empymod Documentation

Release 1.1.0

Dieter Werthmüller

December 23, 2016

Contents

Version: 1.1.0; Date: December 23, 2016

Manual for *empymod*, a one-dimensional, electromagnetic forward modeller in Python.

The **e**lectro**m**agnetic **py**thon **mod**eller *empymod* can model electric or magnetic responses due to a three-dimensional electric or magnetic source in a layered-earth model with electric vertical isotropy (ρ_h, λ), electric permittivity (ϵ_h, ϵ_v), and magnetic permeability (μ_h, μ_v), from very low frequencies ($f \to 0$ Hz) to very high frequencies ($f \to$ GHz).

Contents:

CHAPTER 1

Info

1.1 Installation & requirements

Just add the path to *empymod* to your python-path variable.

Alternatively, to install it in your python distribution (linux), run:

```
python setup.py install
```

Required are python version 3 or higher and the modules *NumPy*, *SciPy*, and *numexpr*.

1.2 Citation

I am in the process of publishing an article regarding *empymod*, and I will put the info here once it is reality. If you publish results for which you used *empymod*, please consider citing this article. Also consider citing *[Hunziker_et_al_2015]* and *[Key_2012]*, without which *empymod* would not exist.

1.3 License

1.4 Missing features

A list of things that should or could be added and improved, in decreasing priority:

- **Tests**, **tests**, and **more tests**: The modeller *empymod* is lacking an extensive testing suite. But it should have one. This would ideally be combined with automated testing by, for instance, Travis. It should also include some proper benchmarks.
- Rewrite *model* and *utils* in order to provide the survey and model parameters as well as the modelling options as *structured array/dict/class* (which one is suited best?), so that the main (potentially only) calculation routine would be *empymod(survey, model, options)*. Improved abstraction of the calling part.
- **More modelling routines:**
  - arbitrary source and receiver dipole lengths
  - arbitrary source and receiver rotations
  - convolution with a wavelet for GPR (proper version of *model.gpr*)
  - pure wavenumber output-routine (proper version of *model.wavenumber*)
  - variable receiver depths within one calculation
  - various source-receiver arrangements (loops etc)
  - multiple sources within one calculation
  - Load and Save functions to store and load model, together with all information.
- **Kernel**
  - Include *scipy.integrate.quad* as an additional Hankel transform. There are cases when both *QWE* and *FHT* struggle, e.g. at very short offsets with very high frequencies (GPR).
  - A *cython* or *numba* (pure 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-version for calculations (inversions), and a Python-version to tinker with for interested folks.
- GUI frontend

1.5 Notice

This product includes software developed at *The Mexican Institute of Petroleum IMP* (*Instituto Mexicano del Petróleo*, http://www.imp.mx).

The project was funded through *The Mexican National Council of Science and Technology* (*Consejo Nacional de Ciencia y Tecnología*, http://www.conacyt.mx).

This product is a derivative work of *[Hunziker_et_al_2015]* and *[Key_2012]*, and their publicly available software:

1. 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, F1-F18; DOI: 10.1190/geo2013-0411.1; Software: software.seg.org/2015/0001.
2. Key, K., 2012, Is the fast Hankel transform faster than quadrature?: Geophysics, 77, F21-F30; DOI: 10.1190/GEO2011-0237.1; Software: software.seg.org/2012/0003.

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 *FHT* is faster than *QWE*, but *QWE* is much easier on memory usage. I doubt you will ever run into memory issues with *QWE*, whereas for *FHT* you might for ten thousands of offsets or hundreds of layers. Furthermore, *QWE* allows you to control the accuracy.

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!

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.

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!

1.6.1 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 Γ in one way or another, often calculating square roots or exponentials. As Γ has dimensions (#frequencies, #offsets, #layers, #lambdas), it can become fairly big.

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 included *jupyter notebook*-benchmark.)

1.6.2 Spline interpolation

If `opt = 'spline'`, the so-called *lagged convolution* or *splined* variant of the *FHT* (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 *FHT*. (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 *FHT*, big improvements are achieved for long FHT-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.

