comet Documentation

Release 1.1.2

Nico Skibbe

Oct 23, 2021

Installation

Version: 1.1.2

CHAPTER 1

COupled Magnetic resonance and Electrical resistivity Tomography (**COMET**)

The code developed in this project has the main goal to invert magnetic resonance data, particularly in 2D, utilizing the resistivity information of 2D ERT measurements.

GitLab repository: https://gitlab.com/Skibbe/comet/

Documentation: https://comet-project.readthedocs.io/

1.1 Third-party package availability and dependencies

pyGIMLi: The documentation and the repository can be accessed at https://www.pygimli.org/.

TetGen: The 3D tetrahedral mesh generator http://tetgen.org. Available using conda (see below).

custEM: The custEM repository is located at https://gitlab.com/Rochlitz.R/custEM/, the documentation at https://custem.readthedocs.io/en/latest/.

1.2 Other Dependencies

The complete list of dependencies are listed below. Note that almost all dependencies are shared with **pyGIMLi** as well, so installation of **pyGIMLi** or **custEM** usually meets all requirements for **COMET** as well. If working on linux, the installation of **custEM** (and implicetly all other requirements including **comet**) via conda is highly recommended.

- python >= 3.8
- numpy >= 1.17 # The version depends on the pygimli installation as they share binary files (main reason why conda installation is recommended)
- pyGIMLi >= 1.2
- TetGen
- h5py

- scipy
- matplotlib

Optional (Linux only):

- custEM >= 1.0

Note: An installation of custEM actually requires comet, so a simple installation of custEM will usually take care of everything for you.

1.3 Installation

The easiest way of handling the dependencies is to use the conda (Anaconda or miniconda) system. For the full installation guide please read the installation instructions on the main documentation webpage of the project: Installation

CHAPTER 2

License

2.1 Installation

2.1.1 Windows

This is the most easy installation using the conda environment. If conda is not used, please see the installation "from source" chapter.

After installation of anaconda or miniconda, open the *anaconda navigator*. On the left side you can switch from the *Home* tab to the *Environments* tab. Under the *base (root)* environment click *create* to create a new environment using python=3.7. Click on the new environment to switch to it, the right side shows the installed packages and should be nearly empty. Click on the green triangle right to the environment name and select *Open Terminal*. Proceed with the following conda commands to install packages.

```
conda install -c gimli -c conda-forge comet=1.1
```

Switching back from *Environments* to *Home* Anaconda provides some apps, one of them being *Spyder*. Click install and then start Spyder from here to make sure the correct environment is used (it is highlighted at the top next to "Applications on"). This should not interfere with previously installed python packages, due to the own environment. With Spyder installed you can proceed with the tutorials or examples. The tutorials are also available as Jupyter Notebook. The Notebook can be installed the same way as Spyder.

Note that on Windows **COMET** is restricted to 1D resistivity since there is no Windows version of the underlying FEniCS library that is used by **custEM**. For using 2D resistivities on Windows 10, consider the Windows Subsystem for Linux.

The Anaconda Navigator can be used to install packages. Since **COMET** and **pyGIMLi** are hosted in the channel gimli and **TetGen** in the channel conda-forge. These channels must be added to the channel list before installing **COMET** along with **pyGIMLi** and **TetGen**.

2.1.2 Linux

Miniconda or Anaconda is required to install the package via conda. The installation line for **comet** should look like this:

For the handling of 2D resistivities **custEM** is required. Compatibility is ensured for version 0.99.14 and newer. Newer versions will be tested added as soon as they are available.

```
conda create -n cometcustem -c gimli -c conda-forge -c anaconda custem=0.99.14
comet
```

This creates a new environment with the name "cometcustem" and installs the **COMET** package as well as the **custEM** and all dependencies. Note that the -c argument are adding the gimli (for **COMET**, **pyGIMLi**, and **custEM**) and conda-forge (for **TetGen**) channels, otherwise another package with the name comet will be downloaded instead (not a geophysical package at all). Versions for python 3.6 and 3.7 (technically 3.8 as well, however those are untested) are provided. We highly recommend python 3.7 for now. You can activate the environment using the following line:

```
conda activate cometcustem
```

Please add `export OMP_NUM_THREADS=1` to your `.bashrc`.

If any problems are encountered installing **custEM**, please consult https://custem.readthedocs.io/en/latest/.

For a **COMET** version without **custEM** instead use the following command for installation (same as for windows):

```
conda create -n comet11 -c gimli -c conda-forge -c anaconda comet=1.1
```

This creates a new environment with the name "comet11" (the names are your choices really. . .) and installs the comet package and all dependencies. Then activate environment via:

```
conda activate comet
```

Again if conda is not used, see installation "from source".

2.1.3 MacOS

Unless using a virtual Linux machine, MacOS is currently not supported. COMET without custEM could work if using the installation from source, however, this is not tested.

2.1.4 From Source

Installation of the **comet** python package is easy as only python code is involved. No compiling is needed, a clean copy-paste or *git clone* of the git repository is sufficient. Clone the **COMET** repository manually using git or download the zip archive from the project page `git clone https://gitlab.com/Skibbe/comet/` The PYTHONPATH has to be set to find the directory containing the __init__.py (see below). **No** execution of the setup.py is required.

If this way of installation is chosen and a **custEM** installation is aimed as well (Linux only), we recommend to install **custEM** first (it brings all compatibilities for **comet** as well, see weblink above). You can additioanlly clone **COMET** as mentioned above and set the *PYTHONPATH* accordingly if you want to work with comet from the source and still not having to deal with dependancies.

If **custEM** is not installed or also cloned from gitlab, than **pyGIMLi** and **TetGen** need to be installed additionally, either by using binary installers or building from source. Please be referred to the webpages linked in the "Where to find the third party packages" section.

The PYTHONPATH needs to be set as global environment variable (or locally inside Spyder using PYTHONPATH manager).

To set the PYTHONPATH directly add the following line to your ~/.bashrc directly (with appropriate path, Linux).

```
export PYTHONPATH=$PYTHONPATH:$HOME/some_directory/comet
```

Setting the PYTHONPATH can also be done in Spyder directly (Tools -> PYTHONPATH manager -> Add path). We suggest to close Spyder afterwards and re-open it as the built-in synchronization will not always work.

2.1.5 Where to find the third party packages

A **pyGIMLi** installation can be found here: https://www.pygimli.org/installation.html. There is also the support for prebuild binary installers for windows, if conda is not used. However this does not include a **TetGen** installation. Also follow the installation instructions of **pyGIMLi** concerning the ~/.bashrc.

A **TetGen** installation can be found here: http://tetgen.org. If not using conda, you would have to build **TetGen** using the source files. Add the **TetGen** directory to the PATH variable if **TetGen** was build from source.

A **custEM** installation can be found here: https://custem.readthedocs.io/en/latest/install.html.

2.2 Tutorials

2.2.1 Tutorial 1 Loop

2.2.2 Tutorial 2 Survey and FID

2.2.3 Tutorial 3 Kernel

2.3 Classes

2.3.1 Loop

class comet.pyhed.loop.**Loop**(*Input*, *config=None*, *verbose=False*)

Class for the computation of arbitrary shaped polygon loops. Some functions automatically return this loopclass as result. It is recommended to use these (you may take a look at the example)

Parameters

- **Input** (*string or raw loop class*) – Filename of a prior saved loopfile (recommended). Alternatively the output of the function **computeLoopPositions** (not recommended). For the latter case plenty of convenience functions are found in the the *loop* submodule of *pyhed* starting with "build". . . .
- **config** (*string or pyhed.config*) – Defines the configuration file for the loop.

Example

```
>>> # example: import
>>> loopclass = Loop('path/to/loopfile')
>>> # example: create circular loop
>>> loopclass = buildCircle(10, 12)  # 10 m radius, 12 dipoles
```

calcAndExportFieldsForFenics(*export_vtk=False*, *num_cpu=32*, ***kwargs*)
Calculates and export primary fields for fenics secondary field calculation.

Parameters **kwargs** (*dict*) – Keyword parameters are redirected to *calculate*.

calculate(*num_cpu=12*, *loop_mesh=None*, *dipole_mesh=None*, *interpolate=False*, *savename=None*, *cell_center=False*, *verbose=False*, *mode='auto'*, *matrix=False*, *field_matrix=None*, *max_node_count=None*, ***kwargs*)
Computation of the loop field with respect to the config.

Parameters

- **num_cpu** (*integer [12]*) – Maximum number of processes allowed for this task.
- **loop_mesh** (*string or mesh instance [None]*) – Optional. Possibility to give a user defined mesh for the calculation.
- **dipole_mesh** (*string or mesh instance [None]*) – Optional. Possibility to give a user defined mesh for the calculation (interpolate=True or matrix=True only).
- **interpolate** (*boolean [True]*) – The loop dipoles can either be calculated directly (False) or once on a seperated mesh (dipolemesh) and then interpolated to the loopmesh (True). If a dipoleFieldName is given, this field will be used for the interpolation.
- **savename** (*string [None]*) – Optional. If savename is not None, the loop will be saved under the name defined in savename.
- **cell_center** (*boolean [True]*) – A default the field of the loop will be calculated at the cell center of the mesh cells. This flag allows for calculation at the mesh nodes. Affects only the definition of the final loopmesh, the dipolemesh will always be calculated at the nodes for interpolation reasons.
- **verbose** (*boolean [False]*) – Turn on verbose mode.
- **mode** (*string ['auto']*) – Five posibilities: 'auto', 'config', 'te', 'tm', 'tetm'

  'auto': Automatic detection wether the loop is grounded or not. Grounded wires are calculated with te and tm mode (see HED). Non grounded wires are calculated with te mode only (sufficient).

  'config': the default config decides the mode the field is calculated in.

  'te', 'tm', 'tetm': Calculates the field in the choosen mode.
- **matrix** (*boolean [False]*) – Alternatively calculation approach. At first the field on a highly dense dipole mesh will be triggered. After that the field will be interpolated to the single dipole positions by the means of a matrix vector multiplication with a matrix containing appropriate weighting factors. This Approach takes longer than direct calculation in the first run, but the calculated matrix can be used for further calculations with different frequencies or resistivity models (as long as the loopmesh and dipolemesh remain the same).
- **field_matrix** (*list or string [None]*) – Interpolation matrices or file path if calculation with matrix=True. Will be calculated automatically if None.
- **max_node_count** (*integer [None]*) – As all points will be calculated at once, the computational effort scales lineary with the reciever count, the transmitter count and the used hankel factors. If the limits of the available memory is reached *max_node_count* can be used to define the maximum chunk of nodes to be computated at once. Other nodes will be computed afterwards.

Keyword Arguments

- **`arguments are redirected to loop.save and to define`** (*`Keyword`*) –
- **`drop_tol (float [ 1e-2 ]) in the cylindrical coordinate`** (*`the`*) –
- **`to avoid instabilities around the source.`** (*`transformation`*) –

`calculateDipoleField`(*verbose=False*, *drop_tol=0.01*, *num_cpu=12*, *max_node_count=None*)
Calculates field on dipole mesh.

Parameters

- **verbose** (*boolean [False]*) – Turn on verbose mode.
- **drop_tol** (*float [1e-2]*) – Singularity fix. All horizontal distances between *drop_tol* and the transmitter dipole are placed between the first reciever outside the tolerance and the tolerance, maintaining the correct order and angle.
- **num_cpu** (*integer [12]*) – Maximum number of processes allowed for this task.
- **max_node_count** (*integer [None]*) – As all points will be calculated at once, the computational effort scales lineary with the reciever count, the transmitter count and the used hankel factors. If the limits of the available memory is reached *max_node_count* can be used to define the maximum chunk of nodes to be computated at once. Other nodes will be computed afterwards.

`calculateFieldFromMatrix`()
Calculates the primary field on basis of the interpolation matrix and the dipole field.

`calculateFieldMatrix`(*num_cpu=8*, *verbose=False*)
If wished the calcualtion of the total loop field can be done by interpolation and superposition of one highly accurate dipole field to the different transmitter positions of the loop. This is done either done directly or via a vector matrix multiplication.

This function is called to initialize and append the weights to the interpolation matrix from the *dipolemesh* to the *loopmesh* for all tx positions with respect to *pos*, *phi*, and *ds*.

This function will be called if *calculate* is called with *matrix=True*.

Parameters

- **num_cpu** (*integer [8]*) – Define the maximum number of cores allowed for this operation.
- **verbose** (*boolean [False]*) – Turn on verbose mode.

`calculateInterpolationMatrix`(*Pos*)
Calculates the interpolation matrix.

If one wished the field can be interpolated to another mesh. The interpolation matrix from the loopmesh to an arbitrary set of coordinates is calculated with this function. This function is called to initialize and append the weights to the interpolation matrix.

Note: The loop class does not hold a reference of the resulting matrix, instead gives it back to the caller.

Parameters **Pos** (*np.ndarray or pg.core.PosVector*) – Transmitter positions of shape (n, 3) with n positions. Values are expected to be floats (the conversion to a pg.PosVector will not check again).

Returns **mat** – Sparse interpolation matrix with number of columns equal to the number of nodes in the loopmesh and number of rows equal to the number of input positions.

Return type pg.core.SparseMapMatrix

calculateSecField(*num_cpu=8*, ***kwargs*)
Calculates the secondary field using custEM.

Calculates primary field as well if not found.

Needs a FEM suited mesh as well as a parameter distribution provided by other functions of this class (See *createFEMMesh* and *prepareSecondaryFieldCalculation*).

Parameters

- **num_cpu** (*integer [8]*) – Maximum number of processes allowed for this task. The actual calculation will be done in an mpirun environment with the selected number of cores.
- **kwargs** (*dict*) – Keyword arguments are redirected to *local_apps*.

createDefaultSecondaryConfig(*base=None*, *prefix=''*, *suffix=''*, *m_dir='.'*, *r_dir='.'*)
Short cut to generate a secondary config with some default params.

Parameters

- **prefix** (*string*) – String to be added to the *getDefaultLoopMeshBaseName* string to define the automatic generated names for the default secondary config.
- **suffix** (*string*) – String to be added to the *getDefaultLoopMeshBaseName* string to define the automatic generated names for the default secondary config.

createDipoleMesh(*quadratic=True*, *savename='_default_dipole_mesh.bms'*, *save=False*, *verbose=False*)
Creates a suitable dipole mesh for calculation via a single dipole.

Parameters

- **quadratic** (*boolean [True]*) – If chosen, uses a quadratic (2nd order) mesh for dipole calculation.
- **savename** (*string ['_default_dipole_mesh.bms']*) – Define output name.
- **save** (*boolean [True]*) – Additional save of dipole mesh under *savename*.
- **verbose** (*boolean [False]*) – Turn on verbose mode.

createFEMMesh(*para_mesh_2d=None, savename=None, exportVTK=False, exportH5=True, box_x=[None, None], box_y=[None, None], box_z=None, box_cell_size=None, source_poly=None, source_setup='edges', source_loops=None, inner_area_cell_size=0.3, outer_area_cell_size=10, subsurface_cell_size=None, poly_2d=None, number_of_loops=None, **kwargs*)
Builds the FEM mesh for the secondary field computation.

Needs at least on of the two possible parameter meshes in order to continue.

para_mesh_2d: string or pg.Mesh [None] Used to get the outer dimensions of the FEMMesh.

savename: string [None] Define output save name of FEM mesh. Default name will be generated if None. If no savename is given, the dafaultname will be '_default_LoopMesh' + looptype + number of dipoles.

exportVTK: boolean [False] Turn on optional vtk export.

source_setup: string ['edges'] Defines the way the sources are incorporated into the mesh. "nodes" simply insert the dipole positions (fallback), "edges" defines strait edges between the nodes (usually the best approach). "etra" can be used for a special setup where multiple loops are build in an elongated transmitter with inline receiver array. Raises an exception if source_setup differs from the three options.

source_loops: list [None] If a list of loop classes is given, their tx representation after custEM is implemented in the mesh for custEM magnetic field calculations using automatic source detection.

inner_area_cell_size: float [0.3] Maximum allowed area (m^2) for all cell in the source plane within the source polygons (if closed loop). Very important for kernel calculation! See tutorial for custEM for further explanations.

outer_area_cell_size: float [10] Maximum allowed area (m^2) for all cells in the source plane outside the source polygons (or anywhere for not closed loop). See tutorial for custEM for further explanations.

subsurface_cell_size: float [None] Maximum allowed volume (m^3) for all cells within inner mesh box (not the tetrahedron boundary to 10 km). Optional.

limits: list of len 2 [None] Minimum and maximum y value, the anomalies should be set in the fem mesh. Uses the x limits of the 2D parameter mesh as default if *None*.

custEM: Install via conda on Linux only. See install instructions of comet.

`createLoopMesh`(*savename=None*, *exportVTK=False*, *airspace=False*, *verbose=False*, *xmax=None*, *xmin=None*, *ymax=None*, *ymin=None*, *zmin=None*)

Builds the mesh where the loop will be calculated in.

savename: string [None] Saves the created mesh under savename, as long as savenmae is not none. If no basename is given, the dafaultname will be '_default_LoopMesh' + looptype + number of dipoles + '.bms'.

exportVTK: boolean [False] Switch to export the resulting mesh to a vtk with the given savename.

airspace: boolean [False] Enables airspace.

verbose: boolean [False] Turn on berbose mode.

`createSecondaryConfig`(*mod_name*, *mesh_name*, *m_dir='.'*, *r_dir='.'*, *pf_name=None*, *p2=False*, *approach='E_s'*, *pf_EH_flag='E'*)

Initializes an instance of a secondary config for use of custEM.

Parameters

- **mod_name** (*string*) – Name of the mod instance (for saving and import in mpi environment)
- **mesh_name** (*string*) – Basename of mesh imported by the fenics functions (.h5). Mind the subfolder '/_h5' that will be added to the string.
- **m_dir** (*string*) – Path to mesh directory of custEM.
- **r_dir** (*string*) – Path to result directory of custEM.
- **pf_name** (*string*) – File name under which the primary field will be saved in the appropriate directory of custEM.

`effectiveArea`()

Returns *self.area* * *self.turns* (0 for not closed loops).

`exportFenicsHDF5Mesh`(*save_h5*, *dipole_mesh=False*, ***kwargs*)

Exports the mesh in a h5 file. Can save the loopmesh or the dipole mesh seperately.

Need pygimli to work.

Parameters

- **save_h5** (*string*) – Filename of the resulting h5 mesh (hdf5 data container in fenics syntax).

- **dipole_mesh** (*boolean [False]*) – Save dipole mesh instead of loop mesh (Call this function twice if you want to save both meshes).
- **kwargs** (*dict*) – Keyword arguments are redirected to *pygimli.meshtools.exportFenicsHDF5Mesh*

`exportVTK`(*save_vtk*, *secondary=False*, ***kwargs*)
Exports the field in a vtk file.

Uses the *loopmesh* to save *field* with default configurations in a vtk file.

Parameters

- **save_vtk** (*string*) – Filename of the resulting vtk file.
- **kwargs** (*dict*) – Keyword arguments are redirected to the function *pyhed.IO.savefieldvtk*.

`getDefaultLoopMeshBaseName`()
Returns string with default base name of the loop mesh.

`initCustEM`(*secondary_config=None*, *init_primary_field_class=True*, *procs_per_proc=2*)
Initalizes instance of custEM mod class for FEM calculation.

Parameters

- **secondary_config** (*string or pyhed.SecondaryConfig [None]*) – Initialized secondary config class to be used for the mod instance or path to corresponding file containing the secondary config. Uses secondary_config over *loop.secondary_config*. Throws Exception if both values are None.
- **init_primary_field_class** (*boolean [True]*) – Additionally initializing the primary field class of the mod class instance (used for primary field export).

`load`(*savename=None*, *config=None*, *config2=None*, *verbose=True*, *load_meshes=True*, *overwrite_dir=False*)
Load Loop from files.

Parameters

- **savename** (*string [None]*) – Basename of the lop class files. Other names are autogenerated using this basename.
- **config** (*string [None]*) – Tell the load function to explicitely load config from given path. Else the saved filepath in the main archive is used.
- **config2** (*string [None]*) – See *config*, but for secondary configuration.
- **verbose** (*boolean [True]*) – Turn on verbose mode.
- **load_meshes** (*boolean [True]*) – If originally saved, the meshes are loaded by default. However, this takes more time then the rest of the load function and can be ommitted if only the other parts are of interest.

`loadFieldMatrix`(*name*, *verbose=True*)
Loads the three matrices needed for recalculation of the primary field from numpy archive. See saevFieldmatrix for detailed description.

Parameters

- **name** (*string*) – Path to file to be loaded.
- **verbose** (*boolean [True]*) – Turn on verbose mode.

`loadSecondaryConfig`(*savename=None*)
Imports previously saved secondary config.

Parameters **savename** (*string [None]*) – Used savename over *loop.sec_savename*. Throws Exception if both values are None. Replaces *loop.sec_savename*.

prepareSecondaryFieldCalculation(*savename=None*, *secondary_config=None*, *fem_mesh=None*, *para_mesh_2d=None*, *set_marker=False*, *anomaly_vector=None*, *valid_marker=None*, *verbose=False*, *num_cpu=32*, *force_primary=False*, *export_vtk=False*, *mod_name=None*, ***kwargs*)

Based on the given secondary config a MOD instance using the third party module custEM will be initialized. This includes the optional generation of a FEM suited mesh containing resistivity information from a 2D parameter mesh.

Parameters

- **savename** (*string [None]*) – Name under which loopclass and secondary config (+= '_sec.cfg') are to be saved. Needed for secondary approach.
- **secondary_config** (*pyhed.SecondaryConfig or string [None]*) – Filename of configuration file or initialized class instance of a secondary configuration. Optional if already given manually.
- **fem_mesh** (*pg.Mesh or string [None]*) – FEM suited mesh or filename, respectively. Optional. If not given a suited mesh will be generated if a valid para_mesh_2d is provided.
- **para_mesh_2d** (*pg.Mesh or string [None]*) – 2D parameter mesh providing cell indices for the appending of resitivity information. Needed for automatic FEM mesh generation. Can be set manually beforehand.
- **set_marker** (*boolean [True]*) – Flag to decide if the fem mesh has got the needed marker for the resitivity distribution. Can be omitted if already done and saved (e.g. if same mesh is used again).
- **anomaly_vector** (*np.ndarray [None]*) – Conductivity values [S/m] of the parameter mesh to be used in the seocondary field approach. Uses given value over array found in secondary config. Raises Exception if neither found nor given.
- **ground_marker** (*np.ndarray [None]*) – Corresponding marker for each entry in the anomaly vector. Each marker corresponds to a layer number of the 1d primary field beginning at 1 for the first layer, counting upward (0 belongs to the air layer). None results in np.ones_like(anomaly_vector, dtype=int).
- **verbose** (*boolean [False]*) – Turn on verbose mode.
- **num_cpu** (*integer [32]*) – Maximum number of processes allowed for this task.
- **force_primary** (*boolean [False]*) – Force a recalculation of the primary field.
- **mod_name** (*string or None [None]*) – Overrides mod name. Useful if looping over many loops, as default name could be similar.
- **magnetic** (*boolean [True]*) – Prepares magnetic primary fields. If False only dummies are created to avoid error messages from custEM during import. Set to False if secondary electric approach is used for secondary field calculation.
- **electric** (*boolean [True]*) – Prepares electric primary fields. If False only dummies are created to avoid error messages from custEM during import. Set to False if secondary magnetic approach is used for secondary field calculation.
- **Returns**
- **———**

- **tuple** (*(savename, sec_savename)*) – Absolute file paths for the secondary approach.
- **Usage**
- **——**
- **In order to prepare a secondary field calculation you need**
- **- a secondary config (default is provided)**
- **- a conductivity vector (*)**
- **- a 2d parameter mesh matching the anomalies (*)**
- **- a marker_vector (*)**
- ***if not in secondary config or proviedd beforehand**
- **and optionally either**
- **- fem_mesh (without marker -> set_marker=True (default))**
- **or**
- **- fem_mesh (with marker -> set_marker=False)**
- **or**
- **- no fem_mesh (auto creation)**

`save`(*savename=None*, *config_savename=None*, *config2_savename=None*, *save_mesh=True*, *save_field=True*)

Saves the loop class in files.

Saves npz archive with loop itself.

Saves config.

Saves secondary config if initialized.

Saves mesh if save_mesh=True.

Saves field if save_field=True.

Parameters

- **savename** (*string [None]*) – File basename for saving loop class and its components.
- **config_savename** (*string [None]*) – Explicit savename for config. Automatically generated if None.
- **config2_savename** (*string [None]*) – Explicit savename for secondary config. Automatically generated if None.
- **save_mesh** (*boolean [True]*) – Saves mesh.
- **save_field** (*boolean [True]*) – Saves fields.

`saveFieldMatrix`(*name*, *verbose=True*)

Saves the three matrices needed for recalculation of the primary field.

A compressed numpy archive is loaded and the matrices are build afterwards, therefore import time is ~20% higher compared to the pure pygimli way (*.field_matrix.save('… ')*). However, because the single arrays (indices and values) are saved in one compressed file archive they need only one third space on the hard disk compared to saving three separate matrices using pygimli syntax.

Parameters

- **name** (*string*) – Path for file to be saved.

- **verbose** (*boolean [True]*) – Turn on verbose mode.

`saveLoopMesh`(*savename=None*)

Saves loopmesh using the given savename or an autogenerated name.

Updates *self.loop_mesh_name* in case of changes.

Parameters **savename** – Export path name. Used over default name if given.

`saveSecondaryConfig`(*savename=None*)

Saves secondary config in ASCII file.

Parameters **savename** (*string [None]*) – Used savename over *loop.sec_savename*. Throws Exception if both values are None. Replaces *loop.sec_savename*.

`setAnomalies`(*anomaly*, *sort=True*)

Handle anomaly vector and marker of the 2d parameter mesh.

Parameters

- **anomaly** (*array_like [None]*) – Vector with conductivities in S/m. Expect one entry for each cell in parameter mesh.
- **sort** (*boolean [False]*) – If True, set the same marker for double values in anomaly vector. This is for blocky 2d structures, where only a few different regions are required. Use default False if dealing with smooth inversion results, for example in a structural coupling.

`setDipoleMesh`(*mesh*, *savename='_default_dipole_mesh'*, *verbose=True*)

Sets the dipolemesh and saves it under savename.

Parameters

- **mesh** (*string or mesh instance*) – Pygimli mesh instance or file path to pygimli mesh.
- **savename** (*string [None]*) – Used savename for mesh, if mesh is already a mesh instance.
- **verbose** (*boolean [False]*) – Turn on verbose mode.

`setFEMMarker_old`(*valid_marker=None*)

Sets and checks the domain marker of the 3D FEM mesh.

Parameters **valid_marker** (*array_like [None]*) – If None, checks which domains of the 2D mesh are actually transferred to the 3D FEM mesh. The markers are saved in the *valid_marker* attribute. If given, sets vector directly after some checks.

`setFEMMesh`(*mesh*, *valid_marker=None*, *savename=None*)

Sets the FEM mesh as loopmesh and handles the domain markers.

Parameters

- **mesh** (*string or mesh instance*) – Pygimli mesh instance or file path to pygimli mesh.
- **valid_marker** (*array_like [None]*) – If None, checks which domains of the 2D mesh are actually transferred to the 3D FEM mesh. The markers are saved in the *valid_marker* attribute. If given, sets vector directly after some checks.
- **savename** (*string [None]*) – Useful if multiple loops are using the same mesh (saves diskspace). Ignored if *mesh* is a string already.
- **Calls *_setFEMMarker* is paramesh has been set.**
- **Furthermore calls *updateFEMAnomaly* if anomaly has been set through**
- **either *setParamesh2D* or *setAnomaly***

- **Produces error message if valid_marker array is given, but no paramesh**
- **is found**

`setFEMMesh_old`(*mesh*, *valid_marker=None*, *savename=None*)
Sets the FEM mesh as loopmesh and handles the domain markers.

Parameters

- **mesh** (*string or mesh instance*) – Pygimli mesh instance or file path to pygimli mesh.
- **valid_marker** (*array_like [None]*) – If None, checks which domains of the 2D mesh are actually transferred to the 3D FEM mesh. The markers are saved in the *valid_marker* attribute. If given, sets vector directly after some checks.
- **savename** (*string [None]*) – Useful if multiple loops are using the same mesh (saves diskspace). Ignored if *mesh* is a string already.

`setFrequency`(*frequency*)
Sets the frequency, not angular frequency for the field calculation.

`setLoopMesh`(*mesh*, *savename=None*)
Sets the loopmesh.

Parameters

- **mesh** (*string or mesh instance*) – Pygimli mesh instance or file path to pygimli mesh.
- **savename** (*string [None]*) – Used savename for mesh, if mesh is already a mesh instance. Alternatively a default name is generated with *getDefaultLoopMeshBaseName*.

`setLoopMeshName`(*savename=None*)
Sets loop mesh name or figures it out from sec config.

`setMeshParameters`(*refinement_para=1.0*, *max_area_factor=1.0*, *tetgen_quality=1.2*)
Alters the Parameter responsible for the quality and size used during automatic mesh generation.

Parameters

- **refinement_para** (*float [1]*) – An increase of refinement_para decreases the size of the smallest cell at the dipoles and therefore incrreases the total number of refinement cells around the dipole. Omitts refinement if value is negative.
- **max_area_factor** (*positive float [1]*) – The max_area_para lineary affects the maximum volume of a cell. An increase of the parameter allows for greater cells and therefore decreases the total number of cells outside of the refined section of the mesh. Set to 0.5 for a fine mesh and anywhere near 2 for a coarse mesh. Highly affects the total number of nodes/cells in the mesh.
- **tetgen_quality** (*float [1.2]*) – The tetgen_quality parameter is directly piped to the corresponding tetgen call in the meshgeneration process. Decrease this parameter (e.g. to 1.12) to increase the homogeneity of the triangles. Be careful with this one, tetgen very easy starts to split cells in smaller and smaller pieces and therefore increase the total cellcount to very high values (millions and more).

