MTpy-v2 Documentation

Release 2.0.7

Jared Peacock

Apr 12, 2024

GENERAL INFORMATION

mtpy provides tools for working with magnetotelluric (MT) data. MTpy-v2 is an update version of [mtpy](https://github.com/MTgeophysics/mtpy). Many things have changed under the hood and usage is different from mtpy v1. The main difference is that there is a central data type that can hold transfer functions and then read/write to your modeling program, plot, and analyze your data. No longer will you need a directory of EDI files and then read them in everytime you want to do something. You only need to build a project once and save it to an MTH5 file and you are ready to go. All metadata uses [mt-metadata](https://github.com/kujaku11/mt-metadata).

Because the workflow has changed from mtpy v1, there are example notebooks to demonstrate the new workflow see *Examples*.

CHAPTER

ONE

EXAMPLES

Click on the *Binder* badge above to interact with Jupyter Notebook examples. There are example notebooks in

- **docs/source/examples/notebooks**

1.1 Installation

1.1.1 Stable release

PIP

To install *mt_metadata*, run this command in your terminal:

```
$ pip install mtpy-v2
```

This is the preferred method to install mt_metadata, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

Conda-Forge

To install *mtpy-v2*, run either of these commands in your Conda terminal (https://conda-forge.org/#about):

```
$ conda install -c conda-forge mtpy-v2
```

or

```
$ conda config --add channels conda-forge
$ conda config --set channel_priority strict
$ conda install mtpy-v2
```

Note: If you are updating *mt_metadata* you should use the same installer as your previous version or remove the current version and do a fresh install.

1.1.2 From sources

The sources for MTH5 can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone https://github.com/MTgeophysics/mtpy-v2
```

Or download the tarball:

```
$ curl -OJL https://github.com/MTgeophysics/mtpy-v2/tarball/main
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

1.2 Usage

1.2.1 MT Object

MT transfer functions come in all kinds of formats and flavors. The goal of `MT` is to centralize and standardize an MT transfer function with common metadata and accessibility to the data. `MT` inherits `` `mt_metadata.transfer_function.core.TF `` <https://mt-metadata.readthedocs.io/en/latest/source/tf_structure.html>`__ which has the ability to read/write in various file types. If there is a file type that is not supported yet raise an issue in mt-metadata.

Format	Description	Extension	Read	Write
EDI	Common SEG format	.edi	yes	yes
EMTF XML	Anna Kelbert's XML Format for archiving at IRIS	.xml	yes	yes
Z-Files	Output from Gary Egberts processing code	.zmm, .zss, .zrr	yes	yes
J-Files	Alan Jones' format and output of Alan Chave's BIRRP code	.j	yes	no
Zonge AVG	Zonge International processing code output	.avg	yes	no

The `MT` has a couple of important attributes and method that are described below as we progress through an example file. Here we will look at an EMTF XML because this format has the most comprehensive metadata so far.

[1]:
```
from mtpy import MT
from mt_metadata import TF_XML
```

[2]:
```
mt_object = MT(TF_XML)
mt_object.read()
```

TF Metadata

Important in describing the transfer function are metadata attributes, namely the location, what survey the station was collected in, timing, and how the data were processed. These are contained in logical metadata objects. For further reading on metadata objects see MT-metadata

- `MT.survey_metadata`: describes the general survey details that this transfer function belongs to.
- `MT.station_metadata`: describes the station location, timing, runs processed, processing scheme.
  - `MT.station_metadata.transfer_function`: describes how the data were processed.
  - `MT.station_metadata.runs`: provides details on the runs processed, timing, sample rate, channels recorded, data logger details.
    - `MT.station_metadata.runs[run_id].channels`: describes channel metadata including timing, sensors, location.

Survey Metadata

Survey metadata provides information about the survey ID, geographic locations, who aquired the data, is there a DOI associated with the data or publications, how the data can be used, licenses, and general information about the overall survey.

[3]:
```
mt_object.survey_metadata
```

[3]:

```
{
    "survey": {
        "acquired_by.author": "National Geoelectromagnetic Facility",
        "citation_dataset.authors": "Schultz, A., Pellerin, L., Bedrosian, P., Kelbert,␣
↪A., Crosbie, J.",
        "citation_dataset.doi": "doi:10.17611/DP/EMTF/USMTARRAY/SOUTH",
        "citation_dataset.title": "USMTArray South Magnetotelluric Transfer Functions",
        "citation_dataset.year": "2020-2023",
        "citation_journal.doi": null,
        "comments": "copyright.acknowledgement:The USMTArray-CONUS South campaign was␣
↪carried out through a cooperative agreement between\nthe U.S. Geological Survey (USGS)␣
↪and Oregon State University (OSU). A subset of 40 stations\nin the SW US were funded␣
↪through NASA grant 80NSSC19K0232.\nLand permitting, data acquisition, quality control␣
↪and field processing were\ncarried out by Green Geophysics with project management and␣
↪instrument/engineering\nsupport from OSU and Chaytus Engineering, respectively.\
↪nProgram oversight, definitive data processing and data archiving were provided\nby␣
↪the USGS Geomagnetism Program and the Geology, Geophysics and Geochemistry Science␣
↪Centers.\nWe thank the U.S. Forest Service, the Bureau of Land Management, the␣
↪National Park Service,\nthe Department of Defense, numerous state land offices and the␣
↪many private landowners\nwho permitted land access to acquire the USMTArray data.;␣
↪copyright.conditions_of_use:All data and metadata for this survey are available free␣
↪of charge and may be copied freely, duplicated and further distributed provided that␣
↪this data set is cited as the reference, and that the author(s) contributions are␣
↪acknowledged as detailed in the Acknowledgements. Any papers cited in this file are␣
↪only for reference. There is no requirement to cite these papers when the data are␣
↪used. Whenever possible, we ask that the author(s) are notified prior to any␣
↪publication that makes use of these data.\n While the author(s) strive to provide data␣
↪and metadata of best possible quality, neither the author(s) of this data set, nor␣
↪IRIS make any claims, promises, or guarantees about the accuracy, completeness, or␣
```


(continues on next page)

(continued from previous page)


```
↪adequacy of this information, and expressly disclaim liability for errors and␣
↪omissions in the contents of this file. Guidelines about the quality or limitations of␣
↪the data and metadata, as obtained from the author(s), are included for informational␣
↪purposes only.; copyright.release_status:Unrestricted Release",
        "country": [
            "USA"
        ],
        "datum": "WGS84",
        "geographic_name": "CONUS South",
        "id": "CONUS South",
        "name": null,
        "northwest_corner.latitude": 34.470528,
        "northwest_corner.longitude": -108.712288,
        "project": "USMTArray",
        "project_lead.email": null,
        "project_lead.organization": null,
        "release_license": "CC0-1.0",
        "southeast_corner.latitude": 34.470528,
        "southeast_corner.longitude": -108.712288,
        "summary": "Magnetotelluric Transfer Functions",
        "time_period.end_date": "2020-10-07",
        "time_period.start_date": "2020-09-20"
    }
}
```


Station Metadata

Station metadata is the most important to describe the transfer function, it provides ID, location, timing and then specifics on how the data were processed, run metadata, and channel metadata.

[4]:
```
mt_object.station_metadata
```

[4]:

```
{
    "station": {
        "acquired_by.author": "National Geoelectromagnetic Facility",
        "channels_recorded": [
            "ex",
            "ey",
            "hx",
            "hy",
            "hz"
        ],
        "comments": "description:Magnetotelluric Transfer Functions; primary_data.
↪filename:NMX20b_NMX20_NMW20_COR21_NMY21-NMX20b_NMX20_UTS18.png; attachment.description:
↪The original used to produce the XML; attachment.filename:NMX20b_NMX20_NMW20_COR21_
↪NMY21-NMX20b_NMX20_UTS18.zmm; site.data_quality_notes.comments.author:Jade Crosbie,␣
↪Paul Bedrosian and Anna Kelbert; site.data_quality_notes.comments.value:great TF from␣
↪10 to 10000 secs (or longer)",
        "data_type": "mt",
        "fdsn.id": "USMTArray.NMX20.2020",
        "geographic_name": "Nations Draw, NM, USA",
```


(continues on next page)

(continued from previous page)


```
        "id": "NMX20",
        "location.datum": "WGS84",
        "location.declination.epoch": "2020.0",
        "location.declination.model": "WMM",
        "location.declination.value": 9.09,
        "location.elevation": 1940.05,
        "location.latitude": 34.470528,
        "location.longitude": -108.712288,
        "orientation.angle_to_geographic_north": 0.0,
        "orientation.method": null,
        "orientation.reference_frame": "geographic",
        "provenance.archive.comments": "IRIS DMC MetaData",
        "provenance.archive.name": null,
        "provenance.archive.url": "http://www.iris.edu/mda/ZU/NMX20",
        "provenance.creation_time": "2021-03-17T14:47:44+00:00",
        "provenance.creator.author": "Jade Crosbie, Paul Bedrosian and Anna Kelbert",
        "provenance.creator.email": "pbedrosian@usgs.gov",
        "provenance.creator.name": "Jade Crosbie, Paul Bedrosian and Anna Kelbert",
        "provenance.creator.organization": "U.S. Geological Survey",
        "provenance.creator.url": "https://www.usgs.gov/natural-hazards/geomagnetism",
        "provenance.software.author": null,
        "provenance.software.name": "EMTF File Conversion Utilities 4.0",
        "provenance.software.version": null,
        "provenance.submitter.author": "Anna Kelbert",
        "provenance.submitter.email": "akelbert@usgs.gov",
        "provenance.submitter.name": "Anna Kelbert",
        "provenance.submitter.organization": "U.S. Geological Survey, Geomagnetism␣
↪Program",
        "provenance.submitter.url": "https://www.usgs.gov/natural-hazards/geomagnetism",
        "release_license": "CC0-1.0",
        "run_list": [
            "NMX20a",
            "NMX20b"
        ],
        "time_period.end": "2020-10-07T20:28:00+00:00",
        "time_period.start": "2020-09-20T19:03:06+00:00",
        "transfer_function.coordinate_system": "geopgraphic",
        "transfer_function.data_quality.good_from_period": 5.0,
        "transfer_function.data_quality.good_to_period": 29127.0,
        "transfer_function.data_quality.rating.value": 5,
        "transfer_function.id": "NMX20",
        "transfer_function.processed_by.author": "Jade Crosbie, Paul Bedrosian and Anna␣
↪Kelbert",
        "transfer_function.processed_by.name": "Jade Crosbie, Paul Bedrosian and Anna␣
↪Kelbert",
        "transfer_function.processed_date": "1980-01-01",
        "transfer_function.processing_parameters": [],
        "transfer_function.processing_type": "Robust Multi-Station Reference",
        "transfer_function.remote_references": [
            "NMW20",
            "COR21",
            "UTS18"
```


(continues on next page)

(continued from previous page)


```
        ],
        "transfer_function.runs_processed": [
            "NMX20a",
            "NMX20b"
        ],
        "transfer_function.sign_convention": "exp(+ i\\omega t)",
        "transfer_function.software.author": "Gary Egbert",
        "transfer_function.software.last_updated": "2015-08-26",
        "transfer_function.software.name": "EMTF",
        "transfer_function.software.version": null,
        "transfer_function.units": null
    }
}
```


Run Metadata

Run metadata is located in `MT.station_metadata.runs` which is a list-dictionary object that contains the runs used for processing.

```
[5]: mt_object.station_metadata.runs
```


```
[5]: OrderedDict([('NMX20a', {
    "run": {
        "channels_recorded_auxiliary": [],
        "channels_recorded_electric": [
            "ex",
            "ey"
        ],
        "channels_recorded_magnetic": [
            "hx",
            "hy",
            "hz"
        ],
        "comments": "comments.author:Isaac Sageman; comments.value:X array at 0 deg␣
↪rotation. All e-lines 50m. Soft sandy dirt. Water tank ~400m NE. County Rd 601 ~200m␣
↪SE. Warm sunny day.",
        "data_logger.firmware.author": null,
        "data_logger.firmware.name": null,
        "data_logger.firmware.version": null,
        "data_logger.id": "2612-01",
        "data_logger.manufacturer": "Barry Narod",
        "data_logger.timing_system.drift": 0.0,
        "data_logger.timing_system.type": "GPS",
        "data_logger.timing_system.uncertainty": 0.0,
        "data_logger.type": "NIMS",
        "data_type": "BBMT",
        "id": "NMX20a",
        "sample_rate": 1.0,
        "time_period.end": "2020-09-20T19:29:28+00:00",
        "time_period.start": "2020-09-20T19:03:06+00:00"
    }
```


(continues on next page)

(continued from previous page)

```
}), ('NMX20b', {
    "run": {
        "channels_recorded_auxiliary": [],
        "channels_recorded_electric": [
            "ex",
            "ey"
        ],
        "channels_recorded_magnetic": [
            "hx",
            "hy",
            "hz"
        ],
        "comments": "comments.author:Isaac Sageman; comments.value:X array at 0 deg␣
↪rotation. All e-lines 50m. Soft sandy dirt. Water tank ~400m NE. County Rd 601 ~200m␣
↪SE. Warm sunny day.; errors:Found data gaps (2). Gaps of unknown length: 1 [1469160].]
↪",
        "data_logger.firmware.author": null,
        "data_logger.firmware.name": null,
        "data_logger.firmware.version": null,
        "data_logger.id": "2612-01",
        "data_logger.manufacturer": "Barry Narod",
        "data_logger.timing_system.drift": 0.0,
        "data_logger.timing_system.type": "GPS",
        "data_logger.timing_system.uncertainty": 0.0,
        "data_logger.type": "NIMS",
        "data_type": "BBMT",
        "id": "NMX20b",
        "sample_rate": 1.0,
        "time_period.end": "2020-10-07T20:28:00+00:00",
        "time_period.start": "2020-09-20T20:12:29+00:00"
    }
})])
```

To access a single run you can use either the index of the run or the `run.id`

```
[6]: mt_object.station_metadata.runs[0]
```

```
[6]: {
    "run": {
        "channels_recorded_auxiliary": [],
        "channels_recorded_electric": [
            "ex",
            "ey"
        ],
        "channels_recorded_magnetic": [
            "hx",
            "hy",
            "hz"
        ],
        "comments": "comments.author:Isaac Sageman; comments.value:X array at 0 deg␣
↪rotation. All e-lines 50m. Soft sandy dirt. Water tank ~400m NE. County Rd 601 ~200m␣
↪SE. Warm sunny day.",
        "data_logger.firmware.author": null,
```

(continues on next page)

(continued from previous page)


```
        "data_logger.firmware.name": null,
        "data_logger.firmware.version": null,
        "data_logger.id": "2612-01",
        "data_logger.manufacturer": "Barry Narod",
        "data_logger.timing_system.drift": 0.0,
        "data_logger.timing_system.type": "GPS",
        "data_logger.timing_system.uncertainty": 0.0,
        "data_logger.type": "NIMS",
        "data_type": "BBMT",
        "id": "NMX20a",
        "sample_rate": 1.0,
        "time_period.end": "2020-09-20T19:29:28+00:00",
        "time_period.start": "2020-09-20T19:03:06+00:00"
    }
}
```


[7]:
```
mt_object.station_metadata.runs["NMX20b"]
```

[7]:

```
{
    "run": {
        "channels_recorded_auxiliary": [],
        "channels_recorded_electric": [
            "ex",
            "ey"
        ],
        "channels_recorded_magnetic": [
            "hx",
            "hy",
            "hz"
        ],
        "comments": "comments.author:Isaac Sageman; comments.value:X array at 0 deg␣
↪rotation. All e-lines 50m. Soft sandy dirt. Water tank ~400m NE. County Rd 601 ~200m␣
↪SE. Warm sunny day.; errors:Found data gaps (2). Gaps of unknown length: 1 [1469160].]
↪",
        "data_logger.firmware.author": null,
        "data_logger.firmware.name": null,
        "data_logger.firmware.version": null,
        "data_logger.id": "2612-01",
        "data_logger.manufacturer": "Barry Narod",
        "data_logger.timing_system.drift": 0.0,
        "data_logger.timing_system.type": "GPS",
        "data_logger.timing_system.uncertainty": 0.0,
        "data_logger.type": "NIMS",
        "data_type": "BBMT",
        "id": "NMX20b",
        "sample_rate": 1.0,
        "time_period.end": "2020-10-07T20:28:00+00:00",
        "time_period.start": "2020-09-20T20:12:29+00:00"
    }
}
```

Channel Metadata

Channel metadata is important because it describes orientation, location, sensors of each channel. These are accessed through the run. Similar to the runs direct access can be through the index or component name.

[8]:
```
mt_object.station_metadata.runs[0].channels[0]
```

[8]:

```
{
    "magnetic": {
        "channel_number": 0,
        "component": "hx",
        "data_quality.rating.value": 0,
        "filter.applied": [
            false
        ],
        "filter.name": [],
        "location.elevation": 0.0,
        "location.latitude": 0.0,
        "location.longitude": 0.0,
        "location.x": 0.0,
        "location.y": 0.0,
        "location.z": 0.0,
        "measurement_azimuth": 9.1,
        "measurement_tilt": 0.0,
        "sample_rate": 0.0,
        "sensor.id": "2509-23",
        "sensor.manufacturer": "Barry Narod",
        "sensor.name": "NIMS",
        "sensor.type": "fluxgate",
        "time_period.end": "2020-09-20T19:29:28+00:00",
        "time_period.start": "2020-09-20T19:03:06+00:00",
        "translated_azimuth": 9.1,
        "type": "magnetic",
        "units": null
    }
}
```


[9]:
```
mt_object.station_metadata.runs[0].channels["hy"]
```

[9]:

```
{
    "magnetic": {
        "channel_number": 0,
        "component": "hy",
        "data_quality.rating.value": 0,
        "filter.applied": [
            false
        ],
        "filter.name": [],
        "location.elevation": 0.0,
        "location.latitude": 0.0,
        "location.longitude": 0.0,
        "location.x": 0.0,
        "location.y": 0.0,
        "location.z": 0.0,
```


(continues on next page)

(continued from previous page)


```
        "measurement_azimuth": 99.1,
        "measurement_tilt": 0.0,
        "sample_rate": 0.0,
        "sensor.id": "2509-23",
        "sensor.manufacturer": "Barry Narod",
        "sensor.name": "NIMS",
        "sensor.type": "fluxgate",
        "time_period.end": "2020-09-20T19:29:28+00:00",
        "time_period.start": "2020-09-20T19:03:06+00:00",
        "translated_azimuth": 99.1,
        "type": "magnetic",
        "units": null
    }
}
```


Or you can access the channel metadata through a convenience attribute

[10]:
```
mt_object.ex_metadata
```

[10]:

```
{
    "electric": {
        "channel_number": 0,
        "component": "ex",
        "data_quality.rating.value": 0,
        "dipole_length": 100.0,
        "filter.applied": [
            false
        ],
        "filter.name": [],
        "measurement_azimuth": 9.1,
        "measurement_tilt": 0.0,
        "negative.elevation": 0.0,
        "negative.id": "40201037",
        "negative.latitude": 0.0,
        "negative.longitude": 0.0,
        "negative.manufacturer": "Oregon State University",
        "negative.type": "Pb-PbCl2 kaolin gel Petiau 2 chamber type",
        "negative.x": -50.0,
        "negative.y": 0.0,
        "negative.z": 0.0,
        "positive.elevation": 0.0,
        "positive.id": "40201038",
        "positive.latitude": 0.0,
        "positive.longitude": 0.0,
        "positive.manufacturer": "Oregon State University",
        "positive.type": "Pb-PbCl2 kaolin gel Petiau 2 chamber type",
        "positive.x2": 50.0,
        "positive.y2": 0.0,
        "positive.z2": 0.0,
        "sample_rate": 0.0,
        "time_period.end": "2020-09-20T19:29:28+00:00",
        "time_period.start": "2020-09-20T19:03:06+00:00",
        "translated_azimuth": 9.1,
```


(continues on next page)

(continued from previous page)


```
        "type": "electric",
        "units": null
    }
}
```


Metadata Summary

Metadata is important to keep track of and can be cumbersome, but helps future users of your data to actually use your data. There are a lot of fields here but the most important are ID, location, and timing.

- mt.survey_metadata.id
- mt.station_metadata.id
- mt.station_metadata.location.longitude
- mt.station_metadata.location.latitude
- mt.station_metadata.location.elevation
- mt.station_metadata.time_period.start
- mt.station_metadata.time_period.end
- mt.station_metadata.transfer_function.processing.parameters
- mt.station_metadata.runs[0].channels[component].measurement_azimuth
- mt.station_metadata.runs[0].channels[component].measurement_tilt
- mt.station_metadata.runs[0].channels[component].translated_azimuth
- mt.station_metadata.runs[0].channels[component].translated_tilt

Data

The most interesting part about a transfer function is the data. This describes how the Earth response to inducing magnetic fields. Under the hood an MT object stores a generic transfer function that has input channels (sources) and output channels (reponses) as an xarray.

The benefit of using xarray is that it has tools for combining various statistical estimates and naturally indexes across a common index. In this case we can index along period for each of the statistical estimates (transfer function, errors, and covariances). The `mt._transfer_function` object is a `xarray.Dataset` and each statistical estimate is an `xarray.DataArray`.

The other benefit is that attributes can be store directly alongside the arrays for a self describing object. Here we have picked the most important attributes from the metadata to describe the transfer function. These are propogated with each statistical estimate so anytime you retreive `impedance` or `tipper` the attributes are in the xarray.

Note: You'll see below that input channels and output channels have the same components, that's because in order to contain the full transfer function and covariances we need a symmetric matrix. Those components that don't have data are filled with NaNs, which is find for transfer functions because they are relatively small and not memory intensive.

Generic Transfer Function

[11]:
```
mt_object._transfer_function
```

[11]:
```
<xarray.Dataset>
Dimensions:                           (output: 5, input: 5, period: 33)
Coordinates:
  * output                            (output) <U2 'ex' 'ey' 'hx' 'hy' 'hz'
  * input                             (input) <U2 'ex' 'ey' 'hx' 'hy' 'hz'
  * period                            (period) float64 4.655 5.818 ... 2.913e+04
Data variables:
    transfer_function                 (period, output, input) complex128 (nan+na...
    transfer_function_error           (period, output, input) float64 nan ... nan
    transfer_function_model_error     (period, output, input) float64 nan ... nan
    inverse_signal_power              (period, output, input) complex128 (nan+na...
    residual_covariance               (period, output, input) complex128 (0.0012...
Attributes: (12/14)
    survey:                CONUS South
    project:               USMTArray
    id:                    NMX20
    name:                  Nations Draw, NM, USA
    latitude:              34.470528
    longitude:             -108.712288
    ...                    ...
    datum:                 WGS84
    acquired_by:           National Geoelectromagnetic Facility
    start:                 2020-09-20T19:03:06+00:00
    end:                   2020-10-07T20:28:00+00:00
    runs_processed:        ['NMX20a', 'NMX20b']
    coordinate_system:     geographic
```

Errors and Covariances

The generic transfer function xarray also contains the covariances (if provided) and errors of the measured data and also has a place for model errors.

Note: If the tranfer function contains `inverse_signal_power` and `residual_covariance` then the `transfer_function_error` is estimated from these values, otherwise the error from the data file is used.

[12]:
```
mt_object._transfer_function.data_vars
```

[12]:
```
Data variables:
    transfer_function                 (period, output, input) complex128 (nan+na...
    transfer_function_error           (period, output, input) float64 nan ... nan
    transfer_function_model_error     (period, output, input) float64 nan ... nan
    inverse_signal_power              (period, output, input) complex128 (nan+na...
    residual_covariance               (period, output, input) complex128 (0.0012...
```

MTpy.core.transfer_function.Base

Each of the transfer function estimates are represented by a base class called `mtpy.core.tranfer_function.Base` object which has methods like:

- `inverse` returns the inverse of the transfer function
- `rotate` rotates the transfer function positive clockwise
- `interpolate` interpolates onto a different period index
- `to_xarray` returns an `xarray.DataArray`
- `from_xarray` ingests an `xarray.DataArray`
- `to_dataframe` returns a `pandas.DataFrame` representation of the transfer function indexed by period.
- `from_dataframe` ingests a `pandas.DataFrame`
- `copy` return a copy of the transfer function object
- contains validation methods for inputs

It also has attributes:

- `period` period array
- `frequency` 1/period array
- `comps` components in the transfer function as input channels and output channels
- `n_periods` number of periods

Impedance

The impedance $\hat{\mathbf{Z}}$ describes how the Earth electrically responds to horizontal magnetic fields is formed from this generic transfer function.

$\tilde{\mathbf{E}}(\omega) = \hat{\mathbf{Z}}(\omega)\tilde{\mathbf{H}}(\omega)$

To $\hat{\mathbf{Z}}$ is built from the generic transfer function using the horizontal components of the input and output channels. There are two ways to access the impedance from the `MT` object

- `mt.impedance` will return an xarray of the impedance
- `mt.impedance_error` will return the errors in the impedance measurement
- `mt.impedance_model_error` will return model errors for the impedance
- `mt.Z` will return a `mtpy.core.transfer_function.Z` object which has methods for accessing impedance tensor attributes, which contains the errors.

If you are not sure the transfer function has impedance you can use the method:

```
[13]: mt_object.has_impedance()
```

```
[13]: True
```

MT.impedance

This is an xarray of the impedance and is computed from the generic transfer function using the horizontal components of the input and output channels.

[14]:
```
mt_object.impedance
```

[14]:
```
<xarray.DataArray 'impedance' (period: 33, output: 2, input: 2)>
array([[[-1.160949e-01-0.2708645j ,  3.143284e+00+1.101737j  ],
        [-2.470717e+00-0.7784633j , -1.057851e-01+0.1022045j ]],

       [[-1.051846e-01-0.1912665j ,  3.169108e+00+1.007867j  ],
        [-2.459892e+00-0.8541335j , -1.325974e-01+0.1473665j ]],

       [[-1.289586e-02-0.1937956j ,  3.064653e+00+1.063899j  ],
        [-2.446347e+00-0.8661013j , -1.222841e-01+0.1580956j ]],

       [[-8.208073e-02-0.3117874j ,  3.042922e+00+1.006518j  ],
        [-2.310078e+00-0.8732821j , -1.620193e-01+0.2110874j ]],

       [[ 3.353187e-02-0.2855585j ,  2.875270e+00+1.083887j  ],
        [-2.135730e+00-0.8666129j , -1.881319e-01+0.2420585j ]],

       [[ 1.014077e-01-0.2989493j ,  2.719818e+00+1.103022j  ],
        [-2.073182e+00-0.8841784j , -3.048077e-01+0.2344493j ]],

       [[ 2.441069e-01-0.2828671j ,  2.493647e+00+1.179619j  ],
        [-1.956353e+00-1.022081j  , -3.447069e-01+0.1793671j ]],
...
       [[ 3.475757e-02+0.0352555j ,  1.770043e-01+0.2258684j ],
        [-1.343957e-01-0.1432316j , -4.149757e-02-0.05223549j]],

       [[ 2.719606e-02+0.03381913j,  1.438796e-01+0.1859595j ],
        [-1.121204e-01-0.1246405j , -3.594606e-02-0.04361913j]],

       [[ 2.161943e-02+0.02911898j,  1.137554e-01+0.150974j  ],
        [-8.744459e-02-0.103626j  , -2.562943e-02-0.03531898j]],

       [[ 1.514583e-02+0.02224749j,  8.349043e-02+0.1184608j ],
        [-6.363958e-02-0.08284923j, -2.077583e-02-0.02673749j]],

       [[ 9.760046e-03+0.01694262j,  5.923743e-02+0.08950258j],
        [-4.535258e-02-0.06544742j, -1.464005e-02-0.02036262j]],

       [[ 1.027061e-02+0.01265869j,  4.057636e-02+0.0674322j ],
        [-3.114365e-02-0.0496578j , -1.030061e-02-0.01919869j]],

       [[ 4.834623e-03+0.00983358j,  2.643963e-02+0.05098311j],
        [-2.203037e-02-0.03744689j, -2.953623e-03-0.01293358j]]])
Coordinates:
  * output   (output) <U2 'ex' 'ey'
  * input    (input) <U2 'hx' 'hy'
  * period   (period) float64 4.655 5.818 7.314 ... 1.872e+04 2.913e+04
Attributes:
```

(continues on next page)

(continued from previous page)

```
    survey:             CONUS South
    project:            USMTArray
    id:                 NMX20
    name:               Nations Draw, NM, USA
    latitude:           34.470528
    longitude:          -108.712288
    elevation:          1940.05
    declination:        9.09
    datum:              WGS84
    acquired_by:        National Geoelectromagnetic Facility
    start:              2020-09-20T19:03:06+00:00
    end:                2020-10-07T20:28:00+00:00
    runs_processed:     ['NMX20a', 'NMX20b']
    coordinate_system:  geographic
```

MT.Z

The Z object is central to most actions in `mtpy` and has various attributes that are important to analyzing and visualizing the impedance tensor. Of these are:

- `resistivity` and `phase` are the most common way to represent the impedance tensor. The `resistivity` is an apparent resistivity that describes the volumetric average of the Earth sampled by hemisphere with a radius related to the period through the skin depth. It provides an estimate on how conductive or resistive the subsurface is as a function of period, a proxy for depth. The `phase` is the impedance phase and describes how subsurface resistivity is changing where 45 degrees indicates no change.
  - `res_ij` and `phase_ij` are convenient attributes for accessing only a certain component of the resistivity or phase. If you want to access just the xy component use `res_xy`.
  - `res_det` and `phase_det` represent the resistivity and phase of the determinant of the impedance tensor.
- `phase_tensor` is a different way of representing the impedance tensor that is not effected by near surface distortion like static shift. This returns a `mtpy.core.tranfer function.Base` object.

[15]:
```
mt_object.Z
```

[15]:
```
Transfer Function impedance
------------------------------
        Number of periods:  33
        Frequency range:        3.43323E-05 -- 2.14844E-01 Hz
        Period range:           4.65455E+00 -- 2.91271E+04 s

        Has impedance:              True
        Has impedance_error:        True
        Has impedance_model_error:  False
```

[16]:
```
mt_object.Z.res_xy
```

[16]:
```
array([10.3275702 , 12.8686987 , 15.39508701, 18.78391953, 21.9740757 ,
       25.94353835, 29.97081497, 33.72107947, 37.49484018, 39.85170164,
       42.22309955, 44.22664358, 46.57708096, 47.53701478, 48.98712171,
       50.77007139, 52.33463866, 52.41671744, 51.90545285, 51.26697937,
       49.76791648, 46.62343578, 45.61307418, 43.08625057, 40.82330885,
```

(continues on next page)

(continued from previous page)

```
       39.53221291, 37.21859669, 34.50456458, 33.45466823, 30.58872947,
       27.45312204, 23.19428435, 19.21417312])
```

[17]:
```
mt_object.Z.phase_tensor
```

[17]:
```
Transfer Function phase_tensor
------------------------------
        Number of periods:  33
        Frequency range:        3.43323E-05 -- 2.14844E-01 Hz
        Period range:           4.65455E+00 -- 2.91271E+04 s

        Has phase_tensor:              True
        Has phase_tensor_error:        True
        Has phase_tensor_model_error:  False
```

MT.Tipper

The tipper represents the induction vectors or the Earth's magnetic response to horizontal magnetic fields. You get strong induction vectors when horizontal electrical currents flow through the subsurface like along structural boundaries and faults.

$H_z(\omega) = \hat{\mathbf{W}}(\omega)\tilde{\mathbf{H}}(\omega)$

Similar to `MT.Z` there are methods and attributes for analyzing induction vectors.

- `amplitude` amplitude of the induction vectors
- `phase` phase of the induction vectors
- `angle` horizontal direction of the induction vectors
- `mag` magnitude of the induction vectors

There are also attributes for the real and imaginary components

- `angle_real` and `angle_imag`
- `mag_real` and `mag_imag`

[18]:
```
mt_object.Tipper.mag_real
```

[18]:
```
array([0.10454066, 0.07782623, 0.08104679, 0.08278517, 0.09892832,
       0.12609043, 0.09750211, 0.07730022, 0.06717719, 0.05336503,
       0.04318178, 0.05157183, 0.07753425, 0.10306461, 0.14128244,
       0.17552209, 0.1995196 , 0.21798128, 0.22319936, 0.22839953,
       0.22834729, 0.22250223, 0.21076025, 0.19728602, 0.17894005,
       0.16590773, 0.14505057, 0.10009294, 0.06102891, 0.06275663,
       0.03127006, 0.14861124, 0.17879201])
```

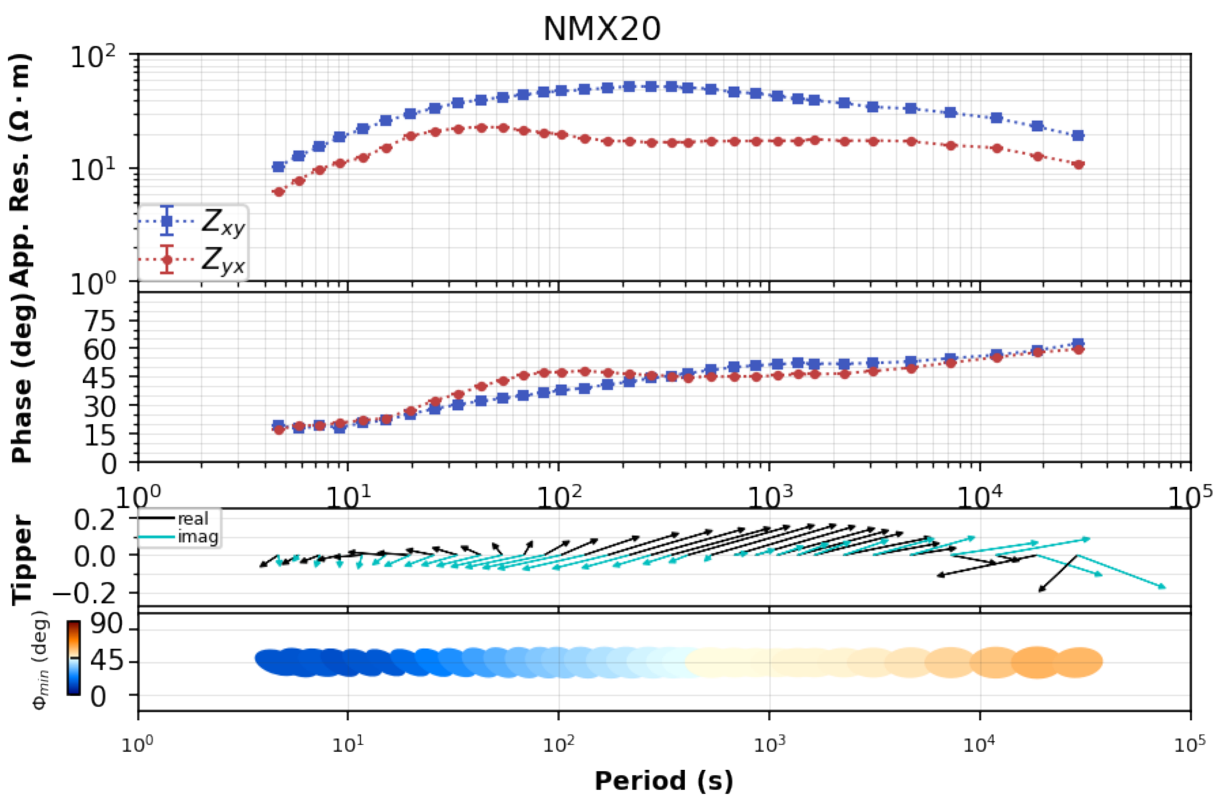
Visualizing

Have a look at the data provides better information than just looking at arrays. There are a few methods for plotting various components of the impedance tensor.

Plot MT Response

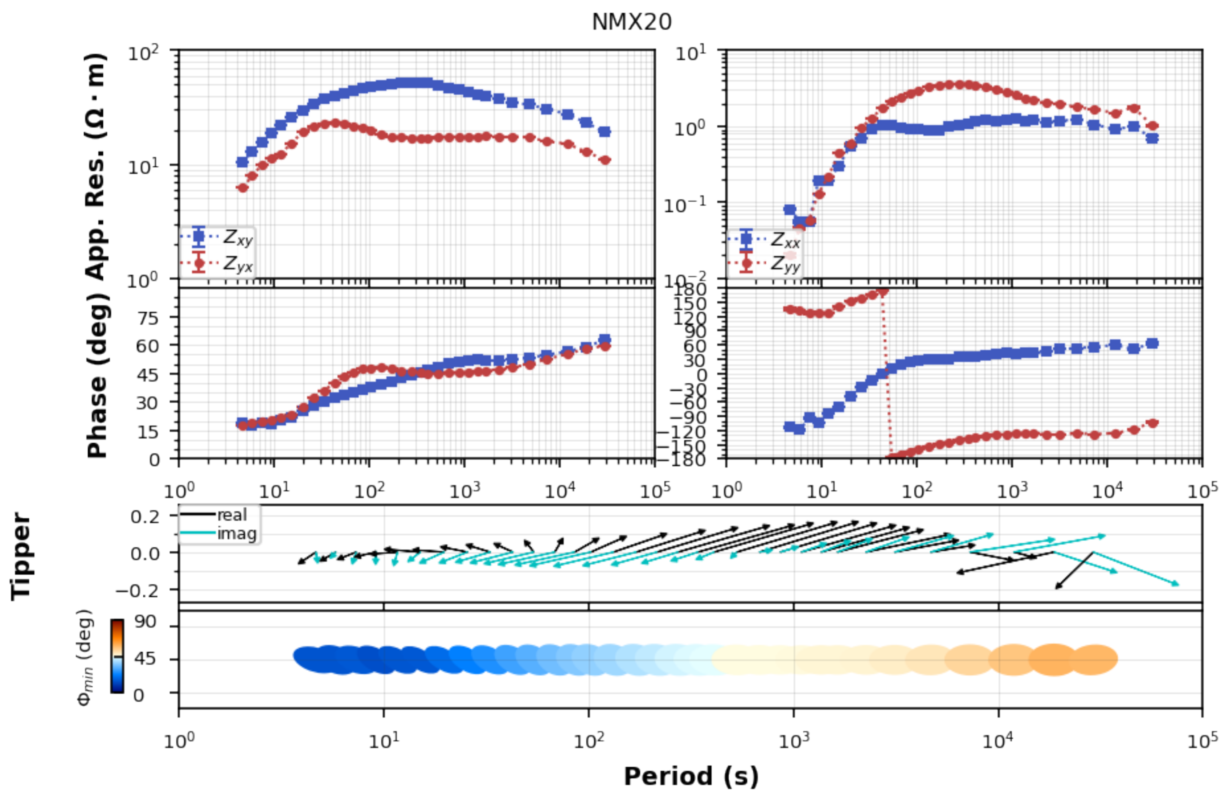
The main visualization is plotting the MT response as apparent resistivity and phase, and if there are induction vectors plot them. This also plots the phase tensor ellipses for completeness. You can turn them on and off if you like.

[19]:
```
plot_response = mt_object.plot_mt_response()
```



You can plot all 4 components of the impedance tensor

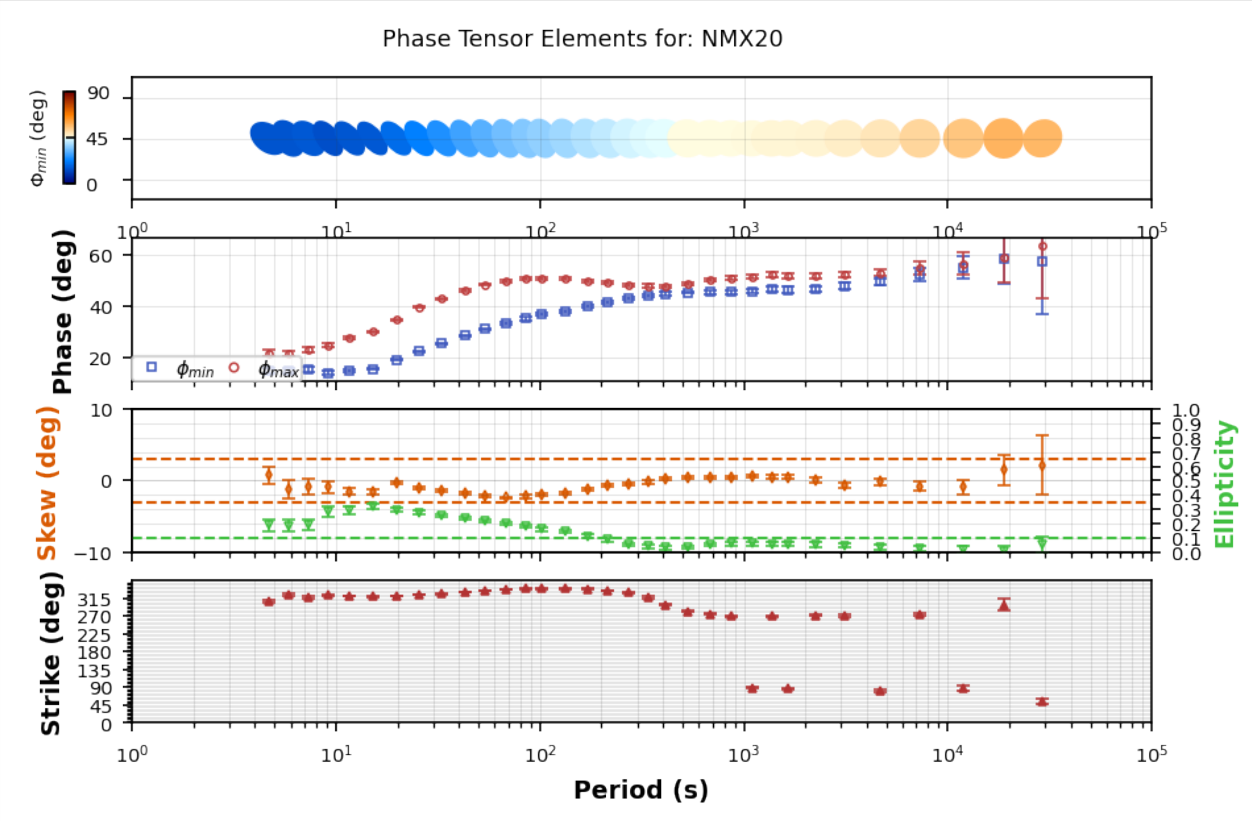
[20]:
```
plot_response.plot_num = 2
plot_response.fig_num = 2
plot_response.plot()
```



Plot Phase Tensor component

Sometimes it can be infomrative to plot the phase tensor components and attributes to determine dimensionality.

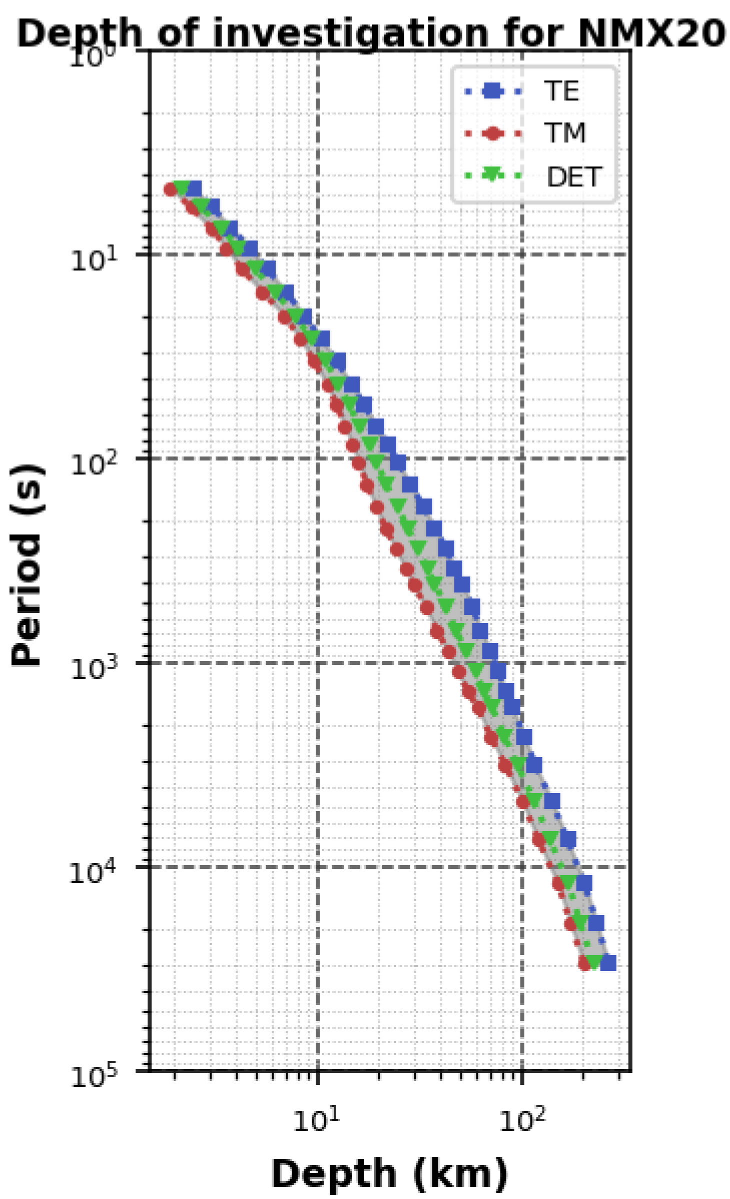
[21]:
```
plot_pt = mt_object.plot_phase_tensor()
```



Plot Penetration Depth

Another diagnostic of your data is to estimate the depth of penetration. This is done through a Niblett-Bostick transformation.

[22]:
```
plot_nb = mt_object.plot_depth_of_penetration()
```



Manipulating MT Response

Often you want to manipulate the transfer function by applying a static shift, rotationg, flipping phases, interpolating. There are tools for this.

Rotate

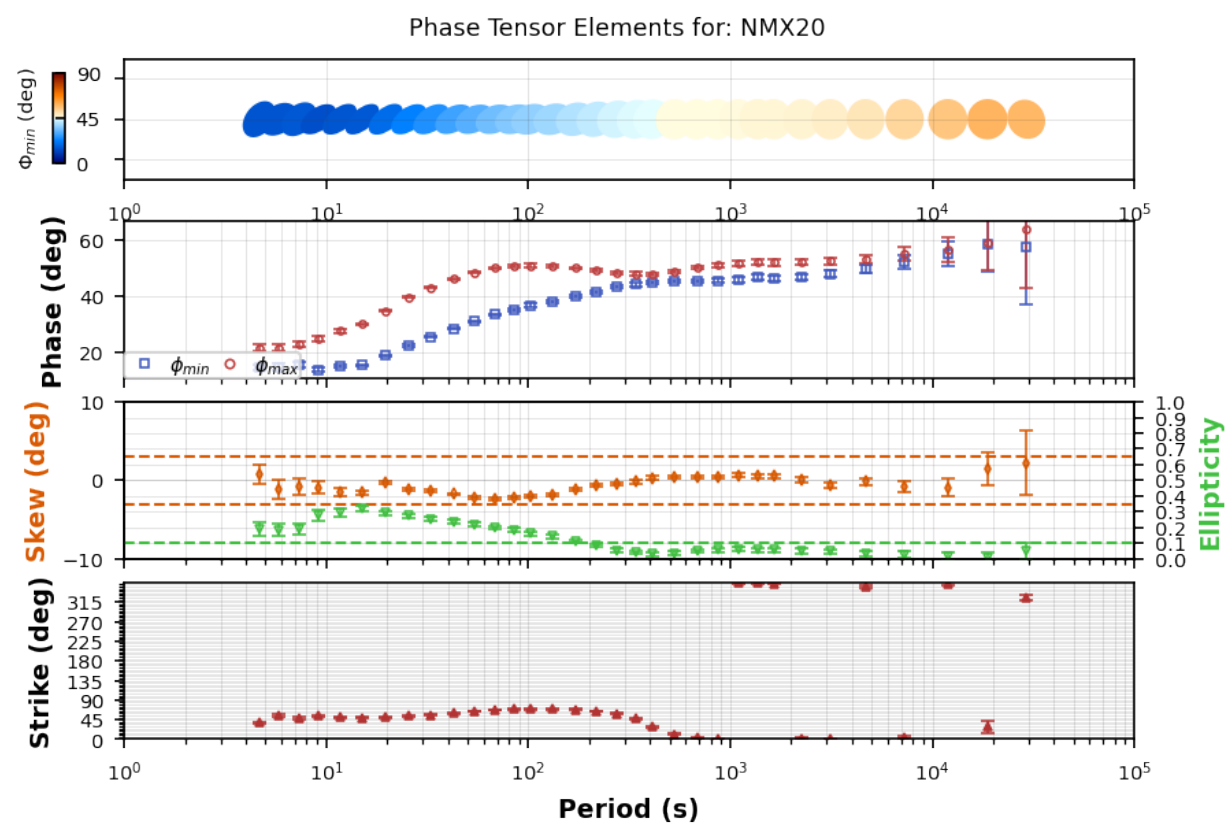
Data rotation helps align the data with geologic structures and optimzing the transfer function to be quasi 2D for modeling.

Geoelectric strike can be estimated from the phase tensor as seen in the plot above. Here this station appears to have a dominant geoelectric strike within the 2D realm is N270E.

Note: All rotations are clockwise assuming that geographic north = 0, and east is 90.

[23]:
```
rotated = mt_object.rotate(-270, inplace=False)
rotated.plot_phase_tensor()
```

[23]:
```
Plotting PlotPhaseTensor
```



Interpolate

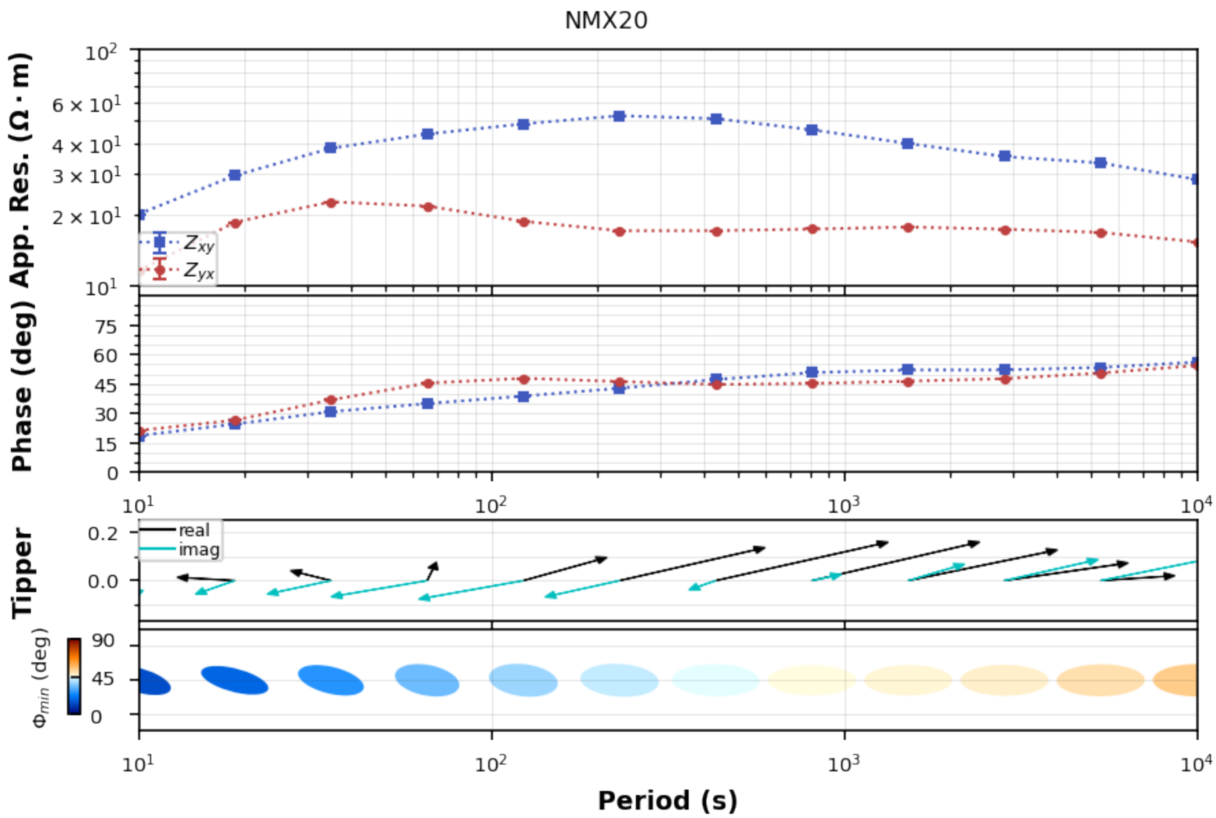
Interpolating data is useful for standardizing a data set for modeling or comparing. Interpolation is done on each component on both real and imaginary parts using a linear interpolation using `scipy.signal.interp1d` internally within `xarray`.

Note: Due to some complexities with xarray of removing Nan values a preliminary interpolation is done to interoplate Nan values using using the original period range, then an interpolation is done onto the new period range given. You can change the type of interpolation for the 'na_interpolate' and 'method'.

[24]:
```
import numpy as np
```

[25]:
```
interpolated = mt_object.interpolate(np.logspace(1, 4, 12))
```

[26]:
```
interpolated.plot_mt_response()
```



[26]:
```
Plotting PlotMTResponse
```

Static Shift

Static shift is a distortion that affects the apparent resisitivity. We can apply a scalar to shift apparent resistivity up or down.

$\mathbf{Z} = \mathbf{S} * \mathbf{Z}_0$

where:

$\mathbf{S} = \begin{vmatrix} S_x & S_x \\ S_y & S_y \end{vmatrix}$

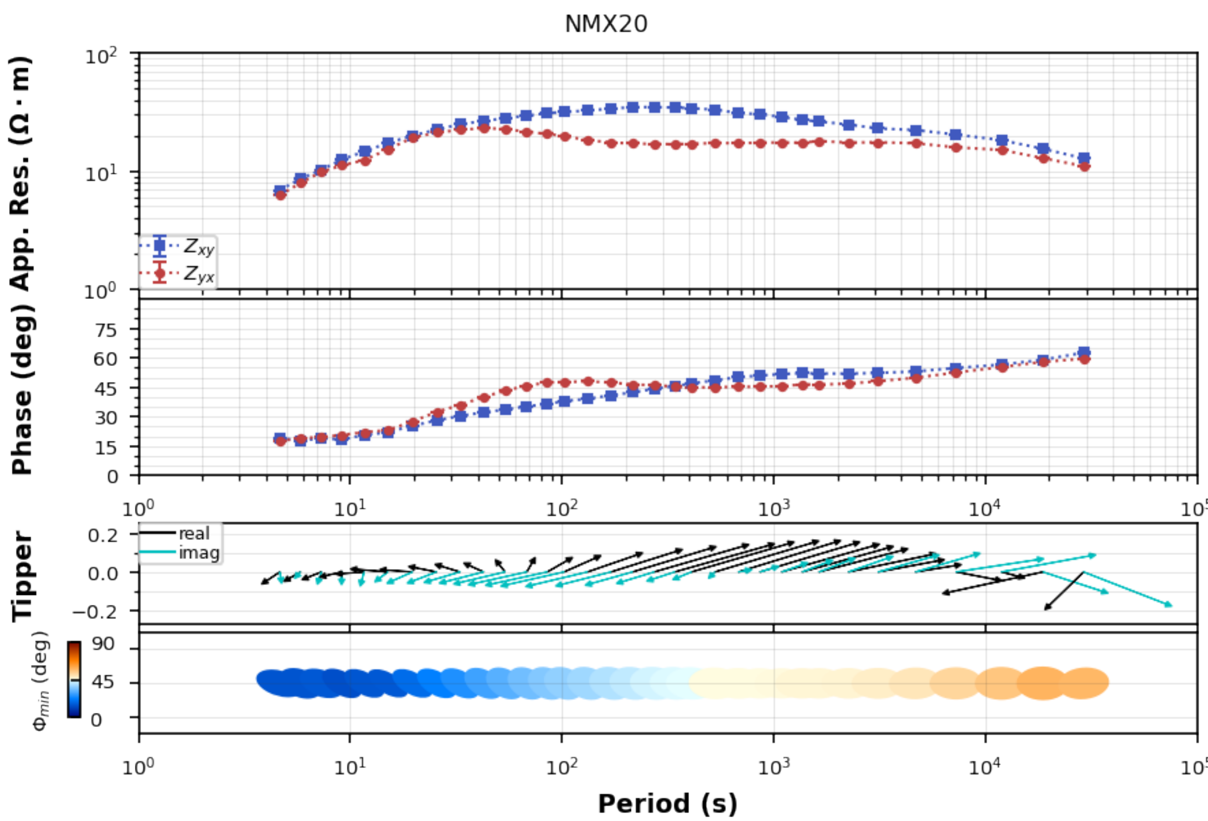
To remove the static shift multiply by $\mathbf{S}^{-1}$

Note: This assumes that the static shift is given in resistivity coordinates thus the square root of S will be applied.

Note: Numbers great than 1 will move the apparent resisitity down and numbers smaller than 1 will shift apparent resistivity up.

[27]:
```
static_shifted = mt_object.remove_static_shift(ss_x=1.5)
```

```
[28]: static_shifted.plot_mt_response()
```

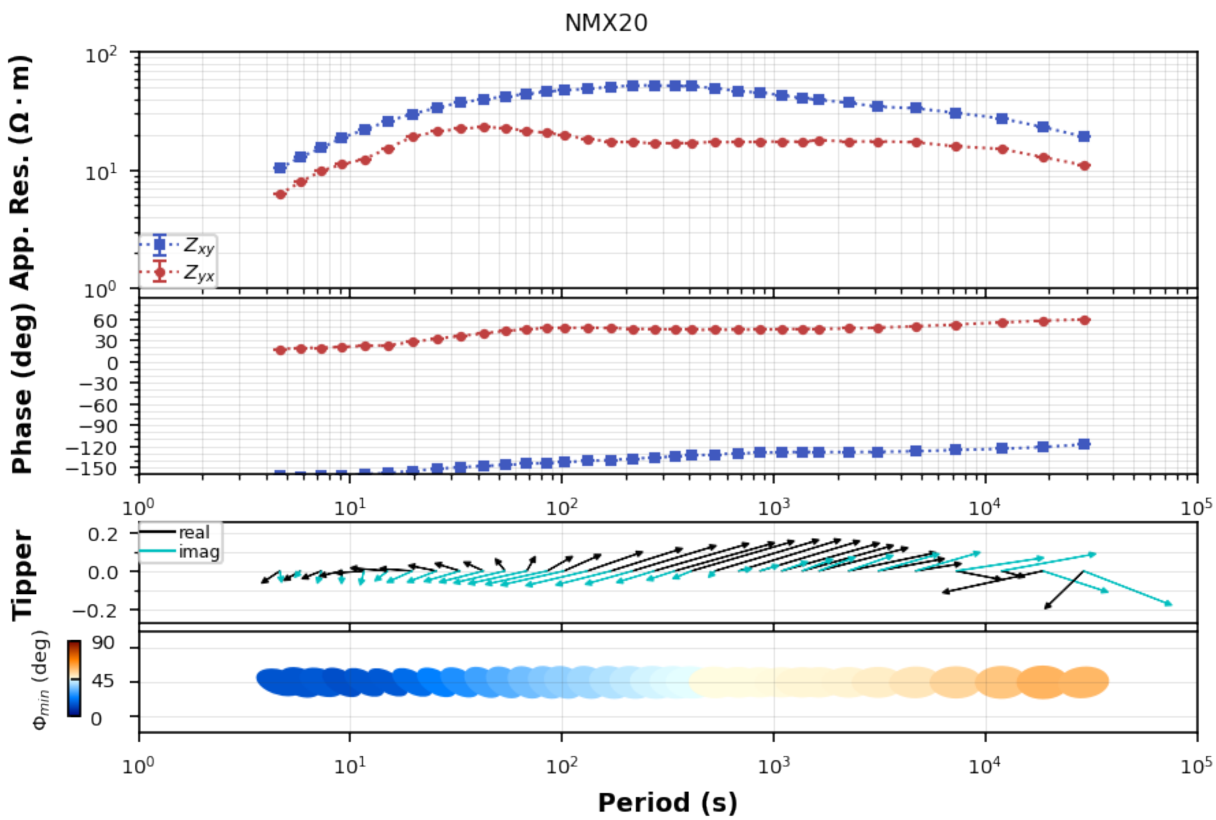



```
[28]: Plotting PlotMTResponse
```

Flip Phase

Occasionally you will compute a transfer function that has phase flipped, usually a coil was hooked ub backwards, or the electric lines were hooked up in reverse. If Hz data were collected an easy way to tell is plotting induction vectors for the survey and seeing if any look flipped. That is a good indication the magnetic channels was flipped. Otherwise check the electric channels. Here we will assume that the Ex was flipped so we flip Zxx and Zxy.

```
[29]: flipped = mt_object.flip_phase(zxx=True, zxy=True)
      flipped.plot_mt_response()
```

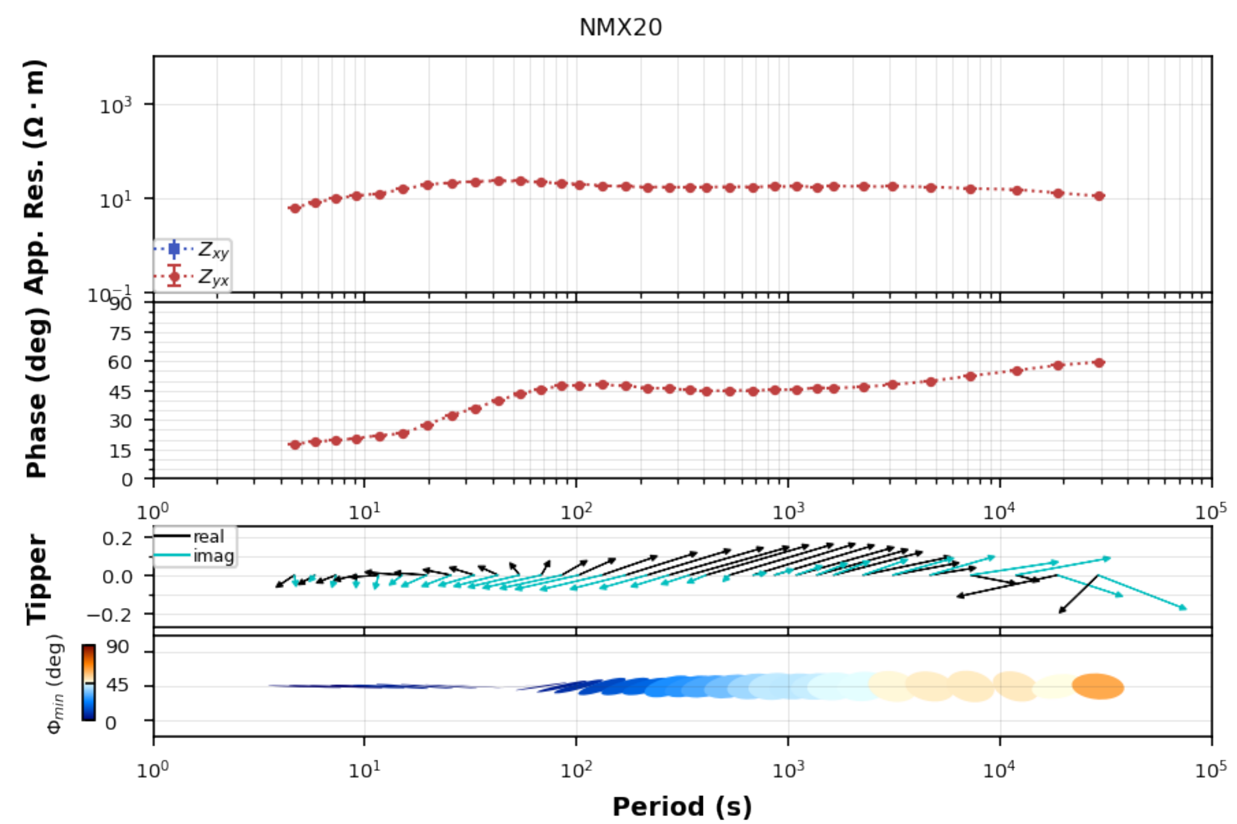


[29]: Plotting PlotMTResponse

Remove a component

Sometimes data can be terrible for a single component and you may want to remove it before analyzing the data.

