

---

# **empymod Documentation**

***Release 1.5.2***

**Dieter Werthmüller**

**26 April 2018**



---

## Contents

---

<b>1</b>	<b>More information</b>	<b>3</b>
<b>2</b>	<b>Citation</b>	<b>5</b>
<b>3</b>	<b>License information</b>	<b>7</b>
3.1	Manual . . . . .	7
3.2	Roadmap . . . . .	14
3.3	Changelog . . . . .	15
3.4	Credits . . . . .	18
3.5	Code . . . . .	19
	<b>Bibliography</b>	<b>51</b>
	<b>Python Module Index</b>	<b>53</b>



Version: 1.5.2 ~ Date: 26 April 2018

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 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.

See <https://empymod.github.io/#features> for a complete list of features.



# CHAPTER 1

---

## More information

---

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

- **Website:** <https://empymod.github.io>,
- **Documentation empymod:** <https://empymod.readthedocs.io>,
- **Documentation add-ons:** <https://empyscripts.readthedocs.io>,
- **Source Code:** <https://github.com/empymod>,
- **Examples:** <https://github.com/empymod/example-notebooks>.





## CHAPTER 2

---

### Citation

---

If you publish results for which you used empymod, please give credit by citing this article:

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](https://doi.org/10.1190/geo2016-0626.1).

All releases have a Zenodo-DOI, provided on the [release-page](#). Also consider citing [Hunziker et al. \(2015\)](#) and [Key \(2012\)](#), without which empymod would not exist.



---

## License information

---

Copyright 2016-2018 Dieter Werthmüller

Licensed under the Apache License, Version 2.0. See the `LICENSE-` and `NOTICE-`files or the documentation for more information.

## 3.1 Manual

### 3.1.1 Theory

The code is principally based on

- [\[Hunziker\\_et\\_al\\_2015\]](#) for the wavenumber-domain calculation (`kernel`),
- [\[Key\\_2012\]](#) for the DLF and QWE transforms,
- [\[Slob\\_et\\_al\\_2010\]](#) for the analytical half-space solutions, and
- [\[Hamilton\\_2000\]](#) 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 [\[Ziolkowski\\_and\\_Slob\]](#), which will be published in late 2018. The book derives in great detail the equations for layered-Earth CSEM modelling.

### 3.1.2 Installation

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

```
conda install -c prisae empymod
```

or via `pip`:

```
pip install empymod
```

Required are Python version 3.4 or higher and the modules `NumPy` and `SciPy`. The module `numexpr` is required additionally (built with Intel's VML) if you want to run parts of the kernel in parallel.

The modeller `empymod` comes with add-ons (`empyscripts`). These add-ons provide some very specific, additional functionalities. To install them just follow the instructions for `empymod`, replacing `empymod`

with `empyscripts` in the command. You can find more information regarding the add-ons directly on [github.com/empymod/empyscripts](https://github.com/empymod/empyscripts).

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](https://www.anaconda.com/download). If you install [Anaconda](<https://www.anaconda.com/download>). If you install Anaconda you can simply start the *Anaconda Navigator*, add the channel `prisae` and both `empymod` and `empyscripts` will appear in the package list and can be installed with a click.

**Warning:** Do not use `scipy == 0.19.0`. It has a memory leak in `quad`, see [github.com/scipy/scipy/pull/7216](https://github.com/scipy/scipy/pull/7216). So if you use QUAD (or potentially QWE) in any of your transforms you might see your memory usage going through the roof.

The structure of `empymod` is:

- **model.py**: EM modelling routines.
- **utils.py**: Utilities for `model` such as checking input parameters.
- **kernel.py**: Kernel of `empymod`, calculates 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).

### 3.1.3 Usage/Examples

A good starting point is [\[Werthmuller\\_2017b\]](#), and more information can be found in [\[Werthmuller\\_2017\]](#). There are a lot of examples of its usage available, in the form of Jupyter notebooks. Have a look at the following repositories:

- Example notebooks: <https://github.com/empymod/example-notebooks>,
- Geophysical Tutorial TLE: <https://github.com/empymod/article-tle2017>, and
- Numerical examples of [\[Ziolkowski\\_and\\_Slob\]](#): <https://github.com/empymod/csem-ziolkowski-and-slob>.

The main modelling routines is `bipole`, which can calculate 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 numpy as np
>>> from empymod import bipole
>>> # 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
>>> # Calculate electric field due to an electric source at 1 Hz.
>>> # [msrc = mrec = True (default)]
>>> EMfield = bipole(src, rec, depth, res, freq, verb=4)
:: empymod START ::
```

```

~
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
frequency  [Hz] : 1
Hankel      : DLF (Fast Hankel Transform)
  > Filter   : Key 201 (2009)
  > pts_per_dec : Defined by filter (lagged)
Hankel Opt. : None
Loop over   : None (all vectorized)
Source(s)   : 1 bipole(s)
  > intpts    : 1 (as dipole)
  > length    [m] : 100
  > 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]

```

### 3.1.4 Contributing

New contributions, bug reports, or any kind of feedback is always welcomed! Have a look at the Roadmap-section 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 either add the path to empymod to your python-path variable, or install it in your python distribution via:

```
python setup.py install
```

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

The 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
> # and then
> cd to/the/empymod/folder # Ensure you are in the right directory,

```

```
> ls -d */                                # your output should look the same.
docs/  empymod/  tests/
> # pytest will find the tests, which are located in the tests-folder.
> # simply run
> pytest --cov=empymod --flake8
```

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.

### 3.1.5 Transforms

Included **Hankel transforms**:

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

Included **Fourier transforms**:

- Digital Linear Filters *DLF*
- Quadrature with Extrapolation *QWE*
- Logarithmic Fast Fourier Transform *FFTLog*
- Fast Fourier Transform *FFT*

#### FFTLog

FFTLog is the logarithmic analogue to the Fast Fourier Transform FFT originally proposed by [Talman\_1978]. The code used by `empymod` was published in Appendix B of [Hamilton\_2000] and is publicly available at [casa.colorado.edu/~ajsh/FFTLog](http://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](https://github.com/prisae/pyfftlog)).

[Haines\_and\_Jones\_1988] proposed a logarithmic Fourier transform (abbreviated by the authors as LFT) for electromagnetic geophysics, also based on [Talman\_1978]. 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 [Anderson\_1982], and Haines and Jones cite [Anderson\_1979]. 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 ([Anderson\_1975]). 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.

## Notes on Fourier Transform

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

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

see, e.g., [Anderson\_1975] or [Key\_2012]. 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  $i\omega$ , 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,

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

Using  $\frac{\partial}{\partial t} \Leftrightarrow i\omega$  and going the other way, from impulse to step, leads to the division by  $i\omega$ . (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='ffht'/'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 know 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

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

and the step-off by

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

### 3.1.6 Note on speed, memory, and accuracy

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.