FHT: 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 the *QWE*- and the *FHT*-Versions for the frequency-to-time transformation *always* use the splined version and *always* loop over offsets.

1.6.3 Looping

By default, you can calculate many offsets and many frequencies all in one go, vectorized (for the *FHT*), 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'` 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.

1.6.4 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.

1.7 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. 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).

1.8 References

CHAPTER 2

Code

2.1 `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!

So far implemented are two routines, both of them for:

- frequency or time
- source and receiver can be either electric or magnetic

The routines are

- ***dipole*:**
  - Point dipole source(s) in direction x, y, or z, all sources at the same depth.
  - Point dipole receivers(s) in direction x, y, or z, all receivers at the same depth.
  - Various frequencies or times.
- ***srcbipole*:**
  - Arbitrary bipole source.
  - Point dipole receivers(s) in direction x, y, or z, all receivers at the same depth.
  - Various frequencies or times.
- *bipole*: *srcbipole* will be superseded eventually by *bipole*, a general source- and receiver-bipole routine.

The above routines make use of the 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*.

Two routines are shortcuts for frequency- and time-domain dipoles, respectively, and mainly in for legacy reasons:

- *frequency*: Shortcut of *dipole* for frequency-domain calculation.
- *time*: Shortcut of *dipole* for time-domain calculation.

Two more routines are more kind of examples and cannot be regarded stable; they can serve as template to create your own routines:

- *gpr*: Calculate the Ground-Penetrating Radar (GPR) response.

- *wavenumber*: Calculate the electromagnetic wavenumber-domain solution.

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=1*)

Return the electromagnetic field due to a dipole source.

Calculate the electromagnetic frequency- or time-domain field due to an infinitesimal small electric or magnetic dipole source, measured by infinitesimal small electric or magnetic dipole receivers; source 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* or *srcbipole* to calculate bipoles of finite length and arbitrary angle.

Parameters **src** : list of floats or arrays

Source coordinates (m): [src-x, src-y, src-z]. The x- and y-coordinates can be arrays, z is a single value. The x- and y-coordinates must have the same dimension.

rec : list of floats or arrays

Receiver coordinates (m): [rec-x, rec-y, rec-z]. The x- and y-coordinates can be arrays, z is a single value. The x- and y-coordinates must have the same dimension.

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).

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 : array_like, optional

Horizontal electric permittivities epsilon_h (-); #epermH = #res. Defaults to ones.

epermV : array_like, optional

Vertical electric permittivities epsilon_v (-); #epermV = #res. Defaults to ones.

mpermH : array_like, optional

Horizontal magnetic permeabilities mu_h (-); #mpermH = #res. Defaults to ones.

mpermV : array_like, optional

Vertical magnetic permeabilities mu_v (-); #mpermV = #res. Defaults to 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'}, optional

Flag to choose either the *Fast Hankel Transform* (FHT) or the *Quadrature-With-Extrapolation* (QWE) for the Hankel transform. Defaults to 'fht'.

htarg : str or filter from empymod.filters or array_like, optional

Depends on the value for *ht*:

- If *ht* = 'fht': array containing: [filter, pts_per_dec]:
  - **filter: string of filter name in *empymod.filters* or** the filter method itself. (default: *empymod.filters.key_401_2009()*)
  - **pts_per_dec: points per decade (only relevant if spline=True)**

    If none, standard lagged convolution is used. (default: None)
- If *ht* = 'qwe': array containing: [rtol, atol, nquad, maxint, pts_per_dec]:
  - 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)

  All are optional, you only have to maintain the order. To only change *nquad* to 11 and use the defaults otherwise, you can provide htarg=['', '', 11].

ft : {'sin', 'cos', 'qwe', 'fftlog'}, optional

Only used if *signal* != None. Flag to choose either the Sine- or Cosine-Filter, the Quadrature-With-Extrapolation (QWE), or FFTLog for the Fourier transform. Defaults to 'sin'.

ftarg : str or filter from empymod.filters or array_like, optional

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

- If *ft* = 'sin' or 'cos': array containing: [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. (Default: None)

- If *ft* = 'qwe': array containing: [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 (only relevant if spline=True) (default: 20)

  All are optional, you only have to maintain the order. To only change *nquad* to 11 and use the defaults otherwise, you can provide ftarg=['', '', 11].