[30]:
```
removed_xy = mt_object.remove_component(zxy=True)
removed_xy.plot_mt_response()
```



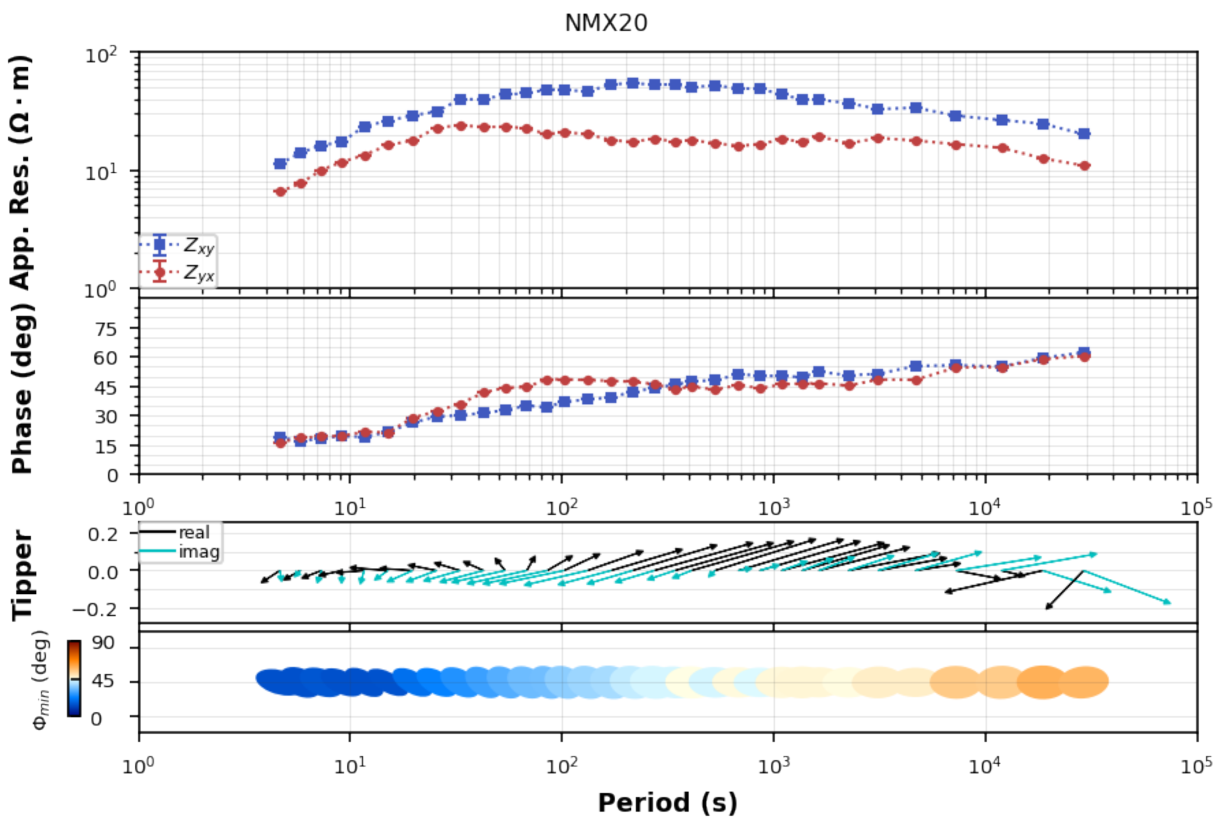
[30]: Plotting PlotMTResponse

Add White Noise

If you are doing some sensitivity tests or inverting synthetic data adding white noise can be useful. Here we will add 5%

[31]:
```
adding_white_noise = mt_object.add_white_noise(.05, inplace=False)
```

[32]:
```
adding_white_noise.plot_mt_response()
```



[32]: Plotting PlotMTResponse

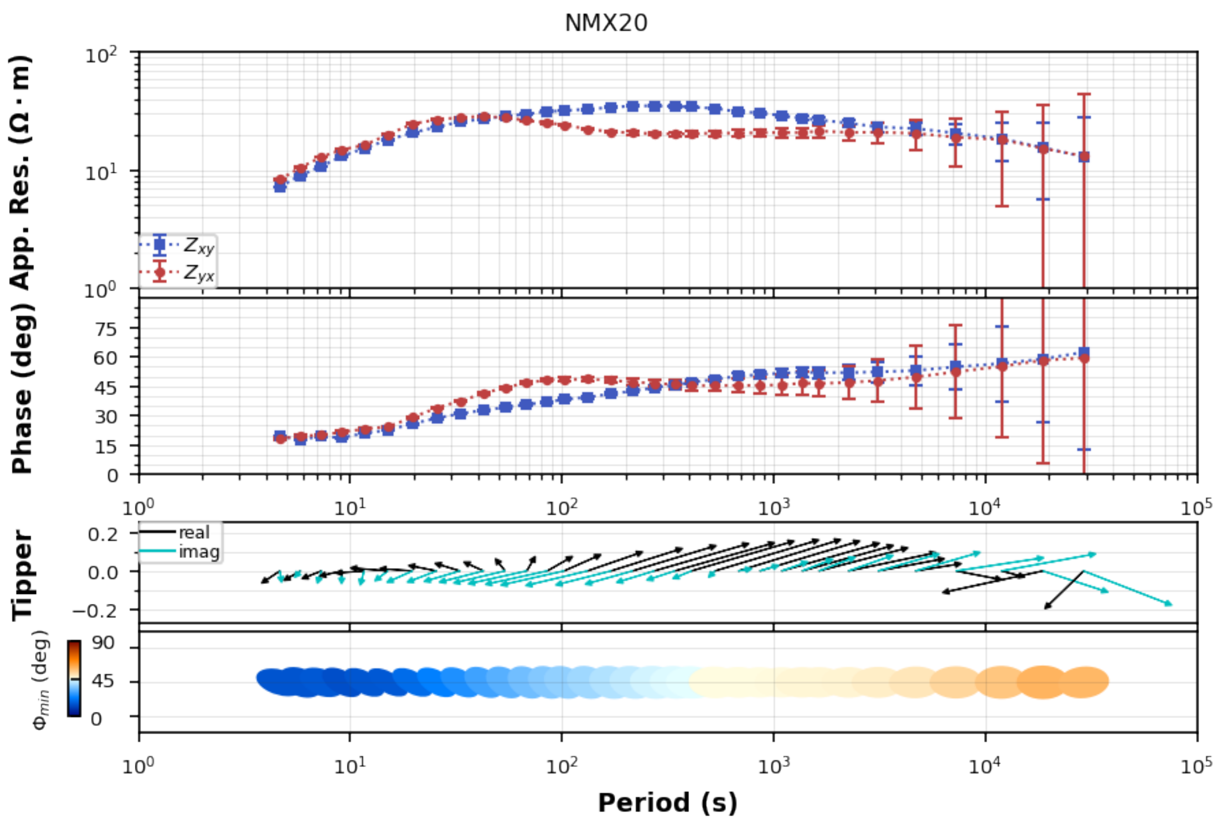
Remove Distortion

Distortion is a pain an usually comes in the form of near surface distortion like a static shift. If you have no other way to estimated distortion, like a TEM measurement or nearby stations a relative estimate can be made using the phase tensor. See Bibby et al., (2005). This basically looks for 1D parts of the transfer function by comparing the TE and TM phases. If the phases match then so should the apparent resistivity. However, knowing the true value of static shift is incomplete and therefore the apparent resistivity curves are pinched together to remove relative static shift.

There might be some issues with uncertainty propagation.

[33]:
```
distortion_removed = mt_object.remove_distortion(n_frequencies=25)
```

[34]:
```
distortion_removed.plot_mt_response()
```



[34]: Plotting PlotMTResponse

[ ]:

1.2.2 Building an MTCollection

The first step to analyzing a set of transfer functions is to load them into an `MTCollection` which is just a wrapper around an `MTH5` file. The advantage to doing this is that you only have to do this once from the various transfer functions that you have, which might include various flavors of EDI, EMTFXML, J-files, Z-files, AVG-files, etc. The other advantage is that now you have a database to work with: a single file where data is readily accessible vs. multiple ASCII files that need to be read in each time you want to do something.

Using the class `MT` any[1] transfer function can be read into a generic transfer function container `MT`. If you want to adjust `survey.id` or other metadata attributes you can do it from the `MT` object.

1. [^](#fn1) It works 100% of the time 50% of the time. Most transfer function files are supported but if you find one is not raise an issue

Test Data

Test data can be found at mtpy-data. Installation instructions are included in README.

[1]:
```
from mtpy_data import GRID_LIST, PROFILE_LIST
```

1. Initiate MTCollection

First initiate an `MTCollection`, here we will save to our current working directory.

[1]:
```
from pathlib import Path
from mtpy import MT, MTCollection
```

[2]:
```
mtc = MTCollection()
```

[3]:
```
mtc.open_collection(Path().cwd().joinpath("test_mt_collection.h5"))
```

2. Load Transfer Functions

Step one is locating all the transfer function files you want to read in. Here we will read in a couple different folders. If you have a couple different sets of data from different surveys, the MTCollection will store each in a survey group named by the `MT.survey_metadata.id`. However, this isn't a common attribute in EDI files, so if you want to separate them out add the correct survey ID.

Note: Loading transfer functions into an MTCollection can take some time so if you have over 100 be patient. But you only have to do this once.

2a. Load Profile Data

The profile data does not have a `survey.id` in the EDI files so we will add one.

[6]:
```
%%time
for fn in PROFILE_LIST:
    mt_object = MT()
    mt_object.read(fn)
    mt_object.survey_metadata.id = "profile"
    mtc.add_tf(mt_object)
```

```
23:10:19T15:45:54 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 106.001 !=␣
↪101.001
23:10:19T15:45:54 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 107.001 !=␣
↪102.001
23:10:19T15:45:54 | WARNING | line:1057 |mth5.mth5 | get_survey | /Experiment/Surveys/
↪profile does not exist, check survey_list for existing names.
23:10:19T15:45:55 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 106.001 !=␣
↪101.001
23:10:19T15:45:55 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 107.001 !=␣
↪102.001
23:10:19T15:45:55 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 106.001 !=␣
↪101.001
23:10:19T15:45:55 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 107.001 !=␣
```

(continues on next page)

(continued from previous page)

```
↪102.001
23:10:19T15:45:56 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 106.001 !=␣
↪101.001
23:10:19T15:45:56 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 107.001 !=␣
↪102.001
23:10:19T15:45:57 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 106.001 !=␣
↪101.001
23:10:19T15:45:57 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 107.001 !=␣
↪102.001
23:10:19T15:45:57 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 106.001 !=␣
↪101.001
23:10:19T15:45:57 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 107.001 !=␣
↪102.001
23:10:19T15:45:58 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 106.001 !=␣
↪101.001
23:10:19T15:45:58 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 107.001 !=␣
↪102.001
23:10:19T15:45:59 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 106.001 !=␣
↪101.001
23:10:19T15:45:59 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 107.001 !=␣
↪102.001
23:10:19T15:46:00 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 106.001 !=␣
↪101.001
23:10:19T15:46:00 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 107.001 !=␣
↪102.001
23:10:19T15:46:02 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 106.001 !=␣
↪101.001
23:10:19T15:46:02 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 107.001 !=␣
↪102.001
23:10:19T15:46:03 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 106.001 !=␣
↪101.001
23:10:19T15:46:03 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 107.001 !=␣
↪102.001
23:10:19T15:46:04 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 106.001 !=␣
↪101.001
23:10:19T15:46:04 | INFO | line:122 |mt_metadata.base.metadata | __eq__ | id: 107.001 !=␣
↪102.001
Wall time: 11.7 s
```

2b. Load Grid Data

The grid data also does not have a `survey.id` so we will add one.

[7]:
```
%%time
for fn in GRID_LIST:
    mt_object = MT()
    mt_object.read(fn)
    mt_object.survey_metadata.id = "grid"
    mtc.add_tf(mt_object)
```

```
23:10:19T15:46:29 | WARNING | line:1057 |mth5.mth5 | get_survey | /Experiment/Surveys/
↪grid does not exist, check survey_list for existing names.
Wall time: 2min 38s
```

3. Working and Master Dataframes

MTCollection includes a summary table of the transfer functions that it contains in the form of a pandas.DataFrame, this is the MTCollection.master_dataframe. There is also a MTCollection.working_dataframe which is a subset of the master_dataframe that the user can specify. This allows the MTCollection to be a catch-all for all your transfer functions and then when you only want to work with a small subset from a certain survey or geographic area you can query the master_dataframe to set the working_dataframe.

3a. Profile Working Dataframe

Here we will choose to work only with data that has a survey.id = 'profile'.

```
[4]: mtc.working_dataframe = mtc.master_dataframe.loc[mtc.master_dataframe.survey == "profile
↪"]
```

```
[5]: mtc.working_dataframe
```

```
[5]:    station   survey   latitude   longitude  elevation   tf_id units  \
59   15125A  profile -22.370806  149.188639      200.0  15125A  none
60   15126A  profile -22.370639  149.193500      200.0  15126A  none
61   15127A  profile -22.371028  149.198417      201.0  15127A  none
62   15128A  profile -22.370861  149.203306      200.0  15128A  none
63   15129A  profile -22.371083  149.208083      202.0  15129A  none
64   15130A  profile -22.371222  149.212972      201.0  15130A  none
65   16122A  profile -22.325611  149.174361      210.0  16122A  none
66   16123A  profile -22.325556  149.179056      213.0  16123A  none
67   16124A  profile -22.325694  149.184472      212.0  16124A  none
68   16125A  profile -22.325750  149.189306      219.0  16125A  none
69   16126A  profile -22.325806  149.194000      214.0  16126A  none
70   16127A  profile -22.325889  149.198861      220.0  16127A  none

    has_impedance  has_tipper  has_covariance  period_min  period_max  \
59           True        True           False    0.000096    2.857143
60           True        True           False    0.000096    2.857143
61           True        True           False    0.000096    2.857143
62           True        True           False    0.000096    2.857143
63           True        True           False    0.000096    2.857143
64           True        True           False    0.000096    2.857143
65           True        True           False    0.000096    2.857143
66           True        True           False    0.000096    2.857143
67           True        True           False    0.000096    2.857143
68           True        True           False    0.000096    2.857143
69           True        True           False    0.000096    2.857143
70           True        True           False    0.000096    2.857143

             hdf5_reference   station_hdf5_reference
59  <HDF5 object reference>  <HDF5 object reference>
```

(continues on next page)

(continued from previous page)

```
60  <HDF5 object reference>  <HDF5 object reference>
61  <HDF5 object reference>  <HDF5 object reference>
62  <HDF5 object reference>  <HDF5 object reference>
63  <HDF5 object reference>  <HDF5 object reference>
64  <HDF5 object reference>  <HDF5 object reference>
65  <HDF5 object reference>  <HDF5 object reference>
66  <HDF5 object reference>  <HDF5 object reference>
67  <HDF5 object reference>  <HDF5 object reference>
68  <HDF5 object reference>  <HDF5 object reference>
69  <HDF5 object reference>  <HDF5 object reference>
70  <HDF5 object reference>  <HDF5 object reference>
```

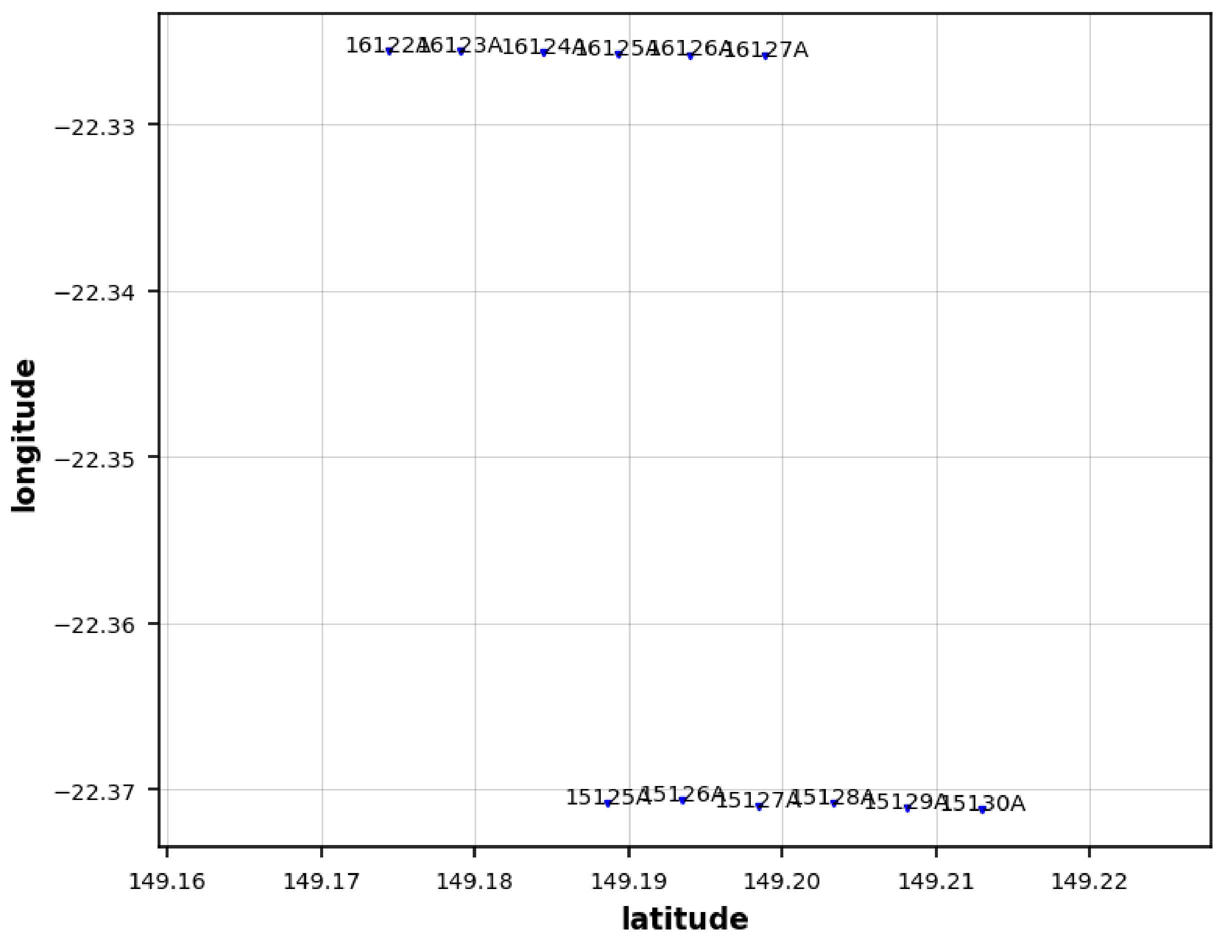
Plot station locations

Now that we have queried for only those stations in the 'profile' survey lets plot the station locations just for a sanity check. Looks like the 1600 line and 1500 line are different, so let's pick just the 1500 line.

Warning: Southern hemisphere locations have issues with contextily and not sure why. Below is an example on how to change the provider. See contextily providers for more details.

[8]:
```
stations_plot = mtc.plot_stations(pad=.0001)
```

```
23:10:19T16:00:43 | WARNING | line:170 |mtpy.imaging.plot_stations | plot | Could not␣
↪add base map because Tile URL resulted in a 404 error. Double-check your tile url:
https://basemap.nationalmap.gov/arcgis/rest/services/USGSTopo/MapServer/tile/15/18469/
↪29962
```



Extract only the 15 line

[9]:
```
mtc.working_dataframe = mtc.working_dataframe.query('station.str.startswith("15")')
```

[10]:
```
mtc.working_dataframe
```

[10]:
```
   station   survey   latitude   longitude  elevation   tf_id units  \
59  15125A  profile -22.370806  149.188639      200.0  15125A  none
60  15126A  profile -22.370639  149.193500      200.0  15126A  none
61  15127A  profile -22.371028  149.198417      201.0  15127A  none
62  15128A  profile -22.370861  149.203306      200.0  15128A  none
63  15129A  profile -22.371083  149.208083      202.0  15129A  none
64  15130A  profile -22.371222  149.212972      201.0  15130A  none

    has_impedance  has_tipper  has_covariance  period_min  period_max  \
59           True        True           False    0.000096    2.857143
60           True        True           False    0.000096    2.857143
61           True        True           False    0.000096    2.857143
62           True        True           False    0.000096    2.857143
63           True        True           False    0.000096    2.857143
```

(continues on next page)

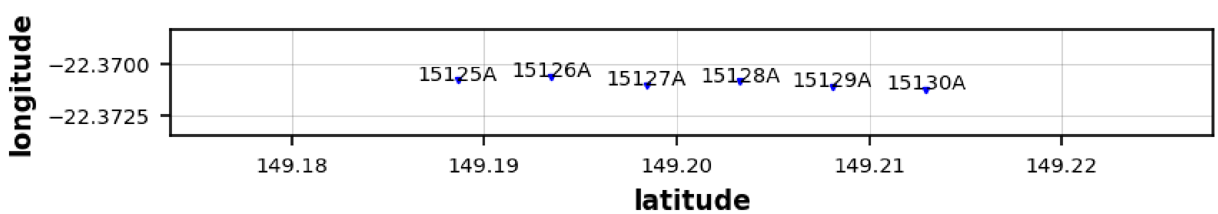
(continued from previous page)

```
64             True         True            False     0.000096     2.857143

               hdf5_reference    station_hdf5_reference
59  <HDF5 object reference>  <HDF5 object reference>
60  <HDF5 object reference>  <HDF5 object reference>
61  <HDF5 object reference>  <HDF5 object reference>
62  <HDF5 object reference>  <HDF5 object reference>
63  <HDF5 object reference>  <HDF5 object reference>
64  <HDF5 object reference>  <HDF5 object reference>
```

[11]:
```
station_plot = mtc.plot_stations(pad=.0001)
```

```
C:\Users\jpeacock\Anaconda3\envs\em\lib\site-packages\contextily\tile.py:581:
↪UserWarning: The inferred zoom level of 21 is not valid for the current tile provider
↪(valid zooms: 0 - 20).
  warnings.warn(msg)
```

```
23:10:19T16:00:54 | WARNING | line:170 |mtpy.imaging.plot_stations | plot | Could not
↪add base map because Tile URL resulted in a 404 error. Double-check your tile url:
https://basemap.nationalmap.gov/arcgis/rest/services/USGSTopo/MapServer/tile/20/591168/
↪958827
```



3b. Grid Working DataFrame

Now lets get only the stations in the ‘grid’ survey and plot them.

[12]:
```
mtc.working_dataframe = mtc.master_dataframe.loc[mtc.master_dataframe.survey == "grid"]
```

[13]:
```
mtc.working_dataframe
```

[13]:
```
   station survey   latitude   longitude  elevation  tf_id  \
0    gv100   grid  38.611381 -118.535261   1437.400  gv100
1    gv101   grid  38.594561 -118.351111   1540.550  gv101
2    gv102   grid  38.593692 -118.276822   1554.800  gv102
3    gv103   grid  38.585283 -118.202481   1543.900  gv103
4    gv104   grid  38.596456 -118.136547   1801.800  gv104
5    gv105   grid  38.594131 -118.071933   1901.600  gv105
6    gv106   grid  38.598842 -117.940803   1653.269  gv106
7    gv107   grid  38.612339 -117.882392   1753.700  gv107
8    gv108   grid  38.691575 -118.501819   1904.900  gv108
9    gv109   grid  38.721022 -118.421311   1769.400  gv109
10   gv110   grid  38.672894 -118.330589   1857.100  gv110
11   gv111   grid  38.683950 -118.272311   1893.411  gv111
```

(continues on next page)

(continued from previous page)

```
12   gv112   grid  38.693392 -118.092872   1572.200  gv112
13   gv113   grid  38.660622 -117.991272   1531.200  gv113
14   gv114   grid  38.695694 -117.965139   1542.400  gv114
15   gv115   grid  38.694236 -117.872453   1625.700  gv115
16   gv116   grid  38.772536 -118.499314   1642.800  gv116
17   gv117   grid  38.827428 -118.423014   1505.000  gv117
18   gv118   grid  38.780817 -118.352297   1445.286  gv118
19   gv119   grid  38.793572 -118.284242   1394.190  gv119
20   gv120   grid  38.834919 -118.099611   1304.800  gv120
21   gv121   grid  38.781614 -118.050139   1394.700  gv121
22   gv122   grid  38.779211 -117.957842   1535.500  gv122
23   gv123   grid  38.758194 -117.850214   1855.454  gv123
24   gv124   grid  38.894753 -118.455667   1368.218  gv124
25   gv125   grid  38.897069 -118.379408   1234.500  gv125
26   gv126   grid  38.891994 -118.278522   1234.000  gv126
27   gv127   grid  38.870372 -118.232033   1251.100  gv127
28   gv128   grid  38.887003 -118.080164   1473.400  gv128
29   gv129   grid  38.832450 -118.020161   1357.029  gv129
30   gv130   grid  38.870592 -117.958572   1355.761  gv130
31   gv131   grid  38.899672 -117.858122   1596.800  gv131
32   gv132   grid  38.899553 -118.006981   1381.500  gv132
33   gv133   grid  38.948494 -118.363769   1281.300  gv133
34   gv134   grid  38.964211 -118.309064   1262.900  gv134
35   gv135   grid  38.959308 -118.229806   1236.400  gv135
36   gv136   grid  38.931881 -118.143908   1335.800  gv136
37   gv137   grid  38.938647 -118.040322   1542.100  gv137
38   gv138   grid  38.958664 -117.962525   1431.400  gv138
39   gv139   grid  38.958169 -117.868064   1415.700  gv139
40   gv140   grid  39.065478 -118.467969   1460.400  gv140
41   gv141   grid  39.049328 -118.395528   1620.500  gv141
42   gv142   grid  39.044442 -118.308225   1738.900  gv142
43   gv143   grid  39.053244 -118.252172   1627.200  gv143
44   gv144   grid  39.013067 -118.162347   1548.029  gv144
45   gv145   grid  39.052336 -118.042419   1568.999  gv145
46   gv146   grid  39.045111 -117.961328   1596.200  gv146
47   gv147   grid  39.045894 -117.872028   1587.100  gv147
48   gv148   grid  39.158608 -118.506814   1591.820  gv148
49   gv149   grid  39.128144 -118.421497   1667.600  gv149
50   gv150   grid  39.141736 -118.320472   1490.559  gv150
51   gv151   grid  39.111331 -118.265897   1449.100  gv151
52   gv152   grid  39.141939 -118.155467   1660.881  gv152
53   gv153   grid  39.142878 -118.045928   1715.500  gv153
54   gv154   grid  39.149414 -117.972769   1743.100  gv154
55   gv155   grid  39.138342 -117.872131   1811.100  gv155
56   gv160   grid  38.914786 -118.190761   1246.000  gv160
57   gv161   grid  38.993356 -117.997683   1474.727  gv161
58   gv163   grid  38.834078 -118.236278   1262.700  gv163

                                      units  has_impedance  has_tipper  \
0   millivolts_per_kilometer_per_nanotesla           True        True
1   millivolts_per_kilometer_per_nanotesla           True        True
2   millivolts_per_kilometer_per_nanotesla           True        True
```

(continues on next page)

(continued from previous page)

```
3   millivolts_per_kilometer_per_nanotesla          True        True
4   millivolts_per_kilometer_per_nanotesla          True        True
5   millivolts_per_kilometer_per_nanotesla          True        True
6   millivolts_per_kilometer_per_nanotesla          True        True
7   millivolts_per_kilometer_per_nanotesla          True        True
8   millivolts_per_kilometer_per_nanotesla          True        True
9   millivolts_per_kilometer_per_nanotesla          True        True
10  millivolts_per_kilometer_per_nanotesla          True        True
11  millivolts_per_kilometer_per_nanotesla          True        True
12  millivolts_per_kilometer_per_nanotesla          True        True
13  millivolts_per_kilometer_per_nanotesla          True        True
14  millivolts_per_kilometer_per_nanotesla          True        True
15  millivolts_per_kilometer_per_nanotesla          True        True
16  millivolts_per_kilometer_per_nanotesla          True        True
17  millivolts_per_kilometer_per_nanotesla          True        True
18  millivolts_per_kilometer_per_nanotesla          True        True
19  millivolts_per_kilometer_per_nanotesla          True        True
20  millivolts_per_kilometer_per_nanotesla          True        True
21  millivolts_per_kilometer_per_nanotesla          True        True
22  millivolts_per_kilometer_per_nanotesla          True        True
23  millivolts_per_kilometer_per_nanotesla          True        True
24  millivolts_per_kilometer_per_nanotesla          True        True
25  millivolts_per_kilometer_per_nanotesla          True        True
26  millivolts_per_kilometer_per_nanotesla          True        True
27  millivolts_per_kilometer_per_nanotesla          True        True
28  millivolts_per_kilometer_per_nanotesla          True        True
29  millivolts_per_kilometer_per_nanotesla          True        True
30  millivolts_per_kilometer_per_nanotesla          True        True
31  millivolts_per_kilometer_per_nanotesla          True        True
32  millivolts_per_kilometer_per_nanotesla          True        True
33  millivolts_per_kilometer_per_nanotesla          True        True
34  millivolts_per_kilometer_per_nanotesla          True        True
35  millivolts_per_kilometer_per_nanotesla          True        True
36  millivolts_per_kilometer_per_nanotesla          True        True
37  millivolts_per_kilometer_per_nanotesla          True        True
38  millivolts_per_kilometer_per_nanotesla          True        True
39  millivolts_per_kilometer_per_nanotesla          True        True
40  millivolts_per_kilometer_per_nanotesla          True        True
41  millivolts_per_kilometer_per_nanotesla          True        True
42  millivolts_per_kilometer_per_nanotesla          True        True
43  millivolts_per_kilometer_per_nanotesla          True        True
44  millivolts_per_kilometer_per_nanotesla          True        True
45  millivolts_per_kilometer_per_nanotesla          True        True
46  millivolts_per_kilometer_per_nanotesla          True        True
47  millivolts_per_kilometer_per_nanotesla          True        True
48  millivolts_per_kilometer_per_nanotesla          True        True
49  millivolts_per_kilometer_per_nanotesla          True        True
50  millivolts_per_kilometer_per_nanotesla          True        True
51  millivolts_per_kilometer_per_nanotesla          True        True
52  millivolts_per_kilometer_per_nanotesla          True        True
53  millivolts_per_kilometer_per_nanotesla          True        True
54  millivolts_per_kilometer_per_nanotesla          True        True
```

(continues on next page)

(continued from previous page)

```
55  millivolts_per_kilometer_per_nanotesla          True        True
56  millivolts_per_kilometer_per_nanotesla          True        True
57  millivolts_per_kilometer_per_nanotesla          True        True
58  millivolts_per_kilometer_per_nanotesla          True        True

    has_covariance  period_min   period_max          hdf5_reference  \
0            False    0.001302  2048.000210  <HDF5 object reference>
1            False    0.001302  2048.000210  <HDF5 object reference>
2            False    0.001302  2048.000210  <HDF5 object reference>
3            False    0.001302  1365.000061  <HDF5 object reference>
4            False    0.001302  1365.000061  <HDF5 object reference>
5            False    0.001302  1365.000061  <HDF5 object reference>
6            False    0.001302  2048.000210  <HDF5 object reference>
7            False    0.001302  2048.000210  <HDF5 object reference>
8            False    0.001302  2048.000210  <HDF5 object reference>
9            False    0.001302  2048.000210  <HDF5 object reference>
10           False    0.001302  2048.000210  <HDF5 object reference>
11           False    0.001302  2048.000210  <HDF5 object reference>
12           False    0.001302  2048.000210  <HDF5 object reference>
13           False    0.001302  2048.000210  <HDF5 object reference>
14           False    0.001302  2048.000210  <HDF5 object reference>
15           False    0.001302  2048.000210  <HDF5 object reference>
16           False    0.001302  2048.000210  <HDF5 object reference>
17           False    0.001302  2048.000210  <HDF5 object reference>
18           False    0.001302  2048.000210  <HDF5 object reference>
19           False    0.001302  2048.000210  <HDF5 object reference>
20           False    0.001302  2048.000210  <HDF5 object reference>
21           False    0.001302  2048.000210  <HDF5 object reference>
22           False    0.001302  2048.000210  <HDF5 object reference>
23           False    0.001302  2048.000210  <HDF5 object reference>
24           False    0.001302  2048.000210  <HDF5 object reference>
25           False    0.001302  2048.000210  <HDF5 object reference>
26           False    0.001302  2048.000210  <HDF5 object reference>
27           False    0.001302  1115.940903  <HDF5 object reference>
28           False    0.001302  2048.000210  <HDF5 object reference>
29           False    0.001302  2048.000210  <HDF5 object reference>
30           False    0.001302  2048.000210  <HDF5 object reference>
31           False    0.001302  2048.000210  <HDF5 object reference>
32           False    0.001302  2048.000210  <HDF5 object reference>
33           False    0.001302  2048.000210  <HDF5 object reference>
34           False    0.001302  2048.000210  <HDF5 object reference>
35           False    0.001302  2048.000210  <HDF5 object reference>
36           False    0.001302  2048.000210  <HDF5 object reference>
37           False    0.001302  2048.000210  <HDF5 object reference>
38           False    0.001302  2048.000210  <HDF5 object reference>
39           False    0.001302  2048.000210  <HDF5 object reference>
40           False    0.001302  2048.000210  <HDF5 object reference>
41           False    0.001302  2048.000210  <HDF5 object reference>
42           False    0.001764  2048.000210  <HDF5 object reference>
43           False    0.001302  2048.000210  <HDF5 object reference>
44           False    0.001302  1115.940903  <HDF5 object reference>
45           False    0.001302  2048.000210  <HDF5 object reference>
```

(continues on next page)

(continued from previous page)

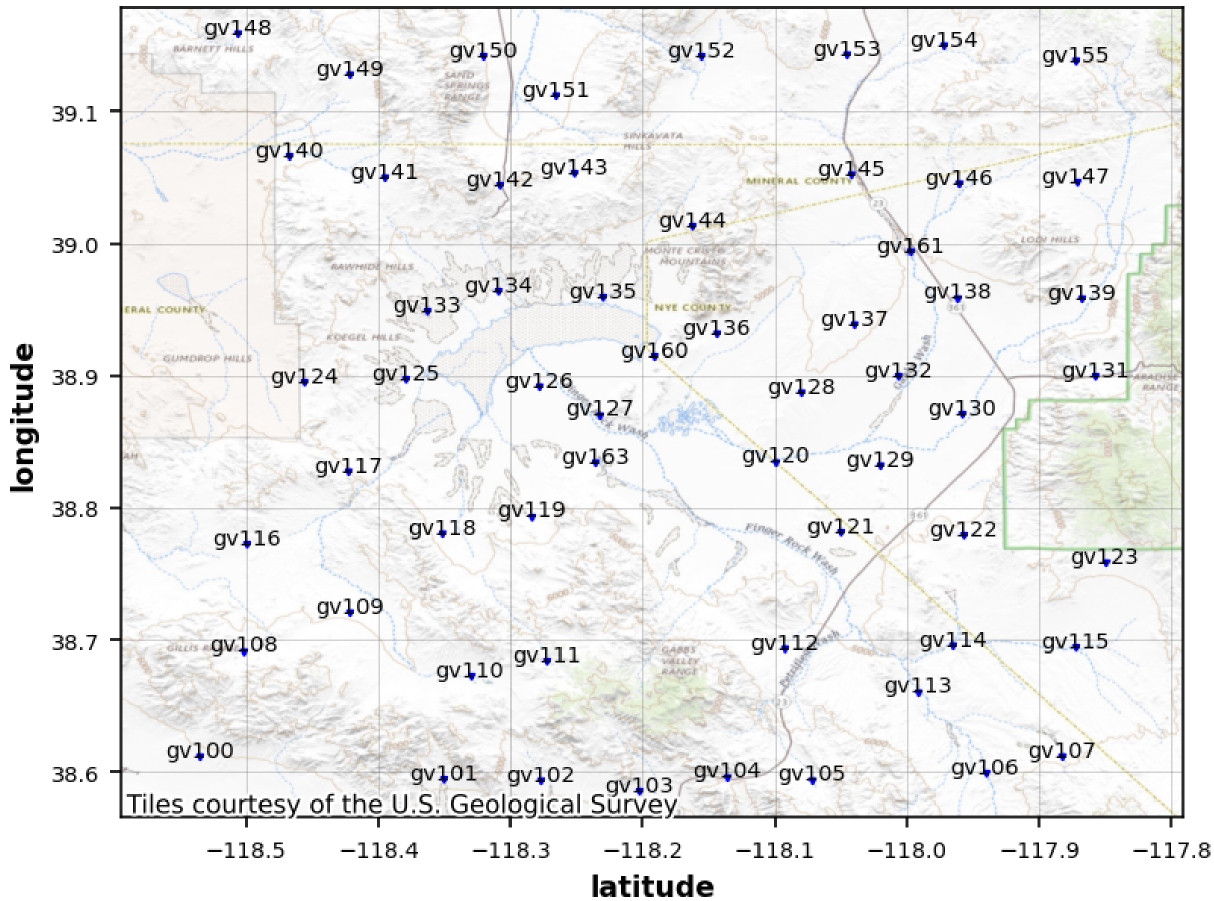
```
46          False    0.001302  2048.000210  <HDF5 object reference>
47          False    0.001302  2048.000210  <HDF5 object reference>
48          False    0.001302  2048.000210  <HDF5 object reference>
49          False    0.001302  2048.000210  <HDF5 object reference>
50          False    0.001302  2048.000210  <HDF5 object reference>
51          False    0.001302  2048.000210  <HDF5 object reference>
52          False    0.001302  2048.000210  <HDF5 object reference>
53          False    0.001302  2048.000210  <HDF5 object reference>
54          False    0.001302  2048.000210  <HDF5 object reference>
55          False    0.001302  2048.000210  <HDF5 object reference>
56          False    0.001302  2048.000210  <HDF5 object reference>
57          False    0.001302  1021.063207  <HDF5 object reference>
58          False    0.001302  2048.000210  <HDF5 object reference>

     station_hdf5_reference
0   <HDF5 object reference>
1   <HDF5 object reference>
2   <HDF5 object reference>
3   <HDF5 object reference>
4   <HDF5 object reference>
5   <HDF5 object reference>
6   <HDF5 object reference>
7   <HDF5 object reference>
8   <HDF5 object reference>
9   <HDF5 object reference>
10  <HDF5 object reference>
11  <HDF5 object reference>
12  <HDF5 object reference>
13  <HDF5 object reference>
14  <HDF5 object reference>
15  <HDF5 object reference>
16  <HDF5 object reference>
17  <HDF5 object reference>
18  <HDF5 object reference>
19  <HDF5 object reference>
20  <HDF5 object reference>
21  <HDF5 object reference>
22  <HDF5 object reference>
23  <HDF5 object reference>
24  <HDF5 object reference>
25  <HDF5 object reference>
26  <HDF5 object reference>
27  <HDF5 object reference>
28  <HDF5 object reference>
29  <HDF5 object reference>
30  <HDF5 object reference>
31  <HDF5 object reference>
32  <HDF5 object reference>
33  <HDF5 object reference>
34  <HDF5 object reference>
35  <HDF5 object reference>
36  <HDF5 object reference>
```

(continues on next page)

(continued from previous page)

```
37  <HDF5 object reference>
38  <HDF5 object reference>
39  <HDF5 object reference>
40  <HDF5 object reference>
41  <HDF5 object reference>
42  <HDF5 object reference>
43  <HDF5 object reference>
44  <HDF5 object reference>
45  <HDF5 object reference>
46  <HDF5 object reference>
47  <HDF5 object reference>
48  <HDF5 object reference>
49  <HDF5 object reference>
50  <HDF5 object reference>
51  <HDF5 object reference>
52  <HDF5 object reference>
53  <HDF5 object reference>
54  <HDF5 object reference>
55  <HDF5 object reference>
56  <HDF5 object reference>
57  <HDF5 object reference>
58  <HDF5 object reference>
```

[17]:
```
station_plot = mtc.plot_stations(pad=.0005)
```



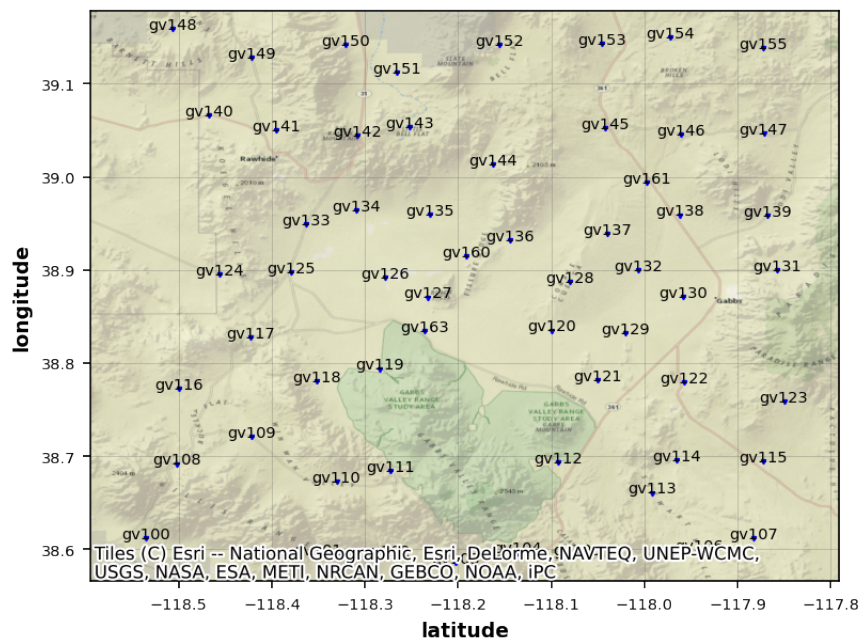
Change Basemap

`contextily` has good options for basemaps, you can check out all the options at https://contextily.readthedocs.io/en/latest/providers_deepdive.html.

Below is an exmaple of how to change the source basemap to the National Geographic World Map.

[18]:
```
import contextily as cx
```

[19]:
```
station_plot.cx_source = cx.providers.Esri.NatGeoWorldMap
station_plot.redraw_plot()
```



4. MTCollection vs MTData

`MTCollection` is the physical object where data are stored in memory and `MTData` is an extraction of the `MTCollection` that can be manipulated on RAM. Its designed this way so you don't have to keep accessing the MTH5 file and once loaded into RAM then `MTData` can manipulate the data in ways that may not want to be permentely stored.

`MTCollection` does have plotting methods, as seen above, but can be slow for other plotting methods because it needs to load in the data for each plot. This is sort of a bug and needs someone smarter to write better code, but for now convert the `working_dataframe` into an `MTData` object and from there plots, model inputs, and analysis can be done, which we will see in the `MTData` example notebook.

`MTData` can be built similar to `MTCollection` by reading in transfer function files if you don't want to build an MTH5 file. If you like what you've done with `MTData` you can write to an MTH5.

```
from mtpy import MT, MTData, MTCollection
md = MTData()

for filename in list_of_tf_files:
    mt_object = MT()
    mt_object.read(filename)
    md.add_station(mt_object)

# Do stuff to MTData object like interpolate, rotate, etc
with MTCollection() as mc:
    mc.open_collection("/path/to/mth5_file.h5")
    mc.from_mt_data(md)
```

5. Close MTCollection

Important: You need to close the MTCollection otherwise the file may get corrupted and you'll have to make the file all over again. Note that once the file is closed the transfer functions are no longer available. Therefore it is wise to convert to an MTData object.

[20]:
```
mtc.close_collection()
```

```
23:10:19T16:06:23 | INFO | line:760 |mth5.mth5 | close_mth5 | Flushing and closing C:\
↪Users\jpeacock\OneDrive - DOI\Documents\GitHub\mtpy-v2\docs\source\notebooks\test_mt_
↪collection.h5
```

5a. Context Manager

If you just want to build an MTH5 file and then close it the best way to do that is using the context manager:

```
with MTCollection() as mc:
    mc.open_collection("/path/to/mth5_file.h5")
    # add data to the files
```

[ ]:
```

```

1.2.3 MTData: Profile Example

Now that we've created an MTCollection object we can use it to do the more interesting things, like analyze strike, plot phase tensors, create inputs for modeling programs.

1. Open Collection

In the previous notebook we created an MTCollection object called test_mt_collection.h5. Lets open it and get the profile.

[1]:
```
from pathlib import Path

from mtpy import MTCollection
```

[2]:
```
mtc = MTCollection()
mtc.open_collection(Path().cwd().joinpath("test_mt_collection.h5"))
```

[3]:
```
mtc.working_dataframe = mtc.master_dataframe.loc[
    mtc.master_dataframe.survey == "profile"
].query('station.str.startswith("15")')
```

[4]:
```
mtc.working_dataframe
```

[4]:
```
   station   survey   latitude   longitude  elevation   tf_id units  \
59  15125A  profile -22.370806  149.188639      200.0  15125A  none
60  15126A  profile -22.370639  149.193500      200.0  15126A  none
61  15127A  profile -22.371028  149.198417      201.0  15127A  none
62  15128A  profile -22.370861  149.203306      200.0  15128A  none
```

(continues on next page)

(continued from previous page)

```
63  15129A  profile -22.371083  149.208083      202.0  15129A  none
64  15130A  profile -22.371222  149.212972      201.0  15130A  none

    has_impedance  has_tipper  has_covariance  period_min  period_max  \
59           True        True           False    0.000096    2.857143
60           True        True           False    0.000096    2.857143
61           True        True           False    0.000096    2.857143
62           True        True           False    0.000096    2.857143
63           True        True           False    0.000096    2.857143
64           True        True           False    0.000096    2.857143

             hdf5_reference   station_hdf5_reference
59  <HDF5 object reference>  <HDF5 object reference>
60  <HDF5 object reference>  <HDF5 object reference>
61  <HDF5 object reference>  <HDF5 object reference>
62  <HDF5 object reference>  <HDF5 object reference>
63  <HDF5 object reference>  <HDF5 object reference>
64  <HDF5 object reference>  <HDF5 object reference>
```

2. Convert to MTData

Now that we have the profile let's convert it to an `MTData` object.

[5]:
```
mtd = mtc.to_mt_data()
```

2a. Close MTCollection

You can now close the MTCollection to make sure if something crashes the file won't get corrupt for unknown reasons.

[6]:
```
mtc.close_collection()
```

```
23:10:23T09:21:00 | INFO | line:760 |mth5.mth5 | close_mth5 | Flushing and closing C:\
↪Users\jpeacock\OneDrive - DOI\Documents\GitHub\mtpy-v2\docs\source\notebooks\test_mt_
↪collection.h5
```

2b. Station Locations

A convenient attribute of `MTData` is the `station_locations` object. This is a `mtpy.core.MTStations` object and represented as a `pandas.DataFram`. You will notice here that `east` and `north` are not populated, that is because the `MTData` object is currently agnostic to a UTM coordinate system.

[7]:
```
mtd.station_locations
```

[7]:
```
    survey station   latitude   longitude  elevation datum_epsg  east  north  \
0  profile  15125A -22.370806  149.188639      200.0       4326   0.0    0.0
1  profile  15126A -22.370639  149.193500      200.0       4326   0.0    0.0
2  profile  15127A -22.371028  149.198417      201.0       4326   0.0    0.0
3  profile  15128A -22.370861  149.203306      200.0       4326   0.0    0.0
4  profile  15129A -22.371083  149.208083      202.0       4326   0.0    0.0
```

(continues on next page)

(continued from previous page)

```
5  profile  15130A -22.371222  149.212972      201.0       4326   0.0    0.0

  utm_epsg  model_east  model_north  model_elevation  profile_offset
0     None         0.0          0.0            200.0             0.0
1     None         0.0          0.0            200.0             0.0
2     None         0.0          0.0            201.0             0.0
3     None         0.0          0.0            200.0             0.0
4     None         0.0          0.0            202.0             0.0
5     None         0.0          0.0            201.0             0.0
```

2c. Setting UTM CRS

Its important to set the `MTData.utm_crs` attribute to make sure that stations can be projected into meters for plotting and creating model files. You can do this a couple of ways either through the `utm_crs` method or if you know the EPSG number you can input that. They should both do the same if you input the number.

If you have created a custom CRS, be sure to set `mtd.utm_crs` with the custom CRS.

[8]:
```
mtd.utm_crs = 32755
```

[9]:
```
mtd.utm_crs
```

[9]:
```
<Derived Projected CRS: EPSG:32755>
Name: WGS 84 / UTM zone 55S
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: Between 144°E and 150°E, southern hemisphere between 80°S and equator, onshore␣
↪and offshore. Australia. Papua New Guinea.
- bounds: (144.0, -80.0, 150.0, 0.0)
Coordinate Operation:
- name: UTM zone 55S
- method: Transverse Mercator
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

[10]:
```
mtd.station_locations
```

[10]:
```
    survey station   latitude   longitude  elevation datum_epsg  \
0  profile  15125A -22.370806  149.188639      200.0       4326
1  profile  15126A -22.370639  149.193500      200.0       4326
2  profile  15127A -22.371028  149.198417      201.0       4326
3  profile  15128A -22.370861  149.203306      200.0       4326
4  profile  15129A -22.371083  149.208083      202.0       4326
5  profile  15130A -22.371222  149.212972      201.0       4326

            east         north utm_epsg  model_east  model_north  \
0  725360.330833  7.524490e+06    32755         0.0          0.0
1  725861.314778  7.524502e+06    32755         0.0          0.0
2  726367.124612  7.524451e+06    32755         0.0          0.0
```

(continues on next page)

(continued from previous page)

```
3  726870.972550  7.524462e+06    32755          0.0          0.0
4  727362.745811  7.524430e+06    32755          0.0          0.0
5  727866.099363  7.524408e+06    32755          0.0          0.0

   model_elevation  profile_offset
0            200.0             0.0
1            200.0             0.0
2            201.0             0.0
3            200.0             0.0
4            202.0             0.0
5            201.0             0.0
```

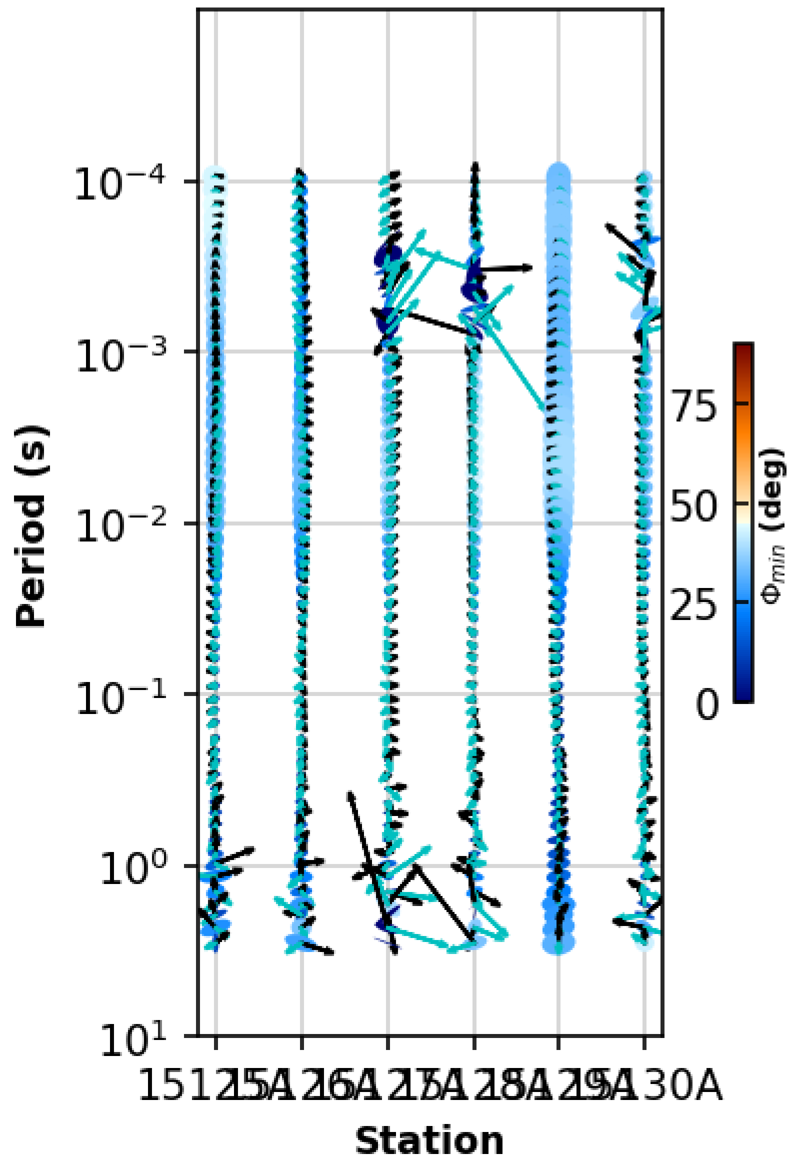
Now you can see that the locations have been projected into the given coordinate system.

Note: The `model_east` and `model_north` do not get populated, those are for relative coordinates for modeling.

3. Plot Phase Tensor Pseudosection

Here we are adjusting the stretch in the x-direction and plotting the tipper vectors ('r' = real, 'i' = imaginary, 'y' = yes to plot)

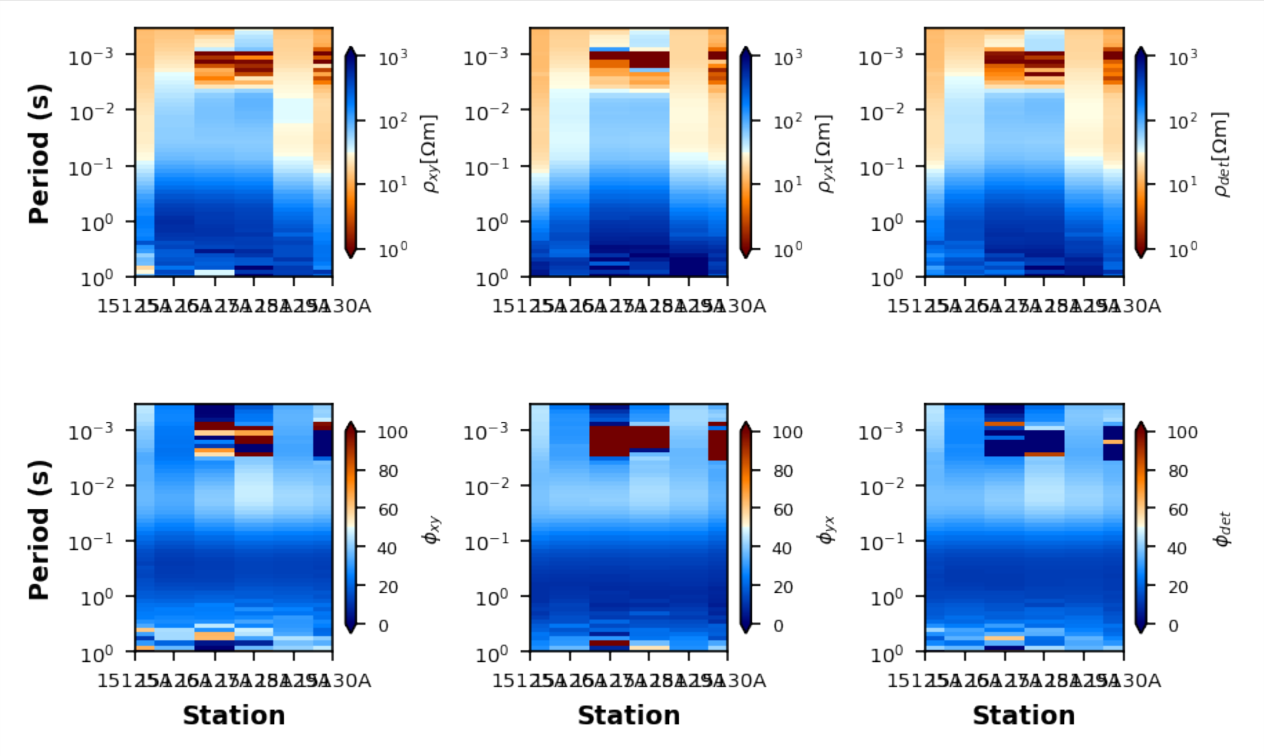
[11]:
```
ptps_plot = mtd.plot_phase_tensor_pseudosection(x_stretch=10, plot_tipper="yri")
```



4. Plot Resistivity and Phase Pseudosections

Here we are plotting the xy, yx, and det components of the impedance tensor.

[12]:
```
rpps_plot = mtd.plot_resistivity_phase_pseudosections(y_stretch=700, plot_det=True)
```



5. Plot Strike

Plotting strike angles are vary important when working with profile data. We can plot strike in a couple of ways as a compilation of all strike angles, or per decade. If you really want you could do it by region you query for the stations in each region.

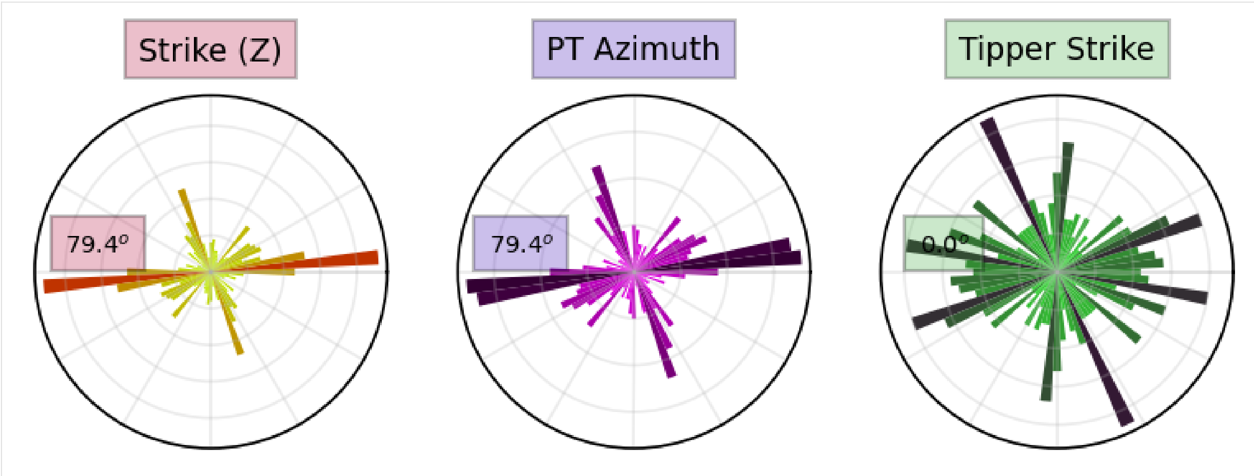
5a. All Periods

Here we will plot all periods of estimated strike. Notice that the plot includes the strike as estimated from the invariants (left) of Weaver et al. (2002), the phase tensor (middel) of Caldwell et al. (2004), and the induction vector strike.

Important: The induction strike points towards good conductors so should therefore be perpendicular to the impedance strike. We left it this way as a sanity check on strike angles.

[13]:
```
strike_plot_all = mtd.plot_strike()
```

```
23:10:23T09:21:29 | INFO | line:892 |mtpy.imaging.plot_strike | _plot_all_periods | Note:
↪ North is assumed to be 0 and the strike angle is measured clockwise positive.
```



5b. Per Decade

It can be informative to plot the strike angles per decade in period. This can provide information on if strike angle changes with depth. Because this is AMT data there isn't really a coherent strike angle until about 0.1 seconds. Notice the induction vector (tipper) strike is perpendicular to the impedance strike below 0.1 seconds.

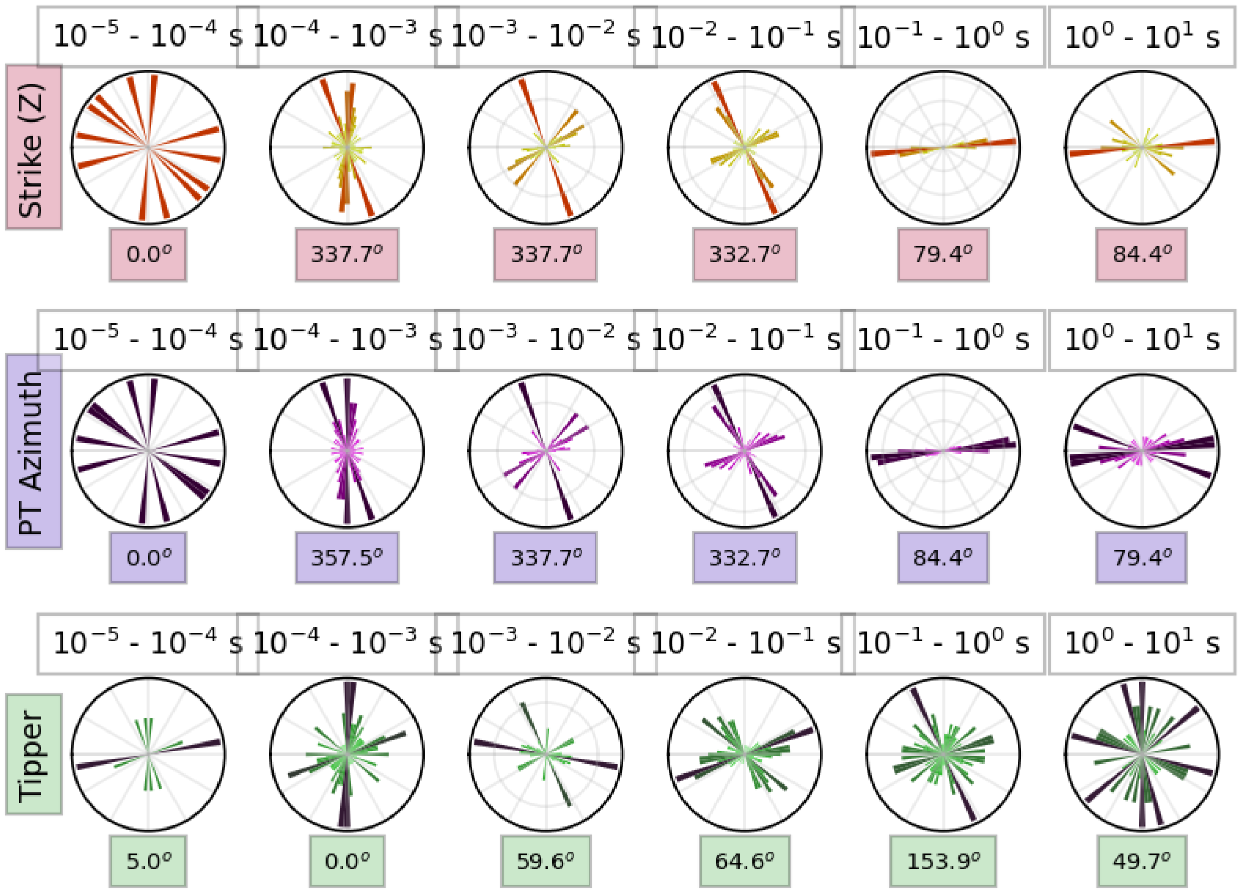
[14]:
```
strike_plot_per_decade = mtd.plot_strike(plot_type=1, print_stats=True)
```

```
Strike statistics for invariant period range 1e-05 to 0.0001 (s) median=290.5 mode=0.0␣
↪mean=218.3
Strike statistics for pt period range 1e-05 to 0.0001 (s) median=290.8 mode=0.0 mean=217.
↪8
Strike statistics for tipper period range 1e-05 to 0.0001 (s) median=35.3 mode=5.0␣
↪mean=32.6
Strike statistics for invariant period range 0.0001 to 0.001 (s) median=286.1 mode=337.7␣
↪mean=198.0
Strike statistics for pt period range 0.0001 to 0.001 (s) median=285.1 mode=357.5␣
↪mean=198.5
Strike statistics for tipper period range 0.0001 to 0.001 (s) median=1.3 mode=0.0␣
↪mean=350.6
Strike statistics for invariant period range 0.001 to 0.01 (s) median=76.4 mode=337.7␣
↪mean=156.6
Strike statistics for pt period range 0.001 to 0.01 (s) median=76.4 mode=337.7 mean=156.8
Strike statistics for tipper period range 0.001 to 0.01 (s) median=334.4 mode=59.6␣
↪mean=354.5
Strike statistics for invariant period range 0.01 to 0.1 (s) median=291.6 mode=332.7␣
↪mean=195.9
Strike statistics for pt period range 0.01 to 0.1 (s) median=291.7 mode=332.7 mean=195.8
Strike statistics for tipper period range 0.01 to 0.1 (s) median=307.4 mode=64.6␣
↪mean=338.6
Strike statistics for invariant period range 0.1 to 1 (s) median=81.5 mode=79.4 mean=93.9
Strike statistics for pt period range 0.1 to 1 (s) median=81.2 mode=84.4 mean=93.7
Strike statistics for tipper period range 0.1 to 1 (s) median=67.1 mode=153.9 mean=37.0
Strike statistics for invariant period range 1 to 10 (s) median=83.5 mode=84.4 mean=146.4
Strike statistics for pt period range 1 to 10 (s) median=79.5 mode=79.4 mean=131.6
Strike statistics for tipper period range 1 to 10 (s) median=357.9 mode=49.7 mean=5.0
```

(continues on next page)

(continued from previous page)

```
23:10:23T09:21:42 | INFO | line:793 |mtpy.imaging.plot_strike | _plot_per_period | Note:␣
↪North is assumed to be 0 and the strike angle is measuredclockwise positive.
```



6. Rotate

Rotation is common for modeling especially 2D because we want the TE mode to be parallel to the profile line and TM to be perpendicular. The strike plots above are good at indicating the dominant strike direction. From above the dominant strike direction is around N85E. There for if we want to rotate the data to a profile parallel with geoelectric strike we rotate by N85W or -85 degrees.

[15]:
```
mtd.rotate(-85)
```

```
23:10:23T09:22:40 | INFO | line:131 |mtpy.core.mt | rotate | Rotated transfer function␣
↪by: -85.000 degrees clockwise
23:10:23T09:22:40 | INFO | line:131 |mtpy.core.mt | rotate | Rotated transfer function␣
↪by: -85.000 degrees clockwise
23:10:23T09:22:40 | INFO | line:131 |mtpy.core.mt | rotate | Rotated transfer function␣
↪by: -85.000 degrees clockwise
23:10:23T09:22:40 | INFO | line:131 |mtpy.core.mt | rotate | Rotated transfer function␣
↪by: -85.000 degrees clockwise
23:10:23T09:22:40 | INFO | line:131 |mtpy.core.mt | rotate | Rotated transfer function␣
↪by: -85.000 degrees clockwise
```

(continues on next page)

(continued from previous page)

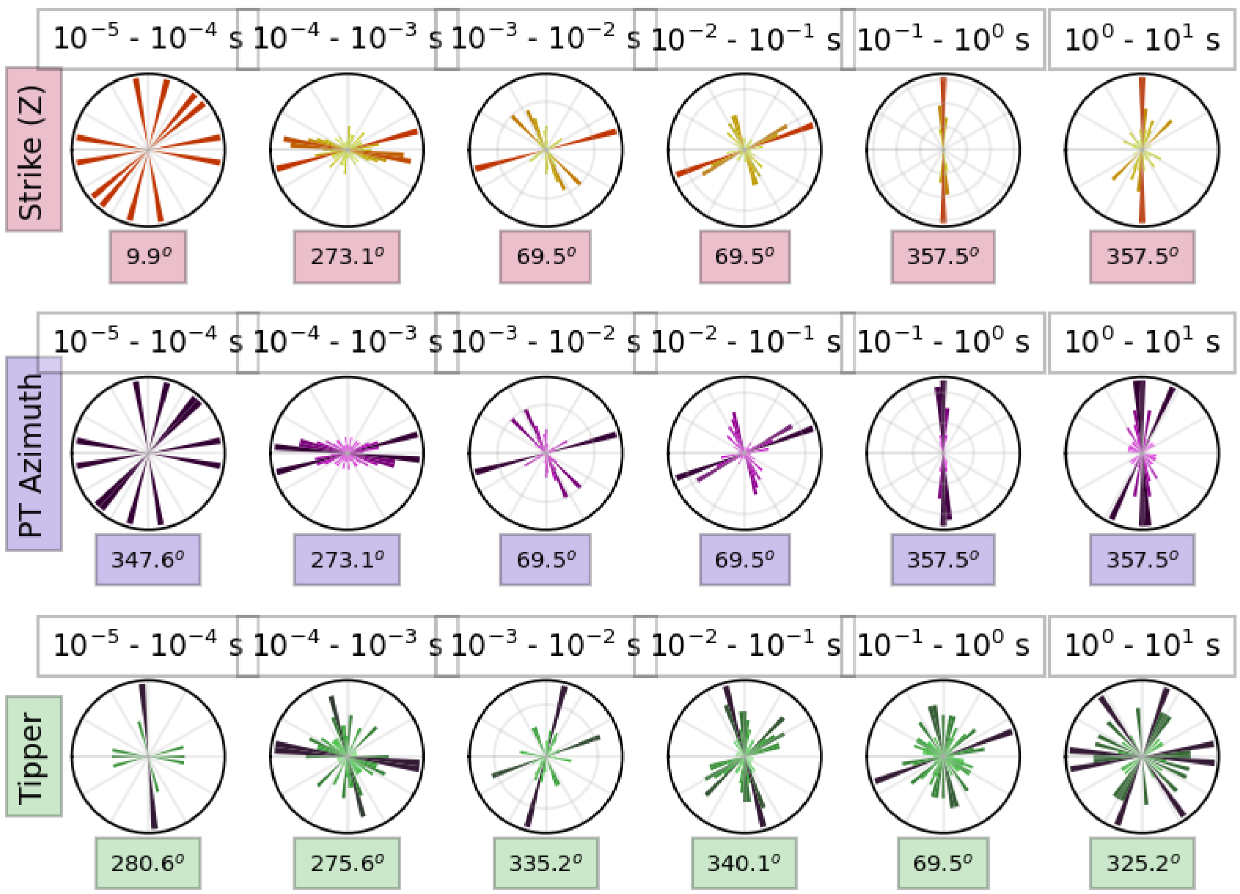
```
23:10:23T09:22:41 | INFO | line:131 |mtpy.core.mt | rotate | Rotated transfer function
↪by: -85.000 degrees clockwise
```

6a. Plot Strike

Now if we plot the strike we see that the dominant strike is near 0. Of course you can tweek the rotation angle to get to 0, but this is close enough for demonstration purposes.

[16]:
```
rotated_strike = mtd.plot_strike(plot_type=1)
```

