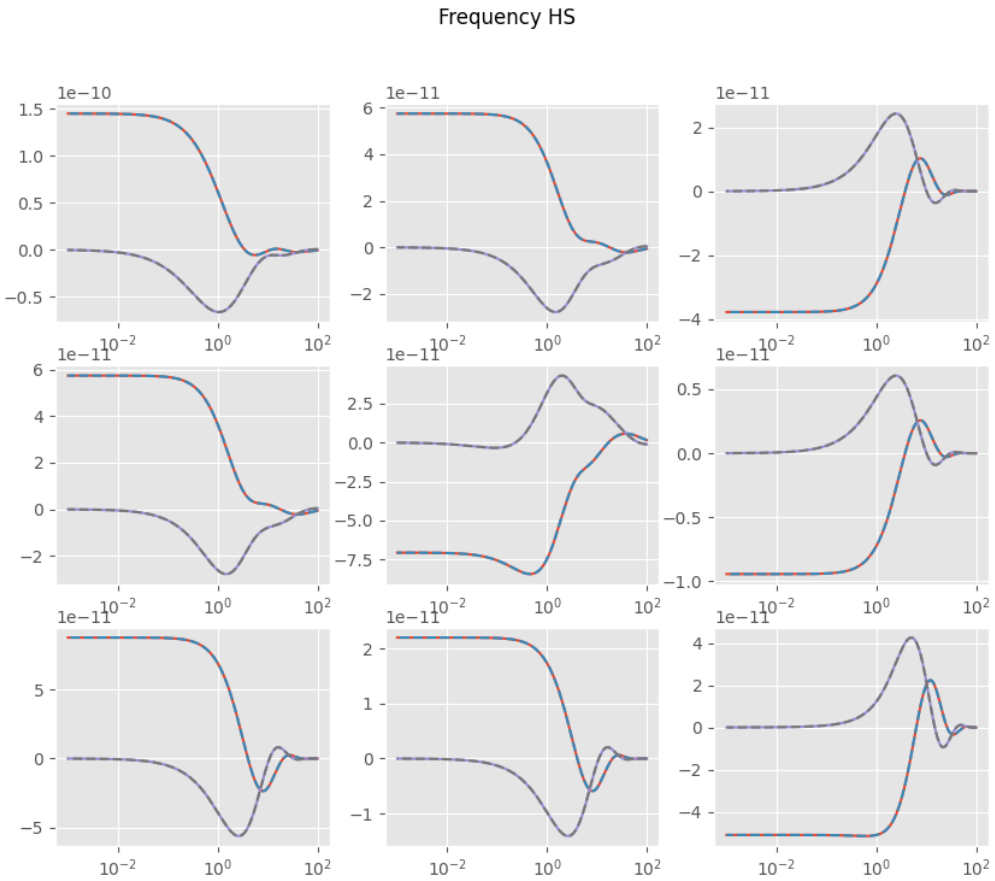


Frequency domain

```
inpEM['freqtime'] = 1/time
inpEMdip['freqtime'] = 1/time


def plot_f(EM, HS, title, i):
    plt.figure(title, figsize=(10, 8))
    plt.subplot(i)
    plt.semilogx(1/time, EM.real)
    plt.semilogx(1/time, HS.real, '--')
    plt.semilogx(1/time, EM.imag)
    plt.semilogx(1/time, HS.imag, '--')
```

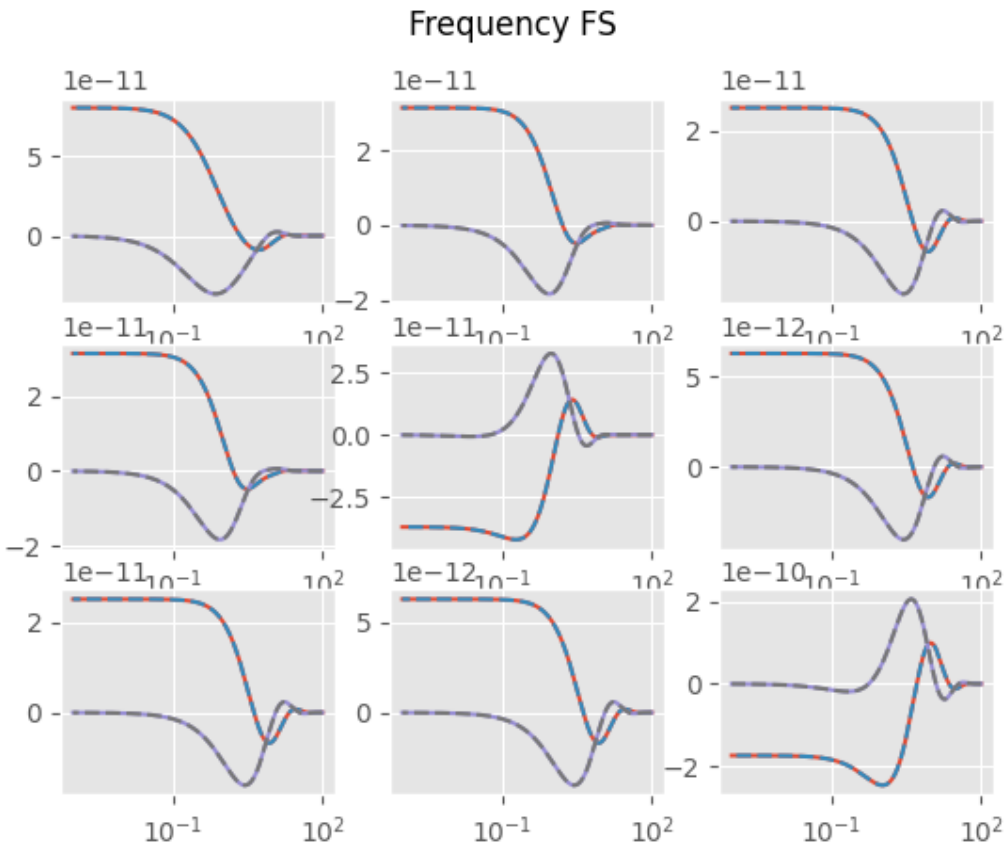
Halfspace

```
i = 330
for ab in all_abs:
    i += 1
    EM = empymod.dipole(**inpEMdip, **modHS, ab=ab, depth=0)
    HS = empymod.analytical(**inpEM, **modFS, solution='dhs', ab=ab)
    plot_f(EM, HS, 'Frequency HS', i)
plt.figure('Frequency HS')
plt.suptitle('Frequency HS')
plt.show()
```



Fullspace

```
plt.figure('Frequency FS')
i = 330
for ab in all_abs:
    i += 1
    EM = empymod.dipole(**inpEMdip, **modFS, ab=ab, depth=[])
    HS = empymod.analytical(**inpEM, **modFS, solution='dfs', ab=ab)
    plot_f(EM, HS, 'Frequency FS', i)
plt.suptitle('Frequency FS')
plt.show()
```




```
empymod.Report()
```

Total running time of the script: (0 minutes 17.331 seconds)

Estimated memory usage: 8 MB

5.4.5 Add-ons

Calculating a Digital Linear Filter.

This is an example for the add-on `fdesign`. The example is taken from the article Werthmüller et al., 2019. Have a look at the article repository on empymod/article-fdesign for many more examples.

Reference

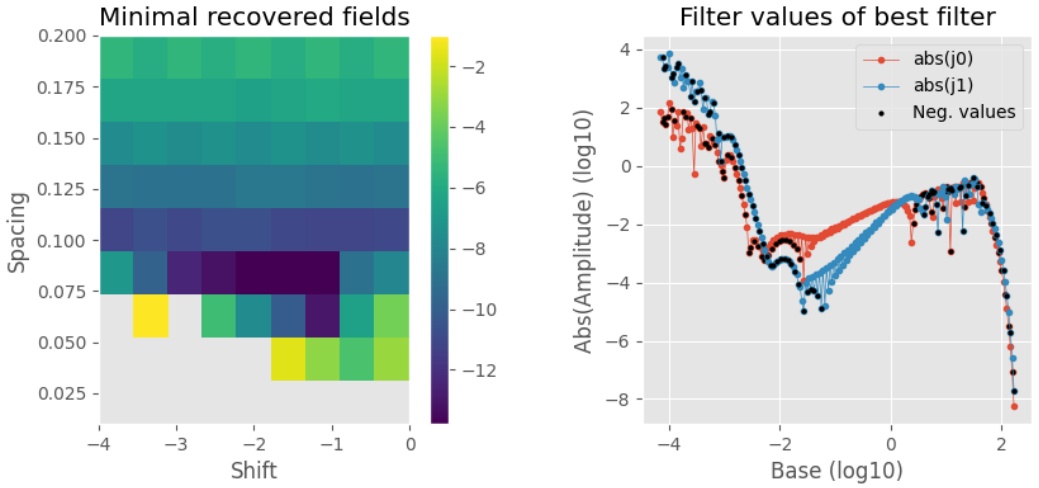
- Werthmüller, D., K. Key, and E. Slob, 2019, **A tool for designing digital filters for the Hankel and Fourier transforms in potential, diffusive, and wavefield modeling**: Geophysics, 84(2), F47-F56; DOI: 10.1190/geo2018-0069.1.

```
import empymod
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

```
inp = {'r': np.logspace(0, 10, 1000),
       'r_def': (1, 1, 2),
       'n': 201,
       'name': 'test',
       'full_output': True,
       'fI': (empymod.fdesign.j0_1(5), empymod.fdesign.j1_1(5))}
```

1. Rough overview over a wide range

```
filt1, out1 = empymod.fdesign.design(
        spacing=(0.01, 0.2, 10), shift=(-4, 0, 10), save=False, **inp)
```



Out:

```
   brute fct calls : 1/100
   brute fct calls : 2/100
   brute fct calls : 3/100
   brute fct calls : 4/100
   brute fct calls : 5/100
   brute fct calls : 6/100
   brute fct calls : 7/100
   brute fct calls : 8/100
   brute fct calls : 9/100
   brute fct calls : 10/100
   brute fct calls : 11/100
   brute fct calls : 12/100
   brute fct calls : 13/100
   brute fct calls : 14/100
   brute fct calls : 15/100
   brute fct calls : 16/100
   brute fct calls : 17/100
   brute fct calls : 18/100
   brute fct calls : 19/100
   brute fct calls : 20/100
   brute fct calls : 21/100
   brute fct calls : 22/100
   brute fct calls : 23/100
   brute fct calls : 24/100
   brute fct calls : 25/100
   brute fct calls : 26/100
   brute fct calls : 27/100
   brute fct calls : 28/100
   brute fct calls : 30/100
   brute fct calls : 31/100
   brute fct calls : 32/100
   brute fct calls : 33/100
   brute fct calls : 34/100
   brute fct calls : 35/100
   brute fct calls : 36/100
```

(continues on next page)

(continued from previous page)

```
   brute fct calls : 37/100
   brute fct calls : 38/100
   brute fct calls : 39/100
   brute fct calls : 40/100
   brute fct calls : 41/100
   brute fct calls : 42/100
   brute fct calls : 43/100
   brute fct calls : 44/100
   brute fct calls : 45/100
   brute fct calls : 46/100
   brute fct calls : 47/100
   brute fct calls : 48/100
   brute fct calls : 49/100
   brute fct calls : 50/100
   brute fct calls : 51/100
   brute fct calls : 52/100
   brute fct calls : 53/100
   brute fct calls : 54/100
   brute fct calls : 55/100
   brute fct calls : 56/100
   brute fct calls : 58/100
   brute fct calls : 59/100
   brute fct calls : 60/100
   brute fct calls : 61/100
   brute fct calls : 62/100
   brute fct calls : 63/100
   brute fct calls : 64/100
   brute fct calls : 65/100
   brute fct calls : 66/100
   brute fct calls : 67/100
   brute fct calls : 68/100
   brute fct calls : 69/100
   brute fct calls : 70/100
   brute fct calls : 71/100
   brute fct calls : 72/100
   brute fct calls : 73/100
   brute fct calls : 74/100
   brute fct calls : 75/100
   brute fct calls : 76/100
   brute fct calls : 77/100
   brute fct calls : 78/100
   brute fct calls : 79/100
   brute fct calls : 80/100
   brute fct calls : 81/100
   brute fct calls : 82/100
   brute fct calls : 83/100
   brute fct calls : 84/100
   brute fct calls : 85/100
   brute fct calls : 86/100
   brute fct calls : 87/100
   brute fct calls : 88/100
   brute fct calls : 89/100
   brute fct calls : 90/100
   brute fct calls : 91/100
   brute fct calls : 92/100
   brute fct calls : 93/100
   brute fct calls : 94/100
   brute fct calls : 95/100
   brute fct calls : 96/100
   brute fct calls : 97/100
   brute fct calls : 98/100
```

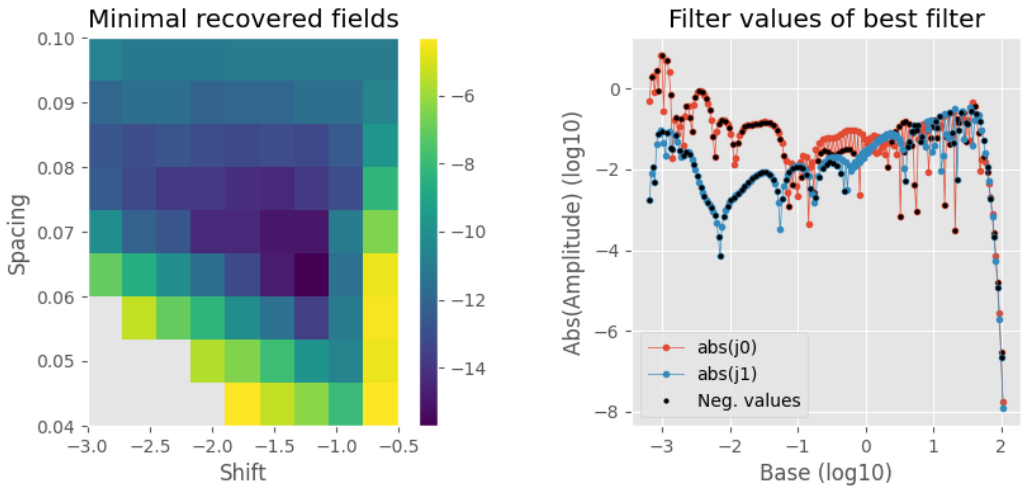
(continues on next page)

(continued from previous page)

```
   brute fct calls : 99/100
   Filter length   : 201
   Best filter
   > Min field     : 1.67412e-14
   > Spacing       : 0.07333333333
   > Shift         : -2.222222222
   > Base min/max  : 7.080680e-05 / 1.658545e+02

:: empymod END; runtime = 0:00:03.220996 ::

* QC: Overview of brute-force inversion:
```

2. First focus

```
filt2, out2 = empymod.fdesign.design(
        spacing=(0.04, 0.1, 10), shift=(-3, -0.5, 10), save=False, **inp)
```



Out:

```
   brute fct calls : 1/100
   brute fct calls : 2/100
   brute fct calls : 3/100
   brute fct calls : 4/100
   brute fct calls : 5/100
   brute fct calls : 6/100
   brute fct calls : 7/100
   brute fct calls : 8/100
   brute fct calls : 9/100
   brute fct calls : 10/100
   brute fct calls : 11/100
   brute fct calls : 12/100
   brute fct calls : 13/100
   brute fct calls : 14/100
   brute fct calls : 15/100
   brute fct calls : 16/100
   brute fct calls : 17/100
   brute fct calls : 18/100
   brute fct calls : 19/100
   brute fct calls : 20/100
```

(continues on next page)

(continued from previous page)

```
  brute fct calls : 21/100
  brute fct calls : 22/100
  brute fct calls : 23/100
  brute fct calls : 24/100
  brute fct calls : 25/100
  brute fct calls : 26/100
  brute fct calls : 27/100
  brute fct calls : 28/100
  brute fct calls : 30/100
  brute fct calls : 31/100
  brute fct calls : 32/100
  brute fct calls : 33/100
  brute fct calls : 34/100
  brute fct calls : 35/100
  brute fct calls : 36/100
  brute fct calls : 37/100
  brute fct calls : 38/100
  brute fct calls : 39/100
  brute fct calls : 40/100
  brute fct calls : 41/100
  brute fct calls : 42/100
  brute fct calls : 43/100
  brute fct calls : 44/100
  brute fct calls : 45/100
  brute fct calls : 46/100
  brute fct calls : 47/100
  brute fct calls : 48/100
  brute fct calls : 49/100
  brute fct calls : 50/100
  brute fct calls : 51/100
  brute fct calls : 52/100
  brute fct calls : 53/100
  brute fct calls : 54/100
  brute fct calls : 55/100
  brute fct calls : 56/100
  brute fct calls : 58/100
  brute fct calls : 59/100
  brute fct calls : 60/100
  brute fct calls : 61/100
  brute fct calls : 62/100
  brute fct calls : 63/100
  brute fct calls : 64/100
  brute fct calls : 65/100
  brute fct calls : 66/100
  brute fct calls : 67/100
  brute fct calls : 68/100
  brute fct calls : 69/100
  brute fct calls : 70/100
  brute fct calls : 71/100
  brute fct calls : 72/100
  brute fct calls : 73/100
  brute fct calls : 74/100
  brute fct calls : 75/100
  brute fct calls : 76/100
  brute fct calls : 77/100
  brute fct calls : 78/100
  brute fct calls : 79/100
  brute fct calls : 80/100
  brute fct calls : 81/100
  brute fct calls : 82/100
  brute fct calls : 83/100
```

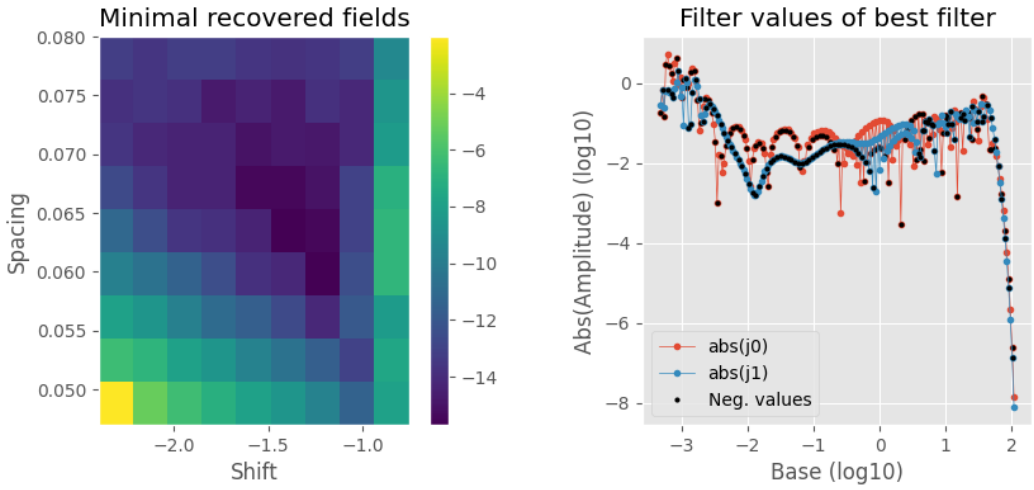
(continues on next page)

(continued from previous page)

```
   brute fct calls : 84/100
   brute fct calls : 85/100
   brute fct calls : 86/100
   brute fct calls : 87/100
   brute fct calls : 88/100
   brute fct calls : 89/100
   brute fct calls : 90/100
   brute fct calls : 91/100
   brute fct calls : 92/100
   brute fct calls : 93/100
   brute fct calls : 94/100
   brute fct calls : 95/100
   brute fct calls : 96/100
   brute fct calls : 97/100
   brute fct calls : 98/100
   brute fct calls : 99/100
   Filter length   : 201
   Best filter
   > Min field     : 1.99577e-16
   > Spacing       : 0.06
   > Shift         : -1.333333333
   > Base min/max  : 6.533920e-04 / 1.063427e+02

:: empymod END; runtime = 0:00:03.335575 ::

* QC: Overview of brute-force inversion:
```

3. Final focus

```
filt, out = empymod.fdesign.design(
        spacing=(0.047, 0.08, 10), shift=(-2.4, -0.75, 10), finish=False,
        save=False, **inp)
```



Out:

```
   brute fct calls : 1/100
   brute fct calls : 2/100
   brute fct calls : 3/100
   brute fct calls : 4/100
```

(continues on next page)

(continued from previous page)

```
  brute fct calls : 5/100
  brute fct calls : 6/100
  brute fct calls : 7/100
  brute fct calls : 8/100
  brute fct calls : 9/100
  brute fct calls : 10/100
  brute fct calls : 11/100
  brute fct calls : 12/100
  brute fct calls : 13/100
  brute fct calls : 14/100
  brute fct calls : 15/100
  brute fct calls : 16/100
  brute fct calls : 17/100
  brute fct calls : 18/100
  brute fct calls : 19/100
  brute fct calls : 20/100
  brute fct calls : 21/100
  brute fct calls : 22/100
  brute fct calls : 23/100
  brute fct calls : 24/100
  brute fct calls : 25/100
  brute fct calls : 26/100
  brute fct calls : 27/100
  brute fct calls : 28/100
  brute fct calls : 30/100
  brute fct calls : 31/100
  brute fct calls : 32/100
  brute fct calls : 33/100
  brute fct calls : 34/100
  brute fct calls : 35/100
  brute fct calls : 36/100
  brute fct calls : 37/100
  brute fct calls : 38/100
  brute fct calls : 39/100
  brute fct calls : 40/100
  brute fct calls : 41/100
  brute fct calls : 42/100
  brute fct calls : 43/100
  brute fct calls : 44/100
  brute fct calls : 45/100
  brute fct calls : 46/100
  brute fct calls : 47/100
  brute fct calls : 48/100
  brute fct calls : 49/100
  brute fct calls : 50/100
  brute fct calls : 51/100
  brute fct calls : 52/100
  brute fct calls : 53/100
  brute fct calls : 54/100
  brute fct calls : 55/100
  brute fct calls : 56/100
  brute fct calls : 58/100
  brute fct calls : 59/100
  brute fct calls : 60/100
  brute fct calls : 61/100
  brute fct calls : 62/100
  brute fct calls : 63/100
  brute fct calls : 64/100
  brute fct calls : 65/100
  brute fct calls : 66/100
  brute fct calls : 67/100
```

(continues on next page)

(continued from previous page)

```
   brute fct calls : 68/100
   brute fct calls : 69/100
   brute fct calls : 70/100
   brute fct calls : 71/100
   brute fct calls : 72/100
   brute fct calls : 73/100
   brute fct calls : 74/100
   brute fct calls : 75/100
   brute fct calls : 76/100
   brute fct calls : 77/100
   brute fct calls : 78/100
   brute fct calls : 79/100
   brute fct calls : 80/100
   brute fct calls : 81/100
   brute fct calls : 82/100
   brute fct calls : 83/100
   brute fct calls : 84/100
   brute fct calls : 85/100
   brute fct calls : 86/100
   brute fct calls : 87/100
   brute fct calls : 88/100
   brute fct calls : 89/100
   brute fct calls : 90/100
   brute fct calls : 91/100
   brute fct calls : 92/100
   brute fct calls : 93/100
   brute fct calls : 94/100
   brute fct calls : 95/100
   brute fct calls : 96/100
   brute fct calls : 97/100
   brute fct calls : 98/100
   brute fct calls : 99/100
   Filter length   : 201
   Best filter
   > Min field     : 1.99577e-16
   > Spacing       : 0.06166666667
   > Shift         : -1.483333333
   > Base min/max  : 4.760441e-04 / 1.081299e+02

:: empymod END; runtime = 0:00:03.266564 ::

* QC: Overview of brute-force inversion:
```

To reproduce exactly the same filter as wer_201_2018:

```
filt_orig, out_orig = fdesign.load_filter('wer201', True)
fdesign.plot_result(filt_orig, out_orig)
filt_orig, out_orig = fdesign.design(
        spacing=out_orig[0][0], shift=out_orig[0][1], **inp)
```

Plot the result

Plot function

```
def plotresult(depth, res, zsrc, zrec):
    x = np.arange(1, 101)*200
```

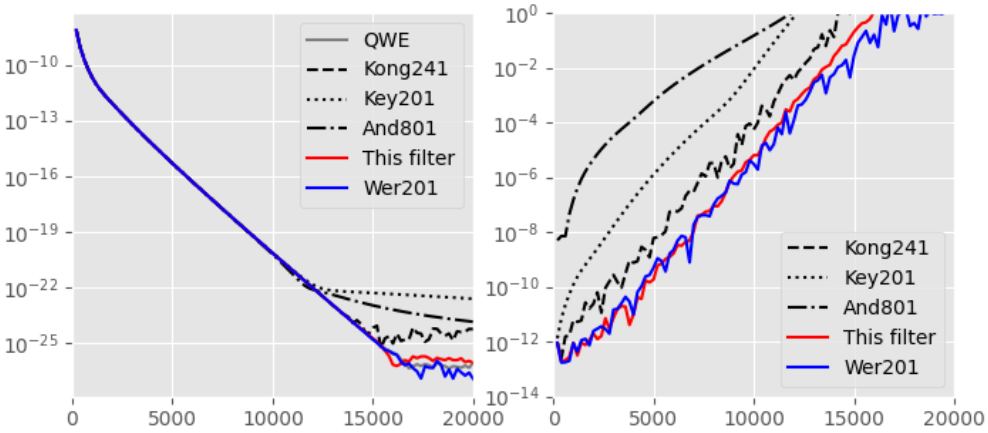
(continues on next page)

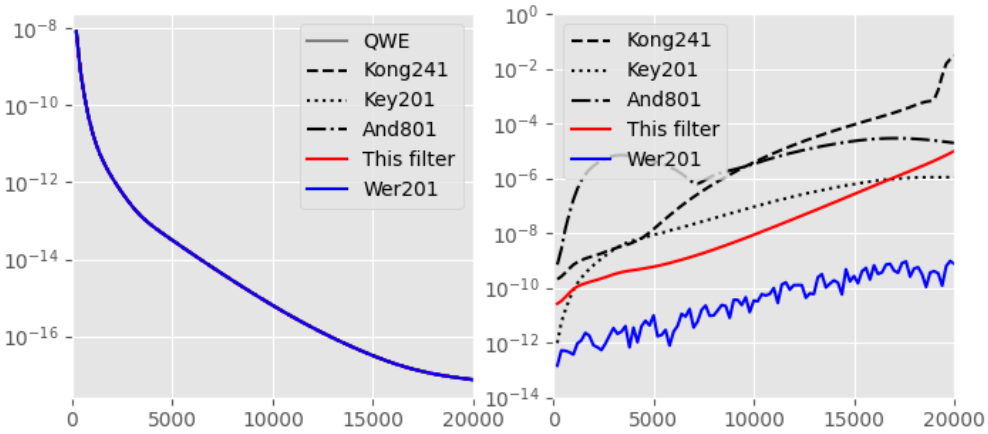
(continued from previous page)

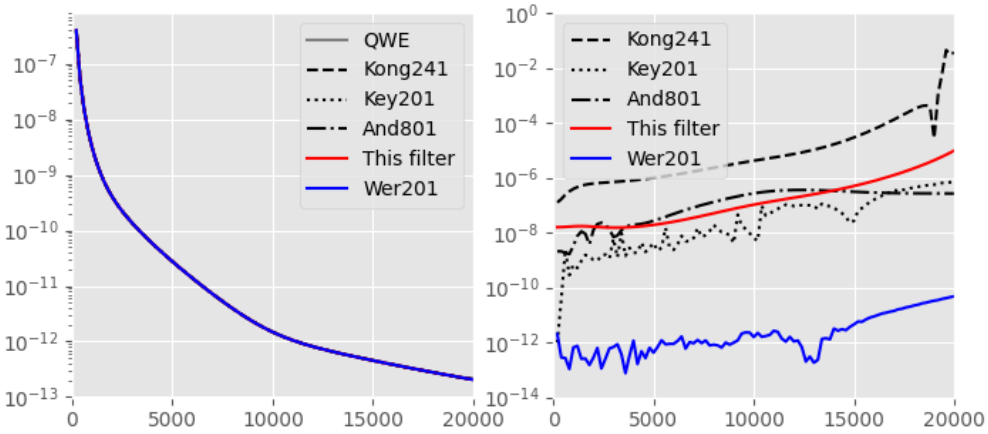
```
    inp = {
        'src': [0, 0, depth[1]-zsrc],
        'rec': [x, x*0, depth[1]-zrec],
        'depth': depth,
        'res': res,
        'ab': 11,
        'freqtime': 1,
        'verb': 1,
    }

    kong241 = empymod.dipole(htarg={'dlf': 'kong_241_2007'}, **inp)
    key201 = empymod.dipole(htarg={'dlf': 'key_201_2012'}, **inp)
    and801 = empymod.dipole(htarg={'dlf': 'anderson_801_1982'}, **inp)
    test = empymod.dipole(htarg={'dlf': filt}, **inp)
    wer201 = empymod.dipole(htarg={'dlf': 'wer_201_2018'}, **inp)
    qwe = empymod.dipole(ht='qwe', **inp)

    plt.figure(figsize=(8, 3.5))
    plt.subplot(121)
    plt.semilogy(x, qwe.amp(), c='0.5', label='QWE')
    plt.semilogy(x, kong241.amp(), 'k--', label='Kong241')
    plt.semilogy(x, key201.amp(), 'k:', label='Key201')
    plt.semilogy(x, and801.amp(), 'k-.', label='And801')
    plt.semilogy(x, test.amp(), 'r', label='This filter')
    plt.semilogy(x, wer201.amp(), 'b', label='Wer201')
    plt.legend()
    plt.xticks([0, 5e3, 10e3, 15e3, 20e3])
    plt.xlim([0, 20e3])

    plt.subplot(122)
    plt.semilogy(x, np.abs((kong241-qwe)/qwe), 'k--', label='Kong241')
    plt.semilogy(x, np.abs((key201-qwe)/qwe), 'k:', label='Key201')
    plt.semilogy(x, np.abs((and801-qwe)/qwe), 'k-.', label='And801')
    plt.semilogy(x, np.abs((test-qwe)/qwe), 'r', label='This filter')
    plt.semilogy(x, np.abs((wer201-qwe)/qwe), 'b', label='Wer201')
    plt.legend()
    plt.xticks([0, 5e3, 10e3, 15e3, 20e3])
    plt.xlim([0, 20e3])
    plt.ylim([1e-14, 1])

    plt.show()
```

Plot the individual models

```
plotresult([-1e50, 2000], [2e14, 1/3.2, 1], 50, 0)  # KONG model
plotresult([0, 1000, 2000, 2100], [1/1e-12, 1/3.3, 1, 100, 1], 10, 0)  # KEY m.
plotresult([0, 1, 1000, 1100], [2e14, 10, 10, 500, 10], 0.5, 0.2)  # LAND model
```

- 

- 

- 

Out:

```
* WARNING :: Hankel-quadrature did not converge at least once;
             => desired `atol` and `rtol` might not be achieved.
* WARNING :: Hankel-quadrature did not converge at least once;
             => desired `atol` and `rtol` might not be achieved.
```

```
empymod.Report()
```

Total running time of the script: (0 minutes 37.284 seconds)

Estimated memory usage: 65 MB

Contributions of up- and downgoing TM- and TE-modes

This is an example for the add-on `tmtemod`. The example is taken from the CSEM-book by Ziolkowski and Slob, 2019. Have a look at the CSEM-book repository on empymod/csem-ziolkowski-and-slob for many more examples.

Reference

- Ziolkowski, A., and E. Slob, 2019, **Introduction to Controlled-Source Electromagnetic Methods**: Cambridge University Press; ISBN 9781107058620.

```
import empymod
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

Model parameters

```
# Offsets
x = np.linspace(10, 1.5e4, 128)

# Resistivity model
rtg = [2e14, 1/3, 1, 70, 1]  # With target
rhs = [2e14, 1/3, 1, 1, 1]   # Half-space

# Common model parameters (deep sea parameters)
model = {
    'src': [0, 0, 975],               # Source location
    'rec': [x, x*0, 1000],            # Receiver location
    'depth': [0, 1000, 2000, 2040],   # 1 km water, target 40 m thick 1 km below
    'freqtime': 0.5,                  # Frequencies
    'verb': 1,                        # Verbosity
}
```

Computation

```
target = empymod.dipole(res=rtg, **model)
tgTM, tgTE = empymod.tmtemod.dipole(res=rtg, **model)

# Without reservoir
notarg = empymod.dipole(res=rhs, **model)
ntTM, ntTE = empymod.tmtemod.dipole(res=rhs, **model)
```

Figure 1

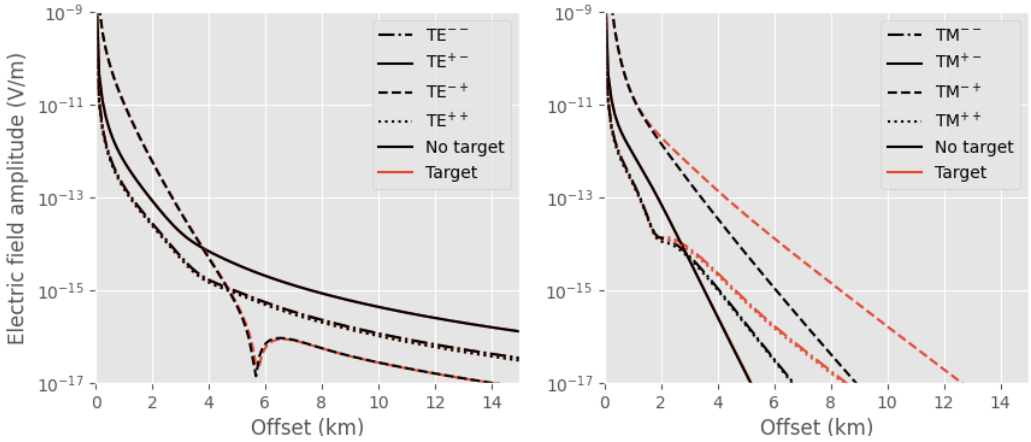
Plot all reflected contributions (without direct field), for the models with and without a reservoir.

```
plt.figure(figsize=(10, 4))

# 1st subplot
ax1 = plt.subplot(121)
plt.semilogy(x/1000, np.abs(tgTE[0]), 'C0-.')
plt.semilogy(x/1000, np.abs(ntTE[0]), 'k-.', label='TE$^{--}$')
plt.semilogy(x/1000, np.abs(tgTE[2]), 'C0-')
```

(continues on next page)

(continued from previous page)

```
plt.semilogy(x/1000, np.abs(ntTE[2]), 'k-', label='TE$^{+-}$')
plt.semilogy(x/1000, np.abs(tgTE[1]), 'C0--')
plt.semilogy(x/1000, np.abs(ntTE[1]), 'k--', label='TE$^{-+}$')
plt.semilogy(x/1000, np.abs(tgTE[3]), 'C0:')
plt.semilogy(x/1000, np.abs(ntTE[3]), 'k:', label='TE$^{++}$')
plt.semilogy(-1, 1, 'k-', label='No target')  # Dummy entries for labels
plt.semilogy(-1, 1, 'C0-', label='Target')     # "
plt.legend()
plt.xlabel('Offset (km)')
plt.ylabel('Electric field amplitude (V/m)')
plt.xlim([0, 15])

# 2nd subplot
plt.subplot(122, sharey=ax1)
plt.semilogy(x/1000, np.abs(tgTM[0]), 'C0-.')
plt.semilogy(x/1000, np.abs(ntTM[0]), 'k-.', label='TM$^{--}$')
plt.semilogy(x/1000, np.abs(tgTM[2]), 'C0-')
plt.semilogy(x/1000, np.abs(ntTM[2]), 'k-', label='TM$^{+-}$')
plt.semilogy(x/1000, np.abs(tgTM[1]), 'C0--')
plt.semilogy(x/1000, np.abs(ntTM[1]), 'k--', label='TM$^{-+}$')
plt.semilogy(x/1000, np.abs(tgTM[3]), 'C0:')
plt.semilogy(x/1000, np.abs(ntTM[3]), 'k:', label='TM$^{++}$')
plt.semilogy(-1, 1, 'k-', label='No target')  # Dummy entries for labels
plt.semilogy(-1, 1, 'C0-', label='Target')     # "
plt.legend()
plt.xlabel('Offset (km)')
plt.ylim([1e-17, 1e-9])
plt.xlim([0, 15])

plt.show()
```



The result shows that mainly the TM-mode contributions are sensitive to the reservoir. For TM, all modes contribute significantly except T^{+-}, which is the field that travels upwards from the source and downwards to the receiver.

Figure 2

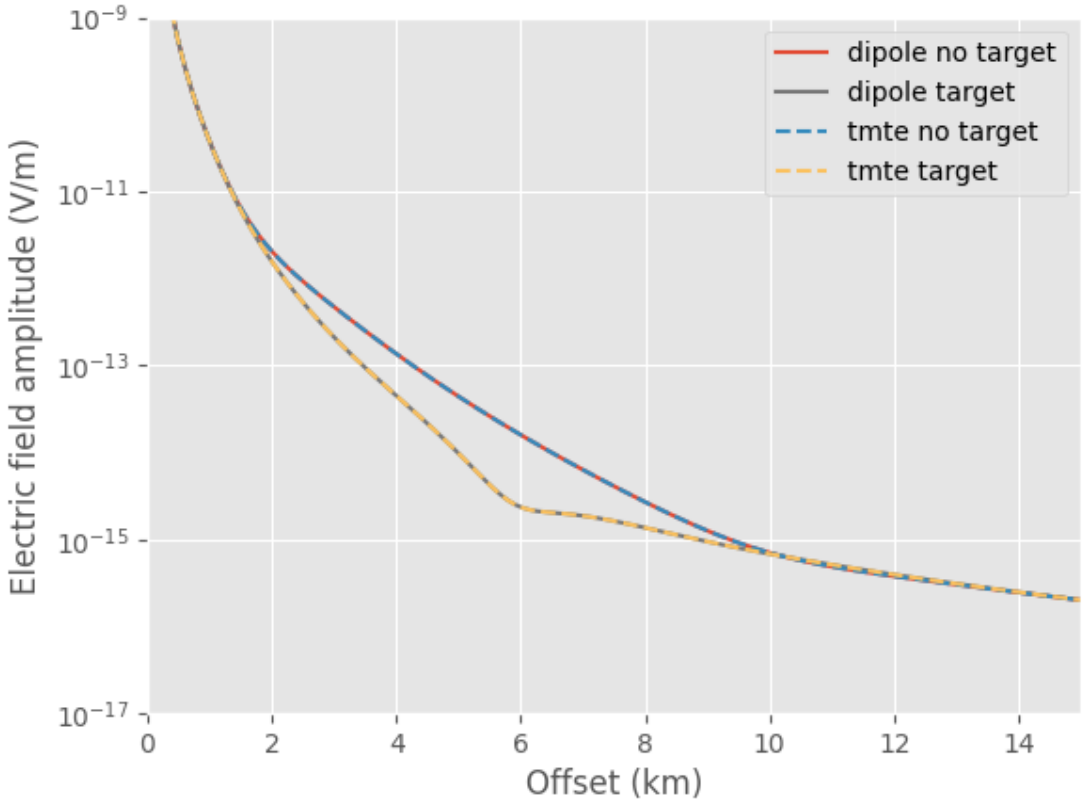
Finally we check if the result from `empymod.dipole` equals the sum of the output of `empymod.tmtemod.dipole`.

```
plt.figure()
```

(continues on next page)

(continued from previous page)

```
nt = ntTM[0]+ntTM[1]+ntTM[2]+ntTM[3]+ntTM[4]
nt += ntTE[0]+ntTE[1]+ntTE[2]+ntTE[3]+ntTE[4]
tg = tgTM[0]+tgTM[1]+tgTM[2]+tgTM[3]+tgTM[4]
tg += tgTE[0]+tgTE[1]+tgTE[2]+tgTE[3]+tgTE[4]

plt.semilogy(x/1000, np.abs(target), 'C0-', label='dipole no target')
plt.semilogy(x/1000, np.abs(notarg), 'C3-', label='dipole target')
plt.semilogy(x/1000, np.abs(tg), 'C1--', label='tmte no target')
plt.semilogy(x/1000, np.abs(nt), 'C4--', label='tmte target')
plt.legend()
plt.xlabel('Offset (km)')
plt.ylabel('Electric field amplitude (V/m)')
plt.ylim([1e-17, 1e-9])
plt.xlim([0, 15])

plt.show()
```




```
empymod.Report()
```

Total running time of the script: (0 minutes 2.074 seconds)

Estimated memory usage: 8 MB

5.4.6 Reproducing published results

CSEM

Reproducing a few published results in the field of Controlled-Source Electromagnetics. So far there are only two examples, Ziolkowski et al., 2007, Figure 3 and Constable and Weiss, 2006, Figure 3. notebooks.