- If *ft* = 'fftlog': array containing: [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 <= q <= 1

  All are optional, you only have to maintain the order. To only change *add_dec* to [-1, 1] and use the defaults otherwise, you can provide ftarg=['', [-1, 1]].

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.
- If 'spline', the *lagged convolution* or *splined* variant of the 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 FHT, then the *lagged* version is used, else the *splined*.

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}, optional

Level of verbosity, defaults to 1:

- 0: Print nothing.
- 1: Print warnings.
- 2: Print warnings and 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.

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.**srcbipole**(*src*, *rec*, *depth*, *res*, *freqtime*, *signal=None*, *aniso=None*, *epermH=None*, *epermV=None*, *mpermH=None*, *mpermV=None*, *msrc=False*, *recdir=1*, *intpts=10*, *xdirect=True*, *ht='fht'*, *htarg=None*, *ft='sin'*, *ftarg=None*, *opt=None*, *loop=None*, *verb=1*)

Return the electromagnetic field due to a bipole source.

Calculate the electromagnetic frequency- or time-domain field due to an arbitrary finite electric or magnetic bipole source, measured by infinitesimal small electric or magnetic dipole receivers; receivers are directed along the principal directions x, y, or z, and all receivers are at the same depth.

Parameters **src** : list of floats

Source coordinates (m): [src-x0, src-x1, src-y0, src-y1, src-z0, src-z1].

rec : list of floats or arrays

Receiver coordinates (m): [rec-x, rec-y, rec-z]. The x- and y-coordinates can be arrays, z is a single value. The x- and y-coordinates must have the same dimension.

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).

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 : array_like, optional

Horizontal electric permittivities epsilon_h (-); #epermH = #res. Defaults to ones.

epermV : array_like, optional

Vertical electric permittivities epsilon_v (-); #epermV = #res. Defaults to ones.

mpermH : array_like, optional

Horizontal magnetic permeabilities mu_h (-); #mpermH = #res. Defaults to ones.

mpermV : array_like, optional

Vertical magnetic permeabilities mu_v (-); #mpermV = #res. Defaults to ones.

msrc : boolean, optional

If True, source is magnetic.

recdir : int, optional

Receiver direction, defaults to 1:

- 1 : Ex
- 2 : Ey
- 3 : Ez
- 4 : Hx
- 5 : Hy
- 6 : Hz

intpts : int, optional

Number of integration points for bipole source, defaults to 10:

- nr < 3 : bipole, but calculated as dipole at centerpoint
- nr >= 3 : bipole

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'}, optional

Flag to choose either the *Fast Hankel Transform* (FHT) or the *Quadrature-With-Extrapolation* (QWE) for the Hankel transform. Defaults to 'fht'.

htarg : str or filter from empymod.filters or array_like, optional

Depends on the value for *ht*:

- If *ht* = 'fht': array containing: [filter, pts_per_dec]:
  - **filter: string of filter name in *empymod.filters* or** the filter method itself. (default: *empymod.filters.key_401_2009()*)
  - **pts_per_dec: points per decade (only relevant if spline=True)**

    If none, standard lagged convolution is used. (default: None)
- If *ht* = 'qwe': array containing: [rtol, atol, nquad, maxint, pts_per_dec]:
  - 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)

  All are optional, you only have to maintain the order. To only change *nquad* to 11 and use the defaults otherwise, you can provide htarg=['', '', 11].

ft : {'sin', 'cos', 'qwe', 'fftlog'}, optional

Only used if *signal* != None. Flag to choose either the Sine- or Cosine-Filter, the Quadrature-With-Extrapolation (QWE), or FFTLog for the Fourier transform. Defaults to 'sin'.

ftarg : str or filter from empymod.filters or array_like, optional

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

- If *ft* = 'sin' or 'cos': array containing: [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. (Default: None)

- If *ft* = 'qwe': array containing: [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 (only relevant if spline=True) (default: 20)

  All are optional, you only have to maintain the order. To only change *nquad* to 11 and use the defaults otherwise, you can provide ftarg=['', '', 11].