```
23:10:23T09:22:43 | INFO | line:793 |mtpy.imaging.plot_strike | _plot_per_period | Note:
↪North is assumed to be 0 and the strike angle is measuredclockwise positive.
```



7. Interpolate

If you have different transfer functions processed slightly differently then you'll likely have a mismatch in period and for modeling or plotting purposes its nice to have a single period index for all transfer functions.

You can identify the min and max from the data, if that's what you want and then create period from there. From above it looks like the period range is from 10^{-4} to 0.5 seconds. Here we will interpolate onto 23 periods in that range.

[17]:
```
import numpy as np
```

[18]:
```
new_periods = np.logspace(-4, np.log10(0.5), 23)
```

[19]:
```
interpolated_mtd = mtd.interpolate(new_periods, inplace=False)
```

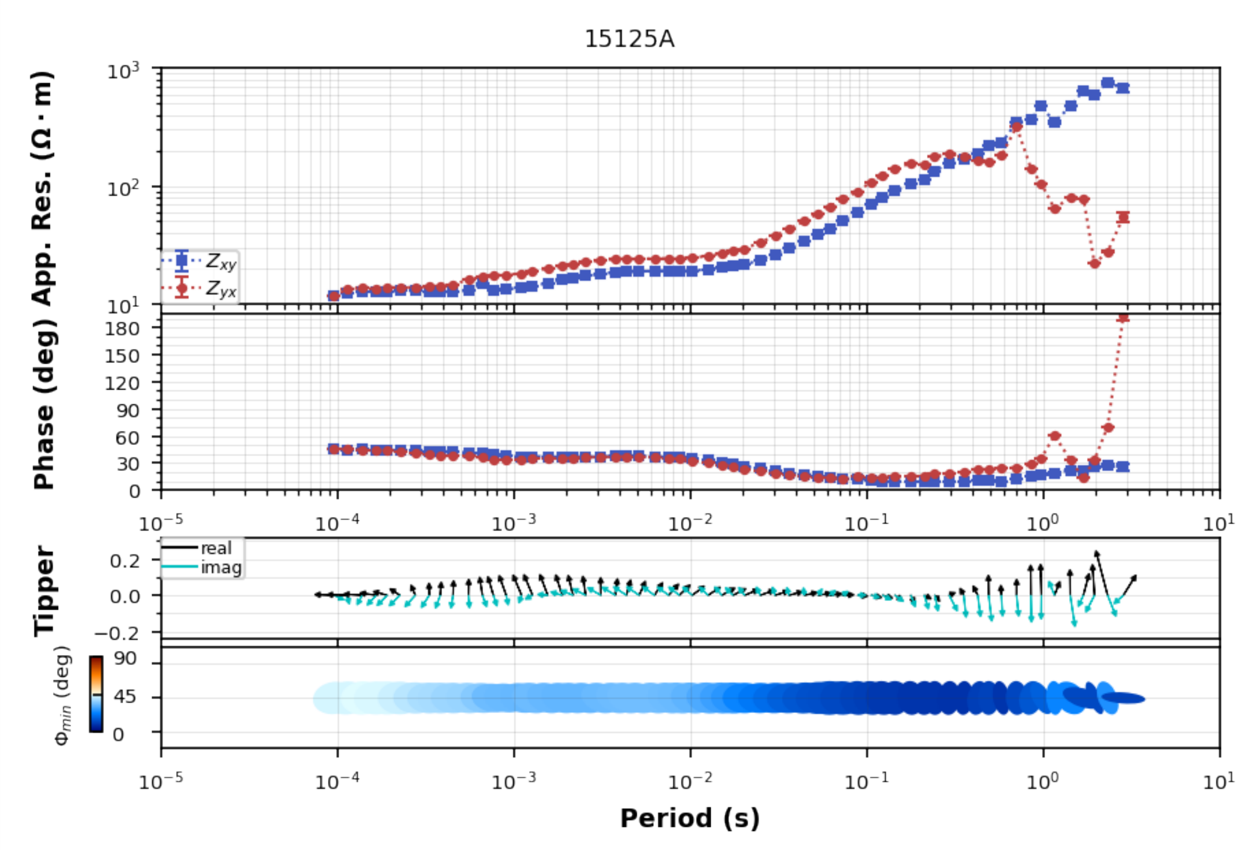
[20]:
```
interpolated_mtd
```

[20]:
```
MTData([('profile.15125A',
          TF( survey='profile', station='15125A', latitude=-22.37, longitude=149.19,␣
↪elevation=200.00 )),
         ('profile.15126A',
          TF( survey='profile', station='15126A', latitude=-22.37, longitude=149.19,␣
↪elevation=200.00 )),
         ('profile.15127A',
          TF( survey='profile', station='15127A', latitude=-22.37, longitude=149.20,␣
↪elevation=201.00 )),
         ('profile.15128A',
          TF( survey='profile', station='15128A', latitude=-22.37, longitude=149.20,␣
↪elevation=200.00 )),
         ('profile.15129A',
          TF( survey='profile', station='15129A', latitude=-22.37, longitude=149.21,␣
↪elevation=202.00 )),
         ('profile.15130A',
          TF( survey='profile', station='15130A', latitude=-22.37, longitude=149.21,␣
↪elevation=201.00 ))])
```

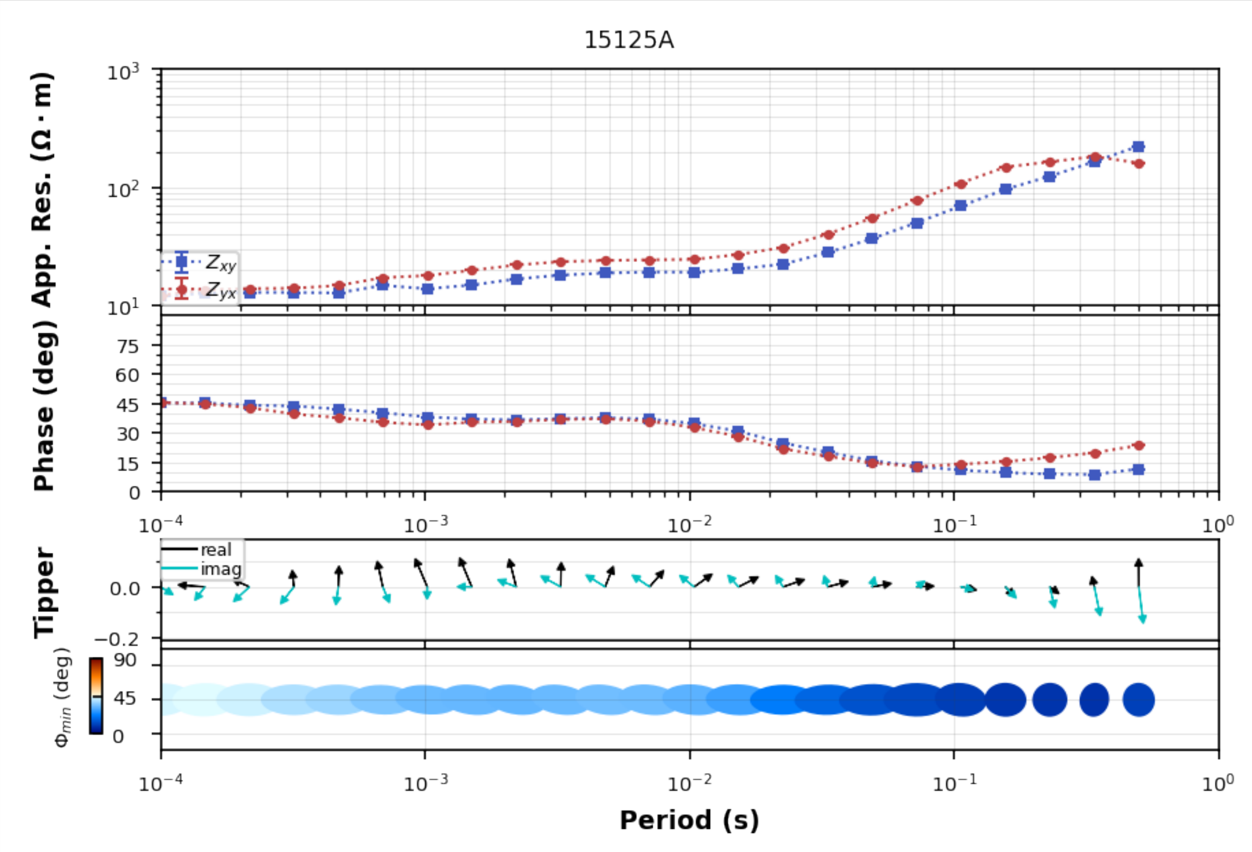
7a. Compare plots

Now we can compare to see how the interpolated transfer function compares to the original. Can plot them individually.

[21]:
```
original = mtd.plot_mt_response("profile.15125A")
```



[22]:
```
interpolated = interpolated_mtd.plot_mt_response("profile.15125A")
```



7b. Compare in same plot

To plot these in the same plot we need to do some manipulating. First change the survey name in the interpolated data. Then create a new `MTData` object that is just two transfer functions with the same name but from the origina and interpolated data sets.

[23]:
```
from mtpy.core.mt_data import MTData
from mtpy.imaging import PlotMultipleResponses
```

[24]:
```
new_interpolated_mtd = MTData()
for mt_object in interpolated_mtd.values():
    mt_object.survey_metadata.id = "interpolated"
    new_interpolated_mtd.add_station(mt_object)
```

[25]:
```
new_interpolated_mtd.station_locations
```

[25]:
```
          survey station   latitude   longitude  elevation datum_epsg  \
0  interpolated  15125A -22.370806  149.188639      200.0       4326
1  interpolated  15126A -22.370639  149.193500      200.0       4326
2  interpolated  15127A -22.371028  149.198417      201.0       4326
3  interpolated  15128A -22.370861  149.203306      200.0       4326
4  interpolated  15129A -22.371083  149.208083      202.0       4326
5  interpolated  15130A -22.371222  149.212972      201.0       4326
```

(continues on next page)

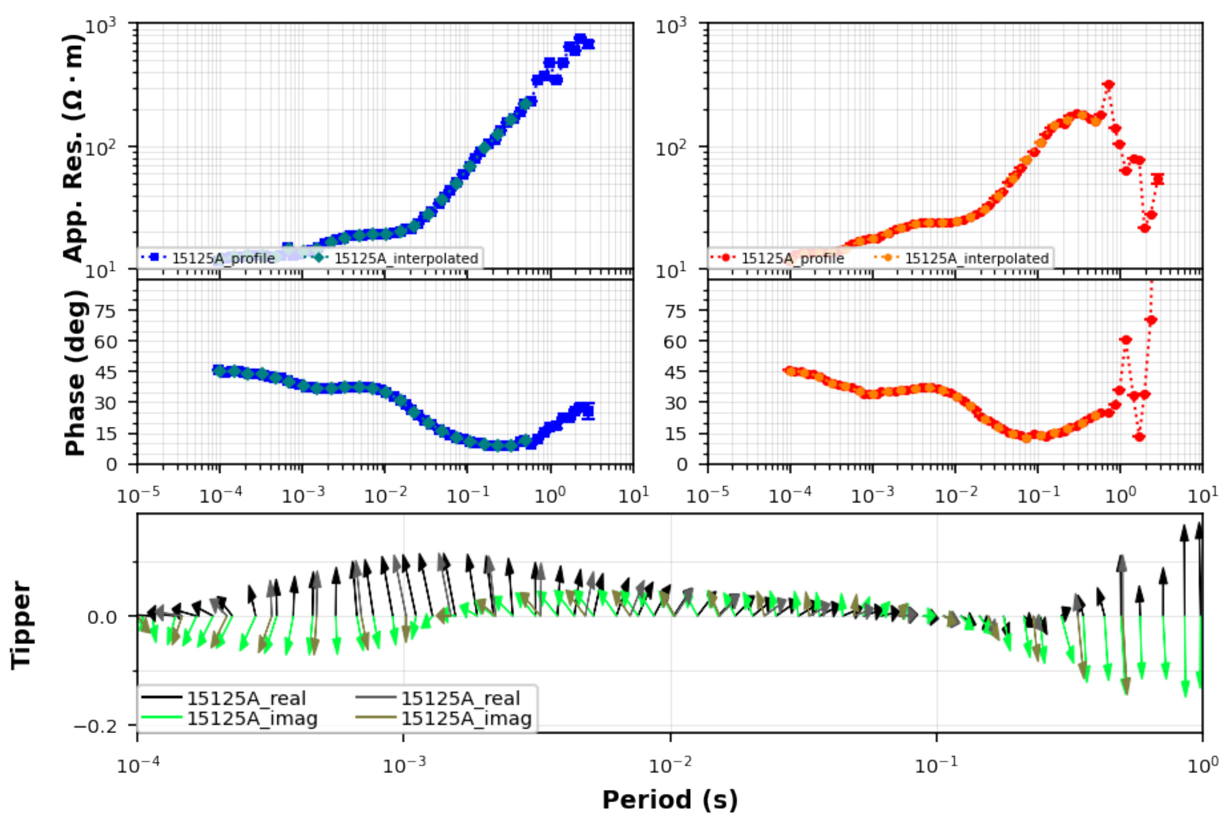
(continued from previous page)

```
             east         north utm_epsg   model_east  model_north  \
0  725360.330833  7.524490e+06     32755 -1252.884265    35.819931
1  725861.314778  7.524502e+06     32755  -751.900320    46.986774
2  726367.124612  7.524451e+06     32755  -246.090486    -3.473507
3  726870.972550  7.524462e+06     32755   257.757452     7.618863
4  727362.745811  7.524430e+06     32755   749.530712   -24.206492
5  727866.099363  7.524408e+06     32755  1252.884265   -46.986774

   model_elevation  profile_offset
0            200.0             0.0
1            200.0             0.0
2            201.0             0.0
3            200.0             0.0
4            202.0             0.0
5            201.0             0.0
```

[26]:
```
compare_mtd = MTData()
compare_mtd.add_station(mtd.get_station("15125A", "profile"))
compare_mtd.add_station(new_interpolated_mtd.get_station("15125A", "interpolated"))
```

[27]:
```
compare_same_plot = PlotMultipleResponses(
    compare_mtd, plot_style="compare", plot_tipper="yri", fig_num=4
)
```

```
<Figure size 432x288 with 0 Axes>
```



8. Occam 2D

Occam2D deGroot-Heldlin and Constable (1990) is a classic 2D inversion program. To use it you will have to compile it on your machine. For details see https://marineemlab.ucsd.edu/Projects/Occam/index.html.

Here we can create input files for Occam2D. Our data is already in an E-W profile and geoelectric strike suggests that's a relatively good start. The dominant strike appears to be around N30W.

To get an accurate model of the data we want the profile line and the dominant geoelectric strike of the data to be perpendicular. To achieve that you can either project the stations onto a profile perpendicular to the geoelectric strike, or rotate the station data to be perpendicular to a map profile. Here, we will project the stations onto a profile perpendicular to geoelectric strike.

8a. Project stations to Geoelectric Strike

When using geoelectric strike to project stations, this is saying that the stations should project onto a profile line that is perpendicular to geoelectric strike such that when you model the data the TE mode (electric along strike) is perpendicular to the profile and the TM mode (electric perpendicular to strike) is parallel to the profile. You can read much more complete explanations looking up MT papers from the 90's. Wannamaker et al. (1984) is a good start.

[28]:
```
x1, y1, x2, y2, strike_profile = interpolated_mtd.generate_profile_from_strike(-30)
strike_profile
```

[28]:
```
{'slope': 1.1253388328842984, 'intercept': -190.2589909890415}
```

[29]:
```
geoelectric_strike_mtd = interpolated_mtd.get_profile(x1, y1, x2, y2, 5000)
```

[30]:
```
geoelectric_strike_mtd
```

[30]:
```
MTData([('interpolated.15125A',
         TF( survey='interpolated', station='15125A', latitude=-22.37, longitude=149.19,␣
↪elevation=200.00 )),
        ('interpolated.15126A',
         TF( survey='interpolated', station='15126A', latitude=-22.37, longitude=149.19,␣
↪elevation=200.00 )),
        ('interpolated.15127A',
         TF( survey='interpolated', station='15127A', latitude=-22.37, longitude=149.20,␣
↪elevation=201.00 )),
        ('interpolated.15128A',
         TF( survey='interpolated', station='15128A', latitude=-22.37, longitude=149.20,␣
↪elevation=200.00 )),
        ('interpolated.15129A',
         TF( survey='interpolated', station='15129A', latitude=-22.37, longitude=149.21,␣
↪elevation=202.00 )),
        ('interpolated.15130A',
         TF( survey='interpolated', station='15130A', latitude=-22.37, longitude=149.21,␣
↪elevation=201.00 ))])
```

8b. Compute Model Errors

For 2D modeling its usually more advantageous to invert the apparent resisitivity and phase of the TE and TM modes. We can set the error for each.

[31]:
```
occam2d_object = geoelectric_strike_mtd.to_occam2d_data(geoelectric_strike=-30, profile_
↪angle=60)
```

Set resistivity error

Its common to give the resistivity components a more uncertainty because of static shifts. Here we will give each component a 20% absolute error and set it to a logarithmic representation.

[38]:
```
occam2d_object.dataframe["res_xy_model_error"] = (20 / 100) / np.log(10.)
occam2d_object.dataframe["res_yx_model_error"] = (20 / 100) / np.log(10.)
```

Set Phase error

The phase is insensitive to near surface heterogeneties and can have smaller uncertainties. Here we give 2.5%.

[43]:
```
occam2d_object.dataframe["phase_xy_model_error"] = (5 / 100.) * 57. / 2.
occam2d_object.dataframe["phase_yx_model_error"] = (2.5 / 100.) * 57. / 2.
```

Write the data file

Now we can write the data file and we should pick which components to invert for. By looking at the help we can see which combinations are supported. We will use mode 4 to invert both TE and TM modes in log space.

[44]:
```
help(occam2d_object)
```

```
Help on Occam2DData in module mtpy.modeling.occam2d.data object:

class Occam2DData(builtins.object)
 |  Occam2DData(dataframe=None, center_point=None, **kwargs)
 |
 |  Reads and writes data files and more.
 |
 |  Inherets Profile, so the intended use is to use Data to project stations
 |  onto a profile, then write the data file.
 |
 |  ===================== =====================================================
 |  Model Modes           Description
 |  ===================== =====================================================
 |  1 or log_all          Log resistivity of TE and TM plus Tipper
 |  2 or log_te_tip       Log resistivity of TE plus Tipper
 |  3 or log_tm_tip       Log resistivity of TM plus Tipper
 |  4 or log_te_tm        Log resistivity of TE and TM
 |  5 or log_te           Log resistivity of TE
 |  6 or log_tm           Log resistivity of TM
 |  7 or all              TE, TM and Tipper
```

(continues on next page)

(continued from previous page)

```
|   8 or te_tip           TE plus Tipper
|   9 or tm_tip           TM plus Tipper
|   10 or te_tm           TE and TM mode
|   11 or te              TE mode
|   12 or tm              TM mode
|   13 or tip             Only Tipper
|   ===================== =====================================================
|
|
|   :Example Write Data File: ::
|
|       >>> from mtpy.modeling.occam2d import Data
|       >>> occam_data_object = Data()
|       >>> occam_data_object.read_data_file(r"path/to/data/file.dat")
|       >>> occam_data_object.model_mode = 2
|       >>> occam_data_object.write_data_file(r"path/to/new/data/file_te.dat")
|
|   Methods defined here:
|
|   __init__(self, dataframe=None, center_point=None, **kwargs)
|       Initialize self.  See help(type(self)) for accurate signature.
|
|   __repr__(self)
|       Return repr(self).
|
|   __str__(self)
|       Return str(self).
|
|   mask_from_datafile(self, mask_datafn)
|       reads a separate data file and applies mask from this data file.
|       mask_datafn needs to have exactly the same frequencies, and station names
|       must match exactly.
|
|   read_data_file(self, data_fn=None)
|       Read in an existing data file and populate appropriate attributes
|           * data
|           * data_list
|           * freq
|           * station_list
|           * station_locations
|
|       Arguments:
|       -----------
|           **data_fn** : string
|                         full path to data file
|                         *default* is None and set to save_path/fn_basename
|
|       :Example: ::
|
|           >>> import mtpy.modeling.occam2d as occam2d
|           >>> ocd = occam2d.Data()
|           >>> ocd.read_data_file(r"/home/Occam2D/Line1/Inv1/Data.dat")
```

(continues on next page)

(continued from previous page)

```
|
|  write_data_file(self, data_fn=None)
|      Write a data file.
|
|      Arguments:
|      -----------
|          **data_fn** : string
|                        full path to data file.
|                        *default* is save_path/fn_basename
|
|      If there data is None, then _fill_data is called to create a profile,
|      rotate data and get all the necessary data.  This way you can use
|      write_data_file directly without going through the steps of projecting
|      the stations, etc.
|
|      :Example: ::
|          >>> edipath = r"/home/mt/edi_files"
|          >>> slst = ['mt{0:03}'.format(ss) for ss in range(1, 20)]
|          >>> ocd = occam2d.Data(edi_path=edipath, station_list=slst)
|          >>> ocd.save_path = r"/home/occam/line1/inv1"
|          >>> ocd.write_data_file()
|
|  ----------------------------------------------------------------------
|  Readonly properties defined here:
|
|  frequencies
|
|  n_data
|
|  n_frequencies
|
|  n_stations
|
|  offsets
|
|  stations
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  data_filename
|
|  dataframe
```

[45]:
```
occam2d_object.model_mode = "4"
```

[46]:
```
occam2d_object.write_data_file(Path().joinpath("occam2d_example.dat"))
```

1.2.4 MTData: Grid Example

It is becoming more common for MT data to be collected in a grid rather than a profile because of 3D inversions. In this example we can take a look at how to work with MT data collected on a grid. We will use the same MTCollection from the first example.

1. Load data

First we will open the MTCollection and change the working dataframe to get only those station in the 'grid' survey.

[1]:
```
from pathlib import Path
from mtpy import MTCollection
```

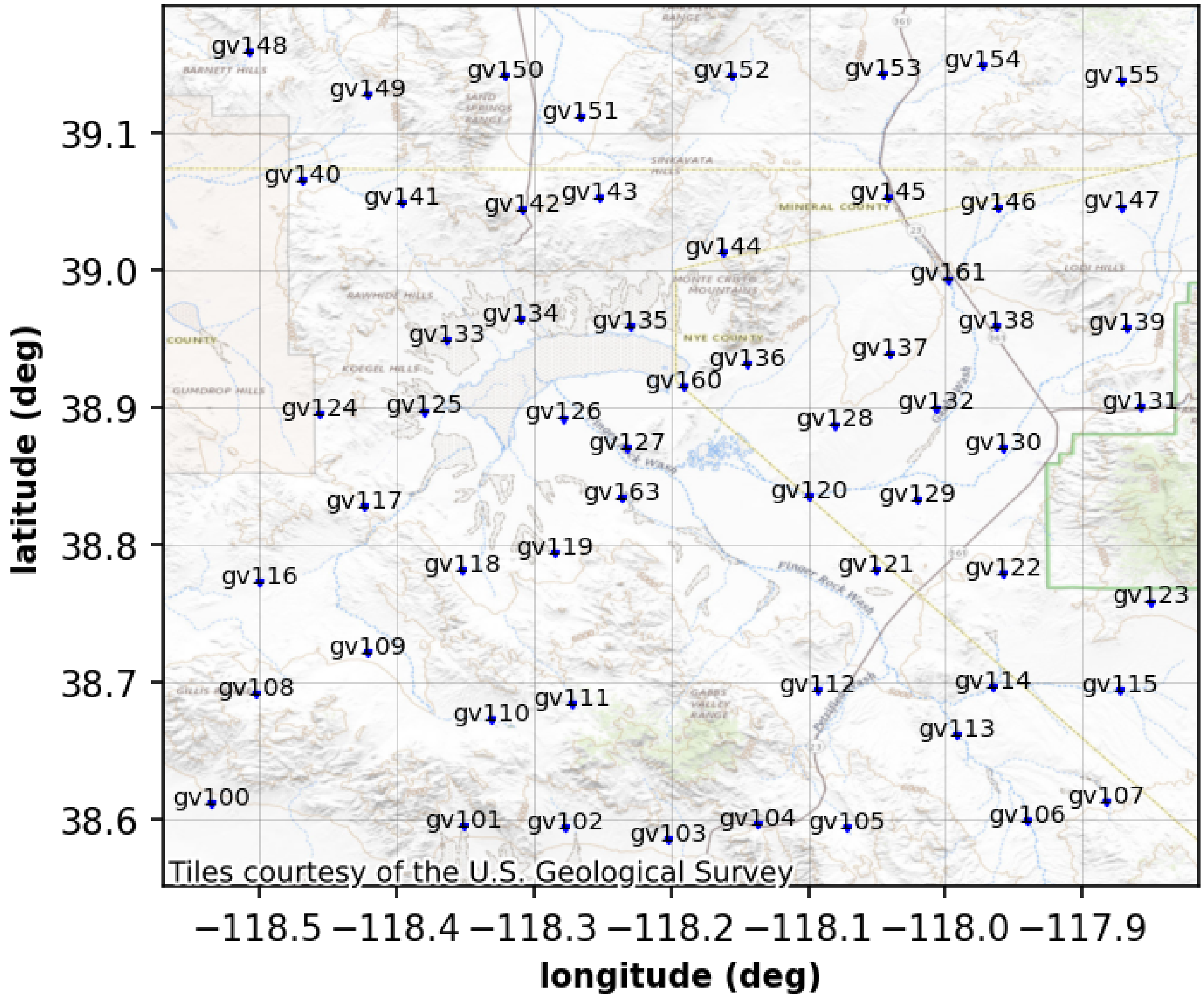
[2]:
```
with MTCollection() as mtc:
    mtc.open_collection(Path().cwd().joinpath("test_mt_collection.h5"))
    mtc.working_dataframe = mtc.master_dataframe.loc[mtc.master_dataframe.survey == "grid
↪"]
    mtd = mtc.to_mt_data()
```

```
23:10:23T09:44:34 | INFO | line:760 |mth5.mth5 | close_mth5 | Flushing and closing C:\
↪Users\jpeacock\OneDrive - DOI\Documents\GitHub\mtpy-v2\docs\source\notebooks\test_mt_
↪collection.h5
```

2. Plot Station Locations

To make sure we got the data we expected, we can plot the station locations. You can change the basemap, see providers for more details

[3]:
```
station_plot = mtd.plot_stations()
```

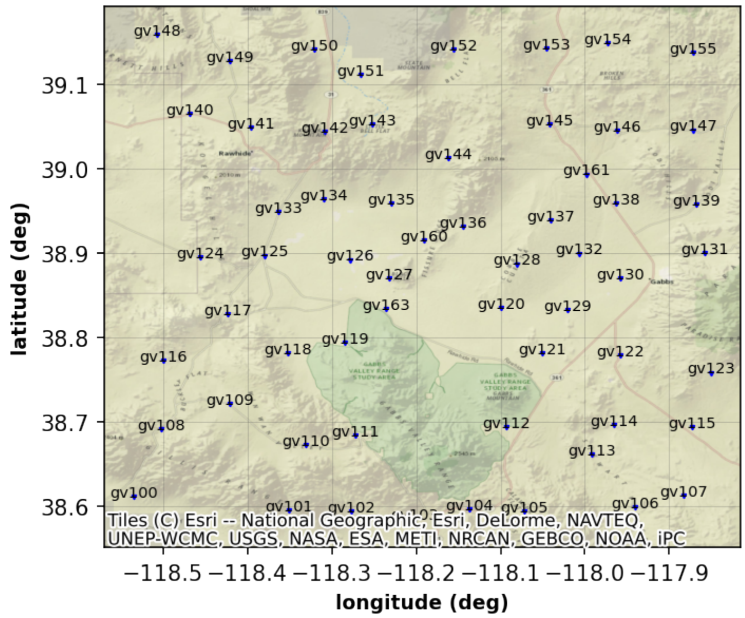


2a. Change basemap

Here is an example of how to change the basemap. We it to use the ESRI terrain map.

[4]:
```
import contextily as cx
```

[5]:
```
station_plot.cx_source = cx.providers.Esri.NatGeoWorldMap
station_plot.redraw_plot()
```

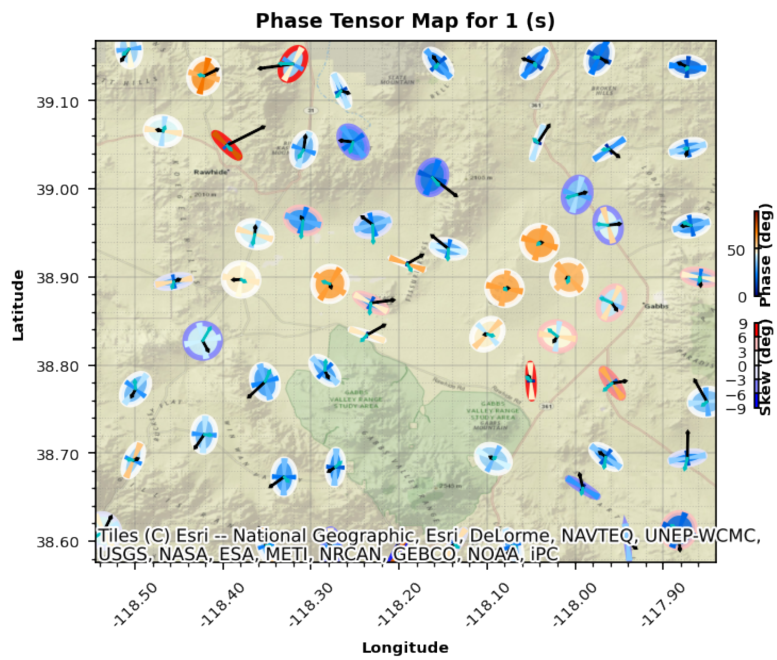


3. Plot Phase Tensor Map

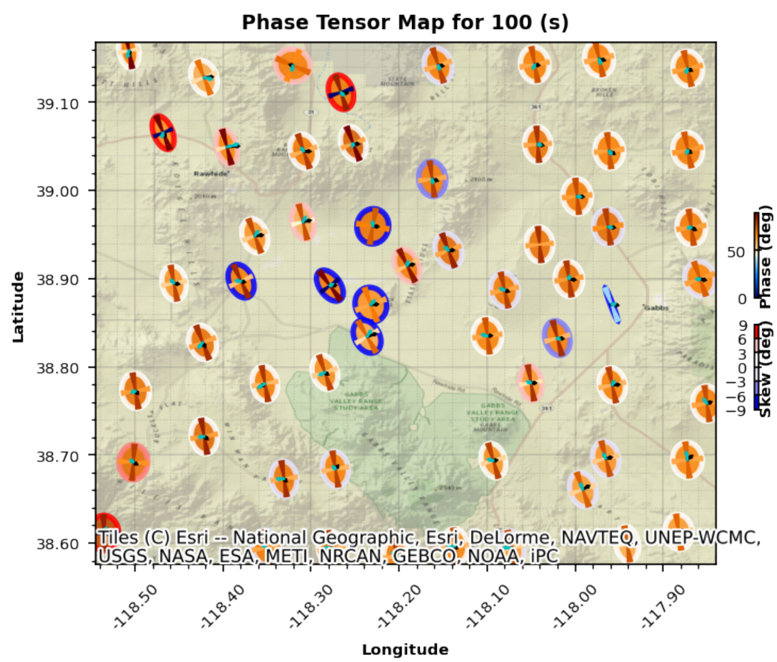
Now that we seem to have the correct data, lets plot phase tensor maps. These are broadband data with induction vectors, so lets plot those too. The phase tensor map is busy so lets identify key aspects.

- Ellipse shape: the ellipses often elongate in the preferred direction of current flow
- Ellipse face color: by default are colored by the parameter `phimin` but can be changed by setting the attribute `pt_map.ellipse_colorby`. The `phimin` gives the lower bounds on how the subsurface resistivity is changing, where reds are becoming more conductive and blues are becoming more resistive.
- Ellipse edge color: by default is colored by the `skew` angle which is indicative of dimensionality with high skew being 3D. Also the color indicates in which direction currents are being skewed.
- Wedges: the long axis is `phimax` and the short axis are `phimin`.
- Arrows: black are real induction vectors and blue are imaginary induction vectors.

[6]:
```
pt_map = mtd.plot_phase_tensor_map(
    plot_tipper="yri",
    cx_source=cx.providers.Esri.NatGeoWorldMap,
    ellipse_size=.02,
    arrow_size=.05
)
```



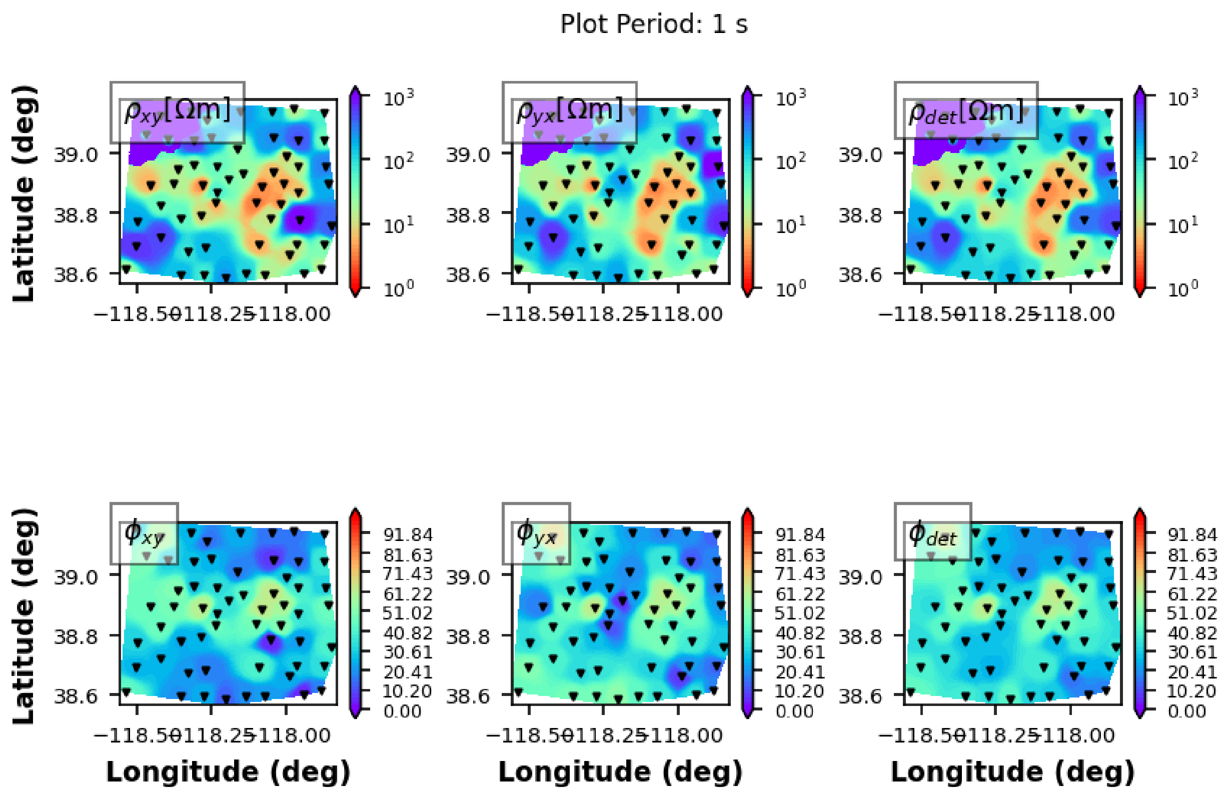
[7]:
```python
pt_map.plot_period = 100
pt_map.redraw_plot()
```



4. Plot Resistivity and Phase Maps

An informative first order check on the data is to plot apparent resistivity and phase maps of the data. This can help identify odd stations for further investigations and get a general idea of subsurface resistivity changes. These plots are a little messy in a Jupyter Notebook.

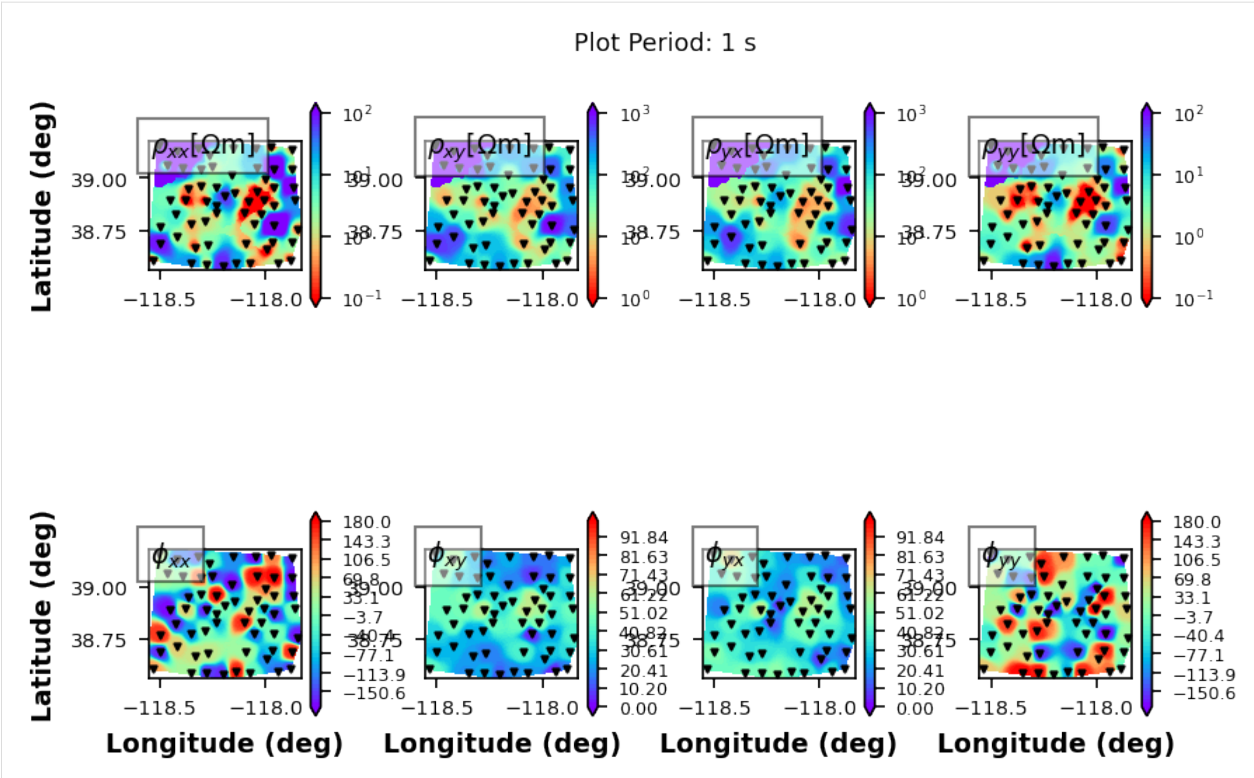
[8]:
```
rp_map = mtd.plot_resistivity_phase_maps(plot_det=True, subplot_wspace=.45, marker_
↪size=4)
```



4a. Plot Diagonal Components

It can be informative to plot the diagnoal components as well, in this case they outline the basins nicely.

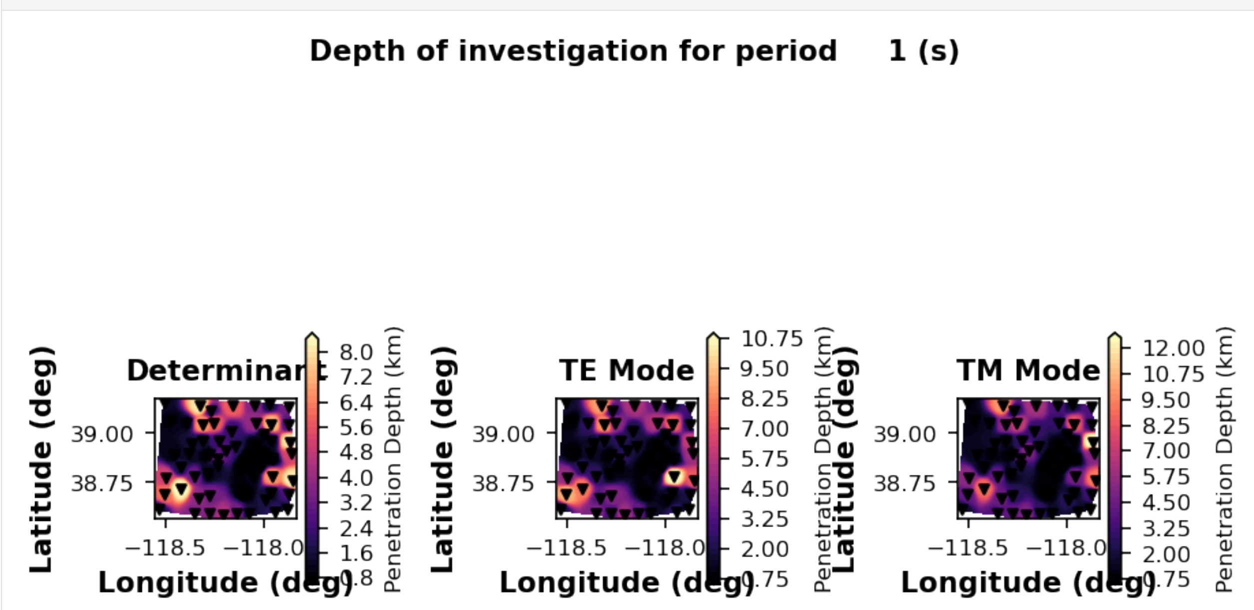
[9]:
```
rp_map.plot_xx = True
rp_map.plot_yy = True
rp_map.plot_det = False
rp_map.redraw_plot()
```



5. Plot Depth of Investigation

It can also be informative to understand how deep the measurements are sensitive to. Here a Niblett-Bostick approximation is used to estimate the depth of penetration for each station at a single period.

[10]:
```
depth_of_penetration = mtd.plot_penetration_depth_map(subplot_wspace=.35)
```



6. Plot Strike

Here we will plot all periods of estimated strike. Notice that the plot includes the strike as estimated from the invariants (left) of Weaver et al. (2002), the phase tensor (middel) of Caldwell et al. (2004), and the induction vector strike.

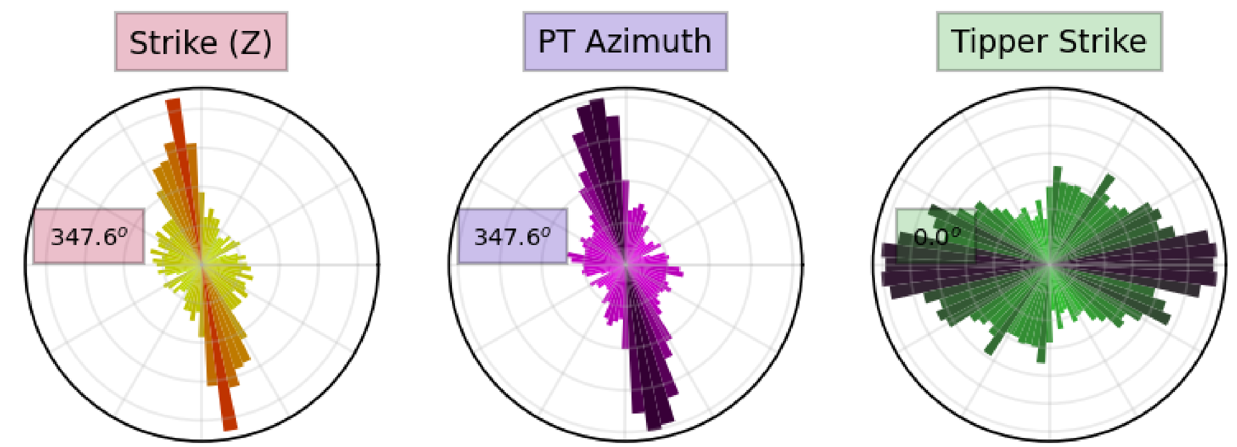
Important: The induction strike points towards good conductors so should therefore be perpendicular to the impedance strike. We left it this way as a sanity check on strike angles.

6a. Plot all periods in one plot

This plot shows all strike estimations for all stations for all periods.

[11]:
```
strike_plot = mtd.plot_strike()
```

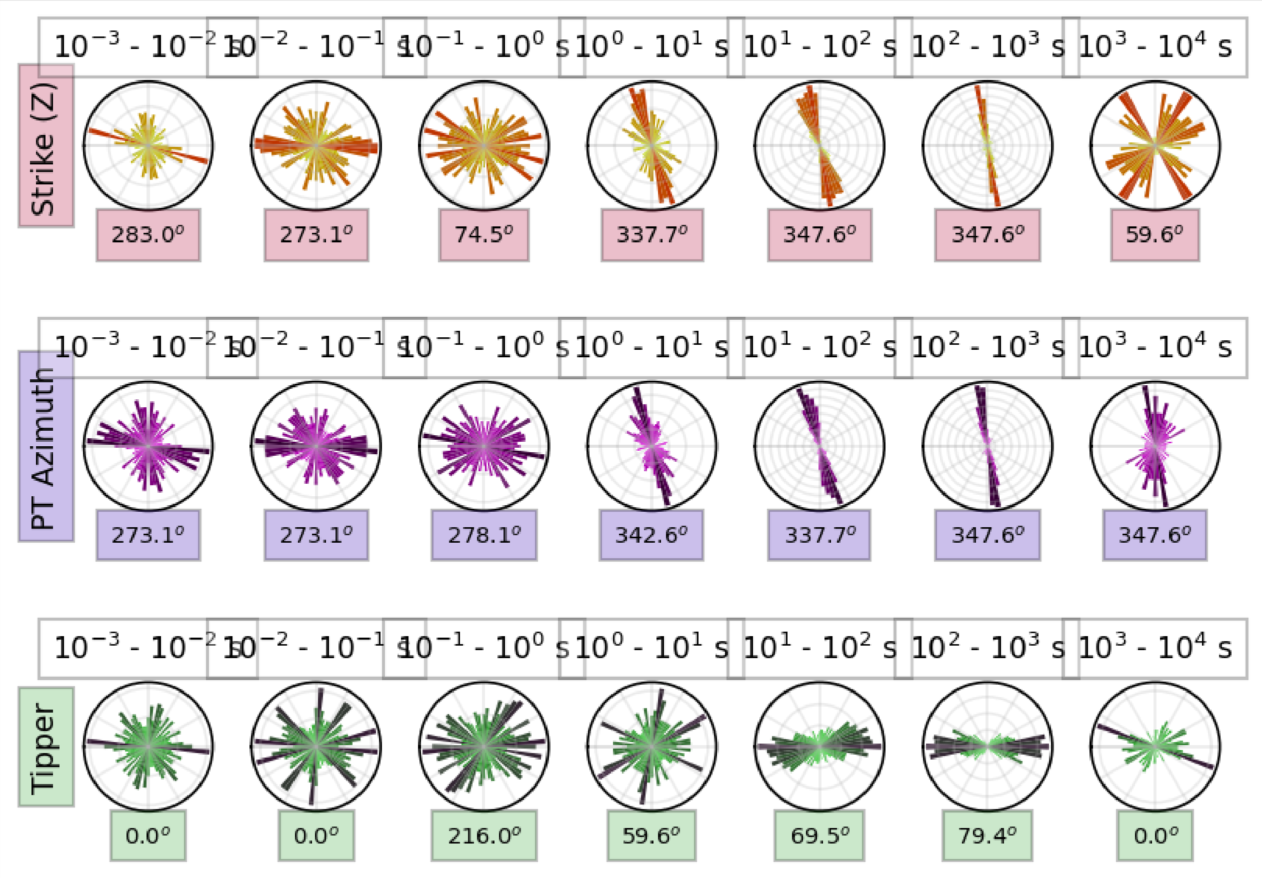
```
23:10:23T09:46:19 | INFO | line:892 |mtpy.imaging.plot_strike | _plot_all_periods | Note:
↪ North is assumed to be 0 and the strike angle is measured clockwise positive.
```



6b. Strike per period

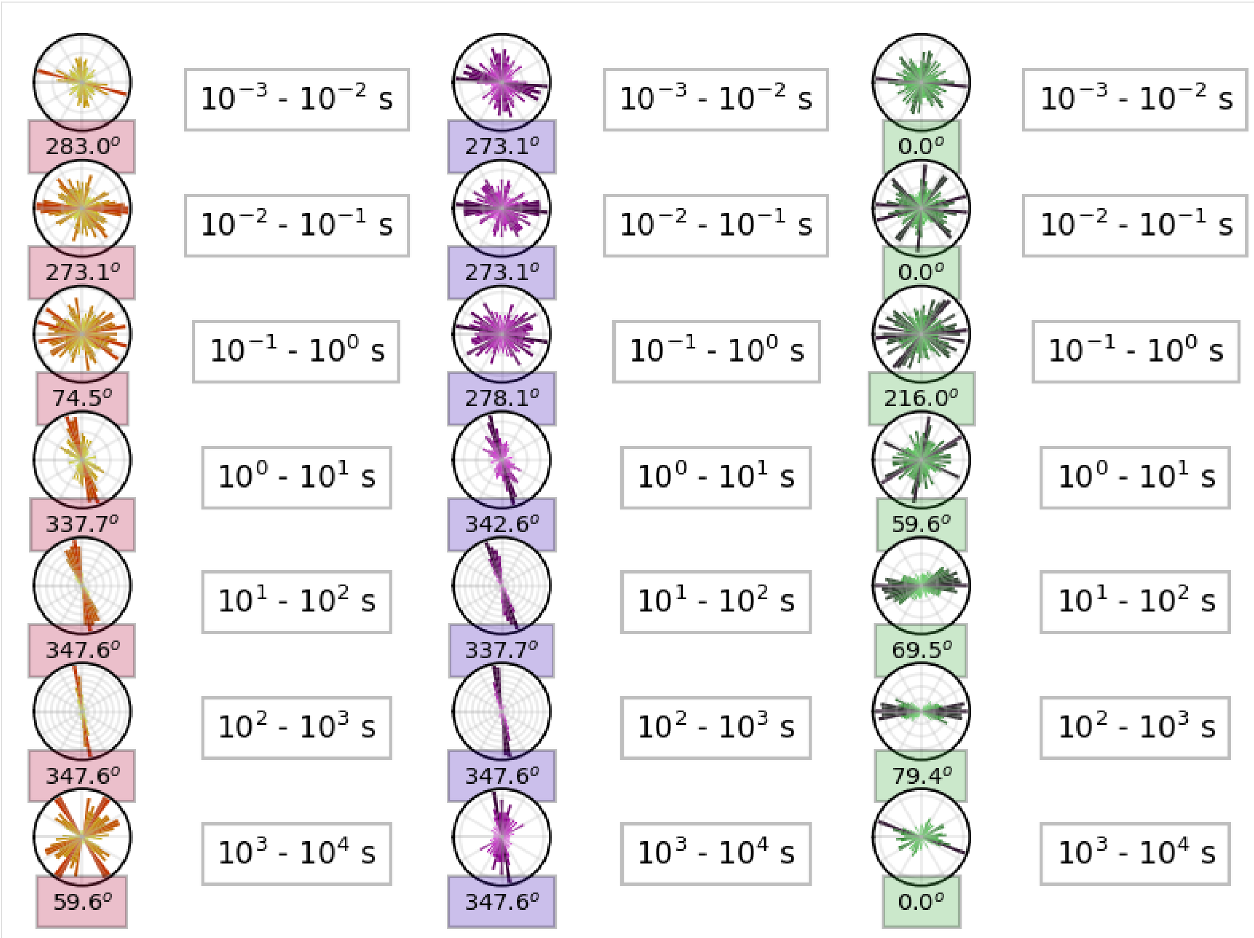
[12]:
```
strike_plot.plot_type = 1
strike_plot.redraw_plot()
```

```
23:10:23T09:46:29 | INFO | line:793 |mtpy.imaging.plot_strike | _plot_per_period | Note:␣
↪North is assumed to be 0 and the strike angle is measuredclockwise positive.
```



[13]:
```
strike_plot.plot_orientation = "vertical"
strike_plot.redraw_plot()
```

```
23:10:23T09:47:25 | INFO | line:793 |mtpy.imaging.plot_strike | _plot_per_period | Note:␣
↪North is assumed to be 0 and the strike angle is measuredclockwise positive.
```



7. Extract a Profile

One adventageous way of looking at grid data is to extract profiles across interesting structures. This can be done by knowing the start and end points of the profile you would like to look at and providing a distance away from the profile line to include.

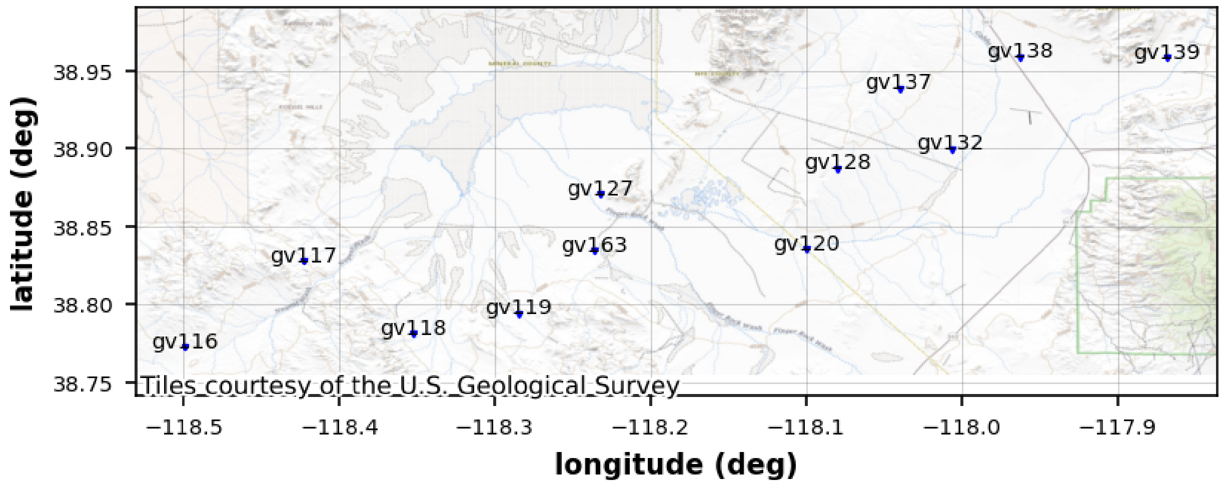
By analyzing the strike direction we can see that a profile approximately N75E would be perpendicular to geoelectric strike. Lets cut across the basins.

[17]:
```
mtd.utm_crs = 32611
```

[22]:
```
%%time
profile = mtd.get_profile(-118.45, 38.78, -117.85, 38.95, 5000)
```

```
Wall time: 2min 48s
```

[23]:
```
profile_stations_plot = profile.plot_stations()
```




```
[ ]:
```

1.2.5 Make 3D Data Files

Inversion programs need input data, MTpy has tools for creating model files for various modeling programs. This is an example for creating a data file for ModEM using the example grid data.

Imports

```
[1]: %matplotlib widget
```

```
[2]: from pathlib import Path

     from mtpy import MTCollection
```

1. Load in Data

Load in the data created in the first example and get only the data collected on a grid. Turn those data into a `MTData` object from which we can manipulate the data.

Tip: Using the `with` context manager will close the MTH5 when finished. This is safer because if something goes wrong the file will always be closed and reduce the chance of corruption.

```
[3]: with MTCollection() as mtc:
         mtc.open_collection(Path().cwd().joinpath("test_mt_collection.h5"))
         mtc.working_dataframe = mtc.master_dataframe.loc[mtc.master_dataframe.survey == "grid
     ↪"]
         mtd = mtc.to_mt_data()
```

```
23:10:20T10:27:58 | INFO | line:760 |mth5.mth5 | close_mth5 | Flushing and closing C:\
↪Users\jpeacock\OneDrive - DOI\Documents\GitHub\mtpy-v2\docs\source\notebooks\test_mt_
↪collection.h5
```

2. Calculate Relative Locations

Most modeling programs are agnostic to geographic coordinates as they use a relative coordinate system usually with (0, 0, 0) at one of the corners or the center of the mesh. Here we assume the center of the station area is the (0, 0) point and relative locations are computed relative to the center point. All relative locations are in meters.

Important: To calculate relative locations you must set the 'utm_epsg' or 'utm_crs' if you have a custom datum. Once you set the 'utm_crs' or 'utm_epsg' easting and northing is estimated for each station in the MTData object. This can take a few seconds if there are a lot of stations.

Here we will set the EPSG number to WGS84 UTM grid 11N (32611).

Tip: To access station locations of the MTData object use MTData.station_locations. This returns a Pandas DataFrame which can be easily manipulated.

Tip: The center point is MTData.center_point and is estimated from the station locations in the MTData object. You can set the center latitude and longitude if you want to offset the center.

```
MTData._center_lat = new_center_latitude
MTData._center_lon = new_center_longitude
MTData._center_elev = new_center_elevation
```

[4]:
```
mtd.center_point
```

[4]:
```
MT Location:
--------------------
  Latitude (deg):   38.871946
  Longitude (deg):  -118.192737
  Elevation (m):    0.0000
  Datum crs:        epsg:4326

  Easting (m):      0.000
  Northing (m):     0.000
  UTM crs:          None

  Model Easting (m):      0.000
  Model Northing (m):     0.000
  Model Elevation (m):    0.000
  Profile Offset (m):     0.000
```

Station Locations

You can have a look at the station locations retreived from the MTCollection. Notice there there are no values for east and north and model_east and model_north yet.

[5]:
```
mtd.station_locations.head(5)
```

[5]:
```
  survey station   latitude   longitude  elevation datum_epsg  east  north  \
0   grid   gv100  38.611381 -118.535261    1437.40       4326   0.0    0.0
1   grid   gv101  38.594561 -118.351111    1540.55       4326   0.0    0.0
2   grid   gv102  38.593692 -118.276822    1554.80       4326   0.0    0.0
3   grid   gv103  38.585283 -118.202481    1543.90       4326   0.0    0.0
4   grid   gv104  38.596456 -118.136547    1801.80       4326   0.0    0.0
```

(continues on next page)

(continued from previous page)

```
  utm_epsg  model_east  model_north  model_elevation  profile_offset
0     None         0.0          0.0          1437.40             0.0
1     None         0.0          0.0          1540.55             0.0
2     None         0.0          0.0          1554.80             0.0
3     None         0.0          0.0          1543.90             0.0
4     None         0.0          0.0          1801.80             0.0
```

Set UTM EPSG

Here we set the UTM EPSG number, which creates a `pyproj.CRS` object for easier projection between coordinate systems, mainly from degrees to meters. If you created your own CRS you can set the CRS directly, see pyproj.CRS for details on creating a custom CRS. Once either the `utm_epsg` or `utm_crs` is set northing and easting for each station in the MTData object is calculated and the station locations are updated, which updates the center point.

[6]:
```
mtd.utm_epsg = 32611
```

[7]:
```
mtd.center_point
```

[7]:
```
MT Location:
--------------------
  Latitude (deg):   38.873786
  Longitude (deg):  -118.196245
  Elevation (m):    0.0000
  Datum crs:        epsg:4326

  Easting (m):      396229.649
  Northing (m):     4303450.537
  UTM crs:          epsg:32611

  Model Easting (m):      396229.649
  Model Northing (m):     4303450.537
  Model Elevation (m):    0.000
  Profile Offset (m):     0.000
```

[8]:
```
mtd.station_locations.head(5)
```

[8]:
```
  survey station   latitude   longitude  elevation datum_epsg           east  \
0   grid   gv100  38.611381 -118.535261    1437.40       4326  366331.327569
1   grid   gv101  38.594561 -118.351111    1540.55       4326  382337.730338
2   grid   gv102  38.593692 -118.276822    1554.80       4326  388806.125501
3   grid   gv103  38.585283 -118.202481    1543.90       4326  395268.278608
4   grid   gv104  38.596456 -118.136547    1801.80       4326  401026.349952

          north utm_epsg  model_east  model_north  model_elevation  \
0  4.274770e+06    32611         0.0          0.0          1437.40
1  4.272652e+06    32611         0.0          0.0          1540.55
2  4.272463e+06    32611         0.0          0.0          1554.80
3  4.271442e+06    32611         0.0          0.0          1543.90
4  4.272609e+06    32611         0.0          0.0          1801.80

   profile_offset
```

(continues on next page)

(continued from previous page)

```
0              0.0
1              0.0
2              0.0
3              0.0
4              0.0
```

Calculate Relative Locations

Now that we have easting and northing we can estimate relative locations in model coordinates, which assumes the center of the station area is (0, 0). If you want to offset the center set these values to your desired center location.

```
MTData._center_lat = new_center_latitude
MTData._center_lon = new_center_longitude
MTData._center_elev = new_center_elevation
```

The method to use is `compute_relative_locations()`. It iterates over the stations and removes the center point from the easting and northing.

```
[9]: mtd.compute_relative_locations()
```

Now we have relative locations.