More examples could be implemented. In the references are some papers listed that have interesting 1D modelling results.

References

- **Constable, S., and C.~J. Weiss, 2006**, Mapping thin resistors and hydrocarbons with marine EM methods: Insights from 1d modeling Geophysics, 71, G43-G51; DOI: 10.1190/1.2187748.
- **Constable, S., 2010**, Ten years of marine CSEM for hydrocarbon exploration: Geophysics, 75, 75A67-75A81; DOI: 10.1190/1.3483451.
- **MacGregor, L., and J. Tomlinson, 2014**, Marine controlled-source electromagnetic methods in the hydrocarbon industry: A tutorial on method and practice: Interpretation, 2, SH13-SH32; DOI: 10.1190/INT-2013-0163.1.
- **Ziolkowski, A., B. Hobbs, and D. Wright, 2007**, Multitransient electromagnetic demonstration survey in france Geophysics, 72, F197-F209; DOI: 10.1190/1.2735802.
- **Ziolkowski, A., and D. Wright, 2012**, The potential of the controlled source electromagnetic method: A powerful tool for hydrocarbon exploration, appraisal, and reservoir characterization Signal Processing Magazine, IEEE, 29, 36-52; DOI: 10.1109/MSP.2012.2192529.


```
import empymod
import numpy as np
from copy import deepcopy as dc
import matplotlib.pyplot as plt
```

1. Ziolkowski et al. (2007), Figure 3

A land MTEM example.

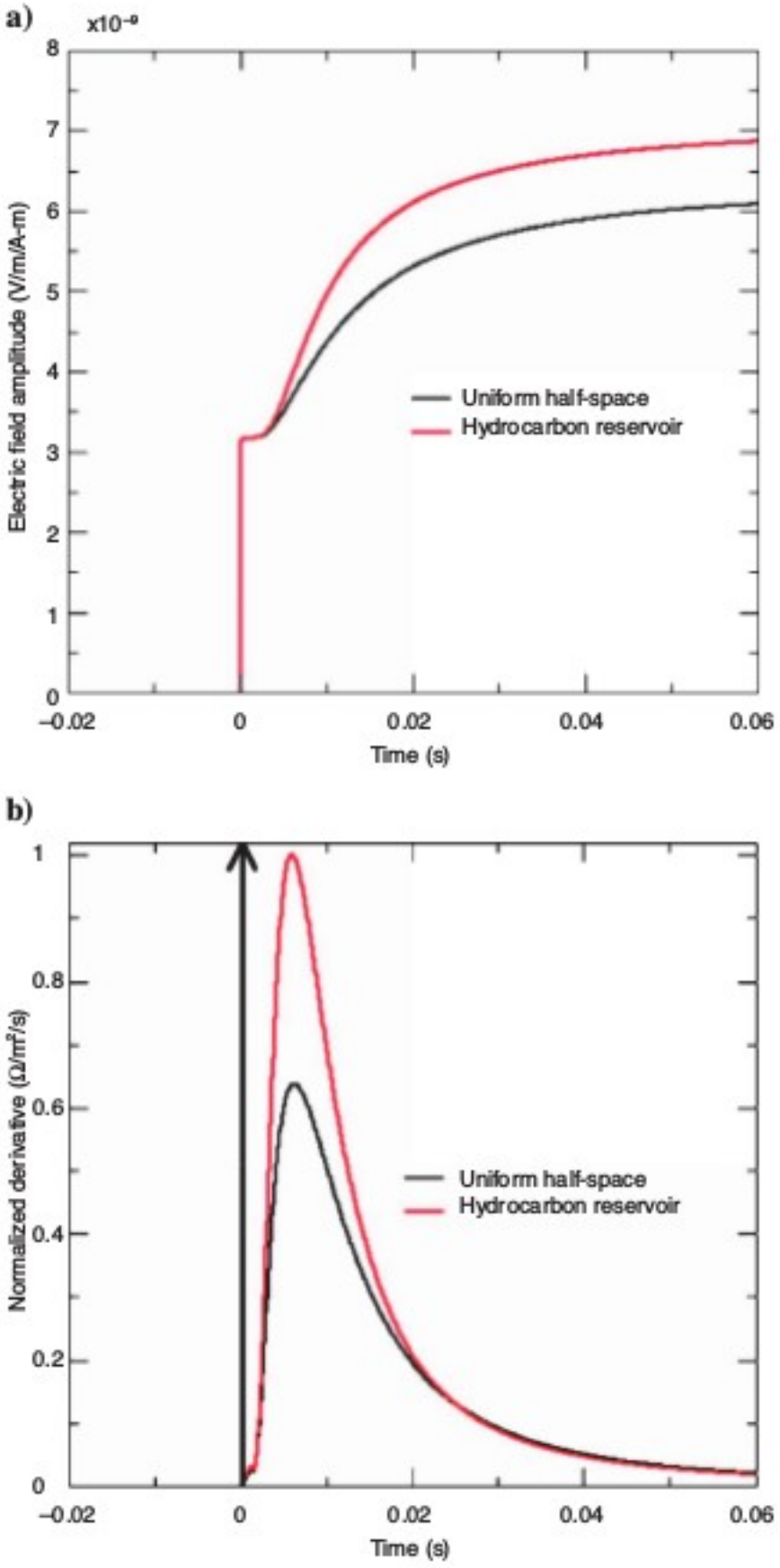


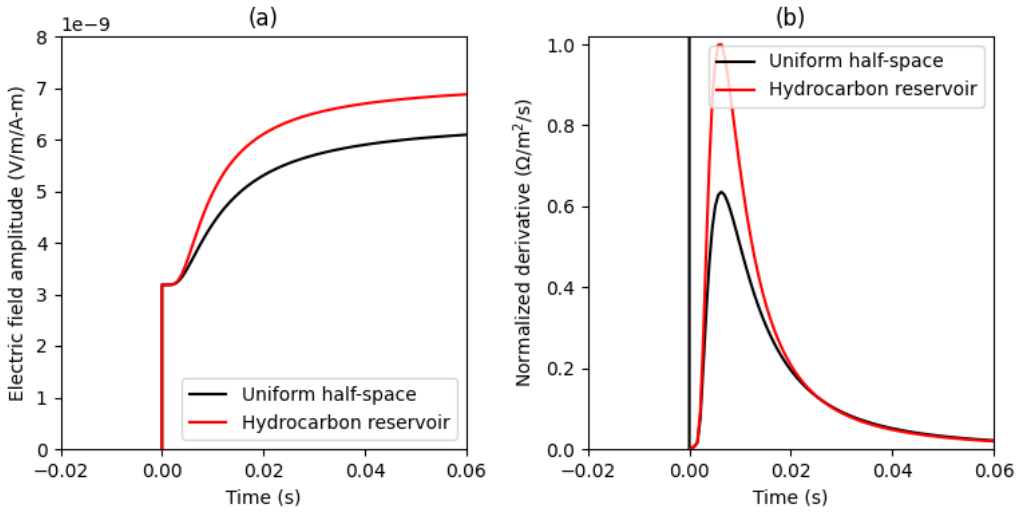
Figure 3. (a) Response of a 20 ohm-m half-space at an offset of 1000 m to a 1-A-m step at the source dipole (black curve) and with a 25-m-thick, 500-ohm-m resistive layer at a depth of 500 m (red curve). (b) Normalized impulse response, with normalization factor 3.433E + 6 for a 20 ohm-m half-space (black curve), with peak at 0.00628 s, and with a 25-m-thick, 500-ohm-m resistive layer at a depth of 500 m (red curve), with peak at 0.00585 s. The source and receiver are 1 km apart. The black vertical arrow at time = 0.0 repre-

Computation

```
# Time
t = np.linspace(0.001, 0.06, 101)

# Target model
inp2 = {'src': [0, 0, 0.001],
        'rec': [1000, 0, 0.001],
        'depth': [0, 500, 525],
        'res': [2e14, 20, 500, 20],
        'freqtime': t,
        'verb': 1}

# HS model
inp1 = dc(inp2)
inp1['depth'] = inp2['depth'][0]
inp1['res'] = inp2['res'][:2]

# Compute responses
sths = empymod.dipole(**inp1, signal=1)  # Step, HS
sttg = empymod.dipole(**inp2, signal=1)  # " "   Target
imhs = empymod.dipole(**inp1, signal=0, ft='fftlog')  # Impulse, HS
imtg = empymod.dipole(**inp2, signal=0, ft='fftlog')  # " "      Target
```

Plot

```
plt.figure(figsize=(9, 4))
plt.subplots_adjust(wspace=.3)

# Step response
plt.subplot(121)
plt.title('(a)')
plt.plot(np.r_[0, 0, t], np.r_[0, sths[0], sths], 'k',
         label='Uniform half-space')
plt.plot(np.r_[0, 0, t], np.r_[0, sttg[0], sttg], 'r',
         label='Hydrocarbon reservoir')
plt.axis([-.02, 0.06, 0, 8e-9])
plt.xlabel('Time (s)')
plt.ylabel('Electric field amplitude (V/m/A-m)')
plt.legend()

# Impulse response
plt.subplot(122)
plt.title('(b)')

# Normalize by max-response
ntg = np.max(np.r_[imtg, imhs])

plt.plot(np.r_[0, 0, t], np.r_[2, 0, imhs/ntg], 'k',
         label='Uniform half-space')
plt.plot(np.r_[0, t], np.r_[0, imtg/ntg], 'r', label='Hydrocarbon reservoir')
plt.axis([-.02, 0.06, 0, 1.02])
plt.xlabel('Time (s)')
plt.ylabel(r'Normalized derivative ($\Omega$/m$^2$/s)')
plt.legend()
plt.show()
```



2. Constable and Weiss (2006), Figure 3

Note: Exact reproduction is not possible, as source and receiver depths are not explicitly specified in the publication. I made a few checks, and it looks like a source-depth of 900 meter gives good accordance. Receivers are on the sea-floor.

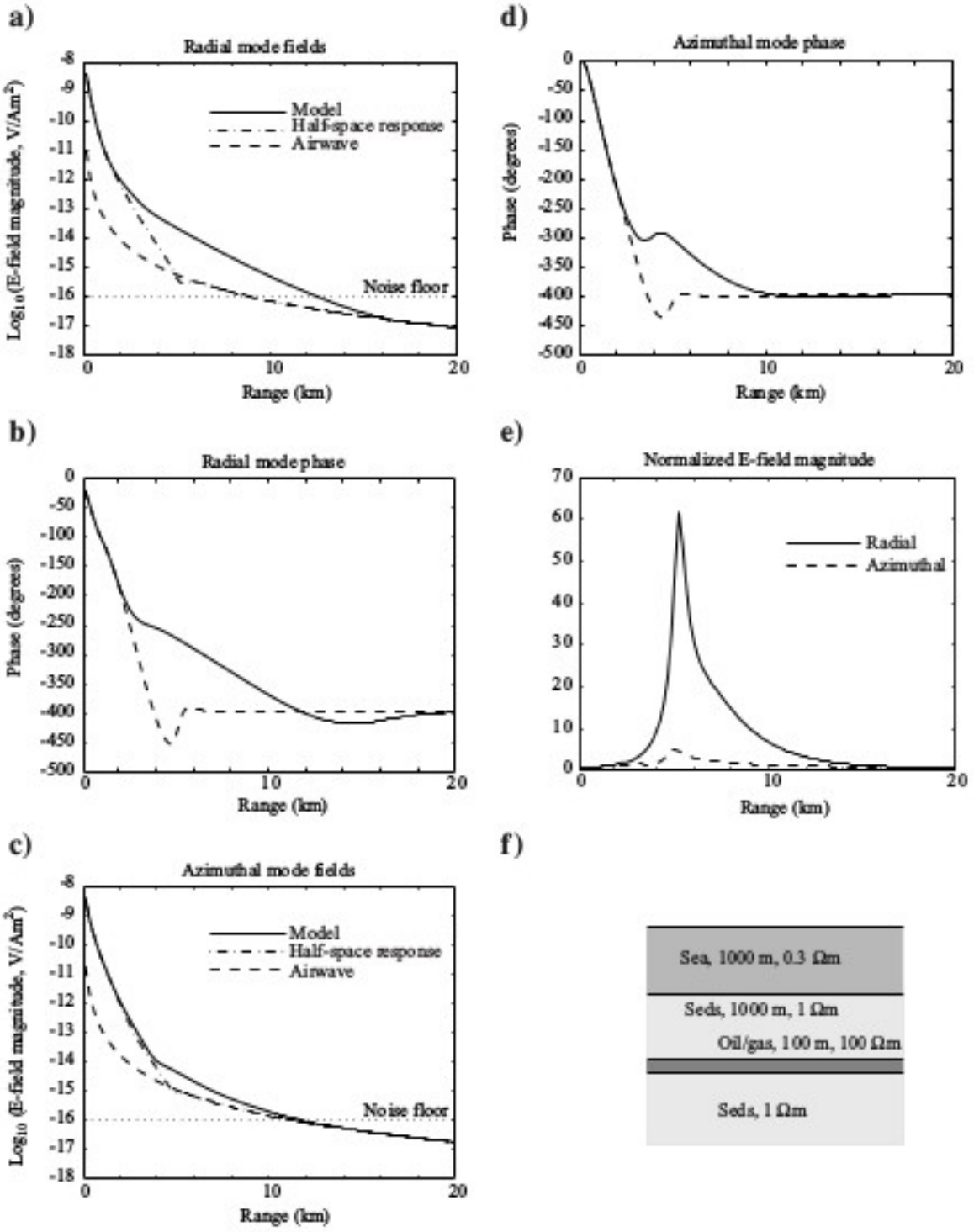


Figure 3. (a, b) Radial and (c, d) azimuthal mode CSEM amplitudes and phases as a function of source–receiver range at a frequency of 1 Hz, calculated for the model shown in (f) with (model) and without (half-space) the thin resistive layer. In (e) we show the radial and azimuthal field amplitudes normalized by the response of the half-space.

Computation

```
# Offsets
x = np.linspace(0, 20000, 101)

# TG model
inp3 = {'src': [0, 0, 900],
        'rec': [x, np.zeros(x.shape), 1000],
        'depth': [0, 1000, 2000, 2100],
        'res': [2e14, 0.3, 1, 100, 1],
        'freqtime': 1,
        'verb': 1}

# HS model
inp4 = dc(inp3)
inp4['depth'] = inp3['depth'][:2]
inp4['res'] = inp3['res'][:3]

# Compute radial responses
rhs = empymod.dipole(**inp4)  # Step, HS
rhs = empymod.utils.EMArray(np.nan_to_num(rhs))
rtg = empymod.dipole(**inp3)  # " "   Target
rtg = empymod.utils.EMArray(np.nan_to_num(rtg))

# Compute azimuthal response
ahs = empymod.dipole(**inp4, ab=22)  # Step, HS
ahs = empymod.utils.EMArray(np.nan_to_num(ahs))
atg = empymod.dipole(**inp3, ab=22)  # " "   Target
atg = empymod.utils.EMArray(np.nan_to_num(atg))
```

Out:

```
* WARNING :: Offsets < 0.001 m are set to 0.001 m!
* WARNING :: Offsets < 0.001 m are set to 0.001 m!
* WARNING :: Offsets < 0.001 m are set to 0.001 m!
* WARNING :: Offsets < 0.001 m are set to 0.001 m!
```

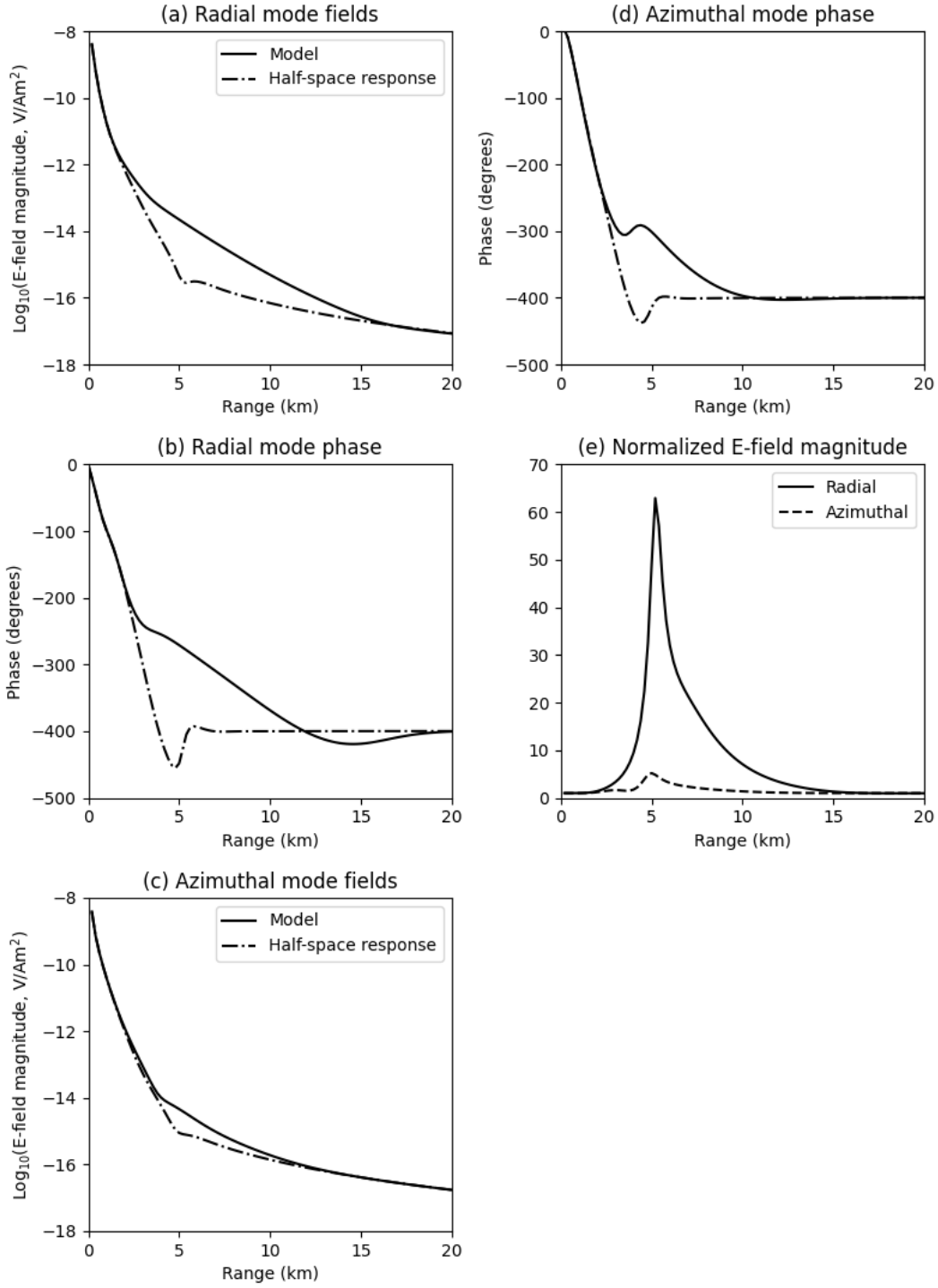
Plot

```
plt.figure(figsize=(9, 13))
plt.subplots_adjust(wspace=.3, hspace=.3)

# Radial amplitude
plt.subplot(321)
plt.title('(a) Radial mode fields')
plt.plot(x/1000, np.log10(rtg.amp()), 'k', label='Model')
plt.plot(x/1000, np.log10(rhs.amp()), 'k-.', label='Half-space response')
plt.axis([0, 20, -18, -8])
plt.xlabel('Range (km)')
plt.ylabel(r'Log$_{10}$(E-field magnitude, V/Am$^2$)')
plt.legend()

# Radial phase
plt.subplot(323)
plt.title('(b) Radial mode phase')
plt.plot(x/1000, rtg.pha(deg=True), 'k')
plt.plot(x/1000, rhs.pha(deg=True), 'k-.')
plt.axis([0, 20, -500, 0])
plt.xlabel('Range (km)')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Phase (degrees)')

# Azimuthal amplitude
plt.subplot(325)
plt.title('(c) Azimuthal mode fields')
plt.plot(x/1000, np.log10(atg.amp()), 'k', label='Model')
plt.plot(x/1000, np.log10(ahs.amp()), 'k-.', label='Half-space response')
plt.axis([0, 20, -18, -8])
plt.xlabel('Range (km)')
plt.ylabel(r'Log$_{10}$(E-field magnitude, V/Am$^2$)')
plt.legend()

# Azimuthal phase
plt.subplot(322)
plt.title('(d) Azimuthal mode phase')
plt.plot(x/1000, atg.pha(deg=True)+180, 'k')
plt.plot(x/1000, ahs.pha(deg=True)+180, 'k-.')
plt.axis([0, 20, -500, 0])
plt.xlabel('Range (km)')
plt.ylabel('Phase (degrees)')

# Normalized
plt.subplot(324)
plt.title('(e) Normalized E-field magnitude')
plt.plot(x/1000, np.abs(rtg/rhs), 'k', label='Radial')
plt.plot(x/1000, np.abs(atg/ahs), 'k--', label='Azimuthal')
plt.axis([0, 20, 0, 70])
plt.xlabel('Range (km)')
plt.legend()

plt.show()
```




```
empymod.Report()
```

Total running time of the script: (0 minutes 2.240 seconds)

Estimated memory usage: 8 MB

Magnetic loop sources and magnetic receivers

Frequency and time-domain modelling of magnetic loop sources and magnetic receivers

Reproducing Figures 2.2-2.5, 4.2-4.5, and 4.7-4.8 of Ward and Hohmann (1988): Frequency- and time-domain isotropic solutions for a full-space (2.2-2.5) and a half-space, where source and receiver are at the interface (4.2-4.5, 4.7-4.8). Source is a loop, receiver is a magnetic dipole.

Reference

- Ward, S. H., and G. W. Hohmann, 1988, Electromagnetic theory for geophysical applications, Chapter 4 of Electromagnetic Methods in Applied Geophysics: SEG, Investigations in Geophysics No. 3, 130–311; DOI: 10.1190/1.9781560802631.ch4.

Requires *empymod v1.10.0* or newer.

```
import empymod
import numpy as np
from scipy.special import erf
import matplotlib.pyplot as plt
from scipy.constants import mu_0
```

Ward and Hohmann, 1988, Fig 4.4

Ward and Hohmann (1988), Equations 4.69a and 4.70:

$$h_z = \frac{m}{4\pi r^3}\left[\frac{9}{2\theta^2 r^2}\mathrm{erf}(\theta \mathrm{r}) - \mathrm{erf}(\theta \mathrm{r}) - \frac{1}{\pi^{1/2}}\left(\frac{9}{\theta \mathrm{r}} + 4\theta \mathrm{r}\right)\exp(-\theta^2 \mathrm{r}^2)\right], \qquad (4.69a)$$

and

$$\frac{\partial h_z}{\partial t} = -\frac{m\rho}{2\pi\mu_0 r^5}\left[9\mathrm{erf}(\theta \mathrm{r}) - \frac{2\theta \mathrm{r}}{\pi^{1/2}}\left(9 + 6\theta^2 \mathrm{r}^2 + 4\theta^4 \mathrm{r}^4\right)\exp(-\theta^2 \mathrm{r}^2)\right], \qquad (4.70)$$

where

$$\theta = \sqrt{\frac{\mu_0}{4t\rho}},$$

t is time (s), ρ is resistivity (Ω is offset (m), and m the magnetic moment (A m^2).

Analytical solutions

```
def hz(t, res, r, m=1.):
    r"""Return equation 4.69a, Ward and Hohmann, 1988.

    Switch-off response (i.e., Hz(t)) of a homogeneous isotropic half-space,
    where the vertical magnetic source and receiver are at the interface.

    Parameters
    ----------
    t : array
        Times (t)
    res : float
        Halfspace resistivity (Ohm.m)
    r : float
        Offset (m)
    m : float, optional
```

(continues on next page)

(continued from previous page)

```
        Magnetic moment, default is 1.

    Returns
    -------
    hz : array
        Vertical magnetic field (A/m)

    """
    theta = np.sqrt(mu_0/(4*res*t))
    theta_r = theta*r

    s = -(9/theta_r+4*theta_r)*np.exp(-theta_r**2)/np.sqrt(np.pi)
    s += erf(theta_r)*(9/(2*theta_r**2)-1)
    s *= m/(4*np.pi*r**3)

    return s
```

```
def dhzdt(t, res, r, m=1.):
    r"""Return equation 4.70, Ward and Hohmann, 1988.

    Impulse response (i.e., dHz(t)/dt) of a homogeneous isotropic half-space,
    where the vertical magnetic source and receiver are at the interface.

    Parameters
    ----------
    t : array
        Times (t)
    res : float
        Halfspace resistivity (Ohm.m)
    r : float
        Offset (m)
    m : float, optional
        Magnetic moment, default is 1.

    Returns
    -------
    dhz : array
        Time-derivative of the vertical magnetic field (A/m/s)

    """
    theta = np.sqrt(mu_0/(4*res*t))
    theta_r = theta*r

    s = (9 + 6 * theta_r**2 + 4 * theta_r**4) * np.exp(-theta_r**2)
    s *= -2 * theta_r / np.sqrt(np.pi)
    s += 9 * erf(theta_r)
    s *= -(m*res)/(2*np.pi*mu_0*r**5)

    return s
```

Survey parameters

```
time = np.logspace(-8, 0, 301)

src = [0, 0, 0, 0, 90]
rec = [100, 0, 0, 0, 90]
depth = 0
res = [2e14, 100]
```

Analytical result

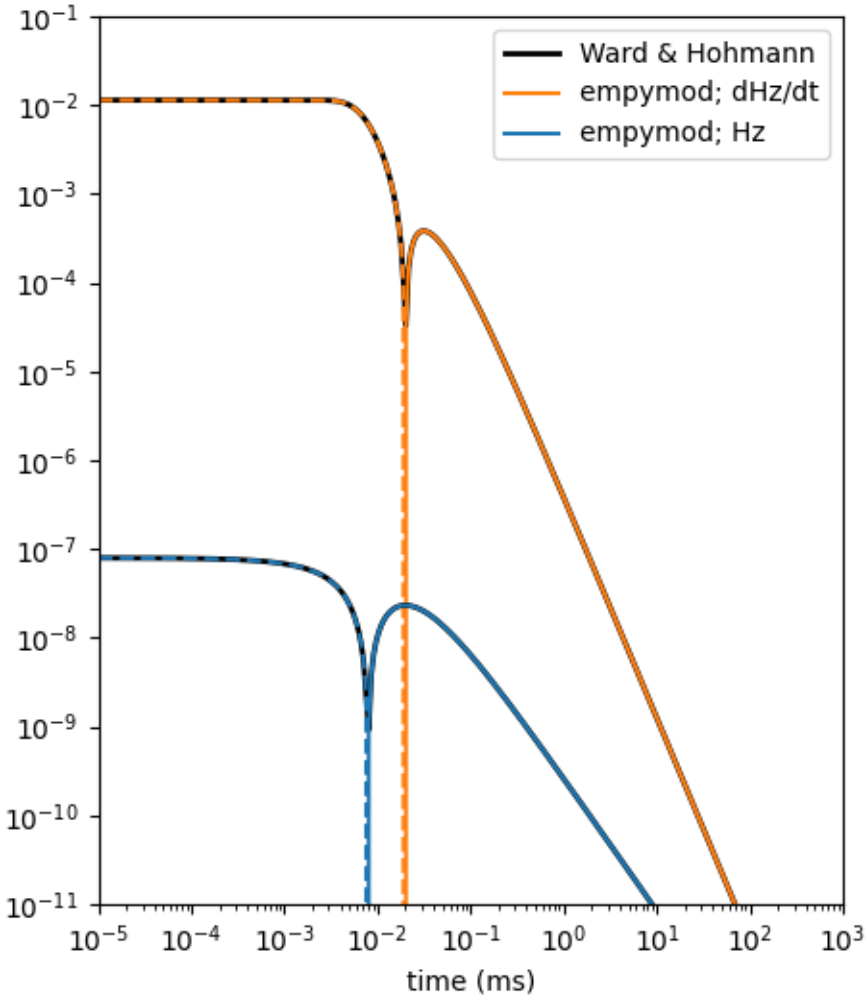
```
hz_ana = hz(time, res[1], rec[0])
dhz_ana = dhzdt(time, res[1], rec[0])
```

Numerical result

```
eperm = [0, 0]  # Reduce early time numerical noise (diffusive approx for air)
inp = {'src': src, 'rec': rec, 'depth': depth, 'res': res,
       'freqtime': time, 'verb': 1, 'xdirect': True, 'epermH': eperm}

hz_num = empymod.loop(signal=-1, **inp)
dhz_num = empymod.loop(signal=0, **inp)
```

Plot the result

```
plt.figure(figsize=(5, 6))

plt.plot(time*1e3, abs(dhz_ana), 'k-', lw=2, label='Ward & Hohmann')
plt.plot(time*1e3, dhz_num, 'C1-', label='empymod; dHz/dt')
plt.plot(time*1e3, -dhz_num, 'C1--')

plt.plot(time*1e3, abs(hz_ana), 'k-', lw=2)
plt.plot(time*1e3, hz_num, 'C0-', label='empymod; Hz')
plt.plot(time*1e3, -hz_num, 'C0--')

plt.xscale('log')
plt.yscale('log')
plt.xlim([1e-5, 1e3])
plt.yticks(10**np.arange(-11., 0))
plt.ylim([1e-11, 1e-1])
plt.xlabel('time (ms)')
plt.legend()

plt.show()
```


Ward & Hohmann
empymod; dHz/dt
empymod; Hz
time (ms)

Original Figure

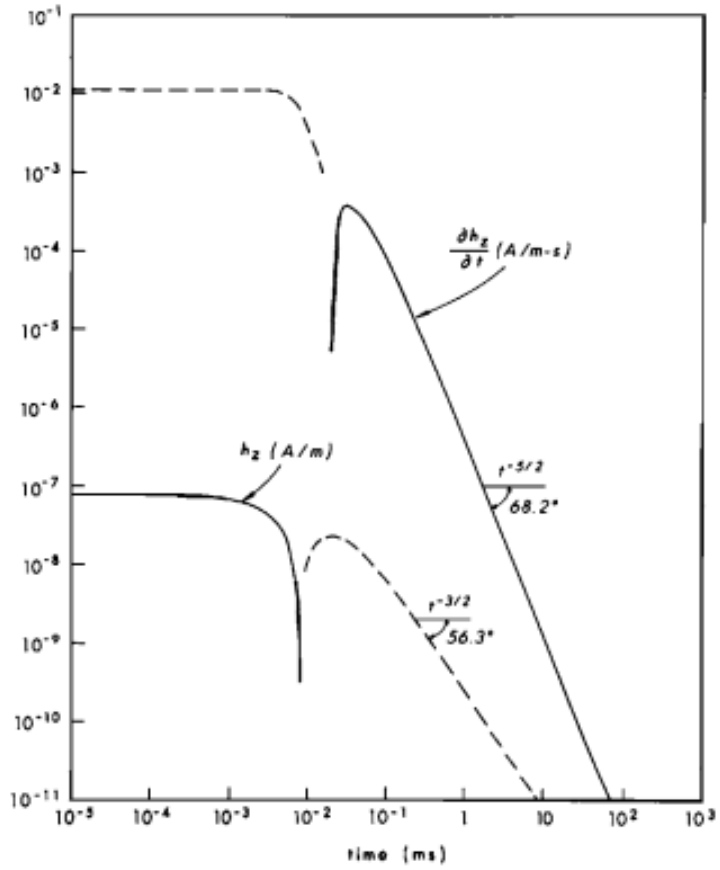


FIG. 4.4. Vertical magnetic field and its time derivative 100 m from a vertical magnetic dipole that is terminated abruptly. Source and receiver at surface of a 100 Ω · m homogeneous earth. Solid lines positive, dashed lines negative.

The following examples are just compared to the figures, without the provided analytical solutions.

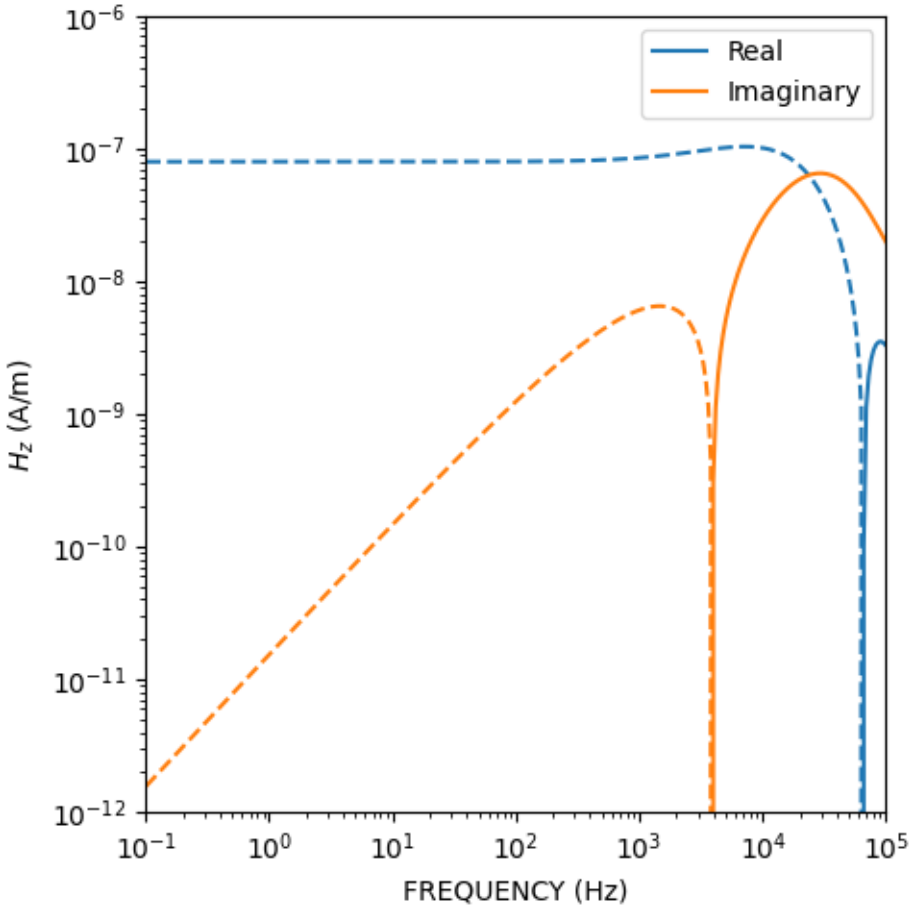
Ward and Hohmann, 1988, Fig 4.2

```
# Survey parameters
freq = np.logspace(-1, 5, 301)
src = [0, 0, 0, 0, 90]
rec = [100, 0, 0, 0, 90]
depth = 0
res = [2e14, 100]

# Computation
inp = {'src': src, 'rec': rec, 'depth': depth, 'res': res,
       'freqtime': freq, 'verb': 1}
fhz_num = empymod.loop(**inp)

# Figure
plt.figure(figsize=(5, 5))

plt.plot(freq, fhz_num.real, 'C0-', label='Real')
plt.plot(freq, -fhz_num.real, 'C0--')
```

(continues on next page)

(continued from previous page)

```
plt.plot(freq, fhz_num.imag, 'C1-', label='Imaginary')
plt.plot(freq, -fhz_num.imag, 'C1--')

plt.xscale('log')
plt.yscale('log')
plt.xlim([1e-1, 1e5])
plt.ylim([1e-12, 1e-6])
plt.xlabel('FREQUENCY (Hz)')
plt.ylabel('$H_z$ (A/m)')
plt.legend()

plt.tight_layout()

plt.show()
```



Original Figure

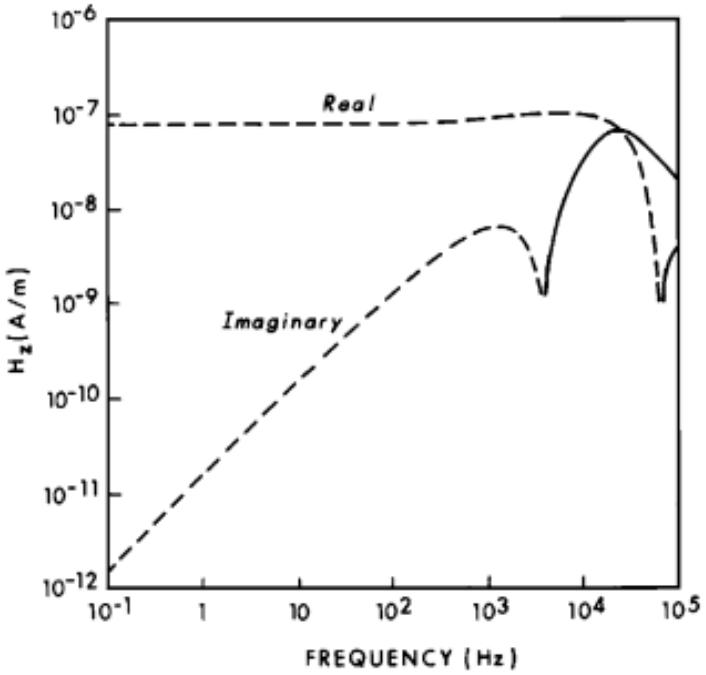


FIG. 4.2. Vertical magnetic field as a function of frequency 100 m from a vertical magnetic dipole at the surface of a 100 Ω · m homogeneous earth.