There are two optimisation possibilities included via the `opt`-flag: parallelisation (`opt='parallel'`) and spline interpolation (`opt='spline'`). They are switched off by default. The optimization `opt='parallel'` only affects speed and memory usage, whereas `opt='spline'` also affects precision!

I am sure *empymod* could be made much faster with cleverer coding style or with the likes of *cython* or *numba*. Suggestions and contributions are welcomed!

## Memory

By default *empymod* will try to carry out the calculation 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 calculating 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 calculated internally.

## Depths, Rotation, and Bipole

**Depths:** Calculation of many source and receiver positions is fastest if they remain at the same depth, as they can be calculated 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 calculated 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 calculation 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 calculate it in 1 kernel call if you set both integration points to 1, and therefore calculate 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 calculation 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 3.1: Example as a table for comparison: 1 source, 10 receiver (one or many frequencies).

kernel calls	source bipole			receiver bipole			
	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

## Parallelisation

If `opt = 'parallel'`, a good dozen of the most time-consuming statements are calculated by using the `numexpr` package (<https://github.com/pydata/numexpr/wiki/Numexpr-Users-Guide>). These statements are all in the kernel-functions `greenfct`, `reflections`, and `fields`, and all involve  $\Gamma$  in one way or another, often calculating square roots or exponentials. As  $\Gamma$  has dimensions (#frequencies, #offsets, #layers, #lambdas), it can become fairly big.

The package `numexpr` has to be built with Intel's VML, otherwise it won't be used. You can check if it uses VML with

```
>>> import numexpr
>>> numexpr.use_vml
```

The module `numexpr` uses by default all available cores up to a maximum of 8. You can change this behaviour to a lower or a higher value with the following command (in the example it is changed to 4):

```
>>> import numexpr
>>> numexpr.set_num_threads(4)
```

This parallelisation will make `empymod` faster if you calculate a lot of offsets/frequencies at once, but slower for few offsets/frequencies. Best practice is to check first which one is faster. (You can use the benchmark-notebook in the [empymod/example-notebooks-repository](#).)



## Spline interpolation

If `opt = 'spline'`, the so-called *lagged convolution* or *splined* variant of the *DLF* (depending on `htarg`) or the *splined* version of the *QWE* are applied. The spline option should be used with caution, as it is an interpolation and therefore less precise than the non-spline version. However, it significantly speeds up *QWE*, and massively speeds up *DLF*. (The `numexpr`-version of the spline option is slower than the pure spline one, and therefore it is only possible to have either `'parallel'` or `'spline'` on.)

Setting `opt = 'spline'` is generally faster. Good speed-up is achieved for *QWE* by setting `maxint` as low as possible. Also, the higher `nquad` is, the higher the speed-up will be. The variable `pts_per_dec` has also some influence. For *DLF*, big improvements are achieved for long *DLF*-filters and for many offsets/frequencies (thousands). Additionally, spline minimizes memory requirements a lot. Speed-up is greater if all source-receiver angles are identical.

*DLF*: Default for `pts_per_dec = None`, which is the original *lagged convolution*, where the spacing is defined by the filter-base, the transform is carried out first followed by spline-interpolation. You can set this parameter to an integer, which defines the number of points to evaluate per decade. In this case the spline-interpolation is carried out first, followed by the transformation. The original *lagged convolution* is generally the fastest for a very good precision. However, by setting `pts_per_dec` appropriately one can achieve higher precision, normally at the cost of speed.

**Warning:** Keep in mind that it uses interpolation, and is therefore not as accurate as the non-spline version. Use with caution and always compare with the non-spline version if you can apply the spline-version to your problem at hand!

Be aware that *QUAD* (Hankel transform) *always* use the splined version and *always* loop over offsets. The Fourier transforms *FFTlog*, *QWE*, and *FFT* always use interpolation too, either in the frequency or in the time domain. With the *DLF* Fourier transform (sine and cosine transforms) you can choose between no interpolation and interpolation (splined or lagged).

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 notebook `7a_DLF-Standard-Lagged-Splined` in the [example-notebooks](#) repository.

## Looping

By default, you can calculate 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 `opt = 'spline'`. 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 together with `opt = 'parallel'` can have a huge influence.

## Vertical components

It is advised to use `xdirect = True` (the default) if source and receiver are in the same layer to calculate

- 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.

### 3.1.7 License

Copyright 2016-2018 Dieter Werthmüller

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

See the LICENSE- and NOTICE-files on GitHub for more information.

---

**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, <http://www.conacyt.gob.mx>)*, carried out at *The Mexican Institute of Petroleum IMP (Instituto Mexicano del Petróleo, <http://www.gob.mx/imp>)*.

---

### 3.1.8 References

## 3.2 Roadmap

A collection of ideas of what could be added or improved in empymod. Please get in touch if you would like to tackle one of these problems!

- **Additional modelling routines**
  - `tdem (TEM)` [empymod#8]: Issues that have to be addressed: ramp waveform, windowing, loop integration, zero-offset (coincident loop).
    - \* in-loop
    - \* coincident loop
    - \* ...
  - **Ramp waveform** [empymod#7]
  - **Arbitrary waveform** [empymod#7]
  - Improve the GPR-routine [empymod#9]
  - Load and save functions to easily store and load model information (resistivity model, acquisition parameters, and modelling parameters) together with the modelling data (using `pickle` or `shelve`).
- **Inversion** [empyscripts#1]: Inversion routines, preferably a selection of different ones.
- Additional (semi-)analytical functions (where possible)
  - Complete full-space (electric and magnetic source and receiver); space-time domain
  - Extend diffusive half-space solution to magnetic sources and receivers; space-frequency and space-time domains
  - Complete half-space
- Fourier transform
  - Include the method outlined by Mulder et al., 2008, Geophysics (piecewise-cubic Hermite interpolation with a FFT).
  - Change `fft` to use discrete sine/cosine transforms instead, as all other Fourier transforms

- If previous step is successful, clean up the internal decisions (`utils.check_time`) when to use sine/cosine transform (not consistent at the moment, some choice only exists with `ffht` impulse responses, `fqwe` and `fftlog` use sine for impulse, and all three use sine for step-on responses and cosine for step-off responses)
- Hankel transform
  - Add the `fht`-module from `FFTLog` for the Hankel transform.
- Extend examples (example-notebooks)
  - Add different methods (e.g. DC)
  - Reproduce published results
- A `cython`, `numba`, or pure C/C++ implementation of the `kernel` and the `transform` modules. Maybe not worth it, as it may improve speed, but decrease accessibility. Both at the same time would be nice. A fast C/C++-version for calculations (inversions), and a Python-version to tinker with for interested folks. (Probably combined with default parallelisation, removing the `numexpr` variant.)
- Abstraction of the code.
- GUI.
- Add a benchmark suite, e.g. <http://asv.readthedocs.io>, in addition to the testing suite.
- Add some clever checks, e.g. as in Key (2012): abort loops if the field is strongly attenuated (more relevant if once an inversion is implemented).
- Move `empymod` from channel ‘`prisa`’ to ‘`conda-forge`’ (pros/cons?).