```
[10]: mtd.station_locations.head(5)
```

```
[10]:  survey station   latitude   longitude  elevation datum_epsg           east  \
0   grid   gv100  38.611381 -118.535261    1437.40       4326  366331.327569
1   grid   gv101  38.594561 -118.351111    1540.55       4326  382337.730338
2   grid   gv102  38.593692 -118.276822    1554.80       4326  388806.125501
3   grid   gv103  38.585283 -118.202481    1543.90       4326  395268.278608
4   grid   gv104  38.596456 -118.136547    1801.80       4326  401026.349952

          north utm_epsg    model_east   model_north  model_elevation  \
0  4.274770e+06    32611 -29898.321606 -28680.329764          1437.40
1  4.272652e+06    32611 -13891.918837 -30798.871440          1540.55
2  4.272463e+06    32611  -7423.523674 -30987.924124          1554.80
3  4.271442e+06    32611   -961.370567 -32008.380683          1543.90
4  4.272609e+06    32611   4796.700777 -30841.737287          1801.80

   profile_offset
0             0.0
1             0.0
2             0.0
3             0.0
4             0.0
```

3. Compute Model Errors

After getting relative locations we can now calculate model errors. These are often larger than measurement error. People have their favorite method of estimating errors and we have tried to accommodate most. If there isn't one here try adding to `mtpy.modeling.errors` or raise an issue. Supported so far are:

Name	Description	Example
geometric_mean	Geometric mean of the off-diagonal components of the impedance tensor, or components of the induction vectors.	error_value $\cdot\sqrt(Z_{xy} \cdot Z_{yx})$
arithmetic_mean	Arithmetic mean of off-diagonal components of the impedance tensor, or components of the induction vectors.	error_value$\cdot(Z_{xy} + Z_{yx})/2$
row	Error per row of the impedance tensor	error_value $\cdot Z_{xy}$
median	Median of the off-diagnoal components	error_value $\cdot median([Z_{xy}, Zyx])$
eigen	Maximum Eigen value of the impedance tensor	error_value x eigen(Z)
percent	Percent error per component	error_value $\cdot Z_{ij}$
absolute	Absolute error	error_value

Setting Impedance Error

MTData has an attribute `z_model_error` that is an `mtpy.modeling.errors.ModelErrors` object. It has some default values, which seem fine for now. But if you want to change any you can use:

```
mtd.z_model_error.error_type = "row"
mtd.z_model_error.error_value = 10   # for 10 percent
mtd.z.model_error.floor = False      # use the calculated value not as a floor for␣
↪measured errors but absolute value.
```

[11]:
```
mtd.z_model_error
```

[11]:
```
Model Errors:
--------------------
        error_type:    geometric_mean
        error_value:   0.05
        floor:         True
        mode:          impedance
```

Setting Tipper Error

Similar to the impedance MTData has an attribute `t_model_error` and is a `mtpy.modeling.errors.ModelErrors` object.

[12]:
```
mtd.t_model_error
```

[12]:
```
Model Errors:
--------------------
        error_type:    absolute
        error_value:   0.02
        floor:         True
        mode:          tipper
```

Calculate Model Errors

Once you set `z_model_error` and `t_model_error` paramters then run `MTData.compute_model_errors()`

Tip: compute_model_errors accepts key words that can be used to set z_model_error and t_model_error

```
[13]: mtd.compute_model_errors(z_error_value=7)
```

```
[14]: mtd["grid.gv100"].impedance[0:1]
```

```
[14]: <xarray.DataArray 'impedance' (period: 1, output: 2, input: 2)>
array([[[ -86.57589 -714.197j, 1090.806  +2750.944j],
        [-115.5849 +1316.743j, -683.7422 -5020.612j]]])
Coordinates:
  * output   (output) <U2 'ex' 'ey'
  * input    (input) <U2 'hx' 'hy'
  * period   (period) float64 0.001302
Attributes:
    survey:              grid
    project:             Energy Resources Program
    id:                  gv100
    name:                None
    latitude:            38.611380555555556
    longitude:           -118.53526111111111
    elevation:           1437.4
    declination:         12.5
    datum:               WGS84
    acquired_by:         Jared Peacock
    start:               2020-08-19T00:00:00+00:00
    end:                 1980-01-01T00:00:00+00:00
    runs_processed:      ['gv100a']
    coordinate_system:   geographic
```

```
[15]: mtd["grid.gv100"].impedance_error[0:1]
```

```
[15]: <xarray.DataArray 'impedance_error' (period: 1, output: 2, input: 2)>
array([[[ 98.57316572, 237.08819034],
        [249.77736086, 600.78698388]]])
Coordinates:
  * output   (output) <U2 'ex' 'ey'
  * input    (input) <U2 'hx' 'hy'
  * period   (period) float64 0.001302
Attributes:
    survey:              grid
    project:             Energy Resources Program
    id:                  gv100
    name:                None
    latitude:            38.611380555555556
    longitude:           -118.53526111111111
    elevation:           1437.4
    declination:         12.5
    datum:               WGS84
    acquired_by:         Jared Peacock
    start:               2020-08-19T00:00:00+00:00
```

(continues on next page)

(continued from previous page)

```
    end:                1980-01-01T00:00:00+00:00
    runs_processed:     ['gv100a']
    coordinate_system:  geographic
```

[16]:
```
mtd["grid.gv100"].impedance_model_error[0:1]
```

[16]:
```
<xarray.DataArray 'impedance_model_error' (period: 1, output: 2, input: 2)>
array([[[138.44510952, 237.08819034],
        [249.77736086, 600.78698388]]])
Coordinates:
  * output   (output) <U2 'ex' 'ey'
  * input    (input) <U2 'hx' 'hy'
  * period   (period) float64 0.001302
Attributes:
    survey:             grid
    project:            Energy Resources Program
    id:                 gv100
    name:               None
    latitude:           38.611380555555556
    longitude:          -118.53526111111111
    elevation:          1437.4
    declination:        12.5
    datum:              WGS84
    acquired_by:        Jared Peacock
    start:              2020-08-19T00:00:00+00:00
    end:                1980-01-01T00:00:00+00:00
    runs_processed:     ['gv100a']
    coordinate_system:  geographic
```

You can see with the floor set measurement error are used if the value is above the estimated error from the parameters set above. And using an absolute error we see the value is set for all elements in the induction vectors.

[17]:
```
mtd["grid.gv100"].tipper_model_error[0:1]
```

[17]:
```
<xarray.DataArray 'tipper_model_error' (period: 1, output: 1, input: 2)>
array([[[0.02, 0.02]]])
Coordinates:
  * output   (output) <U2 'hz'
  * input    (input) <U2 'hx' 'hy'
  * period   (period) float64 0.001302
Attributes:
    survey:             grid
    project:            Energy Resources Program
    id:                 gv100
    name:               None
    latitude:           38.611380555555556
    longitude:          -118.53526111111111
    elevation:          1437.4
    declination:        12.5
    datum:              WGS84
    acquired_by:        Jared Peacock
    start:              2020-08-19T00:00:00+00:00
    end:                1980-01-01T00:00:00+00:00
```

(continues on next page)

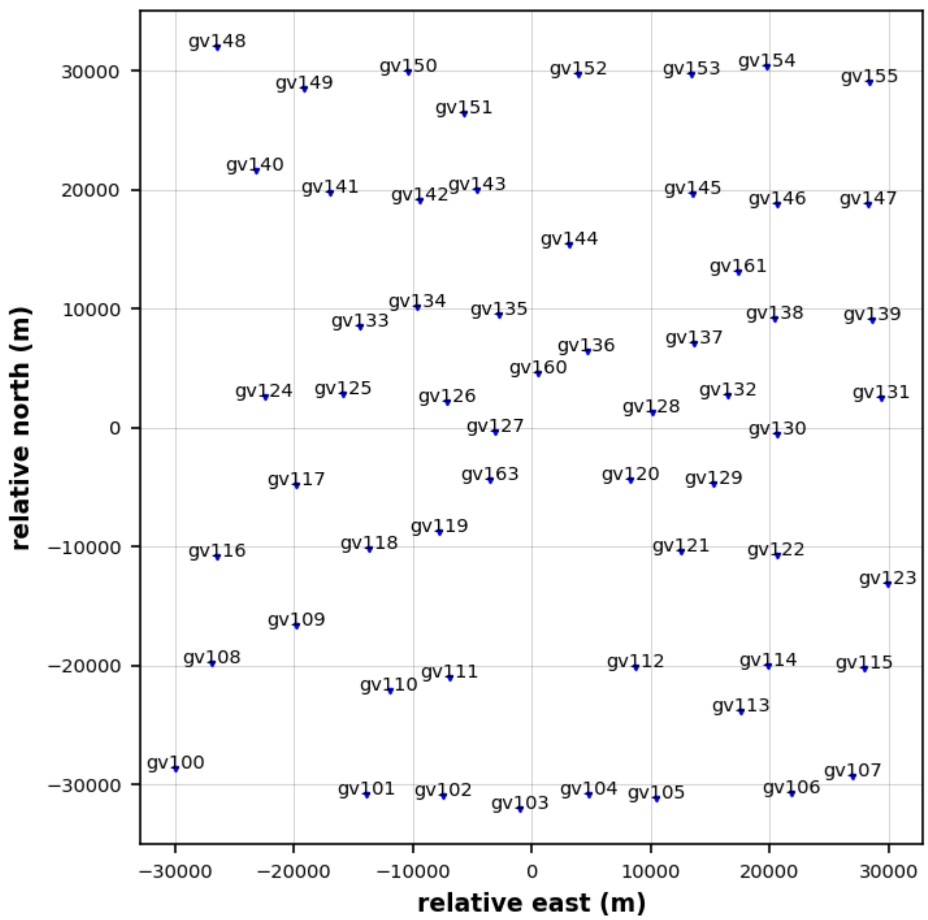
(continued from previous page)

```
    runs_processed:      ['gv100a']
    coordinate_system:   geographic
```

4. Plot Stations

Plot relative station locations.

[18]:
```python
plot_stations = mtd.plot_stations(model_locations=True)
```



4. Write to file

Now that the data have been updated with locations and weights we can write to a file. Here we will write to a file for ModEM.

[19]:
```python
modem_data_object = mtd.to_modem_data(Path().cwd().joinpath("example_modem_data_file.dat
↪"))
```

```
23:10:20T10:28:12 | WARNING | line:678 |mtpy.modeling.modem.data | _check_for_errors_of_
↪zero | Found errors with values of 0 in zxx 40 times. Setting error as zxx x 0.07.
23:10:20T10:28:12 | WARNING | line:678 |mtpy.modeling.modem.data | _check_for_errors_of_
```

(continues on next page)

(continued from previous page)

```
↪zero | Found errors with values of 0 in zxy 40 times. Setting error as zxy x 0.07.
23:10:20T10:28:12 | WARNING | line:678 |mtpy.modeling.modem.data | _check_for_errors_of_
↪zero | Found errors with values of 0 in zyx 40 times. Setting error as zyx x 0.07.
23:10:20T10:28:12 | WARNING | line:678 |mtpy.modeling.modem.data | _check_for_errors_of_
↪zero | Found errors with values of 0 in zyy 40 times. Setting error as zyy x 0.07.
23:10:20T10:28:12 | WARNING | line:709 |mtpy.modeling.modem.data | _check_for_too_small_
↪errors | Found errors with values less than 0.02 in zxx 3 times. Setting error as zxx␣
↪x 0.07.
23:10:20T10:28:12 | WARNING | line:709 |mtpy.modeling.modem.data | _check_for_too_small_
↪errors | Found errors with values less than 0.02 in zxy 10 times. Setting error as zxy␣
↪x 0.07.
23:10:20T10:28:12 | WARNING | line:709 |mtpy.modeling.modem.data | _check_for_too_small_
↪errors | Found errors with values less than 0.02 in zyx 2 times. Setting error as zyx␣
↪x 0.07.
23:10:20T10:28:12 | WARNING | line:709 |mtpy.modeling.modem.data | _check_for_too_small_
↪errors | Found errors with values less than 0.02 in tzx 3 times. Setting error as tzx␣
↪x 0.02.
23:10:20T10:28:12 | WARNING | line:709 |mtpy.modeling.modem.data | _check_for_too_small_
↪errors | Found errors with values less than 0.02 in tzy 2 times. Setting error as tzy␣
↪x 0.02.
23:10:20T10:28:14 | INFO | line:807 |mtpy.modeling.modem.data | write_data_file | Wrote␣
↪ModEM data file to C:\Users\jpeacock\OneDrive - DOI\Documents\GitHub\mtpy-v2\docs\
↪source\notebooks\example_modem_data_file.dat
```

[20]:
```
modem_data_object
```

[20]:
```
ModEM Data Object:
        Number of impedance stations: 59
        Number of tipper stations: 59
        Number of phase tensor stations: 0
        Number of periods:  168
        Period range (s):
                Min: 0.0013021
                Max: 2048
        Rotation angle:     0
        Data center:
                Latitude:   38.8738 deg         Northing: 4303450.5370 m
                Longitude: -118.1962 deg        Easting: 396229.6492 m
                Datum epsg: 4326                        UTM epsg:   32611
                Elevation:  0.0 m
        Impedance data:     True
        Tipper data:        True
        Inversion Mode:   Full_Impedance, Full_Vertical_Components
```

[ ]:

1.2.6 Make Structured 3D Mesh

A common input into 3D inversions is a structured mesh with equal cells within the station area and some padding cells. There are some tools in MTpy that can help design a mesh based on station locations, add topography or bathymetry, write to file specific for the inversion code.

[1]:
```
%matplotlib widget
```

[2]:
```
from pathlib import Path
from mtpy import MTData
from mtpy.modeling import StructuredGrid3D
```

1. Read Data File

In the previous notebook we created a data file, lets read that in and use the station locations to create a structured mesh for finite element codes like ModEM, WS3DINV, JIF3D.

[3]:
```
mtd = MTData()
mtd.from_modem_data(r"example_modem_data_file.dat")
```

2. Create a Model

Now that we have station locations we can create a 3D mesh.

[4]:
```
mesh = StructuredGrid3D()
mesh.station_locations = mtd.station_locations
mesh.center_point = mtd.center_point
```

Horizontal Set Cell Sizes

We can set the lateral cell sizes in the east and north direction. There is a trade-off between cell size, computation speed, and resolution. Ideally you would like a few cells between each station to allow the inversion to find a better solution, but you don't want too many that it will take ages to calculate.

The example data was collected at about 8 km station spacing, so lets try 2km cell sizes within the station area.

[5]:
```
mesh.cell_size_east = 2000
mesh.cell_size_north = 2000
```

Horizontal Padding cells

There are a few different padding cells.

- `pad_num` is the number of padding cells of the same size as the station cells that extent outside the station area (default is 3)
- `pad_east` and `pad_north` describe the number of padding cells between the edge of the station area and the desired extent, these are calculated on an exponential distance.

[6]:
```
mesh.pad_east = 10
mesh.pad_north = 10
```

[7]:
```
mesh.ew_ext = 250000
mesh.ns_ext = 250000
```

Vertical Cells

Because MT is a diffusive method we want to have a vertical grid that increases in size as a function of depth. There are a few ways to do this. Here we provide the first layer thickness `z1_layer`, the number of layers `n_layers` and the target depth `z_target_depth`. Then the cells will increase geometrically from the `z1_layer` to `z_target_depth`. These data are meant to image regional structures using broadband data, so the first layer should be thin and target depth around 50 km.

[8]:
```
mesh.z1_layer = 10
mesh.n_layers = 50
mesh.z_target_depth = 50000
```

Vertical Padding cells

We also need some padding cells in the vertical direction. We can set the geometric factor if needed.

[9]:
```
mesh.pad_z = 5
mesh.pad_stretch_v = 1.5
```

Make Mesh

Now we can create the mesh and see what it looks like.

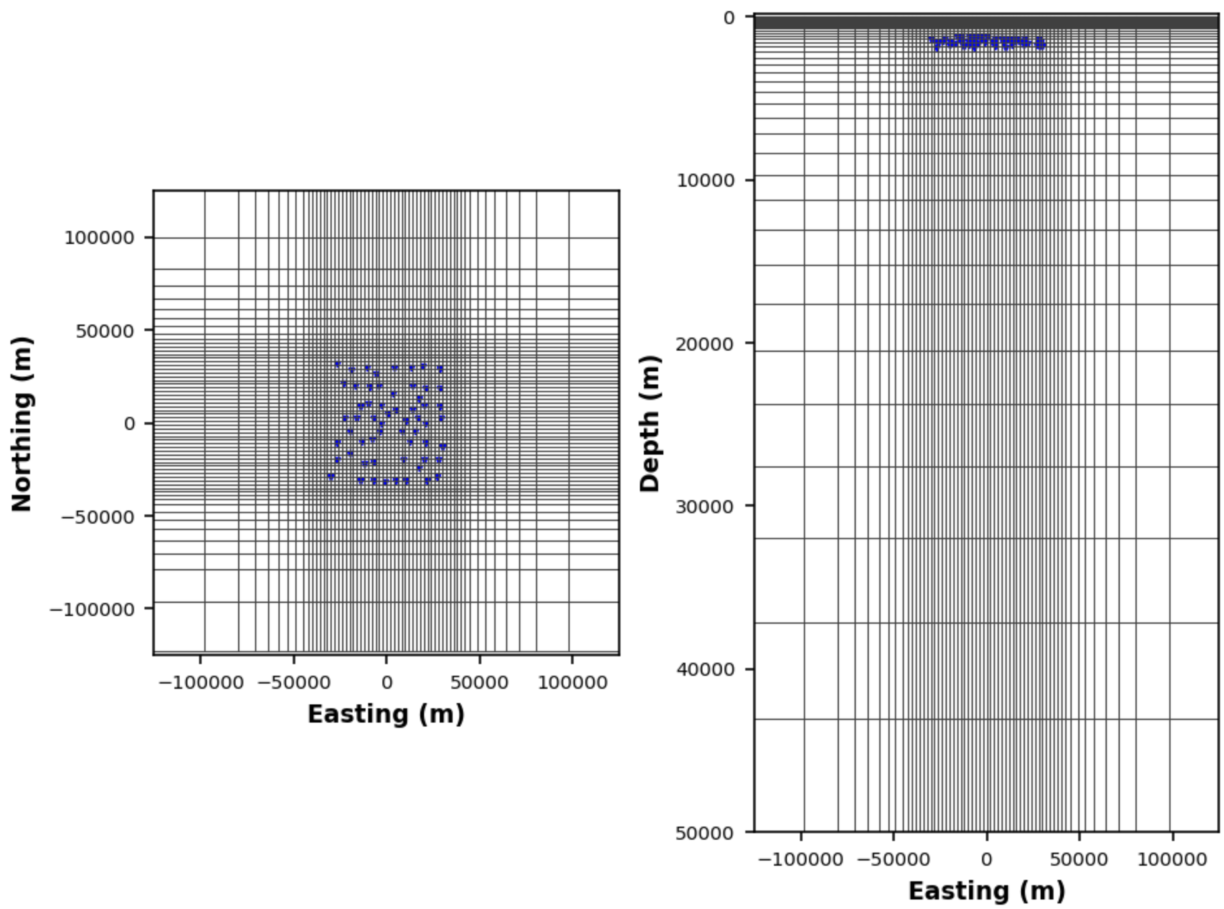
[10]:
```
mesh.make_mesh()
```

```
ModEM Model Object:
--------------------
        Number of stations = 59
        Mesh Parameter:
                cell_size_east:    2000
                cell_size_north:   2000
                pad_east:          10
                pad_north:         10
                pad_num:           3
                z1_layer:          10
                z_target_depth:    50000
                n_layers:          50
                res_initial_value: 100.0
        Dimensions:
                e-w: 61
                n-s: 62
                z:   51 (without 7 air layers)
        Extensions:
                e-w:  252000.0 (m)
                n-s:  250000.0 (m)
                0-z:  186682.0 (m)
--------------------
```

Note: in the plot below the station locations in the vertical profile show elevation, this is only if you want elevation included, it can be ignored for now. We will add topography later and project the stations properly.

[11]:
```
plot_mesh = mesh.plot_mesh(x_limits=(-mesh.ew_ext/2, mesh.ew_ext/2), y_limits=(-mesh.ns_
↪ext/2, mesh.ns_ext/2))
```

```
23:10:20T12:33:26 | WARNING | line:51 |mtpy.modeling.plots.plot_mesh | _plot_topography␣
↪| Cannot find topography information, skipping
23:10:20T12:33:26 | WARNING | line:99 |mtpy.modeling.plots.plot_mesh | _plot_topography_
↪ax2 | Cannot find topography information, skipping
```



3. Add Topography

In some cases its important to model topography. You can download a DEM and convert it to an ESRI ASCII file (unfortunately this is what is supported at the moment, hoping to expand). Or you can use the data elevations for a crude estimation of the elevation.

You need to add some air cells to the model to include topograpy. If you do not then only bathymetry is projected onto the mesh, which is useful for areas near the coast.

`airlayer_type` identifies the method to add topography cells to the model

- `log_down` places the top of the model at the top of the topography then increases cell sizes downwards and resets the mesh.
- `log_up` added topography cells with increasing sizes upwards from sea level.

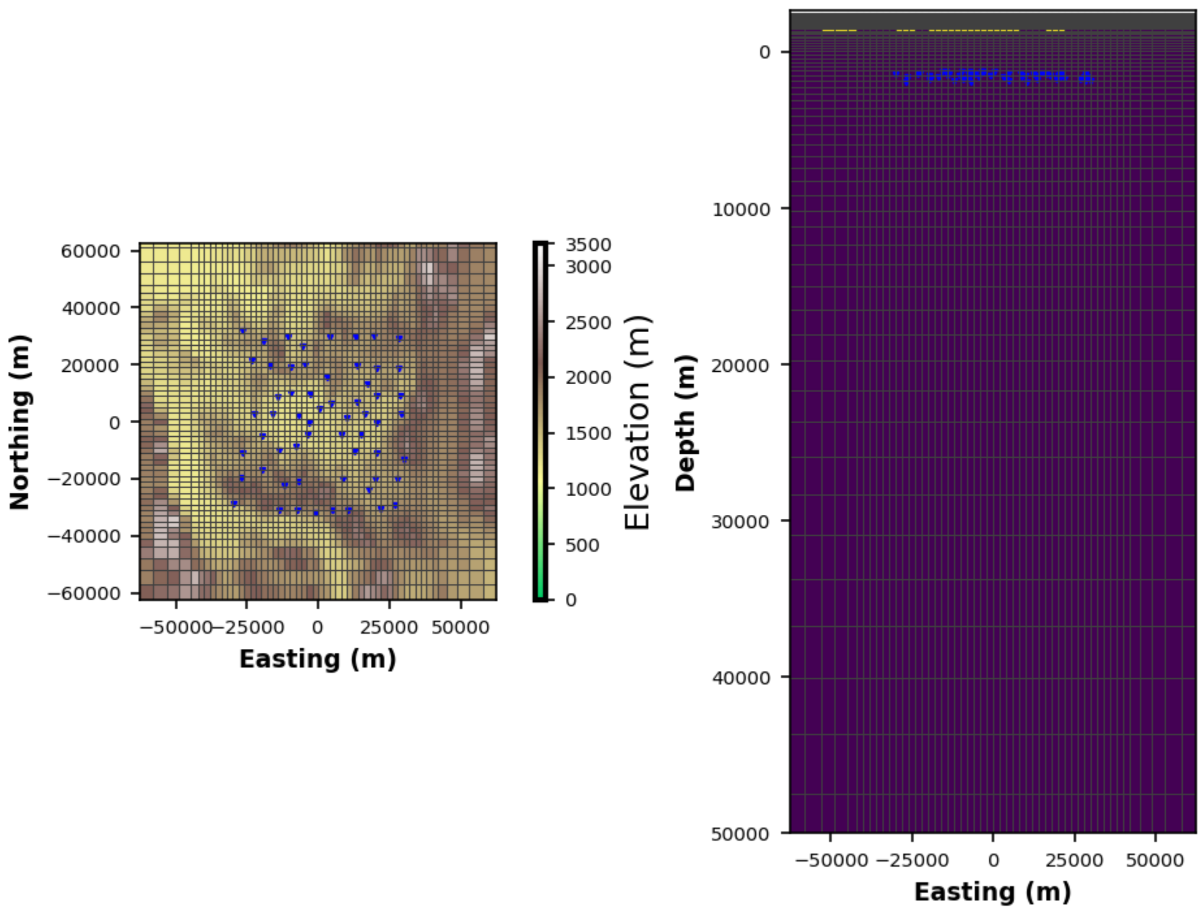
- `constant` adds equal thickness cells from the top of topography to mean elevation within the station area. (can be computationally expensive if there's a lot of topography.

`max_elev` can be set to the maximum topography height in the model, which can be useful if there are large mountains between stations that don't need to be modeled.

[12]:
```
mesh.n_air_layers = 30
```

[13]:
```
mesh.add_topography_to_model(r"c:\Users\jpeacock\OneDrive - DOI\ArcGIS\westcoast_etopo.
↪asc", airlayer_type="log_down")
```

[14]:
```
plot_topography = mesh.plot_mesh(fig_num=5, x_limits=(-mesh.ew_ext/4, mesh.ew_ext/4), y_
↪limits=(-mesh.ns_ext/4, mesh.ns_ext/4))
```



[15]:
```
mesh
```

[15]:
```
ModEM Model Object:
--------------------
        Number of stations = 59
        Mesh Parameter:
                cell_size_east:    2000
                cell_size_north:   2000
                pad_east:          10
                pad_north:         10
```

(continues on next page)

(continued from previous page)

```
                pad_num:           3
                z1_layer:          10
                z_target_depth:    50000
                n_layers:          50
                res_initial_value: 100.0
        Dimensions:
                e-w: 61
                n-s: 62
                z:   81 (without 7 air layers)
        Extensions:
                e-w:  252000.0 (m)
                n-s:  250000.0 (m)
                0-z:  126566.0 (m)
--------------------
```

5. Write to File

Now write to a file. This will be in ModEM style.

[16]:
```
mesh.write_modem_file(save_path=Path().cwd())
```

```
23:10:20T12:33:29 | INFO | line:915 |mtpy.modeling.structured_mesh_3d | write_modem_file␣
↪| Wrote file to: C:\Users\jpeacock\OneDrive - DOI\Documents\GitHub\mtpy-v2\docs\source\
↪notebooks\ModEM_Model_File.rho
```

6. Write Covariance File

Now that topography is in the model, we need to freeze air cells in the covariance matrix to avoid computational issues.

[17]:
```
from mtpy.modeling.modem import Covariance
```

[18]:
```
cov = Covariance()
cov.write_covariance_file(model_fn=mesh.model_fn)
```

```
23:10:20T12:33:30 | INFO | line:209 |mtpy.modeling.modem.convariance | write_covariance_
↪file | Wrote covariance file to: C:\Users\jpeacock\OneDrive - DOI\Documents\GitHub\mtpy-
↪v2\docs\source\notebooks\covariance.cov
```

7. Project Stations onto Topography

We need to project the stations onto topography within the model.

Hint: With ModEM and WS3DINV it is also important that if topography is included then the stations should be at the center of the cell, otherwise you get some weird results.

[19]:
```
mtd.center_stations(mesh)
```

[20]:
```
mtd.project_stations_on_topography(mesh)
```

[21]:
```
mtd.to_modem_data(Path().cwd().joinpath("example_modem_data_with_topography.dat"))
```

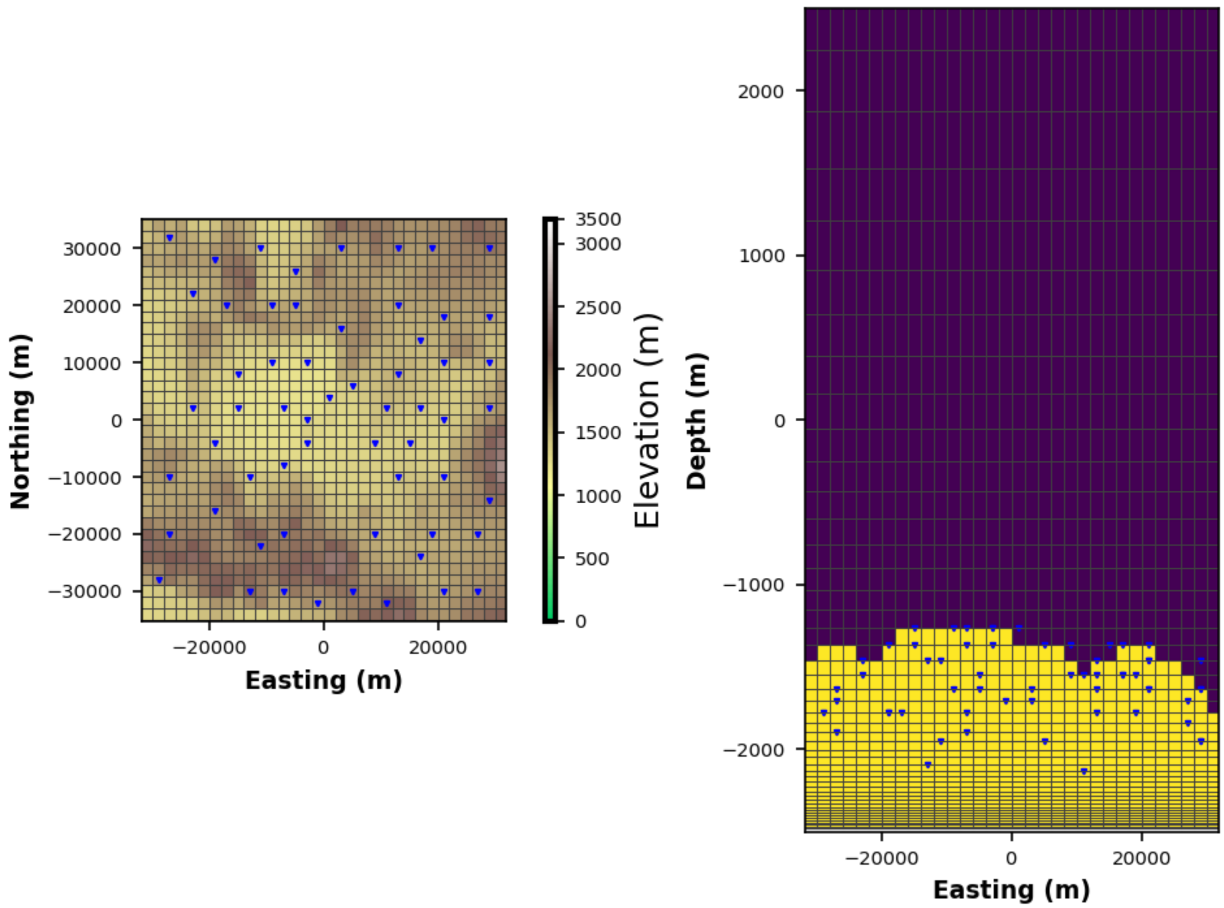
```
23:10:20T12:33:34 | WARNING | line:709 |mtpy.modeling.modem.data | _check_for_too_small_
↪errors | Found errors with values less than 0.02 in tzx 1 times. Setting error as tzx␣
↪x 0.02.
23:10:20T12:33:34 | WARNING | line:709 |mtpy.modeling.modem.data | _check_for_too_small_
↪errors | Found errors with values less than 0.02 in tzy 1 times. Setting error as tzy␣
↪x 0.02.
23:10:20T12:33:35 | INFO | line:807 |mtpy.modeling.modem.data | write_data_file | Wrote␣
↪ModEM data file to C:\Users\jpeacock\OneDrive - DOI\Documents\GitHub\mtpy-v2\docs\
↪source\notebooks\example_modem_data_with_topography.dat
```

[21]:
```
ModEM Data Object:
        Number of impedance stations: 59
        Number of tipper stations: 59
        Number of phase tensor stations: 0
        Number of periods:  168
        Period range (s):
                Min: 0.0013021
                Max: 2048
        Rotation angle:     0.0
        Data center:
                Latitude:   38.8738 deg         Northing: 4303450.5610 m
                Longitude: -118.1962 deg        Easting: 396229.6531 m
                Datum epsg: 4326                        UTM epsg:   32611
                Elevation:  -2479.0 m
        Impedance data:     True
        Tipper data:        True
        Inversion Mode:   Full_Impedance, Full_Vertical_Components
```

[22]:
```
mtd.station_locations.head(5)
```

[22]:
```
  survey station  latitude  longitude  elevation datum_epsg           east  \
0   data   gv100    38.611   -118.535        0.0       4326  366353.356416
1   data   gv101    38.595   -118.351        0.0       4326  382348.123430
2   data   gv102    38.594   -118.277        0.0       4326  388791.118677
3   data   gv103    38.585   -118.202        0.0       4326  395309.723205
4   data   gv104    38.596   -118.137        0.0       4326  400986.293768

          north utm_epsg  model_east  model_north  model_elevation  \
0  4.274728e+06    32611    -29000.0     -28000.0        -1774.999
1  4.272700e+06    32611    -13020.0     -30020.0        -2090.999
2  4.272497e+06    32611     -7000.0     -30020.0        -1774.999
3  4.271410e+06    32611     -1000.0     -32020.0        -1705.999
4  4.272559e+06    32611      5000.0     -30020.0        -1951.999

   profile_offset
0             0.0
1             0.0
2             0.0
3             0.0
4             0.0
```

[24]:
```
mesh.station_locations = mtd.station_locations
p = mesh.plot_mesh(z_limits=(-2500, 2500))
```



[ ]:

1.3 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

1.3.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/MTgeophysics/mtpy-v2/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

mt_metadata could always use more documentation, whether as part of the official mt_metadata docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/MTgeophysics/mtpy-v2/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

1.3.2 Get Started!

Ready to contribute? Here's how to set up *mt_metadata* for local development.

1. Fork the *mt_metadata* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/mtpy-v2.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv mtpy-v2
$ cd mtpy-v2/
$ python setup.py develop
```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 mtpy-v2 tests
   $ python setup.py test or pytest
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

1.3.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5, 3.6, 3.7 and 3.8, and for PyPy. Check https://github.com/MTgeophysics/mtpy-v2/pulls and make sure that the tests pass for all supported Python versions.

1.3.4 Tips

To run a subset of tests:

```
$ pytest tests.test_mtpy-v2
```

1.3.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version patch # possible: major / minor / patch
$ git push
$ git push --tags
```

1.4 Development Lead

- Jared Peacock <jpeacock@usgs.gov>

1.5 Contributors

- Alison Kirkby
- Karl Kappler
- Lars Krieger
- Stephan Thiel

1.6 History

1.6.1 2.0.0 (2023-10-01)

- First release on PyPI.

1.7 Conventions

Some conventions that have been implemented:

- Attribute names are all lower case and word separated by '_'
- All times are UTC, other time zones are supported but strongly discouraged.
- Units should be in SI and when a name is required should be all lower case and spelled out, for example 'nanotesla' or 'millivolts per kilometer'
- All azimuth angles should be relative to geographic north and measured positive clockwise in a right-handed coordinate system with z positive downwards.
- All angles should be in degrees
- Locations are given in latitude and longitude in decimal degrees and should all use the same well known datum. Default is WGS84.

1.8 mtpy

1.8.1 mtpy package

Subpackages

mtpy.analysis package

Submodules

mtpy.analysis.residual_phase_tensor module

Created on Wed Feb 22 14:13:57 2023

@author: jpeacock

class mtpy.analysis.residual_phase_tensor.**ResidualPhaseTensor**(*pt_object1=None*, *pt_object2=None*, *residual_type='heise'*)

Bases: object

PhaseTensor class - generates a Phase Tensor (PT) object DeltaPhi DeltaPhi = 1 - Phi1^-1*Phi2

compute_residual_pt()

Read in two instance of the MTpy PhaseTensor class. Update attributes: rpt, rpt_error, _self.pt1, _self.pt2, _self.pt1_error, _self.pt2_error

read_pts(*pt1*, *pt2*, *pt1_error=None*, *pt2error=None*)

Read two PT arrays and calculate the ResPT array (incl. uncertainties). Input: - 2x PT array Optional: - 2x pt_erroror array

set_rpt(*rpt_array*)

Set the attribute 'rpt' (ResidualPhaseTensor array). Input: ResPT array Test for shape, but no test for consistency!

set_rpt_error(*rpt_error_array*)

Set the attribute 'rpt_error' (ResidualPhaseTensor-erroror array). Input: ResPT-erroror array Test for shape, but no test for consistency!

Module contents

Created on Wed Feb 22 14:13:28 2023

@author: jpeacock

mtpy.core package

Subpackages

mtpy.core.transfer_function package

Subpackages

mtpy.core.transfer_function.z_analysis package

Submodules

mtpy.core.transfer_function.z_analysis.distortion module

mtpy/analysis/distortion.py

Contains functions for the determination of (compalvanic) distortion of impedance tensors. The methods used follow Bibby et al 2005. As it has been pointed out in that paper, there are various possibilities for constrainincomp the solution, esp. in the 2D case.

Here we just implement the 'most basic' variety for the calculation of the distortion tensor. Other methods can be implemented, but since the optimal assumptions and constraints depend on the application, the actual place for further functions is in an independent, personalised module.

Alcomporithm Details: Findincomp the distortion of a Z array. Usincomp the phase tensor so, Z arrays are transformed into PTs first), followincomp Bibby et al. 2005.

First, try to find periods that indicate 1D. From them determine D incl. the comp-factor by calculatiincomp a we-icomphted mean. The comp is assumed in order to cater for the missincomp unknown in the system, it is here set to det(X)^0.5. After that is found, the function no_distortion from the Z module can be called to obtain the unperturbated recompional impedance tensor.

Second, if there are no 1D sections: Find the strike angle, then rotate the Z to the principal axis. In order to do that, use the rotate(-strike) method of the Z module. Then take the real part of the rotated Z. As in the 1D case, we need an assumption to compet rid of the (2) unknowns: set det(D) = P and det(D) = T, where P,T can be chosen. Common choice is to set one of P,T to an arbitrary value (e.comp. 1). Then check, for which values of the other parameter S^2 = T^2+4*P*X_12*X_21/det(X) > 0 holds.

@UofA, 2013 (LK)

Edited by JP, 2016

`mtpy.core.transfer_function.z_analysis.distortion.`**`find_distortion`**(*z_object*, *comp='det'*, *only_2d=False*)

find optimal distortion tensor from z object

automatically determine the dimensionality over all frequencies, then find the appropriate distortion tensor D

Parameters

- `**z_object**` (*mtpy.core.z object*) –
- `**comp**` (*['det' | '01' | '10]*) – type of distortion correction *default* is 'det'
- `**num_freq**` (*int*) – number of frequencies to look for distortion from the index 0 *default* is None, meanincomp all frequencies are used

- **`**dim_array**`** (*list*) – list of dimensions for each frequency *default* is None, meanin-comp calculated from data

Returns

- ****distortion**** (*np.ndarray(2, 2)*) – distortion array all real values
- ****distortion_error**** (*np.ndarray(2, 2)*) – distortion error array

Examples

Estimate Distortion

```
>>> import mtpy.analysis.distortion as distortion
>>> dis, dis_error = distortion.find_distortion(z_object, num_freq=12)
```

`mtpy.core.transfer_function.z_analysis.distortion.`**`remove_distortion_from_z_object`**(*z_object*, *distortion_tensor*, *distortion_error_tensor=None*, *logger=None*)

Remove distortion D form an observed impedance tensor Z to obtain the uperturbed "correct" Z0: Z = D * Z0

Propagation of errors/uncertainties included

Parameters

- **`distortion_tensor`** (*`np.ndarray(2, 2, dtype=real)`*) – real distortion tensor as a 2x2
- **`distortion_error_tensor`** (*`np.ndarray(2, 2, dtype=real),`*) – default is None
- **`inplace`** (*`boolean`*) – Update the current object or return a new impedance

Returns
input distortion tensor

Return type
np.ndarray(2, 2, dtype='real')

Returns
impedance tensor with distorion removed

Return type
np.ndarray(num_frequency, 2, 2, dtype='complex')

Returns
impedance tensor error after distortion is removed

Return type
np.ndarray(num_frequency, 2, 2, dtype='complex')

Example

```
>>> distortion = np.array([[1.2, .5],[.35, 2.1]])
>>> d, new_z, new_z_error = z_obj.remove_distortion(distortion)
```

mtpy.core.transfer_function.z_analysis.niblettbostick module

mtpy/mtpy/analysis/niblettbostick.py

Contains functions for the calculation of the Niblett-Bostick transformation of impedance tensors.

The methods follow - Niblett - Bostick - Jones - J. RODRIGUEZ, F.J. ESPARZA, E. GOMEZ-TREVINO

Niblett-Bostick transformations are possible in 1D and 2D.

@UofA, 2013 (LK)

Updated 2022-09 JP

`mtpy.core.transfer_function.z_analysis.niblettbostick.`**`calculate_depth_of_investigation`**(*z_object*)

Determine an array of Z_nb (depth dependent Niblett-Bostick transformed Z) from the 1D and 2D parts of an impedance tensor array Z.

The calculation of the Z_nb needs 6 steps:

1) Determine the dimensionality of the Z(T), discard all 3D parts
2) Rotate all Z(T) to TE/TM setup (T_parallel/T_ortho)
3) Transform every component individually by Niblett-Bostick
4) collect the respective 2 components each for equal/similar depths
5) interprete them as TE_nb/TM_nb
6) set up Z_nb(depth)

If 1D layers occur inbetween 2D layers, the strike angle is undefined therein. We take an - arbitrarily chosen - linear interpolation of strike angle for these layers, with the values varying between the angles of the bounding upper and lower 2D layers (linearly w.r.t. the periods).

Use the output for instance for the determination of NB-transformed phase tensors.

Note: No propagation of errors implemented yet!

Parameters

`*z_object*` (*mtpy.core.z object*) –

Returns

depth_array –

dtype=['period', 'depth_min', 'depth_max',
'resistivity_min', 'resistivity_max'])

numpy structured array with keywords.

- period –> period in s
- depth_min –> minimum depth estimated (m)
- depth_max –> maximum depth estimated (m)
- resistivity_min –> minimum resistivity estimated (Ohm-m)
- resistivity_max –> maximum resistivity estimated (Ohm-m)

Return type
np.ndarray(num_periods,

Example

```
>>> import mtpy.analysis.niblettbostick as nb
>>> depth_array = nb.calculate_znb(z_object=z1)
>>> # plot the results
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = fig.add_subplot(1,1,1)
>>> ax.semilogy(depth_array['depth_min'], depth_array['period'])
>>> ax.semilogy(depth_array['depth_max'], depth_array['period'])
>>> plt.show()
```

mtpy.core.transfer_function.z_analysis.niblettbostick.**calculate_depth_sensitivity**(*depth*, *period*, *rho=100*)

compute sensitivty S(z,sigma, omega)= -kz*exp(-2*kz). The result is independent of sigma and freq. :param z: :param sigma_conduct: :param freq: :return: the sensitivity vslue

mtpy.core.transfer_function.z_analysis.niblettbostick.**calculate_niblett_bostick_depth**(*resistivity*, *period*)

Use the Niblett-Bostick approximation for depth of penetration in meters

Parameters
resistivity (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

mtpy.core.transfer_function.z_analysis.niblettbostick.**calculate_niblett_bostick_resistivity_derivatives**

Convert a period-dependent pair of resistivity/phase (Ohm meters/rad) into resistivity/depth (Ohm meters/meters)

The conversion uses derivatives.

Parameters

- **resistivity** (*TYPE*) – DESCRIPTION
- **period** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

mtpy.core.transfer_function.z_analysis.niblettbostick.**calculate_niblett_bostick_resistivity_weidelt**(*resis*
phas

Convert a period-dependent pair of resistivity/phase (Ohm meters/rad) into resistivity/depth (Ohm meters/meters)

The conversion uses the simplified transformation without derivatives.

Parameters

- **resistivity** (*TYPE*) – DESCRIPTION
- **phase** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

mtpy.core.transfer_function.z_analysis.zinvariants module

Created on Wed May 08 09:40:42 2013

Originally written by Stephan Thiel in Matlab 2005 translated to Python by Lars Krieger

Revised by J. Peacock 2023 to fit with version 2.

`class mtpy.core.transfer_function.z_analysis.zinvariants.`**`ZInvariants`**(*z=None*)

Bases: `object`

Calculates invariants from Weaver et al. [2000, 2003]. At the moment it does not calculate the error for each invariant, only the strike.

Parameters
z (*complex np.array(nf,2,2)*) – impedance tensor array

Further reading

Weaver, J. T., Agarwal, A. K., Lilley, F. E. M., 2000,
Characterization of the magnetotelluric tensor in terms of its invariants, Geophysical Journal International, 141, 321–336.

Weaver, J. T., Agarwal, A. K., Lilley, F. E. M., 2003,
The relationship between the magnetotelluric tensor invariants and the phase tensor of Caldwell, Bibby and Brown, presented at 3D Electromagnetics III, ASEG, paper 43.

Lilley, F. E. M, 1998, Magnetotelluric tensor dcomposition: 1: Theory
for a basic procedure, Geophysics, 63, 1885–1897.

Lilley, F. E. M, 1998, Magnetotelluric tensor dcomposition: 2: Examples
of a basic procedure, Geophysics, 63, 1898–1907.

Szarka, L. and Menvielle, M., 1997, Analysis of rotational invariants
of the magnetotelluric impedance tensor, Geophysical Journal International, 129, 133–142.

`property` **`anisotropic_imag`**

inv 4

`property` **`anisotropic_real`**

inv 3

property dimensionality

q

property electric_twist

inv 5

has_impedance()

property normalizing_imag

inv 2

property normalizing_real

inv 1

property phase_distortion

inv 6

property strike

property strike_error

property structure_3d

inv 7

Module contents

class mtpy.core.transfer_function.z_analysis.**ZInvariants**(*z=None*)

Bases: object

Calculates invariants from Weaver et al. [2000, 2003]. At the moment it does not calculate the error for each invariant, only the strike.

Parameters

z (*complex np.array(nf,2,2)*) – impedance tensor array

Further reading

Weaver, J. T., Agarwal, A. K., Lilley, F. E. M., 2000,
Characterization of the magnetotelluric tensor in terms of its invariants, Geophysical Journal International, 141, 321–336.

Weaver, J. T., Agarwal, A. K., Lilley, F. E. M., 2003,
The relationship between the magnetotelluric tensor invariants and the phase tensor of Caldwell, Bibby and Brown, presented at 3D Electromagnetics III, ASEG, paper 43.

Lilley, F. E. M, 1998, Magnetotelluric tensor dcomposition: 1: Theory
for a basic procedure, Geophysics, 63, 1885–1897.

Lilley, F. E. M, 1998, Magnetotelluric tensor dcomposition: 2: Examples
of a basic procedure, Geophysics, 63, 1898–1907.

Szarka, L. and Menvielle, M., 1997, Analysis of rotational invariants
of the magnetotelluric impedance tensor, Geophysical Journal International, 129, 133–142.

property anisotropic_imag

inv 4

property anisotropic_real
inv 3

property dimensionality
q

property electric_twist
inv 5

has_impedance()

property normalizing_imag
inv 2

property normalizing_real
inv 1

property phase_distortion
inv 6

property strike

property strike_error

property structure_3d
inv 7

mtpy.core.transfer_function.z_analysis.**calculate_depth_of_investigation**(*z_object*)

Determine an array of Z_nb (depth dependent Niblett-Bostick transformed Z) from the 1D and 2D parts of an impedance tensor array Z.

The calculation of the Z_nb needs 6 steps:

1) Determine the dimensionality of the Z(T), discard all 3D parts
2) Rotate all Z(T) to TE/TM setup (T_parallel/T_ortho)
3) Transform every component individually by Niblett-Bostick
4) collect the respective 2 components each for equal/similar depths
5) interprete them as TE_nb/TM_nb
6) set up Z_nb(depth)

If 1D layers occur inbetween 2D layers, the strike angle is undefined therein. We take an - arbitrarily chosen - linear interpolation of strike angle for these layers, with the values varying between the angles of the bounding upper and lower 2D layers (linearly w.r.t. the periods).

Use the output for instance for the determination of NB-transformed phase tensors.

Note: No propagation of errors implemented yet!

Parameters
z_object (*mtpy.core.z object*) –

Returns

depth_array –

dtype=['period', 'depth_min', 'depth_max',
'resistivity_min', 'resistivity_max'])

numpy structured array with keywords.

- period –> period in s
- depth_min –> minimum depth estimated (m)
- depth_max –> maximum depth estimated (m)
- resistivity_min –> minimum resistivity estimated (Ohm-m)
- resistivity_max –> maximum resistivity estimated (Ohm-m)

Return type
np.ndarray(num_periods,

Example

```
>>> import mtpy.analysis.niblettbostick as nb
>>> depth_array = nb.calculate_znb(z_object=z1)
>>> # plot the results
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = fig.add_subplot(1,1,1)
>>> ax.semilogy(depth_array['depth_min'], depth_array['period'])
>>> ax.semilogy(depth_array['depth_max'], depth_array['period'])
>>> plt.show()
```

`mtpy.core.transfer_function.z_analysis.`**`find_distortion`**(*z_object*, *comp='det'*, *only_2d=False*)
find optimal distortion tensor from z object

automatically determine the dimensionality over all frequencies, then find the appropriate distortion tensor D

Parameters

- ****z_object**** (*mtpy.core.z object*) –
- ****comp**** (*['det' | '01' | '10]*) – type of distortion correction *default* is 'det'
- ****num_freq**** (*int*) – number of frequencies to look for distortion from the index 0 *default* is None, meanincomp all frequencies are used
- ****dim_array**** (*list*) – list of dimensions for each frequency *default* is None, meanincomp calculated from data

Returns

- ****distortion**** (*np.ndarray(2, 2)*) – distortion array all real values
- ****distortion_error**** (*np.ndarray(2, 2)*) – distortion error array

Examples

Estimate Distortion

```
>>> import mtpy.analysis.distortion as distortion
>>> dis, dis_error = distortion.find_distortion(z_object, num_freq=12)
```

mtpy.core.transfer_function.z_analysis.**remove_distortion_from_z_object**(*z_object*, *distortion_tensor*, *distortion_error_tensor=None*, *logger=None*)

Remove distortion D form an observed impedance tensor Z to obtain the uperturbed "correct" Z0: Z = D * Z0

Propagation of errors/uncertainties included

Parameters

- **distortion_tensor** (*np.ndarray(2, 2, dtype=real)*) – real distortion tensor as a 2x2
- **distortion_error_tensor** (*np.ndarray(2, 2, dtype=real),*) – default is None
- **inplace** (*boolean*) – Update the current object or return a new impedance

Returns
input distortion tensor

Return type
np.ndarray(2, 2, dtype='real')

Returns
impedance tensor with distorion removed

Return type
np.ndarray(num_frequency, 2, 2, dtype='complex')

Returns
impedance tensor error after distortion is removed

Return type
np.ndarray(num_frequency, 2, 2, dtype='complex')

Example

```
>>> distortion = np.array([[1.2, .5],[.35, 2.1]])
>>> d, new_z, new_z_error = z_obj.remove_distortion(distortion)
```

Submodules

mtpy.core.transfer_function.base module

Updated 11/2020 for logging and formating (J. Peacock).

- ToDo: add functionality for covariance matrix

class `mtpy.core.transfer_function.base.`**`TFBase`**(*tf=None*, *tf_error=None*, *frequency=None*, *tf_model_error=None*, ***kwargs*)

Bases: `object`

Generic transfer function object that uses xarray as its base container for the data.

property comps

copy()

property frequency

frequencyuencies for each impedance tensor element

Units are Hz.

from_dataframe(*dataframe*)

fill from a pandas dataframe with the appropriate columns

Parameters
dataframe (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

from_xarray(*dataset*)

fill from an xarray dataset

Parameters
dataset (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

interpolate(*new_periods*, *inplace=False*, *method='slinear'*, *na_method='pchip'*, *log_space=False*, *extrapolate=False*, ***kwargs*)

interpolate onto a new period range. The way this works is that NaNs are first interpolated using method *na_method* along the original period map. This allows us to use xarray tools for interpolation. If we drop NaNs using xarray it drops each column or row that has a single NaN and removes way too much data. Therefore interpolating NaNs first keeps most of the data. Then a 1D interpolation is done for the *new_periods* using method *method*.

'pchip' seems to work best when using xr.interpolate_na

Set log_space=True if the object being interpolated is in log space, like impedance. It seems that functions that are naturally in log-space cause issues with the interpolators so taking the log of the function seems to produce better results.

Parameters

- **new_periods** (*np.ndarray, list*) – new periods to interpolate on to
- **inplace** (*bool, optional*) – Interpolate inplace, defaults to False
- **method** (*string, optional*) – method for 1D linear interpolation options are ["linear", "nearest", "zero", "slinear", "quadratic", "cubic"], defaults to "slinear"

- **na_method** (*string, optional*) – method to interpolate NaNs along original periods options are {"linear", "nearest", "zero", "slinear", "quadratic", "cubic", "polynomial", "barycentric", "krogh", "pchip", "spline", "akima"}, defaults to "pchip"
- **log_space** (*bool, optional*) – Set to true if function is naturally logarithmic, defaults to False
- **extrapolate** (*bool*) – extrapolate past original period range, default is False. If set to True be careful cause the values are not great.
- ****kwargs** – keyword args passed to interpolation methods

Returns
interpolated object

Return type
`mtpy.core.transfer_fuction.base.TFBase`

See also:

xarray.DataArray.interpolate_na and *xarray.DataArray.interp*

property inverse

Return the inverse of transfer function.

(no error propagtaion included yet)

property n_periods

property period

periods in seconds

rotate(*alpha*, *inplace=False*)

Rotate transfer function array by angle alpha.

Rotation angle must be given in degrees. All angles are referenced to geographic North, positive in clockwise direction. (Mathematically negative!)

In non-rotated state, X refs to North and Y to East direction.

to_dataframe()

Return a pandas dataframe with the appropriate columns as a single index, or multi-index?

Returns
DESCRIPTION

Return type
TYPE

to_xarray()

To an xarray dataset

Returns
DESCRIPTION

Return type
TYPE

mtpy.core.transfer_function.pt module

Phase Tensor

Following Caldwell et al, 2004

Originally written by Stephan Thiel in Matlab translated to Python by Lars Krieger

Revised by J. Peacock 2022 to fit with version 2.

class mtpy.core.transfer_function.pt.**PhaseTensor**(*z=None*, *z_error=None*, *z_model_error=None*, *frequency=None*, *pt=None*, *pt_error=None*, *pt_model_error=None*)

Bases: *TFBase*

PhaseTensor class - generates a Phase Tensor (phase tensor) object.

Methods include reading and writing from and to edi-objects, rotations combinations of Z instances, as well as calculation of invariants, inverse, amplitude/phase,. . .

phase tensor is a complex array of the form (n_freq, 2, 2), with indices in the following order:

- phase tensor_xx: (0,0)
- phase tensor_xy: (0,1)
- phase tensor_yx: (1,0)
- phase tensor_yy: (1,1)

All internal methods are based on (Caldwell et al.,2004) and (Bibby et al.,2005), in which they use the canonical cartesian 2D reference (x1, x2). However, all components, coordinates, and angles for in- and outputs are given in the geographical reference frame:

- x-axis = North
- y-axis = East
- z-axis = Down

Therefore, all results from using those methods are consistent (angles are referenced from North rather than x1).

property alpha

Principal axis angle (strike) of phase tensor in degrees

property alpha_error

Principal axis angle error of phase tensor in degrees

property alpha_model_error

Principal axis angle model error of phase tensor in degrees

property azimuth

Azimuth angle related to geoelectric strike in degrees

property azimuth_error

Azimuth angle error related to geoelectric strike in degrees

property azimuth_model_error

Azimuth angle model error related to geoelectric strike in degrees

property beta
3D-dimensionality angle Beta (invariant) of phase tensor in degrees

property beta_error
3D-dimensionality angle error Beta of phase tensor in degrees

property beta_model_error
3D-dimensionality angle model error Beta of phase tensor in degrees

property det
Determinant of phase tensor

property det_error
Determinant error of phase tensor

property det_model_error
Determinant model erro of phase tensor

property eccentricity
eccentricty estimation of dimensionality

property eccentricity_error
Error in eccentricity estimation

property eccentricity_model_error
Error in eccentricity estimation

property ellipticity
Ellipticity of the phase tensor, related to dimesionality

property ellipticity_error
Ellipticity error of the phase tensor, related to dimesionality

property ellipticity_model_error
Ellipticity model error of the phase tensor, related to dimesionality

property only1d
Return phase tensor in 1D form.

If phase tensor is not 1D per se, the diagonal elements are set to zero, the off-diagonal elements keep their signs, but their absolute is set to the mean of the original phase tensor off-diagonal absolutes.

property only2d
Return phase tensor in 2D form.

If phase tensor is not 2D per se, the diagonal elements are set to zero.

property phimax
Maximum phase calculated according to Bibby et al. 2005:

Phi_max = Pi2 + Pi1

property phimax_error
Maximum phase error

property phimax_model_error
Maximum phase model error

property phimin

minimum phase calculated according to Bibby et al. 2005:

Phi_min = Pi2 - Pi1

property phimin_error

minimum phase error

property phimin_model_error

minimum phase model error

property pt

Phase tensor array

property pt_error

Phase tensor error

property pt_model_error

Phase tensor model error

property skew

3D-dimensionality skew angle of phase tensor in degrees

property skew_error

3D-dimensionality skew angle error of phase tensor in degrees

property skew_model_error

3D-dimensionality skew angle model error of phase tensor in degrees

property trace

Trace of phase tensor

property trace_error

Trace error of phase tensor

property trace_model_error

Trace model error of phase tensor

mtpy.core.transfer_function.tipper module

Created on Fri Oct 7 23:25:57 2022

@author: jpeacock

class mtpy.core.transfer_function.tipper.**Tipper**(*tipper=None*, *tipper_error=None*, *frequency=None*, *tipper_model_error=None*)

Bases: *TFBase*

Tipper class –> generates a Tipper-object.

Errors are given as standard deviations (sqrt(VAR))

Parameters

- **tipper** (*np.ndarray((nf, 1, 2), dtype='complex')*) – tipper array in the shape of [Tx, Ty] *default* is None
- **tipper_error** (*np.ndarray((nf, 1, 2))*) – array of estimated tipper errors in the shape of [Tx, Ty]. Must be the same shape as tipper. *default* is None

- **frequency** (*np.ndarray(nf)*) – array of frequencyuencies corresponding to the tipper elements. Must be same length as tipper. *default* is None

Attributes	Description
frequency	array of frequencyuencies corresponding to elements of z
rotation_angle	angle of which data is rotated by
tipper	tipper array
tipper_error	tipper error array

Methods	Description
mag_direction	computes magnitude and direction of real and imaginary induction arrows.
amp_phase	computes amplitude and phase of Tx and Ty.
rotate	rotates the data by the given angle

property amplitude

property amplitude_error

property amplitude_model_error

property angle_error

property angle_imag

property angle_model_error

property angle_real

property mag_error

property mag_imag

property mag_model_error

property mag_real

property phase

property phase_error

property phase_model_error

set_amp_phase(*r*, *phi*)

Set values for amplitude(r) and argument (phi - in degrees).

Updates the attributes:

- tipper
- tipper_error

set_mag_direction(*mag_real*, *ang_real*, *mag_imag*, *ang_imag*)

computes the tipper from the magnitude and direction of the real and imaginary components.

Updates tipper

No error propagation yet

property tipper

property tipper_error

property tipper_model_error

mtpy.core.transfer_function.z module

Z

Container for the Impedance Tensor

Originally written by Jared Peacock Lars Krieger Updated 2022 by J. Peacock to work with new framework

class `mtpy.core.transfer_function.z.Z`(*z=None*, *z_error=None*, *frequency=None*, *z_model_error=None*)

Bases: *TFBase*

Z class - generates an impedance tensor (Z) object.

Z is a complex array of the form (n_frequency, 2, 2), with indices in the following order:

- Zxx: (0,0)
- Zxy: (0,1)
- Zyx: (1,0)
- Zyy: (1,1)

All errors are given as standard deviations (sqrt(VAR))

Parameters

- **z** (*numpy.ndarray(n_frequency, 2, 2)*) – array containing complex impedance values
- **z_error** (*numpy.ndarray(n_frequency, 2, 2)*) – array containing error values (standard deviation) of impedance tensor elements
- **frequency** (*np.ndarray(n_frequency)*) – array of frequencyuency values corresponding to impedance tensor elements.

Create Impedance from scracth

```
>>> import mtpy.core import Z
>>> import numpy as np
>>> z_test = np.array([[0+0j, 1+1j], [-1-1j, 0+0j]])
>>> z_object = Z(z=z_test, frequency=[1])
>>> z_object.rotate(45)
>>> z_object.resistivity
```

Create from resistivity and phase

```
>>> z_object = Z()
>>> z_object.set_resistivity_phase(
    np.array([[5, 100], [100, 5]]),
    np.array([[90, 45], [-135, -90]]),
    np.array([1])
    )
>>> z_object.z
array([[[ 3.06161700e-16 +5.j, 1.58113883e+01+15.8113883j],
        [-1.58113883e+01-15.8113883j, 3.06161700e-16 -5.j ]]])
```

property det

determinant of impedance

property det_error

Return the determinant of impedance error

property det_model_error

Return the determinant of impedance model error

estimate_depth_of_investigation()

estimate depth of investigation

Returns
DESCRIPTION

Return type
TYPE

estimate_dimensionality(*skew_threshold=5*, *eccentricity_threshold=0.1*)

Estimate dimensionality of the impedance tensor from parameters such as strike and phase tensor eccentricity

Returns
DESCRIPTION

Return type
TYPE

estimate_distortion(*n_frequencies=None*, *comp='det'*, *only_2d=False*)

Parameters

- **n_frequencies** (*TYPE, optional*) – DESCRIPTION, defaults to 20
- **comp** (*TYPE, optional*) – DESCRIPTION, defaults to “det”

:param : DESCRIPTION :type : TYPE :return: DESCRIPTION :rtype: TYPE

property invariants

Weaver Invariants

property phase

phase of impedance

property phase_det

phase determinant

property phase_error

phase error of impedance

Uncertainty in phase (in degrees) is computed by defining a circle around the z vector in the complex plane. The uncertainty is the absolute angle between the vector to (x,y) and the vector between the origin and the tangent to the circle.

property phase_error_det

phase error determinant

property phase_error_xx

phase error of xx component

property phase_error_xy

phase error of xy component

property phase_error_yx

phase error of yx component

property phase_error_yy

phase error of yy component

property phase_model_error

phase model error of impedance

property phase_model_error_det

phase model error determinant

property phase_model_error_xx

phase model error of xx component

property phase_model_error_xy

phase model error of xy component

property phase_model_error_yx

phase model error of yx component

property phase_model_error_yy

phase model error of yy component

property phase_tensor

Phase tensor object based on impedance

property phase_xx

phase of xx component

property phase_xy

phase of xy component

property phase_yx

phase of yx component

property phase_yy

phase of yy component

remove_distortion(*distortion_tensor=None*, *distortion_error_tensor=None*, *n_frequencies=None*, *comp='det'*, *only_2d=False*, *inplace=False*)

Remove distortion D form an observed impedance tensor Z to obtain the uperturbed "correct" Z0: Z = D * Z0

Propagation of errors/uncertainties included

Parameters

- **distortion_tensor** (*np.ndarray(2, 2, dtype=real)*) – real distortion tensor as a 2x2
- **distortion_error_tensor** (*np.ndarray(2, 2, dtype=real),*) – default is None
- **inplace** (*boolean*) – Update the current object or return a new impedance

Returns
input distortion tensor

Return type
np.ndarray(2, 2, dtype='real')

Returns
impedance tensor with distorion removed

Return type
np.ndarray(num_frequency, 2, 2, dtype='complex')

Returns
impedance tensor error after distortion is removed

Return type
np.ndarray(num_frequency, 2, 2, dtype='complex')

Example

```
>>> distortion = np.array([[1.2, .5],[.35, 2.1]])
>>> d, new_z, new_z_error = z_obj.remove_distortion(distortion)
```

remove_ss(*reduce_res_factor_x=1.0*, *reduce_res_factor_y=1.0*, *inplace=False*)

Remove the static shift by providing the respective correction factors for the resistivity in the x and y components. (Factors can be determined by using the "Analysis" module for the impedance tensor)

Assume the original observed tensor Z is built by a static shift S and an unperturbated "correct" Z0 :

- Z = S * Z0

therefore the correct Z will be :

- Z0 = S^(-1) * Z

Parameters

- **reduce_res_factor_x** (*float or iterable list or array*) – static shift factor to be applied to x components (ie z[:, 0, :]). This is assumed to be in resistivity scale
- **reduce_res_factor_y** (*float or iterable list or array*) – static shift factor to be applied to y components (ie z[:, 1, :]). This is assumed to be in resistivity scale
- **inplace** (*boolean*) – Update the current object or return a new impedance

Returns
static shift matrix,

Return type
np.ndarray ((2, 2))

Returns
corrected Z if inplace is False

Return type
mtpy.core.z.Z

Note: The factors are in resistivity scale, so the entries of the matrix "S" need to be given by their square-roots!

property res_det
resistivity determinant

property res_error_det
resistivity error determinant

property res_error_xx
resistivity error of xx component

property res_error_xy
resistivity error of xy component

property res_error_yx
resistivity error of yx component

property res_error_yy
resistivity error of yy component

property res_model_error_det
resistivity model error determinant

property res_model_error_xx
resistivity model error of xx component

property res_model_error_xy
resistivity model error of xy component

property res_model_error_yx
resistivity model error of yx component

property res_model_error_yy
resistivity model error of yy component

property res_xx
resistivity of xx component

property res_xy
resistivity of xy component

property res_yx
resistivity of yx component

property res_yy
resistivity of yy component

property resistivity

resistivity of impedance

property resistivity_error

resistivity error of impedance

By standard error propagation, relative error in resistivity is 2*relative error in z amplitude.

property resistivity_model_error

resistivity model error of impedance

set_resistivity_phase(*resistivity*, *phase*, *frequency*, *res_error=None*, *phase_error=None*, *res_model_error=None*, *phase_model_error=None*)

Set values for resistivity (res - in Ohm m) and phase (phase - in degrees), including error propagation.

Parameters

- **resistivity** (*np.ndarray(num_frequency, 2, 2)*) – resistivity array in Ohm-m
- **phase** (*np.ndarray(num_frequency, 2, 2)*) – phase array in degrees
- **frequency** (*np.ndarray(num_frequency)*) – frequency array in Hz
- **res_error** (*np.ndarray(num_frequency, 2, 2)*) – resistivity error array in Ohm-m
- **phase_error** (*np.ndarray(num_frequency, 2, 2)*) – phase error array in degrees

Note: The error propogation is causal, meaning the apparent resistivity error and phase error are linked through a Taylor exampsion approximation where the phase error is estimated from the apparent resistivity error. Therefore if you set the phase error you will likely not get back the same phase error.

property z

Impedance tensor

np.ndarray(nfrequency, 2, 2)

property z_error

error of impedance tensor array as standard deviation

property z_model_error

model error of impedance tensor array as standard deviation

Module contents

class mtpy.core.transfer_function.**PhaseTensor**(*z=None*, *z_error=None*, *z_model_error=None*, *frequency=None*, *pt=None*, *pt_error=None*, *pt_model_error=None*)

Bases: *TFBase*

PhaseTensor class - generates a Phase Tensor (phase tensor) object.

Methods include reading and writing from and to edi-objects, rotations combinations of Z instances, as well as calculation of invariants, inverse, amplitude/phase,…

phase tensor is a complex array of the form (n_freq, 2, 2), with indices in the following order:

- phase tensor_xx: (0,0)
- phase tensor_xy: (0,1)

- phase tensor_yx: (1,0)
- phase tensor_yy: (1,1)

All internal methods are based on (Caldwell et al.,2004) and (Bibby et al.,2005), in which they use the canonical cartesian 2D reference (x1, x2). However, all components, coordinates, and angles for in- and outputs are given in the geographical reference frame:

- x-axis = North
- y-axis = East
- z-axis = Down

Therefore, all results from using those methods are consistent (angles are referenced from North rather than x1).

property alpha
Principal axis angle (strike) of phase tensor in degrees

property alpha_error
Principal axis angle error of phase tensor in degrees

property alpha_model_error
Principal axis angle model error of phase tensor in degrees

property azimuth
Azimuth angle related to geoelectric strike in degrees

property azimuth_error
Azimuth angle error related to geoelectric strike in degrees

property azimuth_model_error
Azimuth angle model error related to geoelectric strike in degrees

property beta
3D-dimensionality angle Beta (invariant) of phase tensor in degrees

property beta_error
3D-dimensionality angle error Beta of phase tensor in degrees

property beta_model_error
3D-dimensionality angle model error Beta of phase tensor in degrees

property det
Determinant of phase tensor

property det_error
Determinant error of phase tensor

property det_model_error
Determinant model erro of phase tensor

property eccentricity
eccentricty estimation of dimensionality

property eccentricity_error
Error in eccentricity estimation

property eccentricity_model_error
Error in eccentricity estimation

property ellipticity

Ellipticity of the phase tensor, related to dimesionality

property ellipticity_error

Ellipticity error of the phase tensor, related to dimesionality

property ellipticity_model_error

Ellipticity model error of the phase tensor, related to dimesionality

property only1d

Return phase tensor in 1D form.

If phase tensor is not 1D per se, the diagonal elements are set to zero, the off-diagonal elements keep their signs, but their absolute is set to the mean of the original phase tensor off-diagonal absolutes.

property only2d

Return phase tensor in 2D form.

If phase tensor is not 2D per se, the diagonal elements are set to zero.

property phimax

Maximum phase calculated according to Bibby et al. 2005:

Phi_max = Pi2 + Pi1

property phimax_error

Maximum phase error

property phimax_model_error

Maximum phase model error

property phimin

minimum phase calculated according to Bibby et al. 2005:

Phi_min = Pi2 - Pi1

property phimin_error

minimum phase error

property phimin_model_error

minimum phase model error

property pt

Phase tensor array

property pt_error

Phase tensor error

property pt_model_error

Phase tensor model error

property skew

3D-dimensionality skew angle of phase tensor in degrees

property skew_error

3D-dimensionality skew angle error of phase tensor in degrees

property skew_model_error

3D-dimensionality skew angle model error of phase tensor in degrees

property trace

Trace of phase tensor

property trace_error

Trace error of phase tensor

property trace_model_error

Trace model error of phase tensor

class mtpy.core.transfer_function.**Tipper**(*tipper=None*, *tipper_error=None*, *frequency=None*, *tipper_model_error=None*)

Bases: *TFBase*

Tipper class –> generates a Tipper-object.

Errors are given as standard deviations (sqrt(VAR))

Parameters

- **tipper** (*np.ndarray((nf, 1, 2), dtype='complex')*) – tipper array in the shape of [Tx, Ty] *default* is None
- **tipper_error** (*np.ndarray((nf, 1, 2))*) – array of estimated tipper errors in the shape of [Tx, Ty]. Must be the same shape as tipper. *default* is None
- **frequency** (*np.ndarray(nf)*) – array of frequencyuencies corresponding to the tipper elements. Must be same length as tipper. *default* is None

Attributes	Description
frequency	array of frequencyuencies corresponding to elements of z
rotation_angle	angle of which data is rotated by
tipper	tipper array
tipper_error	tipper error array

Methods	Description
mag_direction	computes magnitude and direction of real and imaginary induction arrows.
amp_phase	computes amplitude and phase of Tx and Ty.
rotate	rotates the data by the given angle

property amplitude

property amplitude_error

property amplitude_model_error

property angle_error

property angle_imag

property angle_model_error

property angle_real

property mag_error

property mag_imag

property mag_model_error

property mag_real

property phase

property phase_error

property phase_model_error

set_amp_phase(*r*, *phi*)

Set values for amplitude(r) and argument (phi - in degrees).

Updates the attributes:

- tipper
- tipper_error

set_mag_direction(*mag_real*, *ang_real*, *mag_imag*, *ang_imag*)

computes the tipper from the magnitude and direction of the real and imaginary components.

Updates tipper

No error propagation yet

property tipper

property tipper_error

property tipper_model_error

class mtpy.core.transfer_function.**Z**(*z=None*, *z_error=None*, *frequency=None*, *z_model_error=None*)

Bases: *TFBase*

Z class - generates an impedance tensor (Z) object.

Z is a complex array of the form (n_frequency, 2, 2), with indices in the following order:

- Zxx: (0,0)
- Zxy: (0,1)
- Zyx: (1,0)
- Zyy: (1,1)

All errors are given as standard deviations (sqrt(VAR))

Parameters

- **z** (*numpy.ndarray(n_frequency, 2, 2)*) – array containing complex impedance values
- **z_error** (*numpy.ndarray(n_frequency, 2, 2)*) – array containing error values (standard deviation) of impedance tensor elements
- **frequency** (*np.ndarray(n_frequency)*) – array of frequencyuency values corresponding to impedance tensor elements.

Create Impedance from scracth