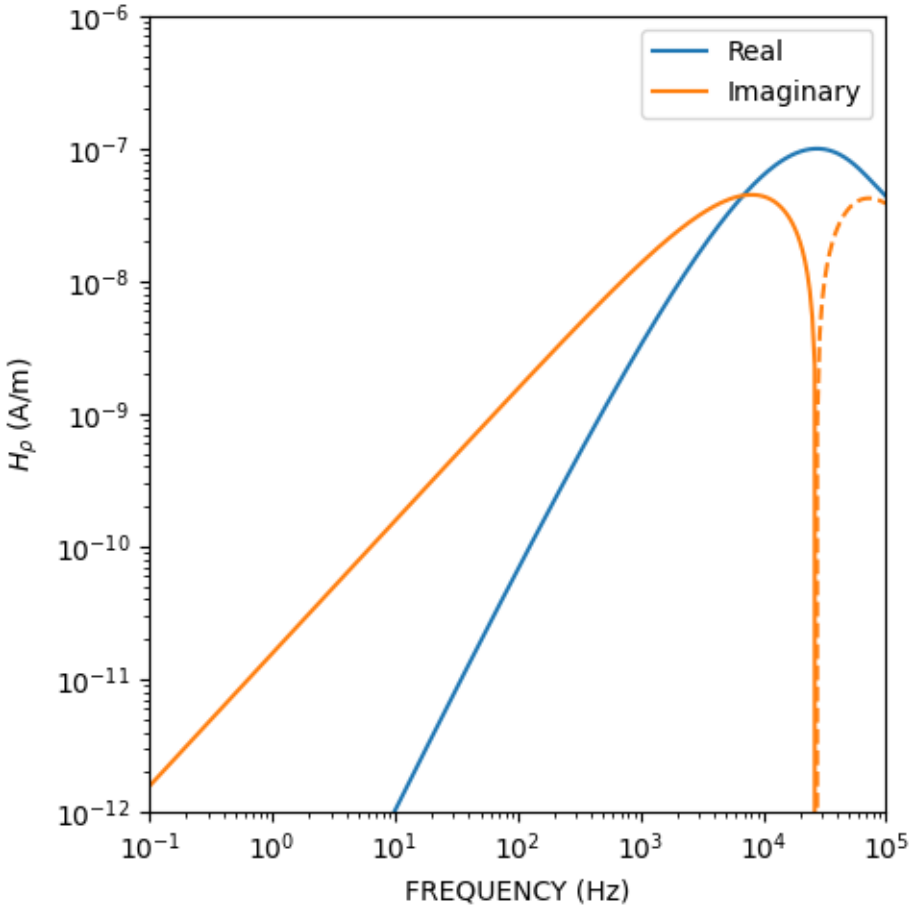
Ward and Hohmann, 1988, Fig 4.3

```
# Survey parameters
freq = np.logspace(-1, 5, 301)
src = [0, 0, 0, 0, 90]
rec = [100, 0, 0, 0, 0]
depth = 0
res = [2e14, 100]

# Computation
inp = {'src': src, 'rec': rec, 'depth': depth, 'res': res,
       'freqtime': freq, 'verb': 1}
fhz_num = empymod.loop(**inp)

# Figure
plt.figure(figsize=(5, 5))

plt.plot(freq, fhz_num.real, 'C0-', label='Real')
plt.plot(freq, -fhz_num.real, 'C0--')

plt.plot(freq, fhz_num.imag, 'C1-', label='Imaginary')
plt.plot(freq, -fhz_num.imag, 'C1--')

plt.xscale('log')
plt.yscale('log')
plt.xlim([1e-1, 1e5])
plt.ylim([1e-12, 1e-6])
plt.xlabel('FREQUENCY (Hz)')
plt.ylabel(r'$H_{\rho}$ (A/m)')
```

(continues on next page)

(continued from previous page)

```
plt.legend()

plt.tight_layout()

plt.show()
```



Original Figure

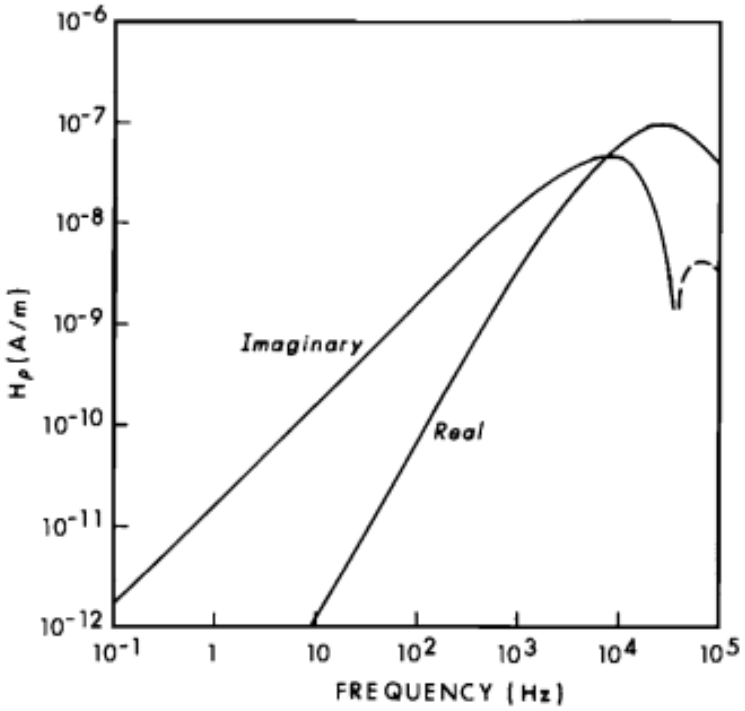


FIG. 4.3. Horizontal magnetic field as a function of frequency 100 m from a vertical magnetic dipole at the surface of a 100 $\Omega \cdot$ m homogeneous earth.

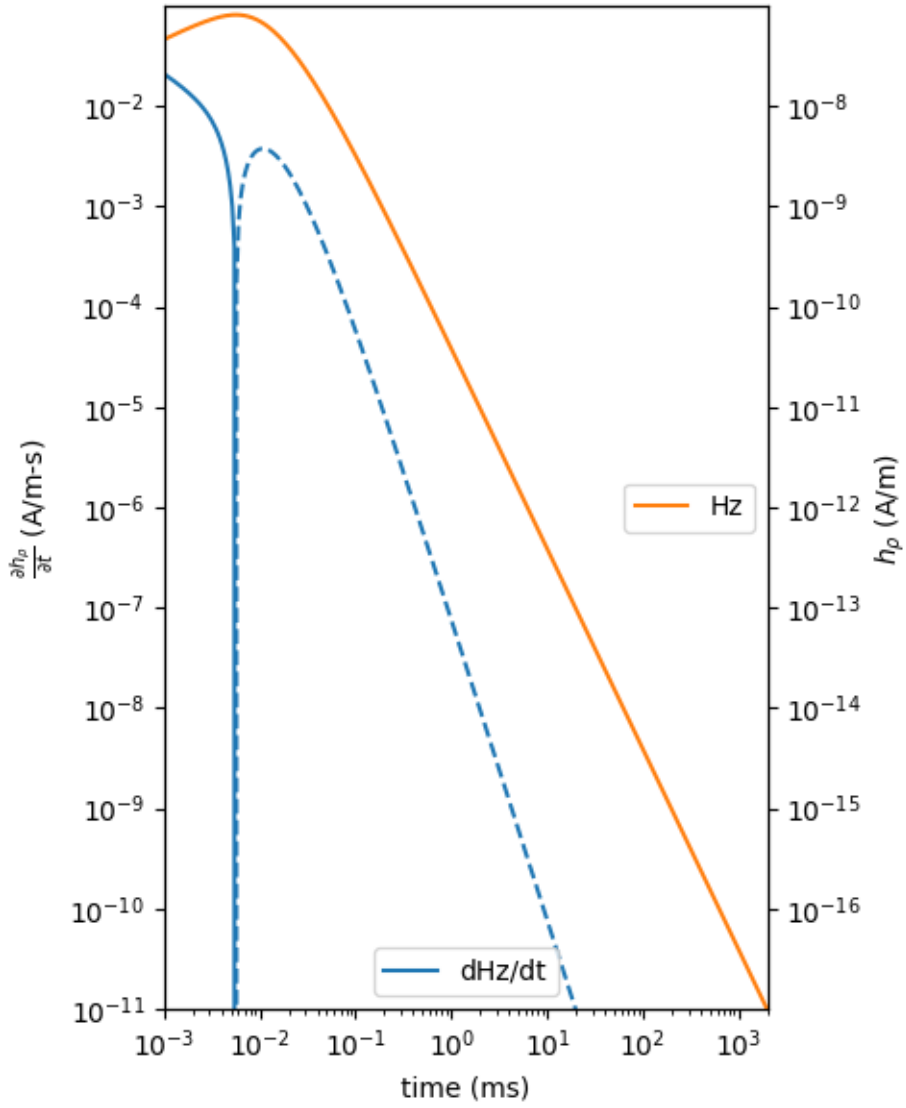
Ward and Hohmann, 1988, Fig 4.5

```
# Survey parameters
time = np.logspace(-6, 0.5, 301)
src = [0, 0, 0, 0, 90]
rec = [100, 0, 0, 0, 0]
depth = 0
res = [2e14, 100]

# Computation
inp = {'src': src, 'rec': rec, 'depth': depth, 'res': res,
       'epermH': eperm, 'freqtime': time, 'verb': 1}
fhz_num = empymod.loop(signal=1, **inp)
fdhz_num = empymod.loop(signal=0, **inp)

# Figure
plt.figure(figsize=(5, 6))

ax1 = plt.subplot(111)
plt.plot(time*1e3, fdhz_num, 'C0-', label='dHz/dt')
plt.plot(time*1e3, -fdhz_num, 'C0--')

plt.xscale('log')
plt.yscale('log')
plt.xlim([1e-3, 2e3])
plt.yticks(10**np.arange(-11., -1))
plt.ylim([1e-11, 1e-1])
plt.xlabel('time (ms)')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel(r'$\frac{\partial h_{\rho}}{\partial t}$ (A/m-s)')
plt.legend(loc=8)

ax2 = ax1.twinx()

plt.plot(time*1e3, fhz_num, 'C1-', label='Hz')
plt.plot(time*1e3, -fhz_num, 'C1--')

plt.xscale('log')
plt.yscale('log')
plt.xlim([1e-3, 2e3])
plt.yticks(10**np.arange(-16., -7))
plt.ylim([1e-17, 1e-7])
plt.ylabel(r'$h_{\rho}$ (A/m)')
plt.legend(loc=5)

plt.tight_layout()
plt.show()
```



Original Figure

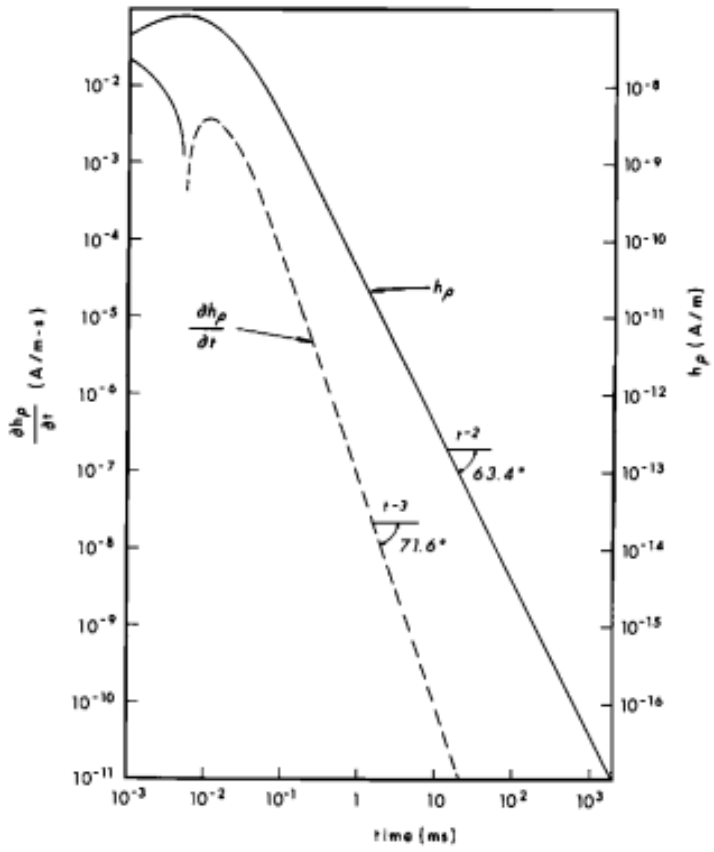


FIG. 4.5. Horizontal magnetic field and its time derivative 100 m from a vertical magnetic dipole that is terminated abruptly. Source and receiver at surface of a 100 $\Omega \cdot$ m homogeneous earth. Solid lines positive, dashed lines negative.

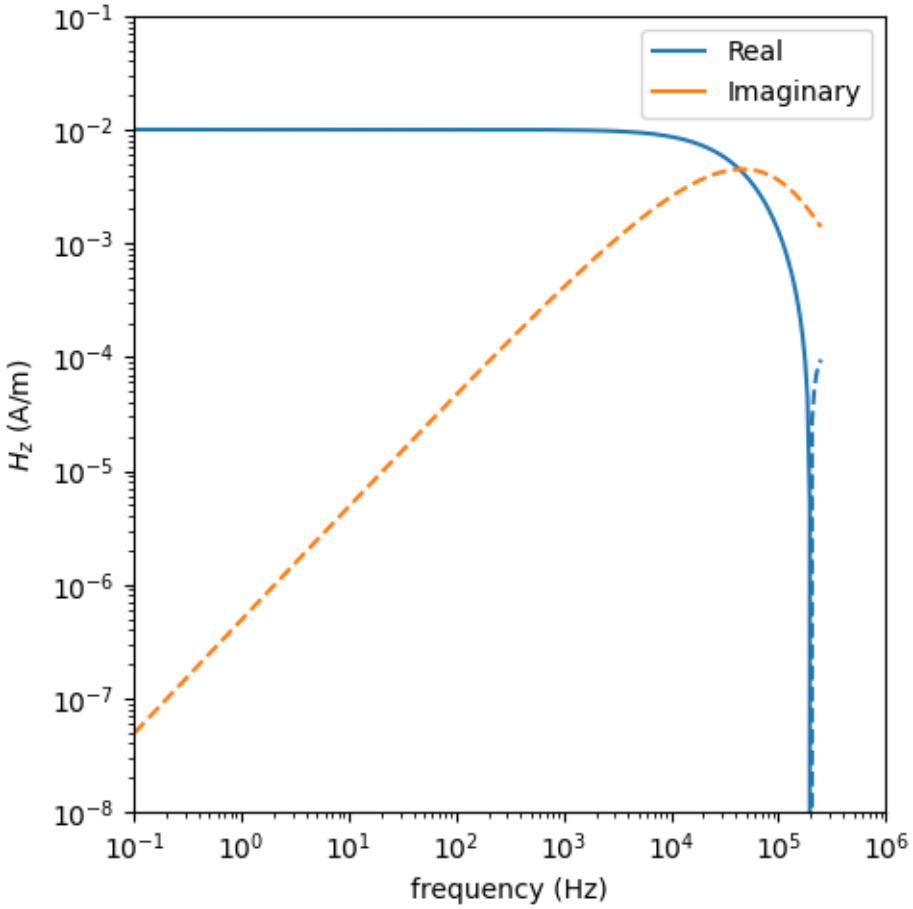
Ward and Hohmann, 1988, Fig 4.7

```
# Survey parameters
radius = 50
area = radius**2*np.pi
freq = np.logspace(-1, np.log10(250000), 301)
src = [radius, 0, 0, 90, 0]
rec = [0, 0, 0, 0, 90]
depth = 0
res = [2e14, 100]
strength = area/(radius/2)
mrec = True

# Computation
inp = {'src': src, 'rec': rec, 'depth': depth, 'res': res,
       'freqtime': freq, 'strength': strength, 'mrec': mrec,
       'verb': 1}
fhz_num = empymod.bipole(**inp)

# Figure
plt.figure(figsize=(5, 5))
```

(continues on next page)

(continued from previous page)

```
plt.plot(freq, fhz_num.real, 'C0-', label='Real')
plt.plot(freq, -fhz_num.real, 'C0--')

plt.plot(freq, fhz_num.imag, 'C1-', label='Imaginary')
plt.plot(freq, -fhz_num.imag, 'C1--')

plt.xscale('log')
plt.yscale('log')
plt.xlim([1e-1, 1e6])
plt.ylim([1e-8, 1e-1])
plt.xlabel('frequency (Hz)')
plt.ylabel('$H_z$ (A/m)')
plt.legend()

plt.tight_layout()

plt.show()
```



Original Figure

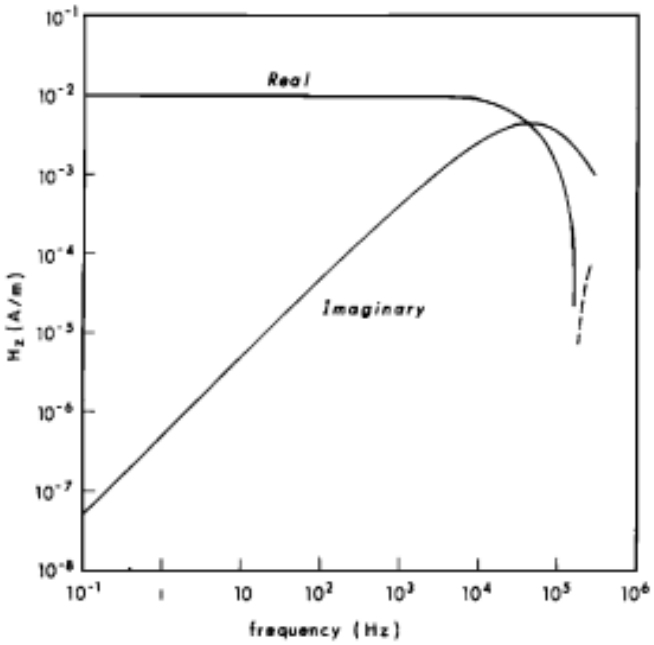


FIG. 4.7. Vertical magnetic field as a function of frequency at the center of a circular loop on a 100 $\Omega \cdot$ m homogeneous earth. The loop has a radius of 50 m and carries a current of 1A.

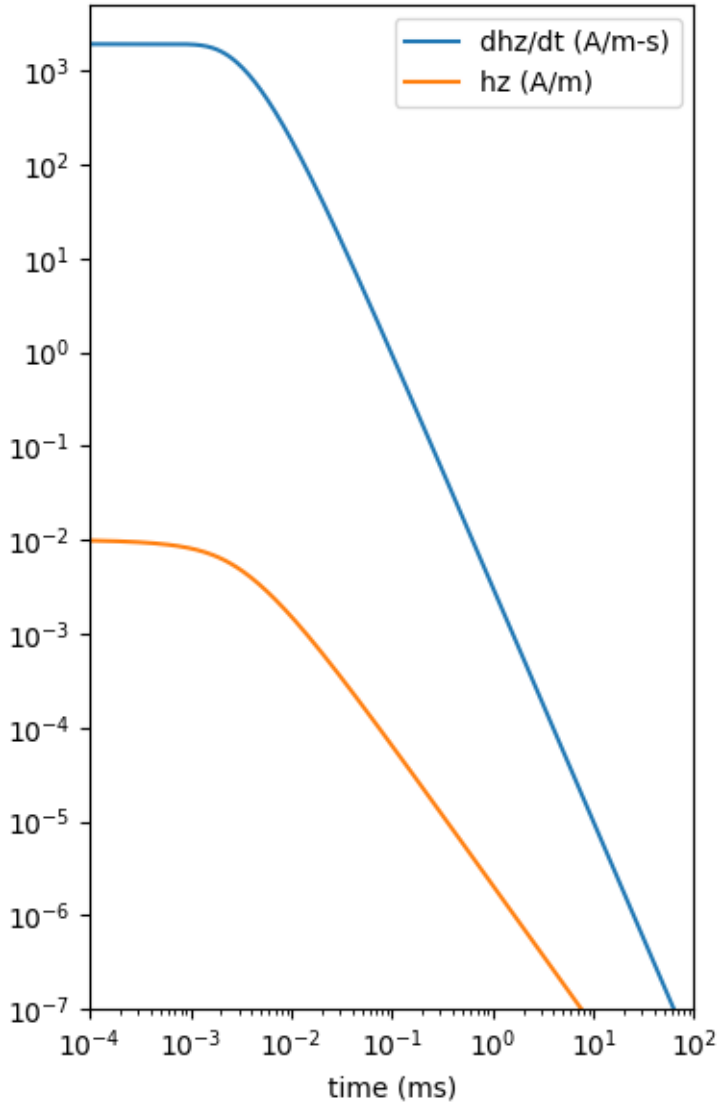
Ward and Hohmann, 1988, Fig 4.8

```
# Survey parameters
radius = 50
area = radius**2*np.pi
time = np.logspace(-7, -1, 301)
src = [radius, 0, 0, 90, 0]
rec = [0, 0, 0, 0, 90]
depth = 0
res = [2e14, 100]
strength = area/(radius/2)
mrec = True

# Computation
inp = {'src': src, 'rec': rec, 'depth': depth, 'res': res,
       'freqtime': time, 'strength': strength, 'mrec': mrec,
       'epermH': eperm, 'verb': 1}

fhz_num = empymod.bipole(signal=-1, **inp)
fdhz_num = empymod.bipole(signal=0, **inp)

# Figure
plt.figure(figsize=(4, 6))

ax1 = plt.subplot(111)
plt.plot(time*1e3, fdhz_num, 'C0-', label=r'dhz/dt (A/m-s)')
plt.plot(time*1e3, -fdhz_num, 'C0--')

plt.plot(time*1e3, fhz_num, 'C1-', label='hz (A/m)')
plt.plot(time*1e3, -fhz_num, 'C1--')
```

(continues on next page)

(continued from previous page)

```
plt.xscale('log')
plt.yscale('log')
plt.xlim([1e-4, 1e2])
plt.yticks(10**np.arange(-7., 4))
plt.ylim([1e-7, 5e3])

plt.xlabel('time (ms)')
plt.legend()

plt.tight_layout()
plt.show()
```



Original Figure

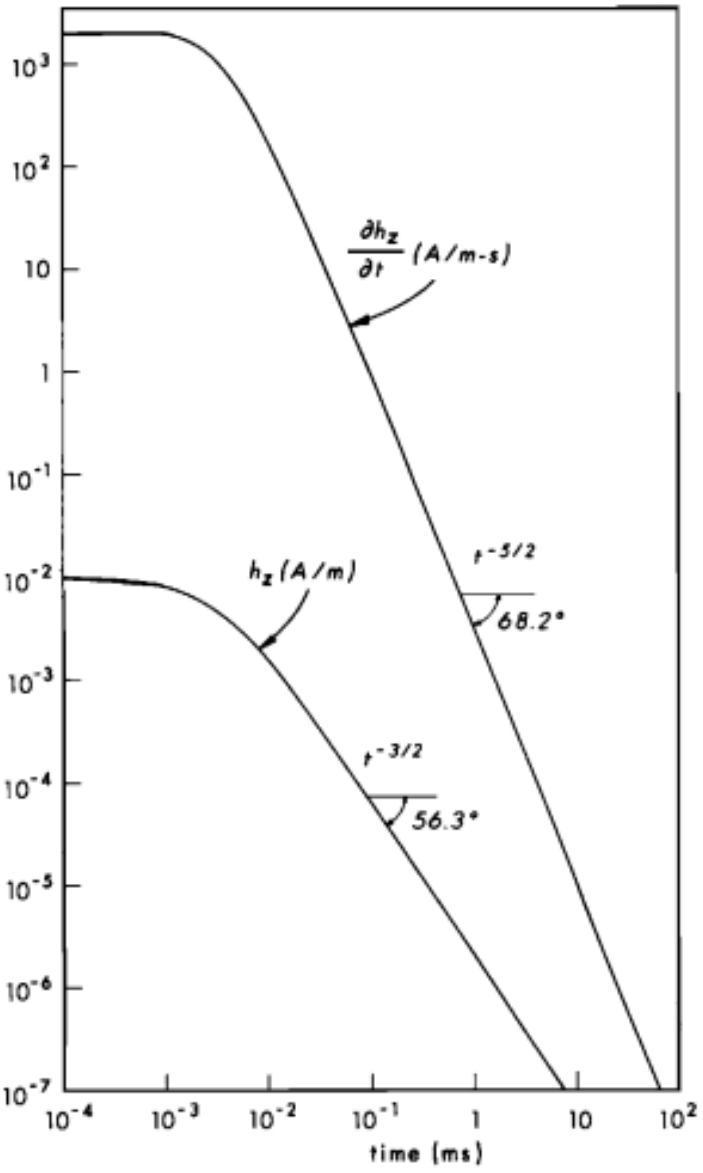


FIG. 4.8. Vertical magnetic field and its time derivative at the center of a circular loop of radius 50 m on a 100 $\Omega \cdot$ m homogeneous earth. A 1A current in the loop is turned off abruptly at zero time.

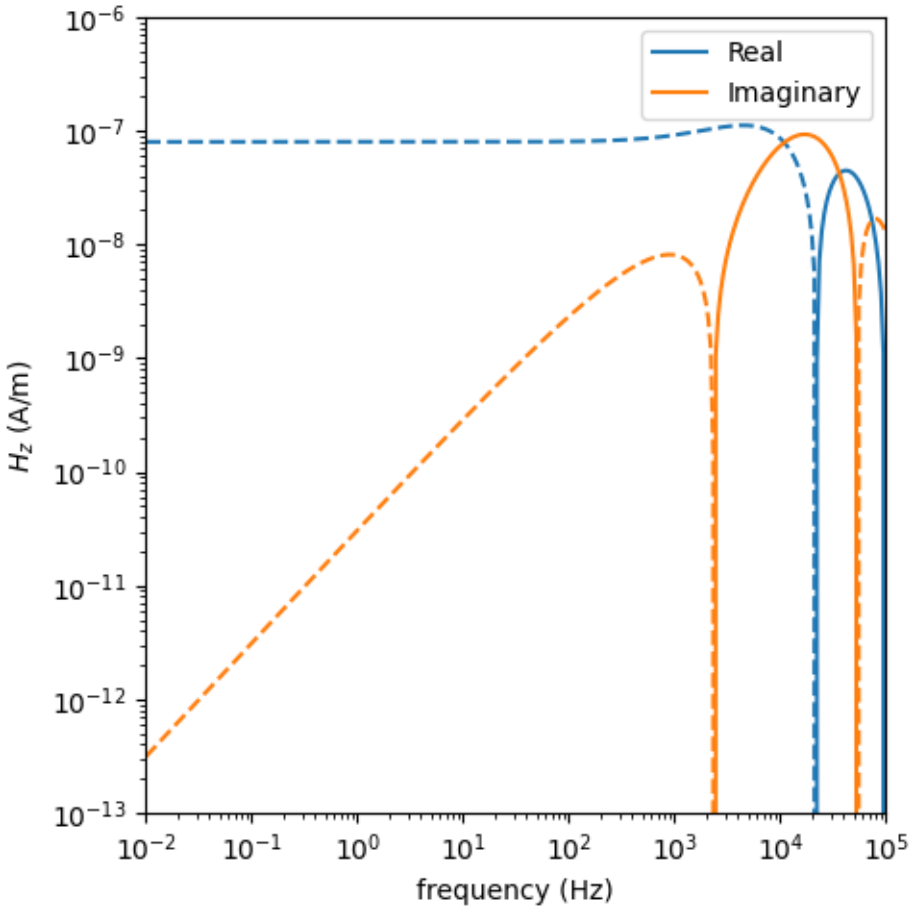
Ward and Hohmann, 1988, Fig 2.2

```
# Survey parameters
freq = np.logspace(-2, 5, 301)
src = [0, 0, 0, 0, 0]
rec = [0, 100, 0, 0, 0]
depth = []
res = 100

# Computation
inp = {'src': src, 'rec': rec, 'depth': depth, 'res': res,
       'freqtime': freq, 'verb': 1}
fhz_num = empymod.loop(**inp)
```

(continues on next page)

(continued from previous page)

```
# Figure
plt.figure(figsize=(5, 5))

plt.plot(freq, fhz_num.real, 'C0-', label='Real')
plt.plot(freq, -fhz_num.real, 'C0--')

plt.plot(freq, fhz_num.imag, 'C1-', label='Imaginary')
plt.plot(freq, -fhz_num.imag, 'C1--')

plt.xscale('log')
plt.yscale('log')
plt.xlim([1e-2, 1e5])
plt.ylim([1e-13, 1e-6])
plt.xlabel('frequency (Hz)')
plt.ylabel('$H_z$ (A/m)')
plt.legend()

plt.tight_layout()

plt.show()
```



Original Figure

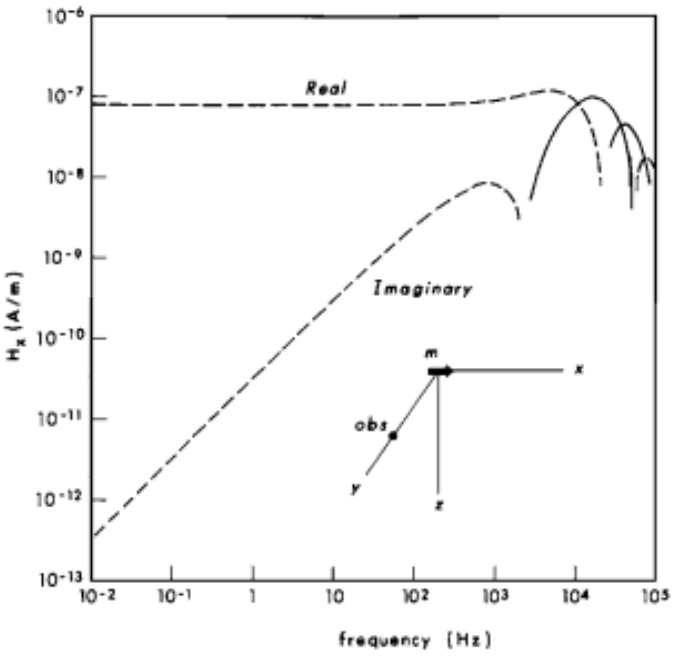


FIG. 2.2. Frequency-domain magnetic field 100 m from a magnetic dipole with moment 1 A-m^2 in a 0.01 S/m whole space. Observation point in equatorial plane of dipole. Solid line positive, dashed line negative.

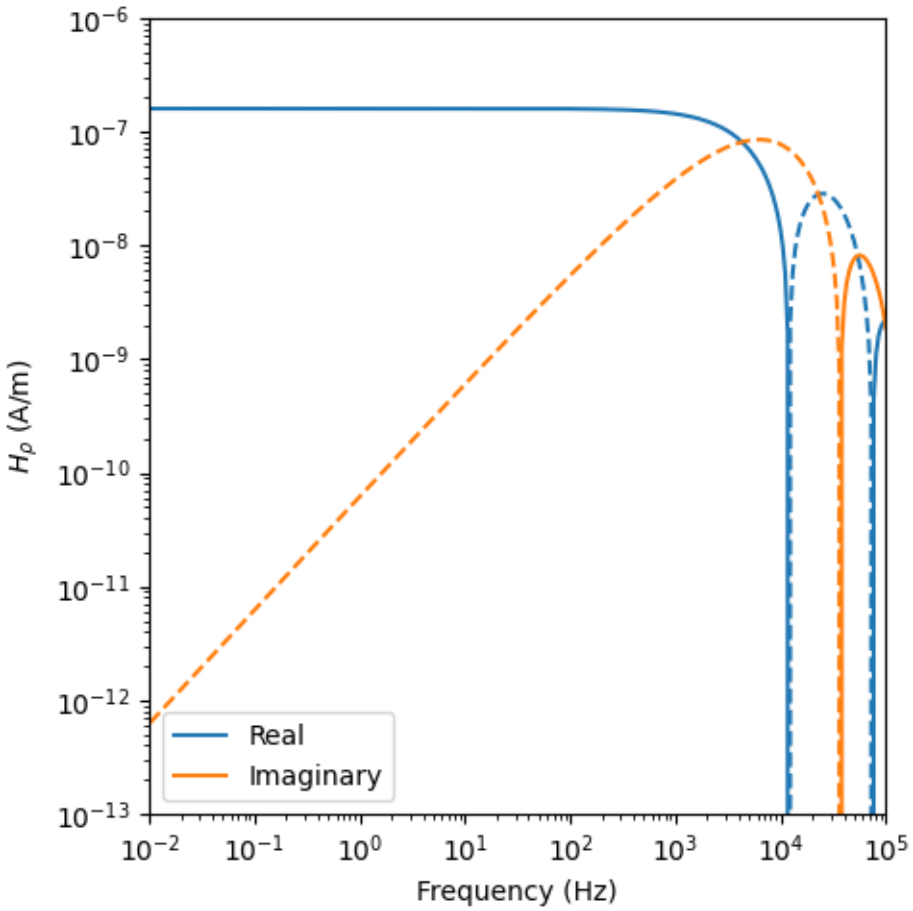
Ward and Hohmann, 1988, Fig 2.3

```
# Survey parameters
freq = np.logspace(-2, 5, 301)
src = [0, 0, 0, 0, 0]
rec = [100, 0, 0, 0, 0]
depth = []
res = 100

# Computation
inp = {'src': src, 'rec': rec, 'depth': depth, 'res': res,
       'freqtime': freq, 'verb': 1}
fhz_num = empymod.loop(**inp)

# Figure
plt.figure(figsize=(5, 5))

plt.plot(freq, fhz_num.real, 'C0-', label='Real')
plt.plot(freq, -fhz_num.real, 'C0--')

plt.plot(freq, fhz_num.imag, 'C1-', label='Imaginary')
plt.plot(freq, -fhz_num.imag, 'C1--')

plt.xscale('log')
plt.yscale('log')
plt.xlim([1e-2, 1e5])
plt.ylim([1e-13, 1e-6])
plt.xlabel('Frequency (Hz)')
plt.ylabel(r'$H_{\rho}$ (A/m)')
```

(continues on next page)

(continued from previous page)

```
plt.legend()

plt.tight_layout()
plt.show()
```



Original Figure

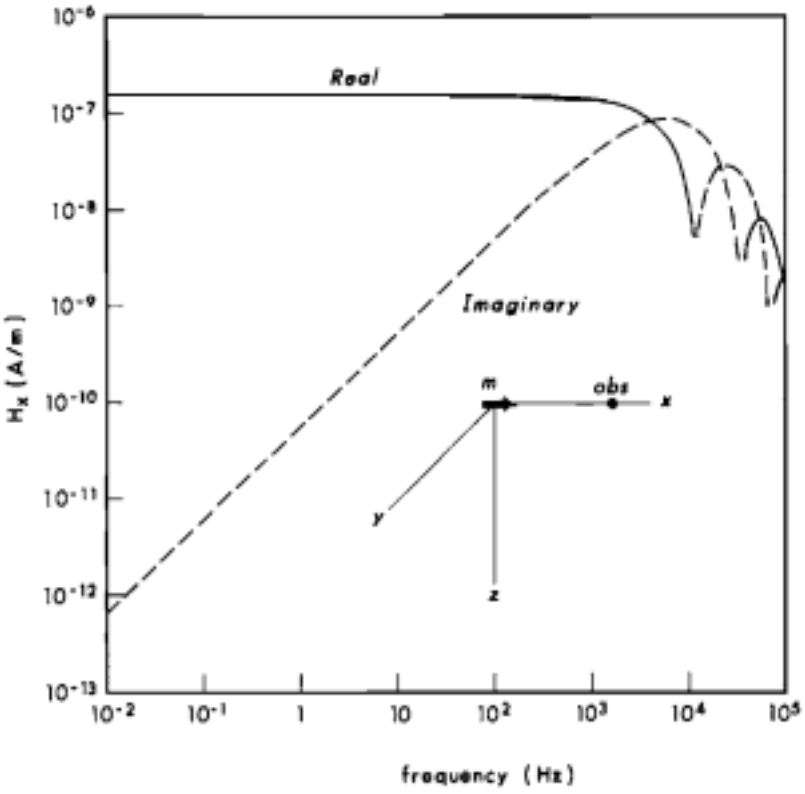


FIG. 2.3. Same as Figure 2.2 except observation point along axis of dipole.

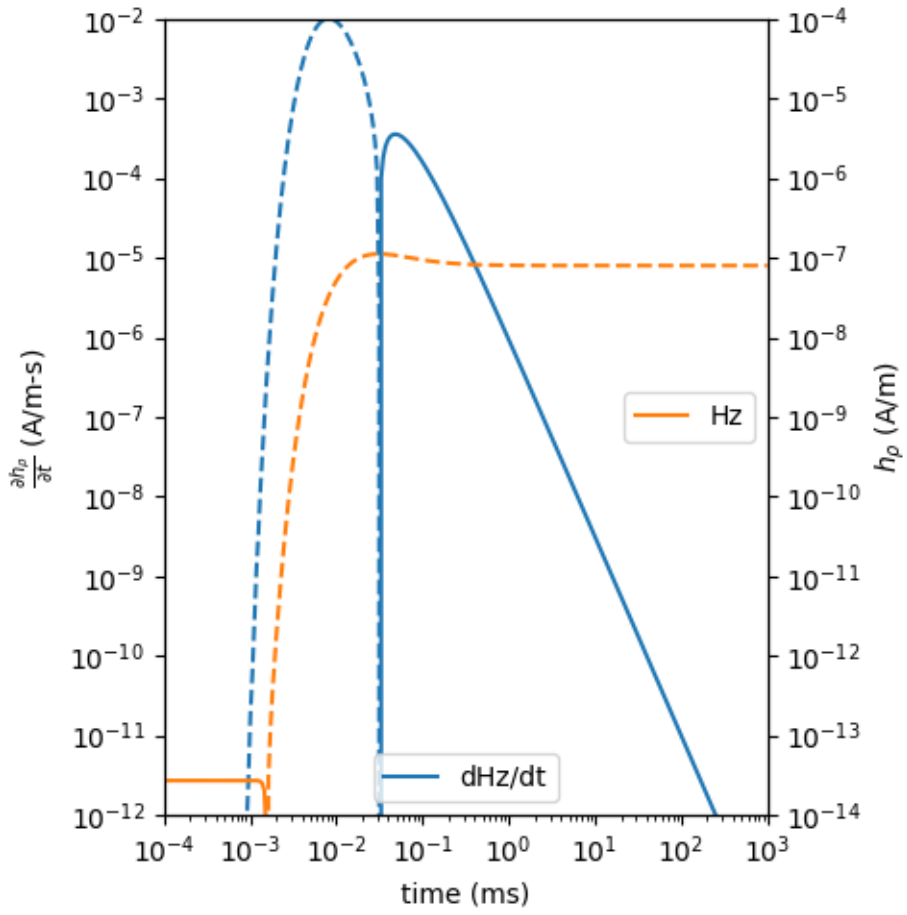
Ward and Hohmann, 1988, Fig 2.4

```
# Survey parameters
time = np.logspace(-7, 0, 301)
src = [0, 0, 0, 0, 0]
rec = [0, 100, 0, 0, 0]
depth = []
res = 100

# Computation
inp = {'src': src, 'rec': rec, 'depth': depth, 'res': res,
       'xdirect': True, 'freqtime': time, 'verb': 1}
fhz_num = empymod.loop(signal=1, **inp)
fdhz_num = empymod.loop(signal=0, **inp)

# Figure
plt.figure(figsize=(5, 5))

ax1 = plt.subplot(111)

plt.plot(time*1e3, fdhz_num, 'C0-', label='dHz/dt')
plt.plot(time*1e3, -fdhz_num, 'C0--')

plt.xscale('log')
plt.yscale('log')
```

(continues on next page)

(continued from previous page)

```
plt.xlim([1e-4, 1e3])
plt.yticks(10**np.arange(-12., -1))
plt.ylim([1e-12, 1e-2])
plt.xlabel('time (ms)')
plt.ylabel(r'$\frac{\partial h_{\rho}}{\partial t}$ (A/m-s)')
plt.legend(loc=8)

ax2 = ax1.twinx()

plt.plot(time*1e3, fhz_num, 'C1-', label='Hz')
plt.plot(time*1e3, -fhz_num, 'C1--')

plt.xscale('log')
plt.yscale('log')
plt.xlim([1e-4, 1e3])
plt.yticks(10**np.arange(-14., -3))
plt.ylim([1e-14, 1e-4])
plt.ylabel(r'$h_{\rho}$ (A/m)')
plt.legend(loc=5)

plt.tight_layout()
plt.show()
```



Original Figure

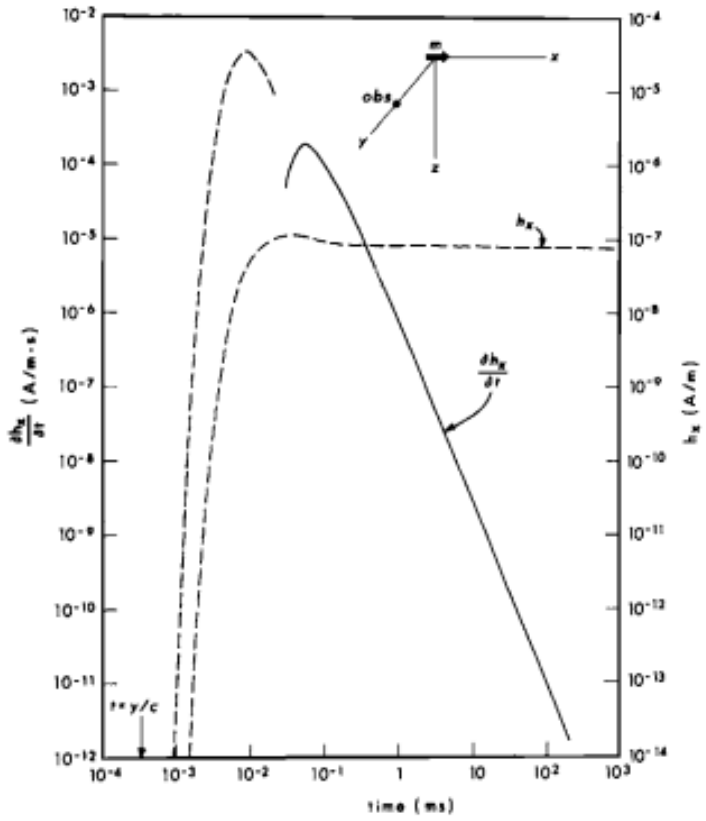


FIG. 2.4. Transient magnetic field 100 m from a magnetic dipole with step-function moment of 1 A-m^2 in a 0.01 S/m whole space. Observation point in equatorial plane of dipole.