## 3.3 Changelog

### 3.3.1 v1.5.2 - 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 than the corresponding filter.

### 3.3.2 v1.5.1 - 2018-02-24

- Documentation:
  - Simplifications: avoid duplication as much as possible between the website ([empymod.github.io](http://empymod.github.io)), the manual ([empymod.readthedocs.io](http://empymod.readthedocs.io)), and the `README` ([github.com/empymod/empymod](https://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.

### 3.3.3 v1.5.0 - 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.

### 3.3.4 v1.4.4 - 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
- Version of CSEM book.

### 3.3.5 v1.4.2 - 2017-06-04

- Bugfix: Fixed squeeze in `model.analytical` with `solution='dsplit'`.
- Version of final submission of manuscript to Geophysics.

### 3.3.6 v1.4.1 - 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](http://empymod.github.io) as entry point, and the project page on [github.com/empymod](https://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).

### 3.3.7 v1.3.0 - 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

### 3.3.8 v1.2.1 - 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

### 3.3.9 v1.2.0 - 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

### 3.3.10 v1.1.0 - 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

### 3.3.11 v1.0.0 - 2016-11-29

- Initial release; state of manuscript submission to geophysics.

## 3.4 Credits

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).

## 3.5 Code

### 3.5.1 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`

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.

Further routines are:

- `analytical`: Calculate analytical fullspace and halfspace solutions.
- `wavenumber`: Calculate the electromagnetic wavenumber-domain solution.
- `gpr`: Calculate the Ground-Penetrating Radar (GPR) response.

The `wavenumber` routine can be used if you are interested in the wavenumber-domain result, without Hankel nor Fourier transform. It calls straight the `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, xdirect=True, ht='fht', htarg=None,
                    ft='sin', ftarg=None, opt=None, loop=None, verb=2)
```

Return the electromagnetic field due to an electromagnetic source.

Calculate 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. 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 variable z (dipole) or z0 and z1 (bipole) must either be single values or having the same dimension as the other coordinates.
- The variables azimuth and dip must be single values. If they have different angles, you have to use the bipole-method (with `srcpts/recpts = 1`, so it is calculated as dipoles).

Angles (coordinate system is left-handed, positive z down (East-North-Depth):

- azimuth (°): horizontal deviation from x-axis, anti-clockwise.
- dip (°): vertical deviation from xy-plane downwards.

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.

**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. Default is ones.

**mpermH, mpermV** : array\_like, optional

Relative horizontal/vertical magnetic permeabilities  $\mu_h/\mu_v$  (-); #mpermH = #mpermV = #res. Default is ones.

**msrc, mrec** : boolean, 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.

**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. Defaults to True.



**ht** : { 'fht', 'qwe', 'quad' }, optional

Flag to choose either the *Digital Linear Filter* method (FHT, *Fast Hankel Transform*), the *Quadrature-With-Extrapolation* (QWE), or a simple *Quadrature* (QUAD) for the Hankel transform. Defaults to 'fht'.

**htarg** : dict or list, optional

**Depends on the value for ht:**

- If **ht** = 'fht': [filter, pts\_per\_dec]:
  - **filter**: string of filter name in **empymod.filters** or the filter method itself. (default: `empymod.filters.key_201_2009()`)
  - **pts\_per\_dec**: points per decade (only relevant if **spline=True**)
    - If none, standard lagged convolution is used. (default: None)
- If **ht** = 'qwe': [rtol, atol, nquad, maxint, pts\_per\_dec, diff\_quad, a, b, limit]:
  - **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; only relevant if **opt='spline'** (default: 80)
  - **diff\_quad**: criteria when to swap to QUAD (only relevant if **opt='spline'**) (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': [atol, rtol, limit, lmin, lmax, pts\_per\_dec]:
  - **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)

The values can be provided as dict with the keywords, or as list. However, if provided as list, you have to follow the order given above. A few examples, assuming **ht = qwe**:

- **Only changing rtol**: { 'rtol': 1e-4 } or [1e-4] or 1e-4
- **Changing rtol and nquad**: { 'rtol': 1e-4, 'nquad': 101 } or [1e-4, '', 101]
- **Only changing diff\_quad**: { 'diffquad': 10 } or ['', '', '', '', '', 10]

**ft** : { '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 'sin'.

**ftarg** : dict or list, optional

**Only used if `signal != None`. Depends on the value for `ft`:**

- If `ft = 'sin' or 'cos'`: [`filter`, `pts_per_dec`]:
  - **filter**: 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. If none, standard lagged convolution is used. If <1, no interpolation is used at all. (Default: None)
- If `ft = 'qwe'`: [`rtol`, `atol`, `nquad`, `maxint`, `pts_per_dec`]:
  - `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`, `add_dec`, `q`]:
  - `pts_per_dec`: samples 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`, `nfreq`, `ntot`]:
  - `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.

The values can be provided as dict with the keywords, or as list. However, if provided as list, you have to follow the order given above. See `htarg` for a few examples.

**opt** : {None, 'parallel', 'spline'}, optional

**Optimization flag. Defaults to None:**

- None: Normal case, no parallelization nor interpolation is used.
- If 'parallel', the package `numexpr` is used to evaluate the most expensive statements. Always check if it actually improves performance for a specific problem. It can speed up the calculation for big arrays, but will most likely be slower for small arrays. It will use all available cores for these specific statements, which all contain Gamma in one way or another, which has dimensions (`#frequencies`, `#offsets`, `#layers`, `#lambdas`), therefore can grow pretty big. The module `numexpr` uses by default all available cores up to a maximum of 8. You can change this behaviour to your desired number of threads `nthreads` with `numexpr.set_num_threads(nthreads)`.
- If 'spline', the *lagged convolution* or *splined* variant of the DLF/FHT or the *splined* version of the QWE are used. Use with caution and check with the non-spline version for a specific problem. (Can be faster, slower, or plainly wrong, as it uses interpolation.) If spline is set it will make use of the parameter `pts_per_dec` that can be defined in `htarg`. If `pts_per_dec` is not set for DLF/FHT, then the *lagged* version is used, else the *splined*. This option has no effect on QUAD.

The option 'parallel' only affects speed and memory usage, whereas 'spline' also affects precision! Please read the note in the *README* documentation for more information.

**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 `opt = 'spline'`. 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!

**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** **EM** : ndarray, (nfreq, nrec, nsrc)

**Frequency- or time-domain EM field (depending on `signal`):**

- If `rec` is electric, returns  $E$  [V/m].
- If `rec` is magnetic, returns  $B$  [T] (not  $H$  [A/m!]).

In the case of the impulse time-domain response, the unit is further divided by seconds [1/s].

However, source and receiver are normalised (unless `strength != 0`). So for instance in the electric case the source strength is 1 A and its length is 1 m. So the electric field could also be written as  $[V/(A.m2)]$ .

In the magnetic case the source strength is given by  $i\omega\mu_0 A I^e$ , where  $A$  is the loop area ( $m^2$ ), and  $I^e$  the electric source strength. For the normalized magnetic source  $A = 1m^2$  and  $I^e = 1Ampere$ . A magnetic source is therefore frequency dependent.

The shape of EM is (nfreq, nrec, nsrc). However, single dimensions are removed.

See also:

**fem** Electromagnetic frequency-domain response.

**tem** Electromagnetic time-domain response.

## Examples