- If *ft* = 'fftlog': array containing: [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 <= q <= 1

  All are optional, you only have to maintain the order. To only change *add_dec* to [-1, 1] and use the defaults otherwise, you can provide ftarg=['', [-1, 1]].

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.
- If 'spline', the *lagged convolution* or *splined* variant of the 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 FHT, then the *lagged* version is used, else the *splined*.

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}, optional

Level of verbosity, defaults to 1:

- 0: Print nothing.
- 1: Print warnings.
- 2: Print warnings and 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.

Examples

```
>>> import numpy as np
>>> from empymod import srcbipole
>>> src = [-50, 50, -50, 50, 75, 100]
>>> rec = [np.arange(1, 11)*500, np.zeros(10), 200]
>>> depth = [0, 300, 1000, 1050]
>>> res = [1e20, .3, 1, 50, 1]
>>> EMfield = srcbipole(src, rec, depth, res, freqtime=1, verb=0)
>>> print(EMfield)
[  1.14061401e-10 -2.07836149e-10j  -5.38410978e-12 -2.53976216e-11j
  -2.06275951e-12 -3.33525423e-12j  -1.10983809e-13 -8.09426365e-13j
   1.92903597e-13 -4.55252836e-13j   1.69353850e-13 -3.19059245e-13j
   1.18960468e-13 -2.29643807e-13j   8.09247406e-14 -1.68298859e-13j
   5.50535020e-14 -1.25316192e-13j   3.80049013e-14 -9.44840453e-14j]
```

empymod.model.**frequency**(*src*, *rec*, *depth*, *res*, *freq*, *ab=11*, *aniso=None*, *epermH=None*, *epermV=None*, *mpermH=None*, *mpermV=None*, *xdirect=True*, *ht='fht'*, *htarg=None*, *opt=None*, *loop=None*, *verb=1*)

Shortcut for frequency-domain *dipole*. See *dipole* for info.

empymod.model.**time**(*src*, *rec*, *depth*, *res*, *time*, *ab=11*, *signal=0*, *aniso=None*, *epermH=None*, *epermV=None*, *mpermH=None*, *mpermV=None*, *xdirect=True*, *ht='fht'*, *htarg=None*, *ft='sin'*, *ftarg=None*, *opt=None*, *loop='off'*, *verb=1*)

Shortcut for time-domain *dipole*. See *dipole* for info.

empymod.model.**gpr**(*src*, *rec*, *depth*, *res*, *fc=250*, *ab=11*, *gain=None*, *aniso=None*, *epermH=None*, *epermV=None*, *mpermH=None*, *mpermV=None*, *xdirect=True*, *ht='fht'*, *htarg=None*, *opt=None*, *loop='off'*, *verb=1*)

Return the Ground-Penetrating Radar signal.

THIS FUNCTION IS IN DEVELOPMENT, USE WITH CAUTION.

Or in other words it is merely an example how one could calculate the GPR-response. However, the currently included *FHT* and *QWE* struggle for these high frequencies, and another Hankel transform has to be included to make GPR work properly (e.g. *scipy.integrate.quad*).

- *QWE* is slow, but does a pretty good job except for very short offsets: only direct wave for offset < 0.1 m, triangle-like noise at later times.
- *FHT* is fast. Airwave, direct wave and first reflection are well visible, but afterwards it is very noisy.

A lot is still hard-coded in this routine, for instance the frequency-range used to calculate the response.

For input parameters see *frequency*, except for:

Parameters **fc** : float

Centre frequency of GPR-signal (MHz). Sensible values are between 10 MHz and 3000 MHz.

gain : float

Power of gain function. If None, no gain is applied.

Returns **t** : array

Times (s)

gprEM : ndarray

GPR response

empymod.model.**wavenumber**(*src*, *rec*, *depth*, *res*, *freq*, *wavenumber*, *ab=11*, *aniso=None*, *epermH=None*, *epermV=None*, *mpermH=None*, *mpermV=None*, *xdirect=True*, *verb=1*)

Return the electromagnetic wavenumber-domain field.