`setModel`(*rho*, *d=None*, *thickness=True*, *resistivity=True*)
Sets the synthetic 1D layered earth model for dipole calculation.

Parameters

- **rho** (*float or array_like*) – Resistivity/conductivity distribution for a layered earth.
- **d** (*float or array_like or None [None]*) – Thickness or layer depth. Empty (None, 0, or []) for halfspace.

- **thickness** (*boolean [True]*) – The parameter d is used as thickness (True, len(rho) - 1) or depth (False, len(rho)), respectively.
- **resistivity** (*boolean [True]*) – The parameter rho is used as Resistivity (True) or conductivity (False), respectively.

`setParaMesh2D` (*para_mesh_2d*, *limits=None*, *append_boundary=False*, *preserve_edges=False*, *anomaly=None*, *sort=True*, ***kwargs*)

Sets 2D parameter mesh for secondary field calculation.

Parameters

- **para_mesh_2d** (*string or pg.Mesh*) – 2D parameter mesh or path to mesh.
- **limits** (*[float, float] or None*) – Minimum and maximum values for y of the area where 2D parameters are to be transferred to the 3D FEM mesh. Default are the x extension of the 2D parameter mesh.
- **append_boundary** (*boolean [False]*) – Fills in an additional boundary with prolongated resistivity values around the transferred 2D values. This is useful as it reduces artifacts at the edge of the 2D domain oin the FEM mesh.
- **anomaly** (*None or np.ndarray [None]*) – Optionally. Alternatively use *setAnomalies*. Anomaly vector (conductivity vector) with values for each cell in the 2D parameter domain. Attention: conductivity is used, not resistivity!
- **sort** (*boolean [False]*) – Optionally. Alternatively use *setAnomalies*. If True, set the same marker for double values in anomaly vector. This is for blocky 2d structures, where only a few different regions are required. Use default False if dealing with smooth inversion results, for example in a structural coupling.
- **kwargs to *appendTriangleBoundary***
- **Calls *setAnomalies* of anomaly is given.**
- **Furthermore calls *updateFEMAnomaly* if FEMMesh has been set already.**

`setParaMeshMarkerAndVals` (*anomaly=None*, *sort=True*)

Handle anomaly vector and marker of the 2d parameter mesh.

Parameters

- **anomaly** (*array_like [None]*) – Vector with conductivities in S/m. Expect one entry for each cell in parameter mesh. If not given, and sort is True an error is raised.
- **sort** (*boolean [False]*) – If True, set the same marker for double values in anomaly vector. This is for blocky 2d structures, where only a few different regions are required. Use default False if dealing with smooth inversion results, for example in a structural coupling.

`setPrimaryConfig` (*config*)

Sets the primary config which handles the resistivity distribution as well as the frequency of the primary field. For setting the 1D model directly see *setModel*.

Parameters **config** (*path or comet.pyhed.config.Config instance*) – Configuration class instance or file path.

`setSecondaryConfig` (*secondary_config*)

Sets class attribute with secondary config or loads file.

Parameters **secondary_config** (*string or pyhed.SecondaryConfig*) – Seondary config class instance or file path.

show(***kwargs*)

Plots the Loopdiscretisation and the dipole directions and Length. For inspection of the loop-class and debugging purpose. Or for your curiosity.

Parameters kwargs (*dict*) – Keyword arguments are redirected to *pyhed.plot.plot_bib.showLoop*.

updateFEMAnomaly(*anomaly=None*, *set_marker=True*, *set_attributes=False*, *vtk_name=None*, *ground_marker=None*, *export_H5=False*, *sort=True*)

Transfers resistivity anomalies from 2D para mesh in FEM mesh.

Parameters

- **anomaly_vector** (*array_like [None]*) – Array containing the resistivity anomalies of the 2D parameter mesh. If None, the secondary config is asked for a anomaly vector. (For setting the marker for exmaple).
- **set_marker** (*boolean [True]*) – Transfers the marker from the parameter mesh to the FEM mesh. This only has to be done once and can then switched off for performance.
- **set_attribute** (*boolean [False]*) – Sets the attribute in the FEM mesh for debugging purposes. The anomaly vector for calculation is stored in secondary_config.
- **vtk_name** (*string [None]*) – Optional vtk export with name = *vtk_name* if *vtk_name* is not None.
- **ground_marker** (*array_like [None]*) – Corresponding marker for each entry in the anomaly vector. Each marker corresponds to a layer number of the 1d primary field beginning at 1 for the first layer, counting upward (0 belongs to the air layer). None results in np.ones_like(anomaly_vector, dtype=int).

updateFEMAnomaly_old(*anomaly=None*, *set_marker=True*, *set_attributes=False*, *vtk_name=None*, *ground_marker=None*, *export_H5=False*)

Transfers resistivity anomalies from 2D para mesh in FEM mesh.

Parameters

- **anomaly_vector** (*array_like [None]*) – Array containing the resistivity anomalies of the 2D parameter mesh. If None, the secondary config is asked for a anomaly vector. (For setting the marker for exmaple).
- **set_marker** (*boolean [True]*) – Transfers the marker from the parameter mesh to the FEM mesh. This only has to be done once and can then switched off for performance.
- **set_attribute** (*boolean [False]*) – Sets the attribute in the FEM mesh for debugging purposes. The anomaly vector for calculation is stored in secondary_config.
- **vtk_name** (*string [None]*) – Optional vtk export with name = *vtk_name* if *vtk_name* is not None.
- **ground_marker** (*array_like [None]*) – Corresponding marker for each entry in the anomaly vector. Each marker corresponds to a layer number of the 1d primary field beginning at 1 for the first layer, counting upward (0 belongs to the air layer). None results in np.ones_like(anomaly_vector, dtype=int).

2.3.2 Survey

class comet.snmr.survey.**Survey**(*earth=None*, *loops=None*)

Survey class for containment and handling of SNMR datasets (FIDS).

addLoop(*loop*)
Appends a given loop instance to the loops in survey and returns id

addSounding(*fid*)
Appends a given sounding instance to the sounds in survey and returns id

createKernel(*fid=0*, *dimension=1*)
Returns a initialized kernel instance for the chosen sounding.

Parameters

- **sound_index** (*integer*) – Index of the sounding the kernelclass is calcualting the kernel for. In order to calculate the kernel, pulses, tx and rx are taken as references from the sounding.
- **Note** (*createKernel does not set or change any values in survey nor in*)
- **the corresponding sounding. However when calculating, the kernel class**
- **will override the frequency in the given loops (tx and rx) and set it**
- **to the larmor frequency calculated from the earth magnetic fields**
- **magnitude. Use the *setEarth* method before or after you generate the**
- **kernel instances, but obviously before calculation.**

createSounding(*tx=0*, *rx=0*, *check_double=True*)
Creates a new sounding based on the given ids for tx and rx.

Parameters

- **tx** (*integer [0]*) – Index of the transmitter loop in loops.
- **rx** (*integer [0]*) – Index of the receiver loop in loops. Same number than tx indicates a coincident measurement.
- **check_double** (*boolean [True]*) – If True, omits creating another instance of the same fid (tx/rx combination). Instead the index of the original fid is returned. If False new fid is created and its index is returned.
- **Note** (*tx and rx indices can be setted regardless if there is an actual*)
- **loop in loops or just a *None* placeholder. In other words you can**
- **create your soundings and loops in arbitrary order.**

data
Complex data cube (pulses * gates) from soundings.

data_phases
Single data phases of the FIDs.

error
Complex error cube (pulses * gates) from soundings.

gates
Time gates gathered from soundings.

loadLoopMesh(*savename*, *indices=None*, *dipolename=None*)
Loads mesh and distribute reference to given indices.

loadMRSD(*filename*, *remove_df=True*, *build_loops=False*, *x_offsets=None*, *segments=80*, *max_length=None*, *tx=None*, *rx=None*, *fids=None*, *debug=False*)

Parameters

- **filename** (*string*) – Path to .mrsd file to be imported.
- **build_loops** (*boolean [True]*) – If True, the saved config in the mrsd file is used to construct loops for transmitter and receiver. However, the information in the mrsd fiel is not complete. There are some defaults we assume in autogenerating the loops, especially when it comes to figure-of-eight loops. Feel free to replace the loops with custom created loops of the *pyhed* library. Or switch this off if you only want to see the data or define all the loops yourself.
- **x_offsets** (*list or None [None]*) – One information that is missing in mrsd files, is the relative position of the loops to each other. Here one can fill in this information giving a simple list of offsets in positive x direction (all loops (midpoints) are placed at y=0 and z=0). Expect one float per used loop by the data file or raises an error. Ignored if None and multiple loops are found (in this case no loops are build at all). Coincident measurements do not require this, x is set to 0 by default.
- **segments** (*integer [80]*) – Number of dipoles used to auto build the loops. Ignored if *build_loops* is False or not given any *x_offsets*.
- **max_length** (*float [None]*) – Maximum length of a dipole when auto generating the loops. Overrides segments. Ignored if *build_loops* is False or not given any *x_offsets*.

`loadMRSD_h5`(*filename*, *remove_df=True*, *build_loops=False*, *x_offsets=None*, *segments=80*, *max_length=None*, *tx=None*, *rx=None*, *fids=None*, *debug=False*)
See loadMRSD instead.

`loadMRSD_mat`(*filename*, *remove_df=True*, *build_loops=False*, *x_offsets=None*, *segments=80*, *max_length=None*, *tx=None*, *rx=None*, *fids=None*, *debug=False*)
See loadMRSD instead.

`pulses`
Pulse moment vectors gathered from soundings.

`response`
Complex data cube (pulses * gates) from soundings.

`rx_indices`
Indices of the used receiver of each sounding.

`set1DModel`(*thk=[], res=[1000.0]*)
Modifies loop config in terms of primary field resistivity.

`setEarth`(*earth=None*, *incl=60.0*, *decl=2.0*, *mag=4.8e-05*, *rad=False*)
Defines the Earth in terms of inclination, declination and mag.

Parameters

- **earth** (*comet.snmr.survey.Earth [None]*) – Already initialized earth class will be setted. Or created through the other optional arguments.
- **inclination** (*float [60.]*) – Inclination of the earth magnetic field in rad or degree.
- **declination** (*float [2.]*) – Declination of the earth magnetic field in rad or degree.
- **magnitude** (*float [48000 * 1e-9]*) – Magnitude of the earth magnetic field in Tesla.
- **rad** (*boolean [False]*) – Input inclination and declination in rad?

`setLoopConfig`(*config*, *update_loop_configs=True*)
Loop config in terms of primary field resistivity and frequency.

`setResponse`(*array*)
Set a response array from e.g. an inversion as data set for plotting.

tx_indices
Indices of the used transitter of each sounding.

2.3.3 FID

class comet.snmr.survey.**FID**(*tx=0*, *rx=0*, *pulses=None*)
Single SNMR experiment (sounding) using a simple Free Induction Decay (FID).

Attributes to be setted directly:

amperes
Ampere vector [A].

curie
Curie factor for kernel calculation. Read only. Calculated automatically by setting temperature.

deadtime
Effective deadtime (device + half pulse) [s].

filterGates(*mint=0.0*, *maxt=2.0*)
Dismiss not desired time gates.

Parameters

- **mint** (*float [0.0]*) – Cut all data reqired before mint (in seconds). This is done using the gate midpoints including deadtime.
- **maxt** (*float [2.0]*) – Cut all data reqired after maxt (in seconds). This is done using the gate midpoints including deadtime.
- **Append new .gating to restore old gates**
- **raw_data remain untouched)**

gates
Time gate midpoint vector [s] (including deadtime).

gating(*num_gates=42*, *verbose=False*)
(extracted from MRSMatlab, 2017)

y=exp(x) For some interval x(a:b) the exact mean within exp(x(a:b)) yAverage = exp(mean(log(y(a:b)))) t(yAverage) = mean(t(a:b))

Problem: Logarithm is nice for exact average of exponential function. But signals are noise contaminated. 1. Logarithm of gaussian noise changes noise structure from gaussian to lorenzian. Averaging of lorenzian distributed noise is not zero. 2. Since noise can make signal negative a dc shift is added to make signals positive. This deminishes the accurancy of averaging in logspace. For large constant shift averaging in logspace becomes equivalent to average in linspace. However this is nice for noise structure. So we have a tradeoff. Finally, from some amount of intervals on, e.g. 20 within interval [0 1]/s averaging is sufficiently exact in any case.

MMP 18/10/2011

getRotatedAmplitudes()
Returns Data and Error as real component of the rotated Vecs.

load(*savename*, *df_removed=True*)
Load previously saved FID class instance from savename (.npz) (numpy compressed binary data structure).

Usually imported data are cleansed from frequency offsets (df) before saving. However there is no auto detection for that. In rare cases (if you know what youre doing) data are saved without removing df first. Then df_removed has to be set to False. Otherwise the raw data

`pulses`
Pulse moment vector [As].

`rotateAmplitudes`(*raw_data=False*)
One of the three main ways for NMR forward modelling is to use rotated amplitudes, instead of using the amplitudes of the complex data or the complex data itself. If the phase information of the noise free data is known (synthetic data) or fitted (e.g. monoexponential fit) the rotated Amplitudes (also complex, do not confuse) have the advantage of containing all the information in the real part (together with noise), where the imaginary part contians only noise and can therefore be discarded later.

Can be used on gated or ungated data, however this call alters the raw_data!

Parameters **raw_data** (*boolean [True]*) – Flag to decide if raw data or gated data are rotated. Default is raw data, however if no raw data are

Returns

Return type complex rotated raveled data.

`save`(*savename*)
Saves FID class instance under savename. Expect savename with ending .npz (numpy compressed binary data structure).

`setDataPhase`(*data_phase*)
Sets variable data_phase. Expect single float value for data phase in rad.

`setFrequencyOffset`(*df*)
Sets frequency offset of tx pulse to larmor frequency.

Expect one value per pulse or one single value (used for all pulses). None is treated as zero offset (internal initialization).

`setGatedDataErrorAndGates`(*data*, *error*, *gates*, *rotated=False*, *phases=None*, *midpoints=True*)
Sets the processed and gated data vector along with the gates (time discretization) and error cube.

Parameters

- **data** (*np.ndarray*) – Data vector of shape (number of pulses, number of gates). Expect complex valued vector.
- **error** (*np.ndarray*) – Error vector of the same shape as the data vector.
- **gates** (*np.ndarray*) – Simple time vector in seconds with shape matching the dimension 1 of the data and error vector. Expect gates without deadtime.
- **rotated** (*boolean [False]*) – Define whether the data are already rotated or not. thee is no autodetect for that.
- **phases** (*np.ndarray [None]*) – Define phases as simple vector containing phases in rad. Expect one value per pulse.
- **midpoints** (*boolean [True]*) – If True (default) the given times in the gates vector are interpreted as midpoint of gates. However if False the vector is interpreted as outer limits of the gates, so gate 1 would be defined between time 1 and time 2 and gate 2 between time 2 and 3 and so on.
- **Sets**
- **—-**
- **This functionality fills the following attributes**
- ***data_gated*, *gates*, *error_gated*, *rotated***
- **and optionally**

- ***phi* (phases)**

`setGates` (*gates*, *midpoints=True*)
Define time gates.

Parameters

- **gates** (*np.ndarray*) – Define gates midpoints. Expect array with float in [s]. See midpoints for definition of how the input array is interpreted.
- **midpoints** (*boolean [True]*) – If True (default) the given times in the gates vector are interpreted as midpoint of gates. However if False the vector is interpreted as outer limits of the gates, so gate 1 would be defined between time 1 and time 2 and gate 2 between timne 2 and 3 and so on.
- **Sets**
- **—-** – *gates* and *_gates_thk* if not the midpoints are given

`setPhases` (*phi*)
Sets variable phi. No check for length if vector is done. See setGatedDataErrorAndGates or setRawDataErrorAndTimes for more details.

`setPulseDuration` (*taup*, *deadtime_device=0.005*)
Sets pulse duration [s] and internal deadtime from the device.

Parameters

- **taup** (*float*) – Pulse duration in seconds.
- **deadtime_device** (*float [0.005]*) – Internal deadtime of the measurement device in seconds. 0.005 seconds are default for synthetic studies.
- **Sets**
- **—-**
- ***taup1*,**
- ***deadtime_device*,**
- ***deadtime* (half pulse + deadtime_device)**

`setPulses` (*pulses*)
Set pulse moment vector. Expect array with float in [As].

pulses

`setRawDataErrorAndTimes` (*data*, *error*, *times*, *rotated=False*, *phases=None*, *remove_df=True*, *omit_regating=False*)
Sets the raw (processed but ungated) data vector along with the time discretization and errorvector.

Parameters

- **data** (*np.ndarray*) – Data vector of shape (number of pulses, times). Expect complex valued vector.
- **error** (*np.ndarray*) – Error vector of the same shape as the data vector.
- **times** (*np.ndarray*) – Simple time vector in seconds with shape matching the dimension 1 of the data and error vector, expect times without deadtime!
- **rotated** (*boolean [False]*) – Define whether the data are already rotated or not. There is no autodetect for that.
- **phases** (*np.ndarray [None]*) – Define phases as simple vector containing phases in rad. Expect one value per pulse.

- **remove_df** (*boolean [True]*) – Removes the frequency offset in the given data stored in the attribute **df** [Hz].
- **omit_regating** (*boolean [False]*) – When setting the raw data, the gated data need to be recalculated. By default this is done via regating with the original settings for the gating.
- **Sets**
- **—-**
- **This functionality fills the following attributes**
- ***data_raw*, *times*, *error_raw*, *raw_rotated***
- **and optionally**
- ***phi* (phases)**

`setResponse`(*array*)
Sets a respinse array with the same shape as the data e.g. from an inversion instance. For plotting only.

`setRotated`(*rotated*, *raw_data=False*)
Sets rotation of data. True = rotatedAmplitudes, False = complex.

`setRx`(*index*, *turns=None*)
Define index of receiver loop and turns.

`setTx`(*index*, *turns=None*)
Define index of transmitter loop and turns.

`temperature`
Middle temperature [K]. Default = 281 K (8°C or 46.4°F).

`times`
Time vector [s] of raw data (including deadtime).

2.3.4 Kernel

class `comet.snmr.kernel.`**`Kernel`**(*survey=None*, *fid=0*, *dimension=1*, *name=None*)
Basic class to solve the NMR kernel computation.

Parameters

- **name** (*string [None]*) – If kernel is loaded from file.
- **survey** (*survey class instance [None]*) – Calls *setSurvey* to define underlaying survey class. Holds important attributes like pulse moments and the loops for tx and rx.
- **tx** (*integer [0]*) – Transmitter index in corresponding survey.
- **rx** (*integer [0]*) – Receiver index in corresponding survey.
- **fid** (*interger [0]*) – Sounding index in corresponding survey.
- **dimension** (*integer [1]*) – Defines the kernel integration.

Example

```
>>> from comet.snmr import kernel as k
>>> from comet.snmr import survey
>>> site = survey.Survey()
>>> kernel = k.kernel(site)
```

(continues on next page)

(continued from previous page)

```
>>> kernel.calculate()
>>> kernel.save('savename')
>>> kernel.show()
```

BFieldCalculation(*loop_mesh=None*, *dipole_mesh=None*, *interpolate=False*, *just_loop_fields=False*, *recalc_loop_fields=False*, *recalc_primary=False*, *num_cpu=12*, ***kwargs*)
Calculates the Bfield for the kernel function for tx and rx.

internal call of *loop.calculate()* including decision if cell based or node based Bfield is needed. All optional parameters are piped to the *loop.calculate()* call. Based on the desired dimension of the kernel a specialised mesh may be automatically generated for the calculation.

Part 1/3 of the kernel calculation. Called automatically if *kernel.calculate* is called.

calcMagnetization()
Creates 3D mesh and calcualtes magnetization vector after excitation. Returns magnetization vector of shape (num_pulses, num_cells_3d, 3)

calculate(*loop_mesh=None*, *dipole_mesh=None*, *interpolate=False*, *savename=None*, *forceNew=False*, *slices=True*, *slice_name=None*, ***kwargs*)
All three parts of the kernel calulation are called here.

All given kwargs are directed to BfieldCalculation(), see function info for details about possible keyword arguments.

```
>>> self.BFieldCalculation(**kwargs)
```

```
>>> self.ellipticalDecomposition()
```

```
>>> self.kernelIntegration()
```

```
>>> if savename is not None:
        self.save(savename)
```

Keyword Arguments destinations – none for now, with exception of “num_cpu”, [12] which is directed to BfieldCalculation and/or sliceKernel

create1DKernelMesh(*max_length=0.1*, *area=100.0*, *quality=32*, *zvec=None*, *size_factor=2.5*, *z_factor=2.5*, *export_xyplane=None*, *max_dipoles=2000*, *calc_3D_stats=True*, *xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*)
In order to integrate the kernel to a 1D structure without interpolation errors, a special mesh consisting of triangular zylinders has to be defined.

Parameters

- **max_length** (*float [0.1]*) – Defines the smallest edge length for the discretisation of the loop . In order to get admirable kernel results a value of 0.1 meters should be the maximum.
- **area** (*float [100.]*) – Defines the maximum Area a triangle in the loop slice can have.
- **quality** (*float [32.]*) – Defines the smallest angle inside a triangle. Be careful with values above 35.
- **zvec** (*array_like [None]*) – Usualy the zvec is defined automatically, this flag gives the user the optional possibility to give a zvec from outside the funktion.

- **size_factor** (*float [2.5]*) – Extension of the kernel mesh (and therefore integration volume) in the x and y direction. Should be at least 2 times the loop diameter or shortest edge length. This value defines the multipier.
- **z_factor** (*float [2.5]*) – Maximum depth of the Kernel. Should be at least 2 times the loop diameter or shortest edge length. This value defines the multipier.
- **export_xyplane** (*string [None]*) – Filename for the resulting kernel mesh plane in 2D can be exported for debugging or simply to check the mesh (vtk).
- **max_dipoles** (*interger [2000]*) – Fallback for high node density loops. This sets an overall maximum for the number of dipoles used for the loop discretization. However this only comes into account in rare cases.

create2DKernelMesh(*area=15.0*, *quality=34*, *yvec=None*, *x_factor=5*, *z_factor=2*, *savename=None*, *export_xzplane=None*, *calc_3D_stats=True*, *order=0*)

Similary to the mesh in the 1D case a special mesh consisting of triangluar zylinders is generated. The Zylinders are pointing in the y direction to allow a perfect integration to the x-z plane.

Parameters

- **area** (*float [15.]*) – Affects the maximum area a triangle in the 2D slice is allowed to have. Higher Values lead to bigger cells.
- **quality** (*float [34]*) – Defines the smallest angle inside a triangle. Be careful with values above 34.5. Higher values = more cells.
- **yvec** (*ndarray, list [None]*) – Usualy the y vector is defined automatically, this flag gives the user the optional possibility to give a YVec from outside the function.
- **x_factor** (*float [2]*) – Extension of the kernel mesh (and therefore integration volume) in the x direction. Should be at least 2 times the loop diameter or shortest edge length. This value defines the multipier.
- **z_factor** (*float [2]*) – Extension of the kernel mesh (and therefore integration volume) in the z direction. Should be at least 2 times the loop diameter or shortest edge length. This value defines the multipier.
- **savename** (*string [None]*) – If a savename is given, the resulting 2D Mesh is saved in the .bms format for later use.
- **export_xyplane** (*string [None]*) – Filename for the resulting kernel mesh plane in 2D can be exported for debugging or simply to check the mesh (vtk).

createMagnetizationMesh()

Creates full 3D mesh for display and calcualtion of magnetization vectors. Not needed for normal kernel calculation routine and big, therefore separate.

createSeperatedLoopMesh(*name='SepLoopMesh'*, *dipole=True*, *exportVTK=False*, *refinement_para=1.0*, *max_area_factor=1.0*)

Creates a mesh that contains the receiver and the transmitter loop.

createYVec(*max_length=0.2*, *max_num=300*, *y_factor=2.0*, *calc_3D_stats=True*)

Creates the y vector discretization for the 2D kernel mesh.

The y vector represents the y values of the 3D Kernel mesh before the integration to 2D.

Parameters

- **max_length** (*float [0.2]*) – Maximum distance between two slices inbetween the source dipoles.
- **max_num** (*integer [300]*) – Maximum number of slices. Overrides max_length if they conflict.

- **y_factor** (*float [2.]*) – Extension of the kernel mesh (and therefore integration volume) in the y direction. Should be at least 2 times the loop diameter or shortest edge length. This value defines the multipier.

`createZVector` (*numz*, *minz*, *min_thk=0.5*)
Creates a sinus hyperbolicus shaped Z discretisation in numz steps between 0 and minz.

`ellipticalDecomposition` ()
Computes the counter and corotating parts of the given magnetic fields with respect to a given earth magnetic field.

Parameters

- **Bfield** (*complex field [3, n] or string*) – Optional. Possibility to insert a pre calculated field.
- **Inclination** (*float*) – Inclination of the earth magnetic field at the loop site in rad [0… 2pi]
- **Declination** (*float*) – Declination of the magnetic field at the loop site in rad [0… 2pi]
- **B** (*np.array of shape (3, n)*) – Magnetic field of the loop
- **Second part of the kernel calculation.**
- **- mainly from Weichman et al. (2000)**

`static ellipticalDecomposition_multi` (*Bfield*, *earth*)
Computes the counter and corotating parts of the given magnetic fields with respect to a given earth magnetic field.

Parameters

- **Bfield** (*complex field [3, n] or string*) – Optional. Possibility to insert a pre calculated field.
- **Inclination** (*float*) – Inclination of the earth magnetic field at the loop site in rad [0… 2pi]
- **Declination** (*float*) – Declination of the magnetic field at the loop site in rad [0… 2pi]
- **B** (*np.array of shape (3, n)*) – Magnetic field of the loop
- **Second part of the kernel calculation.**
- **Literature**
- **———-**
- **- Weichman et al. (2000)**
- **- Hertrich (2005, Appendix)**
- **- Hertrich (2008, eq. 6 ff.)**

`export2DKernel` (*fig=None*, *ax=None*, *savename=None*, *png_dpi=300*, *noYLabel=False*, *index=0*, *colorBar=True*, *size=13*, *pdf=None*, *fixed_cbar=False*, ***kwargs*)
Exports 2D Kernel for given pulse moment. Kwargs are redirected to *show*.

`export2DKernel2PDF` (*name*, *fixed_cbar=False*, ***kwargs*)
Export 2D Kernel for all pulse moments as stiched pdf. Kwargs are redirected to *export2DKernel*.

`exportMagnetization` (*name*, *vtk_export=False*, *pulse=0*)
Export a previously calculated magnetization vector as numpy vector and optionally vtk file.

`fid`
Reference to sounding (FID) class instance in survey.

getSliceCoords()
Returns input coordinates for custEM Slice interpolation of magnetic fields to the kernel slices.

interpolateBFieldToKernel(*recalc_prim_on_kernel=False*, *recalc_primary=False*, *num_cpu=32*, *calc_3D_stats=True*)
Takes the rx Bfield and interpolates it to the kernel mesh.

static kernelCalculation_multi(*fid*, *earth*, *txalpha*, *txbeta*, *txzeta*, *txperpend*, *rxalpha=None*, *rxbeta=None*, *rxzeta=None*, *rxperpend=None*, *calc_theta=False*)

kernelIntegration(*calc_theta=False*)
Computes the integration of the kernel with respect to the desired dimension.

Parameters

- **decomposition** (*(alpha, beta, zeta)*) – Bfield_part essentially consists of the output from the elliptical decomposition of the magnetic field.
- **measurement** (*class*) – An instance of a measurement class has to be given in order to keep the number of input arguments manageable.
- **earthmagnitude** (*float*) – Magnitude of the earth magnetic field [Tesla]. Aproximatly about 30000 to 65000 nT (1 nT = 1e-9 Tesla).
- **Third part of the kernel calculation.**

larmor
Larmor frequency [Hz] from earth defined in survey.

load(*savename*, *load_loopmesh=True*, *kernelmesh2d=None*, *load_kernelmesh=True*, *use_order_refinement=True*)
Load a previously saved kernel (.npz-format).

pulses
Reference to pulse moments from sounding (FID).

release_memory()
Calling this function is releasing some attributes that are using a fairly big amount of memory.

Sets the following attributes back to None:

- The interpolation matrix between the loop meshes and the kernel mesh

interpolationMatrix

- local copies of the magnetic fields (fields in tx and rx are not

effected) *txBfield*, *rxBfield*

- the 3D kernel mesh cell center and volumes

kernelMeshCellVolume, *kernelMeshCellCenter*

- the elliptical decomposition of the tx and rx bfields

txalpha, *txbeta*, *txzeta*, *txperpend*, *rxalpha*, *rxbeta*, *rxzeta*, *rxperpend*

Note: a recalculation of the kernel will take about the same amount of time as the first call, as all cached variables are gone, however apart from a recalculation, the other purposes of the kernel class (export, figures, inversion(without recalculation)) are not effected.

Another note: If you want to use this method only for saving disk space in case you save the kernel class, then you might consider the *light* flag of the *.save* method instead.

rx
Reference to receiver class instance in survey.

rx_area
Area of the receiver loop.

save (*savename=None*, *save_interpolation_mat=False*, *save_loopmesh=False*, *light=True*, *kernelmesh_name=None*)
Save the basic information to restore the Kernel class later.

set1DKernelMesh (*mesh*, *calc_3D_stats=True*)
Sets the 1D kernel mesh.