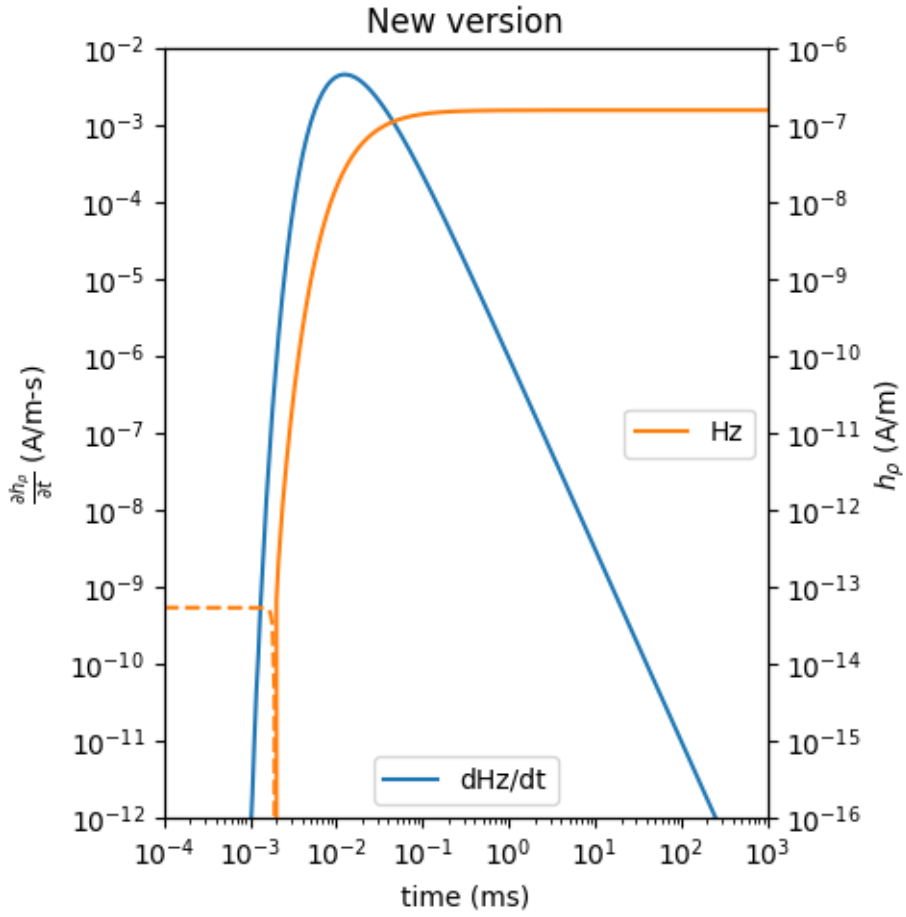
Ward and Hohmann, 1988, Fig 2.5

```
# Survey parameters
time = np.logspace(-7, 0, 301)
src = [0, 0, 0, 0, 0]
rec = [100, 0, 0, 0, 0]
depth = []
res = 100

# Computation
inp = {'src': src, 'rec': rec, 'depth': depth, 'res': res,
       'xdirect': True, 'freqtime': time, 'verb': 1}
fhz_num = empymod.loop(signal=1, **inp)
fdhz_num = empymod.loop(signal=0, **inp)

# Figure
plt.figure(figsize=(5, 5))

ax1 = plt.subplot(111)
plt.title('New version')

plt.plot(time*1e3, fdhz_num, 'C0-', label='dHz/dt')
plt.plot(time*1e3, -fdhz_num, 'C0--')
```

(continues on next page)

(continued from previous page)

```
plt.xscale('log')
plt.yscale('log')
plt.xlim([1e-4, 1e3])
plt.yticks(10**np.arange(-12., -1))
plt.ylim([1e-12, 1e-2])
plt.xlabel('time (ms)')
plt.ylabel(r'$\frac{\partial h_{\rho}}{\partial t}$ (A/m-s)')
plt.legend(loc=8)

ax2 = ax1.twinx()

plt.plot(time*1e3, fhz_num, 'C1-', label='Hz')
plt.plot(time*1e3, -fhz_num, 'C1--')

plt.xscale('log')
plt.yscale('log')
plt.xlim([1e-4, 1e3])
plt.yticks(10**np.arange(-16., -5))
plt.ylim([1e-16, 1e-6])
plt.ylabel(r'$h_{\rho}$ (A/m)')
plt.legend(loc=5)

plt.tight_layout()
plt.show()
```



Original Figure

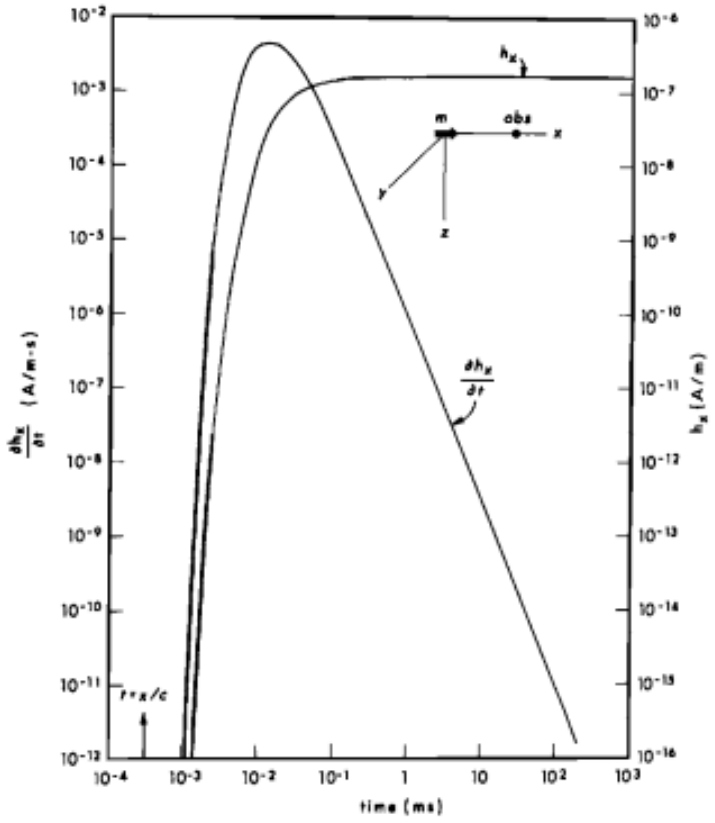


FIG. 2.5. Same as Figure 2.4 except observation point along axis of dipole. Solid line positive, dashed line negative.

```
empymod.Report()
```

Total running time of the script: (0 minutes 9.568 seconds)

Estimated memory usage: 8 MB

5.4.7 Explanations and educational material

Difference between magnetic dipole and loop sources

In this example we look at the differences between an electric loop loop, which results in a magnetic source, and a magnetic dipole source.

The derivation of the electromagnetic field in Hunziker et al. (2015) is for electric and magnetic point-dipole sources and receivers. The magnetic field due to a magnetic source (mm) is obtain from the electric field due to an electric source (ee) using the duality principle, given in their Equation (11),

$$\hat{G}^{mm}_{pq}(\mathbf{x}, \mathbf{x}', s, \eta_{kr}, \zeta_{ij}) = -\hat{G}^{ee}_{pq}(\mathbf{x}, \mathbf{x}', s, -\zeta_{kr}, -\eta_{ij})\,. \qquad (1)$$

Without going into the details of the different parameters, we can focus on the difference between the mm and ee fields for a homogeneous, isotropic fullspace by simplifying this further to

$$\mathbf{G}^{mm}_{\text{dip-dip}} = \frac{\eta}{\zeta}\mathbf{G}^{ee} \quad \xrightarrow{\text{diff. approx}} \quad \frac{\sigma}{\mathrm{i}\omega\mu}\mathbf{G}^{ee}_{\text{dip-dip}}\,. \qquad (2)$$

Here, σ is conductivity (S/m), $\omega = 2\pi f$ is angular frequency (Hz), and μ is the magnetic permeability (H/m). So from Equation (2) we see that the mm field differs from the ee field by a factor $\sigma/(\mathrm{i}\omega\mu)$.

A magnetic dipole source has a moment of $I^m ds$; however, a magnetic dipole source is basically never used in geophysics. Instead a loop of an electric wire is used, which generates a magnetic field. The moment generated by this loop is given by $I^m = \mathrm{i}\omega\mu NAI^e$, where A is the area of the loop (m:math:^2), and N the number of turns of the loop. So the difference between a unit magnetic dipole and a unit loop ($A = 1, N = 1$) is the factor $\mathrm{i}\omega\mu$, hence Equation (2) becomes

$$\mathbf{G}^{mm}_{\text{loop-dip}} = \mathrm{i}\omega\mu\mathbf{G}^{mm}_{\text{dip-dip}} = \sigma\,\mathbf{G}^{ee}_{\text{dip-dip}}\,. \qquad (3)$$

This notebook shows this relation in the frequency domain, as well as for impulse, step-on, and step-off responses in the time domain.

We can actually model an **electric loop** instead of adjusting the magnetic dipole solution to correspond to a loop source. This is shown in the second part of the notebook.

References

- Hunziker, J., J. Thorbecke, and E. Slob, 2015, The electromagnetic response in a layered vertical transverse isotropic medium: A new look at an old problem: Geophysics, 80(1), F1–F18; DOI: 10.1190/geo2013-0411.1.


```
import empymod
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

1. Using the magnetic dipole solution

Survey parameters

- Homogenous fullspace of σ = 0.01 S/m.
- Source at the origin, x-directed.
- Inline receiver with offset of 100 m, x-directed.

```
freq = np.logspace(-1, 5, 301)  # Frequencies (Hz)
time = np.logspace(-6, 0, 301)  # Times (s)
src = [0, 0, 0, 0, 0]     # x-dir. source at the origin [x, y, z, azimuth, dip]
rec = [100, 0, 0, 0, 0]   # x-dir. receiver 100m away from source, inline
cond = 0.01               # Conductivity (S/m)
```

Computation using `empymod`

```
# Collect common parameters
inp = {'src': src, 'rec': rec, 'depth': [], 'res': 1/cond, 'verb': 1}

# Frequency domain
inp['freqtime'] = freq
fee_dip_dip = empymod.bipole(**inp)
fmm_dip_dip = empymod.bipole(msrc=True, mrec=True, **inp)
f_loo_dip = empymod.loop(**inp)

# Time domain
inp['freqtime'] = time

# ee
ee_dip_dip_of = empymod.bipole(signal=-1, **inp)
ee_dip_dip_im = empymod.bipole(signal=0, **inp)
ee_dip_dip_on = empymod.bipole(signal=1, **inp)
```

(continues on next page)

(continued from previous page)

```
# mm dip-dip
dip_dip_of = empymod.bipole(signal=-1, msrc=True, mrec=True, **inp)
dip_dip_im = empymod.bipole(signal=0, msrc=True, mrec=True, **inp)
dip_dip_on = empymod.bipole(signal=1, msrc=True, mrec=True, **inp)

# mm loop-dip
loo_dip_of = empymod.loop(signal=-1, **inp)
loo_dip_im = empymod.loop(signal=0, **inp)
loo_dip_on = empymod.loop(signal=1, **inp)
```

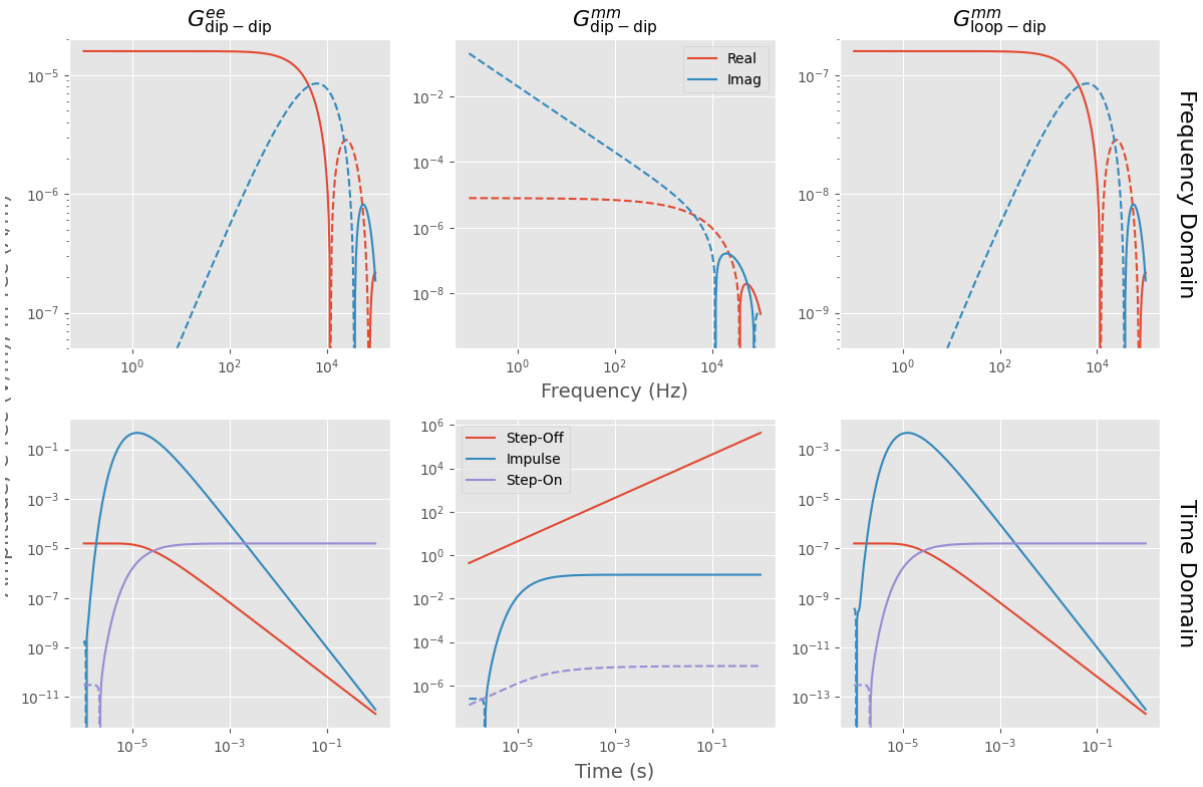
Plot the result

```
fs = 16  # Fontsize

# Figure
fig = plt.figure(figsize=(12, 8))

# Frequency Domain
plt.subplot(231)
plt.title(r'$G^{ee}_{\rm{dip-dip}}$', fontsize=fs)
plt.plot(freq, fee_dip_dip.real, 'C0-', label='Real')
plt.plot(freq, -fee_dip_dip.real, 'C0--')
plt.plot(freq, fee_dip_dip.imag, 'C1-', label='Imag')
plt.plot(freq, -fee_dip_dip.imag, 'C1--')
plt.xscale('log')
plt.yscale('log')
plt.ylim([5e-8, 2e-5])

ax1 = plt.subplot(232)
plt.title(r'$G^{mm}_{\rm{dip-dip}}$', fontsize=fs)
plt.plot(freq, fmm_dip_dip.real, 'C0-', label='Real')
plt.plot(freq, -fmm_dip_dip.real, 'C0--')
plt.plot(freq, fmm_dip_dip.imag, 'C1-', label='Imag')
plt.plot(freq, -fmm_dip_dip.imag, 'C1--')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Frequency (Hz)', fontsize=fs-2)
plt.legend()

plt.subplot(233)
plt.title(r'$G^{mm}_{\rm{loop-dip}}$', fontsize=fs)
plt.plot(freq, f_loo_dip.real, 'C0-', label='Real')
plt.plot(freq, -f_loo_dip.real, 'C0--')
plt.plot(freq, f_loo_dip.imag, 'C1-', label='Imag')
plt.plot(freq, -f_loo_dip.imag, 'C1--')
plt.xscale('log')
plt.yscale('log')
plt.ylim([5e-10, 2e-7])

plt.text(1.05, 0.5, "Frequency Domain", {'fontsize': fs},
         horizontalalignment='left', verticalalignment='center',
         rotation=-90, clip_on=False, transform=plt.gca().transAxes)

# Time Domain
plt.subplot(234)
plt.plot(time, ee_dip_dip_of, 'C0-', label='Step-Off')
plt.plot(time, -ee_dip_dip_of, 'C0--')
plt.plot(time, ee_dip_dip_im, 'C1-', label='Impulse')
```

(continues on next page)

(continued from previous page)

```
plt.plot(time, -ee_dip_dip_im, 'C1--')
plt.plot(time, ee_dip_dip_on, 'C2-', label='Step-On')
plt.plot(time, -ee_dip_dip_on, 'C2--')
plt.xscale('log')
plt.yscale('log')

plt.subplot(235)
plt.plot(time, dip_dip_of, 'C0-', label='Step-Off')
plt.plot(time, -dip_dip_of, 'C0--')
plt.plot(time, dip_dip_im, 'C1-', label='Impulse')
plt.plot(time, -dip_dip_im, 'C1--')
plt.plot(time, dip_dip_on, 'C2-', label='Step-On')
plt.plot(time, -dip_dip_on, 'C2--')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Time (s)', fontsize=fs-2)
plt.legend()

plt.subplot(236)
plt.plot(time, loo_dip_of, 'C0-', label='Step-Off')
plt.plot(time, -loo_dip_of, 'C0--')
plt.plot(time, loo_dip_im, 'C1-', label='Impulse')
plt.plot(time, -loo_dip_im, 'C1--')
plt.plot(time, loo_dip_on, 'C2-', label='Step-On')
plt.plot(time, -loo_dip_on, 'C2--')
plt.xscale('log')
plt.yscale('log')

plt.text(1.05, 0.5, "Time Domain", {'fontsize': fs},
         horizontalalignment='left', verticalalignment='center',
         rotation=-90, clip_on=False, transform=plt.gca().transAxes)

fig.text(-0.01, 0.5, 'Amplitude; e-rec (V/m); m-rec (A/m)',
         va='center', rotation='vertical', fontsize=fs, color='.4')

plt.tight_layout()
plt.show()
```



The figure shows the main points of Equations (2) and (3):

- The magnetic dipole-dipole response differs by a factor $\sigma/(\mathrm{i}\omega\mu)$ from the electric dipole-dipole response. That means for the time-domain that the magnetic response looks more like the time derivative of the electric response (e.g., the magnetic impulse responses resembles the electric step-on response).
- The magnetic loop-dipole response differs only by σ from the electric dipole-dipole response, hence a factor of 0.01.

The units of the response only depend on the receiver, what the receiver actually measures. So if we change the source from a dipole to a loop it does not change the units of the received responses.

2. Using an electric loop

We can use `empymod` to model arbitrary shaped sources by simply adding point dipole sources together. This is what `empymod` does internally to model a finite length dipole (`empymod.bipole`), where it uses a Gaussian quadrature with a few points.

Here, we are going to compare the result from `loop`, as presented above, with two different simulations of an electric loop source, assuming a square loop which sides are 1 m long, so the area correspond to one square meter.

Plotting routines

```
def discrete_cmap(N, base_cmap=None):
    """Create an N-bin discrete colormap from the specified input map
    https://gist.github.com/jakevdp/91077b0cae40f8f8244a
    """
    base = plt.cm.get_cmap(base_cmap)
    color_list = base(np.linspace(0, 1, N))
    cmap_name = base.name + str(N)
    return base.from_list(cmap_name, color_list, N)
```

```
def plot_result(data1, data2, x, title, vmin=-15., vmax=-7., rx=0):
    """Plot result."""
    fig = plt.figure(figsize=(18, 10))

    def setplot(name):
        """Plot settings"""
        plt.title(name)
        plt.xlim(rx.min(), rx.max())
        plt.ylim(rx.min(), rx.max())
        plt.axis("equal")

    # Plot Re(data)
    ax1 = plt.subplot(231)
    setplot(r"(a) |Re(magn.dip*iwu)|")
    cf0 = plt.pcolormesh(rx, rx, np.log10(np.abs(data1.real)), linewidth=0,
                         rasterized=True, cmap="viridis", vmin=vmin, vmax=vmax)

    ax2 = plt.subplot(232)
    setplot(r"(b) |Re(el. square)|")
    plt.pcolormesh(rx, rx, np.log10(np.abs(data2.real)), linewidth=0,
                   rasterized=True, cmap="viridis", vmin=vmin, vmax=vmax)

    ax3 = plt.subplot(233)
    setplot(r"(c) Error real part")
    error_r = np.abs((data1.real-data2.real)/data1.real)*100
    cf2 = plt.pcolormesh(rx, rx, np.log10(error_r), vmin=-2, vmax=2,
                         linewidth=0, rasterized=True,
                         cmap=discrete_cmap(8, "RdBu_r"))

    # Plot Im(data)
    ax4 = plt.subplot(234)
    setplot(r"(d) |Im(magn.dip*iwu)|")
    plt.pcolormesh(rx, rx, np.log10(np.abs(data1.imag)), linewidth=0,
                   rasterized=True, cmap="viridis", vmin=vmin, vmax=vmax)

    ax5 = plt.subplot(235)
    setplot(r"(e) |Im(el. square)|")
    plt.pcolormesh(rx, rx, np.log10(np.abs(data2.imag)), linewidth=0,
                   rasterized=True, cmap="viridis", vmin=vmin, vmax=vmax)

    ax6 = plt.subplot(236)
    setplot(r"(f) Error imag part")
    error_i = np.abs((data1.imag-data2.imag)/data1.imag)*100
    plt.pcolormesh(rx, rx, np.log10(error_i), vmin=-2, vmax=2,
                   linewidth=0, rasterized=True,
                   cmap=discrete_cmap(8, "RdBu_r"))

    # Colorbars
    fig.colorbar(cf0, ax=[ax1, ax2, ax3], label=r"$\log_{10}$ Amplitude (A/m)")
    cbar = fig.colorbar(cf2, ax=[ax4, ax5, ax6], label=r"Relative Error")
    cbar.set_ticks([-2, -1, 0, 1, 2])
    cbar.ax.set_yticklabels([r"$0.01\,\%$", r"$0.1\,\%$", r"$1\,\%$",
                             r"$10\,\%$", r"$100\,\%$"])

    # Axis label
    fig.text(0.4, 0.05, "Inline Offset (m)", fontsize=14)
    fig.text(0.08, 0.5, 'Crossline Offset (m)', rotation=90, fontsize=14)

    # Title
    fig.suptitle(title, y=.95, fontsize=20)
    plt.show()
```

Model parameters

- Resistivity: 1Ω m fullspace

Survey

- Source at [0, 0, 0]
- Receivers at [x, y, 10]
- frequencies: 100 Hz.
- Offsets: -250 m - 250 m

```
# Survey parameters
x = ((np.arange(502))-250.5)
rx = np.repeat([x, ], np.size(x), axis=0)
ry = rx.transpose()
rxx = rx.ravel()
ryy = ry.ravel()

# Model
model = {
    'depth': [],         # Fullspace
    'res': 1.,           # 1 Ohm.m
    'freqtime': 100,     # 100 Hz
    'htarg': {'pts_per_dec': -1},
    'verb': 1,
}
```

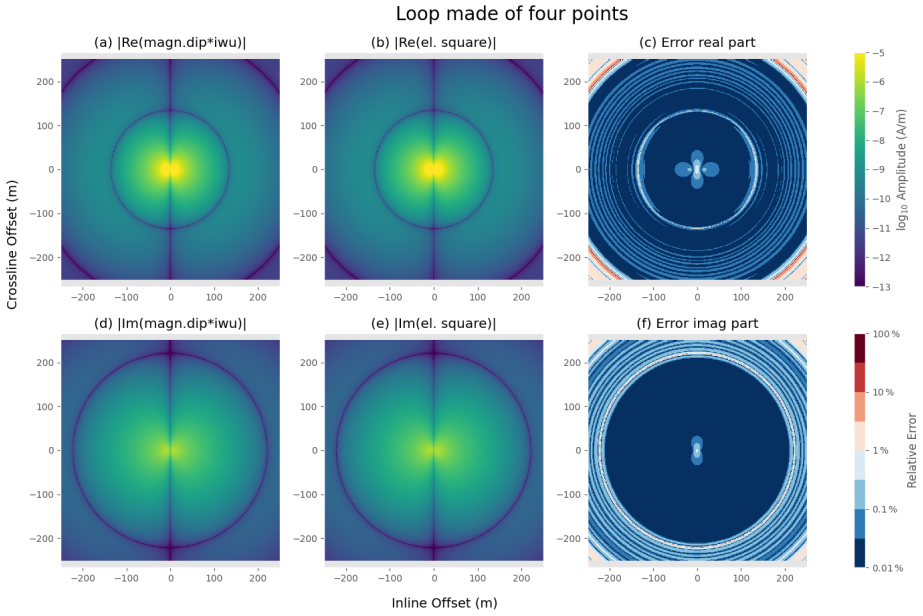
Compute `empymod.loop` result

```
epm_loop = empymod.loop(src=[0, 0, 0, 0, 90], rec=[rxx, ryy, 10, 0, 0],
                        **model).reshape(np.shape(rx))
```

2.1 Point dipoles at (x, y) using `empymod.dipole`

- (0.5, 0), ab=42
- (0, 0.5), ab=41
- (-0.5, 0), ab=-42
- (0, -0.5), ab=-41

```
rec_dip = [rxx, ryy, 10]

square_pts = +empymod.dipole(src=[+0.5, +0.0, 0], rec=rec_dip, ab=42,
                             **model).reshape(np.shape(rx))
square_pts += empymod.dipole(src=[+0.0, +0.5, 0], rec=rec_dip, ab=41,
                             **model).reshape(np.shape(rx))
square_pts -= empymod.dipole(src=[-0.5, +0.0, 0], rec=rec_dip, ab=42,
                             **model).reshape(np.shape(rx))
square_pts -= empymod.dipole(src=[+0.0, -0.5, 0], rec=rec_dip, ab=41,
                             **model).reshape(np.shape(rx))

plot_result(epm_loop, square_pts, x, 'Loop made of four points',
            vmin=-13, vmax=-5, rx=x)
```

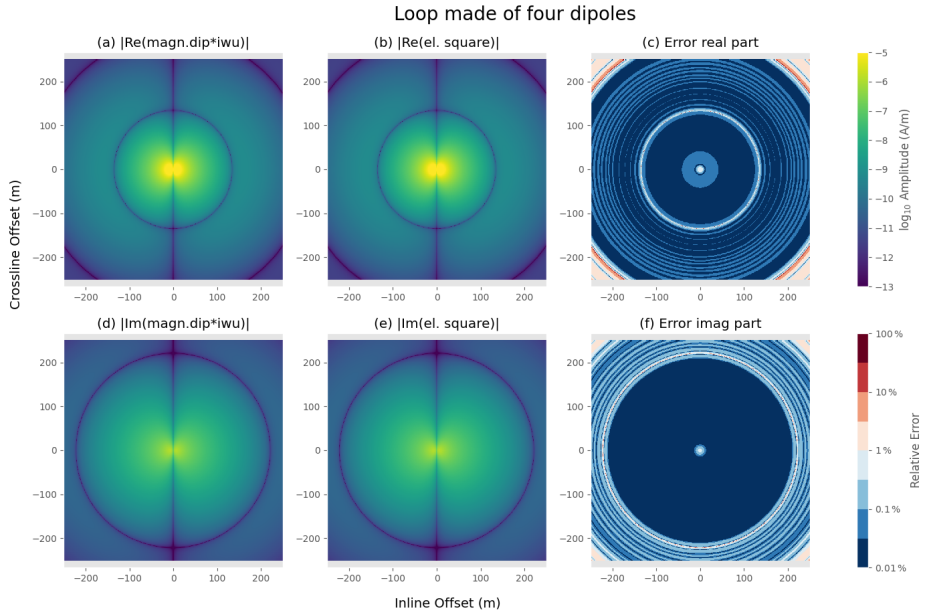


2.2 Finite length dipoles using `empymod.bipole`

Each simulated with a 5pt Gaussian quadrature. The dipoles are:

- (-0.5, -0.5) to (+0.5, -0.5)
- (+0.5, -0.5) to (+0.5, +0.5)
- (+0.5, +0.5) to (-0.5, +0.5)
- (-0.5, +0.5) to (-0.5, -0.5)

```
inp_dip = {
    'rec': [rxx, ryy, 10, 0, 0],
    'mrec': True,
    'srcpts': 5  # Gaussian quadr. with 5 pts to simulate a finite length dip.
}

square_dip = +empymod.bipole(src=[+0.5, +0.5, -0.5, +0.5, 0, 0],
                             **inp_dip, **model)
square_dip += empymod.bipole(src=[+0.5, -0.5, +0.5, +0.5, 0, 0],
                             **inp_dip, **model)
square_dip += empymod.bipole(src=[-0.5, -0.5, +0.5, -0.5, 0, 0],
                             **inp_dip, **model)
square_dip += empymod.bipole(src=[-0.5, +0.5, -0.5, -0.5, 0, 0],
                             **inp_dip, **model)
square_dip = square_dip.reshape(np.shape(rx))

plot_result(epm_loop, square_dip, x, 'Loop made of four dipoles',
            vmin=-13, vmax=-5, rx=x)
```



Close to the source the results between

- (1) a magnetic dipole,
- (2) an electric loop conisting of four point sources, and
- (3) an electric loop consisting of four finite length dipoles,

differ, as expected. However, for the vast majority they are identical. Skin depth for our example with $\rho = 1\Omega$ m and $f = 100$ Hz is roughly 50 m, so the results are basically identical for 4-5 skin depths, after which the signal is very low.

```
empymod.Report()
```

Total running time of the script: (0 minutes 10.073 seconds)

Estimated memory usage: 158 MB

Digital Linear Filters

Graphical explanation of the differences between standard DLF, lagged convolution DLF, and splined DLF.

The comments here apply generally to the digital linear filter method. Having `empymod` in mind, they are particularly meant for the Hankel (Bessel-Fourier) transform from the wavenumber-frequency domain ($k - f$) to the space-frequency domain ($x - f$), and for the Fourier transform from the space-frequency domain ($x - f$) to the space-time domain ($x - t$).

1. Introduction

This introduction is taken from Werthmüller et al. (2018), which can be found in the repo empymod/article-fdesign.

In electromagnetics we often have to evaluate integrals of the form

$$F(r) = \int_0^\infty f(l)K(lr)\,dl\ ,$$

where l and r denote input and output evaluation values, respectively, and K is the kernel function. In the specific case of the Hankel transform l corresponds to wavenumber, r to offset, and K to Bessel functions; in the case of the Fourier transform l corresponds to frequency, r to time, and K to sine or cosine functions. In both cases it is

an infinite integral which numerical integration is very time-consuming because of the slow decay of the kernel function and its oscillatory behaviour.

By substituting $r = e^x$ and $l = e^{-y}$ we get

$$e^x F(e^x) = \int_{-\infty}^{\infty} f(e^{-y}) K(e^{x-y}) e^{x-y}\, dy\ .$$

This can be re-written as a convolution integral and be approximated for an N-point filter by

$$F(r) \approx \sum_{n=1}^{N} \frac{f(b_n/r)h_n}{r}\ ,$$

where h is the digital linear filter, and the logarithmically spaced filter abscissae is a function of the spacing Δ and the shift δ,

$$b_n = \exp\left\{\Delta(-N/2 + n) + \delta\right\}\ .$$

From the penultimate equation it can be seen that the filter method requires N evaluations at each r. For example, to compute the frequency domain result for 100 offsets with a 201 pt filter requires 20,100 evaluations in the wavenumber domain. This is why the DLF often uses interpolation to minimize the required evaluations, either for $F(r)$ in what is referred to as *lagged convolution DLF*, or for $f(l)$, which we call here *splined DLF*.

```
import empymod
import numpy as np
import matplotlib.pyplot as plt
from copy import deepcopy as dc
plt.style.use('ggplot')
```

2. How the DLF works

Design a very short, digital linear filter

For this we use `empymod.fdesign`. This is outside the scope of this notebook. If you are interested have a look at the article-fdesign-repo for more information regarding the design of digital linear filters.

We design a 5pt filter using the theoretical transform pair

$$\int_0^{\infty} l \exp\left(-l^2\right) J_0(lr) dl = \exp\left(\frac{-r^2}{4}\right)/2\ .$$

A 5 pt filter is very short for this problem, so we expect a considerable error level. In designing the filter we force the filter to be better than a relative error of 5 %.

If you want to play around with this example I recommend to set `verb=2` and `plot=2` to get some feedback from the minimization procedure.

```
filt = empymod.fdesign.design(
        n=5,                            # 5 point filter
        spacing=(0.55, 0.65, 101),
        shift=(0.6, 0.7, 101),
        fI=empymod.fdesign.j0_1(),
        r=np.logspace(0, 1, 100),
        r_def=(1, 1, 10),
        error=0.05,    # 5 % error level. If you set this too low you will
        verb=1,        # #                not find a filter with only 5 points.
        plot=0,
)

print('Filter base    ::', filt.base)
print('Filter weights ::', filt.j0)
```

Out:

```
Filter base    :: [0.59929579 1.07250818 1.91937574 3.43494186 6.14722033]
Filter weights :: [ 0.84042401 -0.00226984  0.57950981 -0.82310148  0.22837621]
```

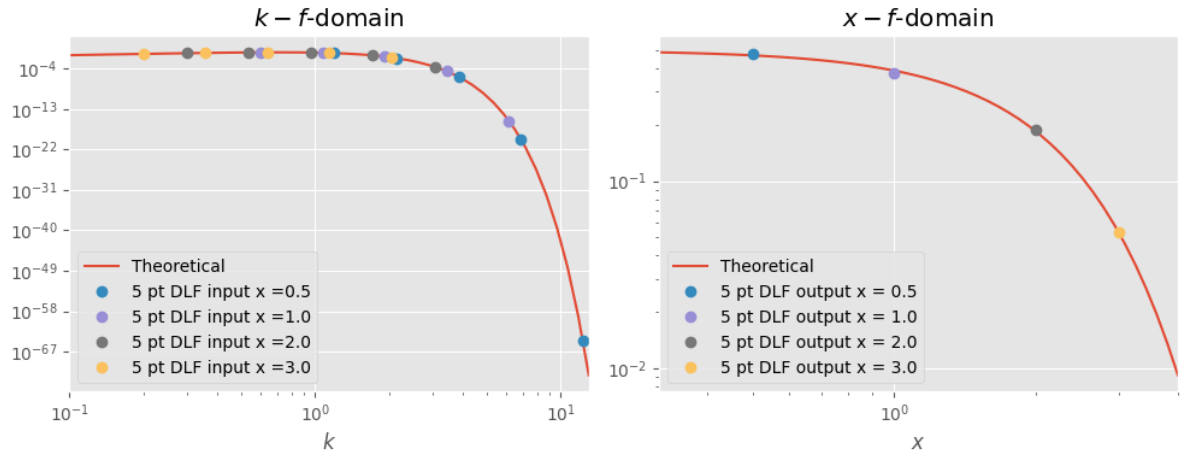
Now we carry out the DLF and check how good it is.

```
# Desired x-f-domain points (output domain)
x = np.array([0.5, 1, 2, 3])

# Required k-f-domain points (input domain)
k = filt.base/x[:, None]

# Get the theoretical transform pair
tp = empymod.fdesign.j0_1()

# Compute the value at the five required wavenumbers
k_val = tp.lhs(k)

# Weight the values and sum them up
x_val_filt = np.dot(k_val, filt.j0)/x

# Compute the theoretical value for comparison
x_val_theo = tp.rhs(x)

# Compute relative error
print('A DLF for this problem with only a 5 pt filter is difficult. We used')
print('an error-limit of 0.05 in the filter design, so we expect the result')
print('to have a relative error of less than 5 %.\n')
print('Theoretical value   ::', '; '.join(
      ['{:G}'.format(i) for i in x_val_theo]))
print('DLF value           ::', '; '.join(
      ['{:G}'.format(i) for i in x_val_filt]))
relerror = np.abs((x_val_theo-x_val_filt)/x_val_theo)
print('Rel. error 5 pt DLF ::', ' %   ; '.join(
      ['{:G}'.format(i) for i in np.round(relerror*100, 1)]), '%')

# Figure
plt.figure(figsize=(10, 4))
plt.suptitle(r'DLF example for $J_0$ Hankel transform using 5 pt filter',
             y=1.05)

# x-axis values for the theoretical plots
x_k = np.logspace(-1, np.log10(13))
x_x = np.logspace(-0.5, np.log10(4))

# k-f-domain
plt.subplot(121)
plt.title(r'$k-f$-domain')
plt.loglog(x_k, tp.lhs(x_k), label='Theoretical')
for i, val in enumerate(x):
    plt.loglog(k[i, :], k_val[i, ], 'o', label='5 pt DLF input x ='+str(val))
plt.legend()
plt.xlabel(r'$k$')
plt.xlim([x_k.min(), x_k.max()])

# x-f-domain
plt.subplot(122)
plt.title(r'$x-f$-domain')
plt.loglog(x_x, tp.rhs(x_x), label='Theoretical')
for i, val in enumerate(x):
    plt.loglog(val, x_val_filt[i], 'o', label='5 pt DLF output x = '+str(val))
plt.legend()
```

(continues on next page)

(continued from previous page)

```
plt.xlabel(r'$x$')
plt.xlim([x_x.min(), x_x.max()])

plt.tight_layout()
plt.show()
```



Out:

```
A DLF for this problem with only a 5 pt filter is difficult. We used
an error-limit of 0.05 in the filter design, so we expect the result
to have a relative error of less than 5 %.