```
>>> import numpy as np
>>> from empymod import bipole
>>> # 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
>>> # Calculate electric field due to an electric source at 1 Hz.
>>> # [msrc = mrec = True (default)]
>>> EMfield = bipole(src, rec, depth, res, freq, verb=4)
:: empymod START ::
~
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
frequency   [Hz] :  1
Hankel      :  DLF (Fast Hankel Transform)
  > Filter    :  Key 201 (2009)
  > pts_per_dec :  Defined by filter (lagged)
Hankel Opt. :  None
Loop over   :  None (all vectorized)
Source(s)   :  1 bipole(s)
  > intpts    :  1 (as dipole)
  > length    [m] :  100
  > 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, xdirect=True, ht='fht', htarg=None, ft='sin', ftarg=None, opt=None, loop=None, verb=2)`

Return the electromagnetic field due to a dipole source.

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 (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.

**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** : 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. Default is ones.

**mpermH, mpermV** : array\_like, optional

Relative horizontal/vertical magnetic permeabilities  $\mu_h/\mu_v$  (-); #mpermH = #mpermV = #res. Default is ones.

**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. Defaults to True.

**ht** : { 'fht', 'qwe', 'quad' }, optional

Flag to choose either the *Digital Linear Filter* method (FHT, *Fast Hankel Transform*), the *Quadrature-With-Extrapolation* (QWE), or a simple *Quadrature* (QUAD) for the Hankel transform. Defaults to 'fht'.

**htarg** : dict or list, optional

**Depends on the value for ht:**

- If **ht** = 'fht': [filter, pts\_per\_dec]:
  - **filter**: string of filter name in **empymod.filters** or the filter method itself. (default: `empymod.filters.key_201_2009()`)
  - **pts\_per\_dec**: points per decade (only relevant if **spline**=True)

**If none, standard lagged convolution is used.** (default: None)
- If **ht** = 'qwe': [rtol, atol, nquad, maxint, pts\_per\_dec, diff\_quad, a, b, limit]:
  - **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; only relevant if **opt**='spline' (default: 80)
  - **diff\_quad**: criteria when to swap to QUAD (only relevant if **opt**='spline') (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': [atol, rtol, limit, lmin, lmax, pts\_per\_dec]:
  - **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)

The values can be provided as dict with the keywords, or as list. However, if provided as list, you have to follow the order given above. A few examples, assuming `ht = qwe`:

- **Only changing rtol:** `{'rtol': 1e-4}` or `[1e-4]` or `1e-4`
- **Changing rtol and nquad:** `{'rtol': 1e-4, 'nquad': 101}` or `[1e-4, ' ', 101]`
- **Only changing diff\_quad:** `{'diffquad': 10}` or `[' ', ' ', ' ', ' ', ' ', 10]`

**ft** : `{'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 'sin'.

**ftarg** : dict or list, optional

Only used if `signal != None`. Depends on the value for **ft**:

- If `ft = 'sin'` or `'cos'`: `[filter, pts_per_dec]`:
  - **filter:** 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. If none, standard lagged convolution is used. If <1, no interpolation is used at all. (Default: None)
- If `ft = 'qwe'`: `[rtol, atol, nquad, maxint, pts_per_dec]`:
  - 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, add_dec, q]`:
  - 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);  $-1 \leq q \leq 1$
- If `ft = 'fft'`: `[dfreq, nfreq, ntot]`:
  - 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.

The values can be provided as dict with the keywords, or as list. However, if provided as list, you have to follow the order given above. See `htarg` for a few examples.

**`opt`** : {None, 'parallel', 'spline'}, optional

**Optimization flag. Defaults to None:**

- None: Normal case, no parallelization nor interpolation is used.
- If 'parallel', the package `numexpr` is used to evaluate the most expensive statements. Always check if it actually improves performance for a specific problem. It can speed up the calculation for big arrays, but will most likely be slower for small arrays. It will use all available cores for these specific statements, which all contain Gamma in one way or another, which has dimensions (#frequencies, #offsets, #layers, #lambdas), therefore can grow pretty big. The module `numexpr` uses by default all available cores up to a maximum of 8. You can change this behaviour to your desired number of threads `nthreads` with `numexpr.set_num_threads(nthreads)`.
- If 'spline', the *lagged convolution* or *splined* variant of the DLF/FHT or the *splined* version of the QWE are used. Use with caution and check with the non-spline version for a specific problem. (Can be faster, slower, or plainly wrong, as it uses interpolation.) If spline is set it will make use of the parameter `pts_per_dec` that can be defined in `htarg`. If `pts_per_dec` is not set for DLF/FHT, then the *lagged* version is used, else the *splined*. This option has no effect on QUAD.

The option 'parallel' only affects speed and memory usage, whereas 'spline' also affects precision! Please read the note in the *README* documentation for more information.

**`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 `opt = 'spline'`. 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!

**`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** **EM** : ndarray, (nfreq, nrec, nsrc)

**Frequency- or time-domain EM field (depending on signal):**

- If rec is electric, returns E [V/m].
- If rec is magnetic, returns B [T] (not H [A/m]!).

In the case of the impulse time-domain response, the unit is further divided by seconds [1/s].

However, source and receiver are normalised. So for instance in the electric case the source strength is 1 A and its length is 1 m. So the electric field could also be written as [V/(A.m2)].

The shape of EM is (nfreq, nrec, nsrc). However, single dimensions are removed.