Parameters

- **mesh** (*stirng or pygimli.Mesh*) – Filename or mesh instance of a 2D mesh in the x-y plane.
- **Need**
- **—-**
- **z discretization** – Can be setted via *createZVector*, *setZVector* or direct use of *create1DKernelMesh*. However the needed information to do that may not be available on the fly, therefore no default z vector is created.

set2DKernelMesh (*inmesh*, *yvec=None*, *order=0*, *integration_mat=None*, *calc_3D_stats=True*)
kwargs to createYVec if YVec is None

setModel (**args*, ***kwargs*)
Pipes args and kwargs to *self.tx.setModel*. Same for rx.

setPulsesDirectly (*pulses*)
Set pulse moment vector manually if not supported by survey + fid. (This is called when loading a kernel from the harddisk, mainly for plotting reasons). For all calculation purposes a survey and fid class is recommended.

setRx (*rx*, ***kwargs*)
Sets initialized loop or pipe arg and kwargs to *loadLoop*.

setSurvey (*survey*, *fid=0*)
Sets survey class containing necessary information for the kernel.

Parameters

- **survey** (*comet.snmr.survey.Survey or None*) – Sets given survey class instance or create empty class instance.
- **fid** (*integer [0]*) – Index of corresponding sounding in the survey.

setTx (*tx*, ***kwargs*)
Sets initialized loop or pipe arg and kwargs to *loadLoop*.

setZVector (*vector*, *min_thk=0.5*)
Defines the attribute zvec.

Sets the given vector as z discretization. Attention: the value for min_thk defines the minimum thickness of the discretization used in the end. For all thicknesses in vector smaller than min_thk, the Kernel is integrated to match the min_thk. For calulation of the kernel function the original given vector is used.

Parameters

- **vector** (*array_like*) – Z discretization in m to be used for the kernel calculation. If a new vector is to be created, please also take a look at the method *createZVector*.

- **min_thk** (*float*) – Minimum thickness te kernel and zvec is integrated if returned. This leads to higher accuracy in the vicinity of the loop.

show(*toplot=['real', 'imag', 'amp', 'phase', '0D'], indices=None, savename=None, normed=False, suptitle=None, ax=None, pulse_in_log=False, kernel_absolute_values=False, cbar_percentage=0.99, fixed_cbar=False, lut=33, show_marked_edges=False, **kwargs*)

Visualise the Kernel with respect to the desired dimension.

Automatically defined within the kernel class via the parameter kernel.dimension = [0. . . 3]. Plotting of a kernel in the desired dimension is only possible if the kernel is also calculated with respect to that dimension. It's not possible to calculate the kernel with kernel.dimension = 1 and then plot the kernel with kernel.dimension = 2.

0D : Simple Graph plotting kernel-values over pulsemoments

1D : Graph with 1D integrated kernels over the depth of the model

2D : Slice of the x-z-plane with triangle mesh containing the 2D

3D : Export of the kernel in vtk format for visualising.

none so far

Plots the 1D integrated Kernel with a given z discretisation over the measured pulse sequences.

toplot: list [['real', 'imag', 'amp', 'phase', '1D']] There are different possibilities to plot the kernel. This parameter defines which part of the kernel is shown. Possible options are: 'real', 'imag', 'amp', 'phase', '0D' (integrated over z). All strings in the toplot variable will be plotted in the same order given in the list.

cMap: string ['viridis'] Defines the colormap used to display the kernel. In order to get a good contrast between the max and min as well as being useful in comparison with MRSMatlab, 'viridis' is the default colormap. Any colormap reachable by the plt.get_cmap(. . .) method can be chosen.

normed: bool [True] A on the dimension based normalisation of the plot permits a better assessment of the kernel distribution.

ax: plotting ax or list of axes [None] Plot on a predefined ax and gives back the ax. A onedimensionla list of axes is also accepted, if the number of items in 'toplot' is the same as the available axes.

lut: None or int [None] Number of colors for the colorbar. If lut is not None it must be an integer giving the number of entries desired in the lookup table, and name must be a standard mpl colormap name.

indices: list By default one 2D plot is created for each pulsemoment. In order to limit the number of plots the optional paramter indices can be given as a list of indices referring to the pulse moments to be shown.

cMap: string ['viridis'] See Parameter 1D.

normed: bool [True] A on the dimension based normalisation of the plot permits a better assessment of the kernel distribution.

show_marked_edges: boolean [False] Whether or not marked edges gets drawn.

possible kwargs for matplotlib: cMin, cMax for range of the colorbar. All other kwargs are reaching matplotlib functions.

default label 2D: 'integrated kernel (2D) [nV/m^2] pulsemoment: {:.3f} As' .format(self.pulses[i])

A self-sufficient plot of the kernel without any integration would result in a set of 3D Cubes and is not implemented for now.

Instead the kernel will be saved in vtk format which can be easily handled.

savename: string A String defining the relative path to the vtk-file the kernel will be saved in. If not given the default savename will be flagged with the string '_default_' and contain some information about the kernel.

Example

2D:

```
>>> ax, cbar = kernel.show(indices=[16], cMin=-1,
>>>                        cMax=2, size=20, pad=0.7)
>>> ax.set_ylim(-50, 0)
```

sliceKernel2D(*savename=None*, *forceNew=False*, *loopSaveName=None*, *num_cpu=None*, *new_bfield=False*, *loop_mesh=None*, *slice_name=None*, ***kwargs*)
2D Kernel in a memory saving parallel computation approach.

tx
Reference to transmitter class instance in survey.

tx_area
Area of the transmitter loop.

zvec
z discretisation

2.4 comet

2.4.1 comet package

Subpackages

comet.pyhed package

Subpackages

comet.pyhed.IO package

Submodules

comet.pyhed.IO.saveload module

Part of comet/pyhed/IO

exception `comet.pyhed.IO.saveload.`**ArgsError**(*value*)
Bases: `Exception`

exception `comet.pyhed.IO.saveload.`**TetgenNotFoundError**
Bases: `Exception`

Special Exception to catch in a try except.

comet.pyhed.IO.saveload.**addVolumeConstraintToPoly**(*name*, *regions*, *float_format='6.3f'*)

Append region information in form of volume constraints to a tetgen.poly file. The given regions has to be of shape (n, 5 or 6), with n times: [number, x, y, z, regional attribute, volume constraint]

comet.pyhed.IO.saveload.**checkDirectory**(*savename*, *filename=False*, *verbose=False*)

Checks for directory and creates if not.

comet.pyhed.IO.saveload.**checkForFile**(*name*)

Checks if file exists and creates a directory if it does not.

comet.pyhed.IO.saveload.**createCustEMDirectories**(*m_dir='.'*, *r_dir='.'*)

Creates the used custEM directories based on *m_dir* and *r_dir*.

comet.pyhed.IO.saveload.**cutExtension**(*path*)

comet.pyhed.IO.saveload.**delLastLine**(*opened_file*, *line_ending='\n'*)

Efficient way of deleting the last line of a large file.

comet.pyhed.IO.saveload.**getItem**(*archive*, *key*, *default=None*)

Get item for *key* from numpy *archive* via try except with given *default* value for None.

comet.pyhed.IO.saveload.**searchforTetgen**(*returnPathfile=False*)

Try to find a valid tetgen installation for meshing purposes.

path: string Path to tetgen installation or path to pathfile of pyhed itself.

comet.pyhed.IO.vtk module

Part of comet/pyhed/IO

comet.pyhed.IO.vtk.**add_vector_to_vtk**(*vtk*, *vector*, *vectorname*, *dtype_string='double'*)

Appends a vector fields to an existing vtk file.

Parameters

- **vtk** (*string*) – Path to vtk file, where the field is to be appended.
- **vector** (*np.ndarray*) – Real valued array of shape (3, n) n being either the number of nodes or the number of cells.
- **vectorname** (*string*) – Name under which the array is to be identified in the vtk file.
- **dtype_string** (*string*) – Format string in the vtk file. Default 'double' is used for float values.

comet.pyhed.IO.vtk.**fieldCell2Node**(*mesh*, *field*)

comet.pyhed.IO.vtk.**savefieldvtk**(*vtk_name*, *mesh*, *field*, *itype='mesh'*, *components=False*, *scalar=False*, *save=['real', 'imag']*, *field_name='field'*, *verbose=True*)

Basic VTK export routine when it comes to complex vector fields on unstructured meshes.

Parameters

- **vtk_name** (*string*) – Path to the resulting vtk file.
- **mesh** (*string or pg.Mesh or np.ndarray*) – Pygimli Mesh instance or path to a mesh file. Alternatively a bare numpy array containing coordinates or meshgrid ranges can be used.
- **field** (*np.ndarray*) – Complex vector field of shape (3, n) with *n* corresponding either to mesh cell count or node count.

- **itype** (*string ['mesh']*) – Defines input type of *mesh*. Possible choices are 'mesh' for pg.Mesh (instance or file path), 'coords' for direct 3d coordinates ranges to build a regular meshgrid, or 'grid' if input is a 3d meshgrid.
- **components** (*boolean [False]*) – Separately saves the spatial components of the vector field for debugging purposes.
- **scalar** (*boolean [False]*) – Input is a simple scalar field (e.g. potential).
- **save** (*list ['real', 'imag']*) – The vector is saved once for each entry in the list. Possible choices are 'real' to save the real component, 'imag' to save the imaginary component, 'aps' or 'amp' to save the amplitude, and 'phase' to save the phase component of the field. Only works with vector fields.
- **field_name** (*string*) – Name under which the array is to be identified in the vtk file.
- **verbose** (*boolean [True]*) – Turn on verbose mode.

Returns

Return type True if succesful.

Module contents

Module comet/pyhed/IO

Init file for IO subpackage of pyhed. Mainly used to load and save a bunch of stuff or handling some checks.

comet.pyhed.hed package

Subpackages

comet.pyhed.hed.reference package

Submodules

comet.pyhed.hed.reference.dipole1d module

comet.pyhed.hed.reference.homogeneous_fullspace module

Part of comet/pyhed/hed/reference

`comet.pyhed.hed.reference.homogeneous_fullspace.`**`hedx_electric`**(*model*, *f*, *sigma*, *I*, *ds*, *drop_tol=1e-06*)

Analytic calculation of the electric field for an electric dipole in x direction. Formula given in Ward and Hohmann (1988), page 173 number 2.40. Model in cartesian coordinates, as well as the output. Sigma != 0 drop_tol to avoid singularities [1e-6] No grounding!

`comet.pyhed.hed.reference.homogeneous_fullspace.`**`hedx_magnetic`**(*model*, *f*, *sigma*, *I*, *ds*, *drop_tol=1e-06*)

Analytic calculation of the magnatic field for an electric dipole in x direction. Formula given in Ward and Hohmann (1988), page 174 number 2.41. Model in cartesian coordinates, as well as the output. Sigma != 0 drop_tol to avoid singularities [1e-6] No grounding!

comet.pyhed.hed.reference.homogeneous_halfspace module

Part of comet/pyhed/hed/reference

`comet.pyhed.hed.reference.homogeneous_halfspace.`**`hed_field`**(*r*, *f*, *sigma*, *phi*, *I*, *ds*, *BorH='B'*)

Semi analytic solution for electrical and magnetical fields at the surface of a homogeneous halfspace of conductivity sigma.

`comet.pyhed.hed.reference.homogeneous_halfspace.`**`hed_field_hohmann`**(*model*, *f*, *sigma*, *I*, *ds*, *ftype='H'*, ***kwargs*)

semi analytic solution for a magnetic field at the surface of a homogeneous halfspace of conductivity sigma ward hohmann formula: page 235-236 No 4.166, 4.171 and 4.173

edit: E-term: grounding term only (4.159)

Module contents

Submodules

comet.pyhed.hed.hed_bib module

Part of comet/pyhed/hed

`comet.pyhed.hed.hed_bib.`**`btp`**(*u*, *model*, *rho*, *d*, *f*, *mode*)

Airspace only, internal function, for imput see **downout**.

Do not call directly.

`comet.pyhed.hed.hed_bib.`**`calcField`**(*polar*, *rho*, *d*, *f*, *Ids*, *ftype*, *mode*)

Calculates field for a given dipole on given polar coords. Internally.

Internally used by *makeField*. Please be referred to the docstrings of **makeField**. And please use makeField directly!!!

`comet.pyhed.hed.hed_bib.`**`downout`**(*u*, *model*, *rho*, *d*, *f*, *mode*)

Overall call function for recursive calculation.

Parameters

- **u** (*np.ndarray*) – Horizontal wavenumbers based on Hankel factors and horizontal tx-rx distance. Shape: (Hankel, n_rx)
- **model** (*np.ndarray*) – Polar coords of the receiver pos (3, n).
- **rho** (*np.ndarray*) – Resistivities for each layer (Ohm*m).
- **d** (*np.ndarray*) – Thicknesses of each layer (m).
- **f** (*float*) – Frequency (Hz).
- **mode** (*str*) – Calculation for 'te', 'tm' or 'tetm' possible. Mode 'te' for closed loops and 'tetm' for grounded wires. Single 'tm' is only for debug.

Returns

- **aa** (*np.ndarray*) – Ratio of the partial wave amplitude A(z,u)/A(0,u)
- **aap** (*np.andarray*) – Ratio of the partial wave amplitude A'(z,u)/A'(0,u)

- **bt** (*np.ndarray*) – Admittance at the surface of the layerd halfspace

`comet.pyhed.hed.hed_bib.`**`downward`**(*u*, *model*, *rho*, *d*, *f*, *mode*)

Downward attenuation.

Parameters

- **u** (*np.ndarray*) – Horizontal wavenumbers based on Hankel factors and horizontal tx-rx distance. Shape: (Hankel, n_rx)
- **model** (*np.ndarray*) – Polar coords of the receiver pos (3, n).
- **rho** (*np.ndarray*) – Resistivities for each layer (Ohm*m).
- **d** (*np.ndarray*) – Thicknesses of each layer (m).
- **f** (*float*) – Frequency (Hz).

Returns

- **aa** (*np.ndarray*) – Ratio of the partial wave amplitude A(z,u)/A(0,u)
- **aap** (*np.andarray*) – Ratio of the partial wave amplitude A'(z,u)/A'(0,u)
- **bt** (*np.ndarray*) – Admittance at the surface of the layerd halfspace

`comet.pyhed.hed.hed_bib.`**`efield_3D_hed_te`**(*polar*, *u*, *aa*, *aap*, *bt*, *f*, *Ids*)

Calculation of electric field for transversal electric mode.

Computes the transversal electric induced electric field of a x-directed dipole at (0, 0, 0). Field shape (3, n) with x, y, z components for each reciever point in *polar*.

Internal function. Called by **makeField** if *ftype* == 'E' and *mode* in ('te', 'tetm').

Parameters

- **polar** (*np.ndarray*) – Polar coords of the receiver pos (3, n).
- **u** (*np.ndarray*) – Horizontal wavenumbers based on Hankel factors and horizontal tx-rx distance. Shape: (Hankel, n_rx)
- **aa** (*np.ndarray*) – Ratio of the partial wave amplitude A(z,u)/A(0,u)
- **aap** (*np.andarray*) – Ratio of the partial wave amplitude A'(z,u)/A'(0,u)
- **bt** (*np.ndarray*) – Admittance at the surface of the layerd halfspace
- **f** (*float*) – Frequency (Hz).
- **Ids** (*float*) – Dipole current * dipole length.

Returns **field** – Transversal electric component of the electric field of a x-directed dipole at (0, 0, 0). field.shape = polar.shape.

Return type np.ndarray

`comet.pyhed.hed.hed_bib.`**`hankelfc`**(*order*)

Filter coefficients for hankel transformation by Anderson (1980)

`comet.pyhed.hed.hed_bib.`**`hfield_3D_hed_te`**(*polar*, *u*, *aa*, *aap*, *bt*, *f*, *Ids*)

Calculation of magnetic field for transversal electric mode.

Computes the transversal electric induced magnetic field of a x-directed dipole at (0, 0, 0). Field shape (3, n) with x, y, z components for each reciever point in *polar*.

Internal function. Called by **makeField** if *ftype* == 'H' and *mode* in ('te', 'tetm').

Parameters

- **polar** (*np.ndarray*) – Polar coords of the receiver pos (3, n).
- **u** (*np.ndarray*) – Horizontal wavenumbers based on Hankel factors and horizontal tx-rx distance. Shape: (Hankel, n_rx)
- **aa** (*np.ndarray*) – Ratio of the partial wave amplitude A(z,u)/A(0,u)
- **aap** (*np.andarray*) – Ratio of the partial wave amplitude A'(z,u)/A'(0,u)
- **bt** (*np.ndarray*) – Admittance at the surface of the layerd halfspace
- **f** (*float*) – Frequency (Hz).
- **Ids** (*float*) – Dipole current * dipole length.

Returns field – Transversal electric induced magnetic field of a x-directed dipole at (0, 0, 0). field.shape = polar.shape.

Return type np.ndarray

`comet.pyhed.hed.hed_bib.`**`hfield_3D_hed_tm`**`(`*polar*, *u*, *aa*, *aap*, *bt*, *f*, *Ids*`)`

Calculation of magnetic field for transversal magnetic mode.

Computes the transversal magnetic component of the magnetic field of a x-directed dipole at (0, 0, 0). Field shape (3, n) with x, y, z components for each reciever point in *polar*.

Internal function. Called by **makeField** if *ftype* == 'H' and *mode* in ('tm', 'tetm').

Parameters

- **polar** (*np.ndarray*) – Polar coords of the receiver pos (3, n).
- **u** (*np.ndarray*) – Horizontal wavenumbers based on Hankel factors and horizontal tx-rx distance. Shape: (Hankel, n_rx)
- **aa** (*np.ndarray*) – Ratio of the partial wave amplitude A(z,u)/A(0,u)
- **aap** (*np.andarray*) – Ratio of the partial wave amplitude A'(z,u)/A'(0,u)
- **bt** (*np.ndarray*) – Admittance at the surface of the layerd halfspace
- **f** (*float*) – Frequency (Hz).
- **Ids** (*float*) – Dipole current * dipole length.

Returns field – Transversal magnetic component of the magnetic field of a x-directed dipole at (0, 0, 0). field.shape = polar.shape.

Return type np.ndarray

`comet.pyhed.hed.hed_bib.`**`makeField`**`(`*coords*, *rho_in*, *d_in*, *f=2000*, *Ids=1*, *pos=(0, 0, 0)*, *angle=0*, *mode='te'*, *inputType='M'*, *ftype='B'*, *cell_center=False*, *drop_tol=0.01*, *src_z=-0.01*`)`

Calculation of the electric or magnetic field of a horizontal electric dipole at position pos, pointing in a direction defined by angle on given cartesian coordinates.

Parameters

- **coords** (*np.ndarray or string*) – Reciever coords. Possible input types are numpy ndarrays for direct cartesian coordinates, ranges for (x, y, z) or pygimli Meshes.
- **rho_in** (*float or np.ndarray*) – Float or Array of resistivity values for the 1d layered earth model. Airspace is at the level of the source dipole.
- **d_in** (*float or np.ndarray*) – Layer thicknesses in m. As the lower halfspace is considered to have an infinite thickness, *d_in* is always one value short of *rho_in* (an empty list ar array or a 0 for homogenous halfspace.)

- **f** (*float [2000]*) – Frequency (Hz).
- **Ids** (*float [1]*) – Dipole current * dipole length. Used for simple scaling of the calcualted field.
- **pos** (*tuple of length 3 [(0 ,0, 0)]*) – Absolute position of the source dipole in cartesian coordinates. Values for z are used for a shift of the airspace. Currently only sources at the upper halfpsce boundary are permitted.
- **angle** (*float [0]*) – Rotation of the dipole with respect of an x-directed dipole counting positive clockwise.
- **mode** (*string ['te']*) – For a closed loop consisting of a finite number of dipoles the total field can be seen as superposition of the transversal electric components of the single dipole fields ('te'). For grounded dipoles 'tetm' is needed.
- **inputType** (*string ['M']*) – Specifier for input coordinates. Possible choices are 'M' if *coords* is a pygimli mesh or file path to a pygimli mesh, 'C' if coords is a np.ndarray with ranges to build a meshgrid, or 'V' to indicate that *coords* is a vector of cartesian coordinates.
- **ftype** (*string ['B']*) – Flag to control calculated field type. Possible choices are 'E', 'B' or 'H' (asuming B = 4e-7 * pi * H).
- **verbose** (*boolean [False]*) – Turn on verbose mode.
- **cell_center** (*boolean [False]*) – If coords is a pygimli mesh, there is the additional possibility to calulate the fields in the cell Centers, instead of the node positions.
- **drop_tol** (*float [1e-2]*) – Singularity fix. All horizontal distances between *drop_tol* and the transmitter dipole are placed between the first reciever outside the tolerance and the tolerance, maintaining the correct order and angle. This has been very useful for later usage of the fields in FEM approaches.
- **src_z** (*float [-0.001]*) – This is only used if grounded terms for an electric field are used. In this case the source has to be buried in order to get the correct results. Default is 1 cm (remember: z defined positive upwards). So in most cases this value should be negative.

comet.pyhed.hed.hed_para module

Part of comet/pyhed/hed

`comet.pyhed.hed.hed_para.`**`InterpolationWorker`**(*num*, *pos_queue*, *out_queue*, *data*, *srcmeshName*, *outmeshName*, *outtype*, *verbose*)

MPI Worker used to interpolate fields to target source location.

`comet.pyhed.hed.hed_para.`**`SummationWorker`**(*queueIn*, *queueSum*, *queueEnd*, *verbose*)

MPI Worker used to sum up single fields.

`comet.pyhed.hed.hed_para.`**`multiInterpolation`**(*DipoleDataName*, *SrcMeshName*, *OutMesh*, *DipolePos=None*, *verbose=False*)

Call function for multiprocessing interpoaltion of dipole fields.

comet.pyhed.hed.libHED module

Part of comet/pyhed/hed

Earth class for calculation of dipole (HED) fields for 1d layered earth.

The algorithms in method *calcFieldForLayer* of HED class is partly taken from Kerry Key Dipole1D.f90 after the algorithms published in [Key2009G] (Appendix A).

Hankel factors of Hankelfc are based on the original values of Anderson (1990).

References:

class `comet.pyhed.hed.libHED.`**`HED`**(*src_z=-0.01*, *src_theta=0.0*, *src_ids=1.0*, *config=None*, *timer=None*, *debug=False*)

Bases: `object`

`calcFieldForLayer`(*rx_lay*, *lay_indices*, *lam*, *lam_2*, *lam_c*, *lam_c2*, *R_p*, *R_m*, *S_p*, *S_m*, *exp*)

`calculate`()

Calculates the 1d layered earth recursive formula.

Calculates the recursive attenuation and reflection coefficients for each layer on basis of the given set of cylindrical coordinates.

Fills the variables **R_p**, **R_m**, **r_p**, **r_s**, **S_p**, **S_m**, **s_p**, **s_m**, **hem_a**, **hem_b**, **hem_c**, and **hem_d**. The used formulas correspond to equations A-6 to A-13 in [Key2009G] (Appendix A).

`reflectionCoefficients`(*rx_lay*, *lam*, *lam_2*, *lam_c*, *lam_c2*, *exp*)

Calculation of the general reflection coefficients R+, R-, S+, and S- as stated in [Key2009G] (Appendix A, equations A-06 to A-09).

Computed from the air and halfspace, respectively, inward to the source layer.

`setCoords`(*cartesian*, *nodes=True*, *drop_tol=0.01*)

Sets coordinates of the receiver for calculation.

All calcualtions will be performed in cylindrically coordinates.

Parameters

- **cartesian** (*np.ndarray*) – Cartesian coordinates (N points) of the receiver points with shape (3, N). Z is defined positive upwards.
- **drop_tol** (*float*) – Tolerance in meter, where the horizontal src distance is capped to ensure a safe division (singularity fix). Distances smaller than drop_tol are distributed between droptol and 20% of the first value outside the droptol. Raises Exception if all points within drop_tol. Default value of 1cm.

`setTheta`(*theta*)

class `comet.pyhed.hed.libHED.`**`World1D`**(*rho=1000.0*, *thk=None*, *airspace_interface=0.0*, *f=2000.0*)

Bases: `object`

`evalSrcIdx`(*src_depth*)

Evaluates in which layer the source is considered.

`setFrequency`(*freq*)

Simple setter for frequency + implicit omega/sigma calculation

`setRes`(*rho=1000.0*, *thk=None*, *air_resistivty=10000000000000.0*)

Sets the resisitvity model for the dipoles + calc sigma complex.

Parameters

- **rho** (*float or array_like*) – Resistivity distribution in Ohm*m. Airspace is considered to have 0 Ohm*m. The first entry of rho correspond to the first layer of the subsurface. The airspace interface is considered to be at z = 0 m which simplyfies the calculations. For offsets in z, a cordinate transformation has to be performed externally.
- **thk** (*float or array_like*) – Layer thicknesses of each subsurface layer except the last.

class comet.pyhed.hed.libHED.**hankelfc**

Bases: object

getFactors(*string*)

Returns the requested set of Hankel factors.

Parameters **string** (*[str]*) – Evaluates which Hankel factors the wavenumbers is calculated. Possible choices are **sin**, **cos**, **j0**, or **j1**.

Returns **factors** – Hankel factors.

Return type [np.ndarray]

getWavenumbes(*string*)

Calcualtes the wavenumbers for the requested set of Hankel factors.

Parameters **string** (*[str]*) – Evaluates for which Hankel factors the wavenumbers is calculated. Possible choices are **sin**, **cos**, **j0**, or **j1**.

Returns **wavenumbers** – Normed wavenumber factors for evaluation of the Hankel integral. Divide by horizontal distance of the receiver to get horizontal wavenumber lambda = sqrt(k_x^2 + k_y^2).

Return type [np.ndarray]

class comet.pyhed.hed.libHED.**wer_201_2018**

Bases: object

Hankel factors after Werthmüller 2018 implemented from the empymod package after consultation with Dieter Werthmüller. Thank you very much!

The filter coefficient are published in:

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

under the Apache 2.0 license.

Module contents

module comet/pyhed/hed

Init file for HED calculation routines inside pyhed.

comet.pyhed.loop package

Submodules

comet.pyhed.loop.loop_bib module

Part of comet/pyhed/loop

This script contains the main class for the sources as well as several scripts for the initialization of loop classes (build…).

class comet.pyhed.loop.loop_bib.**Geometry**

Bases: object

class comet.pyhed.loop.loop_bib.**Loop**(*Input*, *config=None*, *verbose=False*)

Bases: object

Class for the computation of arbitrary shaped polygon loops. Some functions automatically return this loopclass as result. It is recommended to use these (you may take a look at the example)

Parameters

- **Input** (*string or raw loop class*) – Filename of a prior saved loopfile (recommended). Alternatively the output of the function **computeLoopPositions** (not recommended). For the latter case plenty of convenience functions are found in the the *loop* submodule of *pyhed* starting with "build"....
- **config** (*string or pyhed.config*) – Defines the configuration file for the loop.

Example

```
>>> # example: import
>>> loopclass = Loop('path/to/loopfile')
>>> # example: create circular loop
>>> loopclass = buildCircle(10, 12)  # 10 m radius, 12 dipoles
```

calcAndExportFieldsForFenics(*export_vtk=False*, *num_cpu=32*, ***kwargs*)

Calculates and export primary fields for fenics secondary field calculation.

Parameters **kwargs** (*dict*) – Keyword parameters are redirected to *calculate*.

calculate(*num_cpu=12*, *loop_mesh=None*, *dipole_mesh=None*, *interpolate=False*, *savename=None*, *cell_center=False*, *verbose=False*, *mode='auto'*, *matrix=False*, *field_matrix=None*, *max_node_count=None*, ***kwargs*)

Computation of the loop field with respect to the config.

Parameters

- **num_cpu** (*integer [12]*) – Maximum number of processes allowed for this task.
- **loop_mesh** (*string or mesh instance [None]*) – Optional. Possibility to give a user defined mesh for the calculation.
- **dipole_mesh** (*string or mesh instance [None]*) – Optional. Possibility to give a user defined mesh for the calculation (interpolate=True or matrix=True only).
- **interpolate** (*boolean [True]*) – The loop dipoles can either be calculated directly (False) or once on a seperated mesh (dipolemesh) and then interpolated to the loopmesh (True). If a dipoleFieldName is given, this field will be used for the interpolation.
- **savename** (*string [None]*) – Optional. If savename is not None, the loop will be saved under the name defined in savename.
- **cell_center** (*boolean [True]*) – A default the field of the loop will be calculated at the cell center of the mesh cells. This flag allows for calculation at the mesh nodes. Affects only the definition of the final loopmesh, the dipolemesh will always be calculated at the nodes for interpolation reasons.
- **verbose** (*boolean [False]*) – Turn on verbose mode.
- **mode** (*string ['auto']*) – Five posibilities: 'auto', 'config', 'te', 'tm', 'tetm'

  'auto': Automatic detection wether the loop is grounded or not. Grounded wires are calculated with te and tm mode (see HED). Non grounded wires are calculated with te mode only (sufficient).

'config': the default config decides the mode the field is calculated in.

'te', 'tm', 'tetm': Calculates the field in the choosen mode.

- **matrix** (*boolean [False]*) – Alternatively calculation approach. At first the field on a highly dense dipole mesh will be triggered. After that the field will be interpolated to the single dipole positions by the means of a matrix vector multiplication with a matrix containing appropriate weighting factors. This Approach takes longer than direct calculation in the first run, but the calculated matrix can be used for further calculations with different frequencies or resistivity models (as long as the loopmesh and dipolemesh remain the same).
- **field_matrix** (*list or string [None]*) – Interpolation matrices or file path if calculation with matrix=True. Will be calculated automatically if None.
- **max_node_count** (*integer [None]*) – As all points will be calculated at once, the computational effort scales lineary with the reciever count, the transmitter count and the used hankel factors. If the limits of the available memory is reached *max_node_count* can be used to define the maximum chunk of nodes to be computated at once. Other nodes will be computed afterwards.