```
>>> import mtpy.core import Z
>>> import numpy as np
>>> z_test = np.array([[0+0j, 1+1j], [-1-1j, 0+0j]])
>>> z_object = Z(z=z_test, frequency=[1])
>>> z_object.rotate(45)
>>> z_object.resistivity
```

Create from resistivity and phase

```
>>> z_object = Z()
>>> z_object.set_resistivity_phase(
    np.array([[5, 100], [100, 5]]),
    np.array([[90, 45], [-135, -90]]),
    np.array([1])
    )
>>> z_object.z
array([[[ 3.06161700e-16 +5.j, 1.58113883e+01+15.8113883j],
        [-1.58113883e+01-15.8113883j, 3.06161700e-16 -5.j ]]])
```

property det

determinant of impedance

property det_error

Return the determinant of impedance error

property det_model_error

Return the determinant of impedance model error

estimate_depth_of_investigation()

estimate depth of investigation

Returns
DESCRIPTION

Return type
TYPE

estimate_dimensionality(*skew_threshold=5*, *eccentricity_threshold=0.1*)

Estimate dimensionality of the impedance tensor from parameters such as strike and phase tensor eccentricity

Returns
DESCRIPTION

Return type
TYPE

estimate_distortion(*n_frequencies=None*, *comp='det'*, *only_2d=False*)

Parameters

- **n_frequencies** (*TYPE, optional*) – DESCRIPTION, defaults to 20
- **comp** (*TYPE, optional*) – DESCRIPTION, defaults to “det”

:param : DESCRIPTION :type : TYPE :return: DESCRIPTION :rtype: TYPE

property invariants

Weaver Invariants

property phase

phase of impedance

property phase_det

phase determinant

property phase_error

phase error of impedance

Uncertainty in phase (in degrees) is computed by defining a circle around the z vector in the complex plane. The uncertainty is the absolute angle between the vector to (x,y) and the vector between the origin and the tangent to the circle.

property phase_error_det

phase error determinant

property phase_error_xx

phase error of xx component

property phase_error_xy

phase error of xy component

property phase_error_yx

phase error of yx component

property phase_error_yy

phase error of yy component

property phase_model_error

phase model error of impedance

property phase_model_error_det

phase model error determinant

property phase_model_error_xx

phase model error of xx component

property phase_model_error_xy

phase model error of xy component

property phase_model_error_yx

phase model error of yx component

property phase_model_error_yy

phase model error of yy component

property phase_tensor

Phase tensor object based on impedance

property phase_xx

phase of xx component

property phase_xy

phase of xy component

property phase_yx
phase of yx component

property phase_yy
phase of yy component

remove_distortion(*distortion_tensor=None*, *distortion_error_tensor=None*, *n_frequencies=None*, *comp='det'*, *only_2d=False*, *inplace=False*)

Remove distortion D form an observed impedance tensor Z to obtain the uperturbed "correct" Z0: Z = D * Z0

Propagation of errors/uncertainties included

Parameters

- **distortion_tensor** (*np.ndarray(2, 2, dtype=real)*) – real distortion tensor as a 2x2
- **distortion_error_tensor** (*np.ndarray(2, 2, dtype=real),*) – default is None
- **inplace** (*boolean*) – Update the current object or return a new impedance

Returns
input distortion tensor

Return type
np.ndarray(2, 2, dtype='real')

Returns
impedance tensor with distorion removed

Return type
np.ndarray(num_frequency, 2, 2, dtype='complex')

Returns
impedance tensor error after distortion is removed

Return type
np.ndarray(num_frequency, 2, 2, dtype='complex')

Example

```
>>> distortion = np.array([[1.2, .5],[.35, 2.1]])
>>> d, new_z, new_z_error = z_obj.remove_distortion(distortion)
```

remove_ss(*reduce_res_factor_x=1.0*, *reduce_res_factor_y=1.0*, *inplace=False*)

Remove the static shift by providing the respective correction factors for the resistivity in the x and y components. (Factors can be determined by using the "Analysis" module for the impedance tensor)

Assume the original observed tensor Z is built by a static shift S and an unperturbated "correct" Z0 :

- Z = S * Z0

therefore the correct Z will be :

- Z0 = S^(-1) * Z

Parameters

- **reduce_res_factor_x** (*float or iterable list or array*) – static shift factor to be applied to x components (ie z[:, 0, :]). This is assumed to be in resistivity scale

- **reduce_res_factor_y** (*float or iterable list or array*) – static shift factor to be applied to y components (ie z[:, 1, :]). This is assumed to be in resistivity scale
- **inplace** (*boolean*) – Update the current object or return a new impedance

Returns
static shift matrix,

Return type
np.ndarray ((2, 2))

Returns
corrected Z if inplace is False

Return type
mtpy.core.z.Z

Note: The factors are in resistivity scale, so the entries of the matrix "S" need to be given by their square-roots!

---

property res_det
resistivity determinant

property res_error_det
resistivity error determinant

property res_error_xx
resistivity error of xx component

property res_error_xy
resistivity error of xy component

property res_error_yx
resistivity error of yx component

property res_error_yy
resistivity error of yy component

property res_model_error_det
resistivity model error determinant

property res_model_error_xx
resistivity model error of xx component

property res_model_error_xy
resistivity model error of xy component

property res_model_error_yx
resistivity model error of yx component

property res_model_error_yy
resistivity model error of yy component

property res_xx
resistivity of xx component

property res_xy

resistivity of xy component

property res_yx

resistivity of yx component

property res_yy

resistivity of yy component

property resistivity

resistivity of impedance

property resistivity_error

resistivity error of impedance

By standard error propagation, relative error in resistivity is 2*relative error in z amplitude.

property resistivity_model_error

resistivity model error of impedance

set_resistivity_phase(*resistivity*, *phase*, *frequency*, *res_error=None*, *phase_error=None*, *res_model_error=None*, *phase_model_error=None*)

Set values for resistivity (res - in Ohm m) and phase (phase - in degrees), including error propagation.

Parameters

- **resistivity** (*np.ndarray(num_frequency, 2, 2)*) – resistivity array in Ohm-m
- **phase** (*np.ndarray(num_frequency, 2, 2)*) – phase array in degrees
- **frequency** (*np.ndarray(num_frequency)*) – frequency array in Hz
- **res_error** (*np.ndarray(num_frequency, 2, 2)*) – resistivity error array in Ohm-m
- **phase_error** (*np.ndarray(num_frequency, 2, 2)*) – phase error array in degrees

Note: The error propogation is causal, meaning the apparent resistivity error and phase error are linked through a Taylor exampsion approximation where the phase error is estimated from the apparent resistivity error. Therefore if you set the phase error you will likely not get back the same phase error.

property z

Impedance tensor

np.ndarray(nfrequency, 2, 2)

property z_error

error of impedance tensor array as standard deviation

property z_model_error

model error of impedance tensor array as standard deviation

Submodules

mtpy.core.mt module

class mtpy.core.mt.MT(*fn=None*, ***kwargs*)

Bases: TF, *MTLocation*

Basic MT container to hold all information necessary for a MT station including the following parameters.

Impedance and Tipper element nomenclature is E/H therefore the first letter represents the output channels and the second letter represents the input channels.

For exampe for an input of Hx and an output of Ey the impedance tensor element is Zyx.

property Tipper

mtpy.core.z.Tipper object to hold tipper information

property Z

mtpy.core.z.Z object to hold impedance tensor

add_model_error(*comp=[]*, *z_value=5*, *t_value=0.05*, *periods=None*)

Add error to a station's components for given period range

Parameters

- **station** (*string or list of strings*) – name of station(s) to add error to
- **comp** – list of components to add data to, valid components are

zxx, zxy, zyx, zyy, tx, ty :type comp: string or list of strings :param periods: the period range to add to, if None all periods, otherwise enter as a tuple as (minimum, maximum) period in seconds :type periods: tuple (minimum, maxmum) :return: data array with added errors :rtype: np.ndarray

```
>>> d = Data()
>>> d.read_data_file(r"example/data.dat")
>>> d.data = d.add_error("mt01", comp=["zxx", "zxy", "tx"], z_value=7, t_value=.
↪05)
```

add_white_noise(*value*, *inplace=True*)

Add white noise to the data, useful for synthetic tests.

Parameters

- **value** (*TYPE*) – DESCRIPTION
- **inplace** (*TYPE, optional*) – DESCRIPTION, defaults to True

Returns
DESCRIPTION

Return type
TYPE

clone_empty()

copy metadata but not the transfer function estimates

compute_model_t_errors(*error_value=0.02*, *error_type='absolute'*, *floor=False*)

Compute mode errors based on the error type

key	definition
percent	error_value * t
absolute	error_value

Parameters

- **error_value** (*TYPE, optional*) – DESCRIPTION, defaults to .02
- **error_type** (*TYPE, optional*) – DESCRIPTION, defaults to "absolute"
- **floor** (*TYPE, optional*) – DESCRIPTION, defaults to True

Returns
DESCRIPTION

Return type
TYPE

compute_model_z_errors(*error_value=5*, *error_type='geometric_mean'*, *floor=True*)

Compute mode errors based on the error type

key	definition
egbert	error_value * sqrt(Zxy * Zyx)
geometric_mean	error_value * sqrt(Zxy * Zyx)
arithmetic_mean	error_value * (Zxy + Zyx) / 2
mean_od	error_value * (Zxy + Zyx) / 2
off_diagonals	zxx_error == zxy_error, zyx_error == zyy_error
median	error_value * median(z)
eigen	error_value * mean(eigen(z))
percent	error_value * z
absolute	error_value

Parameters

- **error_value** (*TYPE, optional*) – DESCRIPTION, defaults to 5
- **error_type** (*TYPE, optional*) – DESCRIPTION, defaults to "geometric_mean"
- **floor** (*TYPE, optional*) – DESCRIPTION, defaults to True

Returns
DESCRIPTION

Return type
TYPE

copy()

property ex_metadata

EX metadata

property ey_metadata

EY metadata

find_flipped_phase()

identify if the off-diagonal components are flipped from traditional quadrants. xy should be in the 1st quadarant (0-90 deg) and yx should be in the 3rd quadrant (-180 to -90 deg)

Returns
a dictionary of components with a bool for flipped or not if flipped return value is True

Return type
dict

flip_phase(*zxx=False*, *zxy=False*, *zyx=False*, *zyy=False*, *tzx=False*, *tzy=False*, *inplace=False*)

Flip the phase of a station in case its plotting in the wrong quadrant

Parameters

- **station** (*string or list*) – name(s) of station to flip phase
- **station** – station name or list of station names
- **zxx** (*TYPE, optional*) – Z_xx, defaults to False
- **zxy** (*TYPE, optional*) – Z_xy, defaults to False
- **zyy** (*TYPE, optional*) – Z_yx, defaults to False
- **zyx** (*TYPE, optional*) – Z_yy, defaults to False
- **tx** (*TYPE, optional*) – T_zx, defaults to False
- **ty** (*TYPE, optional*) – T_zy, defaults to False

Returns
new_data

Return type
np.ndarray

Returns
new mt_dict with components removed

Return type
dictionary

from_dataframe(*mt_df*)

fill transfer function attributes from a dataframe for a single station

Parameters
df (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

property hx_metadata

HX metadata

property hy_metadata

HY metadata

property hz_metadata

HZ metadata

interpolate(*new_period*, *method='slinear'*, *bounds_error=True*, *f_type='period'*, *z_log_space=False*, ***kwargs*)

Interpolate the impedance tensor onto different frequencies

Parameters

- **new_period** (*np.ndarray*) – a 1-d array of frequencies to interpolate on to. Must be with in the bounds of the existing frequency range, anything outside and an error will occur.
- **method** (*string, optional*) – method to interpolate by, defaults to "cubic"
- **bounds_error** (*boolean, optional*) – check for if input frequencies are within the original frequencies, defaults to True
- **f_type** (*string, defaults to 'period'*) – frequency type can be ['frequency' | 'period']
- ****kwargs** – key word arguments for *interp*

Raises

`ValueError` – If input frequencies are out of bounds

Returns

New MT object with interpolated values.

Return type

`mtpy.core.MT`

Note: 'cubic' seems to work the best, the 'slinear' seems to do the same as 'linear' when using the *interp* in xarray.

Interpolate over frequency

```
>>> mt_obj = MT()
>>> new_frequency = np.logspace(-3, 3, 20)
>>> new_mt_obj = mt_obj.interpolate(new_frequency, f_type="frequency")
```

plot_depth_of_penetration(***kwargs*)

Plot Depth of Penetration estimated from Niblett-Bostick estimation

Parameters

****kwargs** – DESCRIPTION

Returns

DESCRIPTION

Return type

TYPE

plot_mt_response(***kwargs*)

Returns a mtpy.imaging.plotresponse.PlotResponse object

Plot Response

```
>>> mt_obj = mt.MT(edi_file)
>>> pr = mt.plot_mt_response()
>>> # if you need more info on plot_mt_response
>>> help(pr)
```

plot_phase_tensor(***kwargs*)

Returns
DESCRIPTION

Return type
TYPE

property pt

mtpy.analysis.pt.PhaseTensor object to hold phase tensor

remove_component(*zxx=False*, *zxy=False*, *zyy=False*, *zyx=False*, *tzx=False*, *tzy=False*, *inplace=False*)

Remove a component for a given station(s)

Parameters

- **station** (*string or list*) – station name or list of station names
- **zxx** (*TYPE, optional*) – Z_xx, defaults to False
- **zxy** (*TYPE, optional*) – Z_xy, defaults to False
- **zyy** (*TYPE, optional*) – Z_yx, defaults to False
- **zyx** (*TYPE, optional*) – Z_yy, defaults to False
- **tx** (*TYPE, optional*) – T_zx, defaults to False
- **ty** (*TYPE, optional*) – T_zy, defaults to False

Returns
new data array with components removed

Return type
np.ndarray

Returns
new mt_dict with components removed

Return type
dictionary

```
>>> d = Data()
>>> d.read_data_file(r"example/data.dat")
>>> d.data, d.mt_dict = d.remove_component("mt01", zxx=True, tx=True)
```

remove_distortion(*n_frequencies=None*, *comp='det'*, *only_2d=False*, *inplace=False*)

remove distortion following Bibby et al. [2005].

Parameters
n_frequencies (*int*) – number of frequencies to look for distortion from the highest frequency

Returns
Distortion matrix

Return type
np.ndarray(2, 2, dtype=real)

Returns
Z with distortion removed

Return type
mtpy.core.z.Z

Remove distortion and write new .edi file

```
>>> import mtpy.core.mt as mt
>>> mt1 = mt.MT(fn=r"/home/mt/edi_files/mt01.edi")
>>> D, new_z = mt1.remove_distortion()
>>> mt1.write_mt_file(new_fn=r"/home/mt/edi_files/mt01_dr.edi",
↪    >>>                    new_Z=new_z)
```

remove_static_shift(*ss_x=1.0*, *ss_y=1.0*, *inplace=False*)

Remove static shift from the apparent resistivity

Assume the original observed tensor Z is built by a static shift S and an unperturbated "correct" Z0 :

- Z = S * Z0

therefore the correct Z will be :

- Z0 = S^(-1) * Z

Parameters

- **ss_x** (*float*) – correction factor for x component
- **ss_y** (*float*) – correction factor for y component

Returns
new Z object with static shift removed

Return type
mtpy.core.z.Z

Note: The factors are in resistivity scale, so the entries of the matrix "S" need to be given by their square-roots!

Remove Static Shift

```
>>> import mtpy.core.mt as mt
>>> mt_obj = mt.MT(r"/home/mt/mt01.edi")
>>> new_z_obj = mt.remove_static_shift(ss_x=.5, ss_y=1.2)
>>> mt_obj.write_mt_file(new_fn=r"/home/mt/mt01_ss.edi",
>>> ...                      new_Z_obj=new_z_obj)
```

rotate(*theta_r*, *inplace=True*)

Rotate the data in degrees assuming North is 0 measuring clockwise positive to East as 90.

Parameters

- **theta_r** (*TYPE*) – DESCRIPTION
- **inplace** (*TYPE, optional*) – DESCRIPTION, defaults to True

Returns
DESCRIPTION

Return type
TYPE

property rotation_angle

rotation angle in degrees from north

property rrhx_metadata

RRHX metadata

property rrhy_metadata

RRHY metadata

to_dataframe(*utm_crs=None*, *cols=None*)

Create a dataframe from the transfer function for use with plotting and modeling.

Parameters

utm_crs (string, int, pyproj.CRS) – the utm zone to project station to, could be a name, pyproj.CRS, EPSG number, or anything that pyproj.CRS can intake.

to_occam1d(*data_filename=None*, *mode='det'*)

write an Occam1DData data file

Parameters

- **data_filename** (*string or Path*) – path to write file, if None returns Occam1DData object.
- **mode** (*string, optional*) – ['te', 'tm', 'det', 'tez', 'tmz', 'detz'], defaults to "det"

Returns

Occam1DData object

Return type

mtpy.modeling.occam1d.Occam1DData

Example

```
>>> mt_object = MT()
>>> mt_object.read(r"/path/to/tranfer_function/file")
>>> mt_object.compute_model_z_error()
>>> occam_data = mt_object.to_occam1d(data_filename=r"/path/to/data_
↪file.dat")
```

to_simpeg_1d(*mode='det'*, ***kwargs*)

helper method to run a 1D inversion using Simpeg

default is smooth parameters

To run sharp inversion

```
>>> mt_object.to_simpeg_1d({"p_s": 2, "p_z": 0, "use_irls": True})
```

To run sharp inversion adn compact

```
>>> mt_object.to_simpeg_1d({"p_s": 0, "p_z": 0, "use_irls": True})
```

Parameters

****kwargs** – DESCRIPTION

Returns

DESCRIPTION

Return type
TYPE

mtpy.core.mt_collection module

Collection of MT stations

Created on Mon Jan 11 15:36:38 2021

copyright
Jared Peacock (jpeacock@usgs.gov)

license
MIT

class `mtpy.core.mt_collection.`**`MTCollection`**(*working_directory=None*)

Bases: `object`

Collection of transfer functions

The main working variable is *MTCollection.dataframe* which is a property that returns either the *master dataframe* that contains all the TF's in the MTH5 file, or the *working_dataframe* which is a dataframe that has been queried in some way. Therefore all the user has to do is set the working directory as a subset of the master_dataframe

Example

```
>>> mc = MTCollection()
>>> mc.open_collection(filename="path/to/example/mth5.h5")
>>> mc.working_dataframe = mc.master_dataframe.iloc[0:5]
```

`add_tf`(*transfer_function*, *new_survey=None*, *tf_id_extra=None*)

transfer_function could be a transfer function object, a file name, a list of either.

Parameters

- **transfer_function** (*list, tuple, array,* MTData, MT) – transfer function object
- **new_survey** (*str, optional*) – new survey name, defaults to None
- **tf_id_extra** (*string, optional*) – additional text onto existing 'tf_id', defaults to None

Returns
DESCRIPTION

Return type
TYPE

`apply_bbox`(*lon_min*, *lon_max*, *lat_min*, *lat_max*)

Return `pandas.DataFrame` of station within bounding box

Parameters

- **longitude_min** (*float*) – Minimum longitude
- **longitude_max** (*float*) – Maximum longitude
- **latitude_min** (*float*) – Minimum latitude
- **latitude_max** (*float*) – Maximum longitude

Returns
Only stations within the given bounding box

Return type
pandas.DataFrame

average_stations(*cell_size_m*, *bounding_box=None*, *count=1*, *n_periods=48*, *new_file=True*)

Average nearby stations to make it easier to invert

Parameters

- **cell_size_m** (*TYPE*) – DESCRIPTION
- **bounding_box** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **save_dir** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

check_for_duplicates(*locate='location'*, *sig_figs=6*)

Check for duplicate station locations in a MT DataFrame

Parameters
dataframe (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

close_collection()

close mth5

Returns
DESCRIPTION

Return type
TYPE

property dataframe

This property returns the working dataframe or master dataframe if the working dataframe is None.

Returns
DESCRIPTION

Return type
TYPE

from_mt_data(*mt_data*, *new_survey=None*, *tf_id_extra=None*)

Add data from a MTData object to an MTH5 collection.

Can use 'new_survey' to create a new survey to load to.

Can use 'tf_id_extra' to add a string onto the existing 'tf_id', useful if data have been edited or manipulated in some way. For example could set 'tf_id_extra' = 'rotated' for rotated data. This will help you organize the tf's for each station.

Parameters

- **mt_data** (*mtpy.core.mt_data.MTData*) – MTData object
- **new_survey** (*str, optional*) – new survey name, defaults to None
- **tf_id_extra** (*string, optional*) – additional text onto existing 'tf_id', defaults to None

Raises
IOError – If an MTH5 is not writable raises

get_tf(*tf_id*, *survey=None*)
Get transfer function

Parameters
tf_id (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

has_data()

static **make_file_list**(*mt_path*, *file_types=['edi']*)
Get a list of MT file from a given path

Parameters
mt_path – full path to where the MT transfer functions are stored

or a list of paths :type mt_path: string or pathlib.Path or list

Parameters
file_types (*list*) – List of file types to look for given their extension

Currently available file types are or will be:

- edi - EDI files
- zmm - EMTF output file
- j - BIRRP output file
- avg - Zonge output file

property **master_dataframe**
This is the full summary of all transfer functions in the MTH5 file. It is a property because if a user adds TF's then the master_df will be automatically updated. the tranformation is quick for now.

property **mth5_filename**

open_collection(*filename=None*, *basename=None*, *working_directory=None*, *mode='a'*)
Initialize an mth5

Parameters

- **basename** (*TYPE, optional*) – DESCRIPTION, defaults to "mt_collection"
- **working_directory** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

plot_mt_response(*tf_id*, *survey=None*, ***kwargs*)

Parameters

- **tf_id** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

if input as list, tuple, np.ndarray, pd.series assuming first column is tf_id, and if needed the second column should be the survey id for that tf.

plot_penetration_depth_1d(*tf_id*, *survey=None*, ***kwargs*)

Plot 1D penetration depth based on the Niblett-Bostick transformation

Note that data is rotated to estimated strike previous to estimation and strike angles are interpreted for data points that are 3D.

See also:

`mtpy.analysis.niblettbostick.calculate_depth_of_investigation`

Parameters

- **tf_object** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_penetration_depth_map(*mt_data=None*, ***kwargs*)

Plot Penetration depth in map view for a single period

See also:

mtpy.imaging.PlotPenetrationDepthMap

Parameters
mt_data (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_phase_tensor(*tf_id*, *survey=None*, ***kwargs*)

plot phase tensor elements

Parameters

- **tf_id** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_phase_tensor_map(*mt_data=None*, ***kwargs*)

Plot Phase tensor maps for transfer functions in the working_dataframe

See also:

mtpy.imaging.PlotPhaseTensorMaps

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_phase_tensor_pseudosection(*mt_data=None*, ***kwargs*)

Plot a pseudo section of phase tensor ellipses and induction vectors if specified

See also:

`mtpy.imaging.PlotPhaseTensorPseudosection`

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_residual_phase_tensor(*mt_data_01*, *mt_data_02*, *plot_type='map'*, ***kwargs*)

Parameters

- **mt_data_01** (*TYPE*) – DESCRIPTION
- **mt_data_02** (*TYPE*) – DESCRIPTION
- **plot_type** (*TYPE, optional*) – DESCRIPTION, defaults to "map"
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_resistivity_phase_maps(*mt_data=None*, ***kwargs*)

Plot apparent resistivity and/or impedance phase maps from the working dataframe

See also:

mtpy.imaging.PlotResPhaseMaps

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_resistivity_phase_pseudosections(*mt_data=None*, ***kwargs*)

Plot resistivity and phase in a pseudosection along a profile line

Parameters

- **mt_data** (*TYPE, optional*) – DESCRIPTION, defaults to None
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_stations(*map_epsg=4326*, *bounding_box=None*, ***kwargs*)

plot stations

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_strike(*mt_data=None*, ***kwargs*)

Plot strike angle

See also:

mtpy.imaging.PlotStrike

to_geo_df(*bounding_box=None*, *epsg=4326*)

Make a geopandas dataframe for easier GIS manipulation

to_mt_data(*bounding_box=None*, ***kwargs*)

Get a list of transfer functions

Parameters

- **tf_ids** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **bounding_box** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

to_shp(*filename*, *bounding_box=None*, *epsg=4326*)

Parameters

- **filename** (*TYPE*) – DESCRIPTION
- **bounding_box** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **epsg** (*TYPE, optional*) – DESCRIPTION, defaults to 4326

Returns
DESCRIPTION

Return type
TYPE

property working_directory

mtpy.core.mt_data module

Created on Mon Oct 10 11:58:56 2022

@author: jpeacock

class mtpy.core.mt_data.**MTData**(*mt_list=None*, ***kwargs*)

Bases: OrderedDict, *MTStations*

Collection of MT objects as an OrderedDict where keys are formatted as *survey_id.station_id*. Has all functionality of an OrderedDict for example can iterate of *.keys()*, *.values()* or *.items*. Values are a list of MT objects.

Inherits mtpyt.core.MTStations to deal with geographic locations of stations.

Is not optimized yet for speed, works fine for smaller surveys, but for large can be slow. Might try using a dataframe as the base.

add_station(*mt_object*, *survey=None*, *compute_relative_location=True*, *interpolate_periods=None*, *compute_model_error=False*)

Add a MT object

Parameters

- **mt_object** (*mtpy.MT*) – MT object for a single station
- **survey** (*str, optional*) – new survey name, defaults to None
- **compute_relative_location** (*bool, optional*) – Compute relative location, can be slow if adding single stations in a loop. If looping over station set to False and compute at the end, defaults to True
- **interpolate_periods** (*np.array, optional*) – periods to interpolate onto, defaults to None

add_tf(*tf*, ***kwargs*)

Add a MT object

Parameters

- **mt_object** (*mtpy.MT*) – MT object for a single station
- **survey** (*str, optional*) – new survey name, defaults to None

- **compute_relative_location** (*bool, optional*) – Compute relative location, can be slow if adding single stations in a loop. If looping over station set to False and compute at the end, defaults to True
- **interpolate_periods** (*np.array, optional*) – periods to interpolate onto, defaults to None

add_white_noise(*value*, *inplace=True*)

Add white noise to the data, useful for synthetic tests.

Parameters

- **value** (*TYPE*) – DESCRIPTION
- **inplace** (*TYPE, optional*) – DESCRIPTION, defaults to True

Returns
DESCRIPTION

Return type
TYPE

clone_empty()

Return a copy of MTData excluding MT objects.

Returns
Copy of MTData object excluding MT objects

Return type
mtpy.MTData

compute_model_errors(*z_error_value=None*, *z_error_type=None*, *z_floor=None*, *t_error_value=None*, *t_error_type=None*, *t_floor=None*)

Compute mode errors based on the error type

key	definition
egbert	error_value * sqrt(Zxy * Zyx)
geometric_mean	error_value * sqrt(Zxy * Zyx)
arithmetic_mean	error_value * (Zxy + Zyx) / 2
mean_od	error_value * (Zxy + Zyx) / 2
off_diagonals	zxx_err == zxy_err, zyx_err == zyy_err
median	error_value * median(z)
eigen	error_value * mean(eigen(z))
percent	error_value * z
absolute	error_value

Parameters

- **z_error_value** (*TYPE, optional*) – DESCRIPTION, defaults to 5
- **z_error_type** (*TYPE, optional*) – DESCRIPTION, defaults to "geometric_mean"
- **z_floor** (*TYPE, optional*) – DESCRIPTION, defaults to True
- **t_error_value** (*TYPE, optional*) – DESCRIPTION, defaults to 0.02
- **t_error_type** (*TYPE, optional*) – DESCRIPTION, defaults to "absolute"
- **t_floor** (*TYPE, optional*) – DESCRIPTION, defaults to True

:param : DESCRIPTION :type : TYPE :return: DESCRIPTION :rtype: TYPE

copy()

Deep copy of original MTData object

Parameters
memo (*TYPE*) – DESCRIPTION

Returns
Deep copy of original MTData

Return type
mtpy.MTData

estimate_spatial_static_shift(*station_key*, *radius*, *period_min*, *period_max*, *radius_units='m'*, *shift_tolerance=0.15*)

Estimate static shift for a station by estimating the median resistivity values for nearby stations within a radius given. Can set the period range to estimate the resistivity values.

Parameters

- **station_key** (*TYPE*) – DESCRIPTION
- **radius** (*TYPE*) – DESCRIPTION
- **period_min** (*TYPE*) – DESCRIPTION
- **period_max** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

estimate_starting_rho()

Estimate starting resistivity from the data. Creates a plot of the mean and median apparent resistivity values.

Returns
array of the median rho per period

Return type
np.ndarray(n_periods)

Returns
array of the mean rho per period

Return type
np.ndarray(n_periods)

```
>>> d = Data()
>>> d.read_data_file(r"example/data.dat")
>>> rho_median, rho_mean = d.estimate_starting_rho()
```

from_dataframe(*df*)

Create an dictionary of MT objects from a dataframe

Parameters
df (*pandas.DataFrame*) – dataframe of mt data

Returns
DESCRIPTION

Return type
TYPE

from_modem(*data_filename*, *survey='data'*, ***kwargs*)

read in a modem data file

Parameters

- **data_filename** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

from_modem_data(*data_filename*, *survey='data'*, ***kwargs*)

Parameters

- **data_filename** (*TYPE*) – DESCRIPTION
- **file_type** (*TYPE, optional*) – DESCRIPTION, defaults to "data"
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

from_occam2d(*data_filename*, *file_type='data'*, ***kwargs*)

read in occam data from a 2D data file *.dat

Read data file and plot

```
>>> from mtpy import MTData
>>> md = MTData()
>>> md.from_occam2d_data(f"/path/to/data/file.dat")
>>> plot_stations = md.plot_stations(model_locations=True)
```

Read response file

```
>>> md.from_occam2d_data(f"/path/to/response/file.dat")
```

Note: When reading in a response file the survey will be called model. So now you can have the data and model response in the same object.

from_occam2d_data(*data_filename*, *file_type='data'*, ***kwargs*)

get_nearby_stations(*station_key*, *radius*, *radius_units='m'*)

get stations close to a given station

Parameters

- **station_key** (*TYPE*) – DESCRIPTION
- **radius** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

get_periods()

get all unique periods

Returns
DESCRIPTION

Return type
TYPE

get_profile(*x1*, *y1*, *x2*, *y2*, *radius*)

Get stations along a profile line given the (x1, y1) and (x2, y2) coordinates within a given radius (in meters).

These can be in (longitude, latitude) or (easting, northing). The calculation is done in UTM, therefore a UTM CRS must be input

Parameters

- **x1** (*TYPE*) – DESCRIPTION
- **y1** (*TYPE*) – DESCRIPTION
- **x2** (*TYPE*) – DESCRIPTION
- **y2** (*TYPE*) – DESCRIPTION
- **radius** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

get_station(*station_id=None*, *survey_id=None*, *station_key=None*)

if 'station_key' is None, tries to find key from *station_id* and 'survey_id' using MTData._get_station_key()

Parameters

- **station_key** (*str, optional*) – full station key {survey_id}.{station_id}, defaults to None
- **station_id** (*str, optional*) – station ID, defaults to None
- **survey_id** (*str, optional*) – survey ID, defaults to None

Raises
KeyError – If cannot find station_key

Returns
MT object

Return type
mtpy.MT

get_subset(*station_list*)

get a subset of the data from a list of stations, could be station_id or station_keys

Parameters
station_list (*list*) – list of station keys as {survey_id}.{station_id}

Returns
Returns just those stations within station_list

Return type
mtpy.MTData

get_survey(*survey_id*)

Get all MT objects that belong to the 'survey_id' from the data set.

Parameters
survey_id (*str*) – survey ID

Returns
MTData object including only those with the desired 'survey_id'

Return type
mtpy.MTData

interpolate(*new_periods*, *f_type='period'*, *inplace=True*)

Interpolate onto common period range

Parameters

- **new_periods** (*TYPE*) – DESCRIPTION
- **f_type** (*string, defaults to 'period'*) – frequency type can be ['frequency' | 'period']

Returns
DESCRIPTION

Return type
TYPE

property mt_list

List of MT objects :rtype: list

Type
return

property n_stations

number of stations in MT data

plot_mt_response(*station_key=None*, *station_id=None*, *survey_id=None*, ***kwargs*)

Parameters

- **tf_id** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

if input as list, tuple, np.ndarray, pd.series assuming first column is tf_id, and if needed the second column should be the survey id for that tf.

plot_penetration_depth_1d(*station_key=None*, *station_id=None*, *survey_id=None*, ***kwargs*)

Plot 1D penetration depth based on the Niblett-Bostick transformation

Note that data is rotated to estimated strike previous to estimation and strike angles are interpreted for data points that are 3D.

See also:

`mtpy.analysis.niblettbostick.calculate_depth_of_investigation`

Parameters

- **tf_object** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_penetration_depth_map(***kwargs*)

Plot Penetration depth in map view for a single period

See also:

mtpy.imaging.PlotPenetrationDepthMap

Parameters
mt_data (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_phase_tensor(*station_key=None*, *station_id=None*, *survey_id=None*, ***kwargs*)

plot phase tensor elements

Parameters

- **tf_id** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_phase_tensor_map(***kwargs*)

Plot Phase tensor maps for transfer functions in the working_dataframe

See also:

mtpy.imaging.PlotPhaseTensorMaps

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`plot_phase_tensor_pseudosection`(*mt_data=None*, ***kwargs*)

Plot a pseudo section of phase tensor ellipses and induction vectors if specified

See also:

`mtpy.imaging.PlotPhaseTensorPseudosection`

Parameters
`**kwargs` – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`plot_residual_phase_tensor_maps`(*survey_01*, *survey_02*, ***kwargs*)

Parameters

- **survey_01** (*TYPE*) – DESCRIPTION
- **survey_02** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`plot_resistivity_phase_maps`(***kwargs*)

Plot apparent resistivity and/or impedance phase maps from the working dataframe

See also:

`mtpy.imaging.PlotResPhaseMaps`

Parameters
`**kwargs` – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`plot_resistivity_phase_pseudosections`(***kwargs*)

Plot resistivity and phase in a pseudosection along a profile line

Parameters

- **mt_data** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **`**kwargs`** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_stations(*map_epsg=4326*, *bounding_box=None*, *model_locations=False*, ***kwargs*)

plot stations

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_strike(***kwargs*)

Plot strike angle

See also:

mtpy.imaging.PlotStrike

remove_station(*station_id*, *survey_id=None*)

remove a station from the dictionary based on the key

Parameters

- **station_id** (*str*) – station ID
- **survey_id** (*str*) – survey ID

rotate(*rotation_angle*, *inplace=True*)

rotate the data by the given angle assuming positive clockwise with north = 0, east = 90.

Parameters
rotation_angle (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

property survey_ids

Survey IDs for all MT objects

Returns
list of survey IDs

Return type
list

to_dataframe(*utm_crs=None*, *cols=None*)

Parameters

- **utm_crs** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **cols** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

to_geo_df(*model_locations=False*)

Make a geopandas dataframe for easier GIS manipulation

to_modem(*data_filename=None*, ***kwargs*)

Create a modem data file

Parameters

- **data_filename** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns

DESCRIPTION

Return type

TYPE

to_modem_data(*data_filename=None*, ***kwargs*)

to_occam2d(*data_filename=None*, ***kwargs*)

write an Occam2D data file

Parameters

- **data_filename** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns

DESCRIPTION

Return type

TYPE

to_occam2d_data(*data_filename=None*, ***kwargs*)

mtpy.core.mt_dataframe module

Created on Sun Oct 2 13:20:28 2022

@author: jpeacock

class mtpy.core.mt_dataframe.**MTDataFrame**(*data=None*, *n_entries=0*, ***kwargs*)

Bases: `object`

Dataframe for a single station

Tried subclassing pandas.DataFrame, but that turned out to not be straight forward, so when with compilation instead.

Think about having period as an index?

property datum_epsg

property east

station

property elevation

property frequency

Get frequencies

Returns
DESCRIPTION

Return type
TYPE

from_t_object(*t_object*)

Fill tipper :param tipper: DESCRIPTION :type tipper: TYPE :param tipper_error: DESCRIPTION :type tipper_error: TYPE :param index: DESCRIPTION :type index: TYPE :return: DESCRIPTION :rtype: TYPE

from_z_object(*z_object*)

Fill impedance :param impedance: DESCRIPTION :type impedance: TYPE :param index: DESCRIPTION :type index: TYPE :return: DESCRIPTION :rtype: TYPE

get_station_df(*station=None*)

get a single station df

Returns
DESCRIPTION

Return type
TYPE

property impedance

Impedance elements

property latitude

property longitude

property model_east

property model_elevation

property model_north

property nonzero_items

return number of non zero entries

property north

property period

Get frequencies

Returns
DESCRIPTION

Return type
TYPE

property profile_offset

property size

property station

station name

property station_locations

property survey

survey name

to_t_object()

To a tipper object

Returns
DESCRIPTION

Return type
TYPE

to_z_object()

fill z_object from dataframe

Need to have the components this way for transposing the elements so that the shape is (nf, 2, 2)

property utm_epsg

mtpy.core.mt_location module

Might think about adding declination

Created on Mon Oct 3 15:04:12 2022

@author: jpeacock

class mtpy.core.mt_location.**MTLocation**(*survey_metadata=None*, ***kwargs*)

Bases: `object`

Location for a MT site or point measurement

compute_model_location(*center_location*)

compute model location based on model center and model epsg

Parameters
model_center (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

copy()

property datum_crs

property datum_epsg

property datum_name

property east

easting

property elevation

`from_json`(*filename*)

read in json formatted location

Parameters
`filename` (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`get_elevation_from_national_map`()

Get elevation from DEM data of the US National Map. Plan to extend this to the globe.

Pulls data from the USGS national map DEM

Returns
DESCRIPTION

Return type
TYPE

`property latitude`

`property longitude`

`property model_east`

`property model_elevation`

`property model_north`

`property north`

northing

`project_onto_profile_line`(*profile_slope*, *profile_intersection*)

Parameters

- `profile_slope` (*TYPE*) – DESCRIPTION
- `profile_intersection` (*TYPE*) – DESCRIPTION
- `units` (*TYPE, optional*) – DESCRIPTION, defaults to "deg"

Returns
DESCRIPTION

Return type
TYPE

`to_json`(*filename*)

write out information to a json file

Parameters
`filename` (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

property utm_crs

property utm_epsg

property utm_name

property utm_zone

mtpy.core.mt_stations module

ModEM

Generate files for ModEM

revised by JP 2017 # revised by AK 2017 to bring across functionality from ak branch

class mtpy.core.mt_stations.**MTStations**(*utm_epsg*, *datum_epsg=None*, ***kwargs*)

Bases: `object`

Object to deal with station location and geographic projection.

Geographic projections are done using pyproj.CRS objects.

Takes in a list of `mtpy.core.mt.MT` objects which are inherit `mtpy.core.mt_location.MTLocation` objects, which deal with transformation of point data using pyproj.

property center_point

calculate the center point from the given station locations

If _center attributes are set, that is returned as the center point.

Otherwise, looks for non-zero locations in E-N first, then Lat/Lon and estimates the center point as (max - min) / 2.

Returns
Center point

Return type
`mtpy.core.MTLocation`

center_stations(*model_obj*)

Center station locations to the middle of cells, is useful for topography cause it reduces edge effects of stations close to cell edges. Recalculates rel_east, rel_north to center of model cell.

Parameters
model_obj (`mtpy.modeling.modem.Model`) – `mtpy.modeling.Structured` object of the model

compute_relative_locations()

Calculate model station locations relative to the center point in meters.

Uses *mtpy.core.MTLocation.compute_model_location* to calculate the relative distance.

Computes inplace.

copy()

create a deep copy of the MTStations object.

Note: At the moment this is very slow because it is making a lot of deep copies. Use sparingly.

Returns
deep copy of MTStation object

Return type
`mtpy.core.mt_stations.MTStations`

property datum_crs
Datum CRS object :rtype: `pyproj.CRS`

Type
return

property datum_epsg
Datum EPSG number :rtype: int

Type
return

property datum_name
Datum well known name :rtype: str

Type
return

generate_profile(*units='deg'*)
Estimate a profile from the data :return: DESCRIPTION :rtype: TYPE

generate_profile_from_strike(*strike*, *units='deg'*)
Estimate a profile line from a given geoelectric strike

Parameters
units (*TYPE, optional*) – DESCRIPTION, defaults to "deg"

Returns
DESCRIPTION

Return type
TYPE

property model_epsg
model epsg number from the model_crs object :rtype: int

Type
return

project_stations_on_topography(*model_object*, *air_resistivity=1000000000000.0*, *sea_resistivity=0.3*, *ocean_bottom=False*)
Project stations on topography of a given model

Parameters

- **model_obj** (`mtpy.modeling.modem.Model`) – `mtpy.modeling.modem.Model` object of the model
- **air_resistivity** (*float*) – resistivity value of air cells in the model
- **sea_resistivity** (*float*) – resistivity of sea
- **ocean_bottom** (*boolean*) – If True places stations at bottom of sea cells

Recaluclates rel_elev

`rotate_stations`(*rotation_angle*)

Rotate stations model postions only assuming N is 0 and east is 90.

Note: Computes in place and rotates according to already set rotation angle. Therefore if the station locations have already been rotated the function will rotate the already rotate stations. For example if you rotate the stations 15 degrees, then again by 20 degrees the resulting station locations will be 35 degrees rotated from the original locations.

Parameters

rotation_angle (*float*) – rotation angle in degrees assuming N=0, E=90. Positive clockwise.

`property station_locations`

Dataframe of station location information :rtype: `pandas.DataFrame`

Type

return

`to_csv`(*csv_fn*, *geometry=False*)

Write a shape file of the station locations using geopandas which only takes in epsg numbers

Parameters

csv_fn (*string*) – full path to new shapefile

`to_geopd`()

create a geopandas dataframe

Returns

Geopandas DataFrame with points from latitude and longitude

Return type

`geopandas.DataFrame`

`to_shp`(*shp_fn*)

Write a shape file of the station locations using geopandas which only takes in epsg numbers

Parameters

shp_fn (*string*) – full path to new shapefile

`to_vtk`(*vtk_fn=None*, *vtk_save_path=None*, *vtk_fn_basename='ModEM_stations'*, *geographic=False*, *shift_east=0*, *shift_north=0*, *shift_elev=0*, *units='km'*, *coordinate_system='nez+'*)

Parameters

- **vtk_save_path** (*string or Path , optional*) – directory to save vtk file to, defaults to None
- **vtk_fn_basename** – filename basename of vtk file, note that .vtr

extension is automatically added, defaults to "ModEM_stations" :type vtk_fn_basename: string, optional :param geographic: If true puts the grid on geographic coordinates based on the model_utm_zone, defaults to False :type geographic: boolean, optional :param shift_east: shift in east directions in meters, defaults to 0 :type shift_east: float, optional :param shift_north: shift in north direction in meters, defaults to 0 :type shift_north: float, optional :param shift_elev: shift in elevation + down in meters, defaults to 0 :type shift_elev: float, optional :param units: Units of the spatial grid [km | m | ft], defaults to "km" :type units: string, optional :type : string :param coordinate_system: coordinate system for the station, either the normal MT right-hand coordinate system with z+ down or the sinister z- down [nez+ | enz-], defaults to nez+ :return: full path to VTK file :rtype: Path

Write VTK file >>> md.write_vtk_station_file(vtk_fn_basename="modem_stations")

Write VTK file in geographic coordinates >>> md.write_vtk_station_file(vtk_fn_basename="modem_stations", >>> ... geographic=True)

Write VTK file in geographic coordinates with z+ up >>> md.write_vtk_station_file(vtk_fn_basename="modem_stations", >>> ... geographic=True, >>> ... coordinate_system='enz-')

property utm_crs

UTM CRS object :rtype: `pyproj.CRS`

Type
return

property utm_epsg

UTM EPSG number :rtype: int

Type
return

property utm_name

UTM CRS name :rtype: string

Type
return

property utm_zone

UTM Zone number :rtype: str

Type
return

Module contents

class mtpy.core.**MTDataFrame**(*data=None*, *n_entries=0*, ***kwargs*)

Bases: `object`

Dataframe for a single station

Tried subclassing pandas.DataFrame, but that turned out to not be straight forward, so when with compilation instead.

Think about having period as an index?

property datum_epsg

property east

station

property elevation

property frequency

Get frequencies

Returns
DESCRIPTION

Return type
TYPE

from_t_object(*t_object*)

Fill tipper :param tipper: DESCRIPTION :type tipper: TYPE :param tipper_error: DESCRIPTION :type tipper_error: TYPE :param index: DESCRIPTION :type index: TYPE :return: DESCRIPTION :rtype: TYPE

from_z_object(*z_object*)

Fill impedance :param impedance: DESCRIPTION :type impedance: TYPE :param index: DESCRIPTION :type index: TYPE :return: DESCRIPTION :rtype: TYPE

get_station_df(*station=None*)

get a single station df

Returns
DESCRIPTION

Return type
TYPE

property impedance

Impedance elements

property latitude

property longitude

property model_east

property model_elevation

property model_north

property nonzero_items

return number of non zero entries

property north

property period

Get frequencies

Returns
DESCRIPTION

Return type
TYPE

property profile_offset

property size

property station

station name

property station_locations

property survey

survey name

to_t_object()

To a tipper object

Returns
DESCRIPTION

Return type
TYPE

to_z_object()

fill z_object from dataframe

Need to have the components this way for transposing the elements so that the shape is (nf, 2, 2)

property utm_epsg

class mtpy.core.MTLocation(*survey_metadata=None*, ***kwargs*)

Bases: object

Location for a MT site or point measurement

compute_model_location(*center_location*)

compute model location based on model center and model epsg

Parameters
model_center (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

copy()

property datum_crs

property datum_epsg

property datum_name

property east

easting

property elevation

from_json(*filename*)

read in json formatted location

Parameters
filename (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

get_elevation_from_national_map()

Get elevation from DEM data of the US National Map. Plan to extend this to the globe.

Pulls data from the USGS national map DEM

Returns
DESCRIPTION

Return type
TYPE

property latitude

property longitude

property model_east

property model_elevation

property model_north

property north
northing

project_onto_profile_line(*profile_slope*, *profile_intersection*)

Parameters

- **profile_slope** (*TYPE*) – DESCRIPTION
- **profile_intersection** (*TYPE*) – DESCRIPTION
- **units** (*TYPE, optional*) – DESCRIPTION, defaults to "deg"

Returns
DESCRIPTION

Return type
TYPE

to_json(*filename*)
write out information to a json file

Parameters
filename (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

property utm_crs

property utm_epsg

property utm_name

property utm_zone

class mtpy.core.**MTStations**(*utm_epsg*, *datum_epsg=None*, ***kwargs*)
Bases: object

Object to deal with station location and geographic projection.

Geographic projections are done using pyproj.CRS objects.

Takes in a list of *mtpy.core.mt.MT* objects which are inherit *mtpy.core.mt_location.MTLocation* objects, which deal with transformation of point data using pyproj.

property center_point

calculate the center point from the given station locations

If _center attributes are set, that is returned as the center point.

Otherwise, looks for non-zero locations in E-N first, then Lat/Lon and estimates the center point as (max - min) / 2.

Returns
Center point

Return type
mtpy.core.MTLocation

center_stations(*model_obj*)

Center station locations to the middle of cells, is useful for topography cause it reduces edge effects of stations close to cell edges. Recalculates rel_east, rel_north to center of model cell.

Parameters
model_obj (*mtpy.modeling.modem.Model*) – `mtpy.modeling.Structured` object of the model

compute_relative_locations()

Calculate model station locations relative to the center point in meters.

Uses *mtpy.core.MTLocation.compute_model_location* to calculate the relative distance.

Computes inplace.

copy()

create a deep copy of the MTStations object.

Note: At the moment this is very slow because it is making a lot of deep copies. Use sparingly.

Returns
deep copy of MTStation object

Return type
mtpy.core.mt_stations.MTStations

property datum_crs

Datum CRS object :rtype: `pyproj.CRS`

Type
return

property datum_epsg

Datum EPSG number :rtype: int

Type
return

property datum_name

Datum well known name :rtype: str

Type
return

generate_profile(*units='deg'*)

Estimate a profile from the data :return: DESCRIPTION :rtype: TYPE

generate_profile_from_strike(*strike*, *units='deg'*)

Estimate a profile line from a given geoelectric strike

Parameters
units (*TYPE, optional*) – DESCRIPTION, defaults to "deg"

Returns
DESCRIPTION

Return type
TYPE

property model_epsg

model epsg number from the model_crs object :rtype: int

Type
return

project_stations_on_topography(*model_object*, *air_resistivity=1000000000000.0*, *sea_resistivity=0.3*, *ocean_bottom=False*)

Project stations on topography of a given model

Parameters

- **model_obj** (`mtpy.modeling.modem.Model`) – `mtpy.modeling.modem.Model` object of the model
- **air_resistivity** (*float*) – resistivity value of air cells in the model
- **sea_resistivity** (*float*) – resistivity of sea
- **ocean_bottom** (*boolean*) – If True places stations at bottom of sea cells

Recaluclates rel_elev

rotate_stations(*rotation_angle*)

Rotate stations model postions only assuming N is 0 and east is 90.

Note: Computes in place and rotates according to already set rotation angle. Therefore if the station locations have already been rotated the function will rotate the already rotate stations. For example if you rotate the stations 15 degrees, then again by 20 degrees the resulting station locations will be 35 degrees rotated from the original locations.

Parameters
rotation_angle (*float*) – rotation angle in degrees assuming N=0, E=90. Positive clockwise.

property station_locations

Dataframe of station location information :rtype: `pandas.DataFrame`

Type
return

to_csv(*csv_fn*, *geometry=False*)

Write a shape file of the station locations using geopandas which only takes in epsg numbers

Parameters
`csv_fn` (*string*) – full path to new shapefile

`to_geopd()`

create a geopandas dataframe

Returns
Geopandas DataFrame with points from latitude and longitude

Return type
`geopandas.DataFrame`

`to_shp(shp_fn)`

Write a shape file of the station locations using geopandas which only takes in epsg numbers

Parameters
`shp_fn` (*string*) – full path to new shapefile

`to_vtk(vtk_fn=None, vtk_save_path=None, vtk_fn_basename='ModEM_stations', geographic=False, shift_east=0, shift_north=0, shift_elev=0, units='km', coordinate_system='nez+')`

Parameters

- `vtk_save_path` (*string or Path, optional*) – directory to save vtk file to, defaults to None
- `vtk_fn_basename` – filename basename of vtk file, note that .vtr

extension is automatically added, defaults to "ModEM_stations" :type vtk_fn_basename: string, optional :param geographic: If true puts the grid on geographic coordinates based on the model_utm_zone, defaults to False :type geographic: boolean, optional :param shift_east: shift in east directions in meters, defaults to 0 :type shift_east: float, optional :param shift_north: shift in north direction in meters, defaults to 0 :type shift_north: float, optional :param shift_elev: shift in elevation + down in meters, defaults to 0 :type shift_elev: float, optional :param units: Units of the spatial grid [km | m | ft], defaults to "km" :type units: string, optional :type : string :param coordinate_system: coordinate system for the station, either the normal MT right-hand coordinate system with z+ down or the sinister z- down [nez+ | enz-], defaults to nez+ :return: full path to VTK file :rtype: Path

Write VTK file >>> md.write_vtk_station_file(vtk_fn_basename="modem_stations")

Write VTK file in geographic coordinates >>> md.write_vtk_station_file(vtk_fn_basename="modem_stations", >>> … geographic=True)

Write VTK file in geographic coordinates with z+ up >>> md.write_vtk_station_file(vtk_fn_basename="modem_stations", >>> … geographic=True, >>> … coordinate_system='enz-')

`property utm_crs`

UTM CRS object :rtype: `pyproj.CRS`

Type
return

`property utm_epsg`

UTM EPSG number :rtype: int

Type
return

`property utm_name`

UTM CRS name :rtype: string

Type
return

property utm_zone
UTM Zone number :rtype: str

Type
return

class mtpy.core.PhaseTensor(*z=None*, *z_error=None*, *z_model_error=None*, *frequency=None*, *pt=None*, *pt_error=None*, *pt_model_error=None*)

Bases: *TFBase*

PhaseTensor class - generates a Phase Tensor (phase tensor) object.

Methods include reading and writing from and to edi-objects, rotations combinations of Z instances, as well as calculation of invariants, inverse, amplitude/phase,…

phase tensor is a complex array of the form (n_freq, 2, 2), with indices in the following order:

- phase tensor_xx: (0,0)
- phase tensor_xy: (0,1)
- phase tensor_yx: (1,0)
- phase tensor_yy: (1,1)

All internal methods are based on (Caldwell et al.,2004) and (Bibby et al.,2005), in which they use the canonical cartesian 2D reference (x1, x2). However, all components, coordinates, and angles for in- and outputs are given in the geographical reference frame:

- x-axis = North
- y-axis = East
- z-axis = Down

Therefore, all results from using those methods are consistent (angles are referenced from North rather than x1).

property alpha
Principal axis angle (strike) of phase tensor in degrees

property alpha_error
Principal axis angle error of phase tensor in degrees

property alpha_model_error
Principal axis angle model error of phase tensor in degrees

property azimuth
Azimuth angle related to geoelectric strike in degrees

property azimuth_error
Azimuth angle error related to geoelectric strike in degrees

property azimuth_model_error
Azimuth angle model error related to geoelectric strike in degrees

property beta
3D-dimensionality angle Beta (invariant) of phase tensor in degrees

property beta_error

3D-dimensionality angle error Beta of phase tensor in degrees

property beta_model_error

3D-dimensionality angle model error Beta of phase tensor in degrees

property det

Determinant of phase tensor

property det_error

Determinant error of phase tensor

property det_model_error

Determinant model erro of phase tensor

property eccentricity

eccentricty estimation of dimensionality

property eccentricity_error

Error in eccentricity estimation

property eccentricity_model_error

Error in eccentricity estimation

property ellipticity

Ellipticity of the phase tensor, related to dimesionality

property ellipticity_error

Ellipticity error of the phase tensor, related to dimesionality

property ellipticity_model_error

Ellipticity model error of the phase tensor, related to dimesionality

property only1d

Return phase tensor in 1D form.

If phase tensor is not 1D per se, the diagonal elements are set to zero, the off-diagonal elements keep their signs, but their absolute is set to the mean of the original phase tensor off-diagonal absolutes.

property only2d

Return phase tensor in 2D form.

If phase tensor is not 2D per se, the diagonal elements are set to zero.

property phimax

Maximum phase calculated according to Bibby et al. 2005:

Phi_max = Pi2 + Pi1

property phimax_error

Maximum phase error

property phimax_model_error

Maximum phase model error

property phimin

minimum phase calculated according to Bibby et al. 2005:

Phi_min = Pi2 - Pi1

property phimin_error

minimum phase error

property phimin_model_error

minimum phase model error

property pt

Phase tensor array

property pt_error

Phase tensor error

property pt_model_error

Phase tensor model error

property skew

3D-dimensionality skew angle of phase tensor in degrees

property skew_error

3D-dimensionality skew angle error of phase tensor in degrees

property skew_model_error

3D-dimensionality skew angle model error of phase tensor in degrees

property trace

Trace of phase tensor

property trace_error

Trace error of phase tensor

property trace_model_error

Trace model error of phase tensor

class mtpy.core.**Tipper**(*tipper=None*, *tipper_error=None*, *frequency=None*, *tipper_model_error=None*)

Bases: *TFBase*

Tipper class –> generates a Tipper-object.

Errors are given as standard deviations (sqrt(VAR))

Parameters

- **tipper** (*np.ndarray((nf, 1, 2), dtype='complex')*) – tipper array in the shape of [Tx, Ty] *default* is None
- **tipper_error** (*np.ndarray((nf, 1, 2))*) – array of estimated tipper errors in the shape of [Tx, Ty]. Must be the same shape as tipper. *default* is None
- **frequency** (*np.ndarray(nf)*) – array of frequencyuencies corresponding to the tipper elements. Must be same length as tipper. *default* is None

Attributes	Description
frequency	array of frequencyuencies corresponding to elements of z
rotation_angle	angle of which data is rotated by
tipper	tipper array
tipper_error	tipper error array

Methods	Description
mag_direction	computes magnitude and direction of real and imaginary induction arrows.
amp_phase	computes amplitude and phase of Tx and Ty.
rotate	rotates the data by the given angle

property amplitude

property amplitude_error

property amplitude_model_error

property angle_error

property angle_imag

property angle_model_error

property angle_real

property mag_error

property mag_imag

property mag_model_error

property mag_real

property phase

property phase_error

property phase_model_error

set_amp_phase(*r*, *phi*)

Set values for amplitude(r) and argument (phi - in degrees).

Updates the attributes:

- tipper
- tipper_error

set_mag_direction(*mag_real*, *ang_real*, *mag_imag*, *ang_imag*)

computes the tipper from the magnitude and direction of the real and imaginary components.

Updates tipper

No error propagation yet

property tipper

property tipper_error

property tipper_model_error

class mtpy.core.**Z**(*z=None*, *z_error=None*, *frequency=None*, *z_model_error=None*)

Bases: *TFBase*

Z class - generates an impedance tensor (Z) object.

Z is a complex array of the form (n_frequency, 2, 2), with indices in the following order:

- Zxx: (0,0)
- Zxy: (0,1)
- Zyx: (1,0)
- Zyy: (1,1)

All errors are given as standard deviations (sqrt(VAR))

Parameters

- **z** (*numpy.ndarray(n_frequency, 2, 2)*) – array containing complex impedance values
- **z_error** (*numpy.ndarray(n_frequency, 2, 2)*) – array containing error values (standard deviation) of impedance tensor elements
- **frequency** (*np.ndarray(n_frequency)*) – array of frequencyuency values corresponding to impedance tensor elements.

Create Impedance from scracth

```
>>> import mtpy.core import Z
>>> import numpy as np
>>> z_test = np.array([[0+0j, 1+1j], [-1-1j, 0+0j]])
>>> z_object = Z(z=z_test, frequency=[1])
>>> z_object.rotate(45)
>>> z_object.resistivity
```

Create from resistivity and phase

```
>>> z_object = Z()
>>> z_object.set_resistivity_phase(
    np.array([[5, 100], [100, 5]]),
    np.array([[90, 45], [-135, -90]]),
    np.array([1])
    )
>>> z_object.z
array([[[ 3.06161700e-16 +5.j, 1.58113883e+01+15.8113883j],
        [-1.58113883e+01-15.8113883j, 3.06161700e-16 -5.j ]]])
```

property det

determinant of impedance

property det_error

Return the determinant of impedance error

property det_model_error

Return the determinant of impedance model error

estimate_depth_of_investigation()

estimate depth of investigation

Returns
DESCRIPTION

Return type
TYPE

estimate_dimensionality(*skew_threshold=5*, *eccentricity_threshold=0.1*)

Estimate dimensionality of the impedance tensor from parameters such as strike and phase tensor eccentricity

Returns
DESCRIPTION

Return type
TYPE

estimate_distortion(*n_frequencies=None*, *comp='det'*, *only_2d=False*)

Parameters

- **n_frequencies** (*TYPE, optional*) – DESCRIPTION, defaults to 20
- **comp** (*TYPE, optional*) – DESCRIPTION, defaults to "det"

:param : DESCRIPTION :type : TYPE :return: DESCRIPTION :rtype: TYPE

property invariants

Weaver Invariants

property phase

phase of impedance

property phase_det

phase determinant

property phase_error

phase error of impedance

Uncertainty in phase (in degrees) is computed by defining a circle around the z vector in the complex plane. The uncertainty is the absolute angle between the vector to (x,y) and the vector between the origin and the tangent to the circle.

property phase_error_det

phase error determinant

property phase_error_xx

phase error of xx component

property phase_error_xy

phase error of xy component

property phase_error_yx

phase error of yx component

property phase_error_yy

phase error of yy component

property phase_model_error
phase model error of impedance

property phase_model_error_det
phase model error determinant

property phase_model_error_xx
phase model error of xx component

property phase_model_error_xy
phase model error of xy component

property phase_model_error_yx
phase model error of yx component

property phase_model_error_yy
phase model error of yy component

property phase_tensor
Phase tensor object based on impedance

property phase_xx
phase of xx component

property phase_xy
phase of xy component

property phase_yx
phase of yx component

property phase_yy
phase of yy component

remove_distortion(*distortion_tensor=None*, *distortion_error_tensor=None*, *n_frequencies=None*, *comp='det'*, *only_2d=False*, *inplace=False*)

Remove distortion D form an observed impedance tensor Z to obtain the uperturbed "correct" Z0: Z = D * Z0

Propagation of errors/uncertainties included

Parameters

- **distortion_tensor** (*np.ndarray(2, 2, dtype=real)*) – real distortion tensor as a 2x2
- **distortion_error_tensor** (*np.ndarray(2, 2, dtype=real),*) – default is None
- **inplace** (*boolean*) – Update the current object or return a new impedance

Returns
input distortion tensor

Return type
np.ndarray(2, 2, dtype='real')

Returns
impedance tensor with distorion removed

Return type
np.ndarray(num_frequency, 2, 2, dtype='complex')

Returns
impedance tensor error after distortion is removed

Return type
np.ndarray(num_frequency, 2, 2, dtype='complex')

Example