Theoretical value   :: 0.469707; 0.3894; 0.18394; 0.0526996
DLF value           :: 0.478846; 0.378842; 0.188375; 0.053272
Rel. error 5 pt DLF :: 1.9 %   ; 2.7 %   ; 2.4 %   ; 1.1 %
```

3. Difference between standard, lagged convolution, and splined DLF

Filter weights and the actual DLF are ignored in the explanation, we only look at the required data points in the input domain given our desired points in the output domain.

General parameters

```
# Wanted points in the output domain (x-f or x-t domain)
d_out = np.array([0.2, 0.7, 5, 25, 100])

# Made-up filter base for explanatory purpose
base = np.array([1e-2, 1e-1, 1e0, 1e1, 1e2])
```

3.1 Standard DLF

For each point in the output domain you have to compute n points in the input domain, where n is the filter length.

Implementation in ‘‘empymod‘‘

This is the most precise one, as no interpolation is used, but generally the slowest one. It is the default method for the Hankel transform.

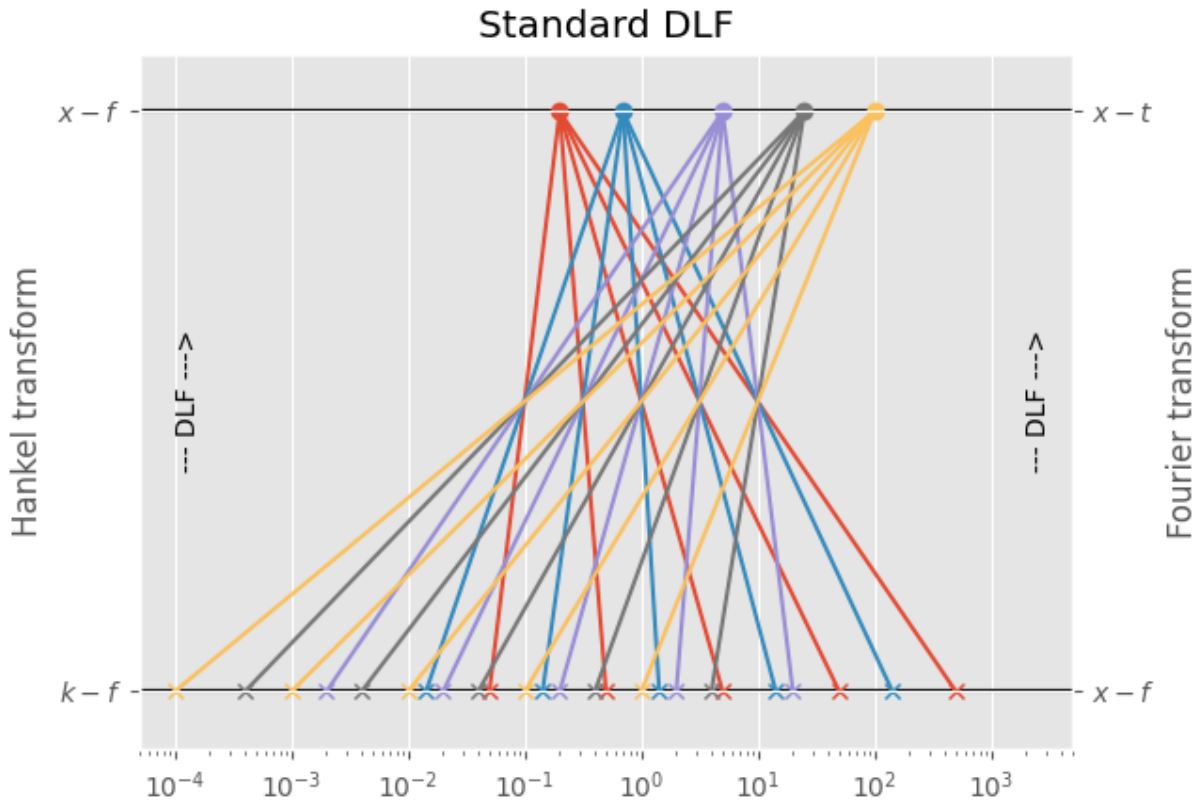
For the **Hankel transform**, use these parameters in `empymod.dipole` or `empymod.bipole` (from version `v1.6.0` onwards):

ht = 'dlf' # Default
htarg = {'pts_per_dec': 0} # Default

For the **Fourier transform**, use these parameters in `empymod.dipole` or `empymod.bipole` (from version `v1.6.0` onwards):

ft = 'sin' or 'cos' # Default is 'sin'
ftarg = {'pts_per_dec': 0}

```
# Required points in the input domain (k-f or x-f domain)
d_in = base/d_out[:, None]

# Print information
print('Points in output domain  ::', d_out.size)
print('Filter length            ::', base.size)
print('Req. pts in input domain ::', d_in.size)

# Figure
plt.figure()
plt.title('Standard DLF')
plt.hlines(1, 1e-5, 1e5)
plt.hlines(0, 1e-5, 1e5)

# Print scheme
for i, val in enumerate(d_out):
    for ii, ival in enumerate(d_in[i, :]):
        plt.plot(ival, 0, 'C'+str(i)+'x')
        plt.plot([ival, val], [0, 1], 'C'+str(i))
    plt.plot(val, 1, 'C'+str(i)+'o')

plt.xscale('log')
plt.xlim([5e-5, 5e3])

# Annotations
plt.text(2e3, 0.5, '--- DLF --->', rotation=90, va='center')
plt.text(1e-4, 0.5, '--- DLF --->', rotation=90, va='center')

plt.yticks([0, 1], (r'$k-f$', r'$x-f$'))
plt.ylabel('Hankel transform')
plt.ylim([-0.1, 1.1])

plt.gca().twinx()
plt.yticks([0, 1], (r'$x-f$', r'$x-t$'))
plt.ylabel('Fourier transform')
plt.ylim([-0.1, 1.1])

plt.show()
```



Out:

```
Points in output domain  :: 5
Filter length            :: 5
Req. pts in input domain :: 25
```

3.2 Lagged Convolution DLF

The spacing of the filter base is used to get from minimum to maximum required input-domain point (k in the case of the Hankel transform, f in the case of the Fourier transform); for each complete set the DLF is executed to compute the output-domain response (f in the case of the Hankel transform, t in the case of the Fourier transform), and interpolation is carried out in the output-domain.

Implementation in ‘‘empymod‘‘

This is usually the fastest option, and generally still more than sufficiently precise. It is the default method for the Fourier transform.

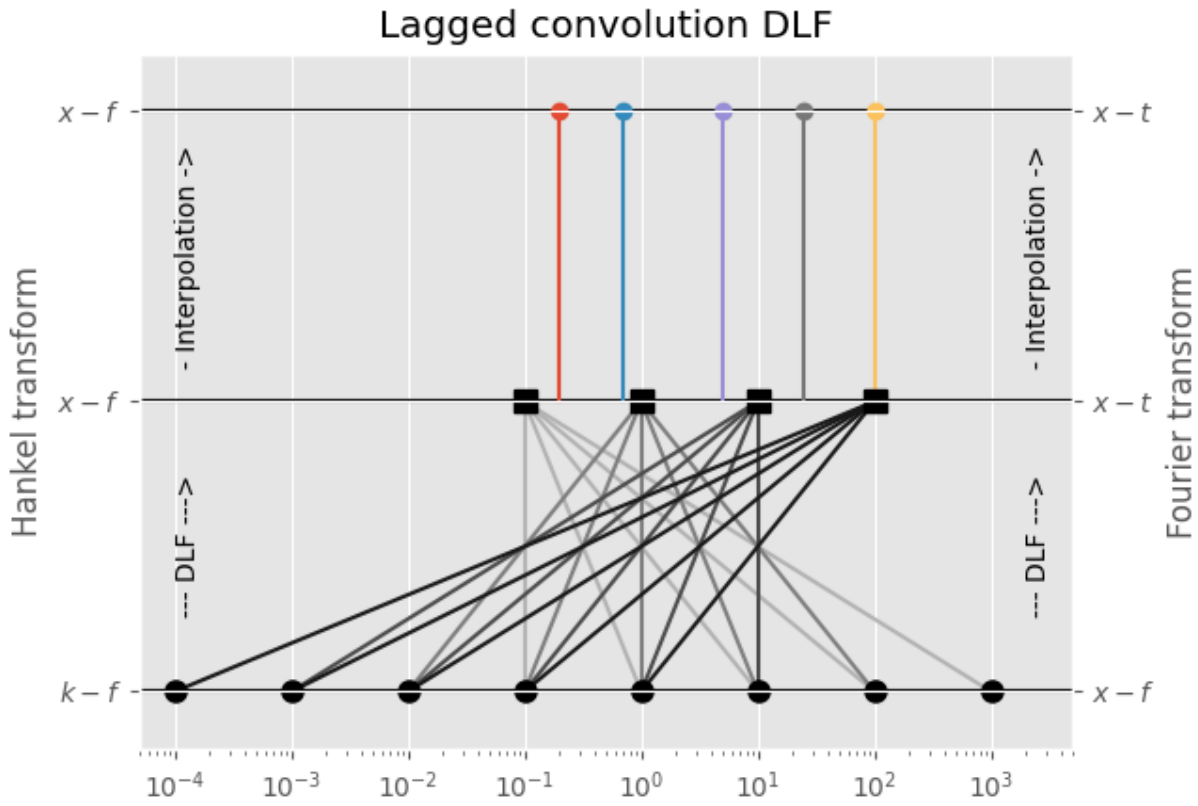
For the **Hankel transform**, use these parameters in `empymod.dipole` or `empymod.bipole` (from version `v1.6.0` onwards):

ht = ‘dlf’ # Default
htarg = {‘pts_per_dec’: int<0}

For the **Fourier transform**, use these parameters in `empymod.dipole` or `empymod.bipole` (from version `v1.6.0` onwards):

ft = 'sin' or 'cos' # Default is 'sin'
ftarg = {'pts_per_dec': int<0} # Default

```
# Required points in the k-domain
d_in2 = np.array([1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3])

# Intermediat values in the f-domain
d_out2 = np.array([1e-1, 1e0, 1e1, 1e2])

# Print information
print('Points in output-domain  ::', d_out.size)
print('Filter length            ::', base.size)
print('Req. pts in input-domain ::', d_in2.size)

# Figure
plt.figure()
plt.title('Lagged convolution DLF')
plt.hlines(1, 1e-5, 1e5)
plt.hlines(0, 1e-5, 1e5)
plt.hlines(0.5, 1e-5, 1e5)

# Plot scheme
for i, val in enumerate(d_out2):
    for ii in range(base.size):
        plt.plot([d_in2[-1-ii-i], val], [0, 0.5], str(0.7-0.2*i))

for iii, val2 in enumerate(d_out):
    plt.plot(val2, 1, 'C'+str(iii)+'o')
    plt.plot([val2, val2], [0.5, 1], 'C'+str(iii))

plt.plot(d_in2, d_in2*0, 'ko', ms=8)
plt.plot(d_out2, d_out2*0+0.5, 'ks', ms=8)
plt.xscale('log')
plt.xlim([5e-5, 5e3])

# Annotations
plt.text(2e3, 0.75, '- Interpolation ->', rotation=90, va='center')
plt.text(2e3, 0.25, '--- DLF --->', rotation=90, va='center')
plt.text(1e-4, 0.75, '- Interpolation ->', rotation=90, va='center')
plt.text(1e-4, 0.25, '--- DLF --->', rotation=90, va='center')

plt.yticks([0, 0.5, 1], (r'$k-f$', r'$x-f$', r'$x-f$'))
plt.ylabel('Hankel transform')
plt.ylim([-0.1, 1.1])

plt.gca().twinx()
plt.yticks([0, 0.5, 1], (r'$x-f$', r'$x-t$', r'$x-t$'))
plt.ylabel('Fourier transform')
plt.ylim([-0.1, 1.1])

plt.show()
```



Out:

```
Points in output-domain  :: 5
Filter length            :: 5
Req. pts in input-domain :: 8
```

3.3 Splined DLF

In the splined DLF m points per decade are used from minimum to maximum required input-domain point (k in the case of the Hankel transform, f in the case of the Fourier transform); then the required input-domain responses are interpolated in the input domain, and the DLF is exececuted subsequently.

Implementation in ‘‘empymod‘‘

This option can, at times, yield more precise results than the lagged convolution DLF, while being slower than the lagged convolution DLF but faster than the standard DLF. However, you have to carefully choose (or better, test) the parameter `pts_per_dec`.

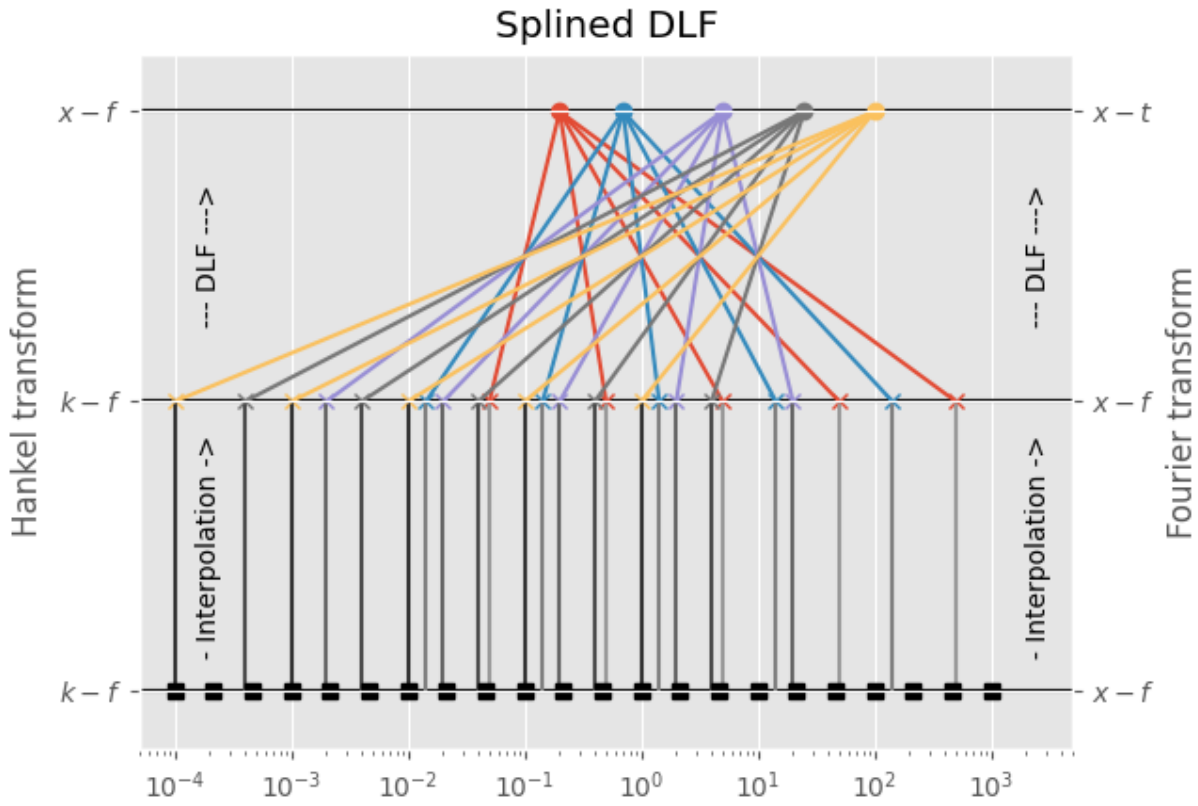
For the **Hankel transform**, use these parameters in `empymod.dipole` or `empymod.bipole` (from version `v1.6.0` onwards):

ht = ‘dlf’ # Default
htarg = {‘pts_per_dec’: int>0}

For the **Fourier transform**, use these parameters in `empymod.dipole` or `empymod.bipole` (from version `v1.6.0` onwards):

ft = 'sin' or 'cos' # Default is 'sin'
ftarg = {'pts_per_dec': int>0}

```
# Required points in the k-domain
d_in_min = np.log10(d_in).min()
d_in_max = np.ceil(np.log10(d_in).max())
pts_per_dec = 3
d_in2 = np.logspace(d_in_min, d_in_max, int((d_in_max-d_in_min)*pts_per_dec+1))

# Print information
print('Points in input-domain    ::', d_out.size)
print('Filter length             ::', base.size)
print('Points per decade         ::', pts_per_dec)
print('Req. pts in output-domain ::', d_in2.size)

# Figure
plt.figure()
plt.title('Splined DLF')
plt.hlines(1, 1e-5, 1e5)
plt.hlines(0.5, 1e-5, 1e5)
plt.hlines(0, 1e-5, 1e5)

# Plot scheme
for i, val in enumerate(d_out):
    for ii, ival in enumerate(d_in[i, :]):
        plt.plot(ival, 0.5, 'C'+str(i)+'x')
        plt.plot([ival, ival], [0, 0.5], str(0.6-0.1*i))
        plt.plot([ival, val], [0.5, 1], 'C'+str(i))
    plt.plot(val, 1, 'C'+str(i)+'o')

plt.plot(d_in2, d_in2*0, 'ks')
plt.xscale('log')
plt.xlim([5e-5, 5e3])

# Annotations
plt.text(2e3, 0.25, '- Interpolation ->', rotation=90, va='center')
plt.text(2e3, 0.75, '--- DLF --->', rotation=90, va='center')
plt.text(1.5e-4, 0.25, '- Interpolation ->', rotation=90, va='center')
plt.text(1.5e-4, 0.75, '--- DLF --->', rotation=90, va='center')

plt.yticks([0, 0.5, 1], (r'$k-f$', r'$k-f$', r'$x-f$'))
plt.ylabel('Hankel transform')
plt.ylim([-0.1, 1.1])

plt.gca().twinx()
plt.yticks([0, 0.5, 1], (r'$x-f$', r'$x-f$', r'$x-t$'))
plt.ylabel('Fourier transform')
plt.ylim([-0.1, 1.1])

plt.show()
```



Out:

```
Points in input-domain     :: 5
Filter length              :: 5
Points per decade          :: 3
Req. pts in output-domain :: 22
```

4. Example for the Hankel transform

The following is an example for the Hankel transform. **Be aware that the actual differences in time and accuracy depend highly on the model.** If time or accuracy is a critical issue in your computation I suggest to run some preliminary tests. It also depends heavily if you have many offsets, or many frequencies, or many layers, as one method might be better for many frequencies but few offsets, but the other method might be better for many offsets but few frequencies.

As general rules we can state that

- the longer the used filter is, or
- the more offsets you have

the higher is the time gain you get by using the lagged convolution or splined version of the DLF.

Here we compare the analytical halfspace solution to the numerical result, using the standard DLF, the lagged convolution DLF, and the splined DLF. Note the oscillating behaviour of the error of the lagged convolution and the splined versions, which comes from the interpolation and is not present in the standard version.

Define model, compute analytical solution

```
x = (np.arange(1, 1001))*10
params = {
```

(continues on next page)

(continued from previous page)

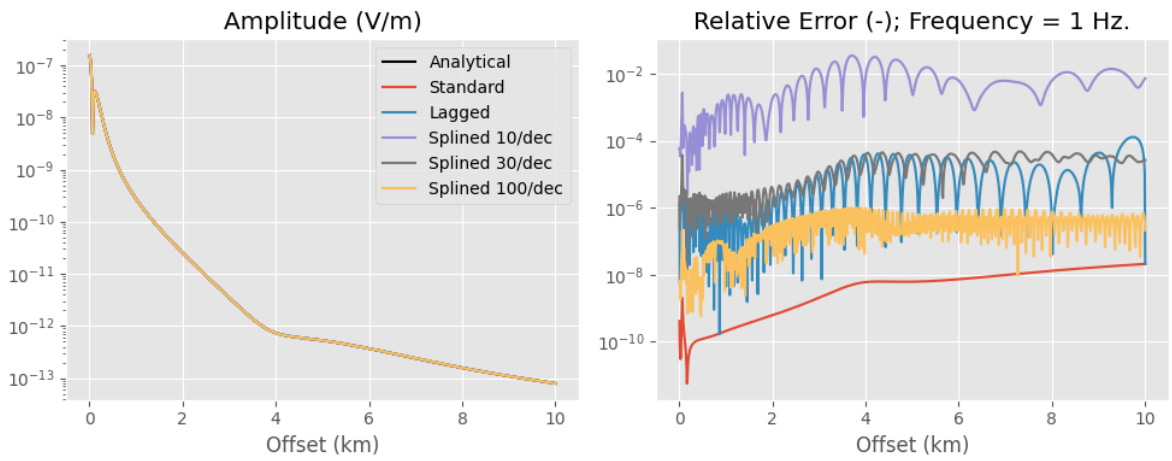
```
    'src': [0, 0, 150],
    'rec': [x, x*0, 200],
    'depth': 0,
    'res': [2e14, 1],
    'freqtime': 1,
    'ab': 11,
    'aniso': [1, 2],
    'xdirect': False,
    'verb': 0,
}

# Used Hankel filter
hfilt = empymod.filters.key_201_2009()

# Compute analytical solution
resp = empymod.analytical(
        params['src'], params['rec'], params['res'][1], params['freqtime'],
        solution='dhs', aniso=params['aniso'][1], ab=params['ab'],
        verb=params['verb']
)
```

Compute numerically the model using different Hankel options

```
standard = empymod.dipole(**params, htarg={'dlf': hfilt, 'pts_per_dec': 0})
laggedco = empymod.dipole(**params, htarg={'dlf': hfilt, 'pts_per_dec': -1})
spline10 = empymod.dipole(**params, htarg={'dlf': hfilt, 'pts_per_dec': 10})
spline30 = empymod.dipole(**params, htarg={'dlf': hfilt, 'pts_per_dec': 30})
splin100 = empymod.dipole(**params, htarg={'dlf': hfilt, 'pts_per_dec': 100})
```

Results

```
plt.figure(figsize=(10, 4))
plt.suptitle('Hankel transform example; frequency = ' +
             str(params['freqtime'])+' Hz', y=1.05, fontsize=15)

plt.subplot(121)
plt.title('Amplitude (V/m)')
plt.semilogy(x/1000, np.abs(resp), 'k', label='Analytical')
plt.semilogy(x/1000, np.abs(standard), label='Standard')
plt.semilogy(x/1000, np.abs(laggedco), label='Lagged')
plt.semilogy(x/1000, np.abs(spline10), label='Splined 10/dec')
plt.semilogy(x/1000, np.abs(spline30), label='Splined 30/dec')
plt.semilogy(x/1000, np.abs(splin100), label='Splined 100/dec')
plt.xlabel('Offset (km)')
plt.legend()

plt.subplot(122)
plt.title('Relative Error (-); Frequency = '+str(params['freqtime'])+' Hz.')
plt.semilogy(x/1000, np.abs((standard-resp)/resp), label='Standard')
plt.semilogy(x/1000, np.abs((laggedco-resp)/resp), label='Lagged')
plt.semilogy(x/1000, np.abs((spline10-resp)/resp), label='Splined 10/dec')
plt.semilogy(x/1000, np.abs((spline30-resp)/resp), label='Splined 30/dec')
plt.semilogy(x/1000, np.abs((splin100-resp)/resp), label='Splined 100/dec')
plt.xlabel('Offset (km)')

plt.tight_layout()
plt.show()
```



Runtimes and number of required wavenumbers for each method:

Hankel DLF Method	Time (ms)	# of wavenumbers
Standard	169	201000
Lagged Convolution	4	295
Splined 10/dec	95	96
Splined 30/dec	98	284
Splined 100/dec	111	944

So the lagged convolution has a relative error between roughly 1e-6 to 1e-4, hence 0.0001 % to 0.01 %, which is more then enough for real-world applications.

If you want to measure the runtime on your machine set `params['verb'] = 2`.

Note: If you use the splined version with about 500 samples per decade you get about the same accuracy as in the standard version. However, you get also about the same runtime.

5. Example for the Fourier transform

The same now for the Fourier transform. Obviously, when we use the Fourier transform we also use the Hankel transform. However, in this example we use only one offset. If there is only one offset then the lagged convolution or splined DLF for the Hankel transform do not make sense, and the standard is the fastest. So we use the standard Hankel DLF here. If you have many offsets then that would be different.

As general rules we can state that

- the longer the used filter is, or
- the more times you have

the higher is the time gain you get by using the lagged convolution or splined version of the DLF.

Define model, compute analytical solution

```
t = np.logspace(0, 2, 100)
xt = 2000

tparam = dc(params)
tparam['rec'] = [xt, 0, 200]
tparam['freqtime'] = t
tparam['signal'] = 0  # Impulse response
```

(continues on next page)

(continued from previous page)

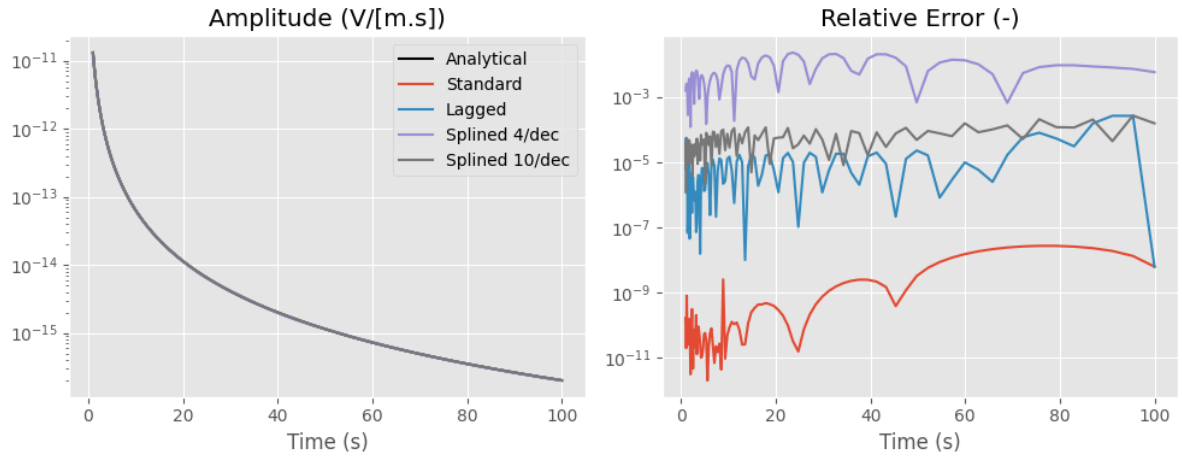
```
# Used Fourier filter
ffilt = empymod.filters.key_81_CosSin_2009()

# Compute analytical solution
tresp = empymod.analytical(
        tparam['src'], tparam['rec'], tparam['res'][1], tparam['freqtime'],
        signal=tparam['signal'], solution='dhs', aniso=tparam['aniso'][1],
        ab=tparam['ab'], verb=tparam['verb']
)
```

Compute numerically the model using different Fourier options

```
htarg = {'dlf': hfilt}

tstandard = empymod.dipole(
        **tparam, htarg=htarg, ftarg={'dlf': ffilt, 'pts_per_dec': 0})
tlaggedco = empymod.dipole(
        **tparam, htarg=htarg, ftarg={'dlf': ffilt, 'pts_per_dec': -1})
tsplined4 = empymod.dipole(
        **tparam, htarg=htarg, ftarg={'dlf': ffilt, 'pts_per_dec': 4})
tspline10 = empymod.dipole(
        **tparam, htarg=htarg, ftarg={'dlf': ffilt, 'pts_per_dec': 10})
```

Results

```
plt.figure(figsize=(10, 4))
plt.suptitle('Fourier transform example: impulse response; offset = ' +
             str(xt/1000)+' km', y=1.05, fontsize=15)

plt.subplot(121)
plt.title('Amplitude (V/[m.s])')
plt.semilogy(t, np.abs(tresp), 'k', label='Analytical')
plt.semilogy(t, np.abs(tstandard), label='Standard')
plt.semilogy(t, np.abs(tlaggedco), label='Lagged')
plt.semilogy(t, np.abs(tsplined4), label='Splined 4/dec')
plt.semilogy(t, np.abs(tspline10), label='Splined 10/dec')
plt.xlabel('Time (s)')
plt.legend()

plt.subplot(122)
plt.title('Relative Error (-)')
plt.semilogy(t, np.abs((tstandard-tresp)/tresp), label='Standard')
plt.semilogy(t, np.abs((tlaggedco-tresp)/tresp), label='Lagged')
plt.semilogy(t, np.abs((tsplined4-tresp)/tresp), label='Splined 4/dec')
plt.semilogy(t, np.abs((tspline10-tresp)/tresp), label='Splined 10/dec')
plt.xlabel('Time (s)')

plt.tight_layout()
plt.show()
```



Runtimes and number of required frequencies for each method:

Fourier DLF Method	Time (ms)	# of frequencies
Standard	1442	8100
Lagged Convolution	17	105
Splined 4/dec	14	37
Splined 10/dec	32	91

All methods require 201 wavenumbers (1 offset, filter length is 201).

So the lagged convolution has a relative error of roughly 1e-5, hence 0.001 %, which is more then enough for real-world applications.

If you want to measure the runtime on your machine set `tparam['verb'] = 2`.

```
empymod.Report()
```

Total running time of the script: (0 minutes 12.531 seconds)

Estimated memory usage: 331 MB

5.4.8 Published results

A lot of examples can be found in the notebooks which belong to published results. Here an overview where you can find those.

Introduction to Controlled-Source Electromagnetic Methods

See the notebooks in the repo empymod/csem-ziolkowski-and-slob for:

> Ziolkowski, A., and E. Slob, 2019, **Introduction to Controlled-Source Electromagnetic Methods**: Cambridge University Press; ISBN 9781107058620.

The notebooks contain all numerical examples of the book, 61 in total, frequency- and time-domain.

An open-source full 3D electromagnetic modeler for 1D VTI media in Python: empymod

See the notebooks in the repo empymod/article-geo2017 for:

> Werthmüller, D., 2017, **An open-source full 3D electromagnetic modeler for 1D VTI media in Python: empymod**: Geophysics, 82(6), WB9-WB19; DOI: 10.1190/geo2016-0626.1.

The notebooks contain:

- Comparison to analytical solution
- DLF filter comparison
- GPR example
- Time-domain example

A tool for designing digital filters for the Hankel and Fourier transforms in potential, diffusive, and wavefield modeling

See the notebooks in the repo empymod/article-fdesign for:

> Werthmüller, D., K. Key, and E. Slob, 2019, **A tool for designing digital filters for the Hankel and Fourier transforms in potential, diffusive, and wavefield modeling**: Geophysics, 84(2), F47-F56; DOI: 10.1190/geo2018-0069.1.

The notebooks contain:

- Examples regarding the design of digital filters for the Hankel and Fourier transforms
- GPR example with a digital filter

Getting started with controlled-source electromagnetic 1D modeling

See the notebooks in the repo empymod/article-tle2017 for:

> Werthmüller, D., 2017, **Getting started with controlled-source electromagnetic 1D modeling**: The Leading Edge, 36, 352-355; doi: 10.1190/tle36040352.1.

Contains 2 notebooks with eight frequency- and time-domain examples.

5.5 References

5.6 Credits

This project was initiated and is still mainly maintained by **Dieter Werthmüller**. However, it is a community effort as many people helped to improve it over time. The copyright is therefore attributed to «The empymod Developers», which includes everyone involved in the project.

Code contributors:

- Dieter Werthmüller

There are various other ways how people have contributed. In the following some particular credits.

Special thanks to

- **Jürg Hunziker**, **Kerry Key**, and **Evert Slob** for answering all my questions regarding their codes and publications (Hunziker et al., 2015, Key, 2009, Key, 2012, Slob et al., 2010).
- **Evert Slob** for the feedback and interaction during the creation of the add-on `tmtemod`, which was developed for the creation of github.com/empymod/csem-ziolkowski-and-slob.
- **Kerry Key** and **Evert Slob** for their inputs and feedback during the development of the add-on `fdesign` (see github.com/empymod/article-fdesign).
- **The electromagnetic group** of Aleksander Mousatov at the Mexican Petroleum Institute, Mexico City (see note below).

Many more helped through their feedback, feature request, bug reports etc. A certainly incomplete list (get in touch if you think you should appear here):

- Seogi Kang

- Svein-Erik Hamran
- Peter van der Sman
- Ralph-Uwe Börner
- Leon Foks

Note: This software was initially (till 01/2017) developed with funding from *The Mexican National Council of Science and Technology* (*Consejo Nacional de Ciencia y Tecnología*, https://www.conacyt.gob.mx), carried out at *The Mexican Institute of Petroleum IMP* (*Instituto Mexicano del Petróleo*, https://www.gob.mx/imp).

5.7 Changelog

5.7.1 Version 2

v2.0.x

v2.0.0: Numba

2020-04-29

This version is backwards incompatible and requires Python 3.6+.

- Numba:
  - Using `numexpr` is no longer a possibility. Instead, `numba` is a new dependency. All four kernel routines (`wavenumber`, `greenfct`, `reflections`, and `fields`) are now numba-jitted functions.
- Removed:
  - Removed all deprecated functions.
  - Dropped support for Python 3.5; moved to f-strings.
  - Dropped testing for channel conda-forge. The problems encountered at the early development cycle of empymod with conda-forge do not exist any longer.
- New defaults:
  - `EMArray`: `.amp` and `.pha` are now methods, not properties. Phase takes three optional boolean parameters `deg=False`, `unwrap=True`, and `lag=True`, to get radians or degrees; unwrapped or not; and lag or lead defined phases.
  - The parameters `epermV` and `mpermV` are set to the values of `epermH` and `mpermH`, respectively, if not provided (hence assuming isotropic behaviour). Before they were set to ones if not provided.
- Renaming:
  - `transform.fht` -> `transform.hankel_dlf`
  - `transform.hqwe` -> `transform.hankel_qwe`
  - `transform.hquad` -> `transform.hankel_quad`
  - `transform.ffht` -> `transform.fourier_dlf`
  - `transform.fqwe` -> `transform.fourier_qwe`
  - `transform.fftlog` -> `transform.fourier_fftlog`
  - `transform.fft` -> `transform.fourier_fft`
  - `transform.fhti` -> `transform.get_fftlog_input`
  - `transform.get_spline_values` -> `transform.get_dlf_points`.

  - `factAng` -> `ang_fact`
  - In `htarg`-dict: `fftfilt`-> `dlf` (filter name for Hankel-DLF)
  - In `ftarg`-dict: `fhtfilt`-> `dlf` (filter name for Fourier-DLF)
  - In `ftarg`-dict: `ft`-> `kind` (method in Fourier-DLF [sine/cosine])
  - Only dictionaries allowed for `htarg` and `ftarg`; strings, lists, or tuples are not allowed any longer. They are also dictionaries internally now.
  - `ht`: There is only one unique name for each method: 'dlf', 'qwe', 'quad'.
  - `ft`: There is only one unique name for each method: 'dlf', 'qwe', 'fftlog', 'fft'.
  - Within `transform`, change `fhtarg`, `qweargs`, and `quadargs` to `htarg`; `qweargs` to `ftarg`.
- Other changes:
  - All settings (`xdirect`, `ht`, `htarg`, `ft`, `ftarg`, `loop`, `verb`) are now extracted from `kwargs`. This makes it possible that all `model`-functions take the same keyword-arguments; warnings are raised if a particular parameter is not used in this function, but it doesn't fail (it fails, however, for unknown parameters). Pure positional calls including those parameters will therefore not work any longer.
  - Undo a change introduced in v1.8.0: `get_dlf_points` is calculated directly within `transform.fht` [empymod#26].
  - Ensured that source and receiver inputs are not altered.
  - Significantly reduced top namespace; only functions from `model` are loaded into the top namespace now.

5.7.2 Version 1

v1.10.x

v1.10.6: Various azimuths and dips at same depth

2020-03-04

- `empymod.bipole`
  - In the source and receiver format `[x, y, z, azimuth, dip]`, azimuth and dip can now be either single values, or the same number as the other coordinates.
  - Bugfix (in `utils.get_abs`): When different orientations were used exactly along the principal axes, at the same depth, only the first source was calculated [empymod#74].

v1.10.5: Continuously in- or decreasing

2020-02-21

This is a small appendix to v1.10.4: Depths can now be defined in increasing or decreasing order, as long as they are consistent. Model parameters have to be defined in the same order. Hence all these are possible:

- `[-100, 0, 1000, 1050]` -> left-handed system, low-to-high
- `[100, 0, -1000, -1050]` -> right-handed system, high-to-low
- `[1050, 1000, 0, -100]` -> left-handed system, high-to-low
- `[-1050, -1000, 0, 100]` -> right-handed system, low-to-high

v1.10.4: Positive z down- or upwards

2020-02-16

- New examples:
  - `empymod` can handle positive z down- or upwards (left-handed or right-handed coordinate systems; it was always possible, but not known nor documented). Adjusted documentation, docstrings, and added an example.
  - Example how to calculate the responses for the WalkTEM system.
- Minor things and bug fixes:
  - Change from relative to absolute imports.
  - Simplified releasing (no badges).
  - Python 3.8 is tested.
  - Fix: numpy now throws an error if the third argument of `logspace` is not an `int`, some casting was therefore necessary within the code.

v1.10.3: Sphinx Gallery

2019-11-11

- Move examples to an integrated Sphinx-Gallery, generated each time.
- Move from conda-channel `prisae` to `conda-forge`.
- Automatic deploy for PyPi and conda-forge.

v1.10.2: Always EMArray

2019-11-06

- Simplified and improved `empymod.utils.EMArray`. Now every returned array from the main modelling routines `bipole`, `dipole`, `loop`, and `analytical` is an EMArray with `.amp`- and `.pha`-attributes.
- Theme and documentation reworked, to be more streamlined with `emg3d` (for easier long-term maintenance).
- 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.)
- Fixes to the `setuptools_scm`-implementation (`MANIFEST.in`).
- `ROADMAP.rst` moved to GitHub-Projects; `MAINTENANCE.rst` included in manual.

v1.10.1: setuptools_scm

2019-10-22

- Typos from v1.10.0; update example in `model.loop`.
- Implement `setuptools_scm` for versioning (adds git hashes for dev-versions).

v1.10.0: Loop source and receiver

2019-10-15

- New modelling routine `model.loop` to model the electromagnetic frequency- or time-domain field due to an arbitrary rotated, magnetic source consisting of an electric loop, measured by arbitrary rotated, finite electric or magnetic bipole receivers or arbitrary rotated magnetic receivers consisting of electric loops.
- Move copyright from «Dieter Werthmüller» to «The empymod Developers», to be more inclusive and open the project for new contributors.

v1.9.x

v1.9.0 : Laplace

2019-10-04

- Laplace-domain calculation: By providing a negative `freq`-value, the calculation is carried out in the real Laplace domain `s = freq` instead of the complex frequency domain `s = 2i*pi*freq`.
- Improvements to filter design and handling:
  - `DigitalFilter` now takes an argument (list of strings) for additional coefficients to the default `j0`, `j1`, `sin`, and `cos`.
  - `fdesign` can now be used with any name as attribute you want to describe the transform pair (until now it had to be either `j0`, `j1`, `j2`, `sin`, or `cos`).
  - The provided sine and cosine transform pairs in `fdesign` can now be asked to return the inverse pair (time to frequency).
- Other tiny improvements and bug fixes.

v1.8.x

v1.8.3 : Scooby

2019-07-05

- Use `scooby` for `Versions` (printinfo), change name to `Report`.
- DOC: Correct return statement if `mrec=True`.
- Typos and correct links for new asv/bench.
- Bump requirement to SciPy>=1.0.0, remove warning regarding memory leak in SciPy 0.19.0.

v1.8.2 : pts_per_dec for DLF are now floats

2019-04-26

- `pts_per_dec` are now floats, not integers, which gives more flexibility.
- Bugfix: `pts_per_dec` for DLF was actually points per `e`, not per decade, as the natural logarithm was used.
- New `Versions`-class; improvement over the `versions`-function, as it automatically detects if it can print html or not.
- Maintenance: Update `np.load` in tests with `allow_pickle=True` for changes in numpy v1.16.3.
- Lots of changes to accommodate `emg3d` within the `empymod`-org:
  - Adjust website, move stuff from website into README.md.

  - /empymod/example-notebooks -> /empymod/empymod-examples.
  - /empymod/asv -> /empymod/empymod-asv (and therefore now available at empymod.github.io/empymod-asv).
  - /empymod/bench -> /empymod/empymod-bench.
- Move manual from `empymod/__init__.py` to the `docs/manual.rst`, and the references to its own file. Change reference style.
- Move credits for initial funding from the license-section of the manual to CREDITS.rst, where it belongs.

v1.8.1 : Version of Filter-article and CSEM-book

2018-11-20

- Many little improvements in the documentation.
- Some code improvements through the use of codacy.
- Remove testing of Python 3.4; officially supported are now Python 3.5-3.7.
- Version of the filter article (DLF) in geophysics and of the CSEM book.

v1.8.0 : Hook for Cole-Cole IP and similar

2018-10-26

- `model.bipole`, `model.dipole`, and `model.analytical` have now a hook which users can exploit to insert their own calculation of `etaH`, `etaV`, `zetaH`, and `zetaV`. This can be used, for instance, to model a Cole-Cole IP survey. See the manual or the example-notebooks for more information.
- `model.wavenumber` renamed to `model.dipole_k` to avoid name clash with `kernel.wavenumber`. For now `model.wavenumber` continues to exist, but raises a deprecation warning.
- `xdirect` default value changed from `True` to `False`.
- Possibility to provide interpolated points (`int_pts`) to `transform.dlf`.

The following changes are backwards incompatible if you directly used `transform.fht`, `transform.hqwe`, or `transform.hquad`. Nothing changes for the user-facing routines in `model`:

- `empymod.fem` now passes `factAng` to `empymod.transform`, not `angle`; this saves some time if looped over offsets or frequencies, as it is not repeatedly calculated within `empymod.transform`.
- Use `get_spline_values` in `empymod.fem` for Hankel DLF, instead of in `empymod.fht`. Gives a speed-up if looped over offsets or frequencies. Should be in `utils`, but that would be heavily backwards incompatible. Move there in version 2.0.

v1.7.x

v1.7.3 : Speed improvements following benchmarks

2018-07-16

- Small improvements related to speed as a result of the benchmarks introduced in v1.7.2:
  - Kernels which do not exist for a given `ab` are now returned as `None` from `kernel.wavenumber` instead of arrays of zeroes. This permits for some time saving in the transforms. This change is backwards incompatible if you directly used `kernel.wavenumber`. Nothing changes for the user-facing routines in `model`.

  - Adjustments in `transform` with regard to the `None` returned by `kernel.wavenumber`. The kernels are not checked anymore if they are all zeroes (which can be slow for big arrays). If they are not None, they will be processed.
  - Various small improvements for speed to `transform.dlf` (i.e. `factAng`; `log10`/`log`; re-arranging).

v1.7.2 : Benchmarked with asv

2018-07-07

- Benchmarks: `empymod` has now a benchmark suite, see empymod/asv.
- Fixed a bug in `bipole` for time-domain responses with several receivers or sources with different depths. (Simply failed, as wrong dimension was provided to `tem`).
- Small improvements:
  - Various simplifications or cleaning of the code base.
  - Small change (for speed) in check if kernels are empty in `transform.dlf` and `transform.qwe`.

v1.7.1 : Load/save filters in plain text

2018-06-19

- New routines in `empymod.filters.DigitalFilter`: Filters can now be saved to or loaded from pure ascii-files.
- Filters and inversion result from `empymod.scripts.fdesign` are now by default saved in plain text. The filters with their internal routine, the inversion result with `np.savetxt`. Compressed saving can be achieved by giving a name with a '.gz'-ending.
- Change in `empymod.utils`:
  - Renamed `_min_param` to `_min_res`.
  - Anisotropy `aniso` is no longer directly checked for its minimum value. Instead, res*aniso**2, hence vertical resistivity, is checked with `_min_res`, and anisotropy is subsequently re-calculated from it.
  - The parameters `epermH`, `epermV`, `mpermH`, and `mpermV` can now be set to 0 (or any positive value) and do not depend on `_min_param`.
- `printinfo`: Generally improved; prints now MKL-info (if available) independently of `numexpr`.
- Simplification of `kernel.reflections` through re-arranging.
- Bug fixes
- Version of re-submission of the DLF article to geophysics.

v1.7.0 : Move empyscripts into empymod.scripts

2018-05-23

Merge `empyscripts` into `empymod` under `empymod.scripts`.

- Clear separation between mandatory and optional imports:
  - Mandatory:
    * `numpy`
    * `scipy`
  - Optional:

    * `numexpr` (for `empymod.kernel`)
    * `matplotlib` (for `empymod.scripts.fdesign`)
    * `IPython` (for `empymod.scripts.printinfo`)

- Broaden namespace of `empymod`. All public functions from the various modules and the modules from `empymod.scripts` are now available under `empymod` directly.

v1.6.x

v1.6.2 : Speed improvements for QUAD/QWE

2018-05-21

These changes should make calculations using `QWE` and `QUAD` for the Hankel transform for cases which do not require all kernels faster; sometimes as much as twice as fast. However, it might make calculations which do require all kernels a tad slower, as more checks had to be included. (Related to [empymod#11]; basically including for `QWE` and `QUAD` what was included for `DLF` in version 1.6.0.)

- `transform`:
  - `dlf`:
    * Improved by avoiding unnecessary multiplications/summations for empty kernels and applying the angle factor only if it is not 1.
    * Empty/unused kernels can now be input as `None`, e.g. `signal=(PJ0, None, None)`.
    * `factAng` is new optional for the Hankel transform, as is `ab`.
  - `hqwe`: Avoids unnecessary calculations for zero kernels, improving speed for these cases.
  - `hquad`, `quad`: Avoids unnecessary calculations for zero kernels, improving speed for these cases.
- `kernel`:
  - Simplify `wavenumber`
  - Simplify `angle_factor`

v1.6.1 : Primary/secondary field

2018-05-05

Secondary field calculation.

- Add the possibility to calculate secondary fields only (excluding the direct field) by passing the argument `xdirect=None`. The complete `xdirect`-signature is now (only affects calculation if src and rec are in the same layer):
  - If True, direct field is calculated analytically in the frequency domain.
  - If False, direct field is calculated in the wavenumber domain.
  - If None, direct field is excluded from the calculation, and only reflected fields are returned (secondary field).
- Bugfix in `model.analytical` for `ab=[36, 63]` (zeroes) [empymod#16].

v1.6.0 : More DLF improvements

2018-05-01

This release is not completely backwards compatible for the main modelling routines in `empymod.model`, but almost. Read below to see which functions are affected.

- Improved Hankel DLF [empymod#11]. `empymod.kernel.wavenumber` always returns three kernels, `PJ0`, `PJ1`, and `PJ0b`. The first one is angle-independent, the latter two depend on the angle. Now, depending of what source-receiver configuration is chosen, some of these might be zero. If-statements were now included to avoid the calculation of the DLF, interpolation, and reshaping for 0-kernels, which improves speed for these cases.
- Unified DLF arguments [empymod#10].

  These changes are backwards compatible for all main modelling routines in `empymod.model`. However, they are not backwards compatible for the following routines:

  - `empymod.model.fem` (removed `use_spline`),
  - `empymod.transform.fht` (removed `use_spline`),
  - `empymod.transform.hqwe` (removed `use_spline`),
  - `empymod.transform.quad` (removed `use_spline`),
  - `empymod.transform.dlf` (`lagged`, `splined` => `pts_per_dec`),
  - `empymod.utils.check_opt` (no longer returns `use_spline`),
  - `empymod.utils.check_hankel` (changes in `pts_per_dec`), and
  - `empymod.utils.check_time` (changes in `pts_per_dec`).

  The function `empymod.utils.spline_backwards_hankel` can be used for backwards compatibility.

  Now the Hankel and Fourier DLF have the same behaviour for `pts_per_dec`:

  - `pts_per_dec = 0`: Standard DLF,
  - `pts_per_dec < 0`: Lagged Convolution DLF, and
  - `pts_per_dec > 0`: Splined DLF.

  There is one exception which is not backwards compatible: Before, if `opt=None` and `htarg={pts_per_dec: != 0}`, the `pts_per_dec` was not used for the FHT and the QWE. New, this will be used according to the above definitions.
- Bugfix in `model.wavenumber` for `ab=[36, 63]` (zeroes).

v1.5.x

v1.5.2 : Improved DLF

2018-04-25

- DLF improvements:
  - Digital linear filter (DLF) method for the Fourier transform can now be carried out without spline, providing 0 for `pts_per_dec` (or any integer smaller than 1).
  - Combine kernel from `fht` and `ffht` into `dlf`, hence separate DLF from other calculations, as is done with QWE (`qwe` for `hqwe` and `fqwe`).
  - Bug fix regarding `transform.get_spline_values`; a DLF with `pts_per_dec` can now be shorter then the corresponding filter.

v1.5.1 : Improved docs

2018-02-24

- Documentation:

  - Simplifications: avoid duplication as much as possible between the website (empymod.github.io), the manual (empymod.readthedocs.io), and the `README` (github.com/empymod/empymod).
    * Website has now only *Features* and *Installation* in full, all other information comes in the form of links.
    * `README` has only information in the form of links.
    * Manual contains the `README`, and is basically the main document for all information.
  - Improvements: Change some remaining `md`-syntax to `rst`-syntax.
  - FHT -> DLF: replace FHT as much as possible, without breaking backwards compatibility.

v1.5.0 : Hankel filter wer_201_2018

2018-01-02

- Minimum parameter values can now be set and verified with `utils.set_minimum` and `utils.get_minimum`.
- New Hankel filter `wer_201_2018`.
- `opt=parallel` has no effect if `numexpr` is not built against Intel's VML. (Use `import numexpr; numexpr.use_vml` to see if your `numexpr` uses VML.)
- Bug fixes
- Version of manuscript submission to geophysics for the DLF article.

v1.4.x

v1.4.4 : TE/TM split

2017-09-18

[This was meant to be 1.4.3, but due to a setup/pypi/anaconda-issue I had to push it to 1.4.4; so there isn't really a version 1.4.3.]

- Add TE/TM split to diffusive ee-halfspace solution.
- Improve `kernel.wavenumber` for fullspaces.
- Extended `fQWE` and `fftlog` to be able to use the cosine-transform. Now the cosine-transform with the real-part frequency response is used internally if a switch-off response (`signal=-1`) is required, rather than calculating the switch-on response (with sine-transform and imaginary-part frequency response) and subtracting it from the DC value.
- Bug fixes

v1.4.2 : Final submission version of Geophysics paper

2017-06-04

- Bugfix: Fixed squeeze in `model.analytical` with `solution='dsplit'`.
- Version of final submission of manuscript to Geophysics.

v1.4.1 : Own organisation github.com/empymod

2017-05-30

[This was meant to be 1.4.0, but due to a setup/pypi/anaconda-issue I had to push it to 1.4.1; so there isn't really a version 1.4.0.]

- New home: empymod.github.io as entry point, and the project page on github.com/empymod. All empymod-repos moved to the new home.
  - /prisae/empymod -> /empymod/empymod
  - /prisae/empymod-notebooks -> /empymod/example-notebooks
  - /prisae/empymod-geo2017 -> /empymod/article-geo2017
  - /prisae/empymod-tle2017 -> /empymod/article-tle2017
- Modelling routines:
  - New modelling routine `model.analytical`, which serves as a front-end to `kernel.fullspace` or `kernel.halfspace`.
  - Remove legacy routines `model.time` and `model.frequency`. They are covered perfectly by `model.dipole`.
  - Improved switch-off response (calculate and subtract from DC).
  - `xdirect` adjustments:
    - `isfullspace` now respects `xdirect`.
    - Removed `xdirect` from `model.wavenumber` (set to `False`).
- Kernel:
  - Modify `kernel.halfspace` to use same input as other kernel functions.
  - Include time-domain ee halfspace solution into `kernel.halfspace`; possible to obtain direct, reflected, and airwave separately, as well as only fullspace solution (all for the diffusive approximation).

v1.3.x

v1.3.0 : New transforms QUAD (Hankel) and FFT (Fourier)

2017-03-30

- Add additional transforms and improve QWE:
  - Conventional adaptive quadrature (QUADPACK) for the Hankel transform;
  - Conventional FFT for the Fourier transform.
  - Add `diff_quad` to `htarg`/`ftarg` of QWE, a switch parameter for QWE/QUAD.
  - Change QWE/QUAD switch from comparing first interval to comparing all intervals.
  - Add parameters for QUAD (a, b, limit) into `htarg`/`ftarg` for QWE.
- Allow `htarg`/`ftarg` as dict additionally to list/tuple.
- Improve `model.gpr`.
- Internal changes:
  - Rename internally the sine/cosine filter from `fft` to `ffht`, because of the addition of the Fast Fourier Transform `fft`.
- Clean-up repository
  - Move `notebooks` to /prisae/empymod-notebooks

  - Move `publications/Geophysics2017` to /prisae/empymod-geo2017
  - Move `publications/TheLeadingEdge2017` to /prisae/empymod-tle2017
- Bug fixes and documentation improvements

v1.2.x

v1.2.1 : Installable via pip and conda

2017-03-11

- Change default filter from `key_401_2009` to `key_201_2009` (because of warning regarding 401 pt filter in source code of `DIPOLE1D`.)
- Since 06/02/2017 installable via pip/conda.
- Bug fixes

v1.2.0 : Bipole

2017-02-02

- New routine:
  - General modelling routine `bipole` (replaces `srcbipole`): Model the EM field for arbitrarily oriented, finite length bipole sources and receivers.
- Added a test suite:
  - Unit-tests of small functions.
  - Framework-tests of the bigger functions:
    * Comparing to status quo (regression tests),
    * Comparing to known analytical solutions,
    * Comparing different options to each other,
    * Comparing to other 1D modellers (EMmod, DIPOLE1D, GREEN3D).
  - Incorporated with Travis CI and Coveralls.
- Internal changes:
  - Add kernel count (printed if verb > 1).
  - `numexpr` is now only required if `opt=='parallel'`. If `numexpr` is not found, `opt` is reset to `None` and a warning is printed.
  - Cleaned-up wavenumber-domain routine.
  - theta/phi -> azimuth/dip; easier to understand.
  - Refined verbosity levels.
  - Lots of changes in `utils`, with regards to the new routine `bipole` and with regards to verbosity. Moved all warnings out from `transform` and `model` into `utils`.
- Bug fixes

v1.1.x

v1.1.0 : Include source bipole

2016-12-22

- New routines:
  - New `srcbipole` modelling routine: Model an arbitrarily oriented, finite length bipole source.
  - Merge `frequency` and `time` into `dipole`. (`frequency` and `time` are still available.)
  - `dipole` now supports multiple sources.
- Internal changes:
  - Replace `get_Gauss_Weights` with `scipy.special.p_roots`
  - `jv(0,x)`, `jv(1,x)` -> `j0(x)`, `j1(x)`
  - Replace `param_shape` in `utils` with `_check_var` and `_check_shape`.
  - Replace `xco` and `yco` by `angle` in `kernel.fullspace`
  - Replace `fftlog` with python version.
  - Additional sine-/cosine-filters: `key_81_CosSin_2009`, `key_241_CosSin_2009`, and `key_601_CosSin_2009`.
- Bug fixes

v1.0.x

v1.0.0 : Initial release

2016-11-29

- Initial release; state of manuscript submission to geophysics.

5.8 Maintainers Guide

5.8.1 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;
- empymod.rtfd.io created a tagged version.

5.8.2 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 manually 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/ empymod
```