THIS FUNCTION IS IN DEVELOPMENT, USE WITH CAUTION.

Or rather, it is for development purposes, to easily get the wavenumber result with the required input checks.

For input parameters see *frequency*, except for:

Parameters **wavenumber** : array

Wavenumbers lambda (1/m)

Returns **PJ0, PJ1, PJ0b** : array

Wavenumber domain EM responses. - PJ0 is angle independent, PJ1 and PJ0b depend on the angle. - PJ0 and PJ0b are J_0 functions, PJ1 is a J_1 function.

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*)

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*)

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.

2.2 `kernel` – Kernel calculation

Kernel of *empymod*, calculates the wavenumber-domain electromagnetic response.

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. 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 angel. The angle-dependency is this factor, which can be applied to the corresponding parts in the wavenumber or in the frequency domain.

The *angle_factor* corresponds to the sine and cosine-functions in Eqs 105-107, 111-116, 119-121, 123-128.

This function is called from one of the Hankel functions in `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.

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

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.

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

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 Rp, Rm.

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

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 Pu+, Pu-, Pd+, Pd-.

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

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`**(*xco*, *yco*, *zsrc*, *zrec*, *res*, *freq*, *aniso=1*, *ab=11*)

Return frequency-space domain VTI half-space solution.

Calculates the frequency-space domain electromagnetic response for a half-space below air using the diffusive approximation, as given in *[Slob_et_al_2010]*.

This routine is not strictly part of *empymod* and not used by it. However, it can be useful to compare the code to the analytical solution.

There are a few known typos in the equations of *[Slob_et_al_2010]*. Write the authors to receive an updated version!

This could be integrated into *empymod* by checking if the top-layer is a very resistive layer, hence air, and the rest is a half-space, and then calling this function instead of *wavenumber*. (Similar to the way *fullspace* is incorporated if all layer parameters are identical.) The time-space domain solution could be implemented as well.

Parameters **xco, yco** : array

Inline and crossline coordinates (m)

zsrc, zrec : float

Source and receiver depth (m)

res : float or array

Half-space resistivity (Ohm.m)

freq : float

Frequency (Hz)

aniso : float, optional

Anisotropy (-), default = 1

ab : int, optional

Src-Rec config, default = 11; {11, 12, 13, 21, 22, 23, 31, 32, 33}

Returns EM half-space solution

Examples

```
>>> from empymod.kernel import halfspace
>>> EM = halfspace(1000, 0, 10, 1, 10, 1)
>>> print('HS response : ', EM)
HS response :  (3.02186073352e-09-3.87322421836e-10j)
```

2.3 `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 FHT 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. 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 Fast Hankel Transform.

The *Fast Hankel Transform* is a *Digital Filter Method*, introduced to geophysics by *[Gosh_1971]*, and made popular and wide-spread by *[Anderson_1975]*, *[Anderson_1979]*, *[Anderson_1982]*.

This implementation of the FHT 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 FHT method rewrites the Hankel transform of the form

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

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

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

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.

`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

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

as a quadrature sum which form is similar to the FHT (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 ϵ-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]*.

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.

`empymod.transform.`**`fft`**(*fEM*, *time*, *freq*, *ftarg*)

Fourier Transform using a Cosine- or a Sine-filter.

It follows the Filter methodology *[Anderson_1975]*, 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*.

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

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]*.

Returns **tEM** : array

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

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 (*fftl* in *FFTLog*), not the Hankel transform (*fht* in *FFTLog*). Furthermore, the following parameters are fixed:
- *mu* = 0.5 (sine-transform)
- *kr* = 1 (initial value)
- *kropt* = 1 (silently adjusts *kr*)
- *dir* = 1 (forward)

Furthermore, *q* is restricted to -1 <= q <= 1.

I am trying to get *FFTLog* into *scipy*. If this happens the current implementation will be replaced by the *scipy.fftpack.fftlog*-version.

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*.

empymod.transform.**qwe**(*rtol*, *atol*, *maxint*, *inp*, *intervals*, *hfstr*, *lambd=None*, *off=None*, *fact-Ang=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*)
Return parameters required for FFTLog.