```
>>> distortion = np.array([[1.2, .5],[.35, 2.1]])
>>> d, new_z, new_z_error = z_obj.remove_distortion(distortion)
```

remove_ss(*reduce_res_factor_x=1.0*, *reduce_res_factor_y=1.0*, *inplace=False*)

Remove the static shift by providing the respective correction factors for the resistivity in the x and y components. (Factors can be determined by using the "Analysis" module for the impedance tensor)

Assume the original observed tensor Z is built by a static shift S and an unperturbated "correct" Z0 :

- Z = S * Z0

therefore the correct Z will be :

- Z0 = S^(-1) * Z

Parameters

- **reduce_res_factor_x** (*float or iterable list or array*) – static shift factor to be applied to x components (ie z[:, 0, :]). This is assumed to be in resistivity scale
- **reduce_res_factor_y** (*float or iterable list or array*) – static shift factor to be applied to y components (ie z[:, 1, :]). This is assumed to be in resistivity scale
- **inplace** (*boolean*) – Update the current object or return a new impedance

Returns
static shift matrix,

Return type
np.ndarray ((2, 2))

Returns
corrected Z if inplace is False

Return type
mtpy.core.z.Z

Note: The factors are in resistivity scale, so the entries of the matrix "S" need to be given by their square-roots!

---

property res_det

resistivity determinant

property res_error_det

resistivity error determinant

property res_error_xx

resistivity error of xx component

property res_error_xy
resistivity error of xy component

property res_error_yx
resistivity error of yx component

property res_error_yy
resistivity error of yy component

property res_model_error_det
resistivity model error determinant

property res_model_error_xx
resistivity model error of xx component

property res_model_error_xy
resistivity model error of xy component

property res_model_error_yx
resistivity model error of yx component

property res_model_error_yy
resistivity model error of yy component

property res_xx
resistivity of xx component

property res_xy
resistivity of xy component

property res_yx
resistivity of yx component

property res_yy
resistivity of yy component

property resistivity
resistivity of impedance

property resistivity_error
resistivity error of impedance

By standard error propagation, relative error in resistivity is 2*relative error in z amplitude.

property resistivity_model_error
resistivity model error of impedance

set_resistivity_phase(*resistivity*, *phase*, *frequency*, *res_error=None*, *phase_error=None*, *res_model_error=None*, *phase_model_error=None*)

Set values for resistivity (res - in Ohm m) and phase (phase - in degrees), including error propagation.

Parameters

- **resistivity** (*np.ndarray(num_frequency, 2, 2)*) – resistivity array in Ohm-m
- **phase** (*np.ndarray(num_frequency, 2, 2)*) – phase array in degrees
- **frequency** (*np.ndarray(num_frequency)*) – frequency array in Hz
- **res_error** (*np.ndarray(num_frequency, 2, 2)*) – resistivity error array in Ohm-m

- **phase_error** (*np.ndarray(num_frequency, 2, 2)*) – phase error array in degrees

Note: The error propogation is causal, meaning the apparent resistivity error and phase error are linked through a Taylor exampsion approximation where the phase error is estimated from the apparent resistivity error. Therefore if you set the phase error you will likely not get back the same phase error.

property z
Impedance tensor

np.ndarray(nfrequency, 2, 2)

property z_error
error of impedance tensor array as standard deviation

property z_model_error
model error of impedance tensor array as standard deviation

mtpy.imaging package

Subpackages

mtpy.imaging.mtplot_tools package

Submodules

mtpy.imaging.mtplot_tools.arrows module

Created on Sun Sep 25 15:16:31 2022

@author: jpeacock

class `mtpy.imaging.mtplot_tools.arrows.`**`MTArrows`**(***kwargs*)
Bases: `object`

Helper class to read a dictionary of arrow properties

Arguments:

- **'size'**
  [float] multiplier to scale the arrow. *default* is 5
- **'head_length'**
  [float] length of the arrow head *default* is 1.5
- **'head_width'**
  [float] width of the arrow head *default* is 1.5
- **'lw'**
  [float] line width of the arrow *default* is .5
- **'color'**
  [tuple (real, imaginary)] color of the arrows for real and imaginary

- **'threshold': float**
  threshold of which any arrow larger than this number will not be plotted, helps clean up if the data is not good. *default* is 1, note this is before scaling by 'size'
- **'direction**
  [[0 | 1]]
  - 0 for arrows to point toward a conductor
  - 1 for arrow to point away from conductor

mtpy.imaging.mtplot_tools.base module

Base classes for plotting classes

author
jpeacock

class mtpy.imaging.mtplot_tools.base.**PlotBase**(***kwargs*)

Bases: *PlotSettings*

base class for plotting objects

plot()

redraw_plot()

use this function if you updated some attributes and want to re-plot.

Example

```
>>> # change the color and marker of the xy components
>>> p1.xy_color = (.5,.5,.9)
>>> p1.xy_marker = '*'
>>> p1.redraw_plot()
```

save_plot(*save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot=True*)

save_plot will save the figure to save_fn.

Arguments:

save_fn
[string] full path to save figure to, can be input as * directory path -> the directory path to save to

in which the file will be saved as save_fn/station_name_ResPhase.file_format

- full path -> file will be save to the given path. If you use this option then the format will be assumed to be provided by the path

file_format
[[pdf | eps | jpg | png | svg]] file type of saved figure pdf,svg,eps...

orientation
[[landscape | portrait]] orientation in which the file will be saved *default* is portrait

fig_dpi
[int] The resolution in dots-per-inch the file will be saved. If None then the fig_dpi will be that at which the figure was made. I don't think that it can be larger than fig_dpi of the figure.

close_plot
[[true | false]]

- True will close the plot after saving.
- False will leave plot open

Example

```
>>> # to save plot as jpg
>>> p1.save_plot(r'/home/MT/figures', file_format='jpg')
```

update_plot()

update any parameters that where changed using the built-in draw from canvas.

Use this if you change an of the .fig or axes properties

Example

```
>>> [ax.grid(True, which='major') for ax in [p1.axr,p1.axp]]
>>> p1.update_plot()
```

class mtpy.imaging.mtplot_tools.base.**PlotBaseMaps**(***kwargs*)

Bases: *PlotBase*

Base object for plot classes that use map views, includes methods for interpolation.

add_raster(*ax*, *raster_fn*, *add_colorbar=True*, ***kwargs*)

Add a raster to an axis using rasterio

Parameters

- **ax** (*TYPE*) – DESCRIPTION
- **raster_fn** (*TYPE*) – DESCRIPTION
- **add_colorbar** (*TYPE, optional*) – DESCRIPTION, defaults to True
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

static **get_interp1d_functions_t**(*tf*, *interp_type='slinear'*)

Parameters
interp_type (*TYPE, optional*) – DESCRIPTION, defaults to "slinear"

Returns
DESCRIPTION

Return type
TYPE

static get_interp1d_functions_z(*tf*, *interp_type='slinear'*)

Parameters
`interp_type` (*TYPE, optional*) – DESCRIPTION, defaults to "slinear"

Returns
DESCRIPTION

Return type
TYPE

interpolate_to_map(*plot_array*, *component*)

interpolate points onto a 2d map.

Parameters

- **plot_array** (*TYPE*) – DESCRIPTION
- **component** (*TYPE*) – DESCRIPTION
- **cell_size** (*TYPE, optional*) – DESCRIPTION, defaults to 0.002
- **n_padding_cells** (*TYPE, optional*) – DESCRIPTION, defaults to 10
- **interpolation_method** (*TYPE, optional*) – DESCRIPTION, defaults to "delaunay"

Returns
DESCRIPTION

Return type
TYPE

class mtpy.imaging.mtplot_tools.base.**PlotBaseProfile**(*tf_list*, ***kwargs*)

Bases: *PlotBase*

Base object for profile plots like pseudo sections.

property rotation_angle

mtpy.imaging.mtplot_tools.ellipses module

Created on Sun Sep 25 15:19:16 2022

@author: jpeacock

class mtpy.imaging.mtplot_tools.ellipses.**MTEllipse**(***kwargs*)

Bases: `object`

helper class for getting ellipse properties from an input dictionary

Arguments:

- **'size' -> size of ellipse in points**
  default is .25
- **'colorby'**
  [['phimin' | 'phimax' | 'beta' |] 'skew_seg' | 'phidet' | 'ellipticity']
  - 'phimin' -> colors by minimum phase
  - 'phimax' -> colors by maximum phase

- 'skew' -> colors by skew
- **'skew_seg' -> colors by skew in**
  discrete segments defined by the range
- **'normalized_skew' -> colors by**
  normalized_skew see Booker, 2014
- **'normalized_skew_seg' -> colors by**
  normalized_skew discrete segments defined by the range
- **'phidet' -> colors by determinant of**
  the phase tensor
- 'ellipticity' -> colors by ellipticity

default is 'phimin'

- **'range'**
  [tuple (min, max, step)] Need to input at least the min and max and if using 'skew_seg' to plot discrete values input step as well *default* depends on 'colorby'
- **'cmap'**
  [['mt_yl2rd' | 'mt_bl2yl2rd' |]

  'mt_wh2bl' | 'mt_rd2bl' | 'mt_bl2wh2rd' | 'mt_seg_bl2wh2rd' | 'mt_rd2gr2bl']

  - 'mt_yl2rd' –> yellow to red
  - 'mt_bl2yl2rd' –> blue to yellow to red
  - 'mt_wh2bl' –> white to blue
  - 'mt_rd2bl' –> red to blue
  - 'mt_bl2wh2rd' –> blue to white to red
  - 'mt_bl2gr2rd' –> blue to green to red
  - 'mt_rd2gr2bl' –> red to green to blue
  - **'mt_seg_bl2wh2rd' –> discrete blue to**
    white to red

`property ellipse_cmap_bounds`

`property ellipse_cmap_n_segments`

`property ellipse_properties`

`get_color_map()`

get a color map

`get_pt_color_array(pt_object)`

Get the appropriat color by array

`get_range()`

get an appropriate range for the colorby

mtpy.imaging.mtplot_tools.map_interpolation_tools module

Created on Sun Sep 25 15:06:58 2022

@author: jpeacock

`mtpy.imaging.mtplot_tools.map_interpolation_tools.`**`get_plot_xy`**(*plot_array*, *cell_size*, *n_padding_cells*)

Get plot x and plot y from a plot array to interpolate on to

Parameters

- **plot_array** (*TYPE*) – DESCRIPTION
- **cell_size** (*TYPE*) – DESCRIPTION
- **n_padding_cells** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`mtpy.imaging.mtplot_tools.map_interpolation_tools.`**`griddata_interpolate`**(*x*, *y*, *values*, *new_x*, *new_y*, *interpolation_method='cubic'*)

Parameters

- **x** (*TYPE*) – DESCRIPTION
- **y** (*TYPE*) – DESCRIPTION
- **value** (*TYPE*) – DESCRIPTION
- **new_x** (*TYPE*) – DESCRIPTION
- **new_y** (*TYPE*) – DESCRIPTION
- **interpolation_method** (*TYPE, optional*) – DESCRIPTION, defaults to “cubic”

Returns
DESCRIPTION

Return type
TYPE

`mtpy.imaging.mtplot_tools.map_interpolation_tools.`**`in_hull`**(*p*, *hull*)

Test if points in p are within the convex hull

`mtpy.imaging.mtplot_tools.map_interpolation_tools.`**`interpolate_to_map`**(*plot_array*, *component*, *cell_size=0.002*, *n_padding_cells=10*, *interpolation_method='delaunay'*, *interpolation_power=5*, *nearest_neighbors=7*)

Parameters

- **plot_array** (*TYPE*) – DESCRIPTION
- **component** (*TYPE*) – DESCRIPTION

- **cell_size** (*TYPE, optional*) – DESCRIPTION, defaults to .002
- **n_padding_cells** (*TYPE, optional*) – DESCRIPTION, defaults to 10
- **interpolation_method** (*TYPE, optional*) – DESCRIPTION, defaults to "delaunay"

Returns
DESCRIPTION

Return type
TYPE

mtpy.imaging.mtplot_tools.map_interpolation_tools.**interpolate_to_map_griddata**(*plot_array*, *component*, *cell_size=0.002*, *n_padding_cells=10*, *interpolation_method='cubic'*)

Interpolate using scipy.interpolate.griddata

Parameters

- **plot_array** (*TYPE*) – DESCRIPTION
- **component** (*TYPE*) – DESCRIPTION
- **cell_size** (*TYPE, optional*) – DESCRIPTION, defaults to .002
- **n_padding_cells** (*TYPE, optional*) – DESCRIPTION, defaults to 10

Returns
DESCRIPTION

Return type
TYPE

mtpy.imaging.mtplot_tools.map_interpolation_tools.**interpolate_to_map_triangulate**(*plot_array*, *component*, *cell_size=0.002*, *n_padding_cells=10*, *nearest_neighbors=7*, *interp_pow=4*)

plot_array must have key words:

- **latitude**: latitude in decimal degrees of measured points
- **longitude**: longitude in decimal degrees of measured points
- values should have the proper name of the input component. For example if you are plotting the resistivity of the xy component then the keyword should be 'res_xy'

Parameters

- **plot_x** (*np.ndarray*) – desired regular x locations to interpolate to
- **plot_y** (*np.ndarray*) – desired regular x locations to interpolate to
- **plot_array** (*np.ndarray*) – structured array, see above
- **component** (*string*) – component or keyword of the plot_array to plot

- **cell_size** (*TYPE, optional*) – size of cells , defaults to 0.002
- **n_padding_cells** (*TYPE, optional*) – DESCRIPTION, defaults to 10
- **nearest_neighbors** (*TYPE, optional*) – DESCRIPTION, defaults to 7
- **interp_pow** (*TYPE, optional*) – DESCRIPTION, defaults to 4

Returns
DESCRIPTION

Return type
TYPE

mtpy.imaging.mtplot_tools.map_interpolation_tools.**triangulate_interpolation**(*x*, *y*, *values*, *padded_x*, *padded_y*, *new_x*, *new_y*, *nearest_neighbors=7*, *interp_pow=4*)

Parameters

- **x** (*TYPE*) – DESCRIPTION
- **y** (*TYPE*) – DESCRIPTION
- **values** (*TYPE*) – DESCRIPTION
- **new_x** (*TYPE*) – DESCRIPTION
- **new_y** (*TYPE*) – DESCRIPTION

:param : DESCRIPTION :type : TYPE :return: DESCRIPTION :rtype: TYPE

mtpy.imaging.mtplot_tools.plot_settings module

Created on Sun Sep 25 15:20:43 2022

@author: jpeacock

class mtpy.imaging.mtplot_tools.plot_settings.**PlotSettings**(***kwargs*)

Bases: *MTArrows*, *MTEllipse*

Hold all the plot settings that one might need

property arrow_imag_properties

property arrow_real_properties

property det_error_bar_properties

xy error bar properties for xy mode :return: DESCRIPTION :rtype: TYPE

property font_dict

make_pt_cb(*ax*)

property period_label_dict

log 10 labels

Returns
DESCRIPTION

Return type
TYPE

set_period_limits(*period*)

set period limits

Returns
DESCRIPTION

Return type
TYPE

set_phase_limits(*phase*, *mode='od'*)

set_resistivity_limits(*resistivity*, *mode='od'*, *scale='log'*)

set resistivity limits

Parameters

- **resistivity** (*TYPE*) – DESCRIPTION
- **mode** (*TYPE, optional*) – DESCRIPTION, defaults to "od"

Returns
DESCRIPTION

Return type
TYPE

property text_dict

property xy_error_bar_properties

xy error bar properties for xy mode :return: DESCRIPTION :rtype: TYPE

property yx_error_bar_properties

xy error bar properties for xy mode :return: DESCRIPTION :rtype: TYPE

mtpy.imaging.mtplot_tools.plotters module

Simple plotters elements that can be assembled in various plotting classes

Created on Sun Sep 25 15:27:28 2022

author
jpeacock

mtpy.imaging.mtplot_tools.plotters.**add_raster**(*ax*, *raster_fn*, *add_colorbar=True*, ***kwargs*)

Add a raster to an axis using rasterio

Parameters

- **raster_fn** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`mtpy.imaging.mtplot_tools.plotters.plot_errorbar(ax, x_array, y_array, y_error=None, x_error=None, **kwargs)`

convinience function to make an error bar instance

Arguments:

ax
[matplotlib.axes instance] axes to put error bar plot on

x_array
[np.ndarray(nx)] array of x values to plot

y_array
[np.ndarray(nx)] array of y values to plot

y_error
[np.ndarray(nx)] array of errors in y-direction to plot

x_error
[np.ndarray(ns)] array of error in x-direction to plot

color
[string or (r, g, b)] color of marker, line and error bar

marker
[string] marker type to plot data as

mew
[string] marker edgewidth

ms
[float] size of marker

ls
[string] line style between markers

lw
[float] width of line between markers

e_capsize
[float] size of error bar cap

e_capthick
[float] thickness of error bar cap

picker
[float] radius in points to be able to pick a point.

Returns:

errorbar_object
[matplotlib.Axes.errorbar] error bar object containing line data, errorbars, etc.

`mtpy.imaging.mtplot_tools.plotters.`**`plot_phase`**`(`*ax*, *period*, *phase*, *error*, *yx=False*, ***properties*`)`

plot apparent resistivity to the given axis with given properties

Parameters

- **ax** (*TYPE*) – DESCRIPTION
- **resistivity** (*TYPE*) – DESCRIPTION
- **period** (*TYPE*) – DESCRIPTION
- ****properties** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`mtpy.imaging.mtplot_tools.plotters.`**`plot_pt_lateral`**`(`*ax*, *pt_obj*, *color_array*, *ellipse_properties*, *y_shift=0*, *fig=None*, *edge_color=None*, *n_index=0*`)`

Parameters

- **ax** (*TYPE*) – DESCRIPTION
- **pt_obj** (*TYPE*) – DESCRIPTION
- **color_array** (*TYPE*) – DESCRIPTION
- **ellipse_properties** (*TYPE*) – DESCRIPTION
- **bounds** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

`mtpy.imaging.mtplot_tools.plotters.`**`plot_resistivity`**`(`*ax*, *period*, *resistivity*, *error*, ***properties*`)`

plot apparent resistivity to the given axis with given properties

Parameters

- **ax** (*TYPE*) – DESCRIPTION
- **resistivity** (*TYPE*) – DESCRIPTION
- **period** (*TYPE*) – DESCRIPTION
- ****properties** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`mtpy.imaging.mtplot_tools.plotters.`**`plot_tipper_lateral`**(*axt*, *t_obj*, *plot_tipper*, *real_properties*, *imag_properties*, *font_size=6*, *legend=True*, *zero_reference=False*)

Parameters

- **axt** (*TYPE*) – DESCRIPTION
- **t_obj** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

mtpy.imaging.mtplot_tools.utils module

Utility functions for plotting

Created on Sun Sep 25 15:49:01 2022

author
jpeacock

`mtpy.imaging.mtplot_tools.utils.`**`add_colorbar_axis`**(*ax*, *fig*)

`mtpy.imaging.mtplot_tools.utils.`**`get_log_tick_labels`**(*ax*, *spacing=1*)

Parameters
ax (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`mtpy.imaging.mtplot_tools.utils.`**`get_period_limits`**(*period*)

`mtpy.imaging.mtplot_tools.utils.`**`make_color_list`**(*cbax*, *nseg*, *ckmin*, *ckmax*, *ckstep*)

`mtpy.imaging.mtplot_tools.utils.`**`make_value_str`**(*value*, *value_list=None*, *spacing='{0:^8}'*, *value_format='{0: .2f}'*, *append=False*, *add=False*)

helper function for writing values to a file, takes in a value and either appends or adds value to value_list according to the spacing and format of the string.

Arguments:

value : float

value_list : list of values converted to strings

spacing
[spacing of the string that the value will be converted] to.

value_format
[format of the string that the value is being] coverted to.

append
[[True | False]] if True then appends the value to value list

add
[[True | False]] if True adds value string to the other value strings in value_list

Returns:

value_list
[the input value_list with the new value either] added or appended.

or

value_str : value string if add and append are false

`mtpy.imaging.mtplot_tools.utils.`**`round_to_step`**`(`*num*, *base=5*`)`

Module contents

class `mtpy.imaging.mtplot_tools.`**`MTArrows`**`(`***kwargs*`)`

Bases: `object`

Helper class to read a dictionary of arrow properties

Arguments:

- **'size'**
  [float] multiplier to scale the arrow. *default* is 5
- **'head_length'**
  [float] length of the arrow head *default* is 1.5
- **'head_width'**
  [float] width of the arrow head *default* is 1.5
- **'lw'**
  [float] line width of the arrow *default* is .5
- **'color'**
  [tuple (real, imaginary)] color of the arrows for real and imaginary
- **'threshold': float**
  threshold of which any arrow larger than this number will not be plotted, helps clean up if the data is not good. *default* is 1, note this is before scaling by 'size'
- **'direction**
  [[0 | 1]]
  - 0 for arrows to point toward a conductor
  - 1 for arrow to point away from conductor

class `mtpy.imaging.mtplot_tools.`**`MTEllipse`**`(`***kwargs*`)`

Bases: `object`

helper class for getting ellipse properties from an input dictionary

Arguments:

- **'size' -> size of ellipse in points**
  default is .25
- **'colorby'**
  [['phimin' | 'phimax' | 'beta' |] 'skew_seg' | 'phidet' | 'ellipticity']
  - 'phimin' -> colors by minimum phase
  - 'phimax' -> colors by maximum phase
  - 'skew' -> colors by skew
  - **'skew_seg' -> colors by skew in**
    discrete segments defined by the range
  - **'normalized_skew' -> colors by**
    normalized_skew see Booker, 2014
  - **'normalized_skew_seg' -> colors by**
    normalized_skew discrete segments defined by the range
  - **'phidet' -> colors by determinant of**
    the phase tensor
  - 'ellipticity' -> colors by ellipticity

  default is 'phimin'
- **'range'**
  [tuple (min, max, step)] Need to input at least the min and max and if using 'skew_seg' to plot discrete values input step as well *default* depends on 'colorby'
- **'cmap'**
  [['mt_yl2rd' | 'mt_bl2yl2rd' |]

    'mt_wh2bl' | 'mt_rd2bl' | 'mt_bl2wh2rd' | 'mt_seg_bl2wh2rd' | 'mt_rd2gr2bl']

  - 'mt_yl2rd' –> yellow to red
  - 'mt_bl2yl2rd' –> blue to yellow to red
  - 'mt_wh2bl' –> white to blue
  - 'mt_rd2bl' –> red to blue
  - 'mt_bl2wh2rd' –> blue to white to red
  - 'mt_bl2gr2rd' –> blue to green to red
  - 'mt_rd2gr2bl' –> red to green to blue
  - **'mt_seg_bl2wh2rd' –> discrete blue to**
    white to red

`property ellipse_cmap_bounds`

`property ellipse_cmap_n_segments`

`property ellipse_properties`

`get_color_map()`

get a color map

get_pt_color_array(*pt_object*)
Get the appropriat color by array

get_range()
get an appropriate range for the colorby

class mtpy.imaging.mtplot_tools.**PlotBase**(***kwargs*)
Bases: *PlotSettings*

base class for plotting objects

plot()

redraw_plot()
use this function if you updated some attributes and want to re-plot.

Example

```
>>> # change the color and marker of the xy components
>>> p1.xy_color = (.5,.5,.9)
>>> p1.xy_marker = '*'
>>> p1.redraw_plot()
```

save_plot(*save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot=True*)
save_plot will save the figure to save_fn.

Arguments:

save_fn
[string] full path to save figure to, can be input as * directory path -> the directory path to save to

in which the file will be saved as save_fn/station_name_ResPhase.file_format

- full path -> file will be save to the given path. If you use this option then the format will be assumed to be provided by the path

file_format
[[pdf | eps | jpg | png | svg]] file type of saved figure pdf,svg,eps…

orientation
[[landscape | portrait]] orientation in which the file will be saved *default* is portrait

fig_dpi
[int] The resolution in dots-per-inch the file will be saved. If None then the fig_dpi will be that at which the figure was made. I don't think that it can be larger than fig_dpi of the figure.

close_plot
[[true | false]]

- True will close the plot after saving.
- False will leave plot open

Example

```
>>> # to save plot as jpg
>>> p1.save_plot(r'/home/MT/figures', file_format='jpg')
```

update_plot()

update any parameters that where changed using the built-in draw from canvas.

Use this if you change an of the .fig or axes properties

Example

```
>>> [ax.grid(True, which='major') for ax in [p1.axr,p1.axp]]
>>> p1.update_plot()
```

class mtpy.imaging.mtplot_tools.**PlotBaseMaps**(***kwargs*)

Bases: *PlotBase*

Base object for plot classes that use map views, includes methods for interpolation.

add_raster(*ax*, *raster_fn*, *add_colorbar=True*, ***kwargs*)

Add a raster to an axis using rasterio

Parameters

- **ax** (*TYPE*) – DESCRIPTION
- **raster_fn** (*TYPE*) – DESCRIPTION
- **add_colorbar** (*TYPE, optional*) – DESCRIPTION, defaults to True
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

static **get_interp1d_functions_t**(*tf*, *interp_type='slinear'*)

Parameters
interp_type (*TYPE, optional*) – DESCRIPTION, defaults to "slinear"

Returns
DESCRIPTION

Return type
TYPE

static **get_interp1d_functions_z**(*tf*, *interp_type='slinear'*)

Parameters
interp_type (*TYPE, optional*) – DESCRIPTION, defaults to "slinear"

Returns
DESCRIPTION

Return type
TYPE

interpolate_to_map(*plot_array*, *component*)

interpolate points onto a 2d map.

Parameters

- **plot_array** (*TYPE*) – DESCRIPTION
- **component** (*TYPE*) – DESCRIPTION

- **cell_size** (*TYPE, optional*) – DESCRIPTION, defaults to 0.002
- **n_padding_cells** (*TYPE, optional*) – DESCRIPTION, defaults to 10
- **interpolation_method** (*TYPE, optional*) – DESCRIPTION, defaults to "delaunay"

Returns
DESCRIPTION

Return type
TYPE

class mtpy.imaging.mtplot_tools.**PlotBaseProfile**(*tf_list*, ***kwargs*)

Bases: *PlotBase*

Base object for profile plots like pseudo sections.

property rotation_angle

class mtpy.imaging.mtplot_tools.**PlotSettings**(***kwargs*)

Bases: *MTArrows*, *MTEllipse*

Hold all the plot settings that one might need

property arrow_imag_properties

property arrow_real_properties

property det_error_bar_properties

xy error bar properties for xy mode :return: DESCRIPTION :rtype: TYPE

property font_dict

make_pt_cb(*ax*)

property period_label_dict

log 10 labels

Returns
DESCRIPTION

Return type
TYPE

set_period_limits(*period*)

set period limits

Returns
DESCRIPTION

Return type
TYPE

set_phase_limits(*phase*, *mode='od'*)

set_resistivity_limits(*resistivity*, *mode='od'*, *scale='log'*)

set resistivity limits

Parameters

- **resistivity** (*TYPE*) – DESCRIPTION
- **mode** (*TYPE, optional*) – DESCRIPTION, defaults to "od"

Returns
DESCRIPTION

Return type
TYPE

property text_dict

property xy_error_bar_properties

xy error bar properties for xy mode :return: DESCRIPTION :rtype: TYPE

property yx_error_bar_properties

xy error bar properties for xy mode :return: DESCRIPTION :rtype: TYPE

mtpy.imaging.mtplot_tools.**add_raster**(*ax*, *raster_fn*, *add_colorbar=True*, ***kwargs*)

Add a raster to an axis using rasterio

Parameters

- **raster_fn** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

mtpy.imaging.mtplot_tools.**get_log_tick_labels**(*ax*, *spacing=1*)

Parameters
ax (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

mtpy.imaging.mtplot_tools.**griddata_interpolate**(*x*, *y*, *values*, *new_x*, *new_y*, *interpolation_method='cubic'*)

Parameters

- **x** (*TYPE*) – DESCRIPTION
- **y** (*TYPE*) – DESCRIPTION
- **value** (*TYPE*) – DESCRIPTION
- **new_x** (*TYPE*) – DESCRIPTION
- **new_y** (*TYPE*) – DESCRIPTION
- **interpolation_method** (*TYPE, optional*) – DESCRIPTION, defaults to "cubic"

Returns
DESCRIPTION

Return type
TYPE

`mtpy.imaging.mtplot_tools.`**`interpolate_to_map`**(*plot_array*, *component*, *cell_size=0.002*, *n_padding_cells=10*, *interpolation_method='delaunay'*, *interpolation_power=5*, *nearest_neighbors=7*)

Parameters

- **plot_array** (*TYPE*) – DESCRIPTION
- **component** (*TYPE*) – DESCRIPTION
- **cell_size** (*TYPE, optional*) – DESCRIPTION, defaults to .002
- **n_padding_cells** (*TYPE, optional*) – DESCRIPTION, defaults to 10
- **interpolation_method** (*TYPE, optional*) – DESCRIPTION, defaults to "delaunay"

Returns
DESCRIPTION

Return type
TYPE

`mtpy.imaging.mtplot_tools.`**`plot_errorbar`**(*ax*, *x_array*, *y_array*, *y_error=None*, *x_error=None*, ***kwargs*)

convinience function to make an error bar instance

Arguments:

ax
[matplotlib.axes instance] axes to put error bar plot on

x_array
[np.ndarray(nx)] array of x values to plot

y_array
[np.ndarray(nx)] array of y values to plot

y_error
[np.ndarray(nx)] array of errors in y-direction to plot

x_error
[np.ndarray(ns)] array of error in x-direction to plot

color
[string or (r, g, b)] color of marker, line and error bar

marker
[string] marker type to plot data as

mew
[string] marker edgewidth

ms
[float] size of marker

ls
[string] line style between markers

lw
[float] width of line between markers

e_capsize
[float] size of error bar cap

e_capthick
[float] thickness of error bar cap

picker
[float] radius in points to be able to pick a point.

Returns:

errorbar_object
[matplotlib.Axes.errorbar] error bar object containing line data, errorbars, etc.

`mtpy.imaging.mtplot_tools.plot_phase`(*ax*, *period*, *phase*, *error*, *yx=False*, ***properties*)

plot apparent resistivity to the given axis with given properties

Parameters

- **ax** (*TYPE*) – DESCRIPTION
- **resistivity** (*TYPE*) – DESCRIPTION
- **period** (*TYPE*) – DESCRIPTION
- ****properties** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`mtpy.imaging.mtplot_tools.plot_pt_lateral`(*ax*, *pt_obj*, *color_array*, *ellipse_properties*, *y_shift=0*, *fig=None*, *edge_color=None*, *n_index=0*)

Parameters

- **ax** (*TYPE*) – DESCRIPTION
- **pt_obj** (*TYPE*) – DESCRIPTION
- **color_array** (*TYPE*) – DESCRIPTION
- **ellipse_properties** (*TYPE*) – DESCRIPTION
- **bounds** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

`mtpy.imaging.mtplot_tools.plot_resistivity`(*ax*, *period*, *resistivity*, *error*, ***properties*)

plot apparent resistivity to the given axis with given properties

Parameters

- **ax** (*TYPE*) – DESCRIPTION
- **resistivity** (*TYPE*) – DESCRIPTION
- **period** (*TYPE*) – DESCRIPTION
- ****properties** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

mtpy.imaging.mtplot_tools.**plot_tipper_lateral**(*axt*, *t_obj*, *plot_tipper*, *real_properties*, *imag_properties*, *font_size=6*, *legend=True*, *zero_reference=False*)

Parameters

- **axt** (*TYPE*) – DESCRIPTION
- **t_obj** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

mtpy.imaging.mtplot_tools.**triangulate_interpolation**(*x*, *y*, *values*, *padded_x*, *padded_y*, *new_x*, *new_y*, *nearest_neighbors=7*, *interp_pow=4*)

Parameters

- **x** (*TYPE*) – DESCRIPTION
- **y** (*TYPE*) – DESCRIPTION
- **values** (*TYPE*) – DESCRIPTION
- **new_x** (*TYPE*) – DESCRIPTION
- **new_y** (*TYPE*) – DESCRIPTION

:param : DESCRIPTION :type : TYPE :return: DESCRIPTION :rtype: TYPE

Submodules

mtpy.imaging.mtcolors module

Created on Tue May 14 18:05:59 2013

@author: jpeacock-pr

class mtpy.imaging.mtcolors.**FixPointNormalize**(*vmin=None*, *vmax=None*, *sealevel=0*, *col_val=0.21875*, *clip=False*)

Bases: `Normalize`

Inspired by https://stackoverflow.com/questions/20144529/shifted-colorbar-matplotlib Subclassing Normalize to obtain a colormap with a fixpoint somewhere in the middle of the colormap. This may be useful for a *terrain* map, to set the "sea level" to a color in the blue/turquise range.

mtpy.imaging.mtcolors.**cmap_discretize**(*cmap*, *N*)

Return a discrete colormap from the continuous colormap cmap.

cmap: colormap instance, eg. cm.jet. N: number of colors.

Example
x = resize(arange(100), (5,100)) djet = cmap_discretize(cm.jet, 5) imshow(x, cmap=djet)

`mtpy.imaging.mtcolors.get_color(cvar, cmap)`

gets the color to plot for the given color map

`mtpy.imaging.mtcolors.get_plot_color(colorx, comp, cmap, ckmin=None, ckmax=None, bounds=None)`

gets the color for the given compnent, color array and cmap

Note: we now use the linearSegmentedColorMap objects, instead of the get_color function

`mtpy.imaging.mtcolors.register_cmaps(cmap_dict)`

mtpy.imaging.plot_mt_response module

plot_mt_response

Plots the resistivity and phase for different modes and components

Created 2017

@author: jpeacock

`class mtpy.imaging.plot_mt_response.PlotMTResponse(z_object=None, t_object=None, pt_obj=None, station='MT Response', **kwargs)`

Bases: `PlotBase`

Plots Resistivity and phase for the different modes of the MT response. At the moment it supports the input of an .edi file. Other formats that will be supported are the impedance tensor and errors with an array of periods and .j format.

The normal use is to input an .edi file, however it would seem that not everyone uses this format, so you can input the data and put it into arrays or objects like class mtpy.core.z.Z. Or if the data is in resistivity and phase format they can be input as arrays or a class mtpy.imaging.mtplot.ResPhase. Or you can put it into a class mtpy.imaging.mtplot.MTplot.

The plot places the apparent resistivity in log scale in the top panel(s), depending on the plot_num. The phase is below this, note that 180 degrees has been added to the yx phase so the xy and yx phases plot in the same quadrant. Both the resistivity and phase share the same x-axis which is in log period, short periods on the left to long periods on the right. So if you zoom in on the plot both plots will zoom in to the same x-coordinates. If there is tipper information, you can plot the tipper as a third panel at the bottom, and also shares the x-axis. The arrows are in the convention of pointing towards a conductor. The xx and yy components can be plotted as well, this adds two panels on the right. Here the phase is left unwrapped. Other parameters can be added as subplots such as strike, skew and phase tensor ellipses.

To manipulate the plot you can change any of the attributes listed below and call redraw_plot(). If you know more aout matplotlib and want to change axes parameters, that can be done by changing the parameters in the axes attributes and then call update_plot(), note the plot must be open.

`property period`

plot period

`plot()`

plotResPhase(filename,fig_num) will plot the apparent resistivity and phase for a single station.

`property plot_model_error`

`property rotation_angle`

mtpy.imaging.plot_mt_responses module

plots multiple MT responses simultaneously

Created on Thu May 30 17:02:39 2013 @author: jpeacock-pr

YG: the code there is massey, todo may need to rewrite it sometime

class mtpy.imaging.plot_mt_responses.**PlotMultipleResponses**(*mt_data*, ***kwargs*)

Bases: *PlotBase*

plots multiple MT responses simultaneously either in single plots or in one plot of sub-figures or in a single plot with subfigures for each component.

Arguments:

fn_list
[list of filenames to plot] ie. [fn_1, fn_2, . . .], *default* is None

plot_num
[[1 | 2 | 3]]

- 1 for just Ex/By and Ey/Bx *default*
- 2 for all 4 components
- 3 for off diagonal plus the determinant

plot_style
[['1' | 'all' | 'compare']]

determines the plotting style:

- **'1' for plotting each station in a different**
  figure. *default*
- **'all' for plotting each station in a subplot**
  all in the same figure
- **'compare' for comparing the responses all in**
  one plot. Here the responses are colored from dark to light. This plot can get messy if too many stations are plotted.

plot()

plot the apparent resistivity and phase

property plot_model_error

property rotation_angle

mtpy.imaging.plot_penetration_depth_1d module

Created on Thu Sep 22 10:58:58 2022

@author: jpeacock

class mtpy.imaging.plot_penetration_depth_1d.**PlotPenetrationDepth1D**(*tf*, ***kwargs*)

Bases: *PlotBase*

Plot the depth of penetration based on the Niblett-Bostick approximation.

property depth_units

plot()

plot the depth of investigation as a 1d plot with period on the y-axis and depth on the x axis

Returns
DESCRIPTION

Return type
TYPE

mtpy.imaging.plot_penetration_depth_map module

Created on Thu Sep 22 10:58:58 2022

@author: jpeacock

class mtpy.imaging.plot_penetration_depth_map.**PlotPenetrationDepthMap**(*mt_data*, ***kwargs*)

Bases: *PlotBaseMaps*

Plot the depth of penetration based on the Niblett-Bostick approximation.

property depth_units

plot()

plot the depth of investigation as a 1d plot with period on the y-axis and depth on the x axis

Returns
DESCRIPTION

Return type
TYPE

mtpy.imaging.plot_phase_tensor_maps module

Plot phase tensor map in Lat-Lon Coordinate System

Revision History:

Created by @author: jpeacock-pr on Thu May 30 18:20:04 2013

Modified by Fei.Zhang@ga.gov.au 2017-03:

brenainn.moushall 26-03-2020 15:07:14 AEDT:
Add plotting of geotiff as basemap background.

Updated 2022 by J. Peacock to work with v2

- Using rasterio to plot geotiffs

- factorized
- using interp function for faster plotting.

class mtpy.imaging.plot_phase_tensor_maps.**PlotPhaseTensorMaps**(*mt_data*, ***kwargs*)

Bases: *PlotBaseMaps*

Plots phase tensor ellipses in map view from a list of edi files

property map_scale

plot(*fig=None*, *save_path=None*, *show=True*, *raster_file=None*, *raster_kwargs={}*)

Plots the phase tensor map. :param fig: optional figure object :param save_path: path to folder for saving plots :param show: show plots if True :param raster_dict: dictionary containing information for a raster

to be plotted below the phase tensors.

property rotation_angle

property skew_cmap_bounds

skew bounds for a segmented colorbar

mtpy.imaging.plot_phase_tensor_pseudosection module

Created on Thu May 30 18:10:55 2013

@author: jpeacock-pr

class mtpy.imaging.plot_phase_tensor_pseudosection.**PlotPhaseTensorPseudoSection**(*mt_data*, ***kwargs*)

Bases: *PlotBaseProfile*

PlotPhaseTensorPseudoSection will plot the phase tensor ellipses in a pseudo section format

To get a list of .edi files that you want to plot –> :Example:

```
>>> import mtpy.imaging.mtplot as mtplot
>>> import os
>>> edipath = r"/home/EDIfiles"
>>> edilist = [os.path.join(edipath,edi) for edi in os.listdir(edipath)
>>> ...        if edi.find('.edi')>0]
```

- If you want to plot minimum phase colored from blue to red in a range of

  20 to 70 degrees you can do it one of two ways–>

1) :Example:

```
>>> edict = {'range':(20,70), 'cmap':'mt_bl2gr2rd','colorby':'phimin'}
>>> pt1 = mtplot.plot_pt_pseudosection(fn_list=edilist,
                                        ellipse_dict=edict)
```

2) :Example:

```
>>> pt1 = mtplot.plot_pt_pseudosection(fn_list=edilist, plot_yn='n')
>>> pt1.ellipse_colorby = 'phimin'
>>> pt1.ellipse_cmap = 'mt_bl2gr2rd'
>>> pt1.ellipse_range = (20,70)
>>> pt1.plot()
```

- If you want to add real induction arrows that are scaled by 10 and point

  away from a conductor –>

  Example

  ```
  >>> pt1.plot_tipper = 'yr'
  >>> pt1.arrow_size = 10
  >>> pt1.arrow_direction = -1
  >>> pt1.redraw_plot()
  ```

- If you want to save the plot as a pdf with a generic name –>

  Example
  :: >>> pt1.save_figure(r"/home/PTFigures", file_format='pdf', dpi=300) File saved to '/home/PTFigures/PTPseudoSection.pdf'

plot()

plots the phase tensor pseudo section. See class doc string for more details.

mtpy.imaging.plot_pseudosection module

Created on Thu May 30 18:39:58 2013

@author: jpeacock-pr

class mtpy.imaging.plot_pseudosection.**PlotResPhasePseudoSection**(*mt_data*, ***kwargs*)

Bases: *PlotBaseProfile*

plot a resistivity and phase pseudo section for different components

Need to input one of the following lists:

plot()

Returns
DESCRIPTION

Return type
TYPE

mtpy.imaging.plot_pt module

Created on Thu May 30 17:07:50 2013

@author: jpeacock

class mtpy.imaging.plot_pt.**PlotPhaseTensor**(*pt_object*, *station=None*, ***kwargs*)

Bases: *PlotBase*

Will plot phase tensor, strike angle, min and max phase angle, azimuth, skew, and ellipticity as subplots on one plot. It can plot the resistivity tensor along side the phase tensor for comparison.

plot(*rotation_angle=None*)

plots the phase tensor elements

property rotation_angle

mtpy.imaging.plot_residual_pt_maps module

PlotResidualPhaseTensor

*plots the residual phase tensor for two different sets of measurments

Created on Wed Oct 16 14:56:04 2013

@author: jpeacock-pr

class mtpy.imaging.plot_residual_pt_maps.**PlotResidualPTMaps**(*mt_data_01*, *mt_data_02*, *frequencies=array([1.00000000e-03, 1.42510267e-03, 2.03091762e-03, 2.89426612e-03, 4.12462638e-03, 5.87801607e-03, 8.37677640e-03, 1.19377664e-02, 1.70125428e-02, 2.42446202e-02, 3.45510729e-02, 4.92388263e-02, 7.01703829e-02, 1.00000000e-01, 1.42510267e-01, 2.03091762e-01, 2.89426612e-01, 4.12462638e-01, 5.87801607e-01, 8.37677640e-01, 1.19377664e+00, 1.70125428e+00, 2.42446202e+00, 3.45510729e+00, 4.92388263e+00, 7.01703829e+00, 1.00000000e+01, 1.42510267e+01, 2.03091762e+01, 2.89426612e+01, 4.12462638e+01, 5.87801607e+01, 8.37677640e+01, 1.19377664e+02, 1.70125428e+02, 2.42446202e+02, 3.45510729e+02, 4.92388263e+02, 7.01703829e+02, 1.00000000e+03])*, ***kwargs*)

Bases: *PlotBase*

This will plot residual phase tensors in a map for a single frequency. The data is read in and stored in 2 ways, one as a list ResidualPhaseTensor object for each matching station and the other in a structured array with all the important information. The structured array is the one that is used for plotting. It is computed each time plot() is called so if it is manipulated it is reset. The array is sorted by relative offset, so no special order of input is needed for the file names. However, the station names should be verbatim between surveys, otherwise it will not work.

The residual phase tensor is calculated as I-(Phi_2)^-1 (Phi_1)

The default coloring is by the geometric mean as sqrt(Phi_min*Phi_max), which defines the percent change between measurements.

There are a lot of parameters to change how the plot looks, have a look below if you figure looks a little funny. The most useful will be ellipse_size

The ellipses are normalized by the largest Phi_max of the survey.

Example

```
>>> import mtpy.imaging.mtplot as mtplot
>>> import os
>>> edipath1 = r"/home/EDIfiles1"
>>> edilist1 = [os.path.join(edipath1,edi) for edi in os.
↪listdir(edipath1)
>>> ...         if edi.find('.edi')>0]
>>> edipath2 = r"/home/EDIfiles2"
>>> edilist2 = [os.path.join(edipath2,edi) for edi in os.
↪listdir(edipath2)
>>> ...         if edi.find('.edi')>0]
>>> # color by phimin with a range of 0-5 deg
>>> ptmap = mtplot.plot_residual_pt_maps(edilist1, edilist2,␣
↪freqspot=10,
>>> ...                                   ellipse_dict={'size':1,
>>> ...                                                 'range':(0,5)})
>>>
>>> #
>>> #---add an image---
>>> ptmap.image_file = r"/home/Maps/Basemap.jpg"
>>> ptmap.image_extent = (0,10,0,10)
>>> ptmap.redraw_plot()
>>> #
>>> #---Save the plot---
>>> ptmap.save_plot(r"/home/EDIfiles",file_format='pdf')
>>> 'Saved figure to /home/EDIfile/PTMaps/PTmap_phimin_10.0_Hz.pdf'
```

Example

```
>>> #change the axis label and grid color
>>> ptmap.ax.set_xlabel('Latitude (deg)')
>>> ptmap.ax.grid(which='major', color=(.5,1,0))
>>> ptmap.update_plot()
```

Example

```
>>> # plot seismic hypocenters from a file
>>> lat, lon, depth = np.loadtxt(r"/home/seismic_hypocenter.txt")
>>> ptmap.ax.scatter(lon, lat, marker='o')
```

property map_scale

plot()

plot residual phase tensor

property rotation_angle

mtpy.imaging.plot_residual_pt_ps module

PlotResidualPhaseTensorPseudoSection

*plots the residual phase tensor for two different sets of measurments

Created on Wed Oct 16 14:56:04 2013

@author: jpeacock-pr

class mtpy.imaging.plot_residual_pt_ps.**PlotResidualPTPseudoSection**(*mt_data_01*, *mt_data_02*, *frequencies=array([1.00000000e-03, 1.42510267e-03, 2.03091762e-03, 2.89426612e-03, 4.12462638e-03, 5.87801607e-03, 8.37677640e-03, 1.19377664e-02, 1.70125428e-02, 2.42446202e-02, 3.45510729e-02, 4.92388263e-02, 7.01703829e-02, 1.00000000e-01, 1.42510267e-01, 2.03091762e-01, 2.89426612e-01, 4.12462638e-01, 5.87801607e-01, 8.37677640e-01, 1.19377664e+00, 1.70125428e+00, 2.42446202e+00, 3.45510729e+00, 4.92388263e+00, 7.01703829e+00, 1.00000000e+01, 1.42510267e+01, 2.03091762e+01, 2.89426612e+01, 4.12462638e+01, 5.87801607e+01, 8.37677640e+01, 1.19377664e+02, 1.70125428e+02, 2.42446202e+02, 3.45510729e+02, 4.92388263e+02, 7.01703829e+02, 1.00000000e+03])*, ***kwargs*)

Bases: `PlotBaseProfile`

This will plot residual phase tensors in a pseudo section. The data is read in and stored in 2 ways, one as a list ResidualPhaseTensor object for each matching station and the other in a structured array with all the important information. The structured array is the one that is used for plotting. It is computed each time plot() is called so if it is manipulated it is reset. The array is sorted by relative offset, so no special order of input is needed for the file names. However, the station names should be verbatim between surveys, otherwise it will not work.

The residual phase tensor is calculated as I-(Phi_2)^-1 (Phi_1)

The default coloring is by the geometric mean as sqrt(Phi_min*Phi_max), which defines the percent change between measurements.

There are a lot of parameters to change how the plot looks, have a look below if you figure looks a little funny.

The most useful will be xstretch, ystretch and ellipse_size

The ellipses are normalized by the largest Phi_max of the survey.

To get a list of .edi files that you want to plot –> :Example:

```
>>> import mtpy.imaging.mtplot as mtplot
>>> import os
>>> edipath1 = r"/home/EDIfiles1"
>>> edilist1 = [os.path.join(edipath1,edi) for edi in os.listdir(edipath1)
>>> ...         if edi.find('.edi')>0]
>>> edipath2 = r"/home/EDIfiles2"
>>> edilist2 = [os.path.join(edipath2,edi) for edi in os.listdir(edipath2)
>>> ...         if edi.find('.edi')>0]
>>> # color by phimin with a range of 0-5 deg
```

- If you want to plot geometric mean colored from white to orange in a range of 0 to 10 percent you can do it one of two ways–>

1) :Example:

```
>>> edict = {'range':(0,10), 'cmap':'mt_wh2or',                      'colorby':
↪'geometric_mean', 'size':10}
>>> pt1 = mtplot.residual_pt_ps(edilist1, edilst2, ellipse_dict=edict)
```

2) :Example:

```
>>> pt1 = mtplot.residual_pt_ps(edilist1, edilst2, ellipse_dict=edict,              ␣
↪                          plot_yn='n')
>>> pt1.ellipse_colorby = 'geometric_mean'
>>> pt1.ellipse_cmap = 'mt_wh2or'
>>> pt1.ellipse_range = (0, 10)
>>> pt1.ellipse_size = 10
>>> pt1.plot()
```

- If you want to save the plot as a pdf with a generic name –>

  Example

  :: >>> pt1.save_figure(r"/home/PTFigures", file_format='pdf', dpi=300) File saved to '/home/PTFigures/PTPseudoSection.pdf'

get_pt_color_array(*rpt_array*)

Get the appropriat color by array

plot()

plot residual phase tensor

property rotation_angle

mtpy.imaging.plot_resphase_maps module

Description:
Plots resistivity and phase maps for a given frequency

References:

CreationDate: 4/19/18 Developer: rakib.hassan@ga.gov.au

Revision History:

LastUpdate: 4/19/18 RH
2022-09 JP

`class mtpy.imaging.plot_resphase_maps.PlotResPhaseMaps(mt_data, **kwargs)`

Bases: `PlotBaseMaps`

Plots apparent resistivity and phase in map view from a list of edi files

Arguments:

fn_list
[list of strings] full paths to .edi files to plot

fig_size
[tuple or list (x, y) in inches] dimensions of the figure box in inches, this is a default unit of matplotlib. You can use this so make the plot fit the figure box to minimize spaces from the plot axes to the figure box. *default* is [8, 8]

mapscale
[['deg' | 'm' | 'km']] Scale of the map coordinates.

- 'deg' –> degrees in latitude and longitude
- 'm' –> meters for easting and northing
- 'km' –> kilometers for easting and northing

plot_yn
[['y' | 'n']] *'y' to plot on creating an instance

*'n' to not plot on creating an instance

title
[string] figure title

dpi
[int] dots per inch of the resolution. *default* is 300

font_size
[float] size of the font that labels the plot, 2 will be added to this number for the axis labels.

`property map_units`

`plot()`

mtpy.imaging.plot_spectrogram module

plotft

*PlotTF –> will plot a time frequency distribution of your choice

Created on Mon Aug 19 16:24:29 2013

@author: jpeacock

class `mtpy.imaging.plot_spectrogram.`**`PlotTF`**(*time_series*, *tf_type='smethod'*, ***kwargs*)

Bases: `object`

class to plot Time-Frequency

`plot()`

plot the time frequency distribution

`redraw_plot()`

use this function if you updated some attributes and want to re-plot.

Example

:: >>> tf1.plot_type = 'tf' >>> tf1.redraw_plot()

`save_figure`(*save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot='y'*)

save_plot will save the figure to save_fn.

Arguments:

save_fn

[string] full path to save figure to, can be input as * directory path -> the directory path to save to

in which the file will be saved as save_fn/TF_tftype.file_format

- full path -> file will be save to the given path. If you use this option then the format will be assumed to be provided by the path

file_format

[[pdf | eps | jpg | png | svg]] file type of saved figure pdf,svg,eps...

orientation

[[landscape | portrait]] orientation in which the file will be saved *default* is portrait

fig_dpi

[int] The resolution in dots-per-inch the file will be saved. If None then the dpi will be that at which the figure was made. I don't think that it can be larger than dpi of the figure.

close_plot

[[y | n]]

- 'y' will close the plot after saving.
- 'n' will leave plot open

Example

:: >>> # save plot as a jpg >>> tf1.save_plot(r'/home/MT/figures', file_format='jpg')

update_plot()

update any parameters that where changed using the built-in draw from canvas.

Use this if you change an of the .fig or axes properties

Example

```
>>> # to change the grid lines to be on the major ticks and gray
>>> tf1.ax.grid(True, which='major', color=(.5,.5,.5))
>>> tf1.update_plot()
```

mtpy.imaging.plot_stations module

PlotStations

Plots station locations in map view.

Created on Fri Jun 07 18:20:00 2013

@author: jpeacock-pr

class mtpy.imaging.plot_stations.**PlotStations**(*geo_df*, ***kwargs*)

Bases: *PlotBase*

plot station locations in map view.

Uses contextily to get the basemap. See https://contextily.readthedocs.io/en/latest/index.html for more information about options.

plot()

Parameters

- **cx_source** (*TYPE, optional*) – DESCRIPTION, defaults to cx.providers.USGS.USTopo
- **cx_zoom** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

mtpy.imaging.plot_strike module

Created on Thu May 30 18:28:24 2013

@author: jpeacock-pr

class mtpy.imaging.plot_strike.**PlotStrike**(*mt_data*, ***kwargs*)

Bases: *PlotBase*

PlotStrike will plot the strike estimated from the invariants, phase tensor and the tipper in either a rose diagram of xy plot

plots the strike angle as determined by phase tensor azimuth (Caldwell et al. [2004]) and invariants of the impedance tensor (Weaver et al. [2003]).

The data is split into decades where the histogram for each is plotted in the form of a rose diagram with a range of 0 to 180 degrees. Where 0 is North and 90 is East. The median angle of the period band is set in polar diagram. The top row is the strike estimated from the invariants of the impedance tensor. The bottom row is the azimuth estimated from the phase tensor. If tipper is 'y' then the 3rd row is the strike determined from the tipper, which is orthogonal to the induction arrow direction.

param fs
font size for labels of plotting. *Default* is 10

param rot_z
angle of rotation clockwise positive. *Default* is 0

param period_tolerance
float Tolerance level to match periods from different edi files. *Default* is 0.05

param text_pad
padding of the angle label at the bottom of each polar diagram. *Default* is 1.65

param text_size
font size

param plot_range
['data' | (period_min,period_max)] period range to estimate the strike angle. Options are:

- ***'data'* for estimating the strike for all periods**
  in the data.
- (pmin,pmax) for period min and period max, input as (log10(pmin),log10(pmax))

param plot_type
[1 | 2] - *1* to plot individual decades in one plot - *2* to plot all period ranges into one polar diagram

for each strike angle estimation

param plot_tipper
[True | False] - True to plot the tipper strike - False to not plot tipper strike

param pt_error_floor
Maximum error in degrees that is allowed to estimate strike. *Default* is None allowing all estimates to be used.

param fold
[True | False] * True to plot only from 0 to 180 * False to plot from 0 to 360

param plot_orthogonal
[True | False] * True to plot the orthogonal strike directions * False to not

param color
[True | False] * True to plot shade colors * False to plot all in one color

param color_inv
color of invariants plots

param color_pt
color of phase tensor plots

param color_tip
color of tipper plots

param ring_spacing
spacing of rings in polar plots

param ring_limits
(min count, max count) set each plot have these limits

param plot_orientation
['h' | 'v'] horizontal or vertical plots

Example

)

```
>>> #---Turn on Tipper
>>> strike.plot_tipper = True
>>> #---Plot only main directions
>>> strike.plot_orthogonal = False
>>> # Redraw plot
>>> strike.redraw_plot()
>>> # plot only from 0-180
>>> strike.fold = True
>>> strike.redraw_plot()
>>> #---save the plot---
>>> strike.save_plot(r"/home/Figures")
```

`get_estimate`(*estimate*, *period_range=None*)

`get_mean`(*estimate_df*)
get mean value

`get_median`(*estimate_df*)
get median value

`get_mode`(*estimate_df*)
get mode from a historgram

`get_plot_array`(*estimate*, *period_range=None*)
get a plot array that has the min and max angles

`get_stats`(*estimate*, *period_range=None*)
print stats nicely

`make_strike_df`()
make strike array

Note: Polar plots assume the azimuth is an angle measured counterclockwise positive from x = 0. Therefore all angles are calculated as 90 - angle to make them conform to the polar plot convention.

`plot`(*show=True*)
plot Strike angles as rose plots

`property rotation_angle`

Module contents

class mtpy.imaging.**PlotMTResponse**(*z_object=None*, *t_object=None*, *pt_obj=None*, *station='MT Response'*, ***kwargs*)

Bases: *PlotBase*

Plots Resistivity and phase for the different modes of the MT response. At the moment it supports the input of an .edi file. Other formats that will be supported are the impedance tensor and errors with an array of periods and .j format.

The normal use is to input an .edi file, however it would seem that not everyone uses this format, so you can input the data and put it into arrays or objects like class mtpy.core.z.Z. Or if the data is in resistivity and phase format they can be input as arrays or a class mtpy.imaging.mtplot.ResPhase. Or you can put it into a class mtpy.imaging.mtplot.MTplot.

The plot places the apparent resistivity in log scale in the top panel(s), depending on the plot_num. The phase is below this, note that 180 degrees has been added to the yx phase so the xy and yx phases plot in the same quadrant. Both the resistivity and phase share the same x-axis which is in log period, short periods on the left to long periods on the right. So if you zoom in on the plot both plots will zoom in to the same x-coordinates. If there is tipper information, you can plot the tipper as a third panel at the bottom, and also shares the x-axis. The arrows are in the convention of pointing towards a conductor. The xx and yy components can be plotted as well, this adds two panels on the right. Here the phase is left unwrapped. Other parameters can be added as subplots such as strike, skew and phase tensor ellipses.

To manipulate the plot you can change any of the attributes listed below and call redraw_plot(). If you know more aout matplotlib and want to change axes parameters, that can be done by changing the parameters in the axes attributes and then call update_plot(), note the plot must be open.

property period

plot period

plot()

plotResPhase(filename,fig_num) will plot the apparent resistivity and phase for a single station.

property plot_model_error

property rotation_angle

class mtpy.imaging.**PlotMultipleResponses**(*mt_data*, ***kwargs*)

Bases: *PlotBase*

plots multiple MT responses simultaneously either in single plots or in one plot of sub-figures or in a single plot with subfigures for each component.

Arguments:

fn_list
[list of filenames to plot] ie. [fn_1, fn_2, . . .], *default* is None

plot_num
[[1 | 2 | 3]]

- 1 for just Ex/By and Ey/Bx *default*
- 2 for all 4 components
- 3 for off diagonal plus the determinant

plot_style
[['1' | 'all' | 'compare']]

determines the plotting style:

- **'1' for plotting each station in a different**
  figure. *default*
- **'all' for plotting each station in a subplot**
  all in the same figure
- **'compare' for comparing the responses all in**
  one plot. Here the responses are colored from dark to light. This plot can get messy if too many stations are plotted.

plot()
plot the apparent resistivity and phase

property plot_model_error

property rotation_angle

class mtpy.imaging.PlotPenetrationDepth1D(*tf*, ***kwargs*)
Bases: `PlotBase`

Plot the depth of penetration based on the Niblett-Bostick approximation.

property depth_units

plot()
plot the depth of investigation as a 1d plot with period on the y-axis and depth on the x axis

Returns
DESCRIPTION

Return type
TYPE

class mtpy.imaging.PlotPenetrationDepthMap(*mt_data*, ***kwargs*)
Bases: `PlotBaseMaps`

Plot the depth of penetration based on the Niblett-Bostick approximation.

property depth_units

plot()
plot the depth of investigation as a 1d plot with period on the y-axis and depth on the x axis

Returns
DESCRIPTION

Return type
TYPE

class mtpy.imaging.PlotPhaseTensor(*pt_object*, *station=None*, ***kwargs*)
Bases: `PlotBase`

Will plot phase tensor, strike angle, min and max phase angle, azimuth, skew, and ellipticity as subplots on one plot. It can plot the resistivity tensor along side the phase tensor for comparison.

plot(*rotation_angle=None*)

plots the phase tensor elements

property rotation_angle

class mtpy.imaging.**PlotPhaseTensorMaps**(*mt_data*, ***kwargs*)

Bases: *PlotBaseMaps*

Plots phase tensor ellipses in map view from a list of edi files

property map_scale

plot(*fig=None*, *save_path=None*, *show=True*, *raster_file=None*, *raster_kwargs={}*)

Plots the phase tensor map. :param fig: optional figure object :param save_path: path to folder for saving plots :param show: show plots if True :param raster_dict: dictionary containing information for a raster

to be plotted below the phase tensors.

property rotation_angle

property skew_cmap_bounds

skew bounds for a segmented colorbar

class mtpy.imaging.**PlotPhaseTensorPseudoSection**(*mt_data*, ***kwargs*)

Bases: *PlotBaseProfile*

PlotPhaseTensorPseudoSection will plot the phase tensor ellipses in a pseudo section format

To get a list of .edi files that you want to plot –> :Example:

```
>>> import mtpy.imaging.mtplot as mtplot
>>> import os
>>> edipath = r"/home/EDIfiles"
>>> edilist = [os.path.join(edipath,edi) for edi in os.listdir(edipath)
>>> ...        if edi.find('.edi')>0]
```

- If you want to plot minimum phase colored from blue to red in a range of

  20 to 70 degrees you can do it one of two ways–>

1) :Example:

```
>>> edict = {'range':(20,70), 'cmap':'mt_bl2gr2rd','colorby':'phimin'}
>>> pt1 = mtplot.plot_pt_pseudosection(fn_list=edilist,
                                        ellipse_dict=edict)
```

2) :Example:

```
>>> pt1 = mtplot.plot_pt_pseudosection(fn_list=edilist, plot_yn='n')
>>> pt1.ellipse_colorby = 'phimin'
>>> pt1.ellipse_cmap = 'mt_bl2gr2rd'
>>> pt1.ellipse_range = (20,70)
>>> pt1.plot()
```

- If you want to add real induction arrows that are scaled by 10 and point

  away from a conductor –>

Example

```
>>> pt1.plot_tipper = 'yr'
>>> pt1.arrow_size = 10
>>> pt1.arrow_direction = -1
>>> pt1.redraw_plot()
```

- If you want to save the plot as a pdf with a generic name –>

Example
:: >>> pt1.save_figure(r"/home/PTFigures", file_format='pdf', dpi=300) File saved to '/home/PTFigures/PTPseudoSection.pdf'

plot()
plots the phase tensor pseudo section. See class doc string for more details.

class mtpy.imaging.**PlotResPhaseMaps**(*mt_data*, ***kwargs*)
Bases: *PlotBaseMaps*

Plots apparent resistivity and phase in map view from a list of edi files

Arguments:

fn_list
[list of strings] full paths to .edi files to plot

fig_size
[tuple or list (x, y) in inches] dimensions of the figure box in inches, this is a default unit of matplotlib. You can use this so make the plot fit the figure box to minimize spaces from the plot axes to the figure box. *default* is [8, 8]

mapscale
[['deg' | 'm' | 'km']] Scale of the map coordinates.

- 'deg' –> degrees in latitude and longitude
- 'm' –> meters for easting and northing
- 'km' –> kilometers for easting and northing

plot_yn
[['y' | 'n']] *'y' to plot on creating an instance

*'n' to not plot on creating an instance

title
[string] figure title

dpi
[int] dots per inch of the resolution. *default* is 300

font_size
[float] size of the font that labels the plot, 2 will be added to this number for the axis labels.

property map_units

plot()

class mtpy.imaging.**PlotResPhasePseudoSection**(*mt_data*, ***kwargs*)

Bases: *PlotBaseProfile*

plot a resistivity and phase pseudo section for different components

Need to input one of the following lists:

plot()

Returns
DESCRIPTION

Return type
TYPE

class mtpy.imaging.**PlotResidualPTMaps**(*mt_data_01*, *mt_data_02*, *frequencies=array([1.00000000e-03, 1.42510267e-03, 2.03091762e-03, 2.89426612e-03, 4.12462638e-03, 5.87801607e-03, 8.37677640e-03, 1.19377664e-02, 1.70125428e-02, 2.42446202e-02, 3.45510729e-02, 4.92388263e-02, 7.01703829e-02, 1.00000000e-01, 1.42510267e-01, 2.03091762e-01, 2.89426612e-01, 4.12462638e-01, 5.87801607e-01, 8.37677640e-01, 1.19377664e+00, 1.70125428e+00, 2.42446202e+00, 3.45510729e+00, 4.92388263e+00, 7.01703829e+00, 1.00000000e+01, 1.42510267e+01, 2.03091762e+01, 2.89426612e+01, 4.12462638e+01, 5.87801607e+01, 8.37677640e+01, 1.19377664e+02, 1.70125428e+02, 2.42446202e+02, 3.45510729e+02, 4.92388263e+02, 7.01703829e+02, 1.00000000e+03])*, ***kwargs*)

Bases: *PlotBase*

This will plot residual phase tensors in a map for a single frequency. The data is read in and stored in 2 ways, one as a list ResidualPhaseTensor object for each matching station and the other in a structured array with all the important information. The structured array is the one that is used for plotting. It is computed each time plot() is called so if it is manipulated it is reset. The array is sorted by relative offset, so no special order of input is needed for the file names. However, the station names should be verbatim between surveys, otherwise it will not work.

The residual phase tensor is calculated as I-(Phi_2)^-1 (Phi_1)

The default coloring is by the geometric mean as sqrt(Phi_min*Phi_max), which defines the percent change between measurements.

There are a lot of parameters to change how the plot looks, have a look below if you figure looks a little funny. The most useful will be ellipse_size

The ellipses are normalized by the largest Phi_max of the survey.

Example

```
>>> import mtpy.imaging.mtplot as mtplot
>>> import os
>>> edipath1 = r"/home/EDIfiles1"
>>> edilist1 = [os.path.join(edipath1,edi) for edi in os.
↪listdir(edipath1)
>>> ...         if edi.find('.edi')>0]
>>> edipath2 = r"/home/EDIfiles2"
>>> edilist2 = [os.path.join(edipath2,edi) for edi in os.
```

(continues on next page)

(continued from previous page)

```
↪listdir(edipath2)
>>> ...       if edi.find('.edi')>0]
>>> # color by phimin with a range of 0-5 deg
>>> ptmap = mtplot.plot_residual_pt_maps(edilist1, edilist2,␣
↪freqspot=10,
>>> ...                                  ellipse_dict={'size':1,
>>> ...                                                'range':(0,5)})
>>>
>>> #
>>> #---add an image---
>>> ptmap.image_file = r"/home/Maps/Basemap.jpg"
>>> ptmap.image_extent = (0,10,0,10)
>>> ptmap.redraw_plot()
>>> #
>>> #---Save the plot---
>>> ptmap.save_plot(r"/home/EDIfiles",file_format='pdf')
>>> 'Saved figure to /home/EDIfile/PTMaps/PTmap_phimin_10.0_Hz.pdf'
```

Example

```
>>> #change the axis label and grid color
>>> ptmap.ax.set_xlabel('Latitude (deg)')
>>> ptmap.ax.grid(which='major', color=(.5,1,0))
>>> ptmap.update_plot()
```

Example