**See also:**

**bipole** Electromagnetic field due to an electromagnetic source.

**fem** Electromagnetic frequency-domain response.

**tem** Electromagnetic time-domain response.

## Examples

```
>>> import numpy as np
>>> from empymod import dipole
>>> 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 = 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.analytical(src, rec, res, freqtime, solution='fs', signal=None, ab=11,
                          aniso=None, epermH=None, epermV=None, mpermH=None,
                          mpermV=None, verb=2)
```

Return the 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 `fullspace` and `halfspace` in `kernel.py` 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, [Hunziker *et al* 2015].

- Diffusive fullspace solution (`solution='dfs'`) for ee-fields, [Slob\_et\_al\_2010].
- Diffusive halfspace solution (`solution='dhs'`) for ee-fields, [Slob\_et\_al\_2010].
- Diffusive direct- and reflected field and airwave (`solution='dsplit'`) for ee-fields, [Slob\_et\_al\_2010].
- Diffusive direct- and reflected field and airwave (`solution='dtetm'`) for ee-fields, split into TE and TM mode [Slob\_et\_al\_2010].

**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.

`res` : float

Horizontal resistivity `rho_h` (Ohm.m).

`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 field 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` (-); default is one. Ignored for the diffusive solution.

**mpermH, mpermV** : float, optional

Relative horizontal/vertical magnetic permeability  $\mu_h/\mu_v$  (-); 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** : ndarray, (nfreq, nrec, nsrc)

**Frequency- or time-domain EM field (depending on signal):**

- If rec is electric, returns E [V/m].
- If rec is magnetic, returns B [T] (not H [A/m]!).

In the case of the impulse time-domain response, the unit is further divided by seconds [1/s].

However, source and receiver are normalised. So for instance in the electric case the source strength is 1 A and its length is 1 m. So the electric field could also be written as [V/(A.m2)].

The shape of EM is (nfreq, 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 numpy as np
>>> from empymod import analytical
>>> src = [0, 0, 0]
>>> rec = [np.arange(1, 11)*500, np.zeros(10), 200]
>>> res = 50
>>> EMfield = 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, xdirect=True,
                  ht='quad', htarg=None, ft='fft', ftarg=None, opt=None, loop=None, verb=2)
```

Return the 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.

For input parameters see the function `dipole`, except for:

**Parameters** `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`.

**Returns** `EM` : ndarray

GPR response

`empymod.model.wavenumber` (*src, rec, depth, res, freq, wavenumber, ab=11, aniso=None, epermH=None, epermV=None, mpermH=None, mpermV=None, verb=2*)

Return the 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 (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. 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. Default is ones.

**mpermH, mpermV** : array\_like, optional

Relative horizontal/vertical magnetic permeabilities  $\mu_h/\mu_v$  (-); #mpermH = #mpermV = #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** **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** Electromagnetic field due to an electromagnetic source (dipoles).

**bipole** Electromagnetic field due to an electromagnetic source (bipoles).

**fem** Electromagnetic frequency-domain response.

**tem** Electromagnetic time-domain response.

## Examples

```
>>> import numpy as np
>>> from empymod.model import wavenumber
>>> src = [0, 0, 100]
>>> rec = [5000, 0, 200]
>>> depth = [0, 300, 1000, 1050]
>>> res = [1e20, .3, 1, 50, 1]
>>> freq = 1
>>> wavenrs = np.logspace(-3.7, -3.6, 10)
>>> PJ0, PJ1 = wavenumber(src, rec, depth, res, freq, wavenrs, 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, use\_spline, use\_ne\_eval, msrc, mrec, loop\_freq, loop\_off, conv=True*)

Return the electromagnetic frequency-domain response.

This function is called from one of the above modelling routines. No input-check is carried out here. See the main description of `model` for information regarding input and output parameters.

This function can be directly used 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 the time-domain response of the frequency-domain response `fEM`.

This function is called from one of the above modelling routines. No input-check is carried out here. See the main description of `model` for information regarding input and output parameters.

This function can be directly used 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.

### 3.5.2 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 [Hunziker\_et\_al\_2015], which can be found at [software.seg.org/2015/0001](http://software.seg.org/2015/0001). These functions are (c) 2015 by Hunziker et al. and the Society of Exploration Geophysicists, <http://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, use\_ne\_eval*)

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 [Hunziker\_et\_al\_2015], and equally loosely to the file `kxwmod.c`.

[Hunziker\_et\_al\_2015] 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 `transform` are set-up with Bessel functions of order 0 and 1 only. This is achieved by applying the recurrence formula

$$J_2(kr) = \frac{2}{kr} J_1(kr) - J_0(kr) .$$

---

**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 `factAng` 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 `transform`. Consult the modelling routines in `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 angle. 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 `transform`. Consult the modelling routines in `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.

$$\hat{G}_{\alpha\beta}^{ee}, \hat{G}_{3\alpha}^{ee}, \hat{G}_{33}^{ee}, \hat{G}_{\alpha\beta}^{em}, \hat{G}_{\alpha 3}^{em}$$

This function corresponds to equations 45–50 in [Hunziker\_et\_al\_2015], 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 `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, use\_ne\_eval*)  
Calculate Green's function for TM and TE.

$$\tilde{g}_{hh}^{tm}, \tilde{g}_{hz}^{tm}, \tilde{g}_{zh}^{tm}, \tilde{g}_{zz}^{tm}, \tilde{g}_{hh}^{te}, \tilde{g}_{zz}^{te}$$

This function corresponds to equations 108–110, 117/118, 122; 89–94, A18–A23, B13–B15; 97–102 A26–A31, and B16–B18 in [Hunziker\_et\_al\_2015], 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 `kernel.wavenumber`.

`empymod.kernel.reflections` (*depth, e\_zH, Gam, lrec, lsrc, use\_ne\_eval*)  
Calculate  $R_p$ ,  $R_m$ .

$$R_n^{\pm}, \bar{R}_n^{\pm}$$

This function corresponds to equations 64/65 and A-11/A-12 in [Hunziker\_et\_al\_2015], and loosely to the corresponding files `Rmin.F90` and `Rplus.F90`.

This function is called from the function `kernel.greenfct`.

`empymod.kernel.fields` (*depth, Rp, Rm, Gam, lrec, lsrc, zsrc, ab, TM, use\_ne\_eval*)  
Calculate  $P_u^+$ ,  $P_u^-$ ,  $P_d^+$ ,  $P_d^-$ .

$$P_s^{u\pm}, P_s^{d\pm}, \bar{P}_s^{u\pm}, \bar{P}_s^{d\pm}, P_{s-1}^{u\pm}, P_n^{u\pm}, \bar{P}_{s-1}^{u\pm}, \bar{P}_n^{u\pm}, P_{s+1}^{d\pm}, P_n^{d\pm}, \bar{P}_{s+1}^{d\pm}, \bar{P}_n^{d\pm}$$

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 [Hunziker\_et\_al\_2015], and loosely to the corresponding files `Pdownmin.F90`, `Pdownplus.F90`, `Pupmin.F90`, and `Pdownmin.F90`.