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

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

5.8.3 CI

- Testing on Travis, includes:
  - Tests using `pytest`
  - Linting / code style with `pytest-flake8`
  - Figures with `pytest-mpl`
  - Ensure all http(s)-links work (`sphinx linkcheck`)
- Line-coverage with `pytest-cov` on Coveralls
- Code-quality on Codacy
- Manual on ReadTheDocs, including the Gallery (examples run each time).
- DOI minting on Zenodo
- Benchmarks with Airspeed Velocity (`asv`) [currently manually]
- Automatically deploys if tagged:
  - PyPi
  - conda -c conda-forge

5.9 Main modelling routines

Electromagnetic modeller to model electric or magnetic responses due to a three-dimensional electric or magnetic source in a layered-earth model with vertical transverse isotropic (VTI) resistivity, VTI electric permittivity, and VTI magnetic permeability, from very low frequencies (DC) to very high frequencies (GPR). The calculation is carried out in the wavenumber-frequency domain, and various Hankel- and Fourier-transform methods are included to transform the responses into the space-frequency and space-time domains.

5.9.1 `empymod.model` – Model EM-responses

EM-modelling routines. The implemented routines might not be the fastest solution to your specific problem. Use these routines as template to create your own, problem-specific modelling routine!

Principal routines:

- *bipole()*

- *dipole()*
- *loop()*

The main routine is *bipole()*, which can model bipole source(s) and bipole receiver(s) of arbitrary direction, for electric or magnetic sources and receivers, both in frequency and in time. A subset of *bipole()* is *dipole()*, which models infinitesimal small dipoles along the principal axes x, y, and z. The third routine, *loop()*, can be used if the source or the receivers are loops instead of dipoles.

Further routines are:

- *analytical()*: Calculate analytical fullspace and halfspace solutions.
- *dipole_k()*: Calculate the electromagnetic wavenumber-domain solution.
- *gpr()*: Calculate the Ground-Penetrating Radar (GPR) response.

The *dipole_k()* routine can be used if you are interested in the wavenumber-domain result, without Hankel nor Fourier transform. It calls straight the *empymod.kernel*. The *gpr()*-routine convolves the frequency-domain result with a wavelet, and applies a gain to the time-domain result. This function is still experimental.

The modelling routines make use of the following two core routines:

- *fem()*: Calculate wavenumber-domain electromagnetic field and carry out the Hankel transform to the frequency domain.
- *tem()*: Carry out the Fourier transform to time domain after *fem()*.

empymod.model.**bipole**(*src*, *rec*, *depth*, *res*, *freqtime*, *signal=None*, *aniso=None*, *epermH=None*, *epermV=None*, *mpermH=None*, *mpermV=None*, *msrc=False*, *srcpts=1*, *mrec=False*, *recpts=1*, *strength=0*, ***kwargs*)

Return EM fields due to arbitrary rotated, finite length EM dipoles.

Calculate the electromagnetic frequency- or time-domain field due to arbitrary rotated, finite electric or magnetic bipole sources, measured by arbitrary rotated, finite electric or magnetic bipole receivers. By default, the electromagnetic response is normalized to source and receiver of 1 m length, and source strength of 1 A.

Parameters

src, rec [list of floats or arrays] Source and receiver coordinates (m):

- [x0, x1, y0, y1, z0, z1] (bipole of finite length)
- [x, y, z, azimuth, dip] (dipole, infinitesimal small)

Dimensions:

- The coordinates x, y, and z (dipole) or x0, x1, y0, y1, z0, and z1 (bipole) can be single values or arrays.
- The variables x and y (dipole) or x0, x1, y0, and y1 (bipole) must have the same dimensions.
- The variables z, azimuth, and dip (dipole) or z0 and z1 (bipole) must either be single values or having the same dimension as the other coordinates.

Angles (coordinate system is either left-handed with positive z down or right-handed with positive z up; East-North-Depth):

- azimuth (°): horizontal deviation from x-axis, anti-clockwise.
- +/-dip (°): vertical deviation from xy-plane down/up-wards.

Sources or receivers placed on a layer interface are considered in the upper layer.

depth [list] Absolute layer interfaces z (m); #depth = #res - 1 (excluding +/- infinity).

res [array_like] Horizontal resistivities rho_h (Ohm.m); #res = #depth + 1.

Alternatively, res can be a dictionary. See the main manual of empymod too see how to exploit this hook to re-calculate etaH, etaV, zetaH, and zetaV, which can be used to, for instance, use the Cole-Cole model for IP.

freqtime [array_like] Frequencies f (Hz) if *signal*==*None*, else times t (s); (f, t > 0).

signal [{None, 0, 1, -1}, optional] Source signal, default is None:

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

aniso [array_like, optional] Anisotropies lambda = sqrt(rho_v/rho_h) (-); #aniso = #res. Defaults to ones.

epermH, epermV [array_like, optional] Relative horizontal/vertical electric permittivities epsilon_h/epsilon_v (-); #epermH = #epermV = #res. If epermH is provided but not epermV, isotropic behaviour is assumed. Default is ones.

mpermH, mpermV [array_like, optional] Relative horizontal/vertical magnetic permeabilities mu_h/mu_v (-); #mpermH = #mpermV = #res. If mpermH is provided but not mpermV, isotropic behaviour is assumed. Default is ones.

msrc, mrec [bool, optional] If True, source/receiver (msrc/mrec) is magnetic, else electric. Default is False.

srcpts, recpts [int, optional] Number of integration points for bipole source/receiver, default is 1:

- srcpts/recpts < 3 : bipole, but calculated as dipole at centre
- srcpts/recpts >= 3 : bipole

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

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

Default is 0.

verb [{0, 1, 2, 3, 4}, optional] Level of verbosity, default is 2:

- 0: Print nothing.
- 1: Print warnings.
- 2: Print additional runtime and kernel calls
- 3: Print additional start/stop, condensed parameter information.
- 4: Print additional full parameter information

ht [{'dlf', 'qwe', 'quad'}, optional] Flag to choose either the *Digital Linear Filter* (DLF) method, the *Quadrature-With-Extrapolation* (QWE), or a simple *Quadrature* (QUAD) for the Hankel transform. Defaults to 'dlf'.

htarg [dict, optional] Possible parameters depends on the value for *ht*:

- If *ht='dlf'*:
  - *dlf*: string of filter name in `empymod.filters` or the filter method itself. (default: `empymod.filters.key_201_2009()`)
  - *pts_per_dec*: points per decade; (default: 0):
    - If 0: Standard DLF.

        * If < 0: Lagged Convolution DLF.
        * If > 0: Splined DLF
    - If *ht='qwe'*:
      - *rtol*: relative tolerance (default: 1e-12)
      - *atol*: absolute tolerance (default: 1e-30)
      - *nquad*: order of Gaussian quadrature (default: 51)
      - *maxint*: maximum number of partial integral intervals (default: 40)
      - *pts_per_dec*: points per decade; (default: 0)
        * If 0, no interpolation is used.
        * If > 0, interpolation is used.
      - *diff_quad*: criteria when to swap to QUAD (only relevant if pts_per_dec=-1) (default: 100)
      - *a*: lower limit for QUAD (default: first interval from QWE)
      - *b*: upper limit for QUAD (default: last interval from QWE)
      - *limit*: limit for quad (default: maxint)
    - If *ht='quad'*:
      - *rtol*: relative tolerance (default: 1e-12)
      - *atol*: absolute tolerance (default: 1e-20)
      - *limit*: An upper bound on the number of subintervals used in the adaptive algorithm (default: 500)
      - *lmin*: Minimum wavenumber (default 1e-6)
      - *lmax*: Maximum wavenumber (default 0.1)
      - *pts_per_dec*: points per decade (default: 40)

ft [{'dlf', 'sin', 'cos', 'qwe', 'fftlog', 'fft'}, optional] Only used if signal!=None. Flag to choose either the Digital Linear Filter method (Sine- or Cosine-Filter), the Quadrature-With-Extrapolation (QWE), the FFTLog, or the FFT for the Fourier transform. Defaults to 'dlf' (which is 'sin' if signal>=0, else 'cos').

ftarg [dict, optional] Only used if signal!=None. Possible parameters depends on the value for *ft*:

- If *ft='dlf'*, 'sin', or 'cos':
  - *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='qwe'*:
  - *rtol*: relative tolerance (default: 1e-8)
  - *atol*: absolute tolerance (default: 1e-20)
  - *nquad*: order of Gaussian quadrature (default: 21)
  - *maxint*: maximum number of partial integral intervals (default: 200)

- *pts_per_dec*: points per decade (default: 20)
- *diff_quad*: criteria when to swap to QUAD (default: 100)
- *a*: lower limit for QUAD (default: first interval from QWE)
- *b*: upper limit for QUAD (default: last interval from QWE)
- *limit*: limit for quad (default: maxint)

- 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
- If *ft='fft'*:
  - *dfreq*: Linear step-size of frequencies (default: 0.002)
  - *nfreq*: Number of frequencies (default: 2048)
  - *ntot*: Total number for FFT; difference between nfreq and ntot is padded with zeroes. This number is ideally a power of 2, e.g. 2048 or 4096 (default: nfreq).
  - *pts_per_dec*: points per decade (default: None)

  Padding can sometimes improve the result, not always. The default samples from 0.002 Hz - 4.096 Hz. If pts_per_dec is set to an integer, calculated frequencies are logarithmically spaced with the given number per decade, and then interpolated to yield the required frequencies for the FFT.

xdirect [bool or None, optional] Direct field calculation (only if src and rec are in the same layer):

- If True, direct field is calculated analytically in the frequency domain.
- If False, direct field is calculated in the wavenumber domain.
- If None, direct field is excluded from the calculation, and only reflected fields are returned (secondary field).

Defaults to False.

loop [{None, 'freq', 'off'}, optional] Define if to calculate everything vectorized or if to loop over frequencies ('freq') or over offsets ('off'), default is None. It always loops over frequencies if *ht='qwe'* or if *pts_per_dec=-1*. Calculating everything vectorized is fast for few offsets OR for few frequencies. However, if you calculate many frequencies for many offsets, it might be faster to loop over frequencies. Only comparing the different versions will yield the answer for your specific problem at hand!

Returns

EM [EMArray, (nfreqtime, nrec, nsrc)] Frequency- or time-domain EM field (depending on *signal*):

- If rec is electric, returns E [V/m].
- If rec is magnetic, returns H [A/m].

EMArray is a subclassed ndarray with *.pha* and *.amp* attributes (only relevant for frequency-domain data).

The shape of EM is (nfreqtime, nrec, nsrc). However, single dimensions are removed.

See also:

`dipole()` EM fields due to infinitesimal small EM dipoles.

`loop()` EM fields due to a magnetic source loop.

Examples

```
>>> import empymod
>>> import numpy as np
>>> # x-directed bipole source: x0, x1, y0, y1, z0, z1
>>> src = [-50, 50, 0, 0, 100, 100]
>>> # x-directed dipole receiver-array: x, y, z, azimuth, dip
>>> rec = [np.arange(1, 11)*500, np.zeros(10), 200, 0, 0]
>>> # layer boundaries
>>> depth = [0, 300, 1000, 1050]
>>> # layer resistivities
>>> res = [1e20, .3, 1, 50, 1]
>>> # Frequency
>>> freq = 1
>>> # Calculate electric field due to an electric source at 1 Hz.
>>> # [msrc = mrec = False (default)]
>>> EMfield = empymod.bipole(src, rec, depth, res, freq, verb=4)
~
:: empymod START  ::  v2.0.0
~
   depth       [m] :  0 300 1000 1050
   res     [Ohm.m] :  1E+20 0.3 1 50 1
   aniso       [-] :  1 1 1 1 1
   epermH      [-] :  1 1 1 1 1
   epermV      [-] :  1 1 1 1 1
   mpermH      [-] :  1 1 1 1 1
   mpermV      [-] :  1 1 1 1 1
   direct field    :  Comp. in wavenumber domain
   frequency  [Hz] :  1
   Hankel          :  DLF (Fast Hankel Transform)
     > Filter      :  Key 201 (2009)
     > DLF type    :  Standard
   Loop over       :  None (all vectorized)
   Source(s)       :  1 bipole(s)
     > intpts      :  1 (as dipole)
     > length  [m] :  100
     > strength[A] :  0
     > x_c     [m] :  0
     > y_c     [m] :  0
     > z_c     [m] :  100
     > azimuth [°] :  0
     > dip     [°] :  0
   Receiver(s)     :  10 dipole(s)
     > x       [m] :  500 - 5000 : 10  [min-max; #]
                   :  500 1000 1500 2000 2500 3000 3500 4000 4500 5000
     > y       [m] :  0 - 0 : 10  [min-max; #]
                   :  0 0 0 0 0 0 0 0 0 0
     > z       [m] :  200
     > azimuth [°] :  0
     > dip     [°] :  0
   Required ab's   :  11
~
:: empymod END; runtime = 0:00:00.005536 :: 1 kernel call(s)
~
>>> print(EMfield)
[  1.68809346e-10 -3.08303130e-10j  -8.77189179e-12 -3.76920235e-11j
  -3.46654704e-12 -4.87133683e-12j  -3.60159726e-13 -1.12434417e-12j
   1.87807271e-13 -6.21669759e-13j   1.97200208e-13 -4.38210489e-13j
   1.44134842e-13 -3.17505260e-13j   9.92770406e-14 -2.33950871e-13j
   6.75287598e-14 -1.74922886e-13j   4.62724887e-14 -1.32266600e-13j]
```

empymod.model.**dipole**(*src*, *rec*, *depth*, *res*, *freqtime*, *signal=None*, *ab=11*, *aniso=None*, *epermH=None*, *epermV=None*, *mpermH=None*, *mpermV=None*, ***kwargs*)

Return EM fields due to infinitesimal small EM dipoles.

Calculate the electromagnetic frequency- or time-domain field due to infinitesimal small electric or magnetic dipole source(s), measured by infinitesimal small electric or magnetic dipole receiver(s); sources and receivers are directed along the principal directions x, y, or z, and all sources are at the same depth, as well as all receivers are at the same depth.

Use the functions `bipole()` to calculate dipoles with arbitrary angles or bipoles of finite length and arbitrary angle.

The function `dipole()` could be replaced by `bipole()` (all there is to do is translate *ab* into *msrc*, *mrec*, *azimuth*'s and *dip*'s). However, `dipole()` is kept separately to serve as an example of a simple modelling routine that can serve as a template.

Parameters

src, rec [list of floats or arrays] Source and receiver coordinates [x, y, z] (m):

- The x- and y-coordinates can be arrays, z is a single value.
- The x- and y-coordinates must have the same dimension.

Sources or receivers placed on a layer interface are considered in the upper layer.

depth [list] Absolute layer interfaces z (m); #depth = #res - 1 (excluding +/- infinity).

res [array_like] Horizontal resistivities rho_h (Ohm.m); #res = #depth + 1.

Alternatively, res can be a dictionary. See the main manual of empymod too see how to exploit this hook to re-calculate etaH, etaV, zetaH, and zetaV, which can be used to, for instance, use the Cole-Cole model for IP.

freqtime [array_like] Frequencies f (Hz) if *signal==None*, else times t (s); (f, t > 0).

signal [{None, 0, 1, -1}, optional] Source signal, default is None:

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

ab [int, optional] Source-receiver configuration, defaults to 11.

		electric source			magnetic source		
		x	**y**	**z**	**x**	**y**	**z**
electric receiver	**x**	11	12	13	14	15	16
	y	21	22	23	24	25	26
	z	31	32	33	34	35	36
magnetic receiver	**x**	41	42	43	44	45	46
	y	51	52	53	54	55	56
	z	61	62	63	64	65	66

aniso [array_like, optional] Anisotropies lambda = sqrt(rho_v/rho_h) (-); #aniso = #res. Defaults to ones.

epermH, epermV [array_like, optional] Relative horizontal/vertical electric permittivities epsilon_h/epsilon_v (-); #epermH = #epermV = #res. If epermH is provided but not epermV, isotropic behaviour is assumed. Default is ones.

mpermH, mpermV [array_like, optional] Relative horizontal/vertical magnetic permeabilities mu_h/mu_v (-); #mpermH = #mpermV = #res. If mpermH is provided but not mpermV, isotropic behaviour is assumed. Default is ones.

verb [{0, 1, 2, 3, 4}, optional] Level of verbosity, default is 2:

- 0: Print nothing.
- 1: Print warnings.
- 2: Print additional runtime and kernel calls
- 3: Print additional start/stop, condensed parameter information.
- 4: Print additional full parameter information

ht, htarg, ft, ftarg, xdirect, loop [settings, optinal] See docstring of *bipole()* for a description.

Returns

EM [EMArray, (nfreqtime, nrec, nsrc)] Frequency- or time-domain EM field (depending on *signal*):

- If rec is electric, returns E [V/m].
- If rec is magnetic, returns H [A/m].

EMArray is a subclassed ndarray with *.pha* and *.amp* attributes (only relevant for frequency-domain data).

The shape of EM is (nfreqtime, nrec, nsrc). However, single dimensions are removed.

See also:

bipole() EM fields due to arbitrary rotated, finite length EM dipoles.

loop() EM fields due to a magnetic source loop.

Examples

```
>>> import empymod
>>> import numpy as np
>>> src = [0, 0, 100]
>>> rec = [np.arange(1, 11)*500, np.zeros(10), 200]
>>> depth = [0, 300, 1000, 1050]
>>> res = [1e20, .3, 1, 50, 1]
>>> EMfield = empymod.dipole(src, rec, depth, res, freqtime=1, verb=0)
>>> print(EMfield)
[  1.68809346e-10 -3.08303130e-10j  -8.77189179e-12 -3.76920235e-11j
  -3.46654704e-12 -4.87133683e-12j  -3.60159726e-13 -1.12434417e-12j
   1.87807271e-13 -6.21669759e-13j   1.97200208e-13 -4.38210489e-13j
   1.44134842e-13 -3.17505260e-13j   9.92770406e-14 -2.33950871e-13j
   6.75287598e-14 -1.74922886e-13j   4.62724887e-14 -1.32266600e-13j]
```

empymod.model.**loop**(*src*, *rec*, *depth*, *res*, *freqtime*, *signal=None*, *aniso=None*, *epermH=None*, *epermV=None*, *mpermH=None*, *mpermV=None*, *mrec=True*, *recpts=1*, *strength=0*, ***kwargs*)

Return EM fields due to a magnetic source loop.

Calculate the electromagnetic frequency- or time-domain field due to an arbitrary rotated, magnetic source consisting of an electric loop, measured by arbitrary rotated, finite electric or magnetic bipole receivers or arbitrary rotated magnetic receivers consisting of electric loops. By default, the electromagnetic response is normalized to source loop area of 1 m2 and receiver length or area of 1 m or 1 m2, respectively, and source strength of 1 A.

A magnetic dipole, as used in *dipole()* and *bipole()*, has a moment of $I^m ds$. However, if the magnetic dipole is generated by an electric-wire loop, this changes to $I^m = i\omega\mu A I^e$, where A is the area of the loop. The same factor $i\omega\mu A$, applies to the receiver, if it consists of an electric-wire loop.

The current implementation only handles loop sources and receivers in layers where $\mu_r^h = \mu_r^v$; the horizontal magnetic permeability is used, and a warning is thrown if the vertical differs from the horizontal one.

Note that the kernel internally still calculates dipole sources and receivers, the moment is a factor that is multiplied in the frequency domain. The logs will therefore still inform about bipoles and dipoles.

Parameters

src, rec [list of floats or arrays] Source and receiver coordinates (m):

- [x0, x1, y0, y1, z0, z1] (bipole of finite length)
- [x, y, z, azimuth, dip] (dipole, infinitesimal small)

Dimensions:

- The coordinates x, y, and z (dipole) or x0, x1, y0, y1, z0, and z1 (bipole) can be single values or arrays.
- The variables x and y (dipole) or x0, x1, y0, and y1 (bipole) must have the same dimensions.
- The variables z, azimuth, and dip (dipole) or z0 and z1 (bipole) must either be single values or having the same dimension as the other coordinates.

Angles (coordinate system is either left-handed with positive z down or right-handed with positive z up; East-North-Depth):

- azimuth (°): horizontal deviation from x-axis, anti-clockwise.
- +/-dip (°): vertical deviation from xy-plane down/up-wards.

Sources or receivers placed on a layer interface are considered in the upper layer.

depth [list] Absolute layer interfaces z (m); #depth = #res - 1 (excluding +/- infinity).

res [array_like] Horizontal resistivities rho_h (Ohm.m); #res = #depth + 1.

Alternatively, res can be a dictionary. See the main manual of empymod too see how to exploit this hook to re-calculate etaH, etaV, zetaH, and zetaV, which can be used to, for instance, use the Cole-Cole model for IP.

freqtime [array_like] Frequencies f (Hz) if *signal*==*None*, else times t (s); (f, t > 0).

signal [{None, 0, 1, -1}, optional] Source signal, default is None:

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

aniso [array_like, optional] Anisotropies lambda = sqrt(rho_v/rho_h) (-); #aniso = #res. Defaults to ones.

epermH, epermV [array_like, optional] Relative horizontal/vertical electric permittivities epsilon_h/epsilon_v (-); #epermH = #epermV = #res. If epermH is provided but not epermV, isotropic behaviour is assumed. Default is ones.

mpermH, mpermV [array_like, optional] Relative horizontal/vertical magnetic permeabilities mu_h/mu_v (-); #mpermH = #mpermV = #res. If mpermH is provided but not mpermV, isotropic behaviour is assumed. Default is ones.

Note that the relative horizontal and vertical magnetic permeabilities in layers with loop sources or receivers will be set to 1.

mrec [bool or string, optional] Receiver options; default is True:

- True: Magnetic dipole receiver;
- False: Electric dipole receiver;

- 'loop': Magnetic receiver consisting of an electric-wire loop.

recpts [int, optional] Number of integration points for bipole receiver, default is 1:

- recpts < 3 : bipole, but calculated as dipole at centre
- recpts >= 3 : bipole

Note that if *mrec='loop'*, *recpts* will be set to 1.

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

- If 0, output is normalized to source of 1 m2 area and receiver of 1 m length or 1 m2 area, and source strength of 1 A.
- If != 0, output is returned for given source strength and receiver length (if *mrec!='loop'*).

The strength is simply a multiplication factor. It can also be used to provide the source and receiver loop area, or also to multiply by :math:mu_0', if you want the B-field instead of the H-field.

Default is 0.

verb [{0, 1, 2, 3, 4}, optional] Level of verbosity, default is 2:

- 0: Print nothing.
- 1: Print warnings.
- 2: Print additional runtime and kernel calls
- 3: Print additional start/stop, condensed parameter information.
- 4: Print additional full parameter information

ht, htarg, ft, ftarg, xdirect, loop [settings, optinal] See docstring of *bipole()* for a description.

Returns

EM [EMArray, (nfreqtime, nrec, nsrc)] Frequency- or time-domain EM field (depending on *signal*):

- If rec is electric, returns E [V/m].
- If rec is magnetic, returns H [A/m].

EMArray is a subclassed ndarray with *.pha* and *.amp* attributes (only relevant for frequency-domain data).

See also:

dipole() EM fields due to infinitesimal small EM dipoles.

bipole() EM fields due to arbitrary rotated, finite length EM dipoles.

Examples

```
>>> import empymod
>>> import numpy as np
>>> # z-directed loop source: x, y, z, azimuth, dip
>>> src = [0, 0, 0, 0, 90]
>>> # z-directed magnetic dipole receiver-array: x, y, z, azimuth, dip
>>> rec = [np.arange(1, 11)*500, np.zeros(10), 200, 0, 90]
>>> # layer boundaries
>>> depth = [0, 300, 500]
>>> # layer resistivities
>>> res = [2e14, 10, 500, 10]
```

(continues on next page)

(continued from previous page)

```
>>> # Frequency
>>> freq = 1
>>> # Calculate magnetic field due to a loop source at 1 Hz.
>>> # [mrec = True (default)]
>>> EMfield = empymod.loop(src, rec, depth, res, freq, verb=4)
~
:: empymod START  ::  w2.0.0
~
   depth       [m] :  0 300 500
   res     [Ohm.m] :  2E+14 10 500 10
   aniso       [-] :  1 1 1 1
   epermH      [-] :  1 1 1 1
   epermV      [-] :  1 1 1 1
   mpermH      [-] :  1 1 1 1
   mpermV      [-] :  1 1 1 1
   direct field    :  Comp. in wavenumber domain
   frequency  [Hz] :  1
   Hankel          :  DLF (Fast Hankel Transform)
     > Filter      :  Key 201 (2009)
     > DLF type    :  Standard
   Loop over       :  None (all vectorized)
   Source(s)       :  1 dipole(s)
     > x       [m] :  0
     > y       [m] :  0
     > z       [m] :  0
     > azimuth [°] :  0
     > dip     [°] :  90
   Receiver(s)     :  10 dipole(s)
     > x       [m] :  500 - 5000 : 10  [min-max; #]
                   :  500 1000 1500 2000 2500 3000 3500 4000 4500 5000
     > y       [m] :  0 - 0 : 10  [min-max; #]
                   :  0 0 0 0 0 0 0 0 0 0
     > z       [m] :  200
     > azimuth [°] :  0
     > dip     [°] :  90
   Required ab's   :  33
~
:: empymod END; runtime = 0:00:00.005025 :: 1 kernel call(s)
```

```
>>> print(EMfield)
[ -3.05449848e-10 -2.00374185e-11j  -7.12528991e-11 -5.37083268e-12j
  -2.52076501e-11 -1.62732412e-12j  -1.18412295e-11 -8.99570998e-14j
  -6.44054097e-12 +5.61150066e-13j  -3.77109625e-12 +7.89022722e-13j
  -2.28484774e-12 +8.08897623e-13j  -1.40021365e-12 +7.32151174e-13j
  -8.55487532e-13 +6.18402706e-13j  -5.15642408e-13 +4.99091919e-13j]
```

empymod.model.**analytical**(*src*, *rec*, *res*, *freqtime*, *solution='fs'*, *signal=None*, *ab=11*, *aniso=None*, *epermH=None*, *epermV=None*, *mpermH=None*, *mpermV=None*, ***kwargs*)

Return analytical full- or half-space solution.

Calculate the electromagnetic frequency- or time-domain field due to infinitesimal small electric or magnetic dipole source(s), measured by infinitesimal small electric or magnetic dipole receiver(s); sources and receivers are directed along the principal directions x, y, or z, and all sources are at the same depth, as well as all receivers are at the same depth.

In the case of a halfspace the air-interface is located at z = 0 m.

You can call the functions *empymod.kernel.fullspace()* and *empymod.kernel.halfspace()* in *empymod.kernel* directly. This interface is just to provide a consistent interface with the same input parameters as for instance for *dipole()*.

This function yields the same result if *solution='fs'* as `dipole()`, if the model is a fullspace.

Included are:

- Full fullspace solution (*solution='fs'*) for ee-, me-, em-, mm-fields, only frequency domain, [HuTS15].
- Diffusive fullspace solution (*solution='dfs'*) for ee-fields, [SlHM10].
- Diffusive halfspace solution (*solution='dhs'*) for ee-fields, [SlHM10].
- Diffusive direct- and reflected field and airwave (*solution='dsplit'*) for ee-fields, [SlHM10].
- Diffusive direct- and reflected field and airwave (*solution='dtetm'*) for ee-fields, split into TE and TM mode [SlHM10].

Parameters

src, rec [list of floats or arrays] Source and receiver coordinates [x, y, z] (m):

- The x- and y-coordinates can be arrays, z is a single value.
- The x- and y-coordinates must have the same dimension.

res [float] Horizontal resistivity rho_h (Ohm.m).