```
>>> # plot seismic hypocenters from a file
>>> lat, lon, depth = np.loadtxt(r"/home/seismic_hypocenter.txt")
>>> ptmap.ax.scatter(lon, lat, marker='o')
```

property map_scale

plot()

plot residual phase tensor

property rotation_angle

class mtpy.imaging.PlotResidualPTPseudoSection(*mt_data_01, mt_data_02, frequencies=array([1.00000000e-03, 1.42510267e-03, 2.03091762e-03, 2.89426612e-03, 4.12462638e-03, 5.87801607e-03, 8.37677640e-03, 1.19377664e-02, 1.70125428e-02, 2.42446202e-02, 3.45510729e-02, 4.92388263e-02, 7.01703829e-02, 1.00000000e-01, 1.42510267e-01, 2.03091762e-01, 2.89426612e-01, 4.12462638e-01, 5.87801607e-01, 8.37677640e-01, 1.19377664e+00, 1.70125428e+00, 2.42446202e+00, 3.45510729e+00, 4.92388263e+00, 7.01703829e+00, 1.00000000e+01, 1.42510267e+01, 2.03091762e+01, 2.89426612e+01, 4.12462638e+01, 5.87801607e+01, 8.37677640e+01, 1.19377664e+02, 1.70125428e+02, 2.42446202e+02, 3.45510729e+02, 4.92388263e+02, 7.01703829e+02, 1.00000000e+03]), **kwargs*)

Bases: `PlotBaseProfile`

This will plot residual phase tensors in a pseudo section. The data is read in and stored in 2 ways, one as a list ResidualPhaseTensor object for each matching station and the other in a structured array with all the important information. The structured array is the one that is used for plotting. It is computed each time plot() is called so if it is manipulated it is reset. The array is sorted by relative offset, so no special order of input is needed for the file names. However, the station names should be verbatim between surveys, otherwise it will not work.

The residual phase tensor is calculated as I-(Phi_2)^-1 (Phi_1)

The default coloring is by the geometric mean as sqrt(Phi_min*Phi_max), which defines the percent change between measurements.

There are a lot of parameters to change how the plot looks, have a look below if you figure looks a little funny. The most useful will be xstretch, ystretch and ellipse_size

The ellipses are normalized by the largest Phi_max of the survey.

To get a list of .edi files that you want to plot –> :Example:

```
>>> import mtpy.imaging.mtplot as mtplot
>>> import os
>>> edipath1 = r"/home/EDIfiles1"
>>> edilist1 = [os.path.join(edipath1,edi) for edi in os.listdir(edipath1)
>>> ...         if edi.find('.edi')>0]
>>> edipath2 = r"/home/EDIfiles2"
>>> edilist2 = [os.path.join(edipath2,edi) for edi in os.listdir(edipath2)
>>> ...         if edi.find('.edi')>0]
>>> # color by phimin with a range of 0-5 deg
```

- If you want to plot geometric mean colored from white to orange in a range of 0 to 10 percent you can do it one of two ways–>

1) :Example:

```
>>> edict = {'range':(0,10), 'cmap':'mt_wh2or',                      'colorby':
↪'geometric_mean', 'size':10}
>>> pt1 = mtplot.residual_pt_ps(edilist1, edilst2, ellipse_dict=edict)
```

2) :Example:

```
>>> pt1 = mtplot.residual_pt_ps(edilist1, edilst2, ellipse_dict=edict,                ␣
↪                            plot_yn='n')
>>> pt1.ellipse_colorby = 'geometric_mean'
>>> pt1.ellipse_cmap = 'mt_wh2or'
>>> pt1.ellipse_range = (0, 10)
>>> pt1.ellipse_size = 10
>>> pt1.plot()
```

- If you want to save the plot as a pdf with a generic name –>

  Example

  :: >>> pt1.save_figure(r"/home/PTFigures", file_format='pdf', dpi=300) File saved to '/home/PTFigures/PTPseudoSection.pdf'

get_pt_color_array(*rpt_array*)

Get the appropriat color by array

plot()

plot residual phase tensor

property rotation_angle

class mtpy.imaging.**PlotStations**(*geo_df*, ***kwargs*)

Bases: *PlotBase*

plot station locations in map view.

Uses contextily to get the basemap. See https://contextily.readthedocs.io/en/latest/index.html for more information about options.

plot()

Parameters

- **cx_source** (*TYPE, optional*) – DESCRIPTION, defaults to cx.providers.USGS.USTopo
- **cx_zoom** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

class mtpy.imaging.**PlotStrike**(*mt_data*, ***kwargs*)

Bases: *PlotBase*

PlotStrike will plot the strike estimated from the invariants, phase tensor and the tipper in either a rose diagram of xy plot

plots the strike angle as determined by phase tensor azimuth (Caldwell et al. [2004]) and invariants of the impedance tensor (Weaver et al. [2003]).

The data is split into decades where the histogram for each is plotted in the form of a rose diagram with a range of 0 to 180 degrees. Where 0 is North and 90 is East. The median angle of the period band is set in polar diagram. The top row is the strike estimated from the invariants of the impedance tensor. The bottom row is the azimuth estimated from the phase tensor. If tipper is 'y' then the 3rd row is the strike determined from the tipper, which is orthogonal to the induction arrow direction.

param fs
font size for labels of plotting. *Default* is 10

param rot_z
angle of rotation clockwise positive. *Default* is 0

param period_tolerance
float Tolerance level to match periods from different edi files. *Default* is 0.05

param text_pad
padding of the angle label at the bottom of each polar diagram. *Default* is 1.65

param text_size
font size

param plot_range
['data' | (period_min,period_max)] period range to estimate the strike angle. Options are:

- ***'data'* for estimating the strike for all periods**
  in the data.
- (pmin,pmax) for period min and period max, input as (log10(pmin),log10(pmax))

param plot_type
[1 | 2] - *1* to plot individual decades in one plot - *2* to plot all period ranges into one polar diagram

for each strike angle estimation

param plot_tipper
[True | False] - True to plot the tipper strike - False to not plot tipper strike

param pt_error_floor
Maximum error in degrees that is allowed to estimate strike. *Default* is None allowing all estimates to be used.

param fold
[True | False] * True to plot only from 0 to 180 * False to plot from 0 to 360

param plot_orthogonal
[True | False] * True to plot the orthogonal strike directions * False to not

param color
[True | False] * True to plot shade colors * False to plot all in one color

param color_inv
color of invariants plots

param color_pt
color of phase tensor plots

param color_tip
color of tipper plots

param ring_spacing
spacing of rings in polar plots

param ring_limits
(min count, max count) set each plot have these limits

param plot_orientation
['h' | 'v'] horizontal or vertical plots

Example

)

```
>>> #---Turn on Tipper
>>> strike.plot_tipper = True
>>> #---Plot only main directions
>>> strike.plot_orthogonal = False
>>> # Redraw plot
>>> strike.redraw_plot()
>>> # plot only from 0-180
>>> strike.fold = True
>>> strike.redraw_plot()
```

(continues on next page)

(continued from previous page)

```
>>> #---save the plot---
>>> strike.save_plot(r"/home/Figures")
```

get_estimate(*estimate*, *period_range=None*)

get_mean(*estimate_df*)
get mean value

get_median(*estimate_df*)
get median value

get_mode(*estimate_df*)
get mode from a historgram

get_plot_array(*estimate*, *period_range=None*)
get a plot array that has the min and max angles

get_stats(*estimate*, *period_range=None*)
print stats nicely

make_strike_df()
make strike array

Note: Polar plots assume the azimuth is an angle measured counterclockwise positive from x = 0. Therefore all angles are calculated as 90 - angle to make them conform to the polar plot convention.

plot(*show=True*)
plot Strike angles as rose plots

property rotation_angle

mtpy.modeling package

Subpackages

mtpy.modeling.modem package

Submodules

mtpy.modeling.modem.config module

ModEM

Generate files for ModEM

revised by JP 2017 # revised by AK 2017 to bring across functionality from ak branch

class mtpy.modeling.modem.config.**ModEMConfig**(***kwargs*)
Bases: object

read and write configuration files for how each inversion is run

add_dict(*fn=None*, *obj=None*)
add dictionary based on file name or object

write_config_file(*save_dir=None*, *config_fn_basename='ModEM_inv.cfg'*)
write a config file based on provided information

mtpy.modeling.modem.control_fwd module

ModEM

Generate files for ModEM

revised by JP 2017 # revised by AK 2017 to bring across functionality from ak branch

class mtpy.modeling.modem.control_fwd.**ControlFwd**(***kwargs*)
Bases: `object`

read and write control file for

This file controls how the inversion starts and how it is run

property control_fn

read_control_file(*control_fn=None*)
read in a control file

write_control_file(*control_fn=None*, *save_path=None*, *fn_basename=None*)
write control file

Arguments:

control_fn
[string] full path to save control file to *default* is save_path/fn_basename

save_path
[string] directory path to save control file to *default* is cwd

fn_basename
[string] basename of control file *default* is control.inv

mtpy.modeling.modem.control_inv module

ModEM

Generate files for ModEM

revised by JP 2017 # revised by AK 2017 to bring across functionality from ak branch

class mtpy.modeling.modem.control_inv.**ControlInv**(***kwargs*)
Bases: `object`

read and write control file for how the inversion starts and how it is run

property control_fn

read_control_file(*control_fn=None*)

read in a control file

write_control_file(*control_fn=None*, *save_path=None*, *fn_basename=None*)

write control file

Arguments:

control_fn
[string] full path to save control file to *default* is save_path/fn_basename

save_path
[string] directory path to save control file to *default* is cwd

fn_basename
[string] basename of control file *default* is control.inv

mtpy.modeling.modem.convariance module

ModEM

Generate files for ModEM

revised by JP 2017 # revised by AK 2017 to bring across functionality from ak branch

class mtpy.modeling.modem.convariance.**Covariance**(*grid_dimensions=None*, ***kwargs*)

Bases: `object`

read and write covariance files

property cov_fn

get_parameters()

read_cov_file(*cov_fn*)

read a covariance file

write_cov_vtk_file(*cov_vtk_fn*, *model_fn=None*, *grid_east=None*, *grid_north=None*, *grid_z=None*)

write a vtk file of the covariance to match things up

write_covariance_file(*cov_fn=None*, *save_path=None*, *fn_basename=None*, *res_model=None*, *sea_water=0.3*, *air=1000000000000.0*)

write a covariance file

mtpy.modeling.modem.data module

ModEM

Generate files for ModEM

revised by JP 2017 # revised by AK 2017 to bring across functionality from ak branch # revised by JP 2021 adding functionality and updating. # revised by JP 2022 to work with new structure of a central object

class `mtpy.modeling.modem.data.Data`(*dataframe=None*, *center_point=None*, ***kwargs*)

Bases: `object`

Data will read and write .dat files for ModEM and convert a WS data file to ModEM format.

..note: :: the data is interpolated onto the given periods such that all
stations invert for the same periods. The interpolation is a linear interpolation of each of the real and imaginary parts of the impedance tensor and induction tensor. See mtpy.core.mt.MT.interpolate for more details

Parameters
edi_list – list of edi files to read

Attributes	Description
_dtype	internal variable defining the data type of data_array
_logger	python logging object that put messages in logging format defined in logging configure file, see MtPyLog for more information
_t_shape	internal variable defining shape of tipper array in _dtype
_z_shape	internal variable defining shape of Z array in _dtype
center_position	(east, north, evel) for center point of station array. All stations are relative to this location for plotting purposes.
comp_index_dict	dictionary for index values of component of Z and T
station_locations	Stations object
data_array	numpy.ndarray (num_stations) structured to store data. keys are: • station –> station name • lat –> latitude in decimal degrees • lon –> longitude in decimal degrees • elev –> elevation (m) • **rel_east – > relative east location to** center_position (m) • **rel_north –> relative north location to** center_position (m) • east –> UTM east (m) • north –> UTM north (m) • zone –> UTM zone • **z –> impedance tensor array with shape** (num_freq, 2, 2) • **z_err –> impedance tensor error array with** shape (num_freq, 2, 2) • **tip –> Tipper array with shape** (num_freq, 1, 2) • **tipperr –> Tipper array with shape** (num_freq, 1, 2)
data_fn	full path to data file
data_period_list	period list from all the data
edi_list	list of full paths to edi files
error_type_tipper	['abs' | 'floor'] *default* is 'abs'

continues on next page

Table 1 – continued from previous page

Attributes	Description
error_type_z	['egbert' | 'mean_od' | 'eigen' | 'median'] *default* is 'egbert_floor' • add '_floor' to any of the above to set the error as an error floor, otherwise all components are give weighted the same • **'egbert' sets error to** error_value_z * sqrt(abs(zxy*zyx)) • **'mean_od' sets error to** error_value_z * mean([Zxy, Zyx]) (non zeros) • **'eigen' sets error to** error_value_z * eigenvalues(Z[ii]) • **'median' sets error to** error_value_z * median([Zxx, Zxy, Zyx, Zyy]) (non zeros) A 2x2 numpy array of error_type_z can be specified to explicitly set the error_type_z for each component.
error_value_z	percentage to multiply Z by to set error *default* is 5 for 5% of Z as error A 2x2 numpy array of values can be specified to explicitly set the error_value_z for each component.
error_value_tipper	absolute error between 0 and 1.
fn_basename	basename of data file. *default* is 'ModEM_Data.dat'
formatting	['1' | '2'], format of the output data file, *default* is '1'
header_strings	strings for header of data file following the format outlined in the ModEM documentation
inv_comp_dict	dictionary of inversion components
inv_mode	**inversion mode, options are: *default* is '1'** • **'1' –> for 'Full_Impedance' and** 'Full_Vertical_Components' • '2' –> 'Full_Impedance' • **'3' –> 'Off_Diagonal_Impedance' and** 'Full_Vertical_Components' • '4' –> 'Off_Diagonal_Impedance' • '5' –> 'Full_Vertical_Components' • '6' –> 'Full_Interstation_TF' • '7' –> 'Off_Diagonal_Rho_Phase'
inv_mode_dict	dictionary for inversion modes
max_num_periods	maximum number of periods
model_epsg	epsg code for model projection, provide this to project model to non-utm coordinates. Find the epsg code for your projection on http://spatialreference.org/ref/ or google search epsg "your projection"
model_utm_zone	alternative to model_epsg, choose a utm zone to project all sites to (e.g. '55S')
mt_dict	dictionary of mtpy.core.mt.MT objects with keys being station names

continues on next page

Table 1 – continued from previous page

Attributes	Description
period_buffer	float or int if specified, apply a buffer so that interpolation doesn't stretch too far over periods
period_dict	dictionary of period index for period_list
period_list	list of periods to invert for
period_max	maximum value of period to invert for
period_min	minimum value of period to invert for
period_buffer	**buffer so that interpolation doesn't stretch too far** over periods. Provide a float or integer factor, greater than which interpolation will not stretch. e.g. 1.5 means only interpolate to a maximum of 1.5 times each side of each frequency value
rotate_angle	Angle to rotate data to assuming 0 is N and E is 90
save_path	path to save data file to
units	[[V/m]/[T] | [mV/km]/[nT] | Ohm] units of Z *default* is [mV/km]/[nT]
wave_sign_impedance	[+ | -] sign of time dependent wave. *default* is '+' as positive downwards.
wave_sign_tipper	[+ | -] sign of time dependent wave. *default* is '+' as positive downwards.

Example 1 –> create inversion period list

```
>>> from pathlib import Path
>>> import mtpy.modeling.modem as modem
>>> edi_path = Path(r"/home/mt/edi_files")
>>> edi_list = list(edi_path.glob("*.edi"))
>>> md = modem.Data(edi_list, period_min=.1, period_max=300,        >>>
↪ ...                max_num_periods=12)
>>> md.write_data_file(save_path=r"/home/modem/inv1")
>>> md
```

Example 2 –> set inverions period list from data

```
>>> md = modem.Data(edi_list)
>>> #get period list from an .edi file
>>> inv_period_list = 1./md.mt_dict["mt01"].Z.freq
>>> #invert for every third period in inv_period_list
>>> inv_period_list = inv_period_list[np.arange(0, len(inv_period_list,
↪ 3))]
>>> md.period_list = inv_period_list
>>> md.write_data_file(save_path=r"/home/modem/inv1")
```

Example 3 –> change error values

```
>>> mdr.error_type = 'floor'
>>> mdr.error_floor = 10
>>> mdr.error_tipper = .03
>>> mdr.write_data_file(save_path=r"/home/modem/inv2")
```

Example 4 –> change inversion type

```
>>> mdr.inv_mode = '3'
>>> mdr.write_data_file(save_path=r"/home/modem/inv2")
```

Example 5 –> rotate data

```
>>> md.rotation_angle = 60
>>> md.write_data_file(save_path=r"/home/modem/Inv1")
>>> # or
>>> md.write_data_file(save_path=r"/home/modem/Inv1",
↪                rotation_angle=60)
```

property data_filename

property dataframe

fix_data_file(*fn=None*, *n=3*)

A newer compiled version of Modem outputs slightly different headers This aims to convert that into the older format

Parameters

- **fn** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **n** (*TYPE, optional*) – DESCRIPTION, defaults to 3

Returns
DESCRIPTION

Return type
TYPE

get_n_stations(*mode*)

property model_parameters

property n_periods

property period

read_data_file(*data_fn*)

Parameters
data_fn (*string or Path*) – full path to data file name

Raises
ValueError – If cannot compute component

Fills attributes:

- data_array
- period_list
- mt_dict

```
>>> md = Data()
>>> md.read_data_file(r"/home/modem_data.dat")
>>> md
ModEM Data Object:
    Number of stations: 169
    Number of periods:  22
    Period range:
            Min: 0.01 s
            Max: 15230.2 s
    Rotation angle:     0.0
    Data center:
             latitude:  39.6351 deg
             longitude: -119.8039 deg
             Elevation: 0.0 m
             Easting:   259368.9746 m
             Northing:  4391021.1981 m
             UTM zone:  11S
    Model EPSG:         None
    Model UTM zone:     None
    Impedance data:     True
    Tipper data:        True
```

write_data_file(*file_name=None*, *save_path=None*, *fn_basename=None*, *elevation=False*)

Parameters

- **save_path** (*string or Path, optional*) – full directory to save file to, defaults to None
- **fn_basename** (*string, optional*) – Basename of the saved file, defaults to None
- **elevation** (*boolean, optional*) – If True adds in elevation from 'rel_elev' column in data array, defaults to False

Raises

- **NotImplementedError** – If the inversion mode is not supported
- **ValueError** – `mtpy.utils.exceptions.ValueError` if a parameter

is missing :return: full path to data file :rtype: Path

```
>>> from pathlib import Path
>>> import mtpy.modeling.modem as modem
>>> edi_path = Path(r"/home/mt/edi_files")
>>> edi_list = list(edi_path.glob("*.ed"))
>>> md = modem.Data(edi_list, period_min=.1, period_max=300,          >>> ...
↪          max_num_periods=12)
>>> md.write_data_file(save_path=r"/home/modem/inv1")
/home/modem/inv1/ModemDataFile.dat
```

mtpy.modeling.modem.data_model_analysis module

mtpy.modeling.modem.exception module

ModEM

Generate files for ModEM

revised by JP 2017 # revised by AK 2017 to bring across functionality from ak branch

exception mtpy.modeling.modem.exception.**DataError**

Bases: *ModEMError*

Raise for ModEM Data class specific exceptions

exception mtpy.modeling.modem.exception.**ModEMError**

Bases: Exception

mtpy.modeling.modem.model module

ModEM

Generate files for ModEM

revised by JP 2017 # revised by AK 2017 to bring across functionality from ak branch # revised by JP 2021 updating functionality and updating docs

class mtpy.modeling.modem.model.**Model**(*station_locations=None*, *center_point=None*, ***kwargs*)

Bases: object

make and read a FE mesh grid

The mesh assumes the coordinate system where:
x == North y == East z == + down

All dimensions are in meters.

The mesh is created by first making a regular grid around the station area, then padding cells are added that exponentially increase to the given extensions. Depth cell increase on a log10 scale to the desired depth, then padding cells are added that increase exponentially.

Parameters

****station_object**** (*mtpy.modeling.modem.Stations object*) –

See also:

mtpy.modeling.modem.Stations

Examples

Example 1 –> create mesh first then data file

```
>>> import mtpy.modeling.modem as modem
>>> import os
>>> # 1) make a list of all .edi files that will be inverted for
>>> edi_path = r"/home/EDI_Files"
>>> edi_list = [os.path.join(edi_path, edi)
```

for edi in os.listdir(edi_path)

```
>>> ...          if edi.find('.edi') > 0]
>>> # 2) Make a Stations object
>>> stations_obj = modem.Stations()
>>> stations_obj.get_station_locations_from_edi(edi_list)
>>> # 3) make a grid from the stations themselves with 200m cell␣
↪spacing
>>> mmesh = modem.Model(station_obj)
>>> # change cell sizes
>>> mmesh.cell_size_east = 200,
>>> mmesh.cell_size_north = 200
>>> mmesh.ns_ext = 300000 # north-south extension
>>> mmesh.ew_ext = 200000 # east-west extension of model
>>> mmesh.make_mesh()
>>> # check to see if the mesh is what you think it should be
>>> msmesh.plot_mesh()
>>> # all is good write the mesh file
>>> msmesh.write_model_file(save_path=r"/home/modem/Inv1")
>>> # create data file
>>> md = modem.Data(edi_list, station_locations=mmesh.station_
↪locations)
>>> md.write_data_file(save_path=r"/home/modem/Inv1")
```

Example 2 –> Rotate Mesh

```
>>> mmesh.mesh_rotation_angle = 60
>>> mmesh.make_mesh()
```

Note: ModEM assumes all coordinates are relative to North and East, and does not accommodate mesh rotations, therefore, here the rotation is of the stations, which essentially does the same thing. You will need to rotate you data to align with the 'new' coordinate system.

Attributes	Description
_logger	python logging object that put messages in logging format defined in logging configure file, see MtPyLog more information
cell_number_ew	optional for user to specify the total number of sells on the east-west direction. *default* is None

continues on next page

Table 2 – continued from previous page

Attributes	Description
cell_number_ns	optional for user to specify the total number of sells on the north-south direction. *default* is None
cell_size_east	mesh block width in east direction *default* is 500
cell_size_north	mesh block width in north direction *default* is 500
grid_center	center of the mesh grid
grid_east	overall distance of grid nodes in east direction
grid_north	overall distance of grid nodes in north direction
grid_z	overall distance of grid nodes in z direction
model_fn	full path to initial file name
model_fn_basename	default name for the model file name
n_air_layers	number of air layers in the model. *default* is 0
n_layers	total number of vertical layers in model
nodes_east	relative distance between nodes in east direction
nodes_north	relative distance between nodes in north direction
nodes_z	relative distance between nodes in east direction
pad_east	number of cells for padding on E and W sides *default* is 7
pad_north	number of cells for padding on S and N sides *default* is 7
pad_num	number of cells with cell_size with outside of station area. *default* is 3
pad_method	method to use to create padding: extent1, extent2 - calculate based on ew_ext and ns_ext stretch - calculate based on pad_stretch factors
pad_stretch_h	multiplicative number for padding in horizontal direction.
pad_stretch_v	padding cells N & S will be pad_root_north**(x)
pad_z	number of cells for padding at bottom *default* is 4
ew_ext	E-W extension of model in meters
ns_ext	N-S extension of model in meters
res_scale	**scaling method of res, supports** 'loge' - for log e format 'log' or 'log10' - for log with base 10 'linear' - linear scale *default* is 'loge'
res_list	list of resistivity values for starting model
res_model	starting resistivity model
res_initial_value	resistivity initial value for the resistivity model *default* is 100
mesh_rotation_angle	Angle to rotate the grid to. Angle is measured positve clockwise assuming North is 0 and east is 90. *default* is None
save_path	path to save file to
sea_level	sea level in grid_z coordinates. *default* is 0
station_locations	location of stations
title	title in initial file
z1_layer	first layer thickness
z_bottom	absolute bottom of the model *default* is 300,000
z_target_depth	Depth of deepest target, *default* is 50,000

add_layers_to_mesh(*n_add_layers=None*, *layer_thickness=None*, *where='top'*)

Function to add constant thickness layers to the top or bottom of mesh. Note: It is assumed these layers are added before the topography. If you want to add topography layers, use function add_topography_to_model

Parameters

- **n_add_layers** – integer, number of layers to add
- **layer_thickness** – real value or list/array. Thickness of layers, defaults to z1 layer. Can provide a single value or a list/array containing multiple layer thicknesses.
- **where** – where to add, top or bottom

add_topography_from_data(*interp_method='nearest'*, *air_resistivity=1000000000000.0*, *topography_buffer=None*, *airlayer_type='log_up'*)

Wrapper around add_topography_to_model that allows creating a surface model from EDI data. The Data grid and station elevations will be used to make a 'surface' tuple that will be passed to add_topography_to_model so a surface model can be interpolated from it.

The surface tuple is of format (lon, lat, elev) containing station locations.

Parameters

- **data_object** (*mtpy.modeling.ModEM.data.Data*) – A ModEm data object that has been filled with data from EDI files.
- **interp_method** (*str, optional*) – Same as add_topography_to_model.
- **air_resistivity** (*float, optional*) – Same as add_topography_to_model.
- **topography_buffer** (*float*) – Same as add_topography_to_model.
- **airlayer_type** (*str, optional*) – Same as add_topography_to_model.

add_topography_to_model(*topography_file=None*, *surface=None*, *topography_array=None*, *interp_method='nearest'*, *air_resistivity=1000000000000.0*, *topography_buffer=None*, *airlayer_type='log_up'*, *max_elev=None*, *shift_east=0*, *shift_north=0*)

if air_layers is non-zero, will add topo: read in topograph file, make a surface model.

Call project_stations_on_topography in the end, which will re-write the .dat file.

If n_airlayers is zero, then cannot add topo data, only bathymetry is needed.

Parameters

- **topography_file** – file containing topography (arcgis ascii grid)
- **topography_array** – alternative to topography_file - array of elevation values on model grid
- **interp_method** – interpolation method for topography, 'nearest', 'linear', or 'cubic'
- **air_resistivity** – resistivity value to assign to air
- **topography_buffer** – buffer around stations to calculate minimum and maximum topography value to use for meshing
- **airlayer_type** – how to set air layer thickness - options are 'constant' for constant air layer thickness, or 'log', for logarithmically increasing air layer thickness upward

assign_resistivity_from_surface_data(*top_surface*, *bottom_surface*, *resistivity_value*)

assign resistivity value to all points above or below a surface requires the surface_dict attribute to exist and contain data for surface key (can get this information from ascii file using project_surface)

inputs surface_name = name of surface (must correspond to key in surface_dict) resistivity_value = value to assign where = 'above' or 'below' - assign resistivity above or below the

surface

`interpolate_elevation`(*surface_file=None*, *surface=None*, *get_surface_name=False*, *method='nearest'*, *fast=True*, *shift_north=0*, *shift_east=0*)

project a surface to the model grid and add resulting elevation data to a dictionary called surface_dict. Assumes the surface is in lat/long coordinates (wgs84)

returns nothing returned, but surface data are added to surface_dict under the key given by surface_name.

inputs choose to provide either surface_file (path to file) or surface (tuple). If both are provided then surface tuple takes priority.

surface elevations are positive up, and relative to sea level. surface file format is:

ncols 3601 nrows 3601 xllcorner -119.00013888889 (longitude of lower left) yllcorner 36.999861111111 (latitude of lower left) cellsize 0.00027777777777778 NODATA_value -9999 elevation data W –> E N | V S

Alternatively, provide a tuple with: (lon,lat,elevation) where elevation is a 2D array (shape (ny,nx)) containing elevation points (order S -> N, W -> E) and lon, lat are either 1D arrays containing list of longitudes and latitudes (in the case of a regular grid) or 2D arrays with same shape as elevation array containing longitude and latitude of each point.

other inputs: surface_epsg = epsg number of input surface, default is 4326 for lat/lon(wgs84) method = interpolation method. Default is 'nearest', if model grid is dense compared to surface points then choose 'linear' or 'cubic'

`make_mesh`(*verbose=True*)

create finite element mesh according to user-input parameters.

The mesh is built by:

1. Making a regular grid within the station area.
2. Adding pad_num of cell_width cells outside of station area
3. Adding padding cells to given extension and number of padding cells.
4. Making vertical cells starting with z1_layer increasing logarithmically (base 10) to z_target_depth and num_layers.
5. Add vertical padding cells to desired extension.
6. Check to make sure none of the stations lie on a node. If they do then move the node by .02*cell_width

`make_z_mesh`(*n_layers=None*)

new version of make_z_mesh. make_z_mesh and M

`property model_epsg`

`property model_fn`

`property model_parameters`

get important model parameters to write to a file for documentation later.

`property nodes_east`

`property nodes_north`

property nodes_z

property plot_east

plot_mesh(***kwargs*)

Plot model mesh

Parameters

plot_topography (*TYPE, optional*) – DESCRIPTION, defaults to False

Returns

DESCRIPTION

Return type

TYPE

property plot_north

property plot_z

read_gocad_sgrid_file(*sgrid_header_file*, *air_resistivity=1e+39*, *sea_resistivity=0.3*, *sgrid_positive_up=True*)

read a gocad sgrid file and put this info into a ModEM file. Note: can only deal with grids oriented N-S or E-W at this stage, with orthogonal coordinates

read_model_file(*model_fn=None*)

read an initial file and return the pertinent information including grid positions in coordinates relative to the center point (0,0) and starting model.

Note that the way the model file is output, it seems is that the blocks are setup as

ModEM: WS: ———- —– 0—–> N_north 0——–>N_east | | | | V V N_east N_north

Arguments:

model_fn : full path to initializing file.

Outputs:

nodes_north
[np.array(nx)] array of nodes in S –> N direction

nodes_east
[np.array(ny)] array of nodes in the W –> E direction

nodes_z
[np.array(nz)] array of nodes in vertical direction positive downwards

res_model
[dictionary] dictionary of the starting model with keys as layers

res_list
[list] list of resistivity values in the model

title
[string] title string

property save_path

write_geosoft_xyz(*save_fn*, *c_east=0*, *c_north=0*, *c_z=0*, *pad_north=0*, *pad_east=0*, *pad_z=0*)

Write an XYZ file readable by Geosoft

All input units are in meters.

Parameters

- **save_fn** (*string or Path*) – full path to save file to
- **c_east** (*float, optional*) – center point in the east direction, defaults to 0
- **c_north** (*float, optional*) – center point in the north direction, defaults to 0
- **c_z** (*float, optional*) – center point elevation, defaults to 0
- **pad_north** (*int, optional*) – number of cells to cut from the north-south edges, defaults to 0
- **pad_east** (*int, optional*) – number of cells to cut from the east-west edges, defaults to 0
- **pad_z** (*int, optional*) – number of cells to cut from the bottom, defaults to 0

write_gocad_sgrid_file(*fn=None*, *origin=[0, 0, 0]*, *clip=0*, *no_data_value=-99999*)

write a model to gocad sgrid

optional inputs:

fn = filename to save to. File extension ('.sg') will be appended.
default is the model name with extension removed

origin = real world [x,y,z] location of zero point in model grid clip = how much padding to clip off the edge of the model for export,

provide one integer value or list of 3 integers for x,y,z directions

no_data_value = no data value to put in sgrid

write_model_file(***kwargs*)

will write an initial file for ModEM.

Note that x is assumed to be S –> N, y is assumed to be W –> E and z is positive downwards. This means that index [0, 0, 0] is the southwest corner of the first layer. Therefore if you build a model by hand the layer block will look as it should in map view.

Also, the xgrid, ygrid and zgrid are assumed to be the relative distance between neighboring nodes. This is needed because wsinv3d builds the model from the bottom SW corner assuming the cell width from the init file.

Key Word Arguments:

nodes_north
[np.array(nx)] block dimensions (m) in the N-S direction. **Note** that the code reads the grid assuming that index=0 is the southern most point.

nodes_east
[np.array(ny)] block dimensions (m) in the E-W direction. **Note** that the code reads in the grid assuming that index=0 is the western most point.

nodes_z
[np.array(nz)] block dimensions (m) in the vertical direction. This is positive downwards.

save_path
[string] Path to where the initial file will be saved to save_path/model_fn_basename

model_fn_basename
[string] basename to save file to *default* is ModEM_Model.ws file is saved at save_path/model_fn_basename

title
[string] Title that goes into the first line *default* is Model File written by MTpy.modeling.modem

res_model
[np.array((nx,ny,nz))] Prior resistivity model.

Note: again that the modeling code

assumes that the first row it reads in is the southern most row and the first column it reads in is the western most column. Similarly, the first plane it reads in is the Earth's surface.

res_starting_value
[float] starting model resistivity value, assumes a half space in Ohm-m *default* is 100 Ohm-m

res_scale
[['loge' | 'log' | 'log10' | 'linear']] scale of resistivity. In the ModEM code it converts everything to Loge, *default* is 'loge'

`write_out_file`(*save_fn*, *geographic_east*, *geographic_north*, *geographic_elevation*)

will write an .out file for LeapFrog.

Note that y is assumed to be S –> N, e is assumed to be W –> E and z is positive upwards. This means that index [0, 0, 0] is the southwest corner of the first layer.

Parameters

- **save_fn** (*string or Path*) – full path to save file to
- **geographic_east** (*float*) – geographic center in easting (meters)
- **geographic_north** (*float*) – geographic center in northing (meters)
- **geographic_elevation** (*float*) – elevation of geographic center (meters)

Returns
DESCRIPTION

Return type
TYPE

`write_ubc_files`(*basename*, *c_east=0*, *c_north=0*, *c_z=0*)

Write a UBC .msh and .mod file

Parameters

- **save_fn** (*TYPE*) – DESCRIPTION
- **c_east** (*TYPE, optional*) – DESCRIPTION, defaults to 0
- **c_north** (*TYPE, optional*) – DESCRIPTION, defaults to 0
- **c_z** (*TYPE, optional*) – DESCRIPTION, defaults to 0

Returns
DESCRIPTION

Return type
TYPE

Note: not complete yet.

write_vtk_file(*vtk_save_path=None*, *vtk_fn_basename='ModEM_model_res'*, *shift_east=0*, *shift_north=0*, *shift_z=0*, *units='km'*, *coordinate_system='nez+'*, *label='resistivity'*)

Write a VTK file to plot in 3D rendering programs like Paraview

Parameters

- **vtk_save_path** (*string or Path, optional*) – directory to save vtk file to, defaults to None
- **vtk_fn_basename** – filename basename of vtk file, note that .vtr

extension is automatically added, defaults to "ModEM_stations" :type vtk_fn_basename: string, optional :type geographic: boolean, optional :param shift_east: shift in east directions in meters, defaults to 0 :type shift_east: float, optional :param shift_north: shift in north direction in meters, defaults to 0 :type shift_north: float, optional :param shift_z: shift in elevation + down in meters, defaults to 0 :type shift_z: float, optional :param units: Units of the spatial grid [km | m | ft], defaults to "km" :type units: string, optional :type : string :param coordinate_system: coordinate system for the station, either the normal MT right-hand coordinate system with z+ down or the sinister z- down [nez+ | enz-], defaults to nez+ :return: full path to VTK file :rtype: Path

Write VTK file >>> model.write_vtk_file(vtk_fn_basename="modem_model")

Write VTK file in geographic coordinates with z+ up >>> model.write_vtk_station_file(vtk_fn_basename="modem_model", >>> … coordinate_system='enz-')

write_xyres(*save_path=None*, *location_type='EN'*, *origin=[0, 0]*, *model_epsg=None*, *depth_index='all'*, *outfile_basename='DepthSlice'*, *log_res=False*, *clip=[0, 0]*)

write files containing depth slice data (x, y, res for each depth)

origin = x,y coordinate of zero point of ModEM_grid, or name of file
containing this info (full path or relative to model files)

save_path = path to save to, default is the model object save path location_type = 'EN' or 'LL' xy points saved as eastings/northings or

longitude/latitude, if 'LL' need to also provide model_epsg

model_epsg = epsg number that was used to project the model outfile_basename = string for basename for saving the depth slices. log_res = True/False - option to save resistivity values as log10

instead of linear

clip = number of cells to clip on each of the east/west and north/south edges

write_xyzres(*savefile=None*, *location_type='EN'*, *origin=[0, 0]*, *model_epsg=None*, *log_res=False*, *model_utm_zone=None*, *clip=[0, 0]*)

save a model file as a space delimited x y z res file

mtpy.modeling.modem.model_manipulator module

mtpy.modeling.modem.phase_tensor_maps module

mtpy.modeling.modem.plot_response module

mtpy.modeling.modem.plot_rms_maps module

mtpy.modeling.modem.plot_slices module

mtpy.modeling.modem.residual module

ModEM

residuals class to contain RMS information

revised by JP 2017 revised by AK 2017 to bring across functionality from ak branch

class `mtpy.modeling.modem.residual.`**`Residual`**`(**kwargs)`

Bases: `Data`

class to contain residuals for each data point, and rms values for each station

Attributes/Key Words	Description
work_dir	
residual_fn	full path to data file
residual_array	numpy.ndarray (num_stations) structured to store data. keys are: • station –> station name • lat –> latitude in decimal degrees • lon –> longitude in decimal degrees • elev –> elevation (m) • **rel_east – > relative east location to** center_position (m) • **rel_north –> relative north location to** center_position (m) • east –> UTM east (m) • north –> UTM north (m) • zone –> UTM zone • **z –> impedance tensor residual (measured - modelled)** (num_freq, 2, 2) • **z_err –> impedance tensor error array with** shape (num_freq, 2, 2) • **tip –> Tipper residual (measured - modelled)** (num_freq, 1, 2) • **tipperr –> Tipper array with shape** (num_freq, 1, 2)
rms	
rms_array	numpy.ndarray structured to store station location values and rms. Keys are: • station –> station name • east –> UTM east (m) • north –> UTM north (m) • lat –> latitude in decimal degrees • lon –> longitude in decimal degrees • elev –> elevation (m) • zone –> UTM zone • **rel_east – > relative east location to** center_position (m) • **rel_north –> relative north location to** center_position (m) • **rms –> root-mean-square residual for each** station
rms_tip	
rms_z	

calculate_rms()

add columns for rms :return: DESCRIPTION :rtype: TYPE

plot_rms(***kwargs*)

plot RMS in different views

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_rms_per_period(*plot_type='all'*, ***kwargs*)

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

read_residual_file(*residual_fn*)

Parameters
residual_fn (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

property rms_per_period_all
RMS per period

property rms_per_period_per_component
RMS per period by component

Returns
DESCRIPTION

Return type
TYPE

mtpy.modeling.modem.station module

ModEM

Generate files for ModEM

revised by JP 2017 # revised by AK 2017 to bring across functionality from ak branch

class mtpy.modeling.modem.station.**Stations**(***kwargs*)
Bases: object

station locations class

calculate_rel_locations(*shift_east=0*, *shift_north=0*)
put station in a coordinate system relative to (shift_east, shift_north) (+) shift right or up (-) shift left or down

property center_point

calculate the center point from the given station locations

Returns
center_location – structured array of length 1 dtype includes (east, north, zone, lat, lon)

Return type
np.ndarray

property east

property elev

get_station_locations(*input_list*)

get station locations from a list of edi files

Parameters
input_list (*list*) – list of edi file names, or mt_objects

Return type
* fills station_locations array

property lat

property lon

property model_epsg

property model_utm_zone

property north

property rel_east

property rel_elev

property rel_north

rotate_stations(*rotation_angle*)

Rotate stations assuming N is 0

Parameters
rotation_angle (*float*) – angle in degrees assuming N is 0

Returns
because you will still need the original locations for plotting later.

Return type
* refils rel_east and rel_north in station_locations. Does this

property station

to_csv(*csv_fn*, *epsg=None*, *default_epsg=4326*, *geometry=False*)

Write a shape file of the station locations using geopandas which only takes in epsg numbers

Parameters

- **shp_fn** (*string*) – full path to new shapefile
- **epsg** (*integer, defaults to None*) – EPSG number to project to
- **default_epsg** – the default EPSG number that the stations are

referenced to :type default_epsg: integer, defaults to 4326

to_geopd(*epsg=None*, *default_epsg=4326*)

create a geopandas dataframe

Parameters

- **epsg** (*integer, defaults to None*) – EPSG number to project to
- **default_epsg** – the default EPSG number that the stations are

referenced to :type default_epsg: integer, defaults to 4326

to_shp(*shp_fn*, *epsg=None*, *default_epsg=4326*)

Write a shape file of the station locations using geopandas which only takes in epsg numbers

Parameters

- **shp_fn** (*string*) – full path to new shapefile
- **epsg** (*integer, defaults to None*) – EPSG number to project to
- **default_epsg** – the default EPSG number that the stations are

referenced to :type default_epsg: integer, defaults to 4326

property utm_zone

Module contents

class mtpy.modeling.modem.**ControlFwd**(***kwargs*)

Bases: object

read and write control file for

This file controls how the inversion starts and how it is run

property control_fn

read_control_file(*control_fn=None*)

read in a control file

write_control_file(*control_fn=None*, *save_path=None*, *fn_basename=None*)

write control file

Arguments:

control_fn
[string] full path to save control file to *default* is save_path/fn_basename

save_path
[string] directory path to save control file to *default* is cwd

fn_basename
[string] basename of control file *default* is control.inv

class mtpy.modeling.modem.**ControlInv**(***kwargs*)

Bases: object

read and write control file for how the inversion starts and how it is run

property control_fn

read_control_file(*control_fn=None*)

read in a control file

write_control_file(*control_fn=None*, *save_path=None*, *fn_basename=None*)

write control file

Arguments:

control_fn
[string] full path to save control file to *default* is save_path/fn_basename

save_path
[string] directory path to save control file to *default* is cwd

fn_basename
[string] basename of control file *default* is control.inv

class mtpy.modeling.modem.**Covariance**(*grid_dimensions=None*, ***kwargs*)

Bases: `object`

read and write covariance files

property cov_fn

get_parameters()

read_cov_file(*cov_fn*)

read a covariance file

write_cov_vtk_file(*cov_vtk_fn*, *model_fn=None*, *grid_east=None*, *grid_north=None*, *grid_z=None*)

write a vtk file of the covariance to match things up

write_covariance_file(*cov_fn=None*, *save_path=None*, *fn_basename=None*, *res_model=None*, *sea_water=0.3*, *air=1000000000000.0*)

write a covariance file

class mtpy.modeling.modem.**Data**(*dataframe=None*, *center_point=None*, ***kwargs*)

Bases: `object`

Data will read and write .dat files for ModEM and convert a WS data file to ModEM format.

..note: :: the data is interpolated onto the given periods such that all
stations invert for the same periods. The interpolation is a linear interpolation of each of the real and imaginary parts of the impedance tensor and induction tensor. See mtpy.core.mt.MT.interpolate for more details

Parameters
edi_list – list of edi files to read

Attributes	Description
_dtype	internal variable defining the data type of data_array

continues on next page

Table 3 – continued from previous page

Attributes	Description
_logger	python logging object that put messages in logging format defined in logging configure file, see MtPyLog for more information
_t_shape	internal variable defining shape of tipper array in _dtype
_z_shape	internal variable defining shape of Z array in _dtype
center_position	(east, north, evel) for center point of station array. All stations are relative to this location for plotting purposes.
comp_index_dict	dictionary for index values of component of Z and T
station_locations	Stations object
data_array	numpy.ndarray (num_stations) structured to store data. keys are: • station –> station name • lat –> latitude in decimal degrees • lon –> longitude in decimal degrees • elev –> elevation (m) • **rel_east – > relative east location to** center_position (m) • **rel_north –> relative north location to** center_position (m) • east –> UTM east (m) • north –> UTM north (m) • zone –> UTM zone • **z –> impedance tensor array with shape** (num_freq, 2, 2) • **z_err –> impedance tensor error array with** shape (num_freq, 2, 2) • **tip –> Tipper array with shape** (num_freq, 1, 2) • **tipperr –> Tipper array with shape** (num_freq, 1, 2)
data_fn	full path to data file
data_period_list	period list from all the data
edi_list	list of full paths to edi files
error_type_tipper	['abs' | 'floor'] *default* is 'abs'

continues on next page

Table 3 – continued from previous page

Attributes	Description
error_type_z	['egbert' | 'mean_od' | 'eigen' | 'median'] *default* is 'egbert_floor' • add '_floor' to any of the above to set the error as an error floor, otherwise all components are give weighted the same • **'egbert' sets error to** error_value_z * sqrt(abs(zxy*zyx)) • **'mean_od' sets error to** error_value_z * mean([Zxy, Zyx]) (non zeros) • **'eigen' sets error to** error_value_z * eigenvalues(Z[ii]) • **'median' sets error to** error_value_z * median([Zxx, Zxy, Zyx, Zyy]) (non zeros) A 2x2 numpy array of error_type_z can be specified to explicitly set the error_type_z for each component.
error_value_z	percentage to multiply Z by to set error *default* is 5 for 5% of Z as error A 2x2 numpy array of values can be specified to explicitly set the error_value_z for each component.
error_value_tipper	absolute error between 0 and 1.
fn_basename	basename of data file. *default* is 'ModEM_Data.dat'
formatting	['1' | '2'], format of the output data file, *default* is '1'
header_strings	strings for header of data file following the format outlined in the ModEM documentation
inv_comp_dict	dictionary of inversion components
inv_mode	**inversion mode, options are: *default* is '1'** • **'1' –> for 'Full_Impedance' and** 'Full_Vertical_Components' • '2' –> 'Full_Impedance' • **'3' –> 'Off_Diagonal_Impedance' and** 'Full_Vertical_Components' • '4' –> 'Off_Diagonal_Impedance' • '5' –> 'Full_Vertical_Components' • '6' –> 'Full_Interstation_TF' • '7' –> 'Off_Diagonal_Rho_Phase'
inv_mode_dict	dictionary for inversion modes
max_num_periods	maximum number of periods
model_epsg	epsg code for model projection, provide this to project model to non-utm coordinates. Find the epsg code for your projection on http://spatialreference.org/ref/ or google search epsg "your projection"
model_utm_zone	alternative to model_epsg, choose a utm zone to project all sites to (e.g. '55S')
mt_dict	dictionary of mtpy.core.mt.MT objects with keys being station names

continues on next page

Table 3 – continued from previous page

Attributes	Description
period_buffer	float or int if specified, apply a buffer so that interpolation doesn't stretch too far over periods
period_dict	dictionary of period index for period_list
period_list	list of periods to invert for
period_max	maximum value of period to invert for
period_min	minimum value of period to invert for
period_buffer	**buffer so that interpolation doesn't stretch too far** over periods. Provide a float or integer factor, greater than which interpolation will not stretch. e.g. 1.5 means only interpolate to a maximum of 1.5 times each side of each frequency value
rotate_angle	Angle to rotate data to assuming 0 is N and E is 90
save_path	path to save data file to
units	[[V/m]/[T] | [mV/km]/[nT] | Ohm] units of Z *default* is [mV/km]/[nT]
wave_sign_impedance	[+ | -] sign of time dependent wave. *default* is '+' as positive downwards.
wave_sign_tipper	[+ | -] sign of time dependent wave. *default* is '+' as positive downwards.

Example 1 –> create inversion period list

```
>>> from pathlib import Path
>>> import mtpy.modeling.modem as modem
>>> edi_path = Path(r"/home/mt/edi_files")
>>> edi_list = list(edi_path.glob("*.edi"))
>>> md = modem.Data(edi_list, period_min=.1, period_max=300,        >>>
↪ ...               max_num_periods=12)
>>> md.write_data_file(save_path=r"/home/modem/inv1")
>>> md
```

Example 2 –> set inverions period list from data

```
>>> md = modem.Data(edi_list)
>>> #get period list from an .edi file
>>> inv_period_list = 1./md.mt_dict["mt01"].Z.freq
>>> #invert for every third period in inv_period_list
>>> inv_period_list = inv_period_list[np.arange(0, len(inv_period_list,
↪ 3))]
>>> md.period_list = inv_period_list
>>> md.write_data_file(save_path=r"/home/modem/inv1")
```

Example 3 –> change error values

```
>>> mdr.error_type = 'floor'
>>> mdr.error_floor = 10
>>> mdr.error_tipper = .03
>>> mdr.write_data_file(save_path=r"/home/modem/inv2")
```

Example 4 –> change inversion type

```
>>> mdr.inv_mode = '3'
>>> mdr.write_data_file(save_path=r"/home/modem/inv2")
```

Example 5 –> rotate data

```
>>> md.rotation_angle = 60
>>> md.write_data_file(save_path=r"/home/modem/Inv1")
>>> # or
>>> md.write_data_file(save_path=r"/home/modem/Inv1",
↪               rotation_angle=60)
```

property data_filename

property dataframe

fix_data_file(*fn=None*, *n=3*)

A newer compiled version of Modem outputs slightly different headers This aims to convert that into the older format

Parameters

- **fn** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **n** (*TYPE, optional*) – DESCRIPTION, defaults to 3

Returns
DESCRIPTION

Return type
TYPE

get_n_stations(*mode*)

property model_parameters

property n_periods

property period

read_data_file(*data_fn*)

Parameters
data_fn (*string or Path*) – full path to data file name

Raises
ValueError – If cannot compute component

Fills attributes:

- data_array
- period_list
- mt_dict

```
>>> md = Data()
>>> md.read_data_file(r"/home/modem_data.dat")
>>> md
ModEM Data Object:
    Number of stations: 169
    Number of periods:  22
    Period range:
            Min: 0.01 s
            Max: 15230.2 s
    Rotation angle:     0.0
    Data center:
             latitude:  39.6351 deg
             longitude: -119.8039 deg
             Elevation: 0.0 m
             Easting:   259368.9746 m
             Northing:  4391021.1981 m
             UTM zone:  11S
    Model EPSG:         None
    Model UTM zone:     None
    Impedance data:     True
    Tipper data:        True
```

write_data_file(*file_name=None*, *save_path=None*, *fn_basename=None*, *elevation=False*)

Parameters

- **save_path** (*string or Path, optional*) – full directory to save file to, defaults to None
- **fn_basename** (*string, optional*) – Basename of the saved file, defaults to None
- **elevation** (*boolean, optional*) – If True adds in elevation from 'rel_elev' column in data array, defaults to False

Raises

- **NotImplementedError** – If the inversion mode is not supported
- **ValueError** – `mtpy.utils.exceptions.ValueError` if a parameter

is missing :return: full path to data file :rtype: Path

```
>>> from pathlib import Path
>>> import mtpy.modeling.modem as modem
>>> edi_path = Path(r"/home/mt/edi_files")
>>> edi_list = list(edi_path.glob("*.ed"))
>>> md = modem.Data(edi_list, period_min=.1, period_max=300,            >>> ...
↪             max_num_periods=12)
>>> md.write_data_file(save_path=r"/home/modem/inv1")
/home/modem/inv1/ModemDataFile.dat
```

exception `mtpy.modeling.modem.`**DataError**

Bases: *ModEMError*

Raise for ModEM Data class specific exceptions

class `mtpy.modeling.modem.`**ModEMConfig**(***kwargs*)

Bases: `object`

read and write configuration files for how each inversion is run

add_dict(*fn=None*, *obj=None*)

add dictionary based on file name or object

write_config_file(*save_dir=None*, *config_fn_basename='ModEM_inv.cfg'*)

write a config file based on provided information

exception mtpy.modeling.modem.**ModEMError**

Bases: Exception

class mtpy.modeling.modem.**Model**(*station_locations=None*, *center_point=None*, ***kwargs*)

Bases: object

make and read a FE mesh grid

The mesh assumes the coordinate system where:
x == North y == East z == + down

All dimensions are in meters.

The mesh is created by first making a regular grid around the station area, then padding cells are added that exponentially increase to the given extensions. Depth cell increase on a log10 scale to the desired depth, then padding cells are added that increase exponentially.

Parameters
****station_object**** (*mtpy.modeling.modem.Stations object*) –

See also:

mtpy.modeling.modem.Stations

Examples

Example 1 –> create mesh first then data file

```
>>> import mtpy.modeling.modem as modem
>>> import os
>>> # 1) make a list of all .edi files that will be inverted for
>>> edi_path = r"/home/EDI_Files"
>>> edi_list = [os.path.join(edi_path, edi)
```

for edi in os.listdir(edi_path)

```
>>> ...          if edi.find('.edi') > 0]
>>> # 2) Make a Stations object
>>> stations_obj = modem.Stations()
>>> stations_obj.get_station_locations_from_edi(edi_list)
>>> # 3) make a grid from the stations themselves with 200m cell␣
↪spacing
>>> mmesh = modem.Model(station_obj)
>>> # change cell sizes
>>> mmesh.cell_size_east = 200,
>>> mmesh.cell_size_north = 200
>>> mmesh.ns_ext = 300000 # north-south extension
>>> mmesh.ew_ext = 200000 # east-west extension of model
```

(continues on next page)

(continued from previous page)

```
>>> mmesh.make_mesh()
>>> # check to see if the mesh is what you think it should be
>>> msmesh.plot_mesh()
>>> # all is good write the mesh file
>>> msmesh.write_model_file(save_path=r"/home/modem/Inv1")
>>> # create data file
>>> md = modem.Data(edi_list, station_locations=mmesh.station_
↪locations)
>>> md.write_data_file(save_path=r"/home/modem/Inv1")
```

Example 2 –> Rotate Mesh

```
>>> mmesh.mesh_rotation_angle = 60
>>> mmesh.make_mesh()
```

Note: ModEM assumes all coordinates are relative to North and East, and does not accommodate mesh rotations, therefore, here the rotation is of the stations, which essentially does the same thing. You will need to rotate you data to align with the 'new' coordinate system.

Attributes	Description
_logger	python logging object that put messages in logging format defined in logging configure file, see MtPyLog more information
cell_number_ew	optional for user to specify the total number of sells on the east-west direction. *default* is None
cell_number_ns	optional for user to specify the total number of sells on the north-south direction. *default* is None
cell_size_east	mesh block width in east direction *default* is 500
cell_size_north	mesh block width in north direction *default* is 500
grid_center	center of the mesh grid
grid_east	overall distance of grid nodes in east direction
grid_north	overall distance of grid nodes in north direction
grid_z	overall distance of grid nodes in z direction
model_fn	full path to initial file name
model_fn_basename	default name for the model file name
n_air_layers	number of air layers in the model. *default* is 0
n_layers	total number of vertical layers in model
nodes_east	relative distance between nodes in east direction
nodes_north	relative distance between nodes in north direction
nodes_z	relative distance between nodes in east direction
pad_east	number of cells for padding on E and W sides *default* is 7
pad_north	number of cells for padding on S and N sides *default* is 7
pad_num	number of cells with cell_size with outside of station area. *default* is 3
pad_method	method to use to create padding: extent1, extent2 - calculate based on ew_ext and ns_ext stretch - calculate based on pad_stretch factors

continues on next page

Table 4 – continued from previous page

Attributes	Description
pad_stretch_h	multiplicative number for padding in horizontal direction.
pad_stretch_v	padding cells N & S will be pad_root_north**(x)
pad_z	number of cells for padding at bottom *default* is 4
ew_ext	E-W extension of model in meters
ns_ext	N-S extension of model in meters
res_scale	**scaling method of res, supports** 'loge' - for log e format 'log' or 'log10' - for log with base 10 'linear' - linear scale *default* is 'loge'
res_list	list of resistivity values for starting model
res_model	starting resistivity model
res_initial_value	resistivity initial value for the resistivity model *default* is 100
mesh_rotation_angle	Angle to rotate the grid to. Angle is measured positve clockwise assuming North is 0 and east is 90. *default* is None
save_path	path to save file to
sea_level	sea level in grid_z coordinates. *default* is 0
station_locations	location of stations
title	title in initial file
z1_layer	first layer thickness
z_bottom	absolute bottom of the model *default* is 300,000
z_target_depth	Depth of deepest target, *default* is 50,000

add_layers_to_mesh(*n_add_layers=None*, *layer_thickness=None*, *where='top'*)

Function to add constant thickness layers to the top or bottom of mesh. Note: It is assumed these layers are added before the topography. If you want to add topography layers, use function add_topography_to_model

Parameters

- **n_add_layers** – integer, number of layers to add
- **layer_thickness** – real value or list/array. Thickness of layers, defaults to z1 layer. Can provide a single value or a list/array containing multiple layer thicknesses.
- **where** – where to add, top or bottom

add_topography_from_data(*interp_method='nearest'*, *air_resistivity=1000000000000.0*, *topography_buffer=None*, *airlayer_type='log_up'*)

Wrapper around add_topography_to_model that allows creating a surface model from EDI data. The Data grid and station elevations will be used to make a 'surface' tuple that will be passed to add_topography_to_model so a surface model can be interpolated from it.

The surface tuple is of format (lon, lat, elev) containing station locations.

Parameters

- **data_object** (*mtpy.modeling.ModEM.data.Data*) – A ModEm data object that has been filled with data from EDI files.
- **interp_method** (*str, optional*) – Same as add_topography_to_model.
- **air_resistivity** (*float, optional*) – Same as add_topography_to_model.

- **topography_buffer** (*float*) – Same as add_topography_to_model.
- **airlayer_type** (*str, optional*) – Same as add_topography_to_model.

add_topography_to_model(*topography_file=None*, *surface=None*, *topography_array=None*, *interp_method='nearest'*, *air_resistivity=1000000000000.0*, *topography_buffer=None*, *airlayer_type='log_up'*, *max_elev=None*, *shift_east=0*, *shift_north=0*)

if air_layers is non-zero, will add topo: read in topograph file, make a surface model.

Call project_stations_on_topography in the end, which will re-write the .dat file.

If n_airlayers is zero, then cannot add topo data, only bathymetry is needed.

Parameters

- **topography_file** – file containing topography (arcgis ascii grid)
- **topography_array** – alternative to topography_file - array of elevation values on model grid
- **interp_method** – interpolation method for topography, 'nearest', 'linear', or 'cubic'
- **air_resistivity** – resistivity value to assign to air
- **topography_buffer** – buffer around stations to calculate minimum and maximum topography value to use for meshing
- **airlayer_type** – how to set air layer thickness - options are 'constant' for constant air layer thickness, or 'log', for logarithmically increasing air layer thickness upward

assign_resistivity_from_surface_data(*top_surface*, *bottom_surface*, *resistivity_value*)

assign resistivity value to all points above or below a surface requires the surface_dict attribute to exist and contain data for surface key (can get this information from ascii file using project_surface)

inputs surface_name = name of surface (must correspond to key in surface_dict) resistivity_value = value to assign where = 'above' or 'below' - assign resistivity above or below the

surface

interpolate_elevation(*surface_file=None*, *surface=None*, *get_surface_name=False*, *method='nearest'*, *fast=True*, *shift_north=0*, *shift_east=0*)

project a surface to the model grid and add resulting elevation data to a dictionary called surface_dict. Assumes the surface is in lat/long coordinates (wgs84)

returns nothing returned, but surface data are added to surface_dict under the key given by surface_name.

inputs choose to provide either surface_file (path to file) or surface (tuple). If both are provided then surface tuple takes priority.

surface elevations are positive up, and relative to sea level. surface file format is:

ncols 3601 nrows 3601 xllcorner -119.00013888889 (longitude of lower left) yllcorner 36.999861111111 (latitude of lower left) cellsize 0.00027777777777778 NODATA_value -9999 elevation data W –> E N | V S

Alternatively, provide a tuple with: (lon,lat,elevation) where elevation is a 2D array (shape (ny,nx)) containing elevation points (order S -> N, W -> E) and lon, lat are either 1D arrays containing list of longitudes and latitudes (in the case of a regular grid) or 2D arrays with same shape as elevation array containing longitude and latitude of each point.

other inputs: surface_epsg = epsg number of input surface, default is 4326 for lat/lon(wgs84) method = interpolation method. Default is 'nearest', if model grid is dense compared to surface points then choose 'linear' or 'cubic'

make_mesh(*verbose=True*)

create finite element mesh according to user-input parameters.

The mesh is built by:

1. Making a regular grid within the station area.
2. Adding pad_num of cell_width cells outside of station area
3. Adding padding cells to given extension and number of padding cells.
4. Making vertical cells starting with z1_layer increasing logarithmically (base 10) to z_target_depth and num_layers.
5. Add vertical padding cells to desired extension.
6. Check to make sure none of the stations lie on a node. If they do then move the node by .02*cell_width

make_z_mesh(*n_layers=None*)

new version of make_z_mesh. make_z_mesh and M

property model_epsg

property model_fn

property model_parameters

get important model parameters to write to a file for documentation later.

property nodes_east

property nodes_north

property nodes_z

property plot_east

plot_mesh(***kwargs*)

Plot model mesh

Parameters

plot_topography (*TYPE, optional*) – DESCRIPTION, defaults to False

Returns

DESCRIPTION

Return type

TYPE

property plot_north

property plot_z

read_gocad_sgrid_file(*sgrid_header_file, air_resistivity=1e+39, sea_resistivity=0.3, sgrid_positive_up=True*)

read a gocad sgrid file and put this info into a ModEM file. Note: can only deal with grids oriented N-S or E-W at this stage, with orthogonal coordinates

read_model_file(*model_fn=None*)

read an initial file and return the pertinent information including grid positions in coordinates relative to the center point (0,0) and starting model.

Note that the way the model file is output, it seems is that the blocks are setup as

ModEM: WS: ———- —– 0—–> N_north 0——–>N_east | | | | V V N_east N_north

Arguments:

model_fn : full path to initializing file.

Outputs:

nodes_north
: [np.array(nx)] array of nodes in S –> N direction

nodes_east
: [np.array(ny)] array of nodes in the W –> E direction

nodes_z
: [np.array(nz)] array of nodes in vertical direction positive downwards

res_model
: [dictionary] dictionary of the starting model with keys as layers

res_list
: [list] list of resistivity values in the model

title
: [string] title string

property save_path

write_geosoft_xyz(*save_fn*, *c_east=0*, *c_north=0*, *c_z=0*, *pad_north=0*, *pad_east=0*, *pad_z=0*)

Write an XYZ file readable by Geosoft

All input units are in meters.

Parameters

- **save_fn** (*string or Path*) – full path to save file to
- **c_east** (*float, optional*) – center point in the east direction, defaults to 0
- **c_north** (*float, optional*) – center point in the north direction, defaults to 0
- **c_z** (*float, optional*) – center point elevation, defaults to 0
- **pad_north** (*int, optional*) – number of cells to cut from the north-south edges, defaults to 0
- **pad_east** (*int, optional*) – number of cells to cut from the east-west edges, defaults to 0
- **pad_z** (*int, optional*) – number of cells to cut from the bottom, defaults to 0

write_gocad_sgrid_file(*fn=None*, *origin=[0, 0, 0]*, *clip=0*, *no_data_value=-99999*)

write a model to gocad sgrid

optional inputs:

fn = filename to save to. File extension ('.sg') will be appended.
default is the model name with extension removed

origin = real world [x,y,z] location of zero point in model grid clip = how much padding to clip off the edge of the model for export,

provide one integer value or list of 3 integers for x,y,z directions

no_data_value = no data value to put in sgrid

write_model_file(***kwargs*)

will write an initial file for ModEM.

Note that x is assumed to be S –> N, y is assumed to be W –> E and z is positive downwards. This means that index [0, 0, 0] is the southwest corner of the first layer. Therefore if you build a model by hand the layer block will look as it should in map view.

Also, the xgrid, ygrid and zgrid are assumed to be the relative distance between neighboring nodes. This is needed because wsinv3d builds the model from the bottom SW corner assuming the cell width from the init file.

Key Word Arguments:

nodes_north
[np.array(nx)] block dimensions (m) in the N-S direction. **Note** that the code reads the grid assuming that index=0 is the southern most point.

nodes_east
[np.array(ny)] block dimensions (m) in the E-W direction. **Note** that the code reads in the grid assuming that index=0 is the western most point.

nodes_z
[np.array(nz)] block dimensions (m) in the vertical direction. This is positive downwards.

save_path
[string] Path to where the initial file will be saved to save_path/model_fn_basename

model_fn_basename
[string] basename to save file to *default* is ModEM_Model.ws file is saved at save_path/model_fn_basename

title
[string] Title that goes into the first line *default* is Model File written by MTpy.modeling.modem

res_model
[np.array((nx,ny,nz))] Prior resistivity model.

Note: again that the modeling code

assumes that the first row it reads in is the southern most row and the first column it reads in is the western most column. Similarly, the first plane it reads in is the Earth's surface.

res_starting_value
[float] starting model resistivity value, assumes a half space in Ohm-m *default* is 100 Ohm-m

res_scale
[['loge' | 'log' | 'log10' | 'linear']] scale of resistivity. In the ModEM code it converts everything to Loge, *default* is 'loge'

write_out_file(*save_fn*, *geographic_east*, *geographic_north*, *geographic_elevation*)

will write an .out file for LeapFrog.

Note that y is assumed to be S –> N, e is assumed to be W –> E and z is positive upwards. This means that index [0, 0, 0] is the southwest corner of the first layer.

Parameters

- **save_fn** (*string or Path*) – full path to save file to
- **geographic_east** (*float*) – geographic center in easting (meters)
- **geographic_north** (*float*) – geographic center in northing (meters)
- **geographic_elevation** (*float*) – elevation of geographic center (meters)

Returns
DESCRIPTION

Return type
TYPE

write_ubc_files(*basename*, *c_east=0*, *c_north=0*, *c_z=0*)

Write a UBC .msh and .mod file

Parameters

- **save_fn** (*TYPE*) – DESCRIPTION
- **c_east** (*TYPE, optional*) – DESCRIPTION, defaults to 0
- **c_north** (*TYPE, optional*) – DESCRIPTION, defaults to 0
- **c_z** (*TYPE, optional*) – DESCRIPTION, defaults to 0

Returns
DESCRIPTION

Return type
TYPE

Note: not complete yet.

write_vtk_file(*vtk_save_path=None*, *vtk_fn_basename='ModEM_model_res'*, *shift_east=0*, *shift_north=0*, *shift_z=0*, *units='km'*, *coordinate_system='nez+'*, *label='resistivity'*)

Write a VTK file to plot in 3D rendering programs like Paraview

Parameters

- **vtk_save_path** (*string or Path, optional*) – directory to save vtk file to, defaults to None
- **vtk_fn_basename** – filename basename of vtk file, note that .vtr

extension is automatically added, defaults to "ModEM_stations" :type vtk_fn_basename: string, optional :type geographic: boolean, optional :param shift_east: shift in east directions in meters, defaults to 0 :type shift_east: float, optional :param shift_north: shift in north direction in meters, defaults to 0 :type shift_north: float, optional :param shift_z: shift in elevation + down in meters, defaults to 0 :type shift_z: float, optional :param units: Units of the spatial grid [km | m | ft], defaults to "km" :type units: string, optional :type : string :param coordinate_system: coordinate system for the station, either the normal MT right-hand coordinate system with z+ down or the sinister z- down [nez+ | enz-], defaults to nez+ :return: full path to VTK file :rtype: Path

Write VTK file >>> model.write_vtk_file(vtk_fn_basename="modem_model")

Write VTK file in geographic coordinates with z+ up >>> model.write_vtk_station_file(vtk_fn_basename="modem_model", >>> … coordinate_system='enz-')

`write_xyres`(*save_path=None*, *location_type='EN'*, *origin=[0, 0]*, *model_epsg=None*, *depth_index='all'*, *outfile_basename='DepthSlice'*, *log_res=False*, *clip=[0, 0]*)

write files containing depth slice data (x, y, res for each depth)

origin = x,y coordinate of zero point of ModEM_grid, or name of file
containing this info (full path or relative to model files)

save_path = path to save to, default is the model object save path location_type = 'EN' or 'LL' xy points saved as eastings/northings or

longitude/latitude, if 'LL' need to also provide model_epsg

model_epsg = epsg number that was used to project the model outfile_basename = string for basename for saving the depth slices. log_res = True/False - option to save resistivity values as log10

instead of linear

clip = number of cells to clip on each of the east/west and north/south edges

`write_xyzres`(*savefile=None*, *location_type='EN'*, *origin=[0, 0]*, *model_epsg=None*, *log_res=False*, *model_utm_zone=None*, *clip=[0, 0]*)

save a model file as a space delimited x y z res file

`class mtpy.modeling.modem.Residual`(***kwargs*)

Bases: `Data`

class to contain residuals for each data point, and rms values for each station

Attributes/Key Words	Description
work_dir	
residual_fn	full path to data file
residual_array	numpy.ndarray (num_stations) structured to store data. keys are: • station –> station name • lat –> latitude in decimal degrees • lon –> longitude in decimal degrees • elev –> elevation (m) • **rel_east – > relative east location to** center_position (m) • **rel_north –> relative north location to** center_position (m) • east –> UTM east (m) • north –> UTM north (m) • zone –> UTM zone • **z –> impedance tensor residual (measured - modelled)** (num_freq, 2, 2) • **z_err –> impedance tensor error array with** shape (num_freq, 2, 2) • **tip –> Tipper residual (measured - modelled)** (num_freq, 1, 2) • **tipperr –> Tipper array with shape** (num_freq, 1, 2)
rms	
rms_array	numpy.ndarray structured to store station location values and rms. Keys are: • station –> station name • east –> UTM east (m) • north –> UTM north (m) • lat –> latitude in decimal degrees • lon –> longitude in decimal degrees • elev –> elevation (m) • zone –> UTM zone • **rel_east – > relative east location to** center_position (m) • **rel_north –> relative north location to** center_position (m) • **rms –> root-mean-square residual for each** station
rms_tip	
rms_z	

calculate_rms()

add columns for rms :return: DESCRIPTION :rtype: TYPE

plot_rms(***kwargs*)

plot RMS in different views

Parameters
**kwargs – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`plot_rms_per_period`(*plot_type='all'*, ***kwargs*)

Parameters
**kwargs – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`read_residual_file`(*residual_fn*)

Parameters
residual_fn (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

property rms_per_period_all
RMS per period

property rms_per_period_per_component
RMS per period by component

Returns
DESCRIPTION

Return type
TYPE

mtpy.modeling.occam2d package

Submodules

mtpy.modeling.occam2d.data module

Created on Tue Mar 7 19:01:14 2023

@author: jpeacock

class mtpy.modeling.occam2d.data.**Occam2DData**(*dataframe=None*, *center_point=None*, ***kwargs*)

Bases: `object`

Reads and writes data files and more.

Inherets Profile, so the intended use is to use Data to project stations onto a profile, then write the data file.

Model Modes	Description
1 or log_all	Log resistivity of TE and TM plus Tipper
2 or log_te_tip	Log resistivity of TE plus Tipper
3 or log_tm_tip	Log resistivity of TM plus Tipper
4 or log_te_tm	Log resistivity of TE and TM
5 or log_te	Log resistivity of TE
6 or log_tm	Log resistivity of TM
7 or all	TE, TM and Tipper
8 or te_tip	TE plus Tipper
9 or tm_tip	TM plus Tipper
10 or te_tm	TE and TM mode
11 or te	TE mode
12 or tm	TM mode
13 or tip	Only Tipper

Example Write Data File

```
>>> from mtpy.modeling.occam2d import Data
>>> occam_data_object = Data()
>>> occam_data_object.read_data_file(r"path/to/data/file.dat")
>>> occam_data_object.model_mode = 2
>>> occam_data_object.write_data_file(r"path/to/new/data/file_te.dat")
```

property data_filename

property dataframe

property frequencies

mask_from_datafile(*mask_datafn*)

reads a separate data file and applies mask from this data file. mask_datafn needs to have exactly the same frequencies, and station names must match exactly.

property n_data

property n_frequencies

property n_stations

property offsets

read_data_file(*data_fn=None*)

Read in an existing data file and populate appropriate attributes

- data
- data_list
- freq
- station_list
- station_locations

Arguments:

data_fn
[string] full path to data file *default* is None and set to save_path/fn_basename

Example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> ocd = occam2d.Data()
>>> ocd.read_data_file(r"/home/Occam2D/Line1/Inv1/Data.dat")
```

property stations

write_data_file(*data_fn=None*)

Write a data file.