2.4 `filters` – Digital Filters for FHT

Filters for the *Fast Hankel Transform* (FHT, *[Anderson_1982]*) and the *Fourier Sine and Cosine Transforms [Anderson_1975]*.

To calculate the *fhtfilter.factor* I used

```
np.around(np.average(fhtfilter.base[1:]/fhtfilter.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 marineemlab.ucsd.edu/Projects/Occam/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 software.seg.org/2012/0003. These filters are distributed under the SEG license.

class empymod.filters.**DigitalFilter**(*name*)
Simple Class for Digital Filters.

empymod.filters.**anderson_801_1982**()
Anderson 801: *[Anderson_1982]*.

Anderson 801 pt filter, as published in *[Anderson_1982]*; taken from file *wa801Hankel.txt* from *[Key_2012]*, published by the Society of Exploration Geophysicists; software.seg.org/2012/0003. License: http://software.seg.org/disclaimer.txt.

empymod.filters.**key_101_2009**()
Key 101 2009: *[Key_2009]*.

Key 101 pt filter, as published in *[Key_2009]*; taken from file *FilterModules.f90* from *[Key_2009]*, available on marineemlab.ucsd.edu/Projects/Occam/1DCSEM. License: Apache License, Version 2.0, http://www.apache.org/licenses/LICENSE-2.0.

empymod.filters.**key_101_2012**()
Key 101 2012: *[Key_2012]*.

Key 101 pt filter, taken from file *kk101Hankel.txt* from *[Key_2012]*, published by the Society of Exploration Geophysicists; software.seg.org/2012/0003. License: http://software.seg.org/disclaimer.txt.

empymod.filters.**key_101_CosSin_2012**()
Key 101 CosSin 2012: *[Key_2012]*.

Key 101 pt filter, taken from file *kk101CosSin.txt* from *[Key_2012]*, published by the Society of Exploration Geophysicists; software.seg.org/2012/0003. License: http://software.seg.org/disclaimer.txt.

empymod.filters.**key_201_2009**()
Key 201 2009: *[Key_2009]*.

Key 201 pt filter, as published in *[Key_2009]*; taken from file *FilterModules.f90* from *[Key_2009]*, available on marineemlab.ucsd.edu/Projects/Occam/1DCSEM. License: Apache License, Version 2.0, http://www.apache.org/licenses/LICENSE-2.0.

empymod.filters.**key_201_2012**()
Key 201 2012: *[Key_2012]*.

Key 201 pt filter, taken from file *kk201Hankel.txt* from *[Key_2012]*, published by the Society of Exploration Geophysicists; software.seg.org/2012/0003. License: http://software.seg.org/disclaimer.txt.

empymod.filters.**key_201_CosSin_2012**()
Key 201 CosSin 2012: *[Key_2012]*.

Key 201 pt filter, taken from file *kk201CosSin.txt* from *[Key_2012]*, published by the Society of Exploration Geophysicists; software.seg.org/2012/0003. License: http://software.seg.org/disclaimer.txt.

empymod.filters.**key_241_CosSin_2009**()
Key 241 CosSin 2009: *[Key_2009]*.

Key 241 pt filter, as published in *[Key_2009]*; taken from file *FilterModules.f90* from *[Key_2009]*, available on marineemlab.ucsd.edu/Projects/Occam/1DCSEM. License: Apache License, Version 2.0, http://www.apache.org/licenses/LICENSE-2.0.

empymod.filters.**key_401_2009**()
Key 401 2009: *[Key_2009]*.

Key 401 pt filter, as published in *[Key_2009]*; taken from file *FilterModules.f90* from *[Key_2009]*, available on marineemlab.ucsd.edu/Projects/Occam/1DCSEM. License: Apache License, Version 2.0, http://www.apache.org/licenses/LICENSE-2.0.

empymod.filters.**key_51_2012**()
Key 51 2012: *[Key_2012]*.

Key 51 pt filter, taken from file *kk51Hankel.txt* from *[Key_2012]*, published by the Society of Exploration Geophysicists; software.seg.org/2012/0003. License: http://software.seg.org/disclaimer.txt.

empymod.filters.**key_601_CosSin_2009**()
Key 601 CosSin 2009: *[Key_2009]*.

Key 601 pt filter, as published in *[Key_2009]*; taken from file *FilterModules.f90* from *[Key_2009]*, available on marineemlab.ucsd.edu/Projects/Occam/1DCSEM. License: Apache License, Version 2.0, http://www.apache.org/licenses/LICENSE-2.0.

empymod.filters.**key_81_CosSin_2009**()
Key 81 CosSin 2009: *[Key_2009]*.

Key 81 pt filter, as published in *[Key_2009]*; taken from file *FilterModules.f90* from *[Key_2009]*, available on marineemlab.ucsd.edu/Projects/Occam/1DCSEM. License: Apache License, Version 2.0, http://www.apache.org/licenses/LICENSE-2.0.