Keyword Arguments

- **`arguments are redirected to loop.save and to define`** (*`Keyword`*) –
- **`drop_tol (float [ 1e-2 ]) in the cylindrical coordinate`** (*`the`*) –
- **`to avoid instabilities around the source.`** (*`transformation`*) –

`calculateDipoleField`(*verbose=False*, *drop_tol=0.01*, *num_cpu=12*, *max_node_count=None*)

Calculates field on dipole mesh.

Parameters

- **verbose** (*boolean [False]*) – Turn on verbose mode.
- **drop_tol** (*float [1e-2]*) – Singularity fix. All horizontal distances between *drop_tol* and the transmitter dipole are placed between the first reciever outside the tolerance and the tolerance, maintaining the correct order and angle.
- **num_cpu** (*integer [12]*) – Maximum number of processes allowed for this task.
- **max_node_count** (*integer [None]*) – As all points will be calculated at once, the computational effort scales lineary with the reciever count, the transmitter count and the used hankel factors. If the limits of the available memory is reached *max_node_count* can be used to define the maximum chunk of nodes to be computated at once. Other nodes will be computed afterwards.

`calculateFieldFromMatrix`()

Calculates the primary field on basis of the interpolation matrix and the dipole field.

`calculateFieldMatrix`(*num_cpu=8*, *verbose=False*)

If wished the calcualtion of the total loop field can be done by interpolation and superposition of one highly accurate dipole field to the different transmitter positions of the loop. This is done either done directly or via a vector matrix multiplication.

This function is called to initialize and append the weights to the interpolation matrix from the *dipolemesh* to the *loopmesh* for all tx positions with respect to *pos*, *phi*, and *ds*.

This function will be called if *calculate* is called with *matrix=True*.

Parameters

- **num_cpu** (*integer [8]*) – Define the maximum number of cores allowed for this operation.
- **verbose** (*boolean [False]*) – Turn on verbose mode.

calculateInterpolationMatrix(*Pos*)

Calculates the interpolation matrix.

If one wished the field can be interpolated to another mesh. The interpolation matrix from the loopmesh to an arbitrary set of coordinates is calculated with this function. This function is called to initialize and append the weights to the interpolation matrix.

Note: The loop class does not hold a reference of the resulting matrix, instead gives it back to the caller.

Parameters **Pos** (*np.ndarray or pg.core.PosVector*) – Transmitter positions of shape (n, 3) with n positions. Values are expected to be floats (the conversion to a pg.PosVector will not check again).

Returns **mat** – Sparse interpolation matrix with number of columns equal to the number of nodes in the loopmesh and number of rows equal to the number of input positions.

Return type pg.core.SparseMapMatrix

calculateSecField(*num_cpu=8*, ***kwargs*)

Calculates the secondary field using custEM.

Calculates primary field as well if not found.

Needs a FEM suited mesh as well as a parameter distribution provided by other functions of this class (See *createFEMMesh* and *prepareSecondaryFieldCalculation*).

Parameters

- **num_cpu** (*integer [8]*) – Maximum number of processes allowed for this task. The actual calculation will be done in an mpirun environment with the selected number of cores.
- **kwargs** (*dict*) – Keyword arguments are redirected to *local_apps*.

createDefaultSecondaryConfig(*base=None*, *prefix=''*, *suffix=''*, *m_dir='.'*, *r_dir='.'*)

Short cut to generate a secondary config with some default params.

Parameters

- **prefix** (*string*) – String to be added to the *getDefaultLoopMeshBaseName* string to define the automatic generated names for the default secondary config.
- **suffix** (*string*) – String to be added to the *getDefaultLoopMeshBaseName* string to define the automatic generated names for the default secondary config.

createDipoleMesh(*quadratic=True*, *savename='_default_dipole_mesh.bms'*, *save=False*, *verbose=False*)

Creates a suitable dipole mesh for calculation via a single dipole.

Parameters

- **quadratic** (*boolean [True]*) – If chosen, uses a quadratic (2nd order) mesh for dipole calculation.
- **savename** (*string ['_default_dipole_mesh.bms']*) – Define output name.
- **save** (*boolean [True]*) – Additional save of dipole mesh under *savename*.
- **verbose** (*boolean [False]*) – Turn on verbose mode.

`createFEMMesh` (*para_mesh_2d=None, savename=None, exportVTK=False, exportH5=True, box_x=[None, None], box_y=[None, None], box_z=None, box_cell_size=None, source_poly=None, source_setup='edges', source_loops=None, inner_area_cell_size=0.3, outer_area_cell_size=10, subsurface_cell_size=None, poly_2d=None, number_of_loops=None, **kwargs*)

Builds the FEM mesh for the secondary field computation.

Needs at least on of the two possible parameter meshes in order to continue.

para_mesh_2d: string or pg.Mesh [None] Used to get the outer dimensions of the FEMMesh.

savename: string [None] Define output save name of FEM mesh. Default name will be generated if None. If no savename is given, the dafaultname will be '_default_LoopMesh' + looptype + number of dipoles.

exportVTK: boolean [False] Turn on optional vtk export.

source_setup: string ['edges'] Defines the way the sources are incorporated into the mesh. "nodes" simply insert the dipole positions (fallback), "edges" defines strait edges between the nodes (usually the best approach). "etra" can be used for a special setup where multiple loops are build in an elongated transmitter with inline receiver array. Raises an exception if source_setup differs from the three options.

source_loops: list [None] If a list of loop classes is given, their tx representation after custEM is implemented in the mesh for custEM magnetic field calculations using automatic source detection.

inner_area_cell_size: float [0.3] Maximum allowed area (m^2) for all cell in the source plane within the source polygons (if closed loop). Very important for kernel calculation! See tutorial for custEM for further explanations.

outer_area_cell_size: float [10] Maximum allowed area (m^2) for all cells in the source plane outside the source polygons (or anywhere for not closed loop). See tutorial for custEM for further explanations.

subsurface_cell_size: float [None] Maximum allowed volume (m^3) for all cells within inner mesh box (not the tetrahedron boundary to 10 km). Optional.

limits: list of len 2 [None] Minimum and maximum y value, the anomalies should be set in the fem mesh. Uses the x limits of the 2D parameter mesh as default if *None*.

custEM: Install via conda on Linux only. See install instructions of comet.

`createLoopMesh` (*savename=None, exportVTK=False, airspace=False, verbose=False, xmax=None, xmin=None, ymax=None, ymin=None, zmin=None*)

Builds the mesh where the loop will be calculated in.

savename: string [None] Saves the created mesh under savename, as long as savenmae is not none. If no basename is given, the dafaultname will be '_default_LoopMesh' + looptype + number of dipoles + '.bms'.

exportVTK: boolean [False] Switch to export the resulting mesh to a vtk with the given savename.

airspace: boolean [False] Enables airspace.

verbose: boolean [False] Turn on berbose mode.

`createSecondaryConfig` (*mod_name, mesh_name, m_dir='.', r_dir='.', pf_name=None, p2=False, approach='E_s', pf_EH_flag='E'*)

Initializes an instance of a secondary config for use of custEM.

Parameters

- **mod_name** (*string*) – Name of the mod instance (for saving and import in mpi environment)
- **mesh_name** (*string*) – Basename of mesh imported by the fenics functions (.h5). Mind the subfolder '/_h5' that will be added to the string.
- **m_dir** (*string*) – Path to mesh directory of custEM.
- **r_dir** (*string*) – Path to result directory of custEM.
- **pf_name** (*string*) – File name under which the primary field will be saved in the appropriate directory of custEM.

effectiveArea()
Returns *self.area* * *self.turns* (0 for not closed loops).

exportFenicsHDF5Mesh(*save_h5*, *dipole_mesh=False*, ***kwargs*)
Exports the mesh in a h5 file. Can save the loopmesh or the dipole mesh seperately.

Need pygimli to work.

Parameters

- **save_h5** (*string*) – Filename of the resulting h5 mesh (hdf5 data container in fenics syntax).
- **dipole_mesh** (*boolean [False]*) – Save dipole mesh instead of loop mesh (Call this function twice if you want to save both meshes).
- **kwargs** (*dict*) – Keyword arguments are redirected to *pygimli.meshtools.exportFenicsHDF5Mesh*

exportVTK(*save_vtk*, *secondary=False*, ***kwargs*)
Exports the field in a vtk file.

Uses the *loopmesh* to save *field* with default configurations in a vtk file.

Parameters

- **save_vtk** (*string*) – Filename of the resulting vtk file.
- **kwargs** (*dict*) – Keyword arguments are redirected to the function *pyhed.IO.savefieldvtk*.

getCustEMLoopTx(*max_length*)

getDefaultLoopMeshBaseName()
Returns string with default base name of the loop mesh.

getParaMesh2D()

initCustEM(*secondary_config=None*, *init_primary_field_class=True*, *procs_per_proc=2*)
Initalizes instance of custEM mod class for FEM calculation.

Parameters

- **secondary_config** (*string or pyhed.SecondaryConfig [None]*) – Initialized secondary config class to be used for the mod instance or path to corresponding file containing the secondary config. Uses secondary_config over *loop.secondary_config*. Throws Exception if both values are None.
- **init_primary_field_class** (*boolean [True]*) – Additionally initializing the primary field class of the mod class instance (used for primary field export).

load(*savename=None*, *config=None*, *config2=None*, *verbose=True*, *load_meshes=True*, *overwrite_dir=False*)
Load Loop from files.

Parameters

- **savename** (*string [None]*) – Basename of the lop class files. Other names are autogenerated using this basename.
- **config** (*string [None]*) – Tell the load function to explicitely load config from given path. Else the saved filepath in the main archive is used.
- **config2** (*string [None]*) – See *config*, but for secondary configuration.
- **verbose** (*boolean [True]*) – Turn on verbose mode.
- **load_meshes** (*boolean [True]*) – If originally saved, the meshes are loaded by default. However, this takes more time then the rest of the load function and can be ommitted if only the other parts are of interest.

`loadFieldMatrix`(*name*, *verbose=True*)

Loads the three matrices needed for recalculation of the primary field from numpy archive. See saevFieldmatrix for detailed description.

Parameters

- **name** (*string*) – Path to file to be loaded.
- **verbose** (*boolean [True]*) – Turn on verbose mode.

`loadSecondaryConfig`(*savename=None*)

Imports previously saved secondary config.

Parameters **savename** (*string [None]*) – Used savename over *loop.sec_savename*. Throws Exception if both values are None. Replaces *loop.sec_savename*.

`model`

`para_mesh_2d`

`prepareSecondaryFieldCalculation`(*savename=None*, *secondary_config=None*, *fem_mesh=None*, *para_mesh_2d=None*, *set_marker=False*, *anomaly_vector=None*, *valid_marker=None*, *verbose=False*, *num_cpu=32*, *force_primary=False*, *export_vtk=False*, *mod_name=None*, ***kwargs*)

Based on the given secondary config a MOD instance using the third party module custEM will be initialized. This includes the optional generation of a FEM suited mesh containing resistivity information from a 2D parameter mesh.

Parameters

- **savename** (*string [None]*) – Name under which loopclass and secondary config (+= '_sec.cfg') are to be saved. Needed for secondary approach.
- **secondary_config** (*pyhed.SecondaryConfig or string [None]*) – Filename of configuration file or initialized class instance of a secondary configuration. Optional if already given manually.
- **fem_mesh** (*pg.Mesh or string [None]*) – FEM suited mesh or filename, respectively. Optional. If not given a suited mesh will be generated if a valid para_mesh_2d is provided.
- **para_mesh_2d** (*pg.Mesh or string [None]*) – 2D parameter mesh providing cell indices for the appending of resitivity information. Needed for automatic FEM mesh generation. Can be set manually beforehand.
- **set_marker** (*boolean [True]*) – Flag to decide if the fem mesh has got the needed marker for the resitivity distribution. Can be omitted if already done and saved (e.g. if same mesh is used again).

- **anomaly_vector** (*np.ndarray [None]*) – Conductivity values [S/m] of the parameter mesh to be used in the seocondary field approach. Uses given value over array found in secondary config. Raises Exception if neither found nor given.
- **ground_marker** (*np.ndarray [None]*) – Corresponding marker for each entry in the anomaly vector. Each marker corresponds to a layer number of the 1d primary field beginning at 1 for the first layer, counting upward (0 belongs to the air layer). None results in np.ones_like(anomaly_vector, dtype=int).
- **verbose** (*boolean [False]*) – Turn on verbose mode.
- **num_cpu** (*integer [32]*) – Maximum number of processes allowed for this task.
- **force_primary** (*boolean [False]*) – Force a recalculation of the primary field.
- **mod_name** (*string or None [None]*) – Overrides mod name. Useful if looping over many loops, as default name could be similar.
- **magnetic** (*boolean [True]*) – Prepares magnetic primary fields. If False only dummies are created to avoid error messages from custEM during import. Set to False if secondary electric approach is used for secondary field calculation.
- **electric** (*boolean [True]*) – Prepares electric primary fields. If False only dummies are created to avoid error messages from custEM during import. Set to False if secondary magnetic approach is used for secondary field calculation.
- **Returns**
- **———**
- **tuple** (*(savename, sec_savename)*) – Absolute file paths for the secondary approach.
- **Usage**
- **——**
- **In order to prepare a secondary field calculation you need**
- **- a secondary config (default is provided)**
- **- a conductivity vector (*)**
- **- a 2d parameter mesh matching the anomalies (*)**
- **- a marker_vector (*)**
- ***if not in secondary config or proviedd beforehand**
- **and optionally either**
- **- fem_mesh (without marker -> set_marker=True (default))**
- **or**
- **- fem_mesh (with marker -> set_marker=False)**
- **or**
- **- no fem_mesh (auto creation)**

save (*savename=None*, *config_savename=None*, *config2_savename=None*, *save_mesh=True*, *save_field=True*)

Saves the loop class in files.

Saves npz archive with loop itself.

Saves config.

Saves secondary config if initialized.

Saves mesh if save_mesh=True.

Saves field if save_field=True.

Parameters

- **savename** (*string [None]*) – File basename for saving loop class and its components.
- **config_savename** (*string [None]*) – Explicit savename for config. Automatically generated if None.
- **config2_savename** (*string [None]*) – Explicit savename for secondary config. Automatically generated if None.
- **save_mesh** (*boolean [True]*) – Saves mesh.
- **save_field** (*boolean [True]*) – Saves fields.

`saveFieldMatrix`(*name*, *verbose=True*)

Saves the three matrices needed for recalculation of the primary field.

A compressed numpy archive is loaded and the matrices are build afterwards, therefore import time is ~20% higher compared to the pure pygimli way (*.field_matrix.save('. . . ')*). However, because the single arrays (indices and values) are saved in one compressed file archive they need only one third space on the hard disk compared to saving three separate matrices using pygimli syntax.

Parameters

- **name** (*string*) – Path for file to be saved.
- **verbose** (*boolean [True]*) – Turn on verbose mode.

`saveLoopMesh`(*savename=None*)

Saves loopmesh using the given savename or an autogenerated name.

Updates *self.loop_mesh_name* in case of changes.

Parameters **savename** – Export path name. Used over default name if given.

`saveSecondaryConfig`(*savename=None*)

Saves secondary config in ASCII file.

Parameters **savename** (*string [None]*) – Used savename over *loop.sec_savename*. Throws Exception if both values are None. Replaces *loop.sec_savename*.

`setAnomalies`(*anomaly*, *sort=True*)

Handle anomaly vector and marker of the 2d parameter mesh.

Parameters

- **anomaly** (*array_like [None]*) – Vector with conductivities in S/m. Expect one entry for each cell in parameter mesh.
- **sort** (*boolean [False]*) – If True, set the same marker for double values in anomaly vector. This is for blocky 2d structures, where only a few different regions are required. Use default False if dealing with smooth inversion results, for example in a structural coupling.

`setDipoleMesh`(*mesh*, *savename='_default_dipole_mesh'*, *verbose=True*)

Sets the dipolemesh and saves it under savename.

Parameters

- **mesh** (*string or mesh instance*) – Pygimli mesh instance or file path to pygimli mesh.

- **savename** (*string [None]*) – Used savename for mesh, if mesh is already a mesh instance.
- **verbose** (*boolean [False]*) – Turn on verbose mode.

setFEMMarker_old (*valid_marker=None*)
Sets and checks the domain marker of the 3D FEM mesh.

Parameters valid_marker (*array_like [None]*) – If None, checks which domains of the 2D mesh are actually transferred to the 3D FEM mesh. The markers are saved in the *valid_marker* attribute. If given, sets vector directly after some checks.

setFEMMesh (*mesh*, *valid_marker=None*, *savename=None*)
Sets the FEM mesh as loopmesh and handles the domain markers.

Parameters

- **mesh** (*string or mesh instance*) – Pygimli mesh instance or file path to pygimli mesh.
- **valid_marker** (*array_like [None]*) – If None, checks which domains of the 2D mesh are actually transferred to the 3D FEM mesh. The markers are saved in the *valid_marker* attribute. If given, sets vector directly after some checks.
- **savename** (*string [None]*) – Useful if multiple loops are using the same mesh (saves diskspace). Ignored if *mesh* is a string already.
- **Calls *_setFEMMarker* is paramesh has been set.**
- **Furthermore calls *updateFEMAnomaly* if anomaly has been set through**
- **either *setParamesh2D* or *setAnomaly***
- **Produces error message if valid_marker array is given, but no paramesh**
- **is found**

setFEMMesh_old (*mesh*, *valid_marker=None*, *savename=None*)
Sets the FEM mesh as loopmesh and handles the domain markers.

Parameters

- **mesh** (*string or mesh instance*) – Pygimli mesh instance or file path to pygimli mesh.
- **valid_marker** (*array_like [None]*) – If None, checks which domains of the 2D mesh are actually transferred to the 3D FEM mesh. The markers are saved in the *valid_marker* attribute. If given, sets vector directly after some checks.
- **savename** (*string [None]*) – Useful if multiple loops are using the same mesh (saves diskspace). Ignored if *mesh* is a string already.

setFType (*ftype*)

setFrequency (*frequency*)
Sets the frequency, not angular frequency for the field calculation.

setLoopMesh (*mesh*, *savename=None*)
Sets the loopmesh.

Parameters

- **mesh** (*string or mesh instance*) – Pygimli mesh instance or file path to pygimli mesh.
- **savename** (*string [None]*) – Used savename for mesh, if mesh is already a mesh instance. Alternatively a default name is generated with *getDefaultLoopMeshBaseName*.

setLoopMeshName (*savename=None*)
Sets loop mesh name or figures it out from sec config.

`setMeshParameters`(*refinement_para=1.0*, *max_area_factor=1.0*, *tetgen_quality=1.2*)
Alters the Parameter responsible for the quality and size used during automatic mesh generation.

Parameters

- **refinement_para** (*float [1]*) – An increase of refinement_para decreases the size of the smallest cell at the dipoles and therefore incrreases the total number of refinement cells around the dipole. Omitts refinement if value is negative.
- **max_area_factor** (*positive float [1]*) – The max_area_para lineary affects the maximum volume of a cell. An increase of the parameter allows for greater cells and therefore decreases the total number of cells outside of the refined section of the mesh. Set to 0.5 for a fine mesh and anywhere near 2 for a coarse mesh. Highly affects the total number of nodes/cells in the mesh.
- **tetgen_quality** (*float [1.2]*) – The tetgen_quality parameter is directly piped to the corresponding tetgen call in the meshgeneration process. Decrease this parameter (e.g. to 1.12) to increase the homogeneity of the triangles. Be careful with this one, tetgen very easy starts to split cells in smaller and smaller pieces and therefore increase the total cellcount to very high values (millions and more).

`setModel`(*rho*, *d=None*, *thickness=True*, *resistivity=True*)
Sets the synthetic 1D layered earth model for dipole calculation.

Parameters

- **rho** (*float or array_like*) – Resistivity/conductivity distribution for a layered earth.
- **d** (*float or array_like or None [None]*) – Thickness or layer depth. Empty (None, 0, or []) for halfspace.
- **thickness** (*boolean [True]*) – The parameter d is used as thickness (True, len(rho) - 1) or depth (False, len(rho)), respectively.
- **resistivity** (*boolean [True]*) – The parameter rho is used as Resistivity (True) or conductivity (False), respectively.

`setParaMesh2D`(*para_mesh_2d*, *limits=None*, *append_boundary=False*, *preserve_edges=False*, *anomaly=None*, *sort=True*, ***kwargs*)
Sets 2D parameter mesh for secondary field calculation.

Parameters

- **para_mesh_2d** (*string or pg.Mesh*) – 2D parameter mesh or path to mesh.
- **limits** (*[float, float] or None*) – Minimum and maximum values for y of the area where 2D parameters are to be transferred to the 3D FEM mesh. Default are the x extension of the 2D parameter mesh.
- **append_boundary** (*boolean [False]*) – Fills in an additional boundary with prolongated resistivity values around the transferred 2D values. This is useful as it reduces artifacts at the edge of the 2D domain oin the FEM mesh.
- **anomaly** (*None or np.ndarray [None]*) – Optionally. Alternatively use *setAnomalies*. Anomaly vector (conductivity vector) with values for each cell in the 2D parameter domain. Attention: conductivity is used, not resistivity!
- **sort** (*boolean [False]*) – Optionally. Alternatively use *setAnomalies*. If True, set the same marker for double values in anomaly vector. This is for blocky 2d structures, where only a few different regions are required. Use default False if dealing with smooth inversion results, for example in a structural coupling.
- **kwargs to *appendTriangleBoundary***

- **Calls *setAnomalies* of anomaly is given.**
- **Furthermore calls *updateFEMAnomaly* if FEMMesh has been set already.**

`setParaMeshMarkerAndVals`(*anomaly=None*, *sort=True*)
Handle anomaly vector and marker of the 2d parameter mesh.

Parameters

- **anomaly** (*array_like [None]*) – Vector with conductivities in S/m. Expect one entry for each cell in parameter mesh. If not given, and sort is True an error is raised.
- **sort** (*boolean [False]*) – If True, set the same marker for double values in anomaly vector. This is for blocky 2d structures, where only a few different regions are required. Use default False if dealing with smooth inversion results, for example in a structural coupling.

`setPrimaryConfig`(*config*)
Sets the primary config which handles the resistivity distribution as well as the frequency of the primary field. For setting the 1D model directly see *setModel*.

Parameters **config** (*path or comet.pyhed.config.Config instance*) – Configuration class instance or file path.

`setSecondaryConfig`(*secondary_config*)
Sets class attribute with secondary config or loads file.

Parameters **secondary_config** (*string or pyhed.SecondaryConfig*) – Seondary config class instance or file path.

`show`(***kwargs*)
Plots the Loopdiscretisation and the dipole directions and Length. For inspection of the loop-class and debugging purpose. Or for your curiosity.

Parameters **kwargs** (*dict*) – Keyword arguments are redirected to *pyhed.plot.plot_bib.showLoop*.

`updateFEMAnomaly`(*anomaly=None*, *set_marker=True*, *set_attributes=False*, *vtk_name=None*, *ground_marker=None*, *export_H5=False*, *sort=True*)
Transfers resistivity anomalies from 2D para mesh in FEM mesh.

Parameters

- **anomaly_vector** (*array_like [None]*) – Array containing the resistivity anomalies of the 2D parameter mesh. If None, the secondary config is asked for a anomaly vector. (For setting the marker for exmaple).
- **set_marker** (*boolean [True]*) – Transfers the marker from the parameter mesh to the FEM mesh. This only has to be done once and can then switched off for performance.
- **set_attribute** (*boolean [False]*) – Sets the attribute in the FEM mesh for debugging purposes. The anomaly vector for calculation is stored in secondary_config.
- **vtk_name** (*string [None]*) – Optional vtk export with name = *vtk_name* if *vtk_name* is not None.
- **ground_marker** (*array_like [None]*) – Corresponding marker for each entry in the anomaly vector. Each marker corresponds to a layer number of the 1d primary field beginning at 1 for the first layer, counting upward (0 belongs to the air layer). None results in np.ones_like(anomaly_vector, dtype=int).

`updateFEMAnomaly_old`(*anomaly=None*, *set_marker=True*, *set_attributes=False*, *vtk_name=None*, *ground_marker=None*, *export_H5=False*)
Transfers resistivity anomalies from 2D para mesh in FEM mesh.

Parameters

- **anomaly_vector** (*array_like [None]*) – Array containing the resistivity anomalies of the 2D parameter mesh. If None, the secondary config is asked for a anomaly vector. (For setting the marker for exmaple).
- **set_marker** (*boolean [True]*) – Transfers the marker from the parameter mesh to the FEM mesh. This only has to be done once and can then switched off for performance.
- **set_attribute** (*boolean [False]*) – Sets the attribute in the FEM mesh for debugging purposes. The anomaly vector for calculation is stored in secondary_config.
- **vtk_name** (*string [None]*) – Optional vtk export with name = *vtk_name* if *vtk_name* is not None.
- **ground_marker** (*array_like [None]*) – Corresponding marker for each entry in the anomaly vector. Each marker corresponds to a layer number of the 1d primary field beginning at 1 for the first layer, counting upward (0 belongs to the air layer). None results in np.ones_like(anomaly_vector, dtype=int).

`comet.pyhed.loop.loop_bib.`**`buildCircle`**(*r*, *num_segs=None*, *max_length=None*, *P=(0, 0, 0)*, *dipole_clockwise=True*, *savename=None*, *turns=1*, ***kwargs*)

this function builds a n-segmented coil in the x-y-plane around the point P = (X, Y, Z), with Radius r. The first point is at the hightest y value with x = X and therefore the point where the dipole is x-directed. Its the point were the field can calculated directly without rotation, but with translation. The rest of the coil is build clockwise.

Parameters

- **r** (*float*) – Radius of the loop.
- **num_segs** (*integer [None]*) – Total number of segments to be used to discretize the Loop. Used internally to define the max_length of a dipole. Inferior usage compared to max_length.
- **max_length** (*float [None]*) – Defines the minimum length of a dipole used for the discretization of the loop. Superior usage compared to num_segs.
- **P** (*list or np.ndarray*) – 2D or 3D coordinate of the mid point of the loop.
- **dipole_clockwise** (*[True]*) – Define the dipole to be ordered in a clockwise direction.
- **savename** (*string [None]*) – Basename for the loop.
- **turns** (*integer [1]*) – Number of turns of a closed loop.

Returns Pyhed loop class instance.

Return type pyhed.loop

Example

```
>>> import pyhed as ph  # this works if pyhed is in your path.
>>> l = ph.loop.buildCircle(10, 11)
>>> print(l)
>>> print(l.config)
>>> l.show()
```

`comet.pyhed.loop.loop_bib.`**`buildDipole`**(*Pos*, *length=1*, *angle=0*, ***kwargs*)

Parameters

- **Pos** (*list*) – 2D or 3D coordinate of the dipole.
- **length** (*float [1]*) – Dipole legth.
- **angle** (*[0]*) – Dipole direction positive clockwise from the x-aixs.
- **kwargs** (*dict*) – Keyword arguments are redirected to the loop class.

Returns Pyhed loop class instance.

Return type pyhed.loop

Example

```
>>> import pyhed as ph  # this works if pyhed is in your path.
>>> l = ph.loop.buildDipole([-3, -3], length=1.3, angle=45)
>>> print(l)
>>> print(l.config)
>>> l.show()
```

`comet.pyhed.loop.loop_bib.`**`buildDummy`**(***kwargs*)

Creates an empty dummy loop class to gain access to certain functionalities.

`comet.pyhed.loop.loop_bib.`**`buildEdgeSourceDiscretization`**(*surface, pos, phi, ds, closed=True*)

Internal function. Used to implement every dipole in the FEM mesh using an appropriate edge that represents it.

`comet.pyhed.loop.loop_bib.`**`buildEtraPoly`**(*x_min, x_max, small, marker=0*)

`comet.pyhed.loop.loop_bib.`**`buildEtraSourceDiscretization`**(*poly, edgelength, max_length=0.251, x_left=None, n_segs=None, number_of_loops=8*)

Internal function.

Implements an etra shaped source in the FEM mesh. Cannot use buildEdgeSourceDiscretization due to overlapping edges.

`comet.pyhed.loop.loop_bib.`**`buildEtraSurvey`**(*edgelength, return_measurements=False, origin=[0, 0], num_loops=8, max_length=None, savenames=None, **kwargs*)

Special etra survey for NMR applications.

`comet.pyhed.loop.loop_bib.`**`buildFig8`**(*points, num_segs=3, max_length=None, mid=0, dipole_clockwise=True, turns=1, **kwargs*)

Builds a figure-of-eight Loop with respect to the given corner Points. This function is part of the pyhed.loops library and returns a 'loop'-class object suitable to calculate electric and magnetic fields based on horizontal electric dipoles. Please always check the loop consistency with the buildin .show() command (see Example) before calculating with experimetal loop layouts.

Parameters

- **num_segs** (*integer [12]*) – Total number of segments to be used to discretize the Loop. Used internally to define the max_length of a dipole. Inferior usage compared to max_length.
- **max_length** (*float [None]*) – Defines the minimum length of a dipole used for the discretization of the loop. Superior usage compared to num_segs.

- **points** (*array_like*) – Points can be of shape (2, 3), two points with three coordinates (x, y, z) and the algorithm will build a loop with edges parallel to the coordiante axes.

  Although the point coordiantes are given with z values, the current implementation of COMET is not able to allow for any z-values other than zero, i appologize for the inconvenience. This flaw will be adressed as soon as COMET moves towards 2D resistivity structures.