This function is called from the function `kernel.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 [Slob\_et\_al\_2010], where the electric source is located at  $[0, 0, zsrc]$ , and the electric receiver at  $[xco, yco, zrec]$ .

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 `model`. Consult these modelling routines for a description of the input and solution parameters.

### 3.5.3 `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 [\[Key\\_2012\]](#), which can be found at [software.seg.org/2012/0003](http://software.seg.org/2012/0003). These functions are (c) 2012 by Kerry Key and the Society of Exploration Geophysicists, <http://software.seg.org/disclaimer.txt>. Please read the NOTICE-file in the root directory for more information regarding the involved licenses.

`empymod.transform.fht` (*zsrc, zrec, lsrc, lrec, off, angle, depth, ab, etaH, etaV, zetaH, zetaV, xdirect, fhtarg, use\_spline, use\_ne\_eval, msrc, mrec*)

Hankel Transform using the Digital Linear Filter method.

The *Digital Linear Filter* method was introduced to geophysics by [\[Ghosh\\_1971\]](#), and made popular and wide-spread by [\[Anderson\\_1975\]](#), [\[Anderson\\_1979\]](#), [\[Anderson\\_1982\]](#). 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 [\[Key\\_2012\]](#), equation 6. Without going into the mathematical details (which can be found in any of the above papers) and following [\[Key\\_2012\]](#), the DLF method rewrites the Hankel transform of the form

$$F(r) = \int_0^{\infty} f(\lambda) J_v(\lambda r) d\lambda$$

as

$$F(r) = \sum_{i=1}^n f(b_i/r) h_i/r,$$

where  $h$  is the digital filter. The Filter abscissae  $b$  is given by

$$b_i = \lambda_i r = e^{ai}, \quad i = -l, -l+1, \dots, l,$$

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 [\[Key\\_2012\]](#).

The function is called from one of the modelling routines in `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.hqwe` (*zsrc, zrec, lsrc, lrec, off, angle, depth, ab, etaH, etaV, zetaH, zetaV, xdirect, qweargs, use\_spline, use\_ne\_eval, msrc, mrec*)

Hankel Transform using Quadrature-With-Extrapolation.

*Quadrature-With-Extrapolation* was introduced to geophysics by [\[Key\\_2012\]](#). It is one of many so-called *ISE* methods to solve Hankel Transforms, where *ISE* stands for Integration, Summation, and Extrapolation.



Following [Key\_2012], but without going into the mathematical details here, the QWE method rewrites the Hankel transform of the form

$$F(r) = \int_0^\infty f(\lambda) J_v(\lambda r) d\lambda$$

as a quadrature sum which form is similar to the DLF (equation 15),

$$F_i \approx \sum_{j=1}^m f(x_j/r) w_j g(x_j) = \sum_{j=1}^m f(x_j/r) \hat{g}(x_j),$$

but with various bells and whistles applied (using the so-called Shanks transformation in the form of a routine called  $\epsilon$ -algorithm ([Shanks\_1955], [Wynn\_1956]; implemented with algorithms from [Trefethen\_2000] and [Weniger\_1989]).

This function is based on `get_CSEM1D_FD_QWE.m`, `qwe.m`, and `getBesselWeights.m` from the source code distributed with [Key\_2012].

In the spline-version, `hqwe` 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 `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.hquad(zsrc, zrec, lsrc, lrec, off, angle, depth, ab, etaH, etaV, zetaH, zetaV, xdirect, quadargs, use_spline, use_ne_eval, 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 `fht` or `hqwe`, 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 `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.fhft(fEM, time, freq, ftarg)`

Fourier Transform using the Digital Linear Filter method.

It follows the Filter methodology [Anderson\_1975], using Cosine- and Sine-filters; see `fht` for more information.

The function is called from one of the modelling routines in `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 [Key\_2012].

**Returns tEM** : array

Returns time-domain EM response of `fEM` for given `time`.

**conv** : bool

Only relevant for QWE/QUAD.

`empymod.transform.fqwe` (*fEM, time, freq, qwargs*)

Fourier Transform using Quadrature-With-Extrapolation.

It follows the QWE methodology [Key\_2012] for the Hankel transform, see `hqwe` for more information.

The function is called from one of the modelling routines in `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 [Key\_2012].

`fqwe` 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.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 [Hamilton\_2000] 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 (`fft1` in FFTLog), not the Hankel transform (`fht` 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  $-1 \leq q \leq 1$ .

The function is called from one of the modelling routines in `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.fft` (*fEM, time, freq, ftarg*)

Fourier Transform using the Fast Fourier Transform.

The function is called from one of the modelling routines in `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, lagged, splined, kind=None, factAng=None, ab=None*)

Digital Linear Filter method.

This is the kernel of the DLF method, used for the Hankel (`fht`) and the Fourier (`ffht`) Transforms. See `fht` 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 requires two additional parameters: *factAng*, 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 DO depend on the angle between source and receiver.

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, factAng=None)`  
 Quadrature-With-Extrapolation.

This is the kernel of the QWE method, used for the Hankel (`hqwe`) and the Fourier (`fqwe`) Transforms. See `hqwe` for an extensive description.

This function is based on `qwe.m` from the source code distributed with [Key\_2012].

`empymod.transform.get_spline_values (filt, inp, nr_per_dec=None)`  
 Return required calculation points.

`empymod.transform.fhti (rmin, rmax, n, q, mu)`  
 Return parameters required for FFTLog.

### 3.5.4 filters – Digital Linear Filters

Filters for the *Digital Linear Filter* (DLF) method for the Hankel [Ghosh\_1971] and the Fourier ([Anderson\_1975]) 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 [Kong\_2007], and `key_101_2009`, `key_201_2009`, `key_401_2009`, `key_81_CosSin_2009`, `key_241_CosSin_2009`, and `key_601_CosSin_2009` from [Key\_2009] are taken from *DIPOLE1D*, [Key\_2009], which can be downloaded at <http://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 [Anderson\_1982] and `key_51_2012`, `key_101_2012`, `key_201_2012`, `key_101_CosSin_2012`, and `key_201_CosSin_2012`, all from [Key\_2012], are taken from the software distributed with [Key\_2012] and available at <http://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)`  
 Simple Class for Digital Linear Filters.

`empymod.filters.anderson_801_1982 ()`  
 Anderson 801 pt Hankel filter, as published in [Anderson\_1982].  
 Taken from file `wa801Hankel.txt` provided with SEG-2012-003.  
 License: <http://software.seg.org/disclaimer.txt>.