empymod.filters.**kong_241_2007**()
Kong 241: *[Kong_2007]*.

Kong 241 pt filter, as published in *[Kong_2007]*; taken from file *FilterModules.f90* from *[Key_2009]*, available on marineemlab.ucsd.edu/Projects/Occam/1DCSEM. License: Apache License, Version 2.0, http://www.apache.org/licenses/LICENSE-2.0.

empymod.filters.**kong_61_2007**()
Kong 61: *[Kong_2007]*.

Kong 61 pt filter, as published in *[Kong_2007]*; taken from file *FilterModules.f90* from *[Key_2009]*, available on marineemlab.ucsd.edu/Projects/Occam/1DCSEM. License: Apache License, Version 2.0, http://www.apache.org/licenses/LICENSE-2.0.

2.5 utils – Utilites

This module consists of four groups of functions:

0. General Settings
1. Class EMArray
2. Input parameter checks for modelling
3. General utilities

Group 0 is to set minimum offset, frequency and time for calculation (in order to avoid divisions by zero). Group 2 are checks organised in modules. So if you create for instance a modelling-routine in which you loop over frequencies, you have to call *check_ab*, *check_model*, *get_coords*, *check_depth* and *check_hankel* only once, but *check_frequency* in each loop. You do not have to run these checks if you are sure your input parameters are in the correct format.

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 *imagpart* by -1 before passing through this function.

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}

Level of verbosity.

Returns **ab_calc** : int

Adjusted source-receiver configuration using reciprocity.

msrc : bool

If True, src is magnetic; if False, src is electric.

mrec : bool

If True, rec is magnetic; if False, rec is electric.

empymod.utils.**get_abs_srcbipole**(*msrc*, *recdir*, *theta*, *phi*, *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 **msrc** : bool

True if src is magnetic, else False.

recdir : {1, 2, 3, 4, 5, 6}

Receiver direction.

theta : float

Horizontal source angle.

phi : float

Vertical source angle.

verb : {0, 1, 2}

Level of verbosity.

Returns **ab** : array of int

ab's to calculate for this bipole.

mrec : bool

Deduced from recdir. Receiver is magnetic if True.

fact : array

Geometrical spreading factors for ab's.

`empymod.utils.`**`check_model`**(*depth*, *res*, *aniso*, *epermH*, *epermV*, *mpermH*, *mpermV*, *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 : array_like

Horizontal electric permittivities epsilon_h (-); #epermH = #res.

epermV : array_like

Vertical electric permittivities epsilon_v (-); #epermV = #res.

mpermH : array_like

Horizontal magnetic permeabilities mu_h (-); #mpermH = #res.

mpermV : array_like

Vertical magnetic permeabilities mu_v (-); #mpermV = #res.

verb : {0, 1, 2}

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 : array

As input, checked for size. If None, defaults to an array of ones.

epermV : array

As input, checked for size. If None, defaults to an array of ones.

mpermH : array

As input, checked for size. If None, defaults to an array of ones.

mpermV : array

As input, checked for size. If None, defaults to an array of ones.

isfullspace : bool

If True, the model is a fullspace (res, aniso, epermH, epermV, mpermM, and mpermV are in all layers the same).

empymod.utils.**check_pole**(*inp*, *name*, *verb*, *intpts=-1*)

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}

Level of verbosity.

intpts : int

Number of integration points for bipole.

Returns **out** : list

List of pole coordinates [x, y, z].

outbp : {tuple, None}

- If pole is a dipole, None.
- **If pole is a bipole, (theta, phi, g_w):**
  - theta : Horizontal pole angle.
  - phi : Vertical pole angle.
  - g_w : Integration weights.

empymod.utils.**get_coords**(*src*, *rec*, *verb*, *intpts=(-1, -1)*)

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** : list of floats or arrays

Source coordinates (m):

- dipole: [src-x, src-y, src-z]
- bipole: [src-x0, src-x1, src-y0, src-y1, src-z0, src-z1]