- **max_length** (*float or None (None)*) – The discretisation between the cornerpoints of the loop can be sampled by any rate (m) you choose. Since the total number of points between the corner points of the loop has to be an integer, the real distance between the points will always be smaller or equal to max_length.

  Its highly recommended to use at least 10 dipoles between the different points, and therefore a total number of dipoles of >= 50 - 60. This leads to a natural max_length of 1/10 the smaller edge of the figure-of-eight loop.

- **num_segs** (*integer (3)*) – The number of segments between the corner points of the loop can also be given directly, but mention that the max_length value (not None) will have priority. Usually max_length will lead to more homogeneous distributions of dipoles between the corner and midpoints of a figure-of-eight loop.

- **mid** (*integer (0)*) – The middle connection depends on the value "mid". The default value (0) sets the middle lines parrallel to the y-axis. Other values are setting the line parrallel to the x axis, respectively.

- **dipole_clockwise** (*boolean (True)*) – The dipoles are orientated clockwise with respect to the first half of the loop or counterclockwise if this switch is set the False. The first half is considered to be the half connected to the upper left point of the loop boundary, so either the left or the upper loop depending on mid.

- **Possible kwargs are** (*savename. Please see "buildLoop" for more*)

- **details.**

Returns Pyhed loop class instance.

Return type pyhed.loop

Example

```
>>> import pyhed as ph  # this works if pyhed is in your path.
>>> l = ph.loop.buildSquare(k=3.25, max_length=0.14)
```

Example

```
>>> import pyhed as ph
>>> p1 = (-1, 1, 0)
>>> p2 = (1, -1, 0)
>>> fig8 = ph.loop.buildfig8((p1, p2), max_length=0.1)
>>> print(fig8)
>>> print(fig8.config)
>>> fig8.show()
```

`comet.pyhed.loop.loop_bib.`**`buildFig8Circle`**(*r*, *num_segs=None*, *max_length=None*, *P=(0, 0, 0)*, *savename=None*, *turns=1*, ***kwargs*)

Build a figure-of-eight loop with circular loops around point P = (X, Y, Z), with Radius r.

Parameters

- **r** (*float*) – Radius of the loop.
- **num_segs** (*integer [None]*) – Total number of segments to be used to discretize the Loop. Used internally to define the max_length of a dipole. Inferior usage compared to max_length. In this case divided between the two circular loops.
- **max_length** (*float [None]*) – Defines the minimum length of a dipole used for the discretization of the loop. Superior usage compared to num_segs.
- **P** (*array_like [(0., 0., 0.)]*) – 2D or coordinate of the mid point of the loop.
- **savename** (*string [None]*) – Basename for the loop.
- **turns** (*integer [1]*) – Number of turns of a closed loop.

Returns Pyhed loop class instance.

Return type *pyhed.loop.Loop*

`comet.pyhed.loop.loop_bib.`**`buildLine`**(*Start*, *End*, *num_segs=0*, *max_length=None*, *savename=None*, *grounded=True*, ***kwargs*)

Parameters

- **Start** (*list*) – 2D or 3D coordinate of start of the line (z value will be ignored for now).
- **Start** (*list*) – 2D or 3D coordinate of end of the line (z value will be ignored for now).
- **num_segs** (*integer [0]*) – Total number of segments to be used to discretize the Loop. Used internally to define the max_length of a dipole. Inferior usage compared to max_length.
- **max_length** (*float [None]*) – Defines the minimum length of a dipole used for the discretization of the loop. Superior usage compared to num_segs.
- **savename** (*string [None]*) – Basename for the loop.
- **dipole_clockwise** (*[True]*) – Define the dipole to be ordered in a clockwise direction.
- **turns** (*integer [1]*) – Number of turns of a closed loop.
- **kwargs** (*dict*) – Keyword arguments are redirected to the loop class.

Returns Pyhed loop class instance.

Return type pyhed.loop

Example

```
>>> import pyhed as ph  # this works if pyhed is in your path.
>>> l = ph.loop.buildLine([-3, -3], [4, 2], max_length=0.14)
>>> print(l)
>>> print(l.config)
>>> l.show()
```

`comet.pyhed.loop.loop_bib.`**`buildLoop`**(*Points*, *num_segs=1*, *max_length=None*, *savename=None*, *grounded=False*, *ltype=None*, *dipole_clockwise=None*, *turns=1*, ***kwargs*)

Creates an arbitrary shaped loop out of given coordinates.

The returnes object is an initialized loop class. Most general function to build a loop and called by most of the other specialized functions after input preparation.

Parameters

- **Points** (*list*) – List or Array containing 2D or 3D coordinates of shape (n, 2 or 3) for n corner point of an arbitrary shaped polygon (z_vlaues will be set to 0 or now).
- **num_segs** (*integer [1]*) – Total number of segments to be used to discretize the Loop. Used internally to define the max_length of a dipole. Inferior usage compared to max_length.
- **max_length** (*float [None]*) – Defines the minimum length of a dipole used for the discretization of the loop. Superior usage compared to num_segs.
- **savename** (*string [None]*) – Basename for the loop. Trigger to save the loop.
- **grounded** (*boolean [False]*) – Defines wether the loop is closed or not. Also defines which default field mode will be calculated, as 'tm' field of a closed loop is zero.
- **dipole_clockwise** (*[True]*) – Define the dipole to be ordered in a clockwise direction.
- **turns** (*integer [1]*) – Number of turns of a closed loop.
- **kwargs** (*dict*) – Keyword arguments are redirected to the loop class.

Returns Pyhed loop class instance.

Return type pyhed.loop

Example

```
>>> import pyhed as ph  # this works if pyhed is in your path.
>>> points = [[-5, -5], [-10, 5], [2.3, 3.14], [7, -7]]
>>> l = ph.loop.buildLoop(points, max_length=0.64)
>>> print(l)
>>> print(l.config)
>>> l.show()
```

comet.pyhed.loop.loop_bib.**buildPointSourceDiscretization**(*surface*, *pos*)

Internal function. Used to implement each source dipole as simple node in the FEM mesh.

comet.pyhed.loop.loop_bib.**buildRectangle**(*points*, *num_segs=1*, *max_length=None*, *savename=None*, *dipole_clockwise=None*, *turns=1*, ***kwargs*)

Creates a rectangular shaped loop and manages dipole discretization.

Creates a rectangular loop out of four given corner points, with a given discretisation between the points. Per default the function returns the position of the dipoles, the angle between its orientation and the x-direction and the dipole Length it represents.

Parameters

- **points** (*list*) – List or Array containing 2D or 3D coordinates (however z-values ignored for now). In case of a rectangle two or four coords are needed. In case of two coordinates, the rectangle will have edges parallel to the coordinate axes.
- **num_segs** (*integer [1]*) – Total number of segments to be used to discretize the Loop. Used internally to define the max_length of a dipole. Inferior usage compared to max_length.
- **max_length** (*float [None]*) – Defines the minimum length of a dipole used for the discretization of the loop. Superior usage compared to num_segs.
- **savename** (*string [None]*) – Basename for the loop. Trigger to save the loop.
- **dipole_clockwise** (*[True]*) – Define the dipole to be ordered in a clockwise direction.
- **turns** (*integer [1]*) – Number of turns of a closed loop.

- **kwargs** (*dict*) – Keyword arguments are redirected to the loop class.

Returns Pyhed loop class instance.

Return type pyhed.loop

Example

```
>>> from comet import pyhed as ph
>>> l = ph.loop.buildRectangle([[-5, -5], [5, 5]], max_length=0.64)
>>> print(l)
>>> print(l.config)
>>> l.show()
```

`comet.pyhed.loop.loop_bib.`**`buildSpiral`**(*r1, r2, sp_turns=2, sp_segs=36, max_length=None, P=(0, 0, 0), dipole_clockwise=True, savename=None, theta=90.0, **kwargs*)

this function builds a spiral coil with sp_turns number of turns in the x-y-plane around the point P = (X, Y, Z), with inner radius r1 and outer radius r2. The angle theta defines the start of the spiral (theta = 0 -> East, theta = 90 -> North, etc.). The spiral is build clockwise from r1 to r2.

Parameters

- **r1** (*float*) – Inner radius of the spiral.
- **r2** (*float*) – Outer radius of the spiral.
- **sp_turns** (*integer [None]*) – Total number of spiral turns.
- **sp_segs** (*integer [None]*) – Total number of segments to be used to discretize the spiral.
- **max_length** (*float [None]*) – Defines the minimum length of a dipole used for the discretization of the loop.
- **P** (*list or np.ndarray*) – 2D or 3D coordinate of the mid point of the loop.
- **dipole_clockwise** (*[True]*) – Define the dipole to be ordered in a clockwise direction.
- **savename** (*string [None]*) – Basename for the loop.
- **theta** (*float, deg [90.0]*) – Orientation of start-end-connection of the spiral in degrees.

Returns Pyhed loop class instance.

Return type pyhed.loop

Example

```
>>> import pyhed as ph  # this works if pyhed is in your path.
>>> l = ph.loop.buildSpiral(1, 2, sp_turns=5)
>>> print(l)
>>> print(l.config)
>>> l.show()
```

`comet.pyhed.loop.loop_bib.`**`buildSquare`**(*k=1, num_segs=12, P=(0, 0, 0), max_length=None, savename=None, dipole_clockwise=True, turns=1, **kwargs*)

Square loop around P with edge length k.

Parameters

- **k** (*float [1]*) – Length of one edge.
- **num_segs** (*integer [12]*) – Total number of segments to be used to discretize the Loop. Used internally to define the max_length of a dipole. Inferior usage compared max_length.
- **max_length** (*float [None]*) – Defines the minimum length of a dipole used for the discretization of the loop. Superior usage compared to num_segs.
- **savename** (*string [None]*) – Basename for the loop. Trigger to save the loop.
- **dipole_clockwise** (*[True]*) – Define the dipole to be ordered in a clockwise direction.
- **turns** (*integer [1]*) – Number of turns of a closed loop.
- **kwargs** (*dict*) – Keyword arguments are redirected to the loop class

Returns Pyhed loop class instance.

Return type pyhed.loop

Example

```
>>> from comet import pyhed as ph
>>> l = ph.loop.buildSquare(k=3.25, max_length=0.24)
>>> print(l)
>>> print(l.config)
>>> l.show()
```

`comet.pyhed.loop.loop_bib.`**`calcWithEmpymod`**`(`*loop*, *use_bipole=False*`)`

`comet.pyhed.loop.loop_bib.`**`computeLoopPositions`**`(`*Coordinates*, *ltype='arbitrary'*, *middle=None*, *grounded=True*`)`

This function calculates the position of the dipoles in order to represent an arbitrary shaped loop with the given coordinates, the angle between its orientation and the x-direction and the dipole Length it represents.

Parameters

- **Coordinates** (*np.ndarray*) – Input list/array of points of shape: (n, 3) for n dipoles.
- **ltype** (*string*) – Defines the general type of the loop and therefore some internal attributes. Choices are:
  - 'rectangle' (also for square loops)
  - 'circle'
  - 'arbitrary' (for all other loops)
- **middle** (*np.ndarray [None]*) – Midpoint of the circular loop to calculate radius correct (and therefore the correct source coordinates).
- **grounded** (*boolean [True]*) – For non grounded wires there are dipole placed bewtween the last coordinate point and the first. This is ommitted for grounded wires.

`comet.pyhed.loop.loop_bib.`**`copyPrimaryFields`**`(`*rdir1*, *rdir2*`)`

For recalculation purpose it sometimes is uneccessary to calc the primary fields again. This function copies the primery fields from on custEM result dir (rdir1) to another (rdir2). names of the fields are not changed. Only the .h5 files are copied.

`comet.pyhed.loop.loop_bib.`**`createEtraMesh`**(*loops*, *mesh2d*, *anomaly*, *savename=None*, *extend_x=0.0*, *extend_y=-0.3*, *extend_z=-0.5*, *max_volume=25.0*, *append_boundary=True*, *sort=True*, *return_loop=False*)

Creates a finite element mesh suited for ETRA surveys.

`comet.pyhed.loop.loop_bib.`**`createMultipleLoopMesh`**(*loops*, *savename=None*, *source_setup='etra'*, *triangle_quality=33.8*, *source_max_area=None*, *inner_area_volume=None*, *mid_area_volume=None*, *outer_area_volume=None*, *minx=None*, *maxx=None*, *miny=None*, *maxy=None*, *minz=None*, *air_refinement=False*, *source_poly=None*)

Build a suited mesh for magnetic field calculation for NMR purpose. Sources are included in a way defined by **source_setup**.

Parameters

- **loops** (*ph.loop.loop or array_like*) – Input loops for which the mesh shall be created.
- **savename** (*string [None]*) – Savename for mesh (.bms will be added).
- **source_setup** (*string ['etra']*) – In case of multiple loops, the source_setup is important to define how the loops are included in the mesh. Default is 'etra' for NMR Etra setups. Alternatively 'edges' can be used to implement each dipole as edge with the dipole as midpoint, length as well as direction is defined by the dipole. This is not working for overlaping loops (e.g. Etras). For those a source_poly can be provided or 'nodes' is chosen to simply implement each dipole as node in the mesh.
- **triangle_quality** (*float [34.0]*) – The surface where the sources are implemented is meshed in 2D an then later inserted in the 3D mesh. This controls the triangle quality for this surface mesh.
- **source_max_area** – maximum area allowed in the 2D surface mesh (sources and 5 meter around the sources). Automatically defined if None (based on size of the area to ensure a minimum amount of trinagle cells). 2D surface mesh is exported if logger is set to debug level (10).
- **inner_area_volume** (*float [None]*) – The inner refinement volume is defined 5 meter around and below the sources. The maximum cell volume can be defined here. Automatically defined if None (based on size of the volume to ensure a minimum amount of cells).
- **mid_area_volume** (*float [None]*) – The median refinement volume is defined through minx, maxx, miny, maxy, and minz meter around and below the sources. The maximum cell volume can be defined here. Automatically defined if None (based on size of the volume to ensure a minimum amount of cells).
- **outer_area_volume** (*float [None]*) – The outer sides of the mesh (3 times miny, maxx, miny, maxy, minz) is to ensure interpolation of the field values are secured without the need for extrapolation. Usually the max cell volume is not contraint to minimize the computational effort.
- **minx** (*float [None]*) – Minimum x extention (in addition to the extend of the inner refinement area) of median refinement volume. If None this value is defined as maximum distance of two dipoles of the input loops.

- **maxx** (*float [None]*) – Maximum x extention (in addition to the extend of the inner refinement area) of median refinement volume. If None this value is defined as maximum distance of two dipoles of the input loops.
- **miny** (*float [None]*) – Minimum y extention (in addition to the extend of the inner refinement area) of median refinement volume. If None this value is defined as maximum distance of two dipoles of the input loops.
- **maxy** (*float [None]*) – Maximum y extention (in addition to the extend of the inner refinement area) of median refinement volume. If None this value is defined as maximum distance of two dipoles of the input loops.
- **minz** (*float [None]*) – Maximum z extention (in addition to the extend of the inner refinement area) of median refinement volume. If None this value is defined as maximum distance of two dipoles of the input loops. 1/3 of the value is used for maxz if air refinement is enabled.
- **air_refinement** (*boolean [False]*) – If true the airspace is meshed as well.
- **source_poly** (*pg.Mesh or plc [None]*) – For unusual or overlapping, non etra, sources a piecewise linear complex (plc) can be created using the pygimli mesh- and polytools. This is then used as source definition for the 2D source layer mesh. Any region markers with area constraints will be considered, no additional markers will be set.

`comet.pyhed.loop.loop_bib.`**`createSeparatedFEMMesh`**(**loops*, *para_mesh_2d=None*, ***kwargs*)

Build a mesh suited for EM secondary field calculation if more than one loop is used.

`comet.pyhed.loop.loop_bib.`**`createSeparatedLoopMesh`**(**loops*, *dipole_mesh=False*, ***kwargs*)

Build a mesh suited for EM primary field calculation if using more than one loop.

`comet.pyhed.loop.loop_bib.`**`dipolePosFromSimpleLoop`**(*r*, *n*, *P=(0, 0, 0)*, *drop_tol=1e-14*)

Convienience function to have fast access to circular loop coordinates.

`comet.pyhed.loop.loop_bib.`**`loadLoop`**(*name*, ***kwargs*)

Imports loop from file archive. See *ph.loop.Loop.load* for details.

`comet.pyhed.loop.loop_bib.`**`loadLoops`**(*name*, *num=8*, *load_meshes=False*, *cfg_name=None*, *cfg2_name=None*, *overwrite_dir=False*)

Loads n loops with name = …{n}…

`comet.pyhed.loop.loop_bib.`**`mergeLoops`**(**loops*, *true_merge=False*, *config=None*)

Merges given loops to one and optionally merges equal dipoles.

Parameters

- **loops** (*loops-classes*) – Loops to be merged.
- **true_merge** (*boolean [True]*) – Switch for a complete merge of all dipoles to with respect to their phi and dipole length. Attention this can be problematic when merging edge-to-edge loops (for mesh creation for example). If False only the dipole positions are merged not phi/ds (there replaced with dummy values).
- **config** (*ph.config-instance or string [None]*) – Sets config of merged loop either per index (cofig is taken from the corresponding loop) or give a new config. If None the config of the first loop is used instead.

`comet.pyhed.loop.loop_bib.`**`totalFieldCalculation`**(*custem_config*, *num_cpu=16*)

comet.pyhed.loop.loop_para module

Part of comet/pyhed/loop

`comet.pyhed.loop.loop_para.`**`CalculationWorker`**(*num*, *index_start*, *pos_alpha_len*, *out_queue*, *end_queue*, *rho*, *d*, *f*, *current*, *mode*, *ftype*, *outPos*, *verbose*, *drop_tol*, *src_z*, *switch_hankel*, *log_level*)

`comet.pyhed.loop.loop_para.`**`CalculationWorker_perDipole`**(*num*, *in_queue*, *out_queue*, *rho*, *d*, *f*, *current*, *mode*, *ftype*, *outPos*, *drop_tol*)

`comet.pyhed.loop.loop_para.`**`calcFieldMatrix_para`**(*dipoleMeshName*, *dipoleNodeCount*, *loop_mesh*, *PosPhiDs*, *verbose=False*, *num_cpu=12*)

`comet.pyhed.loop.loop_para.`**`loopCalculation`**(*OutMesh*, *PosPhiDs*, *rho*, *d*, *f*, *current*, *mode*, *ftype*, *verbose=False*, *cell_center=False*, *num_cpu=12*, *max_node_count=None*, ***kwargs*)

`comet.pyhed.loop.loop_para.`**`loopCalculation_perDipole`**(*OutMesh*, *PosPhiDs*, *rho*, *d*, *f*, *current*, *mode*, *ftype*, *cell_center=False*, *num_cpu=12*, ***kwargs*)

`comet.pyhed.loop.loop_para.`**`loopInterpolation`**(*dipoledata*, *SrcMeshName*, *OutMesh*, *PosPhiDs*, *verbose=False*, *cell_center=False*, *num_cpu=12*)

Module contents

Module comet/pyhed/loop

comet.pyhed.misc package

Submodules

comet.pyhed.misc.console_call module

Part of comet/pyhed/misc

`comet.pyhed.misc.console_call.`**`embeddedMPIRun`**(*scriptname*, **scriptargs*, ***kwargs*)

Parameters

- **scriptargs** – All given arguments will be piped to the mpirun. Kwargs has to be given in two arguments.
- **kwargs** – Only for use in this function, kwargs are not piped to the embeddedMPIRun.
- **kwargs**
- **——**
- **python_to_call** (*string [‘python’ or ‘python3’]*) – Programname to be called with mpirun.
- **number_of_processes** (*int [12]*) – Number of processes for mpirun.

`comet.pyhed.misc.console_call.`**`embeddedMPIRun_bash`**(*scriptname*, **scriptargs*, ***kwargs*)

Parameters

- **scriptargs** – All given arguments will be piped to the mpirun. Kwargs has to be given in two arguments.
- **kwargs** – Only for use in this function, kwargs are not piped to the embeddedMPIRun.
- **kwargs**
- **——**
- **python_to_call** (*string ['python' or 'python3']*) – Programname to be called with mpirun.
- **number_of_processes** (*int [12]*) – Number of processes for mpirun.

`comet.pyhed.misc.console_call.`**`local_apps`**(*name*, **args*, ***kwargs*)

Finds local apps in the comet/pyhed/apps directory by name and call an embeddedMPIRun and returns the subprocess.call.

`comet.pyhed.misc.console_call.`**`local_apps_bash`**(*name*, **args*, ***kwargs*)

Finds local apps in the comet/pyhed/apps directory by name and call an embeddedMPIRun and returns the subprocess.call.

`comet.pyhed.misc.console_call.`**`tetgen151`**(*meshname*, *maxArea=''*, *quality=1.2*, *path=None*, *verbose=False*, *paraString=None*, *preserve_facets=False*, *addparams=''*, *suppress_tetgen_files=False*, *vtk_out=True*)

TetGen A Quality Tetrahedral Mesh Generator and 3D Delaunay Triangulator Version 1.5 May 31, 2014

Copyright (C) 2002 - 2014

What Can TetGen Do?

TetGen generates Delaunay tetrahedralizations, constrained Delaunay tetrahedralizations, and quality tetrahedral meshes.

Command Line Syntax:

Below is the basic command line syntax of TetGen with a list of short descriptions. Underscores indicate that numbers may optionally follow certain switches. Do not leave any space between a switch and its numeric parameter. 'input_file' contains input data depending on the switches you supplied which may be a piecewise linear complex or a list of nodes. File formats and detailed description of command line switches are found in user's manual.

tetgen [-pYrq_Aa_miO_S_T_XMwcdzfenvgkJBNEFICQVh] input_file

-p	Tetrahedralizes a piecewise linear complex (PLC).
-Y	Preserves the input surface mesh (does not modify it).
-r	Reconstructs a previously generated mesh.
-q	Refines mesh (to improve mesh quality).
-R	Mesh coarsening (to reduce the mesh elements).
-A	Assigns attributes to tetrahedra in different regions.
-a	Applies a maximum tetrahedron volume constraint.
-m	Applies a mesh sizing function.
-i	Inserts a list of additional points.
-O	Specifies the level of mesh optimization.

-S	Specifies maximum number of added points.
-T	Sets a tolerance for coplanar test (default 1e-8).
-X	Suppresses use of exact arithmetic.
-M	No merge of coplanar facets or very close vertices.
-w	Generates weighted Delaunay (regular) triangulation.
-c	Retains the convex hull of the PLC.
-d	Detects self-intersections of facets of the PLC.
-z	Numbers all output items starting from zero.
-f	Outputs all faces to .face file.
-e	Outputs all edges to .edge file.
-n	Outputs tetrahedra neighbors to .neigh file.
-v	Outputs Voronoi diagram to files.
-g	Outputs mesh to .mesh file for viewing by Medit.
-k	Outputs mesh to .vtk file for viewing by Paraview.
-J	No jettison of unused vertices from output .node file.
-B	Suppresses output of boundary information.
-N	Suppresses output of .node file.
-E	Suppresses output of .ele file.
-F	Suppresses output of .face and .edge file.
-I	Suppresses mesh iteration numbers.
-C	Checks the consistency of the final mesh.
-Q	Quiet: No terminal output except errors.
-V	Verbose: Detailed information, more terminal output.
-h	Help: A brief instruction for using TetGen.

-o2 quadratic mesh

Examples of How to Use TetGen:

'tetgen object' reads vertices from object.node, and writes their Delaunay tetrahedralization to object.1.node, object.1.ele (tetrahedra), and object.1.face (convex hull faces).

'tetgen -p object' reads a PLC from object.poly or object.smesh (and possibly object.node) and writes its constrained Delaunay tetrahedralization to object.1.node, object.1.ele, object.1.face, (boundary faces) and object.1.edge (boundary edges).

'tetgen -pq1.414a.1 object' reads a PLC from object.poly or object.smesh (and possibly object.node), generates a mesh whose tetrahedra have radius-edge ratio smaller than 1.414 and have volume of 0.1 or less, (and writes the mesh to object.1.node, object.1.ele, object.1.face, and object.1.edge… not anymore)

comet.pyhed.misc.load_save module

Part of comet/pyhed/misc

`comet.pyhed.misc.load_save.`**`dump2Json`**(*json_name=None*, ***kwargs*)
Dumps all keyword-value combinations given in kwargs into a json file. Supported variable types can by found here:

https://docs.python.org/3/library/json.html

`comet.pyhed.misc.load_save.`**`exportSparseMatrixAsNumpyArchive`**(*name*, **sparseMats*)

`comet.pyhed.misc.load_save.`**`json2Dict`**(*name*)
Reads a json file from disk and converts it to python dictionary.

`comet.pyhed.misc.load_save.`**`loadSparseMatrixFromNumpyArchive`**(*name*, *csr=True*, *verbose=True*)

comet.pyhed.misc.matrixWrapper module

Part of comet/pyhed/misc

`comet.pyhed.misc.matrixWrapper.`**`ComplexNumpyMatrix`**(*mat*, *copy=False*)

`comet.pyhed.misc.matrixWrapper.`**`NumpyMatrix`**(*mat*, *copy=False*)
Matrix Wrapper for for ndarrays, providing syntax for pygimli c++ core algorithms (rows, cols, mult, transMult, save(numpy)).

class `comet.pyhed.misc.matrixWrapper.`**`RealNumpyMatrix`**(*mat*, *copy=False*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Matrix Wrapper for for ndarrays, providing syntax for pygimli c++ core algorithms. Holds reference to a real matrix, providing the correct multiplication algorithms for the pygimli inversion process.

`cols`()

`mult`(*vector*)

`rows`()

`save`(*name*)

`transMult`(*vector*)

comet.pyhed.misc.mesh_tools module

Part of comet/pyhed/misc

`comet.pyhed.misc.mesh_tools.`**`createConstraintMesh`**(*mesh*)
Creates a refined mesh, where every triangle is divided in three triangles, with the midpoint of the original cells as new node. This is only done for each boundary that has a right and a left cell (no boundary edges) and is usefull for displaing the boundary constraints of the original mesh.

```
>>> import pygimli as pg
>>> import numpy as np
>>> mesh = pg.load('invmesh.bms')
>>> cw = np.load('constraints.npy')
>>> cmesh = createConstraintMesh(mesh)
>>> pg.show(cmesh, data=cw[np.array(cmesh.cellMarkers(), dtype=int)])
```

`comet.pyhed.misc.mesh_tools.`**`createH2`**`(`*inmesh*, *order=1*, *integration_mat=False*`)`
Creates a H-2**N refined mesh with order N and Integration matrix.

`comet.pyhed.misc.mesh_tools.`**`sameGeometry`**`(`*mesh1*, *mesh2*, *atol=1e-08*, *rtol=1e-05*`)`

comet.pyhed.misc.mpi_tools module

Part of comet/pyhed/misc

`comet.pyhed.misc.mpi_tools.`**`abortIfError`**`()`

`comet.pyhed.misc.mpi_tools.`**`importCustemResults`**`(`*name*, *ntx=1*`)`

`comet.pyhed.misc.mpi_tools.`**`saveFenicsField`**`(`*savename_base*, *loop*, *secondary=False*, *htr=None*, *hti=None*, *hsr=None*, *hsi=None*`)`
Save fenics fields from loppclass object in mpirun environment in gimli single core sorting for later use in single core tasks.

savename total field :

- savename_base + '_total.npy'

if secondary is True:

savename secondary field:

- savename_base + '_secondary.npy'

Saved variables are:

- loop.secMOD.PP.H_t_r_cg,
- loop.secMOD.PP.H_t_i_cg,
- loop.secMOD.PP.H_s_r_cg,
- loop.secMOD.PP.H_s_i_cg

comet.pyhed.misc.para_lib module

Part of comet/pyhed/misc

`comet.pyhed.misc.para_lib.`**`InterpolationMatrix_para`**`(`*mesh_name*, *out_coords*, *maxCPUCount=12*, *in_node_count=None*, *verbose=True*`)`
Multiprocessing over outcoords.

comet.pyhed.misc.poly_tools module

Part of comet/pyhed/misc

`comet.pyhed.misc.poly_tools.`**`cleanUpTetgenFiles`**`(`*basename*`)`
Removes temporary tetgen files.

Parameters **basename** (*string*) – File path. All files with that basename and one of the following endings will be removed ('.poly', '.ele', '.node' '.face' '.edge').

comet.pyhed.misc.poly_tools.**createPolyBoxWithHalfspace**(*minx, maxx, miny, maxy, minz, maxz, halfspace_at=0.0, without_halfspace=False, interface_marker=None*)

Creates a simple poly file for further mesh build processes.

Imports pygimli.

Parameters

- **minx** (*float*) – Minimum x dimension.
- **maxx** (*float*) – Maximum x dimension.
- **miny** (*float*) – Minimum y dimension.
- **maxy** (*float*) – Maximum y dimension.
- **minz** (*float*) – Minimum z dimension.
- **maxz** (*float*) – Maximum z dimension.
- **halfspace_at** (*float*) – Z value where the halfspace is considered. Additionally to the corner points of the simple halfspace box, a separation of the z edges will be at *halfspace_at*.
- **without_halfspace** (*booleam [False]*) – An face that closes the 4 edge points at *halfspace_at* is inserted. This can be ommitted if creating but a tetrahedron boundary around another polygon.
- **interface_marker** (*integer [None]*) – Optional marker for the interface face, for later identification.

Returns Closed polygon mesh with or without face at the halfspace interface. Note that the first four nodes in the polygon correspond to the four edge nodes at the halfspace interface, for manual connection to other polygons.