`empymod.filters.key_101_2009 ()`  
 Key 101 pt Hankel filter, as published in [Key\_2009].  
 Taken from file `FilterModules.f90` provided with 1DCSEM.

License: [Apache License, Version 2.0](#).

`empymod.filters.key_101_2012()`

Key 101 pt Hankel filter, as published in [\[Key\\_2012\]](#).

Taken from file `kk101Hankel.txt` provided with SEG-2012-003.

License: <http://software.seg.org/disclaimer.txt>.

`empymod.filters.key_101_CosSin_2012()`

Key 101 pt CosSin filter, as published in [\[Key\\_2012\]](#).

Taken from file `kk101CosSin.txt` provided with SEG-2012-003.

License: <http://software.seg.org/disclaimer.txt>.

`empymod.filters.key_201_2009()`

Key 201 pt Hankel filter, as published in [\[Key\\_2009\]](#).

Taken from file `FilterModules.f90` provided with 1DCSEM.

License: [Apache License, Version 2.0](#).

`empymod.filters.key_201_2012()`

Key 201 pt Hankel filter, as published in [\[Key\\_2012\]](#).

Taken from file `kk201Hankel.txt` provided with SEG-2012-003.

License: <http://software.seg.org/disclaimer.txt>.

`empymod.filters.key_201_CosSin_2012()`

Key 201 pt CosSin filter, as published in [\[Key\\_2012\]](#).

Taken from file `kk201CosSin.txt` provided with SEG-2012-003.

License: <http://software.seg.org/disclaimer.txt>.

`empymod.filters.key_241_CosSin_2009()`

Key 241 pt CosSin filter, as published in [\[Key\\_2009\]](#).

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 [\[Key\\_2009\]](#).

Taken from file `FilterModules.f90` provided with 1DCSEM.

License: [Apache License, Version 2.0](#).

`empymod.filters.key_51_2012()`

Key 51 pt Hankel filter, as published in [\[Key\\_2012\]](#).

Taken from file `kk51Hankel.txt` provided with SEG-2012-003.

License: <http://software.seg.org/disclaimer.txt>.

`empymod.filters.key_601_CosSin_2009()`

Key 601 pt CosSin filter, as published in [\[Key\\_2009\]](#).

Taken from file `FilterModules.f90` provided with 1DCSEM.

License: [Apache License, Version 2.0](#).

`empymod.filters.key_81_CosSin_2009()`

Key 81 pt CosSin filter, as published in [\[Key\\_2009\]](#).

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 [\[Kong\\_2007\]](#).

Taken from file `FilterModules.f90` provided with 1DCSEM.

License: [Apache License, Version 2.0](#).

`empymod.filters.kong_61_2007()`

Kong 61 pt Hankel filter, as published in [\[Kong\\_2007\]](#).

Taken from file `FilterModules.f90` provided with 1DCSEM.

License: [Apache License, Version 2.0](#).

`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](#).

### 3.5.5 utils – Utilites

Utilities for 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`

Subclassing an ndarray: add *amplitude* <amp> and *phase* <pha>.

**Parameters** `realpart`: array

1. Real part of input, if input is real or complex.
2. Imaginary part of input, if input is pure imaginary.
3. Complex input.

In cases 2 and 3, `imagpart` must be `None`.

**imagpart**: array, optional

Imaginary part of input. Defaults to `None`.

#### Examples

```
>>> import numpy as np
>>> from empymod.utils import EMArray
>>> emvalues = EMArray(np.array([1,2,3]), np.array([1, 0, -1]))
>>> print('Amplitude : ', emvalues.amp)
Amplitude : [ 1.41421356  2.          3.16227766]
>>> print('Phase      : ', emvalues.pha)
Phase      : [ 45.          0.        -18.43494882]
```

## Attributes

amp	(ndarray) Amplitude of the input data.
pha	(ndarray) Phase of the input data, in degrees, lag-defined (increasing with increasing offset.) To get lead-defined phases, multiply <code>imagpart</code> by -1 before passing through this function.

`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 `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 `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** : {'sin', 'cos', 'qwe', 'fftlog', 'fft'}

Flag for Fourier transform.

**ftarg** : str or filter from `empymod.filters` or array\_like,

Only used if `signal != None`. 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 `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`, `mpermH`, 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 `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  $\eta_H/\eta_V$ , same size as provided resistivity.

**zetaH, zetaV** : array

Parameters  $\zeta_H/\zeta_V$ , 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 `model`. Consult these modelling routines for a detailed description of the input parameters.

**Parameters** `ht` : {'fht', 'qwe', 'quad'}

Flag to choose the Hankel transform.

**htarg** : str or filter from `empymod.filters` or array\_like,

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_opt` (*opt, loop, ht, htarg, verb*)

Check optimization parameters.

This check-function is called from one of the modelling routines in `model`. Consult these modelling routines for a detailed description of the input parameters.

**Parameters** `opt` : {None, 'parallel', 'spline'}

Optimization flag.

**loop** : {None, 'freq', 'off'}

Loop flag.

**ht** : str

Flag to choose the Hankel transform.



**htarg** : array\_like,

Depends on the value for **ht**.

**verb** : {0, 1, 2, 3, 4}

Level of verbosity.

**Returns use\_spline** : bool

Boolean if to use spline interpolation.

**use\_ne\_eval** : bool

Boolean if to use `numexpr`.

**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 `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 `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 `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 `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 `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 `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 `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 `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 `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_param=None, min_angle=None)`

Set minimum values of parameters.

The given parameters are set to its minimum value if they are smaller.

**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\_param** : float, optional

Minimum aniso, [m/e]perm[H/V] [-] (default 1e-20).

**min\_angle** : float, optional

Minimum angle [-] (default 1e-10).

`empymod.utils.get_minimum()`

Return the current minimum values.

**Returns** **min\_vals** : dict

Dictionary of current minimum values with keys

- **min\_freq** : float
- **min\_time** : float
- **min\_off** : float
- **min\_param** : float
- **min\_angle** : float

For a full description of these options, see *set\_minimum*.