rec : list of floats or arrays

Receiver coordinates (m):

- dipole: [src-x, src-y, src-z]
- bipole: [src-x0, src-x1, src-y0, src-y1, src-z0, src-z1]

verb : {0, 1, 2}

Level of verbosity.

intpts : tuple (int, int)

Number of integration points for bipole for (src, rec).

- nr < 0 : dipole
- 0 <= nr < 3 : bipole, but calculated as dipole at centerpoint
- nr >= 3 : bipole

Returns **zsrc** : array of float

Depth(s) of src (plural only if bipole).

zrec : array of float

Depth(s) of rec (plural only if bipole).

off : array of floats

Offsets

angle : array of floats

Angles

nsrc: int

Number of bipole sources

nrec: int

Number of receivers

srcrecbp: tuple

(srcbp, recbp) If src/rec is dipole: None If src/rec is bipole: tuple containing (theta, phi, g_w)

`empymod.utils.`**`check_depth`**(*zsrc*, *zrec*, *depth*)

Check layer in which source/receiver reside.

Note: If zsrc or zrec are 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 **zsrc** : array of float

Depth(s) of src (plural only if bipole).

zrec : array of float

Depth(s) of rec (plural only if bipole).

depth : array

Depths of layer interfaces.

Returns **lsrc** : int or array_like of int

Layer number(s) in which src resides (plural only if bipole).

lrec : int or array_like of int

Layer number(s) in which rec resides (plural only if bipole).

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'}

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}

Level of verbosity.

Returns ht, htarg

Checked if valid and set to defaults if not provided.

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 : array_like

Horizontal electric permittivities epsilon_h (-); #epermH = #res.

epermV : array_like

Vertical electric permittivities epsilon_v (-); #epermV = #res.

mpermH : array_like

Horizontal magnetic permeabilities mu_h (-); #mpermH = #res.

mpermV : array_like

Vertical magnetic permeabilities mu_v (-); #mpermV = #res.

verb : {0, 1, 2}

Level of verbosity.

Returns **freq** : float

Frequency, checked for size and assured min_freq.

etaH : array

Parameters etaH, same size as provided resistivity.

etaV : array

Parameters etaV, same size as provided resistivity.

zetaH : array

Parameters zetaH, same size as provided resistivity.

zetaV : array

Parameters zetaV, same size as provided resistivity.

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}

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_time**(*freqtime*, *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 **freqtime** : array_like

Frequencies f (Hz) if *signal* == None, else 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'}

Flag for Fourier transform, only used if *signal* != None.

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}

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.**printstartfinish**(*verb*, *inp=None*)

Print start and finish with time measure.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

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.

[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.

[Gosh_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.

[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; Website of FFTLog: 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, F1–F18; DOI: 10.1190/geo2013-0411.1; Software: 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, F9–F20; DOI: 10.1190/1.3058434. Software: marineemlab.ucsd.edu/Projects/Occam/1DCSEM.

[Key_2012] Key, K., 2012, Is the fast Hankel transform faster than quadrature?: Geophysics, 77, F21–F30; DOI: 10.1190/GEO2011-0237.1; Software: 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.

[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.

[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.

[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.

[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.

[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.

[Wynn_1956] Wynn, P., 1956, On a device for computing the $e_m(S_n)$ tranformation: Math. Comput., 10, 91–96; DOI: 10.1090/S0025-5718-1956-0084056-6.

e

W