Return type pg.Mesh

comet.pyhed.misc.test_class module

Part of comet/pyhed/misc

class comet.pyhed.misc.test_class.**BaseTest**(*name*)

Bases: object

test()

testing_function()

comet.pyhed.misc.timer module

Part of comet/pyhed/misc

class comet.pyhed.misc.timer.**NoneTimer**(*verbose=True*)

Bases: object

exportLog(*savename*)

getMessage(*msg*)

importLog(*savename*)

noHist(*msg*)

printHistory()

setTimeFactor(*factor*)

setVerbose(*verbose*)

silent(*msg*)

tick(*msg*, ***kwargs*)

update()

class comet.pyhed.misc.timer.**Timer**(*verbose=True*, *timestamps=True*, *timefactor=1.0*)
Bases: object

exportLog(*savename*)

getMessage(*msg*, *ts=None*)

importLog(*savename*)

noHist(*msg*, *update=True*, *ts=None*)

printHistory()

setTimeFactor(*factor*)

setTimestamps(*bool_timestamps*, *strftime='%Y-%m-%d %H:%M:%S'*)

setVerbose(*verbose*)

silent(*msg*, *update=True*, *ts=None*)

tick(*msg*, *update=True*, *ts=None*, ***printkwargs*)

time_last

time_total

update()

comet.pyhed.misc.toolbox module

Part of comet/pyhed/misc

exception comet.pyhed.misc.toolbox.**NamespaceError**(*value*)
Bases: Exception

Named Error for try except clauses.

comet.pyhed.misc.toolbox.**convertCoordinates**(*gimli*, *dolfin*)
Find sorting between two coordinate arrays if same points. input: arr1, arr2: two coordinate lists of same shape (n, 3) which contains the same coordinates but in a diffrent order. output: arr1_arr2, arr2_arr1: index arrays which converts coordinates from input1 to input2 and from input2 to input1.

comet.pyhed.misc.toolbox.**floatString**(*value*, *frmt='2.2f'*, *replace='_'*)
Converts a Float to a string for filenames etc.

comet.pyhed.misc.toolbox.**getAllValuesByReference**(*mat*, *refarray*)
Gets all values from input hdf5 data set found in given reference array of the same dataset.

```
>>> import h5py
>>> from comet import pyhed as ph
>>> mat = h5py.File('input.mrsd')
>>> # get pulse moments from mrsd file
>>> pulse_mat = mat['proclog']['Q']['q']
>>> pulses = ph.misc.getAllValuesByReference(mat, pulse_mat)
array([ 0.11261871,  0.15802349,  0.1729516 ,  0.24440305,  0.27615926,
...     0.39153588,  0.45558559,  0.64535046,  0.77051771,  1.08620425,
...     1.32817318,  1.85991744,  2.32437798,  3.22896476,  4.11364457,
...     5.66420968,  7.33714431, 10.01091275, 13.18643762, 17.83750801])
```

`comet.pyhed.misc.toolbox.`**`insert`**(*array1*, *array2*, *breaking_point_float=0*, *right=True*)
Utility function to insert points between two arrays. Depricated.

`comet.pyhed.misc.toolbox.`**`plt_ioff`**()
Temporal overrides the interactive mode of matplotlib.

`comet.pyhed.misc.toolbox.`**`plt_ion`**()
Temporal overrides the interactive mode of matplotlib.

`comet.pyhed.misc.toolbox.`**`printv`**(*string*, **args*)
for maintenance and debugging

`comet.pyhed.misc.toolbox.`**`progressBar`**(*it*, *prefix=''*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Iterable progress bar. Usage

exchange:

```
>>> for i in range(12):
```

with:

```
>>> for i in progressBar(range(12), 'some describing string: '):
```

`comet.pyhed.misc.toolbox.`**`project1DModel`**(*thk*, *para*, *out*)
projects a simple synthetic layered model to a given discretisation. The discretisation "disOut" has to be a vector for the corners of the desired discretisation. Therefore the output will be a vector with len(outDis) - 1 elements. "parameter" must have one entry more than thickness.

```
>>> thk = [2.375]
>>> resistivity = [100, 5]
>>> disOut = np.linspace(0, -4.5, 10)
>>> model = project1DModel(thk, resistivity, disOut)
>>> print(model)  # 100 * 0.75 + 5 * 0.25 = 76.25
[ 100.   100.   100.   100.    76.25    5.     5.     5.     5. ]
```

`comet.pyhed.misc.toolbox.`**`refine`**(*array*, *start='0'*, *end='-1'*, *insert=1*, *log=False*, *zerovalue=False*, *invert=False*)
Utility function to refine array. Depricated.

`comet.pyhed.misc.toolbox.`**`setNearestMarkers`**(*outmesh*, *inmesh*, *y_lim*, *marker_air=0*, *marker_half=1*, *fill_air_ground=False*, *air_interface=0.0*)
Set marker from 2d mesh + limits in y to 3d mesh. Returns list with ommitted marker or empty list. Optinally fills air and groundspace outside the 2d mesh with marker.

`comet.pyhed.misc.toolbox.`**`setNearestMarkers_old`**(*outmesh*, *inmesh*, *marker_air=0*, *marker_half=1*, *fill_air_ground=False*)
Set marker from one mesh to another. Returns list with ommitted marker.

`comet.pyhed.misc.toolbox.`**`setdebugging`**(*Bool*, *local=True*)

Temporal function used to control debug mode. Do not use.

`comet.pyhed.misc.toolbox.`**`temporal_printoptions`**(*threshold=5*, ***kwargs*)

Temporal overrides the printoptions of numpy arrays.

comet.pyhed.misc.vec module

Part of comet/pyhed/misc

`comet.pyhed.misc.vec.`**`Ca2Cy`**(*cartesian*, *dtype=<sphinx.ext.autodoc.importer._MockObject object>*, *drop_tol=0.01*, *dipole_z=0.0*)

Convert cartesian coords to cylindrical coords.

Parameters

- **cartesian** (*np.ndarray*) – Coordinate vector of for N positions of shape (3, N).
- **dtype** (*np.dtype or str*) – Optional choice of output data type.
- **drop_tol** – Tolerance in m to avoid zeros in horizontal distances (singlularity removal). All values for the resulting horizontal distance below the drop_tol are redistributed between drop_tol and 20% of the distance to the first point outside droptol. Raises Exception if no point lies outside of drop_tol. drop_tol=None disables the singularity removal (default).

Returns **cylindrical** – Coordinate Vector in cylindrical cordinates (radius, phi, z) in given datra type or input data type and with or without singularities removed.

Return type np.ndarray

`comet.pyhed.misc.vec.`**`Ca2CyField`**(*cartesian*, *field*, *dtype=None*)

Conversion of 3d vector field from cylindrical coords in cartesian.

cartesian = cartesian coordinate system (x, y, z) # x = field[x] * cos(phi) - field[y] * sin(phi) # y = - field[x] * sin(Phi) + field[y] * cos(phi) # z = field[z] # output: # field_cylindrical = data in cylindrical coordinates (r, phi, z)

Parameters

- **cartesian** (*np.ndarray*) – Coordinate vector of for N positions of shape (3, N).
- **field** (*np.ndarray*) – Field vector of for N positions of shape (3, N).
- **dtype** (*np.dtype or str*) – Optional choice of output data type.

Returns **field** – Field vector in cylindrical cordinates (x, y, z) in given datra type or input data type.

Return type np.ndarray

`comet.pyhed.misc.vec.`**`Cy2Ca`**(*cylindrical*, *dtype=None*)

Convert cartesian coords to cylindrical coords.

Parameters

- **cylindrical** (*np.ndarray*) – Coordinate vector of for N positions of shape (3, N).
- **dtype** (*np.dtype or str*) – Optional choice of output data type.

Returns **cartesian** – Coordinate Vector in cartesian cordinates (x, y, z) in given datra type or input data type.

Return type np.ndarray

`comet.pyhed.misc.vec.`**`Cy2CaField`**(*cylindrical*, *field*, *dtype=None*)

Conversion of 3d vector field from cartesian coords in cylindrical.

# polar = polar coordinate system # x = field[r] * cos(phi) - field[phi] * sin(phi) # y = field[r] * sin(phi) + field[phi] * cos(phi) # z = field[z] # output: # field_cartesian = data in zylindrical coordinates

Parameters

- **cylindrical** (*np.ndarray*) – Coordinate vector of for N positions of shape (3, N).
- **field** (*np.ndarray*) – Field vector of for N positions of shape (3, N).
- **dtype** (*np.dtype or str*) – Optional choice of output data type.

Returns **cartesian field** – Field vector in cartesian cordinates (x, y, z) in given datra type or input data type.

Return type np.ndarray

`comet.pyhed.misc.vec.`**`GridtoVector`**(**args*, ***kwargs*)

# transform the matlab grid to a python vector with the correct shape # can take vector field data with x, y, z coordinates or simple one # dimansional vectors # 2D is not implemented yet

Parameters

- **order** (*[‘F’]*) – order = ‘F’ -> Fortran style = x varies fastest, instead of z
- **comp** (*[3]*) – number of components
- **# status** (*implemented*)

`comet.pyhed.misc.vec.`**`KtoP`**(*cartesian*, *dtype=<sphinx.ext.autodoc.importer._MockObject object>*, *drop_tol=0.01*)

# status: depricated, use Ca2Cy instead

`comet.pyhed.misc.vec.`**`KtoP_all`**(*cartesian*, *dtype=<sphinx.ext.autodoc.importer._MockObject object>*, *drop_tol=0.01*)

# r = np.sqrt(model[0]**2 + model[1]**2) # phi = np.arctan2(model[1], model[0]) # z = np.copy(model[2])

`comet.pyhed.misc.vec.`**`KtoP_field`**(*cartesian*, *field*, *dtype=<sphinx.ext.autodoc.importer._MockObject object>*)

# status: depricated, please use Ca2CyField instead.

`comet.pyhed.misc.vec.`**`PtoK`**(*cylindrical*, *dtype=<sphinx.ext.autodoc.importer._MockObject object>*)

# status: depricated use Cy2Ca instead

`comet.pyhed.misc.vec.`**`PtoK_field`**(*cylindrical*, *field*, *dtype=<sphinx.ext.autodoc.importer._MockObject object>*)

# status: depricated, please use Cy2CaField instead.

`comet.pyhed.misc.vec.`**`R3VtoNumpy`**(*R3Vector*, ***kwargs*)

Creates a numpy vector from a pygimli R3Vector.

`comet.pyhed.misc.vec.`**`VectortoGrid`**(*vector*, *shape*, *order=’F’*, *swap=False*)

see ‘GridtoVector’ x == VectortoGrid(GridtoVector(x), x.shape) is True

# status: implemented

`comet.pyhed.misc.vec.`**`angle`**(*ax1*, *ax2*)

Returns angle between two arbitrary vectors of shape (3, . . .). Allows broadcasting.

`comet.pyhed.misc.vec.`**`areaFromPolyPoints`**(*points*)

Get perimeter of a polygon.

`comet.pyhed.misc.vec.`**`convertCRStoMap`**(*rowIdx*, *colPtr*)
Converts CRS indices to map indices.

`comet.pyhed.misc.vec.`**`cumsumDepth`**(*a*, *min_thk=0.5*)
Summs part of a array, until all layers have a given minimum thickness. only use on array with increasing thickness.

`comet.pyhed.misc.vec.`**`fillCRS`**(*crsMat*, *rowIdx*, *colPtr*, *vals*)
Fill CRS format SparseMatrix with values. Very Slow.

`comet.pyhed.misc.vec.`**`fixSingularity`**(*model*, *drop_tol=0.01*, *dipole_z=0.0*)
Points in zero get values of drop_tol. Points on drop_tol get Values of up to 20% the value to the first point out of the drop_tol. If all points are in the drop_tol a warning is printed.

`comet.pyhed.misc.vec.`**`getConstraints`**(*inv*)

`comet.pyhed.misc.vec.`**`getIndicesFromConstraintMatrix`**(*mat*)

`comet.pyhed.misc.vec.`**`getRSparseValues`**(*sparseMapMatrix*, *indices=True*, *getInCRS=False*)
Get CRS Arrays (Row Index, Column Start_End, Values) from SparseMatrix (CRS format).

`comet.pyhed.misc.vec.`**`interpolateField`**(*Mesh*, *positions*, *Field*, *interpolationMatrix=None*, *verbose=False*)
simple case: (meshInput, meshOutput, fieldFromInputMesh) ready

# status: implemented

`comet.pyhed.misc.vec.`**`interpolateField_Matrix`**(*Field*, *InterpolationMatrix*, *verbose=False*)
# status: implemented

`comet.pyhed.misc.vec.`**`interpolateField_rotatedMatrix`**(*Field*, *base_mat=None*, *sin_mat=None*, *cos_mat=None*)
# status: in testing

`comet.pyhed.misc.vec.`**`interpolateVector`**(*Mesh*, *Slice*, *Vector*, *verbose=False*)
Interpolates a given vectorfield(Vector) based on the given Mesh to a second mesh or slice. The field can either be real or complex.

# status: implemented

`comet.pyhed.misc.vec.`**`linspace2D`**(*Point1*, *Point2*, *num*)
Internal function. Like linspace but for twodimensional points.

`comet.pyhed.misc.vec.`**`linspace3D`**(*Point1*, *Point2*, *num*)
Internal function. Like linspace but for threedimensional points.

`comet.pyhed.misc.vec.`**`perimeterFromPolyPoints`**(*points*, *circle_radius=None*, *closed=True*)
Get perimeter of a polygon.

`comet.pyhed.misc.vec.`**`pointDataToCellData_np`**(*mesh*, *field*, *mixed=False*, *weight=True*)
Interpolates vector- or skalarfield data defined on the nodes of the given mesh to its cell midpoints. For now it has to be either a uniform mesh with field (3d, complex or real) or scalar (1d, complex or real) datasets or a mixed mesh with a simple real scalar data set (takes more time).

Parameters

- **mesh** (*pg.Mesh*) – For now the algorithm takes only pygimli meshes.
- **field** (*array of shape (n) or (n, 3) or pg.Vector*) – Data set with n = number of nodes in the mesh.
- **mixed** (*bool [False]*) – Flag to determine if the mesh is either of mixed (True) shape (cells can consist of variable number of nodes) or uniform (False).

- **weight** (*bool [True]*) – The cell data can be calculated as simple average of the surrounding node values (False) or additionally weighted by their inverse distance from the nodes (True).
- **Output**
- **——**
- **newfield** (*array of shape (c) or (3, c)*) – Data set with c = number of cells in the mesh.

`comet.pyhed.misc.vec.`**`regular_slice`**(*dim1*, *dim2*, *direction*, *value*)

out: regular pygimli 2D-mesh object with given discretisation and orientation with respect to a 3D coordinate system.

for now there are only x-, y- and z-orientated slices possible

# status: implemented

`comet.pyhed.misc.vec.`**`regular_sliceFrom3DMesh`**(*mesh*, *discretisation1*, *discretisation2*, *direction*, *value*)

Cut a slice with regular grid discretisation from an arbitrary shaped irregular mesh. Used for plotting purposes with matplotlib.

input: mesh x discretisation y discreatisation normal direction of the slice value for position of the slice on the normal axis

# status: implemented

`comet.pyhed.misc.vec.`**`rotFromAtoB`**(*vec*, *ax1*, *ax2*)

Rotates input vector **vec** from one direction **ax1** (x, y, z) to another direction **ax2** (x, y, z).

`comet.pyhed.misc.vec.`**`rotate3`**(*Vec*, *alpha*, *axis='z'*, *copy=False*)

Rotates 3 dimensional arrays around a given axis with angle alpha.

`comet.pyhed.misc.vec.`**`rotate3_all`**(*Vec*, *alpha*, *axis='z'*, *copy=False*)

'ijk,k…i->k…j' … = number of points, broadcasted dimension i = 3 (3 coords per point) j = 3 (3 coords per point) k = number of different dipoles, number of alphas

`comet.pyhed.misc.vec.`**`rotationMatrix`**(*axis*, *theta*)

Return the rotation matrix associated with counterclockwise rotation about the given axis (3, n) by theta radians (n,). Supports broadcasting along second axis of input **axis**.

Array with rotation matrices of shape (3, 3, n) or (3, 3) if n==1.

`comet.pyhed.misc.vec.`**`sinhZVolumeFunction`**(*z, z_range=[0, -100], area_range=[0.1, 100]*)

Maps values from z per z_range to area_range using a sinh function instead of linear interpolation. Values outside z_range are assigned the limits of area_range.

Example

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> area = []
>>> zrange = np.linspace(30, -130, 160)
>>> for z in zrange:
>>>     area.append(sinhZVolumeFunction([z]))
>>> fig, ax = plt.subplots(1, 1)
>>> ax.plot(zrange, area)
```

`comet.pyhed.misc.vec.`**`sinhspace`**(*start*, *stop*, *num_step*)

Like linspace but using a hyperbolic sine function.

`comet.pyhed.misc.vec.`**`sumBetweenIndices`**(*array*, *indices*, *use_thickness=False*, *axis=0*)
Split array at given axis and indices and sum up the parts. If use_thickness is True, the difference of the absolute values are used, usfull for array representing a depth for example.

`comet.pyhed.misc.vec.`**`translate`**(*Vec*, *x*, *y*, *z=0*, *copy=False*)
Translate a given three dimensional vector and returns either a view of it or a new object.

# status: implemented

`comet.pyhed.misc.vec.`**`translateToDipole`**(*Vec*, *Dipole*, *copy=False*)

`comet.pyhed.misc.vec.`**`translate_all`**(*Vec*, *coords*)
Vec: Dim: num_points, 3 coords: Dim = num_dipoles, 3

output: Dim: num_dipoles, num_points, 3

`comet.pyhed.misc.vec.`**`uniqueAndSum`**(*indices*, *to_sum*, *return_index=False*, *verbose=False*)
Summs double values found by indices in a various number of arrays.

Returns the sorted unique elements of a column_stacked array of indices. Another column_stacked array is returned with values at the unique indices, while values at double indices are properly summed.

Parameters

- **ar** (*array_like*) – Input array. This will be flattened if it is not already 1-D.
- **to_sum** (*array_like*) – Input array to be summed over axis 0. Other exsisting axes will be broadcasted remain untouched.
- **return_index** (*bool, optional*) – If True, also return the indices of *ar* (along the specified axis, if provided, or in the flattened array) that result in the unique array.

Returns

- **unique** (*ndarray*) – The sorted unique values.
- **summed_array** (*ndarray*) – The summed array, whereas all values for a specific index is the sum over all corresponding nonunique values.
- **unique_indices** (*ndarray, optional*) – The indices of the first occurrences of the unique values in the original array. Only provided if *return_index* is True.

Example

```
>>> import numpy as np
>>> from comet.pyhed.misc.vec import uniqueAndSum
>>> idx1 = np.array([0, 0, 1, 1, 2, 2])
>>> idx2 = np.array([0, 0, 1, 2, 3, 3])
>>> # indices at positions 0 and 1 and at positions 5 and 6 are not unique
>>> to_sort = np.column_stack((idx1, idx2))
>>> # its possible to stack more than two array
>>> # you need for example 3 array to find unique node positions in a mesh
>>> values = np.arange(0.1, 0.7, 0.1)
>>> print(values)
[ 0.1  0.2  0.3  0.4  0.5  0.6]
>>> # some values to be summed together (for example attributes of nodes)
>>> unique_idx, summed_vals = uniqueAndSum(to_sort, values)
>>> print(unique_idx)
[[0 0]
 [1 1]
 [1 2]
```

(continues on next page)

(continued from previous page)

```
 [2 3]]
>>> print(summed_vals)
[ 0.3  0.3  0.4  1.1]
>>> # [0.1 + 0.2, 03., 0.4, 0.5 + 0.6]
```

Module contents

Module comet/pyhed/misc

comet.pyhed.plot package

Submodules

comet.pyhed.plot.plotHankel module

Part of comet/pyhed/plot

`comet.pyhed.plot.plotHankel.`**`plotHankel`**`(`*order*`)`

`comet.pyhed.plot.plotHankel.`**`plotKey`**`(`*order*`)`

comet.pyhed.plot.plot_bib module

Part of comet/pyhed/plot

`comet.pyhed.plot.plot_bib.`**`addPatch`**`(`*ax*, *cbar_ax=None*, *offset_left=28.5*, *offset_top=1*, *distance=1*, *color='lightgray'*, *lw=0*, *z_order=0*`)`

`comet.pyhed.plot.plot_bib.`**`amp`**`(`*field*`)`
Amplitude of a complex field. Internally used.

`comet.pyhed.plot.plot_bib.`**`cmap_phase`**`()`

`comet.pyhed.plot.plot_bib.`**`drawCWeight`**`(`*ax*, *mesh*, *cweight*, *lmin=0*, *lmax=0.8*, *cmin=0.2*, *cmax=1*, *min_plot=0.02*, *color='black'*, *cell_indices=None*`)`
Draws the given cweights defined for given mesh on given ax.

Parameters

- **ax** (*plt.ax*) – Ax to plot constraint weights in.
- **mesh** (*pg.Mesh*) – Mesh object where the constraints are defined in.
- **cweight** (*np.ndarray*) – Constraint values to be plotted.
- **lmin** (*float [0]*) – Minimum linewidth for maximum cweight defined via **cmax**. Note that by default high constraint values are plotted with thinner lines.
- **lmin** (*float [0.8]*) – Maximum linewidth for minimum cweight defined via **cmin**.
- **cmin** (*float [0]*) – Minimum constraint weight to plot. All values smaller than cmin are plotted with the same linewidth as cmin.
- **cmax** (*float [1]*) – Maximum constraint weight to plot. All values greater than cmax are plotted with the same linewidth as cmax.

- **min_plot** (*float [0.02]*) – Minimum linewidth to plot to avoid large pdfs.
- **color** (*string ['black']*) – Color of lines.
- **cell_indices [None]**
- **f(cweight) -> linewidth** – (cmin, cmax) -> (lmax, lmin) if cmin < cweight < cmax
- **Returns**
- **———**
- **None**

comet.pyhed.plot.plot_bib.**drawFid**(*ax*, *fid*, *clim=None*, *clab=None*, *draw='data'*, *to_plot='real'*, *cmap=None*, *cbar=True*, *gated=True*, *title=None*)

Plot any data (or response, error, misfit) cube nicely.

if response is True: response vector from fid taken and used for plotting misfit.

comet.pyhed.plot.plot_bib.**drawMeshLines**(*ax*, *mesh*, *color='white'*, *linewidth=0.5*, *marker=None*, ***kwargs*)

Draw all mesh boundaries in given ax.

comet.pyhed.plot.plot_bib.**getCMapAndLim**(*toplot*, *phase=False*, *misfit=False*, *perc=0.99*, *minimum=False*, *lut=None*)

Chooses colorbar limits and appropriate colobar based on input. lut: If lut is not None it must be an integer giving the number of entries desired in the lookup table, and name must be a standard mpl colormap name.

comet.pyhed.plot.plot_bib.**loadPickledFig**(*savename*)

comet.pyhed.plot.plot_bib.**markCbar**(*cbar*, *pos*, *text=None*, *color='white'*, *linewidth=0.5*, *size=None*, *text_y_pos=1.35*, *cbar_horizontal=True*, ***kwargs*)

Marks a given colorbar of a plot at a specific position and optionally displaysa describing text. Useful to remind on a synthetic background or focus the view on a specific range.

Parameters

- **cbar** (*matplotlib colorbar*) – Colorbar to mark.
- **pos** (*float*) – Marker position in values of the colorbar (data values).
- **text** (*string, optional*) – Text to display. The default is None.
- **color** (*string, optional*) – Color used for the marker. The string is redirected to matplotlib. The default is 'white'.
- **linewidth** (*float, optional*) – Thickness of the marker line. The default is 0.5.
- **size** (*integer, optional*) – Size of the Text. If None the size of the ticklabel of the cbar axis is used. If no label is found the size is set to 9. The default is None.
- **text_y_pos** (*flat, optional*) – Vertical Offset for the displayed text. (or horizontal offset for vertical colorbars, see next argument). The default is 1.35.
- **cbar_horizontal** (*boolean, optional*) – Flag for a horizontal colorbar. The default is True.
- ****kwargs** (*dictionary*) – Redirected to the text function. Filled with default values for 'horizontalalignment' ('center') and 'verticalalignment' ('center').

Returns

Return type None.

comet.pyhed.plot.plot_bib.**pickleFig**(*savename*, *fig*) `comet.pyhed.plot.plot_bib.`**`printv`**(*string*, **args*)

print function for maintenance and debugging

`comet.pyhed.plot.plot_bib.`**`quantile`**(*data*, *perc=0.99*, *add_rel=0.1*, *add_abs=0.0*)

Returns the the data point that lies over a given percentage (50 %) of a data set and returns value (+ 10%).

Parameters

- **data** – Dataset for which the value is to be searched.
- **perc** (*float [0.5]*) – Percentage [0. . . 1] defining he search for a parameter. Searches for the value in the dataset that lies above *perc* of the other data.
- **add_rel** (*float [0.1]*) – Relative value added to the result of the search.
- **add_abs** (*float [0.0]*) – Absolute value added to the result of the search.
- **For very small values in data, add_rel should be replaced by add_abs (and**
- **an appropriate value) or set 0.**
- **For add_rel = 0.0 and add_abs = 0.0, the returned value will always be part**
- **of the given dataset.**
- **Hint for colorbars** (*usually a median(data, perc=. . .) with perc = 0.95 for*)
- **small data sets and perc = 0.99 for large data sets will result in nice**
- **colorbar settings as it effectively removes spikes.**

`comet.pyhed.plot.plot_bib.`**`returnFigureAndAx`**(*ax*, **args*, ***kwargs*)

Returns figure and ax of given ax or creates subplots. Not used inside pyhed.

`comet.pyhed.plot.plot_bib.`**`setAxSize`**(*ax*, *size*)

`comet.pyhed.plot.plot_bib.`**`setOuterLabelOnly`**(*ax*, *xlabel='X (m)'*, *ylabel='Z (m)'*)

Removes all ticks from the given axes and labels exept the outer left and lower axes which are labeled using the the given labels. This is a convenience function for multi ax plots, where the subplots have the same outer dimension.

`comet.pyhed.plot.plot_bib.`**`showEtraData`**(*survey*, *to_plot='real'*, *draw='data'*, *savename='auto'*, *rdir='.'*, *praefix=''*, *size=12*, *patch=True*, *clim=None*, *perc=0.995*, *cmap='auto'*, *pdf=True*, *png=False*)

Plot function to create and save data and misfit plots of etra data.

Parameters

- **datas** (*array_like*) – Array containing the measured data in nV. Expect one dimensional array of concatenated datas. First dimension defines the different recievers. If array is real, expect first half to contain the real component and second half to contain the imaginary data.
- **gates** (*array_like*) – Midpoints of the used time gates for plotting in s.
- **pulses** (*array_like*) – Used pulse moments for plotting in As.
- **errors** (*array_like [None]*) – Assumed errors of the datas for plotting of misfit. Same shape as data.
- **draw** (*string ['data']*) – This function can plot 'data', 'response' or 'misfit'. The last two only if fid are eqipped with proper response vector. See setResponse() of Survey class or setResponse() of Fid class for information about setting response vectors.
- **to_plot** (*string ['real']*) – Decides weather real or imaginary part of the data is plotted. Alternatively 'abs' can be used to plot absolute values.

- **savename** (*string ['auto']*) – If on auto, the savename is generated out of the other given parameters. If other than 'auto', the given savename is used to save the resulting figures. If on 'auto', see *rdir* and *praefix* for additional information.
- **rdir** (*string ['.']*) – If *savename*=='auto', rdir defines the directory the results are saved in. This is ignored if savename is not 'auto'.
- **praefix** (*string ['']*) – If *savename*=='auto', praefix can be used to distinguish different data sets in the same *rdir*. This is ignored if savename is not 'auto'.
- **size** (*integer [17]*) – Fontsize for the exported figure ticks and labels.
- **patch** (*boolean [True]*) – As the coincident measurement and the other seven get different colorbars (see *clim*), a grey patch is optionally used as background for the first data plot. This switch can be used to omit this patch.
- **clim** (*list or list of lists [None]*) – The colorbar limits of the plots can be fixed. Except a list of min and maximum value for misfit and two of those lists for the data plot, whereas the first min and max is used fot the coincident measurement, and the second for the other seven measurements.
- **perc** (*float [0.999]*) – Percentage to autodefine the colorbar values. The defaults sets the maximum value to the value that is greater than 99.9 % of the data.

comet.pyhed.plot.plot_bib.**showLoop**(*pos*, *phi*, *ds*, *referenzpunkte=None*, *ax=None*, *color=None*, ***kwargs*)

Plots a loop as set of dipoles on a given axis. Used by *show* method of loop class.

comet.pyhed.plot.plot_bib.**showLoopLayout**(**loops*, *ax=None*, ***kwargs*)

Shows multiple loops at once on a given axis.

Module contents

Module comet/pyhed/plot

Submodules

comet.pyhed.config module

Part of comet/pyhed

class comet.pyhed.config.**SecondaryConfig**(*name=None*, *mod_name=None*, *mesh_name=None*, *m_dir='.'*, *r_dir='.'*, *pf_name='__default__prim_fields'*, *p2=False*, *approach='E_s'*, *pf_EH_flag='E'*, *sigma_ground=[0.001]*, *procs_per_proc=1*, *frequency=2000*)

Bases: object

load(*filename*)

Load config from file.

Parameters **filename** (*sting*) – Relative or absolute path to file.

None

save(*filename*)

Save secondary config in ASCII file format.

Parameters **Filename** (*string*) – Filename for saving. Sub directories are created on the fly if code execution has the proper rights.

setAnomalies(*sigma_anom*, *layer_markers=None*)
Set anomalie vector.

Parameters

- **sigma_anom** (*np.ndarray*) – Array with sigma values for each cell marked as anomaly.
- **layer_markers** (*np.ndarray [None]*) – Array containing the cell marker for each anomaly value (cell) to calculate the sigma anomalies with respect to the 1d background model. None indicates a homogenous background and all marker are set to 1 (0 is airspace).

class comet.pyhed.config.**config**(*name=None*, *rho=[1000.0]*, *d=[]*, *f=2000.0*, *mode='te'*, *ftype='B'*, *current=1.0*, *forceNew=False*)
Bases: object

Basic 1d configuration file.

Contents the values for layer parameters rho and d, the mode (te or tm), the fieldtype that will be calculated and the current of the loop.

Held by instances of the loop class.

load(*name*)
Load config from ASCII file format.

save(*name*)
Saves config in ASCII file format.

Module contents

Module comet/pyhed

class comet.pyhed.**config**(*name=None*, *rho=[1000.0]*, *d=[]*, *f=2000.0*, *mode='te'*, *ftype='B'*, *current=1.0*, *forceNew=False*)
Bases: object

Basic 1d configuration file.

Contents the values for layer parameters rho and d, the mode (te or tm), the fieldtype that will be calculated and the current of the loop.

Held by instances of the loop class.

load(*name*)
Load config from ASCII file format.

save(*name*)
Saves config in ASCII file format.

class comet.pyhed.**SecondaryConfig**(*name=None*, *mod_name=None*, *mesh_name=None*, *m_dir='.'*, *r_dir='.'*, *pf_name='__default__prim_fields'*, *p2=False*, *approach='E_s'*, *pf_EH_flag='E'*, *sigma_ground=[0.001]*, *procs_per_proc=1*, *frequency=2000*)
Bases: object

load(*filename*)
Load config from file.

Parameters **filename** (*sting*) – Relative or absolute path to file.

None

save (*filename*)

Save secondary config in ASCII file format.

Parameters **Filename** (*string*) – Filename for saving. Sub directories are created on the fly if code execution has the proper rights.

setAnomalies (*sigma_anom*, *layer_markers=None*)

Set anomalie vector.

Parameters

- **sigma_anom** (*np.ndarray*) – Array with sigma values for each cell marked as anomaly.
- **layer_markers** (*np.ndarray [None]*) – Array containing the cell marker for each anomaly value (cell) to calculate the sigma anomalies with respect to the 1d background model. None indicates a homogenous background and all marker are set to 1 (0 is airspace).

`comet.pyhed.`**addLogFile** (*name=None*, *new_log=True*)

comet.snmr package

Subpackages

comet.snmr.kernel package

Submodules

comet.snmr.kernel.kernel_bib module

Part of comet/snmr/kernel

This file contents parts of the MRSmatlab Kernel function part

class `comet.snmr.kernel.kernel_bib.`**Kernel** (*survey=None*, *fid=0*, *dimension=1*, *name=None*)

Bases: `object`

Basic class to solve the NMR kernel computation.

Parameters

- **name** (*string [None]*) – If kernel is loaded from file.
- **survey** (*survey class instance [None]*) – Calls *setSurvey* to define underlaying survey class. Holds important attributes like pulse moments and the loops for tx and rx.
- **tx** (*integer [0]*) – Transmitter index in corresponding survey.
- **rx** (*integer [0]*) – Receiver index in corresponding survey.
- **fid** (*interger [0]*) – Sounding index in corresponding survey.
- **dimension** (*integer [1]*) – Defines the kernel integration.

Example