Alternatively, res can be a dictionary. See the main manual of empymod too see how to exploit this hook to re-calculate etaH, etaV, zetaH, and zetaV, which can be used to, for instance, use the Cole-Cole model for IP.

freqtime [array_like] Frequencies f (Hz) if *signal==None*, else times t (s); (f, t > 0).

solution [str, optional] Defines which solution is returned:

- 'fs' : Full fullspace solution (ee-, me-, em-, mm-fields); f-domain.
- 'dfs' : Diffusive fullspace solution (ee-fields only).
- 'dhs' : Diffusive halfspace solution (ee-fields only).
- 'dsplit' : Diffusive direct- and reflected field and airwave (ee-fields only).
- 'dtetm' : as dsplit, but direct fielt TE, TM; reflected field TE, TM, and airwave (ee-fields only).

signal [{None, 0, 1, -1}, optional] Source signal, default is None:

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

ab [int, optional] Source-receiver configuration, defaults to 11.

		electric source			magnetic source		
		x	**y**	**z**	**x**	**y**	**z**
electric receiver	**x**	11	12	13	14	15	16
	y	21	22	23	24	25	26
	z	31	32	33	34	35	36
magnetic receiver	**x**	41	42	43	44	45	46
	y	51	52	53	54	55	56
	z	61	62	63	64	65	66

aniso [float, optional] Anisotropy lambda = sqrt(rho_v/rho_h) (-); defaults to one.

epermH, epermV [float, optional] Relative horizontal/vertical electric permittivity epsilon_h/epsilon_v (-). If epermH is provided but not epermV, isotropic behaviour is assumed. Default is one; ignored for the diffusive solution.

mpermH, mpermV [float, optional] Relative horizontal/vertical magnetic permeability mu_h/mu_v (-); #mpermH = #mpermV = #res. If mpermH is provided but not mpermV, isotropic behaviour is assumed. Default is one; ignored for the diffusive solution.

verb [{0, 1, 2, 3, 4}, optional] Level of verbosity, default is 2:

- 0: Print nothing.
- 1: Print warnings.
- 2: Print additional runtime
- 3: Print additional start/stop, condensed parameter information.
- 4: Print additional full parameter information

Returns

EM [EMArray, (nfreqtime, nrec, nsrc)] Frequency- or time-domain EM field (depending on *signal*):

- If rec is electric, returns E [V/m].
- If rec is magnetic, returns H [A/m].

EMArray is a subclassed ndarray with *.pha* and *.amp* attributes (only relevant for frequency-domain data).

The shape of EM is (nfreqtime, nrec, nsrc). However, single dimensions are removed.

If *solution='dsplit'*, three ndarrays are returned: direct, reflect, air.

If *solution='dtetm'*, five ndarrays are returned: direct_TE, direct_TM, reflect_TE, reflect_TM, air.

Examples

```
>>> import empymod
>>> import numpy as np
>>> src = [0, 0, 0]
>>> rec = [np.arange(1, 11)*500, np.zeros(10), 200]
>>> res = 50
>>> EMfield = empymod.analytical(src, rec, res, freqtime=1, verb=0)
>>> print(EMfield)
[  4.03091405e-08 -9.69163818e-10j   6.97630362e-09 -4.88342150e-10j
   2.15205979e-09 -2.97489809e-10j   8.90394459e-10 -1.99313433e-10j
   4.32915802e-10 -1.40741644e-10j   2.31674165e-10 -1.02579391e-10j
   1.31469130e-10 -7.62770461e-11j   7.72342470e-11 -5.74534125e-11j
   4.61480481e-11 -4.36275540e-11j   2.76174038e-11 -3.32860932e-11j]
```

empymod.model.**gpr**(*src*, *rec*, *depth*, *res*, *freqtime*, *cf*, *gain=None*, *ab=11*, *aniso=None*, *epermH=None*, *epermV=None*, *mpermH=None*, *mpermV=None*, ***kwargs*)

Return Ground-Penetrating Radar signal.

THIS FUNCTION IS EXPERIMENTAL, USE WITH CAUTION.

It is rather an example how you can calculate GPR responses; however, DO NOT RELY ON IT! It works only well with QUAD or QWE (*quad*, *qwe*) for the Hankel transform, and with FFT (*fft*) for the Fourier transform.

It calls internally *dipole()* for the frequency-domain calculation. It subsequently convolves the response with a Ricker wavelet with central frequency *cf*. If signal!=None, it carries out the Fourier transform and applies a gain to the response.

Parameters

src, rec, freqtime [survey parameters] See docstring of *dipole()* for a description.

depth, res, aniso, epermH, epermV, mpermH, mpermV [model parameters] See docstring of *dipole()* for a description.

cf [float] Centre frequency of GPR-signal, in Hz. Sensible values are between 10 MHz and 3000 MHz.

gain [float] Power of gain function. If None, no gain is applied. Only used if signal!=None.

ht, htarg, ft, ftarg, xdirect, loop [settings, optinal] See docstring of *bipole()* for a description.

Returns

EM [ndarray] GPR response

empymod.model.**dipole_k**(*src*, *rec*, *depth*, *res*, *freq*, *wavenumber*, *ab=11*, *aniso=None*, *epermH=None*, *epermV=None*, *mpermH=None*, *mpermV=None*, ***kwargs*)

Return electromagnetic wavenumber-domain field.

Calculate the electromagnetic wavenumber-domain field due to infinitesimal small electric or magnetic dipole source(s), measured by infinitesimal small electric or magnetic dipole receiver(s); sources and receivers are directed along the principal directions x, y, or z, and all sources are at the same depth, as well as all receivers are at the same depth.

Parameters

src, rec [list of floats or arrays] Source and receiver coordinates [x, y, z] (m):

- The x- and y-coordinates can be arrays, z is a single value.
- The x- and y-coordinates must have the same dimension.
- The x- and y-coordinates only matter for the angle-dependent factor.

Sources or receivers placed on a layer interface are considered in the upper layer.

depth [list] Absolute layer interfaces z (m); #depth = #res - 1 (excluding +/- infinity).

res [array_like] Horizontal resistivities rho_h (Ohm.m); #res = #depth + 1.

freq [array_like] Frequencies f (Hz), used to calculate etaH/V and zetaH/V.

wavenumber [array] Wavenumbers lambda (1/m)

ab [int, optional] Source-receiver configuration, defaults to 11.

		electric source			magnetic source		
		x	**y**	**z**	**x**	**y**	**z**
electric receiver	**x**	11	12	13	14	15	16
	y	21	22	23	24	25	26
	z	31	32	33	34	35	36
magnetic receiver	**x**	41	42	43	44	45	46
	y	51	52	53	54	55	56
	z	61	62	63	64	65	66

aniso [array_like, optional] Anisotropies lambda = sqrt(rho_v/rho_h) (-); #aniso = #res. Defaults to ones.

epermH, epermV [array_like, optional] Relative horizontal/vertical electric permittivities epsilon_h/epsilon_v (-); #epermH = #epermV = #res. If epermH is provided but not epermV, isotropic behaviour is assumed. Default is ones.

mpermH, mpermV [array_like, optional] Relative horizontal/vertical magnetic permeabilities mu_h/mu_v (-); #mpermH = #mpermV = #res. If mpermH is provided but not mpermV, isotropic behaviour is assumed. Default is ones.

verb [{0, 1, 2, 3, 4}, optional] Level of verbosity, default is 2:

- 0: Print nothing.
- 1: Print warnings.
- 2: Print additional runtime and kernel calls
- 3: Print additional start/stop, condensed parameter information.
- 4: Print additional full parameter information

Returns

PJ0, PJ1 [array] Wavenumber-domain EM responses:

- PJ0: Wavenumber-domain solution for the kernel with a Bessel function of the first kind of order zero.
- PJ1: Wavenumber-domain solution for the kernel with a Bessel function of the first kind of order one.

See also:

dipole() EM fields due to infinitesimal small EM dipoles.

bipole() EM fields due to arbitrary rotated, finite length EM dipoles.

loop() EM fields due to a magnetic source loop.

Examples

```
>>> import empymod
>>> import numpy as np
>>> src = [0, 0, 100]
>>> rec = [5000, 0, 200]
>>> depth = [0, 300, 1000, 1050]
>>> res = [1e20, .3, 1, 50, 1]
>>> freq = 1
>>> wavenr = np.logspace(-3.7, -3.6, 10)
>>> PJ0, PJ1 = empymod.dipole_k(src, rec, depth, res, freq, wavenr, verb=0)
>>> print(PJ0)
[ -1.02638329e-08 +4.91531529e-09j  -1.05289724e-08 +5.04222413e-09j
  -1.08009148e-08 +5.17238608e-09j  -1.10798310e-08 +5.30588284e-09j
  -1.13658957e-08 +5.44279805e-09j  -1.16592877e-08 +5.58321732e-09j
  -1.19601897e-08 +5.72722830e-09j  -1.22687889e-08 +5.87492067e-09j
  -1.25852765e-08 +6.02638626e-09j  -1.29098481e-08 +6.18171904e-09j]
>>> print(PJ1)
[  1.79483705e-10 -6.59235332e-10j   1.88672497e-10 -6.93749344e-10j
   1.98325814e-10 -7.30068377e-10j   2.08466693e-10 -7.68286748e-10j
   2.19119282e-10 -8.08503709e-10j   2.30308887e-10 -8.50823701e-10j
   2.42062030e-10 -8.95356636e-10j   2.54406501e-10 -9.42218177e-10j
   2.67371420e-10 -9.91530051e-10j   2.80987292e-10 -1.04342036e-09j]
```

empymod.model.**fem**(*ab*, *off*, *angle*, *zsrc*, *zrec*, *lsrc*, *lrec*, *depth*, *freq*, *etaH*, *etaV*, *zetaH*, *zetaV*, *xdirect*, *isfullspace*, *ht*, *htarg*, *msrc*, *mrec*, *loop_freq*, *loop_off*, *conv=True*)

Return electromagnetic frequency-domain response.

This function is called from one of the modelling routines *empymod.model*. Consult those for more details regarding the input and output parameters.

This function can be used directly if you are sure the provided input is in the correct format. This is useful for inversion routines and similar, as it can speed-up the calculation by omitting input-checks.

empymod.model.**tem**(*fEM*, *off*, *freq*, *time*, *signal*, *ft*, *ftarg*, *conv=True*)
Return time-domain response of the frequency-domain response fEM.

This function is called from one of the modelling routines *empymod.model*. Consult those for more details regarding the input and output parameters.

This function can be used directly if you are sure the provided input is in the correct format. This is useful for inversion routines and similar, as it can speed-up the calculation by omitting input-checks.

5.10 Other functions

5.10.1 empymod.kernel – Kernel calculation

Kernel of empymod, calculates the wavenumber-domain electromagnetic response. Plus analytical full- and half-space solutions.

The functions *wavenumber()*, *angle_factor()*, *fullspace()*, *greenfct()*, *reflections()*, and *fields()* are based on source files (specified in each function) from the source code distributed with [HuTS15], which can be found at software.seg.org/2015/0001. These functions are (c) 2015 by Hunziker et al. and the Society of Exploration Geophysicists, https://software.seg.org/disclaimer.txt. Please read the NOTICE-file in the root directory for more information regarding the involved licenses.

empymod.kernel.**wavenumber**(*zsrc*, *zrec*, *lsrc*, *lrec*, *depth*, *etaH*, *etaV*, *zetaH*, *zetaV*, *lambd*, *ab*, *xdirect*, *msrc*, *mrec*)
Calculate wavenumber domain solution.

Return the wavenumber domain solutions *PJ0*, *PJ1*, and *PJ0b*, which have to be transformed with a Hankel transform to the frequency domain. *PJ0*/*PJ0b* and *PJ1* have to be transformed with Bessel functions of order 0 (J_0) and 1 (J_1), respectively.

This function corresponds loosely to equations 105–107, 111–116, 119–121, and 123–128 in [HuTS15], and equally loosely to the file *kxwmod.c*.

[HuTS15] uses Bessel functions of orders 0, 1, and 2 (J_0, J_1, J_2). The implementations of the *Fast Hankel Transform* and the *Quadrature-with-Extrapolation* in *empymod.transform* are set-up with Bessel functions of order 0 and 1 only. This is achieved by applying the recurrence formula

$$J_2(kr) = \frac{2}{kr} J_1(kr) - J_0(kr) \; . \tag{5.5}$$

Note: *PJ0* and *PJ0b* could theoretically be added here into one, and then be transformed in one go. However, *PJ0b* has to be multiplied by `ang_fact()` later. This has to be done after the Hankel transform for methods which make use of spline interpolation, in order to work for offsets that are not in line with each other.

This function is called from one of the Hankel functions in *empymod.transform*. Consult the modelling routines in *empymod.model* for a description of the input and output parameters.

If you are solely interested in the wavenumber-domain solution you can call this function directly. However, you have to make sure all input arguments are correct, as no checks are carried out here.

empymod.kernel.**angle_factor**(*angle*, *ab*, *msrc*, *mrec*)
Return the angle-dependent factor.

The whole calculation in the wavenumber domain is only a function of the distance between the source and the receiver, it is independent of the angel. The angle-dependency is this factor, which can be applied to the corresponding parts in the wavenumber or in the frequency domain.

The `angle_factor()` corresponds to the sine and cosine-functions in Eqs 105-107, 111-116, 119-121, 123-128.

This function is called from one of the Hankel functions in `empymod.transform`. Consult the modelling routines in `empymod.model` for a description of the input and output parameters.

empymod.kernel.**fullspace**(*off*, *angle*, *zsrc*, *zrec*, *etaH*, *etaV*, *zetaH*, *zetaV*, *ab*, *msrc*, *mrec*)
Analytical full-space solutions in the frequency domain.

$$\hat{G}^{ee}_{\alpha\beta}, \hat{G}^{ee}_{3\alpha}, \hat{G}^{ee}_{33}, \hat{G}^{em}_{\alpha\beta}, \hat{G}^{em}_{\alpha 3} \tag{5.6}$$

This function corresponds to equations 45–50 in [HuTS15], and loosely to the corresponding files *Gin11.F90*, *Gin12.F90*, *Gin13.F90*, *Gin22.F90*, *Gin23.F90*, *Gin31.F90*, *Gin32.F90*, *Gin33.F90*, *Gin41.F90*, *Gin42.F90*, *Gin43.F90*, *Gin51.F90*, *Gin52.F90*, *Gin53.F90*, *Gin61.F90*, and *Gin62.F90*.

This function is called from one of the modelling routines in `empymod.model`. Consult these modelling routines for a description of the input and output parameters.

empymod.kernel.**greenfct**(*zsrc*, *zrec*, *lsrc*, *lrec*, *depth*, *etaH*, *etaV*, *zetaH*, *zetaV*, *lambd*, *ab*, *xdirect*, *msrc*, *mrec*)
Calculate Green's function for TM and TE.

$$\tilde{g}^{tm}_{hh}, \tilde{g}^{tm}_{hz}, \tilde{g}^{tm}_{zh}, \tilde{g}^{tm}_{zz}, \tilde{g}^{te}_{hh}, \tilde{g}^{te}_{zz} \tag{5.7}$$

This function corresponds to equations 108–110, 117/118, 122; 89–94, A18–A23, B13–B15; 97–102 A26–A31, and B16–B18 in [HuTS15], and loosely to the corresponding files *Gamma.F90*, *Wprop.F90*, *Ptotalx.F90*, *Ptotalxm.F90*, *Ptotaly.F90*, *Ptotalym.F90*, *Ptotalz.F90*, and *Ptotalzm.F90*.

The Green's functions are multiplied according to Eqs 105-107, 111-116, 119-121, 123-128; with the factors inside the integrals.

This function is called from the function `wavenumber()`.

empymod.kernel.**reflections**(*depth*, *e_zH*, *Gam*, *lrec*, *lsrc*)
Calculate Rp, Rm.

$$R^{\pm}_n, \bar{R}^{\pm}_n \tag{5.8}$$

This function corresponds to equations 64/65 and A-11/A-12 in [HuTS15], and loosely to the corresponding files *Rmin.F90* and *Rplus.F90*.

This function is called from the function `greenfct()`.

empymod.kernel.**fields**(*depth*, *Rp*, *Rm*, *Gam*, *lrec*, *lsrc*, *zsrc*, *ab*, *TM*)
Calculate Pu+, Pu-, Pd+, Pd-.

$$P^{u\pm}_s, P^{d\pm}_s, \bar{P}^{u\pm}_s, \bar{P}^{d\pm}_s; P^{u\pm}_{s-1}, P^{u\pm}_n, \bar{P}^{u\pm}_{s-1}, \bar{P}^{u\pm}_n; P^{d\pm}_{s+1}, P^{d\pm}_n, \bar{P}^{d\pm}_{s+1}, \bar{P}^{d\pm}_n \tag{5.9}$$

This function corresponds to equations 81/82, 95/96, 103/104, A-8/A-9, A-24/A-25, and A-32/A-33 in [HuTS15], and loosely to the corresponding files *Pdownmin.F90*, *Pdownplus.F90*, *Pupmin.F90*, and *Pdownmin.F90*.

This function is called from the function `greenfct()`.

empymod.kernel.**halfspace**(*off*, *angle*, *zsrc*, *zrec*, *etaH*, *etaV*, *freqtime*, *ab*, *signal*, *solution='dhs'*)
Return frequency- or time-space domain VTI half-space solution.

Calculates the frequency- or time-space domain electromagnetic response for a half-space below air using the diffusive approximation, as given in [SlHM10], where the electric source is located at [x=0, y=0, z=zsrc>=0], and the electric receiver at [x=cos(angle)*off, y=sin(angle)*off, z=zrec>=0].

It can also be used to calculate the fullspace solution or the separate fields: direct field, reflected field, and airwave; always using the diffusive approximation. See *solution*-parameter.

This function is called from one of the modelling routines in `empymod.model`. Consult these modelling routines for a description of the input and solution parameters.

5.10.2 `empymod.transform` – Hankel and Fourier Transforms

Methods to carry out the required Hankel transform from wavenumber to frequency domain and Fourier transform from frequency to time domain.

The functions for the QWE and DLF Hankel and Fourier transforms are based on source files (specified in each function) from the source code distributed with [Key12], which can be found at software.seg.org/2012/0003. These functions are (c) 2012 by Kerry Key and the Society of Exploration Geophysicists, https://software.seg.org/disclaimer.txt. Please read the NOTICE-file in the root directory for more information regarding the involved licenses.

`empymod.transform.`**`hankel_dlf`**(*zsrc*, *zrec*, *lsrc*, *lrec*, *off*, *ang_fact*, *depth*, *ab*, *etaH*, *etaV*, *zetaH*, *zetaV*, *xdirect*, *htarg*, *msrc*, *mrec*)

Hankel Transform using the Digital Linear Filter method.

The *Digital Linear Filter* method was introduced to geophysics by [Ghos70], and made popular and widespread by [Ande75], [Ande79], [Ande82]. The DLF is sometimes referred to as the *Fast Hankel Transform* FHT, from which this routine has its name.

This implementation of the DLF follows [Key12], equation 6. Without going into the mathematical details (which can be found in any of the above papers) and following [Key12], the DLF method rewrites the Hankel transform of the form

$$F(r) = \int_0^\infty f(\lambda) J_v(\lambda r)\, \mathrm{d}\lambda \tag{5.10}$$

as

$$F(r) = \sum_{i=1}^{n} f(b_i/r) h_i / r\ , \tag{5.11}$$

where h is the digital filter.The Filter abscissae b is given by

$$b_i = \lambda_i r = e^{ai}, \qquad i = -l, -l+1, \cdots, l\ , \tag{5.12}$$

with $l = (n-1)/2$, and a is the spacing coefficient.

This function is loosely based on *get_CSEM1D_FD_FHT.m* from the source code distributed with [Key12].

The function is called from one of the modelling routines in `empymod.model`. Consult these modelling routines for a description of the input and output parameters.

Returns

fEM [array] Returns frequency-domain EM response.

kcount [int] Kernel count. For DLF, this is 1.

conv [bool] Only relevant for QWE/QUAD.

`empymod.transform.`**`hankel_qwe`**(*zsrc*, *zrec*, *lsrc*, *lrec*, *off*, *ang_fact*, *depth*, *ab*, *etaH*, *etaV*, *zetaH*, *zetaV*, *xdirect*, *htarg*, *msrc*, *mrec*)

Hankel Transform using Quadrature-With-Extrapolation.

Quadrature-With-Extrapolation was introduced to geophysics by [Key12]. It is one of many so-called *ISE* methods to solve Hankel Transforms, where *ISE* stands for Integration, Summation, and Extrapolation.

Following [Key12], but without going into the mathematical details here, the QWE method rewrites the Hankel transform of the form

$$F(r) = \int_0^\infty f(\lambda) J_v(\lambda r)\, \mathrm{d}\lambda \tag{5.13}$$

as a quadrature sum which form is similar to the DLF (equation 15),

$$F_i \approx \sum_{j=1}^{m} f(x_j/r) w_j g(x_j) = \sum_{j=1}^{m} f(x_j/r) \hat{g}(x_j) \;, \tag{5.14}$$

but with various bells and whistles applied (using the so-called Shanks transformation in the form of a routine called ϵ-algorithm ([Shan55], [Wynn56]; implemented with algorithms from [Tref00] and [Weni89]).

This function is based on *get_CSEM1D_FD_QWE.m*, *qwe.m*, and *getBesselWeights.m* from the source code distributed with [Key12].

In the spline-version, `hankel_qwe()` checks how steep the decay of the wavenumber-domain result is, and calls QUAD for the very steep interval, for which QWE is not suited.

The function is called from one of the modelling routines in `empymod.model`. Consult these modelling routines for a description of the input and output parameters.

Returns

fEM [array] Returns frequency-domain EM response.

kcount [int] Kernel count.

conv [bool] If true, QWE/QUAD converged. If not, *htarg* might have to be adjusted.

empymod.transform.**hankel_quad**(*zsrc*, *zrec*, *lsrc*, *lrec*, *off*, *ang_fact*, *depth*, *ab*, *etaH*, *etaV*, *zetaH*, *zetaV*, *xdirect*, *htarg*, *msrc*, *mrec*)

Hankel Transform using the *QUADPACK* library.

This routine uses the `scipy.integrate.quad()` module, which in turn makes use of the Fortran library *QUADPACK* (*qagse*).

It is massively (orders of magnitudes) slower than either `hankel_dlf()` or `hankel_qwe()`, and is mainly here for completeness and comparison purposes. It always uses interpolation in the wavenumber domain, hence it generally will not be as precise as the other methods. However, it might work in some areas where the others fail.

The function is called from one of the modelling routines in `empymod.model`. Consult these modelling routines for a description of the input and output parameters.

Returns

fEM [array] Returns frequency-domain EM response.

kcount [int] Kernel count. For HQUAD, this is 1.

conv [bool] If true, QUAD converged. If not, *htarg* might have to be adjusted.

empymod.transform.**fourier_dlf**(*fEM*, *time*, *freq*, *ftarg*)

Fourier Transform using the Digital Linear Filter method.

It follows the Filter methodology [Ande75], using Cosine- and Sine-filters; see `hankel_dlf()` for more information.

The function is called from one of the modelling routines in `empymod.model`. Consult these modelling routines for a description of the input and output parameters.

This function is based on *get_CSEM1D_TD_FHT.m* from the source code distributed with [Key12].

Returns

tEM [array] Returns time-domain EM response of *fEM* for given *time*.

conv [bool] Only relevant for QWE/QUAD.

empymod.transform.**fourier_qwe**(*fEM*, *time*, *freq*, *ftarg*)

Fourier Transform using Quadrature-With-Extrapolation.

It follows the QWE methodology [Key12] for the Hankel transform, see `hankel_qwe()` for more information.

The function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a description of the input and output parameters.

This function is based on *get_CSEM1D_TD_QWE.m* from the source code distributed with [Key12].

fourier_qwe() checks how steep the decay of the frequency-domain result is, and calls QUAD for the very steep interval, for which QWE is not suited.

Returns

tEM [array] Returns time-domain EM response of *fEM* for given *time*.

conv [bool] If true, QWE/QUAD converged. If not, *ftarg* might have to be adjusted.

empymod.transform.**fourier_fftlog**(*fEM*, *time*, *freq*, *ftarg*)

Fourier Transform using FFTLog.

FFTLog is the logarithmic analogue to the Fast Fourier Transform FFT. FFTLog was presented in Appendix B of [Hami00] and published at http://casa.colorado.edu/~ajsh/FFTLog.

This function uses a simplified version of *pyfftlog*, which is a python-version of *FFTLog*. For more details regarding *pyfftlog* see https://github.com/prisae/pyfftlog.

Not the full flexibility of *FFTLog* is available here: Only the logarithmic FFT (*fftl* in *FFTLog*), not the Hankel transform (*hankel_dlf()* in *FFTLog*). Furthermore, the following parameters are fixed:

- *kr* = 1 (initial value)
- *kropt* = 1 (silently adjusts *kr*)
- *dir* = 1 (forward)

Furthermore, *q* is restricted to -1 <= q <= 1.

The function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a description of the input and output parameters.

Returns

tEM [array] Returns time-domain EM response of *fEM* for given *time*.

conv [bool] Only relevant for QWE/QUAD.

empymod.transform.**fourier_fft**(*fEM*, *time*, *freq*, *ftarg*)

Fourier Transform using the Fast Fourier Transform.

The function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a description of the input and output parameters.

Returns

tEM [array] Returns time-domain EM response of *fEM* for given *time*.

conv [bool] Only relevant for QWE/QUAD.

empymod.transform.**dlf**(*signal*, *points*, *out_pts*, *filt*, *pts_per_dec*, *kind=None*, *ang_fact=None*, *ab=None*, *int_pts=None*)

Digital Linear Filter method.

This is the kernel of the DLF method, used for the Hankel (*hankel_dlf()*) and the Fourier (*fourier_dlf()*) Transforms. See *hankel_dlf()* for an extensive description.

For the Hankel transform, *signal* contains 3 complex wavenumber-domain signals: (PJ0, PJ1, PJ0b), as returned from *kernel.wavenumber*. The Hankel DLF has two additional, optional parameters: *ang_fact*, as returned from *kernel.angle_factor*, and *ab*. The PJ0-kernel is the part of the wavenumber-domain calculation which contains a zeroth-order Bessel function and does NOT depend on the angle between source and receiver, only on offset. PJ0b and PJ1 are the parts of the wavenumber-domain calculation which contain a zeroth- and first-order Bessel function, respectively, and can depend on the angle between source and receiver. PJ0, PJ1, or PJ0b can also be None, if they are not used.

For the Fourier transform, *signal* is a complex frequency-domain signal. The Fourier DLF requires one additional parameter, *kind*, which will be 'cos' or 'sin'.

empymod.transform.**qwe**(*rtol*, *atol*, *maxint*, *inp*, *intervals*, *lambd=None*, *off=None*, *ang_fact=None*)
Quadrature-With-Extrapolation.

This is the kernel of the QWE method, used for the Hankel (`hankel_qwe()`) and the Fourier (`fourier_qwe()`) Transforms. See `hankel_qwe()` for an extensive description.

This function is based on *qwe.m* from the source code distributed with [Key12].

empymod.transform.**get_dlf_points**(*filt*, *inp*, *nr_per_dec*)
Return calculation points required for DLF.

empymod.transform.**get_fftlog_input**(*rmin*, *rmax*, *n*, *q*, *mu*)
Return parameters required for FFTLog.

5.10.3 empymod.filters – Digital Linear Filters

Filters for the *Digital Linear Filter* (DLF) method for the Hankel [Ghos70]) and the Fourier ([Ande75]) transforms.

To calculate the *dlf.factor* I used

```
np.around(np.average(dlf.base[1:]/dlf.base[:-1]), 15)
```

The filters *kong_61_2007* and *kong_241_2007* from [Kong07], and *key_101_2009*, *key_201_2009*, *key_401_2009*, *key_81_CosSin_2009*, *key_241_CosSin_2009*, and *key_601_CosSin_2009* from [Key09] are taken from *DIPOLE1D*, [Key09], which can be downloaded at https://marineemlab.ucsd.edu/Projects/Occam/1DCSEM (1DCSEM). *DIPOLE1D* is distributed under the license GNU GPL version 3 or later. Kerry Key gave his written permission to re-distribute the filters under the Apache License, Version 2.0 (email from Kerry Key to Dieter Werthmüller, 21 November 2016).

The filters *anderson_801_1982* from [Ande82] and *key_51_2012*, *key_101_2012*, *key_201_2012*, *key_101_CosSin_2012*, and *key_201_CosSin_2012*, all from [Key12], are taken from the software distributed with [Key12] and available at https://software.seg.org/2012/0003 (SEG-2012-003). These filters are distributed under the SEG license.

The filter *wer_201_2018* was designed with the add-on *fdesign*, see https://github.com/empymod/article-fdesign.

class empymod.filters.**DigitalFilter**(*name*, *savename=None*, *filter_coeff=None*)
Simple Class for Digital Linear Filters.

Parameters

name [str] Name of the DFL.

savename = str Name with which the filter is saved. If None (default) it is set to the same value as *name*.

filter_coeff = list of str By default, the following filter coefficients are checked:

```
filter_coeff = ['j0', 'j1', 'sin', 'cos']
```

This accounts for the standard Hankel and Fourier DLF in CSEM modelling. However, additional coefficient names can be provided via this parameter (in list format).

fromfile(*self*, *path='filters'*)
Load filter values from ASCII-files.

Load filter base and filter coefficients from ASCII files in the directory *path*; *path* can be a relative or absolute path.

Examples

```
>>> import empymod
>>> # Create an empty filter;
>>> # Name has to be the base of the text files
>>> filt = empymod.filters.DigitalFilter('my-filter')
>>> # Load the ASCII-files
>>> filt.fromfile()
>>> # This will load the following three files:
>>> #    ./filters/my-filter_base.txt
>>> #    ./filters/my-filter_j0.txt
>>> #    ./filters/my-filter_j1.txt
>>> # and store them in filt.base, filt.j0, and filt.j1.
```

tofile(*self*, *path='filters'*)

Save filter values to ASCII-files.

Store the filter base and the filter coefficients in separate files in the directory *path*; *path* can be a relative or absolute path.

Examples

```
>>> import empymod
>>> # Load a filter
>>> filt = empymod.filters.wer_201_2018()
>>> # Save it to pure ASCII-files
>>> filt.tofile()
>>> # This will save the following three files:
>>> #    ./filters/wer_201_2018_base.txt
>>> #    ./filters/wer_201_2018_j0.txt
>>> #    ./filters/wer_201_2018_j1.txt
```

empymod.filters.**kong_61_2007**()

Kong 61 pt Hankel filter, as published in [Kong07].

Taken from file *FilterModules.f90* provided with 1DCSEM.

License: Apache License, Version 2.0,.

empymod.filters.**kong_241_2007**()

Kong 241 pt Hankel filter, as published in [Kong07].

Taken from file *FilterModules.f90* provided with 1DCSEM.

License: Apache License, Version 2.0,.

empymod.filters.**key_101_2009**()

Key 101 pt Hankel filter, as published in [Key09].

Taken from file *FilterModules.f90* provided with 1DCSEM.

License: Apache License, Version 2.0,.

empymod.filters.**key_201_2009**()

Key 201 pt Hankel filter, as published in [Key09].

Taken from file *FilterModules.f90* provided with 1DCSEM.

License: Apache License, Version 2.0,.

empymod.filters.**key_401_2009**()

Key 401 pt Hankel filter, as published in [Key09].

Taken from file *FilterModules.f90* provided with 1DCSEM.

License: Apache License, Version 2.0,.

empymod.filters.**anderson_801_1982**()
Anderson 801 pt Hankel filter, as published in [Ande82].

Taken from file *wa801Hankel.txt* provided with SEG-2012-003.

License: https://software.seg.org/disclaimer.txt.

empymod.filters.**key_51_2012**()
Key 51 pt Hankel filter, as published in [Key12].

Taken from file *kk51Hankel.txt* provided with SEG-2012-003.

License: https://software.seg.org/disclaimer.txt.

empymod.filters.**key_101_2012**()
Key 101 pt Hankel filter, as published in [Key12].

Taken from file *kk101Hankel.txt* provided with SEG-2012-003.

License: https://software.seg.org/disclaimer.txt.

empymod.filters.**key_201_2012**()
Key 201 pt Hankel filter, as published in [Key12].

Taken from file *kk201Hankel.txt* provided with SEG-2012-003.

License: https://software.seg.org/disclaimer.txt.

empymod.filters.**wer_201_2018**()
Werthmüller 201 pt Hankel filter, 2018.

Designed with the empymod add-on *fdesign*, see https://github.com/empymod/article-fdesign.

License: Apache License, Version 2.0,.

empymod.filters.**key_81_CosSin_2009**()
Key 81 pt CosSin filter, as published in [Key09].

Taken from file *FilterModules.f90* provided with 1DCSEM.

License: Apache License, Version 2.0,.

empymod.filters.**key_241_CosSin_2009**()
Key 241 pt CosSin filter, as published in [Key09].

Taken from file *FilterModules.f90* provided with 1DCSEM.

License: Apache License, Version 2.0,.

empymod.filters.**key_601_CosSin_2009**()
Key 601 pt CosSin filter, as published in [Key09].

Taken from file *FilterModules.f90* provided with 1DCSEM.

License: Apache License, Version 2.0,.

empymod.filters.**key_101_CosSin_2012**()
Key 101 pt CosSin filter, as published in [Key12].

Taken from file *kk101CosSin.txt* provided with SEG-2012-003.

License: https://software.seg.org/disclaimer.txt.

empymod.filters.**key_201_CosSin_2012**()
Key 201 pt CosSin filter, as published in [Key12].

Taken from file *kk201CosSin.txt* provided with SEG-2012-003.

License: https://software.seg.org/disclaimer.txt.

5.10.4 empymod.utils – Utilites

Utilities for *empymod.model* such as checking input parameters.

This module consists of four groups of functions:

0. General settings
1. Class EMArray
2. Input parameter checks for modelling
3. Internal utilities

class empymod.utils.**EMArray**
Create an EM-ndarray: add *amplitude* <amp> and *phase* <pha> methods.

Parameters

data [array] Data to which to add *.amp* and *.pha* attributes.

Examples

```
>>> import numpy as np
>>> from empymod.utils import EMArray
>>> emvalues = EMArray(np.array([1+1j, 1-4j, -1+2j]))
>>> print(f"Amplitude          : {emvalues.amp()}")
Amplitude          : [1.41421356 4.12310563 2.23606798]
>>> print(f"Phase (rad)        : {emvalues.pha()}")
Phase (rad)        : [ 0.78539816 -1.32581766 -4.24874137]
>>> print(f"Phase (deg)        : {emvalues.pha(deg=True)}")
Phase (deg)        : [  45.          -75.96375653 -243.43494882]
>>> print(f"Phase (deg; lead) : {emvalues.pha(deg=True, lag=False)}")
Phase (deg; lead) : [-45.           75.96375653 243.43494882]
```

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

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).

empymod.utils.**check_time_only** (*time*, *signal*, *verb*)
Check time and signal parameters.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

time [array_like] Times t (s).

signal [{None, 0, 1, -1}] Source signal:

- None: Frequency-domain response
- -1 : Switch-off time-domain response

- 0 : Impulse time-domain response
- +1 : Switch-on time-domain response

verb [{0, 1, 2, 3, 4}] Level of verbosity.

Returns

time [float] Time, checked for size and assured min_time.

empymod.utils.**check_time**(*time*, *signal*, *ft*, *ftarg*, *verb*)

Check time domain specific input parameters.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

time [array_like] Times t (s).

signal [{None, 0, 1, -1}] Source signal:

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

ft [{'dlf', 'qwe', 'fftlog', 'fft'}] Flag for Fourier transform.

ftarg [dict] Arguments of Fourier transform; depends on the value for *ft*.

verb [{0, 1, 2, 3, 4}] Level of verbosity.

Returns

time [float] Time, checked for size and assured min_time.

freq [float] Frequencies required for given times and ft-settings.

ft, ftarg Checked if valid and set to defaults if not provided, checked with signal.

empymod.utils.**check_model**(*depth*, *res*, *aniso*, *epermH*, *epermV*, *mpermH*, *mpermV*, *xdirect*, *verb*)

Check the model: depth and corresponding layer parameters.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

depth [list] Absolute layer interfaces z (m); #depth = #res - 1 (excluding +/- infinity).

res [array_like] Horizontal resistivities rho_h (Ohm.m); #res = #depth + 1.

aniso [array_like] Anisotropies lambda = sqrt(rho_v/rho_h) (-); #aniso = #res.

epermH, epermV [array_like] Relative horizontal/vertical electric permittivities epsilon_h/epsilon_v (-); #epermH = #epermV = #res.

mpermH, mpermV [array_like] Relative horizontal/vertical magnetic permeabilities mu_h/mu_v (-); #mpermH = #mpermV = #res.

xdirect [bool, optional] If True and source and receiver are in the same layer, the direct field is calculated analytically in the frequency domain, if False it is calculated in the wavenumber domain.

verb [{0, 1, 2, 3, 4}] Level of verbosity.

Returns

depth [array] Depths of layer interfaces, adds -infty at beginning if not present.

res [array] As input, checked for size.

aniso [array] As input, checked for size. If None, defaults to an array of ones.

epermH, epermV [array_like] As input, checked for size. If None, defaults to an array of ones.

mpermH, mpermV [array_like] As input, checked for size. If None, defaults to an array of ones.

isfullspace [bool] If True, the model is a fullspace (res, aniso, epermH, epermV, mpermM, and mpermV are in all layers the same).

empymod.utils.**check_frequency**(*freq*, *res*, *aniso*, *epermH*, *epermV*, *mpermH*, *mpermV*, *verb*)

Calculate frequency-dependent parameters.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

freq [array_like] Frequencies f (Hz).

res [array_like] Horizontal resistivities rho_h (Ohm.m); #res = #depth + 1.

aniso [array_like] Anisotropies lambda = sqrt(rho_v/rho_h) (-); #aniso = #res.

epermH, epermV [array_like] Relative horizontal/vertical electric permittivities epsilon_h/epsilon_v (-); #epermH = #epermV = #res.

mpermH, mpermV [array_like] Relative horizontal/vertical magnetic permeabilities mu_h/mu_v (-); #mpermH = #mpermV = #res.

verb [{0, 1, 2, 3, 4}] Level of verbosity.

Returns

freq [float] Frequency, checked for size and assured min_freq.

etaH, etaV [array] Parameters etaH/etaV, same size as provided resistivity.

zetaH, zetaV [array] Parameters zetaH/zetaV, same size as provided resistivity.

empymod.utils.**check_hankel**(*ht*, *htarg*, *verb*)

Check Hankel transform parameters.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

ht [{'dlf', 'qwe', 'quad'}] Flag to choose the Hankel transform.

htarg [dict] Arguments of Hankel transform; depends on the value for *ht*.

verb [{0, 1, 2, 3, 4}] Level of verbosity.

Returns

ht, htarg Checked if valid and set to defaults if not provided.

empymod.utils.**check_loop**(*loop*, *ht*, *htarg*, *verb*)

Check loop parameter.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

loop [{None, 'freq', 'off'}] Loop flag.

ht [{'dlf', 'qwe', 'quad'}] Flag to choose the Hankel transform.

htarg [dict] Arguments of Hankel transform; depends on the value for *ht*.

verb [{0, 1, 2, 3, 4}] Level of verbosity.

Returns

loop_freq [bool] Boolean if to loop over frequencies.

loop_off [bool] Boolean if to loop over offsets.

empymod.utils.**check_dipole**(*inp*, *name*, *verb*)
Check dipole parameters.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

inp [list of floats or arrays] Pole coordinates (m): [pole-x, pole-y, pole-z].

name [str, {'src', 'rec'}] Pole-type.

verb [{0, 1, 2, 3, 4}] Level of verbosity.

Returns

inp [list] List of pole coordinates [x, y, z].

ninp [int] Number of inp-elements

empymod.utils.**check_bipole**(*inp*, *name*)
Check di-/bipole parameters.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

inp [list of floats or arrays] Coordinates of inp (m): [dipole-x, dipole-y, dipole-z, azimuth, dip] or. [bipole-x0, bipole-x1, bipole-y0, bipole-y1, bipole-z0, bipole-z1].

name [str, {'src', 'rec'}] Pole-type.

Returns

inp [list] As input, checked for type and length.

ninp [int] Number of inp.

ninpz [int] Number of inp depths (ninpz is either 1 or ninp).

isdipole [bool] True if inp is a dipole.

empymod.utils.**check_ab**(*ab*, *verb*)
Check source-receiver configuration.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

ab [int] Source-receiver configuration.

verb [{0, 1, 2, 3, 4}] Level of verbosity.

Returns

ab_calc [int] Adjusted source-receiver configuration using reciprocity.

msrc, mrec [bool] If True, src/rec is magnetic; if False, src/rec is electric.

empymod.utils.**check_solution**(*solution*, *signal*, *ab*, *msrc*, *mrec*)
Check required solution with parameters.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

solution [str] String to define analytical solution.

signal [{None, 0, 1, -1}] Source signal:

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

msrc, mrec [bool] True if src/rec is magnetic, else False.

empymod.utils.**get_abs**(*msrc*, *mrec*, *srcazm*, *srcdip*, *recazm*, *recdip*, *verb*)
Get required ab's for given angles.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

msrc, mrec [bool] True if src/rec is magnetic, else False.

srcazm, recazm [float] Horizontal source/receiver angle (azimuth).

srcdip, recdip [float] Vertical source/receiver angle (dip).

verb [{0, 1, 2, 3, 4}] Level of verbosity.