---

## Bibliography

---

- [Anderson\_1975] Anderson, W.L., 1975, Improved digital filters for evaluating Fourier and Hankel transform integrals: USGS Unnumbered Series; <http://pubs.usgs.gov/unnumbered/70045426/report.pdf>.
- [Anderson\_1979] 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](https://doi.org/10.1190/1.1441007).
- [Anderson\_1982] 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](https://doi.org/10.1145/356012.356014).
- [Ghosh\_1971] Ghosh, D. P., 1971, The application of linear filter theory to the direct interpretation of geoelectrical resistivity sounding measurements: *Geophysical Prospecting*, 19, 192–217; DOI: [10.1111/j.1365-2478.1971.tb00593.x](https://doi.org/10.1111/j.1365-2478.1971.tb00593.x).
- [Haines\_and\_Jones\_1988] 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](https://doi.org/10.1111/j.1365-246X.1988.tb01131.x).
- [Hamilton\_2000] 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](https://doi.org/10.1046/j.1365-8711.2000.03071.x); Website of FFTLog: [casa.colorado.edu/~ajsh/FFTLog](http://casa.colorado.edu/~ajsh/FFTLog).
- [Hunziker\_et\_al\_2015] 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](https://doi.org/10.1190/geo2013-0411.1); Software: [software.seg.org/2015/0001](http://software.seg.org/2015/0001).
- [Key\_2009] 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](https://doi.org/10.1190/1.3058434). Software: [marineemlab.ucsd.edu/Projects/Occam/1DCSEM](http://marineemlab.ucsd.edu/Projects/Occam/1DCSEM).
- [Key\_2012] Key, K., 2012, Is the fast Hankel transform faster than quadrature?: *Geophysics*, 77(3), F21–F30; DOI: [10.1190/geo2011-0237.1](https://doi.org/10.1190/geo2011-0237.1); Software: [software.seg.org/2012/0003](http://software.seg.org/2012/0003).
- [Kong\_2007] 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](https://doi.org/10.1111/j.1365-2478.2006.00585.x).
- [Shanks\_1955] Shanks, D., 1955, Non-linear transformations of divergent and slowly convergent sequences: *Journal of Mathematics and Physics*, 34, 1–42; DOI: [10.1002/sapm19553411](https://doi.org/10.1002/sapm19553411).
- [Slob\_et\_al\_2010] 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](https://doi.org/10.2528/PIER10052807).
- [Talman\_1978] 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](https://doi.org/10.1016/0021-9991(78)90107-9).
- [Trefethen\_2000] 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](https://doi.org/10.1137/1.9780898719598.ch12).

- [Weniger\_1989] 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](https://arxiv.org/abs/math/0306302).
- [Werthmuller\_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](https://doi.org/10.1190/geo2016-0626.1).
- [Werthmuller\_2017b] Werthmüller, D., 2017, Getting started with controlled-source electromagnetic 1D modeling: *The Leading Edge*, 36, 352–355; DOI: [10.1190/tle36040352.1](https://doi.org/10.1190/tle36040352.1).
- [Wynn\_1956] Wynn, P., 1956, On a device for computing the  $e_m(S_n)$  transformation: *Math. Comput.*, 10, 91–96; DOI: [10.1090/S0025-5718-1956-0084056-6](https://doi.org/10.1090/S0025-5718-1956-0084056-6).
- [Ziolkowski\_and\_Slob] Ziolkowski, A., and E. Slob, 2018, *Introduction to Controlled-Source Electromagnetic Methods*: Cambridge University Press; expected to be published late 2018.



### e

- `empymod`, [7](#)
- `empymod.filters`, [39](#)
- `empymod.kernel`, [34](#)
- `empymod.model`, [19](#)
- `empymod.transform`, [36](#)
- `empymod.utils`, [41](#)



## A

analytical() (in module empymod.model), 29  
anderson\_801\_1982() (in module empymod.filters), 39  
angle\_factor() (in module empymod.kernel), 34

## B

bipole() (in module empymod.model), 19

## C

check\_ab() (in module empymod.utils), 46  
check\_bipole() (in module empymod.utils), 45  
check\_dipole() (in module empymod.utils), 45  
check\_frequency() (in module empymod.utils), 43  
check\_hankel() (in module empymod.utils), 44  
check\_model() (in module empymod.utils), 43  
check\_opt() (in module empymod.utils), 44  
check\_solution() (in module empymod.utils), 46  
check\_time() (in module empymod.utils), 42  
check\_time\_only() (in module empymod.utils), 42  
conv\_warning() (in module empymod.utils), 48

## D

DigitalFilter (class in empymod.filters), 39  
dipole() (in module empymod.model), 25  
dlf() (in module empymod.transform), 38

## E

EMArray (class in empymod.utils), 41  
empymod (module), 7  
empymod.filters (module), 39  
empymod.kernel (module), 34  
empymod.model (module), 19  
empymod.transform (module), 36  
empymod.utils (module), 41

## F

fem() (in module empymod.model), 33  
ffht() (in module empymod.transform), 37  
fft() (in module empymod.transform), 38  
fflog() (in module empymod.transform), 38  
fht() (in module empymod.transform), 36  
fhti() (in module empymod.transform), 39  
fields() (in module empymod.kernel), 35

fqwe() (in module empymod.transform), 38  
fullspace() (in module empymod.kernel), 35

## G

get\_abs() (in module empymod.utils), 46  
get\_azm\_dip() (in module empymod.utils), 47  
get\_geo\_fact() (in module empymod.utils), 46  
get\_layer\_nr() (in module empymod.utils), 48  
get\_minimum() (in module empymod.utils), 49  
get\_off\_ang() (in module empymod.utils), 48  
get\_spline\_values() (in module empymod.transform), 39  
gpr() (in module empymod.model), 31  
greenfct() (in module empymod.kernel), 35

## H

halfspace() (in module empymod.kernel), 35  
hquad() (in module empymod.transform), 37  
hqwe() (in module empymod.transform), 36

## K

key\_101\_2009() (in module empymod.filters), 39  
key\_101\_2012() (in module empymod.filters), 40  
key\_101\_CosSin\_2012() (in module empymod.filters), 40  
key\_201\_2009() (in module empymod.filters), 40  
key\_201\_2012() (in module empymod.filters), 40  
key\_201\_CosSin\_2012() (in module empymod.filters), 40  
key\_241\_CosSin\_2009() (in module empymod.filters), 40  
key\_401\_2009() (in module empymod.filters), 40  
key\_51\_2012() (in module empymod.filters), 40  
key\_601\_CosSin\_2009() (in module empymod.filters), 40  
key\_81\_CosSin\_2009() (in module empymod.filters), 40  
kong\_241\_2007() (in module empymod.filters), 40  
kong\_61\_2007() (in module empymod.filters), 41

## P

printstartfinish() (in module empymod.utils), 48

## Q

`qwe()` (in module `empymod.transform`), [39](#)

## R

`reflections()` (in module `empymod.kernel`), [35](#)

## S

`set_minimum()` (in module `empymod.utils`), [48](#)

## T

`tem()` (in module `empymod.model`), [34](#)

## W

`wavenumber()` (in module `empymod.kernel`), [34](#)

`wavenumber()` (in module `empymod.model`), [32](#)

`wer_201_2018()` (in module `empymod.filters`), [41](#)