```
>>> from comet.snmr import kernel as k
>>> from comet.snmr import survey
>>> site = survey.Survey()
>>> kernel = k.kernel(site)
>>> kernel.calculate()
>>> kernel.save('savename')
>>> kernel.show()
```

`BFieldCalculation`(*loop_mesh=None*, *dipole_mesh=None*, *interpolate=False*, *just_loop_fields=False*, *recalc_loop_fields=False*, *recalc_primary=False*, *num_cpu=12*, ***kwargs*)

Calculates the Bfield for the kernel function for tx and rx.

internal call of *loop.calculate()* including decision if cell based or node based Bfield is needed. All optional parameters are piped to the *loop.calculate()* call. Based on the desired dimension of the kernel a specialised mesh may be automatically generated for the calculation.

Part 1/3 of the kernel calculation. Called automatically if *kernel.calculate* is called.

`calcInterpolationMatrix`()

`calcMagnetization`()

Creates 3D mesh and calcualtes magnetization vector after excitation. Returns magnetization vector of shape (num_pulses, num_cells_3d, 3)

`calculate`(*loop_mesh=None*, *dipole_mesh=None*, *interpolate=False*, *savename=None*, *forceNew=False*, *slices=True*, *slice_name=None*, ***kwargs*)

All three parts of the kernel calulation are called here.

All given kwargs are directed to BfieldCalculation(), see function info for details about possible keyword arguments.

```
>>> self.BFieldCalculation(**kwargs)
```

```
>>> self.ellipticalDecomposition()
```

```
>>> self.kernelIntegration()
```

```
>>> if savename is not None:
        self.save(savename)
```

Keyword Arguments `destinations` – none for now, with exception of "num_cpu", [12] which is directed to BfieldCalculation and/or sliceKernel

`coincident`

`create1DInterpolationSlices`()

`create1DKernelMesh`(*max_length=0.1*, *area=100.0*, *quality=32*, *zvec=None*, *size_factor=2.5*, *z_factor=2.5*, *export_xyplane=None*, *max_dipoles=2000*, *calc_3D_stats=True*, *xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*)

In order to integrate the kernel to a 1D structure without interpolation errors, a special mesh consisting of triangular zylinders has to be defined.

Parameters

- **max_length** (*float [0.1]*) – Defines the smallest edge length for the discretisation of the loop . In order to get admirable kernel results a value of 0.1 meters should be the maximum.
- **area** (*float [100.]*) – Defines the maximum Area a triangle in the loop slice can have.
- **quality** (*float [32.]*) – Defines the smallest angle inside a triangle. Be careful with values above 35.
- **zvec** (*array_like [None]*) – Usualy the zvec is defined automatically, this flag gives the user the optional possibility to give a zvec from outside the funktion.
- **size_factor** (*float [2.5]*) – Extension of the kernel mesh (and therefore integration volume) in the x and y direction. Should be at least 2 times the loop diameter or shortest edge length. This value defines the multipier.
- **z_factor** (*float [2.5]*) – Maximum depth of the Kernel. Should be at least 2 times the loop diameter or shortest edge length. This value defines the multipier.
- **export_xyplane** (*string [None]*) – Filename for the resulting kernel mesh plane in 2D can be exported for debugging or simply to check the mesh (vtk).
- **max_dipoles** (*interger [2000]*) – Fallback for high node density loops. This sets an overall maximum for the number of dipoles used for the loop discretization. However this only comes into account in rare cases.

`create2DInterpolationSlices``()`

`create2DKernelMesh``(`*area=15.0*, *quality=34*, *yvec=None*, *x_factor=5*, *z_factor=2*, *savename=None*, *export_xzplane=None*, *calc_3D_stats=True*, *order=0*`)`

Similary to the mesh in the 1D case a special mesh consisting of triangluar zylinders is generated. The Zylinders are pointing in the y direction to allow a perfect integration to the x-z plane.

Parameters

- **area** (*float [15.]*) – Affects the maximum area a triangle in the 2D slice is allowed to have. Higher Values lead to bigger cells.
- **quality** (*float [34]*) – Defines the smallest angle inside a triangle. Be careful with values above 34.5. Higher values = more cells.
- **yvec** (*ndarray, list [None]*) – Usualy the y vector is defined automatically, this flag gives the user the optional possibility to give a YVec from outside the function.
- **x_factor** (*float [2]*) – Extension of the kernel mesh (and therefore integration volume) in the x direction. Should be at least 2 times the loop diameter or shortest edge length. This value defines the multipier.
- **z_factor** (*float [2]*) – Extension of the kernel mesh (and therefore integration volume) in the z direction. Should be at least 2 times the loop diameter or shortest edge length. This value defines the multipier.
- **savename** (*string [None]*) – If a savename is given, the resulting 2D Mesh is saved in the .bms format for later use.
- **export_xyplane** (*string [None]*) – Filename for the resulting kernel mesh plane in 2D can be exported for debugging or simply to check the mesh (vtk).

`createMagnetizationMesh``()`

Creates full 3D mesh for display and calcualtion of magnetization vectors. Not needed for normal kernel calculation routine and big, therefore separate.

createSeperatedLoopMesh(*name='SepLoopMesh'*, *dipole=True*, *exportVTK=False*, *refinement_para=1.0*, *max_area_factor=1.0*)
Creates a mesh that contains the receiver and the transmitter loop.

createYVec(*max_length=0.2*, *max_num=300*, *y_factor=2.0*, *calc_3D_stats=True*)
Creates the y vector discretization for the 2D kernel mesh.

The y vector represents the y values of the 3D Kernel mesh before the integration to 2D.

Parameters

- **max_length** (*float [0.2]*) – Maximum distance between two slices inbetween the source dipoles.
- **max_num** (*integer [300]*) – Maximum number of slices. Overrides max_length if they conflict.
- **y_factor** (*float [2.]*) – Extension of the kernel mesh (and therefore integration volume) in the y direction. Should be at least 2 times the loop diameter or shortest edge length. This value defines the multipier.

createZVector(*numz*, *minz*, *min_thk=0.5*)
Creates a sinus hyperbolicus shaped Z discretisation in numz steps between 0 and minz.

ellipticalDecomposition()
Computes the counter and corotating parts of the given magnetic fields with respect to a given earth magnetic field.

Parameters

- **Bfield** (*complex field [3, n] or string*) – Optional. Possibility to insert a pre calculated field.
- **Inclination** (*float*) – Inclination of the earth magnetic field at the loop site in rad [0… 2pi]
- **Declination** (*float*) – Declination of the magnetic field at the loop site in rad [0… 2pi]
- **B** (*np.array of shape (3, n)*) – Magnetic field of the loop
- **Second part of the kernel calculation.**
- **- mainly from Weichman et al. (2000)**

static ellipticalDecomposition_multi(*Bfield*, *earth*)
Computes the counter and corotating parts of the given magnetic fields with respect to a given earth magnetic field.

Parameters

- **Bfield** (*complex field [3, n] or string*) – Optional. Possibility to insert a pre calculated field.
- **Inclination** (*float*) – Inclination of the earth magnetic field at the loop site in rad [0… 2pi]
- **Declination** (*float*) – Declination of the magnetic field at the loop site in rad [0… 2pi]
- **B** (*np.array of shape (3, n)*) – Magnetic field of the loop
- **Second part of the kernel calculation.**
- **Literature**
- **———-**
- **- Weichman et al. (2000)**
- **- Hertrich (2005, Appendix)**

- **- Hertrich (2008, eq. 6 ff.)**

export1D(*savename*, *loop_layout=True*, *title='{0.survey.earth!r}'*)

export2DKernel(*fig=None*, *ax=None*, *savename=None*, *png_dpi=300*, *noYLabel=False*, *index=0*, *colorBar=True*, *size=13*, *pdf=None*, *fixed_cbar=False*, ***kwargs*)
Exports 2D Kernel for given pulse moment. Kwargs are redirected to *show*.

export2DKernel2PDF(*name*, *fixed_cbar=False*, ***kwargs*)
Export 2D Kernel for all pulse moments as stiched pdf. Kwargs are redirected to *export2DKernel*.

exportMagnetization(*name*, *vtk_export=False*, *pulse=0*)
Export a previously calculated magnetization vector as numpy vector and optionally vtk file.

exportVTK(*savename, save=['abs'], save_in_log=False*)

fid
Reference to sounding (FID) class instance in survey.

getKernel(*reduced=True*)

getSliceCoords()
Returns input coordinates for custEM Slice interpolation of magnetic fields to the kernel slices.

getZVector(*reduced=True*)

interpolateBFieldToKernel(*recalc_prim_on_kernel=False*, *recalc_primary=False*, *num_cpu=32*, *calc_3D_stats=True*)
Takes the rx Bfield and interpolates it to the kernel mesh.

static kernelCalculation_multi(*fid*, *earth*, *txalpha*, *txbeta*, *txzeta*, *txperpend*, *rxalpha=None*, *rxbeta=None*, *rxzeta=None*, *rxperpend=None*, *calc_theta=False*)

kernelIntegration(*calc_theta=False*)
Computes the integration of the kernel with respect to the desired dimension.

Parameters

- **decomposition** (*(alpha, beta, zeta)*) – Bfield_part essentially consists of the output from the elliptical decomposition of the magnetic field.
- **measurement** (*class*) – An instance of a measurement class has to be given in order to keep the number of input arguments manageable.
- **earthmagnitude** (*float*) – Magnitude of the earth magnetic field [Tesla]. Aproximatly about 30000 to 65000 nT (1 nT = 1e-9 Tesla).
- **Third part of the kernel calculation.**

larmor
Larmor frequency [Hz] from earth defined in survey.

load(*savename*, *load_loopmesh=True*, *kernelmesh2d=None*, *load_kernelmesh=True*, *use_order_refinement=True*)
Load a previously saved kernel (.npz-format).

magnetization_magnitude

pulses
Reference to pulse moments from sounding (FID).

release_memory()
Calling this function is releasing some attributes that are using a fairly big amount of memory.

Sets the following attributes back to None:

- The interpolation matrix between the loop meshes and the kernel mesh

interpolationMatrix

- local copies of the magnetic fields (fields in tx and rx are not

effected) *txBfield*, *rxBfield*

- the 3D kernel mesh cell center and volumes

kernelMeshCellVolume, *kernelMeshCellCenter*

- the elliptical decomposition of the tx and rx bfields

txalpha, *txbeta*, *txzeta*, *txperpend*, *rxalpha*, *rxbeta*, *rxzeta*, *rxperpend*

Note: a recalculation of the kernel will take about the same amount of time as the first call, as all cached variables are gone, however apart from a recalculation, the other purposes of the kernel class (export, figures, inversion(without recalculation)) are not effected.

Another note: If you want to use this method only for saving disk space in case you save the kernel class, then you might consider the *light* flag of the *.save* method instead.

rx
Reference to receiver class instance in survey.

rx_area
Area of the receiver loop.

rx_index

save(*savename=None*, *save_interpolation_mat=False*, *save_loopmesh=False*, *light=True*, *kernelmesh_name=None*)
Save the basic information to restore the Kernel class later.

set1DKernelMesh(*mesh*, *calc_3D_stats=True*)
Sets the 1D kernel mesh.

Parameters

- **mesh** (*stirng or pygimli.Mesh*) – Filename or mesh instance of a 2D mesh in the x-y plane.
- **Need**
- **—-**
- **z discretization** – Can be setted via *createZVector*, *setZVector* or direct use of *create1DKernelMesh*. However the needed information to do that may not be available on the fly, therefore no default z vector is created.

set2DKernelMesh(*inmesh*, *yvec=None*, *order=0*, *integration_mat=None*, *calc_3D_stats=True*)
kwargs to createYVec if YVec is None

setDebug(*debug: bool*)

setModel(**args*, ***kwargs*)
Pipes args and kwargs to *self.tx.setModel*. Same for rx.

setPulsesDirectly(*pulses*)
Set pulse moment vector manually if not supported by survey + fid. (This is called when loading a kernel from the harddisk, mainly for plotting reasons). For all calculation purposes a survey and fid class is recommended.

setRx(*rx*, ***kwargs*)
Sets initialized loop or pipe arg and kwargs to *loadLoop*.

setSurvey (*survey*, *fid=0*)
Sets survey class containing necessary information for the kernel.

Parameters

- **survey** (*comet.snmr.survey.Survey or None*) – Sets given survey class instance or create empty class instance.
- **fid** (*integer [0]*) – Index of corresponding sounding in the survey.

setTx (*tx*, ***kwargs*)
Sets initialized loop or pipe arg and kwargs to *loadLoop*.

setYVector (*vector*)

setZVector (*vector*, *min_thk=0.5*)
Defines the attribute zvec.

Sets the given vector as z discretization. Attention: the value for min_thk defines the minimum thickness of the discretization used in the end. For all thicknesses in vector smaller than min_thk, the Kernel is integrated to match the min_thk. For calulation of the kernel function the original given vector is used.

Parameters

- **vector** (*array_like*) – Z discretization in m to be used for the kernel calculation. If a new vector is to be created, please also take a look at the method *createZVector*.
- **min_thk** (*float*) – Minimum thickness te kernel and zvec is integrated if returned. This leads to higher accuracy in the vicinity of the loop.

shape

show (*toplot=['real', 'imag', 'amp', 'phase', '0D'], indices=None, savename=None, normed=False, suptitle=None, ax=None, pulse_in_log=False, kernel_absolute_values=False, cbar_percentage=0.99, fixed_cbar=False, lut=33, show_marked_edges=False, **kwargs*)
Visualise the Kernel with respect to the desired dimension.

Automatically defined within the kernel class via the parameter kernel.dimension = [0. . . 3]. Plotting of a kernel in the desired dimension is only possible if the kernel is also calculated with respect to that dimension. It's not possible to calculate the kernel with kernel.dimension = 1 and then plot the kernel with kernel.dimension = 2.

0D : Simple Graph plotting kernel-values over pulsemoments

1D : Graph with 1D integrated kernels over the depth of the model

2D : Slice of the x-z-plane with triangle mesh containing the 2D

3D : Export of the kernel in vtk format for visualising.

none so far

Plots the 1D integrated Kernel with a given z discretisation over the measured pulse sequences.

toplot: list [['real', 'imag', 'amp', 'phase', '1D']] There are different possibilities to plot the kernel. This parameter defines which part of the kernel is shown. Possible options are: 'real', 'imag', 'amp', 'phase', '0D' (integrated over z). All strings in the toplot variable will be plotted in the same order given in the list.

cMap: string ['viridis'] Defines the colormap used to display the kernel. In order to get a good contrast between the max and min as well as being useful in comparison with MRSMatlab, 'viridis' is the default colormap. Any colormap reachable by the plt.get_cmap(. . .) method can be chosen.

normed: bool [True] A on the dimension based normalisation of the plot permits a better assessment of the kernel distribution.

ax: plotting ax or list of axes [None] Plot on a predefined ax and gives back the ax. A onedimensionla list of axes is also accepted, if the number of items in 'toplot' is the same as the available axes.

lut: None or int [None] Number of colors for the colorbar. If lut is not None it must be an integer giving the number of entries desired in the lookup table, and name must be a standard mpl colormap name.

indices: list By default one 2D plot is created for each pulsemoment. In order to limit the number of plots the optional paramter indices can be given as a list of indices referring to the pulse moments to be shown.

cMap: string ['viridis'] See Parameter 1D.

normed: bool [True] A on the dimension based normalisation of the plot permits a better assessment of the kernel distribution.

show_marked_edges: boolean [False] Whether or not marked edges gets drawn.

possible kwargs for matplotlib: cMin, cMax for range of the colorbar. All other kwargs are reaching matplotlib functions.

default label 2D: 'integrated kernel (2D) [nV/m^2] pulsemoment: {:.3f} As' .format(self.pulses[i])

A self-sufficient plot of the kernel without any integration would result in a set of 3D Cubes and is not implemented for now.

Instead the kernel will be saved in vtk format which can be easily handled.

savename: string A String defining the relative path to the vtk-file the kernel will be saved in. If not given the default savename will be flagged with the string '_default_' and contain some information about the kernel.

Example

2D:

```
>>> ax, cbar = kernel.show(indices=[16], cMin=-1,
>>>                        cMax=2, size=20, pad=0.7)
>>> ax.set_ylim(-50, 0)
```

show2DMesh()

showLoopLayout(*ax=None*, ***kwargs*)

sliceKernel1D(*num_cpu=None*, *loop_mesh=None*, *new_bfield=False*, *interpolate_bfield=True*, *slice_name=None*)

sliceKernel2D(*savename=None*, *forceNew=False*, *loopSaveName=None*, *num_cpu=None*, *new_bfield=False*, *loop_mesh=None*, *slice_name=None*, ***kwargs*)
2D Kernel in a memory saving parallel computation approach.

tx
Reference to transmitter class instance in survey.

tx_area
Area of the transmitter loop.

tx_index

updatable

zvec
z discretisation

`comet.snmr.kernel.kernel_bib.`**`calcInterpolationMatrix_para`**(*source_mesh*, *target_pos*, *num_cpu=8*)

`comet.snmr.kernel.kernel_bib.`**`calculateKernelFromSlices`**(*survey_name*, *invmesh*, *cfgname*, *max_length=0.05*, *max_num=400*, *path_name='kernel'*, *num_cpu=48*, *h_order=1*, *json_name=None*, *force_new_paths=True*, *kernel_name='kernel/kern_{}'*, *slice_export_name='kernel_slice'*)

survey: string Filepath of the survey containing the FIDs.

invmesh: string Filepath for the inversion mesh the kernel is calculated on.

cfgname: string Filepath of the secondary config containingin formation for custEM. The same file should have been used for the field calculation.

max_length: float [0.05] Minimum distance between two slices in y direction.

max_num: integer [400] Number of slices for y discretization.

path_name: string ['kernel'] Final name for the slices will be {mdir}/paths/{path_name}_{number}_path.xml "mdir" is defined in the secondary config.

num_cpu: integer [48] Number of cores used.

h_order: integer [1] Order of h-refinement when setting the invmesh in the kernelclass.

json_name: string [None] Optional name for the json file. Alternatively a tempory file is created.

force_new_path: boolean [True] Deletes old pathfiles (slices) and forces the generation of new ones.

kernel_name: string ['kernel/kern_{}'] Filepath of used for export of the kernel functions. Need to contain a "{}" which is filled with the index of the corresponding FID in the survey class.

slice_export_name: string ['kernel_slice'] Filepath of the interpolated magnetic fields for the individual kernel slices. Full slice path contains: "{r_dir}/{approach}/{mesh_name}/{mod_name}_interpolated/tx_{tx_number}_{slice_export_name}_imesh_{slice_number}.npy"

kernel_name (added a "_{}" if not in original string)

`comet.snmr.kernel.kernel_bib.`**`checkForKernel`**(*name*, *mkdir=False*)

`comet.snmr.kernel.kernel_bib.`**`create1DInterpolationSlices`**(*kern*)

`comet.snmr.kernel.kernel_bib.`**`create2DInterpolationSlices`**(*kern*)

`comet.snmr.kernel.kernel_bib.`**`integrateKernelH2`**(*mat*, *array*)

`comet.snmr.kernel.kernel_bib.`**`simpleZVec`**(*numz*, *minz*, *reduced=False*)

Module contents

Module comet/snmr/kernel

comet.snmr.misc package

Submodules

comet.snmr.misc.IO_pdf module

Part of comet/snmr/misc

`comet.snmr.misc.IO_pdf.`**`closeAxis`**(*ax*)

`comet.snmr.misc.IO_pdf.`**`exportColorBarPDF`**(*name=None*, *cMap='viridis'*, *cmin=0*, *cmax=1*, *orientation='horizontal'*, *label='colorbar label'*, *size=14*, *ax=None*, *dpi=300*)

`comet.snmr.misc.IO_pdf.`**`exportKernelPDF`**(*kern, fig=None, ax=None, savename=None, dpi=300, noYLabel=False, index=0, xl_add='', rotate=False, plotlims=None, colorbar=False, figsize=[10, 6]*)

`comet.snmr.misc.IO_pdf.`**`returnFigure`**(*ax*)

`comet.snmr.misc.IO_pdf.`**`robustPDFSave`**(*fig_or_ax*, *name*, ***kwargs*)

comet.snmr.misc.plot_routines module

Part of comet/snmr/misc

`comet.snmr.misc.plot_routines.`**`drawSoundingCurve`**(*ax, kern_mat, pulses, size=12, color='r', to_plot=['abs'], y_ticks_right=True, plot_abs=True, title='volume-integrated kernel', marker_size=5, **kwargs*)

comet.snmr.misc.plotting_tools module

Part of comet/snmr/misc

`comet.snmr.misc.plotting_tools.`**`grayCBarPalette`**(*steps, lims=[1.0, 0.5]*)

`comet.snmr.misc.plotting_tools.`**`setAxSize`**(*ax*, *size*)

`comet.snmr.misc.plotting_tools.`**`setCBarSize`**(*cbar*, *size*)

Module contents

Module comet/snmr/misc

class `comet.snmr.misc.`**`Constants`**

Bases: `object`

`calcCurieFactor`(*temperature*)

`gamma`

comet.snmr.modelling package

Submodules

comet.snmr.modelling.errors module

Part of comet/snmr/modelling

`comet.snmr.modelling.errors.`**`DepricationWarning`**(*msg=None*)

exception `comet.snmr.modelling.errors.`**`InputError`**(*file=None*, *msg=None*)
Bases: `Exception`

exception `comet.snmr.modelling.errors.`**`KernImportError`**(*value*)
Bases: `Exception`

comet.snmr.modelling.mrs module

Part of comet/snmr/modelling

Magnetic resonance sounding module.

class `comet.snmr.modelling.mrs.`**`MRS`**(*survey=None*, *fid=0*, *kernel=None*, *mtype='smooth'*, *dtype='rotatedAmplitudes'*, ***kwargs*)
Bases: `comet.snmr.modelling.nmr_base.SNMRBase`

Magnetic resonance sounding (MRS) manager class.

`searchForLambda`(*startLam=20000*)
Runs several inversion runs to find the highest lambda which is able to fit the data within its errors.

`showDataAndError`(*ax=None*, *figsize=(10, 8)*, *as_log=False*)
Show data cube along with error cube.

`showDataAndFit`(*compare_to=None*, *figsize=(8, 6)*, *savename=None*, *clim=None*, *suptitle=None*, *separated=False*, *savematrices=False*)
data and error weighted misfit. 1,1 or 2,2 for complex

`showKernel`(*ax=None*, *save=None*, ***kwargs*)
Show the kernel as matrix (Q over z). If Kernel is a class object, the plotting order is redirected to Kernel.show(**kwargs)

To see more about the plotting options type this in the console:

```
>>> import kernel as k
>>> help(k.Kernel.show)
```

`showResult`(*figsize=(10, 8)*, *save=''*, *fig=None*, *ax=None*, *syn=None*, *wclabel=None*, *t2label=None*, *color=None*)
Show theta(z) and T2*(z) (+uncertainties if there).

`showResultAndFit`(*figsize=(12, 10)*, *save=''*, *maxdep=0.0*, *clim=None*, *suptitle=None*, *syn=None*, *wclabel=None*, *t2label=None*)
Show ec(z), T2*(z), data and model response.

`splitModel`(*model=None*)
Split model vector into d, theta and T2*.

class `comet.snmr.modelling.mrs.`**`MRSGenetic`**(**args*, ***kwargs*)
Bases: `comet.snmr.modelling.mrs.MRS`

MRS class derivation using a genetic algorithm for inversion.

genMod(*individual*)
Generate (GA) model from random vector (0-1) using model bounds.

plotEAstatistics(*fname=None*)
Plot EA statistics (best, worst, . . .) over time.

plotPopulation(*maxfitness=None*, *fitratio=1.05*, *savefile=True*)
Plot fittest individuals (fitness<maxfitness) as 1d models

Parameters

- **maxfitness** (*float*) – maximum fitness value (absolute) OR
- **fitratio** (*float [1.05]*) – maximum ratio to minimum fitness

runEA(*nlay=None*, *eatype='GA'*, *pop_size=100*, *num_gen=100*, *runs=1*, *mp_num_cpus=8*, ***kwargs*)
Run evolutionary algorithm using the inspyred library

Parameters

- **nlay** (*int [taken from classic fop if not given]*) – number of layers
- **pop_size** (*int [100]*) – population size
- **num_gen** (*int [100]*) – number of generations
- **runs** (*int [pop_size*num_gen]*) – number of independent runs (with random population)
- **eatype** (*string ['GA']*) –

  algorithm, choose among:

  'GA' - Genetic Algorithm [default]

  'SA' - Simulated Annealing

  'DEA' - Discrete Evolutionary Algorithm

  'PSO' - Particle Swarm Optimization

  'ACS' - Ant Colony Strategy

  'ES' - Evolutionary Strategy

comet.snmr.modelling.mrs.**showErrorBars**(*ax*, *thk*, *val*, *thkL*, *thkU*, *valL*, *valU*, **args*, ***kwargs*)
Plot wc and t2 models with error bars.

comet.snmr.modelling.mrs.**showT2**(*ax*, *thk*, *t2*, *maxdep=0.0*, *label=None*, *color='g'*)
Show T2 function nicely.

comet.snmr.modelling.mrs.**showWC**(*ax*, *thk*, *wc*, *maxdep=0.0*, *dw=0.1*, *label=None*, *color='g'*)
Show water content function nicely.

comet.snmr.modelling.mrs_survey module

Part of comet/snmr/modelling

class comet.snmr.modelling.mrs_survey.**MRT**(*survey=None*, *dim=2*, *dtype='complex'*, *mtype='smooth'*)
Bases: object

create1DKernelMesh(*verbose=False*)

createFOP (*kernelmesh=None*, *secondary=False*, *para_mesh_2d=None*, ***kwargs*)
kwargs: order (h refinement order for kernel mesh)

createFOPMesh ()

createINV (*lam=1000*, *verbose=True*, *debug=False*, ***kwargs*)
Create inversion instance (and fop if necessary with nlay).

lam: float [100] Lambda factor for inversion.

verbose: bool [True] Additional verbose decission, can be True, even if the rest of the Manager should remain silent. Most information of the different iterations is printed in the console. It's recommended to set verbose in this case to True (default).

lambdaFactor, float [0.8] Sets lambda factor for Marquardt scheme.

robust, bool [False] Sets the robust flag for the data. See pg.RInversion for more details

logTrans, bool [True] Applies a logarithmic transformation to the data. Its recommended to do so (default), due to the dealing with water contents, which can't be negative. Logarithmic transformation is the easiest way to archieve that.

blockyModel, bool [False] Instead of the standard L2-Norm a L1 Norm can be used to allow for more blocky models. Heavy changes in watercontent and relaxation times can sometimes be fitted better this way.

data
Concatenated data vectors of sounds.

dataIndices

data_slices
Slices to get single data from self.data.

Data[sound #2] = mrt.data[mrt.data_slices[1]]

dtype

error
Concatenated error vectors of sounds.

getSingleDataAndError (*sounding_idx*)

initSoundings (*override=False*)
Extends the sounding list for the fids in survey. Called automatically is necessary.

kernels
List with underlaying kernels from sounds.

loadResults (*basename*, *gates=True*, *pulses=True*)
returns (model, error, response, chi2)

mtype

saveResults (*basename*)
Saves orig data, model, error and forward model as well as chi2.

setDataAndErrorCube (*data*, *error*, *phase*, *df=None*)
Depricated!

Set data and error cubes using the methods of the single soundings.

Input has to be a list or iterable object of data, and error cubes (pulses x gates) a corresponding list of phase vectors for each pulse and a float defining the frequency offset per sounding.

setDataAndErrorVector(*data*, *error=None*, *phi=None*, *df=None*)
Depricated! Set Data and Error in MRT and the underlaying MRS instances.

setDataType(*dtype*)

setKernelMesh(*mesh*, *order=1*, ***kwargs*)

setKernels(*basename*, *load_loopmesh=False*, *use_order_refinement=True*, *indices=None*)
Sets the kernels for the underlaying soundings. Basename will be formatted with index. Example 5 soundings, basename = 'kern_{}' will result in import of **kernel_0**, **kernel_1**, . . . , **kernel_4**.

setModelTrans(*thk=(10, 1, 30, 'log')*, *wc=(0.3, 0.0, 0.7, 'cot')*, *t2=(0.2, 0.005, 1.0, 'log')*)
Sets model transformation for water content, relaxation times, and thickness (1D). input = (startvalue, min, max, type). Known types are cotangens ('cot') and logarithmic ('log') transformations.

setModelType(*mtype*)

setSurvey(*survey*)
Defines the survey that holds the various soundings and datasets.

setZWeight(*z_weight*)

showFids(*to_plot='abs'*, *rows_cols=None*, *ax=None*, *draw='data'*, ***kwargs*)
kwargs to ph.plot.drawFID(**kwargs)

showSounding(*index*, *ax=None*, *to_plot='abs'*, *draw='data'*, *figsize=(5, 3)*, ***kwargs*)
Shows Data, Error or misfit of a site.

simulate(*model*, *error*, *samplingrate=1000.0*, *max_time=1.0*, *num_gates=50*, *verbose=False*, ***kwargs*)

updateData()
Update data vector in inversion instance.

comet.snmr.modelling.nmr_base module

Part of comet/snmr/modelling

Nuclear magnetic resonance base manager as used by MRS and MRT manager classes

class comet.snmr.modelling.nmr_base.**SNMRBase**(*survey=None*, *fid=0*, *kernel=None*, *mtype='block'*, *dtype='rotatedAmplitudes'*, *update_kernel=False*, *dim=1*, ***kwargs*)
Bases: object

Manager base class for MRS and MRT manager classes.

K

applyBoundsAndTrans()
Append the previously given bounaries for the model transformation to the forward operator.

calcModelCovarianceMatrix()
Compute linear model covariance matrix.

calcModelCovarianceMatrixBounds()
Compute model bounds using covariance matrix diagonals.

createFOP(*nlay=3*)
Creates the forward operator (FOP). Two possibilities are supported: block and smooth. The choice affects the inversion process and therefore its results.

possible keyword argument for block FOP is ‘nlay’ to define the number of layers, the FOP is calculating (default = 3).

`createINV`(*lam=1000*, ***kwargs*)
Create inversion instance (and fop if necessary with nlay).

Parameters

- **lam** (*float [100]*) – Lambda factor for inversion.
- **verbose** (*bool [True]*) – Additional verbose decission, can be True, even if the rest of the Manager should remain silent. Most information of the different iterations is printed in the console. It’s recommended to set verbose in this case to True (default).
- **special kwargs for Marquardt scheme (block)**
- **————————————————-**
- **lambdaFactor, float [0.8]** – Sets lambda factor for Marquardt scheme.
- **robust, bool [False]** – Sets the robust flag for the data. See pg.RInversion for more details
- **special kwargs for smooth scheme**
- **————————————**
- **logTrans, bool [True]** – Applies a logarithmic transformation to the data. Its recommended to do so (default), due to the dealing with water contents, which can’t be negative. Logarithmic transformation is the easiest way to archieve that.
- **blockyModel, bool [False]** – Instead of the standard L2-Norm a L1 Norm can be used to allow for more blocky models. Heavy changes in watercontent and relaxation times can sometimes be fitted better this way.

`data`
Data Vector representation with respect to self.dtype. Returns None if no sounding or data_cube in sounding is found.

`dtype`

`error`

`fid`
Reference to sounding (FID) class instance in survey.

`invert`(*data=None*, *error=None*, *phase=None*, *lam=1000*, *runChi1=False*, ***kwargs*)
# TODO!

`loadMRSI`(*filename*, *verbose=True*)
Load data, error and kernel from mrsi file

`loadSurvey`(*dataname*)

`mtype`

`setBoundsAndTrans`(*thkBounds=[10.0, 1.0, 30.0]*, *wcBounds=[0.3, 0.0, 0.7]*, *t2Bounds=[0.2, 0.005, 1.0]*, *trans=[’log’, ’cot’, ’log’]*)
Sets the boundarys and transformation for the model domain.

Parameters

- **thkBounds** (*list of floats [[10., 1., 30.]]*) – Startvalue, lower and upper boundary for thickness of each layer in 1D. Ignored for smooth models (or 2D).
- **wcBounds** (*list of floats [[0.3, 0.0, 0.7]]*) – Startvalue, lower and upper boundary for water content.

- **t2Bounds** (*list of floats [[0.2, 0.005, 1.0]]*) – Startvalue, lower and upper boundary for relaxation times.
- **trans** (*list of strings [['log', 'cot', 'log']]*) – Defines the type of model transformation. logarithmic ('log') or cotangens ('cot')

setDataType(*dtype*)

setKernel(*kernelfile=None*, *load_loopmesh=True*, *load_kernelmesh=True*, *use_order_refinement=True*)
Load or initialize a new Kernel class instance for calcualting the NMR kernels.

setModelType(*mtype*)

setSurvey(*survey*, *fid=0*)

showCube(*ax=None*, *vec=None*, *islog=None*, *clim=None*, *clab=None*, *cmap='viridis'*, *cbar=True*)
Plot any data (or response, error, misfit) cube nicely.

simulate(*model*, *err=2.5e-07*, *samplingrate=1000.0*, *max_time=1.0*, *num_gates=50*, *verbose=False*, *debug=False*, ***kwargs*)
Creates forward operator and calculates a synthetic response to a given model. Keyword arguments are passed to the function createFOP and to FOP.response. You can also define the 'Type' to be 'smooth' or 'block' or let the simulate function analyse the input.

returns datacube, errorcube (both complex) and phaseinformation (for rotated amplitudes)

Parameters model (*list of lists*) – Given model of shape [water_content, relaxation_time] if forwarded to FOP to generate synthetic data set.

comet.snmr.modelling.nmr_base.**effectiveNoise**(*area*, *noise_lvl=0.0036*, *sample_rate=1000.0*, *time=1.0*)
Calculates the effective noise of a loop for simulation.

noise_lvl = 3.6e-3 nV / m^2 / sqrt(number_of_samples) This is a standart noise_lvl from measurements in Schillerslage, Germany. Output in Volt.

comet.snmr.modelling.nmr_base.**getPhiByGridSearch**(*data*)

comet.snmr.modelling.smooth_syn module

Part of comet/snmr/modelling

comet.snmr.modelling.smooth_syn.**archie**(*porosity*, *saturation*, *water_resistivity*, *tortuosity=1.0*, *cementation=1.3*, *saturation_exponent=2.0*, *formation_factor=None*)
porosity(z), saturation(z)

returns resititvity_bulk(z) cite{}

comet.snmr.modelling.smooth_syn.**brooksCorey**(*z*, *water_table*, *porosity*, *lam=1.6*, *height_zero=0.12*)
after Brooks and Corey (1964) cite{}

comet.snmr.modelling.smooth_syn.**costabel**(*saturation*, *t2_saturation*, *lam=1.6*)
cite{costabel2011NSG} Costabel, S., and U. Yaramanci, 2011, Relative hydraulic conductivity and effective saturation from Earth's field nuclear magnetic resonance – a method for assessing the vadose zone: Near Surface Geophysics, 9, 155–167.

comet.snmr.modelling.smooth_syn.**effectiveSaturationToWater**(*saturation_eff*, *water_saturation*, *water_residual=0.05*)
saturation_eff = (water - water_residual)/ (water_saturated - water_residual)

`comet.snmr.modelling.smooth_syn.`**`modelVadose`**(*z*, *water_table*, *porosity*, *t2_saturated*, *water_resistivity*, *height_zero=0.12*, *water_residual=0.05*, *lam=1.6*, *verbose=False*, ***kwargs*)

Calculates a synthetical vadose zone on basis of a Brooks-Corey model for saturation over the vadose zone, whereas lambda is the pore size distribution index.

Also calculates the electrical resistivity(z) via Archies law, as well as the distribution of relaxation times based on Costabel and Yaramanci (2011).

returns (z, resistivity, water_content, relaxation_times)

`comet.snmr.modelling.smooth_syn.`**`test_local`**()

comet.snmr.modelling.snmrModelling module

Part of comet/snmr/modelling

Modelling classes for core magnetic resonance (1D, 2D)

`class` `comet.snmr.modelling.snmrModelling.`**`MRS1dBlockQTModelling`**(*survey*, *fid=0*, *nlay=3*, *dtype='complex'*, *kernel=None*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

MRS1dBlockQTModelling - pygimli modelling class for block-mono QT inversion

f=MRS1dBlockQTModelling(lay, KR, KI, zvec, t, verbose = False)

`fid`

`forward`(*par*, *verbose=False*, *num_cpu=12*)

yield model response cube as vector

`iscomplex`

`response`(*par*)

`class` `comet.snmr.modelling.snmrModelling.`**`SNMRJointModelling`**(*mrt=None*, *verbose=False*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

Joint modelling operator for multiple transmitter receiver combinations

`addFOP`(**fops*)

`createJacobian`(*model*)

`forward`(*model*)

`response`(*model*)

`setFOPs`(*fops*)

`class` `comet.snmr.modelling.snmrModelling.`**`SNMRModelling`**(*survey*, *kernel*, *fid=0*, *dtype='complex'*, *mesh=None*, *num_cpu=12*, *update_kernel=False*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

Modelling class for surface nuclear magnetic resonance (SNMR).

The class is based on the ModellingBase class of pygimli and therefore contains a various amount of parameters and functions as well as some protected members to ensure a generalized interface suitable for the pygimli inversion engine.

For further details about the spezifications of the modelling base, be referred to the pygimli API available from the official project website www.pygimli.org.

static amplitudeJacobian(*Mcomplex*, *model*)

calculateKernel(*matrix=False*, *interpolate=False*, *forceNew=False*, ***kwargs*)

createJacobian(*model=None*, ***kwargs*)

Caculate the Jacobian Matrix of a NMR Kernel, with or without relaxation times included (model dependancy for this).

kwargs are redirected to kernelClass.calculate()

Example

```
>>> # complex jacobian without relaxation time
>>> FOP = MRModelling('a valid kernel class')
>>> FOP.createJacobian()  # sets FOP.jacobian
```

dimension

fid

Reference to sounding (FID) class instance in survey.

forward(*model*, ***kwargs*)

Forward response of the kernel to a specific destribution of watercontent or relaxation times.

model.shape: array.shape = 2 or 3, numLayers (watercontent only: 2, 3 with relaxation times), first entry = thickness

```
>>> thickness = [1, 5, 10]
>>> # first layer 0...1 m
>>> # second layer 1...6 m
>>> # third layer 6...16 m
>>> # after that homogeneous halfspace
>>> watercontent = [0.2, 0.3, 0.1, 0.2]  # 1 == 100%
>>> # one entry more than thickness, last entry for halfspace
>>> model = np.array((thickness,
>>>                  watercontent,
>>>                  [100, 200, 14, 100]))  # relaxation times
>>> measurement = mrs.response(model)
```

iscomplex

jshape

(data, model) == (pulses * gates, model * number of parameters)

Type Jacobian shape

kshape

(data, model) == (pulses, model)

Type Kernel shape

response(*model*)

Calculates the forward response of a SNMR measurement, and returns an 1D numpy array containing the real and imaginary parts of the response. One Voltage value for each pulse moment q and time gate g.

```
data type: complex

([real(V_11), ..., real(V_1Q),
  real(V_21), ..., real(V_2Q),
  ...,
  real(V_N1), ..., real(V_NQ),
  imag(V_11), ..., ...,
  ...       , ..., imag(V_NQ)]), shape: (2*N, Q)

data type: not complex

([abs(V_11), ..., abs(V_1Q),
  abs(V_21), ..., abs(V_2Q),
  ...,
  abs(V_N1), ..., abs(V_NQ)]), shape: (N, Q)
```

setKernel (*kernel*)

setModel (*model*)

Parameters model (*array_like*) – Array that contains three array_like objects. First the thickness of the different layers (number of layers - 1). The second and third array contains the water contents and relaxation times of each layer.

Example

```
>>> FOP = SNMRModelling('a valid kernel class')
>>> model = [[5., 10.],   # thickness [m]
>>>          [0.1, 0.25, 0.4],  # water content [1]
>>>          [0.1, 0.1, 1.]]  # relaxation times [s]
>>> FOP.setModelVec(model)
```

setSurvey (*survey*, *fid=0*)

updateDataPhase ()

Sets data phase for complex inversion. If no model is given the starting model is used.

vector

Module contents

Part of comet/snmr/modelling

comet.snmr.survey package

Submodules

comet.snmr.survey.survey module

Part of comet/snmr/survey

Enhanced sounding class for SNMR data sets and supporting variables. Sounding class can hold any number of Measurement class instances each representing single FIDs.

class comet.snmr.survey.survey.**Earth**(*incl=60.0*, *decl=2.0*, *mag=4.8e-05*, *rad=False*)

Bases: object

Parameters

- **inclination** (*float [60.]*) – Inclination of the earth magnetic field in rad or degree.
- **declination** (*float [2.]*) – Declination of the earth magnetic field in rad or degree.
- **magnitude** (*float [48000 * 1e-9]*) – Magnitude of the earth magnetic field in Tesla.
- **rad** (*boolean [False]*) – Input inclination and declination in rad?

Example

```
>>> from comet.snmr.survey import Earth
>>> e = Earth(inclination=45, declination=0, magnitude=4.8*1e-5)
>>> print(e)
```

copy()

field

Static magnetic field vector from earth defined in survey.

larmor

magnitude

class comet.snmr.survey.survey.**FID**(*tx=0*, *rx=0*, *pulses=None*)

Bases: object

Single SNMR experiment (sounding) using a simple Free Induction Decay (FID).

Attributes to be setted directly:

amperes

Ampere vector [A].

curie

Curie factor for kernel calculation. Read only. Calculated automatically by setting temperature.

deadtime

Effective deadtime (device + half pulse) [s].

filterGates(*mint=0.0*, *maxt=2.0*)

Dismiss not desired time gates.

Parameters

- **mint** (*float [0.0]*) – Cut all data reqired before mint (in seconds). This is done using the gate midpoints including deadtime.
- **maxt** (*float [2.0]*) – Cut all data reqired after maxt (in seconds). This is done using the gate midpoints including deadtime.
- **Append new .gating to restore old gates**
- **raw_data remain untouched)**

gates

Time gate midpoint vector [s] (including deadtime).

gating(*num_gates=42*, *verbose=False*)
(extracted from MRSMatlab, 2017)

y=exp(x) For some interval x(a:b) the exact mean within exp(x(a:b)) yAverage = exp(mean(log(y(a:b)))) t(yAverage) = mean(t(a:b))

Problem: Logarithm is nice for exact average of exponential function. But signals are noise contaminated. 1. Logarithm of gaussian noise changes noise structure from gaussian to lorenzian. Averaging of lorenzian distributed noise is not zero. 2. Since noise can make signal negative a dc shift is added to make signals positive. This deminishes the accurancy of averaging in logspace. For large constant shift averaging in logspace becomes equivalent to average in linspace. However this is nice for noise structure. So we have a tradeoff. Finally, from some amount of intervals on, e.g. 20 within interval [0 1]/s averaging is sufficiently exact in any case.

MMP 18/10/2011

getComplexData()

getRotatedAmplitudes()
Returns Data and Error as real component of the rotated Vecs.

load(*savename*, *df_removed=True*)
Load previously saved FID class instance from savename (.npz) (numpy compressed binary data structure).

Usually imported data are cleansed from frequency offsets (df) before saving. However there is no auto detection for that. In rare cases (if you know what youre doing) data are saved without removing df first. Then df_removed has to be set to False. Otherwise the raw data

pulses
Pulse moment vector [As].

rotateAmplitudes(*raw_data=False*)
One of the three main ways for NMR forward modelling is to use rotated amplitudes, instead of using the amplitudes of the complex data or the complex data itself. If the phase information of the noise free data is known (synthetic data) or fitted (e.g. monoexponential fit) the rotated Amplitudes (also complex, do not confuse) have the advantage of containing all the information in the real part (together with noise), where the imaginary part contians only noise and can therefore be discarded later.

Can be used on gated or ungated data, however this call alters the raw_data!

Parameters **raw_data** (*boolean [True]*) – Flag to decide if raw data or gated data are rotated. Default is raw data, however if no raw data are

Returns

Return type complex rotated raveled data.

save(*savename*)
Saves FID class instance under savename. Expect savename with ending .npz (numpy compressed binary data structure).

setDataPhase(*data_phase*)
Sets variable data_phase. Expect single float value for data phase in rad.

setFrequencyOffset(*df*)
Sets frequency offset of tx pulse to larmor frequency.

Expect one value per pulse or one single value (used for all pulses). None is treated as zero offset (internal initialization).

setGatedDataErrorAndGates(*data*, *error*, *gates*, *rotated=False*, *phases=None*, *midpoints=True*)
Sets the processed and gated data vector along with the gates (time discretization) and error cube.

Parameters

- **data** (*np.ndarray*) – Data vector of shape (number of pulses, number of gates). Expect complex valued vector.
- **error** (*np.ndarray*) – Error vector of the same shape as the data vector.
- **gates** (*np.ndarray*) – Simple time vector in seconds with shape matching the dimension 1 of the data and error vector. Expect gates without deadtime.
- **rotated** (*boolean [False]*) – Define whether the data are already rotated or not. thee is no autodetect for that.
- **phases** (*np.ndarray [None]*) – Define phases as simple vector containing phases in rad. Expect one value per pulse.
- **midpoints** (*boolean [True]*) – If True (default) the given times in the gates vector are interpreted as midpoint of gates. However if False the vector is interpreted as outer limits of the gates, so gate 1 would be defined between time 1 and time 2 and gate 2 between time 2 and 3 and so on.
- **Sets**
- **—-**
- **This functionality fills the following attributes**
- ***data_gated*, *gates*, *error_gated*, *rotated***
- **and optionally**
- ***phi* (phases)**

setGates(*gates*, *midpoints=True*)

Define time gates.

Parameters

- **gates** (*np.ndarray*) – Define gates midpoints. Expect array with float in [s]. See midpoints for definition of how the input array is interpreted.
- **midpoints** (*boolean [True]*) – If True (default) the given times in the gates vector are interpreted as midpoint of gates. However if False the vector is interpreted as outer limits of the gates, so gate 1 would be defined between time 1 and time 2 and gate 2 between timne 2 and 3 and so on.
- **Sets**
- **—-** – *gates* and *_gates_thk* if not the midpoints are given

setPhases(*phi*)

Sets variable phi. No check for length if vector is done. See setGatedDataErrorAndGates or setRawDataErrorAndTimes for more details.

setPulseDuration(*taup*, *deadtime_device=0.005*)

Sets pulse duration [s] and internal deadtime from the device.

Parameters

- **taup** (*float*) – Pulse duration in seconds.
- **deadtime_device** (*float [0.005]*) – Internal deadtime of the measurement device in seconds. 0.005 seconds are default for synthetic studies.
- **Sets**

- **—-**
- ***taup1*,**
- ***deadtime_device*,**
- ***deadtime* (half pulse + deadtime_device)**

setPulses(*pulses*)
Set pulse moment vector. Expect array with float in [As].

pulses

setRawDataErrorAndTimes(*data*, *error*, *times*, *rotated=False*, *phases=None*, *remove_df=True*, *omit_regating=False*)
Sets the raw (processed but ungated) data vector along with the time discretization and errorvector.

Parameters

- **data** (*np.ndarray*) – Data vector of shape (number of pulses, times). Expect complex valued vector.
- **error** (*np.ndarray*) – Error vector of the same shape as the data vector.
- **times** (*np.ndarray*) – Simple time vector in seconds with shape matching the dimension 1 of the data and error vector, expect times without deadtime!
- **rotated** (*boolean [False]*) – Define whether the data are already rotated or not. There is no autodetect for that.
- **phases** (*np.ndarray [None]*) – Define phases as simple vector containing phases in rad. Expect one value per pulse.
- **remove_df** (*boolean [True]*) – Removes the frequency offset in the given data stored in the attribute **df** [Hz].
- **omit_regating** (*boolean [False]*) – When setting the raw data, the gated data need to be recalculated. By default this is done via regating with the original settings for the gating.
- **Sets**
- **—-**
- **This functionality fills the following attributes**
- ***data_raw*, *times*, *error_raw*, *raw_rotated***
- **and optionally**
- ***phi* (phases)**

setResponse(*array*)
Sets a respinse array with the same shape as the data e.g. from an inversion instance. For plotting only.

setRotated(*rotated*, *raw_data=False*)
Sets rotation of data. True = rotatedAmplitudes, False = complex.

setRx(*index*, *turns=None*)
Define index of receiver loop and turns.

setTx(*index*, *turns=None*)
Define index of transmitter loop and turns.

temperature
Middle temperature [K]. Default = 281 K (8°C or 46.4°F).

times
Time vector [s] of raw data (including deadtime).

class comet.snmr.survey.survey.**Survey**(*earth=None*, *loops=None*)
Bases: object

Survey class for containment and handling of SNMR datasets (FIDS).

addLoop(*loop*)
Appends a given loop instance to the loops in survey and returns id

addSounding(*fid*)
Appends a given sounding instance to the sounds in survey and returns id

createKernel(*fid=0*, *dimension=1*)
Returns a initialized kernel instance for the chosen sounding.

Parameters

- **sound_index** (*integer*) – Index of the sounding the kernelclass is calcualting the kernel for. In order to calculate the kernel, pulses, tx and rx are taken as references from the sounding.
- **Note** (*createKernel does not set or change any values in survey nor in*)
- **the corresponding sounding. However when calculating, the kernel class**
- **will override the frequency in the given loops (tx and rx) and set it**
- **to the larmor frequency calculated from the earth magnetic fields**
- **magnitude. Use the *setEarth* method before or after you generate the**
- **kernel instances, but obviously before calculation.**

createMRS(*fid=0*, *kernel=None*, *mtype='smooth'*, *dtype='complex'*, *nlay=3*, *lam=1000*, *dimension=2*, ***kwargs*)

createSounding(*tx=0*, *rx=0*, *check_double=True*)
Creates a new sounding based on the given ids for tx and rx.

Parameters

- **tx** (*integer [0]*) – Index of the transmitter loop in loops.
- **rx** (*integer [0]*) – Index of the receiver loop in loops. Same number than tx indicates a coincident measurement.
- **check_double** (*boolean [True]*) – If True, omits creating another instance of the same fid (tx/rx combination). Instead the index of the original fid is returned. If False new fid is created and its index is returned.
- **Note** (*tx and rx indices can be setted regardless if there is an actual*)
- **loop in loops or just a *None* placeholder. In other words you can**
- **create your soundings and loops in arbitrary order.**

data
Complex data cube (pulses * gates) from soundings.

data_phases
Single data phases of the FIDs.

error
Complex error cube (pulses * gates) from soundings.

gates
Time gates gathered from soundings.

load (*savename*, *load_meshes=True*, *load_loops=True*)

loadLoopMesh (*savename*, *indices=None*, *dipolename=None*)
Loads mesh and distribute reference to given indices.

loadMRSD (*filename*, *remove_df=True*, *build_loops=False*, *x_offsets=None*, *segments=80*, *max_length=None*, *tx=None*, *rx=None*, *fids=None*, *debug=False*)

Parameters

- **filename** (*string*) – Path to .mrsd file to be imported.
- **build_loops** (*boolean [True]*) – If True, the saved config in the mrsd file is used to construct loops for transmitter and receiver. However, the information in the mrsd fiel is not complete. There are some defaults we assume in autogenerating the loops, especially when it comes to figure-of-eight loops. Feel free to replace the loops with custom created loops of the *pyhed* library. Or switch this off if you only want to see the data or define all the loops yourself.
- **x_offsets** (*list or None [None]*) – One information that is missing in mrsd files, is the relative position of the loops to each other. Here one can fill in this information giving a simple list of offsets in positive x direction (all loops (midpoints) are placed at y=0 and z=0). Expect one float per used loop by the data file or raises an error. Ignored if None and multiple loops are found (in this case no loops are build at all). Coincident measurements do not require this, x is set to 0 by default.
- **segments** (*integer [80]*) – Number of dipoles used to auto build the loops. Ignored if *build_loops* is False or not given any *x_offsets*.
- **max_length** (*float [None]*) – Maximum length of a dipole when auto generating the loops. Overrides segments. Ignored if *build_loops* is False or not given any *x_offsets*.

loadMRSD_h5 (*filename*, *remove_df=True*, *build_loops=False*, *x_offsets=None*, *segments=80*, *max_length=None*, *tx=None*, *rx=None*, *fids=None*, *debug=False*)
See loadMRSD instead.

loadMRSD_mat (*filename*, *remove_df=True*, *build_loops=False*, *x_offsets=None*, *segments=80*, *max_length=None*, *tx=None*, *rx=None*, *fids=None*, *debug=False*)
See loadMRSD instead.

loadMRSK (*filename*, *tx=None*, *rx=None*, *fid=None*, *set_earth=True*, *distribute_loop_config=False*, *x_offsets=None*, *segments=80*, *max_length=None*, *deadtime_device=0.005*, *min_thk=0*, *verbose=True*, *set_df=False*)

pulses
Pulse moment vectors gathered from soundings.

response
Complex data cube (pulses * gates) from soundings.

rx_indices
Indices of the used receiver of each sounding.

save (*savename*, *save_loops=True*, *use_original_loop_names=False*)

set1DModel (*thk=[], res=[1000.0]*)
Modifies loop config in terms of primary field resistivity.

setCustemConfig (*config*, *update_loop_configs=True*)

setEarth(*earth=None*, *incl=60.0*, *decl=2.0*, *mag=4.8e-05*, *rad=False*)
Defines the Earth in terms of inclination, declination and mag.

Parameters

- **earth** (*comet.snmr.survey.Earth [None]*) – Already initialized earth class will be setted. Or created through the other optional arguments.
- **inclination** (*float [60.]*) – Inclination of the earth magnetic field in rad or degree.
- **declination** (*float [2.]*) – Declination of the earth magnetic field in rad or degree.
- **magnitude** (*float [48000 * 1e-9]*) – Magnitude of the earth magnetic field in Tesla.
- **rad** (*boolean [False]*) – Input inclination and declination in rad?

setLoopConfig(*config*, *update_loop_configs=True*)
Loop config in terms of primary field resistivity and frequency.

setLoops(*loops*)

setResponse(*array*)
Set a response array from e.g. an inversion as data set for plotting.

tx_indices
Indices of the used transitter of each sounding.

used_loops

comet.snmr.survey.survey.**createLoopFromMRS**(*looptype*, *length*, *xoff*, *segments=80*, *max_length=None*, *turns=1*)
Returns a loop class object out of input found in a mrsd or mrsk file.

Parameters

- **looptype** (*integer*) – Integer in [1, 2, 3, 4], in this range representing circular, square, circular eight, and square eight loop source types. Error for looptype < 1 and > 4.
- **length** – Length [m] of one side of the loop, or loop diameter for cicular type.
- **xoff** (*float*) – Offset [m] for loop midpoint in positive x direction.
- **segments** (*integer [80]*) – Number of segments used for discretization of the loop wire.
- **max_length** (*integer [None]*) – If given, replaces the segments with a number suited to ensure each dipole represents this distance [m] at maximum.

Module contents

Module comet/snmr/survey

Module contents

Module comet/snmr

Module contents

overall COMET init file, if you want to import comet as one module.

2.5 Module Index

2.5.1 Indices and tables

- genindex
- modindex
- search

2.6 LICENSE

2.6.1 GNU GENERAL PUBLIC LICENSE

2.6.2 Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

2.6.3 TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
- b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
- c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

- d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
- c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
- e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

2.7 Authors

Corresponding author for **COMET**:

Nico Skibbe - nico.skibbe@leibniz-liag.de

Corresponding author for **custEM**:

Raphael Rochlitz - raphael.rochlitz@leibniz-liag.de

Additional support and contact for issues regarding **pyGIMLi**:

Thomas Günther - thomas.guenther@leibniz-liag.de

Bibliography

[Key2009G] Key, K., 2009, 1D inversion of multicomponent, multifrequency marine (CSEM) data: Methodology and synthetic studies for resolving thin resistive layers: Geophysics.

Python Module Index

Index