Returns

ab_calc [array of int] ab's to calculate for this bipole.

empymod.utils.**get_geo_fact**(*ab*, *srcazm*, *srcdip*, *recazm*, *recdip*, *msrc*, *mrec*)
Get required geometrical scaling factor for given angles.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

ab [int] Source-receiver configuration.

srcazm, recazm [float] Horizontal source/receiver angle.

srcdip, recdip [float] Vertical source/receiver angle.

Returns

fact [float] Geometrical scaling factor.

empymod.utils.**get_azm_dip**(*inp*, *iz*, *ninpz*, *intpts*, *isdipole*, *strength*, *name*, *verb*)
Get angles, interpolation weights and normalization weights.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

inp [list of floats or arrays] Input coordinates (m):

- [x0, x1, y0, y1, z0, z1] (bipole of finite length)
- [x, y, z, azimuth, dip] (dipole, infinitesimal small)

iz [int] Index of current di-/bipole depth (-).

ninpz [int] Total number of di-/bipole depths (ninpz = 1 or npinz = nsrc) (-).

intpts [int] Number of integration points for bipole (-).

isdipole [bool] Boolean if inp is a dipole.

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

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

name [str, {'src', 'rec'}] Pole-type.

verb [{0, 1, 2, 3, 4}] Level of verbosity.

Returns

tout [list of floats or arrays] Dipole coordinates x, y, and z (m).

azm [float or array of floats] Horizontal angle (azimuth).

dip [float or array of floats] Vertical angle (dip).

g_w [float or array of floats] Factors from Gaussian interpolation.

intpts [int] As input, checked.

inp_w [float or array of floats] Factors from source/receiver length and source strength.

empymod.utils.**get_off_ang**(*src*, *rec*, *nsrc*, *nrec*, *verb*)
Get depths, offsets, angles, hence spatial input parameters.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

src, rec [list of floats or arrays] Source/receiver dipole coordinates x, y, and z (m).

nsrc, nrec [int] Number of sources/receivers (-).

verb [{0, 1, 2, 3, 4}] Level of verbosity.

Returns

off [array of floats] Offsets

angle [array of floats] Angles

empymod.utils.**get_layer_nr**(*inp*, *depth*)
Get number of layer in which inp resides.

Note: If zinp is on a layer interface, the layer above the interface is chosen.

This check-function is called from one of the modelling routines in *empymod.model*. Consult these modelling routines for a detailed description of the input parameters.

Parameters

inp [list of floats or arrays] Dipole coordinates (m)

depth [array] Depths of layer interfaces.

Returns

linp [int or array_like of int] Layer number(s) in which inp resides (plural only if bipole).

zinp [float or array] inp[2] (depths).

empymod.utils.**printstartfinish**(*verb*, *inp=None*, *kcount=None*)
Print start and finish with time measure and kernel count.

empymod.utils.**conv_warning**(*conv*, *targ*, *name*, *verb*)
Print error if QWE/QUAD did not converge at least once.

empymod.utils.**set_minimum**(*min_freq=None*, *min_time=None*, *min_off=None*, *min_res=None*, *min_angle=None*)

Set minimum values of parameters.

The given parameters are set to its minimum value if they are smaller.

Note: set_minimum and get_minimum are derived after set_printoptions and get_printoptions from arrayprint.py in numpy.

Parameters

min_freq [float, optional] Minimum frequency [Hz] (default 1e-20 Hz).

min_time [float, optional] Minimum time [s] (default 1e-20 s).

min_off [float, optional] Minimum offset [m] (default 1e-3 m). Also used to round src- & rec-coordinates.

min_res [float, optional] Minimum horizontal and vertical resistivity [Ohm.m] (default 1e-20).

min_angle [float, optional] Minimum angle [-] (default 1e-10).

empymod.utils.**get_minimum**()

Return the current minimum values.

Note: set_minimum and get_minimum are derived after set_printoptions and get_printoptions from arrayprint.py in numpy.

Returns

min_vals [dict] Dictionary of current minimum values with keys

- min_freq : float
- min_time : float
- min_off : float
- min_res : float
- min_angle : float

For a full description of these options, see *set_minimum*.

class empymod.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*, *numba*, *empymod*, *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
```

5.11 Add-on functions

5.11.1 `empymod.scripts.fdesign` – Digital Linear Filter (DLF) design

The add-on fdesign can be used to design digital linear filters for the Hankel or Fourier transform, or for any linear transform ([Ghos70]). For this included or provided theoretical transform pairs can be used. Alternatively, one can use the EM modeller empymod to use the responses to an arbitrary 1D model as numerical transform pair.

More information can be found in the following places:

- The article about fdesign is in the repo https://github.com/empymod/article-fdesign
- Example notebooks to design a filter can be found in the repo https://empymod.readthedocs.io/en/stable/examples

This filter designing tool uses the direct matrix inversion method as described in [Kong07] and is based on scripts by [Key12]. The whole project of *fdesign* started with the Matlab scripts from Kerry Key, which he used to design his filters for [Key09], [Key12]. Fruitful discussions with Evert Slob and Kerry Key improved the add-on substantially.

Note that the use of empymod to create numerical transform pairs is, as of now, only implemented for the Hankel transform.

Implemented analytical transform pairs

The following tables list the transform pairs which are implemented by default. Any other transform pair can be provided as input. A transform pair is defined in the following way:

```
from empymod.scripts import fdesign

def my_tp_pair(var):
    '''My transform pair.'''

    def lhs(l):
        return func(l, var)

    def rhs(r):
        return func(r, var)

    return fdesign.Ghosh(name, lhs, rhs)
```

Here, *name* must be one of *j0*, *j1*, *sin*, or *cos*, depending what type of transform pair it is. Additional variables are provided with *var*. The evaluation points of the *lhs* are denoted by *l*, and the evaluation points of the *rhs* are denoted as *r*. As an example here the implemented transform pair *j0_1*

```
def j0_1(a=1):
    '''Hankel transform pair J0_1 ([Ande75]_).'''

    def lhs(l):
        return l*np.exp(-a*l**2)

    def rhs(r):
        return np.exp(-r**2/(4*a))/(2*a)

    return Ghosh('j0', lhs, rhs)
```

Implemented Hankel transforms

- *j0_1* [Ande75]

$$\int_0^\infty l \exp\left(-al^2\right) J_0(lr) dl = \frac{\exp\left(\frac{-r^2}{4a}\right)}{2a} \tag{5.15}$$

- *j0_2* [Ande75]

$$\int_0^\infty \exp\left(-al\right) J_0(lr) dl = \frac{1}{\sqrt{a^2+r^2}} \tag{5.16}$$

- *j0_3* [GuSi97]

$$\int_0^\infty l \exp\left(-al\right) J_0(lr) dl = \frac{a}{(a^2+r^2)^{3/2}} \tag{5.17}$$

- *j0_4* [ChCo82]

$$\int_0^\infty \frac{l}{\beta} \exp\left(-\beta z_v\right) J_0(lr) dl = \frac{\exp\left(-\gamma R\right)}{R} \tag{5.18}$$

- *j0_5* [ChCo82]

$$\int_0^\infty l \exp\left(-\beta z_v\right) J_0(lr) dl = \frac{z_v(\gamma R+1)}{R^3} \exp\left(-\gamma R\right) \tag{5.19}$$

- *j1_1* [Ande75]

$$\int_0^\infty l^2 \exp\left(-al^2\right) J_1(lr) dl = \frac{r}{4a^2} \exp\left(-\frac{r^2}{4a}\right) \tag{5.20}$$

- *j1_2* [Ande75]

$$\int_0^\infty \exp\left(-al\right) J_1(lr) dl = \frac{\sqrt{a^2+r^2}-a}{r\sqrt{a^2+r^2}} \tag{5.21}$$

- *j1_3* [Ande75]

$$\int_0^\infty l \exp\left(-al\right) J_1(lr) dl = \frac{r}{(a^2+r^2)^{3/2}} \tag{5.22}$$

- *j1_4* [ChCo82]

$$\int_0^\infty \frac{l^2}{\beta} \exp\left(-\beta z_v\right) J_1(lr) dl = \frac{r(\gamma R + 1)}{R^3} \exp\left(-\gamma R\right) \tag{5.23}$$

- *j1_5* [ChCo82]

$$\int_0^\infty l^2 \exp\left(-\beta z_v\right) J_1(lr) dl = \frac{r z_v(\gamma^2 R^2 + 3\gamma R + 3)}{R^5} \exp\left(-\gamma R\right) \tag{5.24}$$

Where

$$a > 0, r > 0$$

$$z_v = |z_{rec} - z_{src}|$$
$$R = \sqrt{r^2 + z_v^2}$$
$$\gamma = \sqrt{2j\pi\mu_0 f/\rho}$$
$$\beta = \sqrt{l^2 + \gamma^2}$$

Implemented Fourier transforms

- *sin_1* [Ande75]

$$\int_0^\infty l \exp\left(-a^2 l^2\right) \sin(lr) dl = \frac{\sqrt{\pi} r}{4a^3} \exp\left(-\frac{r^2}{4a^2}\right) \tag{5.25}$$

- *sin_2* [Ande75]

$$\int_0^\infty \exp\left(-al\right) \sin(lr) dl = \frac{r}{a^2 + r^2} \tag{5.26}$$

- *sin_3* [Ande75]

$$\int_0^\infty \frac{l}{a^2 + l^2} \sin(lr) dl = \frac{\pi}{2} \exp\left(-ar\right) \tag{5.27}$$

- *cos_1* [Ande75]

$$\int_0^\infty \exp\left(-a^2 l^2\right) \cos(lr) dl = \frac{\sqrt{\pi}}{2a} \exp\left(-\frac{r^2}{4a^2}\right) \tag{5.28}$$

- *cos_2* [Ande75]

$$\int_0^\infty \exp\left(-al\right) \cos(lr) dl = \frac{a}{a^2 + r^2} \tag{5.29}$$

- *cos_3* [Ande75]

$$\int_0^\infty \frac{1}{a^2 + l^2} \cos(lr) dl = \frac{\pi}{2a} \exp\left(-ar\right) \tag{5.30}$$

empymod.scripts.fdesign.**design**(*n*, *spacing*, *shift*, *fI*, *fC=False*, *r=None*, *r_def=(1, 1, 2)*, *reim=None*, *cvar='amp'*, *error=0.01*, *name=None*, *full_output=False*, *finish=False*, *save=True*, *path='filters'*, *verb=2*, *plot=1*)

Digital linear filter (DLF) design

This routine can be used to design digital linear filters for the Hankel or Fourier transform, or for any linear transform ([Ghos70]). For this included or provided theoretical transform pairs can be used. Alternatively, one can use the EM modeller empymod to use the responses to an arbitrary 1D model as numerical transform pair.

This filter designing tool uses the direct matrix inversion method as described in [Kong07] and is based on scripts by [Key12]. The tool is an add-on to the electromagnetic modeller empymod [Wert17]. Fruitful discussions with Evert Slob and Kerry Key improved the add-on substantially.

Example notebooks of its usage can be found in the documentation-gallery, https://empymod.readthedocs.io/en/stable/examples

Parameters

n [int] Filter length.

spacing: float or tuple (start, stop, num) Spacing between filter points. If tuple, it corresponds to the input for np.linspace with endpoint=True.

shift: float or tuple (start, stop, num) Shift of base from zero. If tuple, it corresponds to the input for np.linspace with endpoint=True.

fI, fC [transform pairs] Theoretical or numerical transform pair(s) for the inversion (I) and for the check of goodness (fC). fC is optional. If not provided, fI is used for both fI and fC.

r [array, optional] Right-hand side evaluation points for the check of goodness (fC). Defaults to r = np.logspace(0, 5, 1000), which are a lot of evaluation points, and depending on the transform pair way too long r's.

r_def [tuple (add_left, add_right, factor), optional] Definition of the right-hand side evaluation points r of the inversion. r is derived from the base values, default is (1, 1, 2).

- rmin = log10(1/max(base)) - add_left
- rmax = log10(1/min(base)) + add_right
- r = logspace(rmin, rmax, factor*n)

reim [np.real or np.imag, optional] Which part of complex transform pairs is used for the inversion. Defaults to np.real.

cvar [string {'amp', 'r'}, optional] If 'amp', the inversion minimizes the amplitude. If 'r', the inversion maximizes the right-hand side evaluation point r. Defaults is 'amp'.

error [float, optional] Up to which relative error the transformation is considered good in the evaluation of the goodness. Default is 0.01 (1 %).

name [str, optional] Name of the filter. Defaults to dlf_+str(n).

full_output [bool, optional] If True, returns best filter and output from scipy.optimize.brute; else only filter. Default is False.

finish [None, True, or callable, optional] If callable, it is passed through to scipy.optimize.brute: minimization function to find minimize best result from brute-force approach. Default is None. You can simply provide True in order to use scipy.optimize.fmin_powell(). Set this to None if you are only interested in the actually provided spacing/shift-values.

save [bool, optional] If True, best filter is saved to plain text files in ./filters/. Can be loaded with fdesign.load_filter(name). If full, the inversion output is stored too. You

can add '.gz' to *name*, which will then save the full inversion output in a compressed file instead of plain text.

path [string, optional] Absolute or relative path where output will be saved if *save=True*. Default is 'filters'.

verb [{0, 1, 2}, optional] Level of verbosity, default is 2:

- 0: Print nothing.
- 1: Print warnings.
- 2: Print additional time, progress, and result

plot [{0, 1, 2, 3}, optional] Level of plot-verbosity, default is 1:

- 0: Plot nothing.
- 1: Plot brute-force result
- 2: Plot additional theoretical transform pairs, and best inv.
- 3: Plot additional inversion result (can result in lots of plots depending on spacing and shift) If you are using a notebook, use %matplotlib notebook to have all inversion results appear in the same plot.

Returns

filter [empymod.filter.DigitalFilter instance] Best filter for the input parameters.

full [tuple] Output from scipy.optimize.brute with full_output=True. (Returned when *full_output* is True.)

empymod.scripts.fdesign.**save_filter**(*name*, *filt*, *full=None*, *path='filters'*)
Save DLF-filter and inversion output to plain text files.

empymod.scripts.fdesign.**load_filter**(*name*, *full=False*, *path='filters'*, *filter_coeff=None*)
Load saved DLF-filter and inversion output from text files.

empymod.scripts.fdesign.**plot_result**(*filt*, *full*, *prntres=True*)
QC the inversion result.

Parameters

- filt, full as returned from fdesign.design with full_output=True

- If prntres is True, it calls fdesign.print_result as well.

empymod.scripts.fdesign.**print_result**(*filt*, *full=None*)
Print best filter information.

Parameters

- filt, full as returned from fdesign.design with full_output=True

class empymod.scripts.fdesign.**Ghosh**(*name*, *lhs*, *rhs*)
Simple Class for Theoretical Transform Pairs.

Named after D. P. Ghosh, honouring his 1970 Ph.D. thesis with which he introduced the digital filter method to geophysics ([Ghos70]).

empymod.scripts.fdesign.**j0_1**(*a=1*)
Hankel transform pair J0_1 ([Ande75]).

empymod.scripts.fdesign.**j0_2**(*a=1*)
Hankel transform pair J0_2 ([Ande75]).

empymod.scripts.fdesign.**j0_3**(*a=1*)
Hankel transform pair J0_3 ([GuSi97]).

empymod.scripts.fdesign.**j0_4**(*f=1*, *rho=0.3*, *z=50*)
Hankel transform pair J0_4 ([ChCo82]).

Parameters

f [float] Frequency (Hz)

rho [float] Resistivity (Ohm.m)

z [float] Vertical distance between source and receiver (m)

empymod.scripts.fdesign.**j0_5**(*f=1*, *rho=0.3*, *z=50*)
Hankel transform pair J0_5 ([ChCo82]).

Parameters

f [float] Frequency (Hz)

rho [float] Resistivity (Ohm.m)

z [float] Vertical distance between source and receiver (m)

empymod.scripts.fdesign.**j1_1**(*a=1*)
Hankel transform pair J1_1 ([Ande75]).

empymod.scripts.fdesign.**j1_2**(*a=1*)
Hankel transform pair J1_2 ([Ande75]).

empymod.scripts.fdesign.**j1_3**(*a=1*)
Hankel transform pair J1_3 ([Ande75]).

empymod.scripts.fdesign.**j1_4**(*f=1*, *rho=0.3*, *z=50*)
Hankel transform pair J1_4 ([ChCo82]).

Parameters

f [float] Frequency (Hz)

rho [float] Resistivity (Ohm.m)

z [float] Vertical distance between source and receiver (m)

empymod.scripts.fdesign.**j1_5**(*f=1*, *rho=0.3*, *z=50*)
Hankel transform pair J1_5 ([ChCo82]).

Parameters

f [float] Frequency (Hz)

rho [float] Resistivity (Ohm.m)

z [float] Vertical distance between source and receiver (m)

empymod.scripts.fdesign.**sin_1**(*a=1*, *inverse=False*)
Fourier sine transform pair sin_1 ([Ande75]).

empymod.scripts.fdesign.**sin_2**(*a=1*, *inverse=False*)
Fourier sine transform pair sin_2 ([Ande75]).

empymod.scripts.fdesign.**sin_3**(*a=1*, *inverse=False*)
Fourier sine transform pair sin_3 ([Ande75]).

empymod.scripts.fdesign.**cos_1**(*a=1*, *inverse=False*)
Fourier cosine transform pair cos_1 ([Ande75]).

empymod.scripts.fdesign.**cos_2**(*a=1*, *inverse=False*)
Fourier cosine transform pair cos_2 ([Ande75]).

empymod.scripts.fdesign.**cos_3**(*a=1*, *inverse=False*)
Fourier cosine transform pair cos_3 ([Ande75]).

empymod.scripts.fdesign.**empy_hankel**(*ftype*, *zsrc*, *zrec*, *res*, *freqtime*, *depth=None*, *aniso=None*, *epermH=None*, *epermV=None*, *mpermH=None*, *mpermV=None*, *htarg=None*, *verblhs=0*, *verbrhs=0*)

Numerical transform pair with empymod.

All parameters except *ftype*, *verblhs*, and *verbrhs* correspond to the input parameters to *empymod.dipole*. See there for more information.

Note that if depth=None or [], the analytical full-space solutions will be used (much faster).

Parameters

ftype [str or list of strings] Either of: {'j0', 'j1', 'j2', ['j0', 'j1']}

- *'j0'*: Analyze J0-term with ab=11, angle=45°
- *'j1'*: Analyze J1-term with ab=31, angle=0°
- *'j2'*: Analyze J0- and J1-terms jointly with ab=12, angle=45°
- *['j0', 'j1']* : Same as calling empy_hankel twice, once with 'j0' and one with 'j1'; can be provided like this to fdesign.design.

Note that ftype='j2' only works for fC, not for fI.

verblhs, verbrhs: int verb-values provided to empymod for lhs and rhs.

5.11.2 empymod.scripts.tmtemod – Calculate up- and down-going TM and TE modes

This add-on for empymod adjusts [HuTS15] for TM/TE-split. The development was initiated by the development of https://github.com/empymod/csem-ziolkowski-and-slob ([ZiSl19]).

This is a stripped-down version of empymod with a lot of simplifications but an important addition. The modeller empymod returns the total field, hence not distinguishing between TM and TE mode, and even less between up- and down-going fields. The reason behind this is simple: The derivation of [HuTS15], on which empymod is based, returns the total field. In this derivation each mode (TM and TE) contains non-physical contributions. The non-physical contributions have opposite signs in TM and TE, so they cancel each other out in the total field. However, in order to obtain the correct TM and TE contributions one has to remove these non-physical parts.

This is what this routine does, but only for an x-directed electric source with an x-directed electric receiver, and in the frequency domain (src and rec in same layer). This version of `empymod.dipole()` returns the signal separated into TM++, TM+-, TM-+, TM–, TE++, TE+-, TE-+, and TE– as well as the direct field TM and TE contributions. The first superscript denotes the direction in which the field diffuses towards the receiver and the second superscript denotes the direction in which the field diffuses away from the source. For both the plus-sign indicates the field diffuses in the downward direction and the minus-sign indicates the field diffuses in the upward direction. It uses empymod wherever possible. See the corresponding functions in empymod for more explanation and documentation regarding input parameters. There are important limitations:

- *ab* == 11 [=> x-directed el. source & el. receivers]
- *signal* == None [=> only frequency domain]
- *xdirect* == False [=> direct field calc. in wavenr-domain]
- *ht* == 'dlf'
- *htarg* == 'key_201_2012'
- Options *ft*, *ftarg*, and *loop* are not available.
- *lsrc* == *lrec* [=> src & rec are assumed in same layer!]
- Model must have more than 1 layer
- Electric permittivity and magnetic permeability are isotropic.

- Only one frequency at once.

Theory

The derivation of [HuTS15], on which empymod is based, returns the total field. Internally it also calculates TM and TE modes, and sums these up. However, the separation into TM and TE mode introduces a singularity at $\kappa = 0$. It has no contribution in the space-frequency domain to the total fields, but it introduces non-physical events in each mode with opposite sign (so they cancel each other out in the total field). In order to obtain the correct TM and TE contributions one has to remove these non-physical parts.

This routine removes the non-physical part. It is basically a heavily simplified version of empymod with the following limitations outlined above.

It returns the signal separated into TM++, TM+-, TM-+, TM–, TE++, TE+-, TE-+, and TE– as well as the direct field TM and TE contributions. The first superscript denotes the direction in which the field diffuses towards the receiver and the second superscript denotes the direction in which the field diffuses away from the source. For both the plus-sign indicates the field diffuses in the downward direction and the minus-sign indicates the field diffuses in the upward direction. The routine uses empymod wherever possible, see the corresponding functions in empymod for more explanation and documentation regarding input parameters.

Please note that the notation in [HuTS15] differs from the notation in [ZiSl19]. I specify therefore always, which notification applies, either *Hun15* or *Zio19*.

We start with equation (105) in *Hun15*:

$$\hat{G}^{ee}_{xx}(\boldsymbol{x}, \boldsymbol{x'}, \omega) = \hat{G}^{ee;i}_{xx;s}(\boldsymbol{x} - \boldsymbol{x'}, \omega) + \frac{1}{8\pi}\int_{\kappa=0}^{\infty}\left(\frac{\Gamma_s \tilde{g}^{tm}_{hh;s}}{\eta_s} - \frac{\zeta_s \tilde{g}^{te}_{zz;s}}{\bar{\Gamma}_s}\right) J_0(\kappa r)\kappa d\kappa \tag{5.31}$$

$$-\frac{\cos(2\phi)}{8\pi}\int_{\kappa=0}^{\infty}\left(\frac{\Gamma_s \tilde{g}^{tm}_{hh;s}}{\eta_s} + \frac{\zeta_s \tilde{g}^{te}_{zz;s}}{\bar{\Gamma}_s}\right) J_2(\kappa r)\kappa d\kappa. \tag{5.32}$$

Ignoring the incident field, and using $J_2 = \frac{2}{\kappa r}J_1 - J_0$ to avoid J_2-integrals, we get

$$\hat{G}^{ee}_{xx}(\boldsymbol{x}, \boldsymbol{x'}, \omega) = \frac{1}{8\pi}\int_{\kappa=0}^{\infty}\left(\frac{\Gamma_s \tilde{g}^{tm}_{hh;s}}{\eta_s} - \frac{\zeta_s \tilde{g}^{te}_{zz;s}}{\bar{\Gamma}_s}\right) J_0(\kappa r)\kappa \,\mathrm{d}\kappa \tag{5.33}$$

$$+\frac{\cos(2\phi)}{8\pi}\int_{\kappa=0}^{\infty}\left(\frac{\Gamma_s \tilde{g}^{tm}_{hh;s}}{\eta_s} + \frac{\zeta_s \tilde{g}^{te}_{zz;s}}{\bar{\Gamma}_s}\right) J_0(\kappa r)\kappa \,\mathrm{d}\kappa \tag{5.34}$$

$$-\frac{\cos(2\phi)}{4\pi r}\int_{\kappa=0}^{\infty}\left(\frac{\Gamma_s \tilde{g}^{tm}_{hh;s}}{\eta_s} + \frac{\zeta_s \tilde{g}^{te}_{zz;s}}{\bar{\Gamma}_s}\right) J_1(\kappa r) \,\mathrm{d}\kappa. \tag{5.35}$$

From this the TM- and TE-parts follow as

$$\mathrm{TE} = \frac{\cos(2\phi) - 1}{8\pi}\int_{\kappa=0}^{\infty}\frac{\zeta_s \tilde{g}^{te}_{zz;s}}{\bar{\Gamma}_s} J_0(\kappa r)\kappa \,\mathrm{d}\kappa - \frac{\cos(2\phi)}{4\pi r}\int_{\kappa=0}^{\infty}\frac{\zeta_s \tilde{g}^{te}_{zz;s}}{\bar{\Gamma}_s} J_1(\kappa r) \,\mathrm{d}\kappa, \tag{5.36}$$

$$\mathrm{TM} = \frac{\cos(2\phi) + 1}{8\pi}\int_{\kappa=0}^{\infty}\frac{\Gamma_s \tilde{g}^{tm}_{hh;s}}{\eta_s} J_0(\kappa r)\kappa \,\mathrm{d}\kappa - \frac{\cos(2\phi)}{4\pi r}\int_{\kappa=0}^{\infty}\frac{\Gamma_s \tilde{g}^{tm}_{hh;s}}{\eta_s} J_1(\kappa r) \,\mathrm{d}\kappa. \tag{5.37}$$

Equations (108) and (109) in Hun15 yield the required parameters $\tilde{g}^{tm}_{hh;s}$ and $\tilde{g}^{te}_{zz;s}$,

$$\tilde{g}^{tm}_{hh;s} = P^{u-}_s W^u_s + P^{d-}_s W^d_s, \tag{5.38}$$

$$\tilde{g}^{te}_{zz;s} = \bar{P}^{u+}_s \bar{W}^u_s + \bar{P}^{d+}_s \bar{W}^d_s\,. \tag{5.39}$$

The parameters $P^{u\pm}_s$ and $P^{d\pm}_s$ are given in equations (81) and (82), $\bar{P}^{u\pm}_s$ and $\bar{P}^{d\pm}_s$ in equations (A-8) and (A-9); W^u_s and W^d_s in equation (74) in Hun15. This yields

$$\tilde{g}^{te}_{zz;s} = \frac{\bar{R}^+_s}{\bar{M}_s}\left\{\exp[-\bar{\Gamma}_s(z_s - z + d^+)] + \bar{R}^-_s \exp[-\bar{\Gamma}_s(z_s - z + d_s + d^-)]\right\} \tag{5.40}$$

$$+\frac{\bar{R}_s^-}{\bar{M}_s}\left\{\exp[-\bar{\Gamma}_s(z-z_{s-1}+d^-)]+\bar{R}_s^+\exp[-\bar{\Gamma}_s(z-z_{s-1}+d_s+d^+)]\right\}, \tag{5.41}$$

$$=\frac{\bar{R}_s^+}{\bar{M}_s}\left\{\exp[-\bar{\Gamma}_s(2z_s-z-z')]+\bar{R}_s^-\exp[-\bar{\Gamma}_s(z'-z+2d_s)]\right\} \tag{5.42}$$

$$+\frac{\bar{R}_s^-}{\bar{M}_s}\left\{\exp[-\bar{\Gamma}_s(z+z'-2z_{s-1})]+\bar{R}_s^+\exp[-\bar{\Gamma}_s(z-z'+2d_s)]\right\}, \tag{5.43}$$

where $d^{\pm}$ is taken from the text below equation (67). There are four terms in the right-hand side, two in the first line and two in the second line. The first term in the first line is the integrand of TE+-, the second term in the first line corresponds to TE++, the first term in the second line is TE-+, and the second term in the second line is TE–.

If we look at TE+-, we have

$$\tilde{g}^{te+-}_{zz;s}=\frac{\bar{R}_s^+}{\bar{M}_s}\exp[-\bar{\Gamma}_s(2z_s-z-z')], \tag{5.44}$$

and therefore

$$\mathrm{TE}^{+-}=\frac{\cos(2\phi)-1}{8\pi}\int_{\kappa=0}^{\infty}\frac{\zeta_s\bar{R}_s^+}{\bar{\Gamma}_s\bar{M}_s}\exp[-\bar{\Gamma}_s(2z_s-z-z')]J_0(\kappa r)\kappa\,\mathrm{d}\kappa \tag{5.45}$$

$$-\frac{\cos(2\phi)}{4\pi r}\int_{\kappa=0}^{\infty}\frac{\zeta_s\bar{R}_s^+}{\bar{\Gamma}_s\bar{M}_s}\exp[-\bar{\Gamma}_s(2z_s-z-z')]J_1(\kappa r)\,\mathrm{d}\kappa. \tag{5.46}$$

We can compare this to equation (4.165) in Zio19, with $\hat{I}_x^e=1$ and slightly re-arranging it to look more alike, we get

$$\hat{E}^{+-}_{xx;H}=\frac{y^2}{4\pi r^2}\int_{\kappa=0}^{\infty}\frac{\zeta_1}{\Gamma_1}\frac{R^-_{H;1}}{M_{H;1}}\exp(-\Gamma_1 h^{+-})J_0(\kappa r)\kappa d\kappa \tag{5.47}$$

$$:label:hun18$$

$$+\frac{x^2-y^2}{4\pi r^3}\int_{\kappa=0}^{\infty}\frac{\zeta_1}{\Gamma_1}\left(\frac{R^-_{H;1}}{M_{H;1}}-\frac{R^-_{H;1}(\kappa=0)}{M_{H;1}(\kappa=0)}\right)\exp(-\Gamma_1 h^{+-})J_1(\kappa r)d\kappa$$

$$-\frac{\zeta_1(x^2-y^2)}{4\pi\gamma_1 r^4}\frac{R^-_{H;1}(\kappa=0)}{M_{H;1}(\kappa=0)}\exp(-\gamma_1 R^{+-}). \tag{5.48}$$

The notation in this equation follows Zio19.

The difference between the two previous equations is that the first one contains non-physical contributions. These have opposite signs in TM+- and TE+-, and therefore cancel each other out. But if we want to know the specific contributions from TM and TE we have to remove them. The non-physical contributions only affect the J_1-integrals, and only for $\kappa=0$.

The following lists for all 8 cases the term that has to be removed, in the notation of Zio19 (for the notation as in Hun15 see the implementation in this file):

$$TE^{++}=+\frac{\zeta_1(x^2-y^2)}{4\pi\gamma_1 r^4}\frac{\exp(-\gamma_1|h^-|)}{M_{H;1}(\kappa=0)}, \tag{5.49}$$

$$TE^{-+}=-\frac{\zeta_1(x^2-y^2)}{4\pi\gamma_1 r^4}\frac{R^+_{H;1}(\kappa=0)\exp(-\gamma_1 h^{-+})}{M_{H;1}(\kappa=0)}, \tag{5.50}$$

$$TE^{+-}=-\frac{\zeta_1(x^2-y^2)}{4\pi\gamma_1 r^4}\frac{R^-_{H;1}(\kappa=0)\exp(-\gamma_1 h^{+-})}{M_{H;1}(\kappa=0)}, \tag{5.51}$$

$$TE^{--}=+\frac{\zeta_1(x^2-y^2)}{4\pi\gamma_1 r^4}\frac{R^+_{H;1}(\kappa=0)R^-_{H;1}(\kappa=0)\exp(-\gamma_1 h^{--})}{M_{H;1}(\kappa=0)}, \tag{5.52}$$

$$TM^{++}=-\frac{\zeta_1(x^2-y^2)}{4\pi\gamma_1 r^4}\frac{\exp(-\gamma_1|h^-|)}{M_{V;1}(\kappa=0)}, \tag{5.53}$$

$$TM^{-+} = -\frac{\zeta_1(x^2-y^2)}{4\pi\gamma_1 r^4}\frac{R^+_{V;1}(\kappa=0)\exp(-\gamma_1 h^{-+})}{M_{V;1}(\kappa=0)}, \tag{5.54}$$

$$TM^{+-} = -\frac{\zeta_1(x^2-y^2)}{4\pi\gamma_1 r^4}\frac{R^-_{V;1}(\kappa=0)\exp(-\gamma_1 h^{+-})}{M_{V;1}(\kappa=0)}, \tag{5.55}$$

$$TM^{--} = -\frac{\zeta_1(x^2-y^2)}{4\pi\gamma_1 r^4}\frac{R^+_{V;1}(\kappa=0)R^-_{V;1}(\kappa=0)\exp(-\gamma_1 h^{--})}{M_{V;1}(\kappa=0)}. \tag{5.56}$$

Note that in the first and fourth equations the correction terms have opposite sign as those in the fifth and eighth equations because at $\kappa = 0$ the TM and TE mode correction terms are equal. Also note that in the second and third equations the correction terms have the same sign as those in the sixth and seventh equations because at $\kappa = 0$ the TM and TE mode reflection responses in those terms are equal but with opposite sign: $R^{\pm}_{V;1}(\kappa=0) = -R^{\pm}_{V;1}(\kappa=0)$.

Hun15 uses ϕ, whereas Zio19 uses x, y, for which we can use

$$\cos(2\phi) = -\frac{x^2-y^2}{r^2}\ . \tag{5.57}$$

empymod.scripts.tmtemod.**dipole**(*src*, *rec*, *depth*, *res*, *freqtime*, *aniso=None*, *eperm=None*, *mperm=None*, *verb=2*)

Return the electromagnetic field due to a dipole source.

This is a modified version of `empymod.model.dipole()`. It returns the separated contributions of TM–, TM-+, TM+-, TM++, TMdirect, TE–, TE-+, TE+-, TE++, and TEdirect.

Parameters

src, rec [list of floats or arrays] Source and receiver coordinates (m): [x, y, z]. The x- and y-coordinates can be arrays, z is a single value. The x- and y-coordinates must have the same dimension.

Sources or receivers placed on a layer interface are considered in the upper layer.

Sources and receivers must be in the same layer.

depth [list] Absolute layer interfaces z (m); #depth = #res - 1 (excluding +/- infinity).

res [array_like] Horizontal resistivities rho_h (Ohm.m); #res = #depth + 1.

freqtime [float] Frequency f (Hz). (The name *freqtime* is kept for consistency with `empymod.model.dipole()`. Only one frequency at once.

aniso [array_like, optional] Anisotropies lambda = sqrt(rho_v/rho_h) (-); #aniso = #res. Defaults to ones.

eperm [array_like, optional] Relative electric permittivities epsilon (-); #eperm = #res. Default is ones.

mperm [array_like, optional] Relative magnetic permeabilities mu (-); #mperm = #res. Default is ones.

verb [{0, 1, 2, 3, 4}, optional] Level of verbosity, default is 2:

- 0: Print nothing.
- 1: Print warnings.
- 2: Print additional runtime and kernel calls
- 3: Print additional start/stop, condensed parameter information.
- 4: Print additional full parameter information

Returns

TM, TE [list of ndarrays, (nfreq, nrec, nsrc)] Frequency-domain EM field [V/m], separated into TM = [TM–, TM-+, TM+-, TM++, TMdirect] and TE = [TE–, TE-+, TE+-, TE++, TEdirect].

However, source and receiver are normalised. So the source strength is 1 A and its length is 1 m. Therefore the electric field could also be written as [V/(A.m2)].

The shape of EM is (nfreq, nrec, nsrc). However, single dimensions are removed.

Bibliography

[Ande75] Anderson, W. L., 1975, Improved digital filters for evaluating Fourier and Hankel transform integrals: USGS, PB242800; pubs.er.usgs.gov/publication/70045426.

[Ande79] Anderson, W. L., 1979, Numerical integration of related Hankel transforms of orders 0 and 1 by adaptive digital filtering: Geophysics, 44, 1287–1305; DOI: 10.1190/1.1441007.

[Ande82] Anderson, W. L., 1982, Fast Hankel transforms using related and lagged convolutions: ACM Trans. on Math. Softw. (TOMS), 8, 344–368; DOI: 10.1145/356012.356014.

[ChCo82] Chave, A. D., and C. S. Cox, 1982, Controlled electromagnetic sources for measuring electrical conductivity beneath the oceans: 1. forward problem and model study: Journal of Geophysical Research, 87, 5327–5338; DOI: 10.1029/JB087iB07p05327.

[Ghos70] Ghosh, D. P., 1970, The application of linear filter theory to the direct interpretation of geoelectrical resistivity measurements: Ph.D. Thesis, TU Delft; UUID: 88a568bb-ebee-4d7b-92df-6639b42da2b2.

[GuSi97] Guptasarma, D., and B. Singh, 1997, New digital linear filters for Hankel J0 and J1 transforms: Geophysical Prospecting, 45, 745–762; DOI: 10.1046/j.1365-2478.1997.500292.x.

[HaJo88] Haines, G. V., and A. G. Jones, 1988, Logarithmic Fourier transformation: Geophysical Journal, 92, 171–178; DOI: 10.1111/j.1365-246X.1988.tb01131.x.

[Hami00] Hamilton, A. J. S., 2000, Uncorrelated modes of the non-linear power spectrum: Monthly Notices of the Royal Astronomical Society, 312, pages 257–284; DOI: 10.1046/j.1365-8711.2000.03071.x; Website of FFTLog: casa.colorado.edu/~ajsh/FFTLog.

[HuTS15] Hunziker, J., J. Thorbecke, and E. Slob, 2015, The electromagnetic response in a layered vertical transverse isotropic medium: A new look at an old problem: Geophysics, 80(1), F1–F18; DOI: 10.1190/geo2013-0411.1; Software: software.seg.org/2015/0001.

[Key09] Key, K., 2009, 1D inversion of multicomponent, multifrequency marine CSEM data: Methodology and synthetic studies for resolving thin resistive layers: Geophysics, 74(2), F9–F20; DOI: 10.1190/1.3058434. Software: marineemlab.ucsd.edu/Projects/Occam/1DCSEM.

[Key12] Key, K., 2012, Is the fast Hankel transform faster than quadrature?: Geophysics, 77(3), F21–F30; DOI: 10.1190/geo2011-0237.1; Software: software.seg.org/2012/0003.

[Kong07] Kong, F. N., 2007, Hankel transform filters for dipole antenna radiation in a conductive medium: Geophysical Prospecting, 55, 83–89; DOI: 10.1111/j.1365-2478.2006.00585.x.

[Shan55] Shanks, D., 1955, Non-linear transformations of divergent and slowly convergent sequences: Journal of Mathematics and Physics, 34, 1–42; DOI: 10.1002/sapm19553411.

[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.

[Talm78] Talman, J. D., 1978, Numerical Fourier and Bessel transforms in logarithmic variables: Journal of Computational Physics, 29, pages 35–48; DOI: 10.1016/0021-9991(78)90107-9.

[Tref00] Trefethen, L. N., 2000, Spectral methods in MATLAB: Society for Industrial and Applied Mathematics (SIAM), volume 10 of Software, Environments, and Tools, chapter 12, page 129; DOI: 10.1137/1.9780898719598.ch12.

[Weni89] Weniger, E. J., 1989, Nonlinear sequence transformations for the acceleration of convergence and the summation of divergent series: Computer Physics Reports, 10, 189–371; arXiv: abs/math/0306302.

[Wert17] Werthmüller, D., 2017, An open-source full 3D electromagnetic modeler for 1D VTI media in Python: empymod: Geophysics, 82(6), WB9–WB19; DOI: 10.1190/geo2016-0626.1.

[Wert17b] Werthmüller, D., 2017, Getting started with controlled-source electromagnetic 1D modeling: The Leading Edge, 36, 352–355; DOI: 10.1190/tle36040352.1.

[Wynn56] Wynn, P., 1956, On a device for computing the $e_m(S_n)$ tranformation: Math. Comput., 10, 91–96; DOI: 10.1090/S0025-5718-1956-0084056-6.

[ZiSl19] Ziolkowski, A., and E. Slob, 2019, Introduction to Controlled-Source Electromagnetic Methods: Cambridge University Press; ISBN: 9781107058620.

Python Module Index

e

Index