Arguments:

data_fn
[string] full path to data file. *default* is save_path/fn_basename

If there data is None, then _fill_data is called to create a profile, rotate data and get all the necessary data. This way you can use write_data_file directly without going through the steps of projecting the stations, etc.

Example
:: >>> edipath = r"/home/mt/edi_files" >>> slst = ['mt{0:03}'.format(ss) for ss in range(1, 20)] >>> ocd = occam2d.Data(edi_path=edipath, station_list=slst) >>> ocd.save_path = r"/home/occam/line1/inv1" >>> ocd.write_data_file()

mtpy.modeling.occam2d.mesh module

Created on Tue Mar 7 18:02:57 2023

@author: jpeacock

class mtpy.modeling.occam2d.mesh.**Mesh**(*station_locations=None*, ***kwargs*)

Bases: `object`

deals only with the finite element mesh. Builds a finite element mesh based on given parameters defined below. The mesh reads in the station locations, finds the center and makes the relative location of the furthest left hand station 0. The mesh increases in depth logarithmically as required by the physics of MT. Also, the model extends horizontally and vertically with padding cells in order to fullfill the assumption of the forward operator that at the edges the structure is 1D. Stations are place on the horizontal nodes as required by Wannamaker's forward operator.

Mesh has the ability to create a mesh that incorporates topography given a elevation profile. It adds more cells to the mesh with thickness z1_layer. It then sets the values of the triangular elements according to the elevation value at that location. If the elevation covers less than 50% of the triangular cell, then the cell value is set to that of air

Note: Mesh is inhereted by Regularization, so the mesh can also be be built from there, same as the example below.

Arguments:

Key Words/Attrib	Description
air_key	letter associated with the value of air *default* is 0
air_value	value given to an air cell, *default* is 1E13
cell_width	width of cells with in station area in meters *default* is 100
elevation_profile	elevation profile along the profile line. given as np.ndarray(nx, 2), where the elements are x_location, elevation. If elevation profile is given add_elevation is called automatically. *default* is None
mesh_fn	full path to mesh file.
mesh_values	letter values of each triangular mesh element if the cell is free value is ?
n_layers	number of vertical layers in mesh *default* is 90
num_x_pad_(	number of horizontal padding cells outside the the station area that will increase in size by x_pad_multiplier. *default* is 7
num_x_pad_:	number of horizonal padding cells just outside the station area with width cell_width. This is to extend the station area if needed. *default* is 2
num_z_pad_(	number of vertical padding cells below z_target_depth down to z_bottom. *default* is 5
rel_station_lc	relative station locations within the mesh. The locations are relative to the center of the station area. *default* is None, filled later
save_path	full path to save mesh file to. *default* is current working directory.
station_location	location of stations in meters, can be on a relative grid or in UTM.
x_grid	location of horizontal grid nodes in meters
x_nodes	relative spacing between grid nodes
x_pad_multip	horizontal padding cells will increase by this multiple out to the edge of the grid. *default* is 1.5
z1_layer	thickness of the first layer in the model. Should be at least 1/4 of the first skin depth *default* is 10
z_bottom	bottom depth of the model (m). Needs to be large enough to be 1D at the edge. *default* is 200000.0
z_grid	location of vertical nodes in meters
z_nodes	relative distance between vertical nodes in meters
z_target_dept	depth to deepest target of interest. Below this depth cells will be padded to z_bottom

Methods	Description
add_elevation	adds elevation to the mesh given elevation profile.
build_mesh	builds the mesh given the attributes of Mesh. If elevation_profile is not None, add_elevation is called inside build_mesh
plot_mesh	plots the built mesh with station location.
read_mesh_fil	reads in an existing mesh file and populates the appropriate attributes.
write_mesh_fi	writes a mesh file to save_path

Example

```
>>> import mtpy.modeling.occam2d as occcam2d
>>> edipath = r"/home/mt/edi_files"
>>> slist = ['mt{0:03}'.format(ss) for ss in range(20)]
>>> ocd = occam2d.Data(edi_path=edipath, station_list=slist)
>>> ocd.save_path = r"/home/occam/Line1/Inv1"
>>> ocd.write_data_file()
>>> ocm = occam2d.Mesh(ocd.station_locations)
>>> # add in elevation
>>> ocm.elevation_profile = ocd.elevation_profile
>>> # change number of layers
>>> ocm.n_layers = 110
>>> # change cell width in station area
>>> ocm.cell_width = 200
>>> ocm.build_mesh()
>>> ocm.plot_mesh()
>>> ocm.save_path = ocd.save_path
>>> ocm.write_mesh_file()
```

add_elevation(*elevation_profile=None*)

the elevation model needs to be in relative coordinates and be a numpy.ndarray(2, num_elevation_points) where the first column is the horizontal location and the second column is the elevation at that location.

If you have a elevation model use Profile to project the elevation information onto the profile line

To build the elevation I'm going to add the elevation to the top of the model which will add cells to the mesh. there might be a better way to do this, but this is the first attempt. So I'm going to assume that the first layer of the mesh without elevation is the minimum elevation and blocks will be added to max elevation at an increment according to z1_layer

Note: the elevation model should be symmetrical ie, starting at the first station and ending on the last station, so for now any elevation outside the station area will be ignored and set to the elevation of the station at the extremities. This is not ideal but works for now.

Arguments:

elevation_profile
[np.ndarray(2, num_elev_points)]

- 1st row is for profile location
- 2nd row is for elevation values

Computes:

mesh_values
[mesh values, setting anything above topography] to the key for air, which for Occam is '0'

build_mesh()

Build the finite element mesh given the parameters defined by the attributes of Mesh. Computes relative station locations by finding the center of the station area and setting the middle to 0. Mesh blocks are built by calculating the distance between stations and putting evenly spaced blocks between the stations being close to cell_width. This places a horizontal node at the station location. If the spacing between stations is smaller than cell_width, a horizontal node is placed between the stations to be sure the model has room to change between the station.

If elevation_profile is given, add_elevation is called to add topography into the mesh.

Populates attributes:

- mesh_values
- rel_station_locations
- x_grid
- x_nodes
- z_grid
- z_nodes

Example
:: >>> import mtpy.modeling.occam2d as occcam2d >>> edipath = r"/home/mt/edi_files" >>> slist = ['mt{0:03}'.format(ss) for ss in range(20)] >>> ocd = occam2d.Data(edi_path=edipath, station_list=slist) >>> ocd.save_path = r"/home/occam/Line1/Inv1" >>> ocd.write_data_file() >>> ocm = occam2d.Mesh(ocd.station_locations) >>> # add in elevation >>> ocm.elevation_profile = ocd.elevation_profile >>> # change number of layers >>> ocm.n_layers = 110 >>> # change cell width in station area >>> ocm.cell_width = 200 >>> ocm.build_mesh()

plot_mesh(***kwargs*)

Plot built mesh with station locations.

Key Words	Description
depth_scale	['km' | 'm'] scale of mesh plot. *default* is 'km'
fig_dpi	dots-per-inch resolution of the figure *default* is 300
fig_num	number of the figure instance *default* is 'Mesh'
fig_size	size of figure in inches (width, height) *default* is [5, 5]
fs	size of font of axis tick labels, axis labels are fs+2. *default* is 6
ls	['-' | '.' | ':'] line style of mesh lines *default* is '-'
marker	marker of stations *default* is r"$lacktriangledown$"
ms	size of marker in points. *default* is 5
plot_triangles	['y' | 'n'] to plot mesh triangles. *default* is 'n'

read_mesh_file(*mesh_fn*)

reads an occam2d 2D mesh file

Arguments:

mesh_fn
[string] full path to mesh file

Populates:

x_grid : array of horizontal locations of nodes (m)

x_nodes: array of horizontal node relative distances
(column locations (m))

z_grid : array of vertical node locations (m)

z_nodes
[array of vertical nodes] (row locations(m))

mesh_values : np.array of free parameters

To do:

incorporate fixed values

Example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> mg = occam2d.Mesh()
>>> mg.mesh_fn = r"/home/mt/occam/line1/Occam2Dmesh"
>>> mg.read_mesh_file()
```

write_mesh_file(*save_path=None*, *basename='Occam2DMesh'*)
Write a finite element mesh file.

Calls build_mesh if it already has not been called.

Arguments:

save_path
[string] directory path or full path to save file

basename
[string] basename of mesh file. *default* is 'Occam2DMesh'

Returns:

mesh_fn
[string] full path to mesh file

example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> edi_path = r"/home/mt/edi_files"
>>> profile = occam2d.Profile(edi_path)
>>> profile.plot_profile()
>>> mesh = occam2d.Mesh(profile.station_locations)
>>> mesh.build_mesh()
>>> mesh.write_mesh_file(save_path=r"/home/occam2d/Inv1")
```

mtpy.modeling.occam2d.model module

Created on Tue Mar 7 19:11:57 2023

@author: jpeacock

class mtpy.modeling.occam2d.model.**Occam2DModel**(*iter_fn=None*, *model_fn=None*, *mesh_fn=None*, ***kwargs*)

Bases: *Startup*

Read .iter file output by Occam2d. Builds the resistivity model from mesh and regularization files found from the .iter file. The resistivity model is an array(x_nodes, z_nodes) set on a regular grid, and the values of the model response are filled in according to the regularization grid. This allows for faster plotting.

Inherets Startup because they are basically the same object.

Argument:

iter_fn
[string] full path to .iter file to read. *default* is None.

model_fn
[string] full path to regularization file. *default* is None and found directly from the .iter file. Only input if the regularization is different from the file that is in the .iter file.

mesh_fn
[string] full path to mesh file. *default* is None Found directly from the model_fn file. Only input if the mesh is different from the file that is in the model file.

Key Words/Attributes	Description
data_fn	full path to data file
iter_fn	full path to .iter file
mesh_fn	full path to mesh file
mesh_x	np.ndarray(x_nodes, z_nodes) mesh grid for plotting
mesh_z	np.ndarray(x_nodes, z_nodes) mesh grid for plotting
model_values	model values from startup file
plot_x	nodes of mesh in horizontal direction
plot_z	nodes of mesh in vertical direction
res_model	np.ndarray(x_nodes, z_nodes) resistivity model values in linear scale

Methods	Description
build_mode	get the resistivity model from the .iter file in a regular grid according to the mesh file with resistivity values according to the model file
read_iter_fi	read .iter file and fill appropriate attributes
write_iter_f	write an .iter file incase you want to set it as the starting model or a priori model

Example

:: >>> model = occam2D.Model(r"/home/occam/line1/inv1/test_01.iter") >>> model.build_model()

build_model()

build the model from the mesh, regularization grid and model file

read_iter_file(*iter_fn=None*)

Read an iteration file.

Arguments:

iter_fn

[string] full path to iteration file if iterpath=None. If iterpath is input then iterfn is just the name of the file without the full path.

Returns:

Example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> itfn = r"/home/Occam2D/Line1/Inv1/Test_15.iter"
>>> ocm = occam2d.Model(itfn)
>>> ocm.read_iter_file()
```

write_iter_file(*iter_fn=None*)

write an iteration file if you need to for some reason, same as startup file

mtpy.modeling.occam2d.profile module

mtpy.modeling.occam2d.regularization module

Created on Tue Mar 7 18:13:52 2023

@author: jpeacock

class `mtpy.modeling.occam2d.regularization.`**`Regularization`**(*station_locations=None*, ***kwargs*)

Bases: *Mesh*

Creates a regularization grid based on Mesh. Note that Mesh is inherited by Regularization, therefore the intended use is to build a mesh with the Regularization class.

The regularization grid is what Occam calculates the inverse model on. Setup is tricky and can be painful, as you can see it is not quite fully functional yet, as it cannot incorporate topography yet. It seems like you'd like to have the regularization setup so that your target depth is covered well, in that the regularization blocks to this depth are sufficiently small to resolve resistivity structure at that depth. Finally, you want the regularization to go to a half space at the bottom, basically one giant block.

Arguments:

station_locations
[np.ndarray(n_stations)] array of station locations along a profile line in meters.

Key Words/Attributes	Description
air_key	letter associated with the value of air *default* is 0
air_value	value given to an air cell, *default* is 1E13
binding_offset	offset from the right side of the furthest left hand model block in meters. The regularization grid is s
cell_width	width of cells with in station area in meters *default* is 100
description	description of the model for the model file. *default* is 'simple inversion'
elevation_profile	elevation profile along the profile line. given as np.ndarray(nx, 2), where the elements are x_location
mesh_fn	full path to mesh file.
mesh_values	letter values of each triangular mesh element if the cell is free value is ?
model_columns	
model_name	
model_rows	
min_block_width	[float] minimum model block width in meters, *default* is 2*cell_width
n_layers	number of vertical layers in mesh *default* is 90
num_free_param	[int] number of free parameters in the model. this is a tricky number to estimate apparently.
num_layers	[int] number of regularization layers.
num_x_pad_cells	number of horizontal padding cells outside the the station area that will increase in size by x_pad_mu
num_x_pad_small_cells	number of horizonal padding cells just outside the station area with width cell_width. This is to exter
num_z_pad_cells	number of vertical padding cells below z_target_depth down to z_bottom. *default* is 5
prejudice_fn	full path to prejudice file *default* is 'none'
reg_basename	basename of regularization file (model file) *default* is 'Occam2DModel'
reg_fn	full path to regularization file (model file) *default* is save_path/reg_basename
rel_station_locations	relative station locations within the mesh. The locations are relative to the center of the station area.
save_path	full path to save mesh and model file to. *default* is current working directory.
statics_fn	full path to static shift file Static shifts in occam may not work. *default* is 'none'
station_locations	location of stations in meters, can be on a relative grid or in UTM.
trigger	[float] multiplier to merge model blocks at depth. A higher number increases the number of model l

Table 5 – continued from previous page

Key Words/Attributes	Description
x_grid	location of horizontal grid nodes in meters
x_nodes	relative spacing between grid nodes
x_pad_multiplier	horizontal padding cells will increase by this multiple out to the edge of the grid. *default* is 1.5
z1_layer	thickness of the first layer in the model. Should be at least 1/4 of the first skin depth *default* is 10
z_bottom	bottom depth of the model (m). Needs to be large enough to be 1D at the edge. *default* is 200000.0
z_grid	location of vertical nodes in meters
z_nodes	relative distance between vertical nodes in meters
z_target_depth	depth to deepest target of interest. Below this depth cells will be padded to z_bottom

Note: regularization does not work with topography yet. Having problems calculating the number of free parameters.

Example

```
>>> edipath = r"/home/mt/edi_files"
>>> profile = occam2d.Profile(edi_path=edi_path)
>>> profile.generate_profile()
>>> reg = occam2d.Regularization(profile.station_locations)
>>> reg.build_mesh()
>>> reg.build_regularization()
>>> reg.save_path = r"/home/occam2d/Line1/Inv1"
>>> reg.write_regularization_file()
```

build_regularization()

Builds larger boxes around existing mesh blocks for the regularization. As the model deepens the regularization boxes get larger.

The regularization boxes are merged mesh cells as prescribed by the Occam method.

get_num_free_params()

estimate the number of free parameters in model mesh.

I'm assuming that if there are any fixed parameters in the block, then that model block is assumed to be fixed. Not sure if this is right cause there is no documentation.

DOES NOT WORK YET

read_regularization_file(*reg_fn*)

Read in a regularization file and populate attributes:

- binding_offset
- mesh_fn
- model_columns
- model_rows
- prejudice_fn
- statics_fn

write_regularization_file(*reg_fn=None*, *reg_basename=None*, *statics_fn='none'*, *prejudice_fn='none'*, *save_path=None*)

Write a regularization file for input into occam.

Calls build_regularization if build_regularization has not already been called.

if reg_fn is None, then file is written to save_path/reg_basename

Arguments:

reg_fn
[string] full path to regularization file. *default* is None and file will be written to save_path/reg_basename

reg_basename
[string] basename of regularization file

statics_fn
[string] full path to static shift file .. note:: static shift does not always work in

occam2d.exe

prejudice_fn
[string] full path to prejudice file

save_path
[string] path to save regularization file. *default* is current working directory

mtpy.modeling.occam2d.response module

mtpy.modeling.occam2d.startup module

Created on Tue Mar 7 18:24:58 2023

@author: jpeacock

`class mtpy.modeling.occam2d.startup.Startup(**kwargs)`

Bases: `object`

Reads and writes the startup file for Occam2D.

Note: Be sure to look at the Occam 2D documentation for description of all parameters

Key Words/Attributes	Description
data_fn	full path to data file
date_time	date and time the startup file was written
debug_level	[0 | 1 | 2] see occam documentation *default* is 1
description	brief description of inversion run *default* is 'startup created by mtpy'
diagonal_penalties	penalties on diagonal terms *default* is 0
format	Occam file format *default* is 'OCCAMITER_FLEX'
iteration	current iteration number *default* is 0
iterations_to_run	maximum number of iterations to run *default* is 20
lagrange_value	starting lagrange value *default* is 5
misfit_reached	[0 | 1] 0 if misfit has been reached, 1 if it has. *default* is 0
misfit_value	current misfit value. *default* is 1000
model_fn	full path to model file
model_limits	limits on model resistivity values *default* is None
model_value_step	limits on the step size of model values *default* is None
model_values	np.ndarray(num_free_params) of model values
param_count	number of free parameters in model
resistivity_start	starting resistivity value. If model_values is not given, then all values with in model_values array will be set to resistivity_start
roughness_type	[0 | 1 | 2] type of roughness *default* is 1
roughness_value	current roughness value. *default* is 1E10
save_path	directory path to save startup file to *default* is current working directory
startup_basename	basename of startup file name. *default* is Occam2DStartup
startup_fn	full path to startup file. *default* is save_path/startup_basename
stepsize_count	max number of iterations per step *default* is 8
target_misfit	target misfit value. *default* is 1.

Example

```
>>> startup = occam2d.Startup()
>>> startup.data_fn = ocd.data_fn
>>> startup.model_fn = profile.reg_fn
>>> startup.param_count = profile.num_free_params
>>> startup.save_path = r"/home/occam2d/Line1/Inv1"
```

write_startup_file(*startup_fn=None*, *save_path=None*, *startup_basename=None*)

Write a startup file based on the parameters of startup class. Default file name is save_path/startup_basename

Arguments:

startup_fn
[string] full path to startup file. *default* is None

save_path
[string] directory to save startup file. *default* is None

startup_basename
[string] basename of starup file. *default* is None

Module contents

class `mtpy.modeling.occam2d.`**`Mesh`**(*station_locations=None*, ***kwargs*)

Bases: `object`

deals only with the finite element mesh. Builds a finite element mesh based on given parameters defined below. The mesh reads in the station locations, finds the center and makes the relative location of the furthest left hand station 0. The mesh increases in depth logarithmically as required by the physics of MT. Also, the model extends horizontally and vertically with padding cells in order to fullfill the assumption of the forward operator that at the edges the structure is 1D. Stations are place on the horizontal nodes as required by Wannamaker's forward operator.

Mesh has the ability to create a mesh that incorporates topography given a elevation profile. It adds more cells to the mesh with thickness z1_layer. It then sets the values of the triangular elements according to the elevation value at that location. If the elevation covers less than 50% of the triangular cell, then the cell value is set to that of air

Note: Mesh is inhereted by Regularization, so the mesh can also be be built from there, same as the example below.

Arguments:

Key Words/Attrib	Description
air_key	letter associated with the value of air *default* is 0
air_value	value given to an air cell, *default* is 1E13
cell_width	width of cells with in station area in meters *default* is 100
elevation_profile	elevation profile along the profile line. given as np.ndarray(nx, 2), where the elements are x_location, elevation. If elevation profile is given add_elevation is called automatically. *default* is None
mesh_fn	full path to mesh file.
mesh_values	letter values of each triangular mesh element if the cell is free value is ?
n_layers	number of vertical layers in mesh *default* is 90
num_x_pad_	number of horizontal padding cells outside the the station area that will increase in size by x_pad_multiplier. *default* is 7
num_x_pad_	number of horizonal padding cells just outside the station area with width cell_width. This is to extend the station area if needed. *default* is 2
num_z_pad_	number of vertical padding cells below z_target_depth down to z_bottom. *default* is 5
rel_station_lc	relative station locations within the mesh. The locations are relative to the center of the station area. *default* is None, filled later
save_path	full path to save mesh file to. *default* is current working directory.
station_location	location of stations in meters, can be on a relative grid or in UTM.
x_grid	location of horizontal grid nodes in meters
x_nodes	relative spacing between grid nodes
x_pad_multip	horizontal padding cells will increase by this multiple out to the edge of the grid. *default* is 1.5
z1_layer	thickness of the first layer in the model. Should be at least 1/4 of the first skin depth *default* is 10
z_bottom	bottom depth of the model (m). Needs to be large enough to be 1D at the edge. *default* is 200000.0
z_grid	location of vertical nodes in meters
z_nodes	relative distance between vertical nodes in meters
z_target_dept	depth to deepest target of interest. Below this depth cells will be padded to z_bottom

Methods	Description
add_elevation	adds elevation to the mesh given elevation profile.
build_mesh	builds the mesh given the attributes of Mesh. If elevation_profile is not None, add_elevation is called inside build_mesh
plot_mesh	plots the built mesh with station location.
read_mesh_fil	reads in an existing mesh file and populates the appropriate attributes.
write_mesh_fi	writes a mesh file to save_path

Example

```
>>> import mtpy.modeling.occam2d as occcam2d
>>> edipath = r"/home/mt/edi_files"
>>> slist = ['mt{0:03}'.format(ss) for ss in range(20)]
>>> ocd = occam2d.Data(edi_path=edipath, station_list=slist)
>>> ocd.save_path = r"/home/occam/Line1/Inv1"
```

(continues on next page)

(continued from previous page)

```
>>> ocd.write_data_file()
>>> ocm = occam2d.Mesh(ocd.station_locations)
>>> # add in elevation
>>> ocm.elevation_profile = ocd.elevation_profile
>>> # change number of layers
>>> ocm.n_layers = 110
>>> # change cell width in station area
>>> ocm.cell_width = 200
>>> ocm.build_mesh()
>>> ocm.plot_mesh()
>>> ocm.save_path = ocd.save_path
>>> ocm.write_mesh_file()
```

add_elevation(*elevation_profile=None*)

the elevation model needs to be in relative coordinates and be a numpy.ndarray(2, num_elevation_points) where the first column is the horizontal location and the second column is the elevation at that location.

If you have a elevation model use Profile to project the elevation information onto the profile line

To build the elevation I'm going to add the elevation to the top of the model which will add cells to the mesh. there might be a better way to do this, but this is the first attempt. So I'm going to assume that the first layer of the mesh without elevation is the minimum elevation and blocks will be added to max elevation at an increment according to z1_layer

Note: the elevation model should be symmetrical ie, starting at the first station and ending on the last station, so for now any elevation outside the station area will be ignored and set to the elevation of the station at the extremities. This is not ideal but works for now.

Arguments:

elevation_profile
[np.ndarray(2, num_elev_points)]

- 1st row is for profile location
- 2nd row is for elevation values

Computes:

mesh_values
[mesh values, setting anything above topography] to the key for air, which for Occam is '0'

build_mesh()

Build the finite element mesh given the parameters defined by the attributes of Mesh. Computes relative station locations by finding the center of the station area and setting the middle to 0. Mesh blocks are built by calculating the distance between stations and putting evenly spaced blocks between the stations being close to cell_width. This places a horizontal node at the station location. If the spacing between stations is smaller than cell_width, a horizontal node is placed between the stations to be sure the model has room to change between the station.

If elevation_profile is given, add_elevation is called to add topography into the mesh.

Populates attributes:

- mesh_values
- rel_station_locations
- x_grid
- x_nodes
- z_grid
- z_nodes

Example

:: >>> import mtpy.modeling.occam2d as occcam2d >>> edipath = r"/home/mt/edi_files" >>> slist = ['mt{0:03}'.format(ss) for ss in range(20)] >>> ocd = occam2d.Data(edi_path=edipath, station_list=slist) >>> ocd.save_path = r"/home/occam/Line1/Inv1" >>> ocd.write_data_file() >>> ocm = occam2d.Mesh(ocd.station_locations) >>> # add in elevation >>> ocm.elevation_profile = ocd.elevation_profile >>> # change number of layers >>> ocm.n_layers = 110 >>> # change cell width in station area >>> ocm.cell_width = 200 >>> ocm.build_mesh()

`plot_mesh`(***kwargs*)

Plot built mesh with station locations.

Key Words	Description
depth_scale	['km' | 'm'] scale of mesh plot. *default* is 'km'
fig_dpi	dots-per-inch resolution of the figure *default* is 300
fig_num	number of the figure instance *default* is 'Mesh'
fig_size	size of figure in inches (width, height) *default* is [5, 5]
fs	size of font of axis tick labels, axis labels are fs+2. *default* is 6
ls	['-' | '.' | ':'] line style of mesh lines *default* is '-'
marker	marker of stations *default* is r"$lacktriangledown$"
ms	size of marker in points. *default* is 5
plot_triangles	['y' | 'n'] to plot mesh triangles. *default* is 'n'

`read_mesh_file`(*mesh_fn*)

reads an occam2d 2D mesh file

Arguments:

mesh_fn
[string] full path to mesh file

Populates:

x_grid : array of horizontal locations of nodes (m)

x_nodes: array of horizontal node relative distances
(column locations (m))

z_grid : array of vertical node locations (m)

z_nodes
[array of vertical nodes] (row locations(m))

mesh_values : np.array of free parameters

To do:

incorporate fixed values

Example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> mg = occam2d.Mesh()
>>> mg.mesh_fn = r"/home/mt/occam/line1/Occam2Dmesh"
>>> mg.read_mesh_file()
```

write_mesh_file(*save_path=None*, *basename='Occam2DMesh'*)

Write a finite element mesh file.

Calls build_mesh if it already has not been called.

Arguments:

save_path
[string] directory path or full path to save file

basename
[string] basename of mesh file. *default* is 'Occam2DMesh'

Returns:

mesh_fn
[string] full path to mesh file

example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> edi_path = r"/home/mt/edi_files"
>>> profile = occam2d.Profile(edi_path)
>>> profile.plot_profile()
>>> mesh = occam2d.Mesh(profile.station_locations)
>>> mesh.build_mesh()
>>> mesh.write_mesh_file(save_path=r"/home/occam2d/Inv1")
```

class `mtpy.modeling.occam2d.`**`Occam2DData`**(*dataframe=None*, *center_point=None*, ***kwargs*)

Bases: `object`

Reads and writes data files and more.

Inherets Profile, so the intended use is to use Data to project stations onto a profile, then write the data file.

Model Modes	Description
1 or log_all	Log resistivity of TE and TM plus Tipper
2 or log_te_tip	Log resistivity of TE plus Tipper
3 or log_tm_tip	Log resistivity of TM plus Tipper
4 or log_te_tm	Log resistivity of TE and TM
5 or log_te	Log resistivity of TE
6 or log_tm	Log resistivity of TM
7 or all	TE, TM and Tipper
8 or te_tip	TE plus Tipper
9 or tm_tip	TM plus Tipper
10 or te_tm	TE and TM mode
11 or te	TE mode
12 or tm	TM mode
13 or tip	Only Tipper

Example Write Data File

```
>>> from mtpy.modeling.occam2d import Data
>>> occam_data_object = Data()
>>> occam_data_object.read_data_file(r"path/to/data/file.dat")
>>> occam_data_object.model_mode = 2
>>> occam_data_object.write_data_file(r"path/to/new/data/file_te.dat")
```

property data_filename

property dataframe

property frequencies

mask_from_datafile(*mask_datafn*)

reads a separate data file and applies mask from this data file. mask_datafn needs to have exactly the same frequencies, and station names must match exactly.

property n_data

property n_frequencies

property n_stations

property offsets

read_data_file(*data_fn=None*)

Read in an existing data file and populate appropriate attributes

- data
- data_list
- freq

- station_list
- station_locations

Arguments:

data_fn
[string] full path to data file *default* is None and set to save_path/fn_basename

Example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> ocd = occam2d.Data()
>>> ocd.read_data_file(r"/home/Occam2D/Line1/Inv1/Data.dat")
```

property stations

write_data_file(*data_fn=None*)

Write a data file.

Arguments:

data_fn
[string] full path to data file. *default* is save_path/fn_basename

If there data is None, then _fill_data is called to create a profile, rotate data and get all the necessary data. This way you can use write_data_file directly without going through the steps of projecting the stations, etc.

Example
:: >>> edipath = r"/home/mt/edi_files" >>> slst = ['mt{0:03}'.format(ss) for ss in range(1, 20)] >>> ocd = occam2d.Data(edi_path=edipath, station_list=slst) >>> ocd.save_path = r"/home/occam/line1/inv1" >>> ocd.write_data_file()

class mtpy.modeling.occam2d.**Occam2DModel**(*iter_fn=None*, *model_fn=None*, *mesh_fn=None*, ***kwargs*)

Bases: *Startup*

Read .iter file output by Occam2d. Builds the resistivity model from mesh and regularization files found from the .iter file. The resistivity model is an array(x_nodes, z_nodes) set on a regular grid, and the values of the model response are filled in according to the regularization grid. This allows for faster plotting.

Inherets Startup because they are basically the same object.

Argument:

iter_fn
[string] full path to .iter file to read. *default* is None.

model_fn
[string] full path to regularization file. *default* is None and found directly from the .iter file. Only input if the regularization is different from the file that is in the .iter file.

mesh_fn
[string] full path to mesh file. *default* is None Found directly from the model_fn file. Only input if the mesh is different from the file that is in the model file.

Key Words/Attributes	Description
data_fn	full path to data file
iter_fn	full path to .iter file
mesh_fn	full path to mesh file
mesh_x	np.ndarray(x_nodes, z_nodes) mesh grid for plotting
mesh_z	np.ndarray(x_nodes, z_nodes) mesh grid for plotting
model_values	model values from startup file
plot_x	nodes of mesh in horizontal direction
plot_z	nodes of mesh in vertical direction
res_model	np.ndarray(x_nodes, z_nodes) resistivity model values in linear scale

Methods	Description
build_mode	get the resistivity model from the .iter file in a regular grid according to the mesh file with resistivity values according to the model file
read_iter_fi	read .iter file and fill appropriate attributes
write_iter_1	write an .iter file incase you want to set it as the starting model or a priori model

Example
:: >>> model = occam2D.Model(r"/home/occam/line1/inv1/test_01.iter") >>> model.build_model()

build_model()
build the model from the mesh, regularization grid and model file

read_iter_file(*iter_fn=None*)
Read an iteration file.

Arguments:

iter_fn
[string] full path to iteration file if iterpath=None. If iterpath is input then iterfn is just the name of the file without the full path.

Returns:

Example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> itfn = r"/home/Occam2D/Line1/Inv1/Test_15.iter"
>>> ocm = occam2d.Model(itfn)
>>> ocm.read_iter_file()
```

write_iter_file(*iter_fn=None*)

write an iteration file if you need to for some reason, same as startup file

class mtpy.modeling.occam2d.**Regularization**(*station_locations=None*, ***kwargs*)

Bases: *Mesh*

Creates a regularization grid based on Mesh. Note that Mesh is inherited by Regularization, therefore the intended use is to build a mesh with the Regularization class.

The regularization grid is what Occam calculates the inverse model on. Setup is tricky and can be painful, as you can see it is not quite fully functional yet, as it cannot incorporate topography yet. It seems like you'd like to have the regularization setup so that your target depth is covered well, in that the regularization blocks to this depth are sufficiently small to resolve resistivity structure at that depth. Finally, you want the regularization to go to a half space at the bottom, basically one giant block.

Arguments:

station_locations

[np.ndarray(n_stations)] array of station locations along a profile line in meters.

Key Words/Attributes	Description
air_key	letter associated with the value of air *default* is 0
air_value	value given to an air cell, *default* is 1E13
binding_offset	offset from the right side of the furthest left hand model block in meters. The regularization grid is s
cell_width	width of cells with in station area in meters *default* is 100
description	description of the model for the model file. *default* is 'simple inversion'
elevation_profile	elevation profile along the profile line. given as np.ndarray(nx, 2), where the elements are x_location
mesh_fn	full path to mesh file.
mesh_values	letter values of each triangular mesh element if the cell is free value is ?
model_columns	
model_name	
model_rows	
min_block_width	[float] minimum model block width in meters, *default* is 2*cell_width
n_layers	number of vertical layers in mesh *default* is 90
num_free_param	[int] number of free parameters in the model. this is a tricky number to estimate apparently.
num_layers	[int] number of regularization layers.
num_x_pad_cells	number of horizontal padding cells outside the the station area that will increase in size by x_pad_mu
num_x_pad_small_cells	number of horizonal padding cells just outside the station area with width cell_width. This is to exter
num_z_pad_cells	number of vertical padding cells below z_target_depth down to z_bottom. *default* is 5
prejudice_fn	full path to prejudice file *default* is 'none'
reg_basename	basename of regularization file (model file) *default* is 'Occam2DModel'
reg_fn	full path to regularization file (model file) *default* is save_path/reg_basename
rel_station_locations	relative station locations within the mesh. The locations are relative to the center of the station area.
save_path	full path to save mesh and model file to. *default* is current working directory.
statics_fn	full path to static shift file Static shifts in occam may not work. *default* is 'none'
station_locations	location of stations in meters, can be on a relative grid or in UTM.
trigger	[float] multiplier to merge model blocks at depth. A higher number increases the number of model
x_grid	location of horizontal grid nodes in meters
x_nodes	relative spacing between grid nodes
x_pad_multiplier	horizontal padding cells will increase by this multiple out to the edge of the grid. *default* is 1.5
z1_layer	thickness of the first layer in the model. Should be at least 1/4 of the first skin depth *default* is 10
z_bottom	bottom depth of the model (m). Needs to be large enough to be 1D at the edge. *default* is 200000.0

Table 6 – continued from previous page

Key Words/Attributes	Description
z_grid	location of vertical nodes in meters
z_nodes	relative distance between vertical nodes in meters
z_target_depth	depth to deepest target of interest. Below this depth cells will be padded to z_bottom

Note: regularization does not work with topography yet. Having problems calculating the number of free parameters.

Example

```
>>> edipath = r"/home/mt/edi_files"
>>> profile = occam2d.Profile(edi_path=edi_path)
>>> profile.generate_profile()
>>> reg = occam2d.Regularization(profile.station_locations)
>>> reg.build_mesh()
>>> reg.build_regularization()
>>> reg.save_path = r"/home/occam2d/Line1/Inv1"
>>> reg.write_regularization_file()
```

build_regularization()

Builds larger boxes around existing mesh blocks for the regularization. As the model deepens the regularization boxes get larger.

The regularization boxes are merged mesh cells as prescribed by the Occam method.

get_num_free_params()

estimate the number of free parameters in model mesh.

I'm assuming that if there are any fixed parameters in the block, then that model block is assumed to be fixed. Not sure if this is right cause there is no documentation.

DOES NOT WORK YET

read_regularization_file(*reg_fn*)

Read in a regularization file and populate attributes:

- binding_offset
- mesh_fn
- model_columns
- model_rows
- prejudice_fn
- statics_fn

write_regularization_file(*reg_fn=None*, *reg_basename=None*, *statics_fn='none'*, *prejudice_fn='none'*, *save_path=None*)

Write a regularization file for input into occam.

Calls build_regularization if build_regularization has not already been called.

if reg_fn is None, then file is written to save_path/reg_basename

Arguments:

reg_fn
[string] full path to regularization file. *default* is None and file will be written to save_path/reg_basename

reg_basename
[string] basename of regularization file

statics_fn
[string] full path to static shift file .. note:: static shift does not always work in

occam2d.exe

prejudice_fn
[string] full path to prejudice file

save_path
[string] path to save regularization file. *default* is current working directory

class `mtpy.modeling.occam2d.Startup`(***kwargs*)

Bases: `object`

Reads and writes the startup file for Occam2D.

Note: Be sure to look at the Occam 2D documentation for description of all parameters

Key Words/Attributes	Description
data_fn	full path to data file
date_time	date and time the startup file was written
debug_level	[0 | 1 | 2] see occam documentation *default* is 1
description	brief description of inversion run *default* is 'startup created by mtpy'
diagonal_penalties	penalties on diagonal terms *default* is 0
format	Occam file format *default* is 'OCCAMITER_FLEX'
iteration	current iteration number *default* is 0
iterations_to_run	maximum number of iterations to run *default* is 20
lagrange_value	starting lagrange value *default* is 5
misfit_reached	[0 | 1] 0 if misfit has been reached, 1 if it has. *default* is 0
misfit_value	current misfit value. *default* is 1000
model_fn	full path to model file
model_limits	limits on model resistivity values *default* is None
model_value_step	limits on the step size of model values *default* is None
model_values	np.ndarray(num_free_params) of model values
param_count	number of free parameters in model
resistivity_start	starting resistivity value. If model_values is not given, then all values with in model_values array will be set to resistivity_start
roughness_type	[0 | 1 | 2] type of roughness *default* is 1
roughness_value	current roughness value. *default* is 1E10
save_path	directory path to save startup file to *default* is current working directory
startup_basename	basename of startup file name. *default* is Occam2DStartup
startup_fn	full path to startup file. *default* is save_path/startup_basename
stepsize_count	max number of iterations per step *default* is 8
target_misfit	target misfit value. *default* is 1.

Example

```
>>> startup = occam2d.Startup()
>>> startup.data_fn = ocd.data_fn
>>> startup.model_fn = profile.reg_fn
>>> startup.param_count = profile.num_free_params
>>> startup.save_path = r"/home/occam2d/Line1/Inv1"
```

write_startup_file(*startup_fn=None*, *save_path=None*, *startup_basename=None*)

Write a startup file based on the parameters of startup class. Default file name is save_path/startup_basename

Arguments:

startup_fn
[string] full path to startup file. *default* is None

save_path
[string] directory to save startup file. *default* is None

startup_basename
[string] basename of starup file. *default* is None

mtpy.modeling.plots package

Submodules

mtpy.modeling.plots.plot_mesh module

Created on Fri Oct 14 08:37:48 2022

@author: jpeacock

`class mtpy.modeling.plots.plot_mesh.PlotMesh(`*model_obj*, ***kwargs*`)`

Bases: `PlotBase`

`plot()`

Plot the mesh to show model grid

Arguments:

z_limits
[tuple (zmin,zmax)] plot min and max distances in meters for the vertical direction. If None, the z_limits is set to the number of layers. Z is positive down *default* is None

mtpy.modeling.plots.plot_modem_rms module

Created on Wed Feb 17 10:57:29 2021

copyright
Jared Peacock (jpeacock@usgs.gov)

license
MIT

`class mtpy.modeling.plots.plot_modem_rms.PlotRMS(`*dataframe*, ***kwargs*`)`

Bases: `PlotBaseMaps`

`property dataframe`

`plot(`***kwargs*`)`

Parameters
`**kwargs` – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`print_suspect_stations`(*rms_threshold=4*)
print stations that are suspect :return: DESCRIPTION :rtype: TYPE

`property rms_array`
arrays for color maps

Returns
DESCRIPTION

Return type
TYPE

`property rms_cmap`

`property rms_per_period_all`
RMS per period

`property rms_per_station`
RMS per period

Module contents

`class mtpy.modeling.plots.PlotMesh`(*model_obj*, ***kwargs*)
Bases: `PlotBase`

`plot`()
Plot the mesh to show model grid

Arguments:

z_limits
[tuple (zmin,zmax)] plot min and max distances in meters for the vertical direction. If None, the z_limits is set to the number of layers. Z is positive down *default* is None

`class mtpy.modeling.plots.PlotRMS`(*dataframe*, ***kwargs*)
Bases: `PlotBaseMaps`

`property dataframe`

`plot`(***kwargs*)

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`print_suspect_stations`(*rms_threshold=4*)
print stations that are suspect :return: DESCRIPTION :rtype: TYPE

property rms_array

arrays for color maps

Returns
DESCRIPTION

Return type
TYPE

property rms_cmap

property rms_per_period_all

RMS per period

property rms_per_station

RMS per period

Submodules

mtpy.modeling.errors module

Created on Tue Oct 11 16:01:37 2022

@author: jpeacock

class `mtpy.modeling.errors.ModelErrors`(*data=None*, *measurement_error=None*, ***kwargs*)

Bases: `object`

compute_absolute_error()

Parameters

- **data** (*TYPE*) – DESCRIPTION
- **error_value** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

compute_arithmetic_mean_error()

error_value * (Zxy + Zyx) / 2

Parameters

- **data** (*TYPE*) – DESCRIPTION
- **error_value** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

compute_eigen_value_error()

error_value * eigen(data).mean()

Parameters

- **data** (*TYPE*) – DESCRIPTION
- **error_value** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

compute_error(*data=None*, *error_type=None*, *error_value=None*, *floor=None*)

Parameters

- **data** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **error_type** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **error_value** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **floor** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

compute_geometric_mean_error()

error_value * sqrt(Zxy * Zyx)

Parameters

- **data** (*TYPE*) – DESCRIPTION
- **error_value** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

compute_median_error()

median(array) * error_value

Parameters

- **array** (*TYPE*) – DESCRIPTION
- **error_value** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

compute_percent_error()

Percent error

Parameters

- **data** (*TYPE*) – DESCRIPTION
- **percent** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`compute_row_error()`

set zxx and zxy the same error and zyy and zyx the same error

Parameters

- `data` (*TYPE*) – DESCRIPTION
- `error_value` (*TYPE*) – DESCRIPTION
- `floor` (*TYPE, optional*) – DESCRIPTION, defaults to True

Returns
DESCRIPTION

Return type
TYPE

`property data`

`property error_parameters`

`property error_type`

`property error_value`

`property floor`

`mask_zeros(data)`

mask zeros

Parameters
`data` (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`property measurement_error`

`property mode`

`resize_output(error_array)`

resize the error estimtion to the same size as the input data

Parameters
`error_array` (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

set_floor(*error_array*)

Set error floor

Parameters

- **array** – DESCRIPTION
- **floor** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

use_measurement_error()

validate_array_shape(*data*)

Parameters
data (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

validate_percent(*value*)

Make sure the percent is a decimal

Parameters
value (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

mtpy.modeling.gocad module

Created on Fri Dec 09 15:50:53 2016

@author: Alison Kirkby

read and write gocad objects

class mtpy.modeling.gocad.**Sgrid**(***kwargs*)

Bases: object

class to read and write gocad sgrid files

need to provide: workdir = working directory fn = filename for the sgrid resistivity = 3d numpy array containing resistivity values, shape (ny,nx,nz) grid_xyz = tuple containing x,y,z locations of edges of cells for each

resistivity value. Each item in tuple has shape (ny+1,nx+1,nz+1)

read_sgrid_file(*headerfn=None*)

write_sgrid_file()

mtpy.modeling.mare2dem module

mtpy.modeling.mesh_tools module

Created on Wed Oct 25 09:35:31 2017

@author: Alison Kirkby

functions to assist with mesh generation

`mtpy.modeling.mesh_tools.`**`get_nearest_index`**`(`*array*, *value*`)`

Return the index of the nearest value to the provided value in an array:

inputs:
array = array or list of values value = target value

`mtpy.modeling.mesh_tools.`**`get_padding_cells`**`(`*cell_width*, *max_distance*, *num_cells*, *stretch*`)`

get padding cells, which are exponentially increasing to a given distance. Make sure that each cell is larger than the one previously.

Parameters

- **`**cell_width**`** (*`float`*) – width of grid cell (m)
- **`**max_distance**`** (*`float`*) – maximum distance the grid will extend (m)
- **`**num_cells**`** (*`int`*) – number of padding cells
- **`**stretch**`** (*`float`*) – base geometric factor

Returns
****padding**** – array of padding cells for one side

Return type
np.ndarray

`mtpy.modeling.mesh_tools.`**`get_padding_cells2`**`(`*cell_width*, *core_max*, *max_distance*, *num_cells*`)`

get padding cells, which are exponentially increasing to a given distance. Make sure that each cell is larger than the one previously.

`mtpy.modeling.mesh_tools.`**`get_padding_from_stretch`**`(`*cell_width*, *pad_stretch*, *num_cells*`)`

get padding cells using pad stretch factor

`mtpy.modeling.mesh_tools.`**`get_rounding`**`(`*cell_width*`)`

Get the rounding number given the cell width. Will be one significant number less than the cell width. This reduces weird looking meshes.

Parameters
`cell_width` (*`float`*) – Width of mesh cell

Returns
digit to round to

Return type
int

```
>>> from mtpy.utils.mesh_tools import get_rounding
>>> get_rounding(9)
0
>>> get_rounding(90)
```

(continues on next page)

(continued from previous page)

```
-1
>>> get_rounding(900)
-2
>>> get_rounding(9000)
-3
```

mtpy.modeling.mesh_tools.**get_station_buffer**(*grid_east*, *grid_north*, *station_east*, *station_north*, *buf=10000.0*)

get cells within a specified distance (buf) of the stations returns a 2D boolean (True/False) array

mtpy.modeling.mesh_tools.**grid_centre**(*grid_edges*)

calculate the grid centres from an array that defines grid edges :param grid_edges: array containing grid edges :returns: grid_centre: centre points of grid

mtpy.modeling.mesh_tools.**interpolate_elevation_to_grid**(*grid_east*, *grid_north*, *utm_epsg=None*, *datum_epsg=4326*, *surface_file=None*, *surface=None*, *method='linear'*, *fast=True*, *buffer=1*)

Note: this documentation is outdated and seems to be copied from # model.interpolate_elevation2. It needs to be updated. This # funciton does not update a dictionary but returns an array of # elevation data.

project a surface to the model grid and add resulting elevation data to a dictionary called surface_dict. Assumes the surface is in lat/long coordinates (wgs84) The 'fast' method extracts a subset of the elevation data that falls within the mesh-bounds and interpolates them onto mesh nodes. This approach significantly speeds up (~ x5) the interpolation procedure.

returns nothing returned, but surface data are added to surface_dict under the key given by surfacename.

inputs choose to provide either surface_file (path to file) or surface (tuple). If both are provided then surface tuple takes priority.

surface elevations are positive up, and relative to sea level. surface file format is:

ncols 3601 nrows 3601 xllcorner -119.00013888889 (longitude of lower left) yllcorner 36.999861111111 (latitude of lower left) cellsize 0.00027777777777778 NODATA_value -9999 elevation data W –> E N | V S

Alternatively, provide a tuple with: (lon,lat,elevation) where elevation is a 2D array (shape (ny,nx)) containing elevation points (order S -> N, W -> E) and lon, lat are either 1D arrays containing list of longitudes and latitudes (in the case of a regular grid) or 2D arrays with same shape as elevation array containing longitude and latitude of each point.

other inputs: surfacename = name of surface for putting into dictionary surface_epsg = epsg number of input surface, default is 4326 for lat/lon(wgs84) method = interpolation method. Default is 'nearest', if model grid is dense compared to surface points then choose 'linear' or 'cubic'

mtpy.modeling.mesh_tools.**make_log_increasing_array**(*z1_layer*, *target_depth*, *n_layers*, *increment_factor=0.9*)

create depth array with log increasing cells, down to target depth, inputs are z1_layer thickness, target depth, number of layers (n_layers)

mtpy.modeling.mesh_tools.**rotate_mesh**(*grid_east*, *grid_north*, *origin*, *rotation_angle*, *return_centre=False*)

rotate a mesh defined by grid_east and grid_north.

Parameters

- **grid_east** – 1d array defining the edges of the mesh in the east-west direction
- **grid_north** – 1d array defining the edges of the mesh in the north-south direction

- **origin** – real-world position of the (0,0) point in grid_east, grid_north
- **rotation_angle** – angle in degrees to rotate the grid by
- **return_centre** – True/False option to return points on centre of grid instead of grid edges

Returns
grid_east, grid_north - 2d arrays describing the east and north coordinates

mtpy.modeling.occam1d module

Created on Mon Oct 30 13:56:36 2023

@author: jpeacock

class mtpy.modeling.occam1d.**Occam1DData**(*mt_dataframe*, ***kwargs*)

Bases: `object`

reads and writes occam 1D data files

Attributes	Description
_data_fn	basename of data file *default* is Occam1DDataFile
_header_line	header line for description of data columns
_ss	string spacing *default* is 6*' '
_string_fmt	format of data *default* is '+.6e'
data	array of data
data_fn	full path to data file
freq	frequency array of data
mode	mode to invert for ['TE' | 'TM' | 'det']
phase_te	array of TE phase
phase_tm	array of TM phase
res_te	array of TE apparent resistivity
res_tm	array of TM apparent resistivity
resp_fn	full path to response file
save_path	path to save files to

Methods	Description
write_data_file	write an Occam1D data file
read_data_file	read an Occam1D data file
read_resp_file	read a .resp file output by Occam1D

Example

```
>>> import mtpy.modeling.occam1d as occam1d
>>> #--> make a data file for TE mode
>>> d1 = occam1d.Data()
>>> d1.write_data_file(edi_file=r'/home/MT/mt01.edi', res_err=10,␣
↪phase_err=2.5,
>>> ...                save_path=r"/home/occam1d/mt01/TE", mode='TE')
```

property mode

property mode_01

property mode_02

read_data_file(*data_fn*)

reads a 1D data file

Arguments:

data_fn : full path to data file

Returns:

Occam1D.rpdict : dictionary with keys:

'freq' : an array of frequencies with length nf

'resxy'
[TE resistivity array with shape (nf,4) for (0) data,]

(1) dataerr, (2) model, (3) modelerr

'resyx'
[TM resistivity array with shape (nf,4) for (0) data,]

(1) dataerr, (2) model, (3) modelerr

'phasexy'
[TE phase array with shape (nf,4) for (0) data,]

(1) dataerr, (2) model, (3) modelerr

'phaseyx'
[TM phase array with shape (nf,4) for (0) data,]

(1) dataerr, (2) model, (3) modelerr

Example

```
>>> old = occam1d.Data()
>>> old.data_fn = r"/home/Occam1D/Line1/Inv1_TE/MT01TE.dat"
>>> old.read_data_file()
```

read_resp_file(*resp_fn=None*, *data_fn=None*)

read response file

resp_fn : full path to response file

data_fn : full path to data file

freq : an array of frequencies with length nf

res_te
[TE resistivity array with shape (nf,4) for (0) data,]

(1) dataerr, (2) model, (3) modelerr

res_tm
[TM resistivity array with shape (nf,4) for (0) data,]

(1) dataerr, (2) model, (3) modelerr

phase_te
[TE phase array with shape (nf,4) for (0) data,]

(1) dataerr, (2) model, (3) modelerr

phase_tm
[TM phase array with shape (nf,4) for (0) data,]

(1) dataerr, (2) model, (3) modelerr

Example
:: >>> o1d = occam1d.Data() >>> o1d.data_fn = r"/home/occam1d/mt01/TE/Occam1D_DataFile_TE.dat" >>> o1d.read_resp_file(r"/home/occam1d/mt01/TE/TE_7.resp")

write_data_file(*filename*, *mode='det'*, *remove_outofquadrant=False*)
make1Ddatafile will write a data file for Occam1D

Arguments:

rp_tuple
[np.ndarray (freq, res, res_err, phase, phase_err)] with res, phase having shape (num_freq, 2, 2).

edi_file
[string] full path to edi file to be modeled.

save_path
[string] path to save the file, if None set to dirname of station if edipath = None. Otherwise set to dirname of edipath.

thetar
[float] rotation angle to rotate Z. Clockwise positive and N=0 *default* = 0

mode
[['te' | 'tm' | 'det']]

mode to model can be (*default*='both'):

- 'te' for just TE mode (res/phase)
- 'tm' for just TM mode (res/phase)
- **'det' for the determinant of Z (converted to**
  res/phase)

add 'z' to any of these options to model impedance tensor values instead of res/phase

res_err
[float] errorbar for resistivity values. Can be set to (*default* = 'data'):

- 'data' for errorbars from the data
- percent number ex. 10 for ten percent

phase_err
[float] errorbar for phase values. Can be set to (*default* = 'data'):

- 'data' for errorbars from the data
- percent number ex. 10 for ten percent

res_errorfloor: float
error floor for resistivity values in percent

phase_errorfloor: float
error floor for phase in degrees

remove_outofquadrant: True/False; option to remove the resistivity and
phase values for points with phases out of the 1st/3rd quadrant (occam requires 0 < phase < 90 degrees; phases in the 3rd quadrant are shifted to the first by adding 180 degrees)

Example

```
>>> import mtpy.modeling.occam1d as occam1d
>>> #--> make a data file
>>> d1 = occam1d.Data()
>>> d1.write_data_file(edi_file=r'/home/MT/mt01.edi', res_err=10,
>>> ...                 phase_err=2.5, mode='TE',
>>> ...                 save_path=r"/home/occam1d/mt01/TE")
```

class mtpy.modeling.occam1d.**Occam1DModel**(*model_fn=None*, ***kwargs*)

Bases: `object`

read and write the model file fo Occam1D

All depth measurements are in meters.

Attributes	Description
_model_fn	basename for model file *default* is Model1D
_ss	string spacing in model file *default* is 3*' '
_string_fmt	format of model layers *default* is '.0f'
air_layer_height	height of air layer *default* is 10000
bottom_layer	bottom of the model *default* is 50000
itdict	dictionary of values from iteration file
iter_fn	full path to iteration file
model_depth	array of model depths
model_fn	full path to model file
model_penalty	array of penalties for each model layer
model_preference_penalty	array of model preference penalties for each layer
model_prefernce	array of preferences for each layer
model_res	array of resistivities for each layer
n_layers	number of layers in the model
num_params	number of parameters to invert for (n_layers+2)
pad_z	padding of model at depth *default* is 5 blocks
save_path	path to save files
target_depth	depth of target to investigate
z1_layer	depth of first layer *default* is 10

Methods	Description
write_model_file	write an Occam1D model file, where depth increases on a logarithmic scale
read_model_file	read an Occam1D model file
read_iter_file	read an .iter file output by Occam1D

Example

```
>>> #--> make a model file
>>> m1 = occam1d.Model()
>>> m1.write_model_file(save_path=r"/home/occam1d/mt01/TE")
```

read_iter_file(*iter_fn=None*, *model_fn=None*)

read an 1D iteration file

Arguments:

imode : mode to read from

Returns:

Occam1D.itdict : dictionary with keys of the header:

model_res
[fills this array with the appropriate] values (0) for data, (1) for model

Example

```
>>> m1 = occam1d.Model()
>>> m1.model_fn = r"/home/occam1d/mt01/TE/Model1D"
>>> m1.read_iter_file(r"/home/Occam1D/Inv1_TE/M01TE_15.iter")
```

read_model_file(*model_fn=None*)

will read in model 1D file

Arguments:

modelfn : full path to model file

Fills attributes:

- model_depth' : depth of model in meters
- model_res : value of resisitivity
- model_penalty : penalty
- model_preference : preference
- model_penalty_preference : preference penalty

Example

```
>>> m1 = occam1d.Model()
>>> m1.savepath = r"/home/Occam1D/Line1/Inv1_TE"
>>> m1.read_model_file()
```

write_model_file(*save_path=None*, ***kwargs*)

Makes a 1D model file for Occam1D.

Arguments:

save_path :path to save file to, if just path saved as
savepathmodel.mod, if None defaults to dirpath

n_layers : number of layers

bottom_layer : depth of bottom layer in meters

target_depth : depth to target under investigation

pad_z : padding on bottom of model past target_depth

z1_layer : depth of first layer in meters

air_layer_height : height of air layers in meters

Returns:

Occam1D.modelfn = full path to model file

..Note: This needs to be redone.

Example

```
>>> old = occam.Occam1D()
>>> old.make1DModelFile(savepath=r"/home/Occam1D/Line1/Inv1_TE",
>>>                     nlayers=50,bottomlayer=10000,z1layer=50)
>>> Wrote Model file: /home/Occam1D/Line1/Inv1_TE/Model1D
```

class mtpy.modeling.occam1d.**Occam1DRun**(*startup_fn=None*, *occam_path=None*, ***kwargs*)

Bases: `object`

run occam 1d from python given the correct files and location of occam1d executable

run_occam1d()

class mtpy.modeling.occam1d.**Occam1DStartup**(*data_fn=None*, *model_fn=None*, ***kwargs*)

Bases: `object`

read and write input files for Occam1D

Attributes	Description
_ss	string spacing
_startup_fn	basename of startup file *default* is OccamStartup1D
data_fn	full path to data file
debug_level	debug level *default* is 1
description	description of inversion for your self *default* is 1D_Occam_Inv
max_iter	maximum number of iterations *default* is 20
model_fn	full path to model file
rough_type	roughness type *default* is 1
save_path	full path to save files to
start_iter	first iteration number *default* is 0
start_lagrange	starting lagrange number on log scale *default* is 5
start_misfit	starting misfit value *default* is 100
start_rho	starting resistivity value (halfspace) in log scale *default* is 100
start_rough	starting roughness (ignored by Occam1D) *default* is 1E7
startup_fn	full path to startup file
target_rms	target rms *default* is 1.0

property data_fn

property model_fn

read_startup_file(*startup_fn*)

reads in a 1D input file

Arguments:

inputfn : full path to input file

Returns:

Occam1D.indict : dictionary with keys following the header and

'res' : an array of resistivity values

Example

```
>>> old = occam.Occam1d()
>>> old.savepath = r"/home/Occam1D/Line1/Inv1_TE"
>>> old.read1DInputFile()
```

write_startup_file(*save_path=None*, ***kwargs*)

Make a 1D input file for Occam 1D

Arguments:

savepath
[full path to save input file to, if just path then] saved as savepath/input

model_fn
[full path to model file, if None then assumed to be in] savepath/model.mod

data_fn
[full path to data file, if None then assumed to be] in savepath/TE.dat or TM.dat

rough_type : roughness type. *default* = 0

max_iter : maximum number of iterations. *default* = 20

target_rms : target rms value. *default* = 1.0

start_rho
[starting resistivity value on linear scale.] *default* = 100

description : description of the inversion.

start_lagrange
[starting Lagrange multiplier for smoothness.] *default* = 5

start_rough : starting roughness value. *default* = 1E7

debuglevel
[something to do with how Fortran debuggs the code] Almost always leave at *default* = 1

start_iter
[the starting iteration number, handy if the] starting model is from a previous run. *default* = 0

start_misfit : starting misfit value. *default* = 100

Returns:

Occam1D.inputfn : full path to input file.

Example

```
>>> old = occam.Occam1D()
>>> old.make1DdataFile('MT01',edipath=r"/home/Line1",
>>>                    savepath=r"/home/Occam1D/Line1/Inv1_TE",
>>>                    mode='TE')
>>> Wrote Data File: /home/Occam1D/Line1/Inv1_TE/MT01TE.dat
>>>
>>> old.make1DModelFile(savepath=r"/home/Occam1D/Line1/Inv1_TE",
>>>                     nlayers=50,bottomlayer=10000,z1layer=50)
>>> Wrote Model file: /home/Occam1D/Line1/Inv1_TE/Model1D
>>>
>>> old.make1DInputFile(rhostart=10,targetrms=1.5,maxiter=15)
>>> Wrote Input File: /home/Occam1D/Line1/Inv1_TE/Input1D
```

class mtpy.modeling.occam1d.**Plot1DResponse**(*data_te_fn=None*, *data_tm_fn=None*, *model_fn=None*, *resp_te_fn=None*, *resp_tm_fn=None*, *iter_te_fn=None*, *iter_tm_fn=None*, ***kwargs*)

Bases: `object`

plot the 1D response and model. Plots apparent resisitivity and phase in different subplots with the model on the far right. You can plot both TE and TM modes together along with different iterations of the model. These will be plotted in different colors or shades of gray depneng on color_scale.

Example

```
>>> import mtpy.modeling.occam1d as occam1d
>>> p1 = occam1d.Plot1DResponse(plot_yn='n')
>>> p1.data_te_fn = r"/home/occam1d/mt01/TE/Occam_DataFile_TE.dat"
>>> p1.data_tm_fn = r"/home/occam1d/mt01/TM/Occam_DataFile_TM.dat"
>>> p1.model_fn = r"/home/occam1d/mt01/TE/Model1D"
>>> p1.iter_te_fn = [r"/home/occam1d/mt01/TE/TE_{0}.iter".format(ii)
>>> ...              for ii in range(5,10)]
>>> p1.iter_tm_fn = [r"/home/occam1d/mt01/TM/TM_{0}.iter".format(ii)
>>> ...              for ii in range(5,10)]
>>> p1.resp_te_fn = [r"/home/occam1d/mt01/TE/TE_{0}.resp".format(ii)
>>> ...              for ii in range(5,10)]
>>> p1.resp_tm_fn = [r"/home/occam1d/mt01/TM/TM_{0}.resp".format(ii)
>>> ...              for ii in range(5,10)]
>>> p1.plot()
```

Attributes	Description
axm	matplotlib.axes instance for model subplot
axp	matplotlib.axes instance for phase subplot
axr	matplotlib.axes instance for app. res subplot
color_mode	[‘color’ | ‘bw’]
cted	color of TE data markers
ctem	color of TM data markers
ctmd	color of TE model markers
ctmm	color of TM model markers
data_te_fn	full path to data file for TE mode
data_tm_fn	full path to data file for TM mode
depth_limits	(min, max) limits for depth plot in depth_units
depth_scale	[‘log’ | ‘linear’] *default* is linear
depth_units	[‘m’ | ‘km’] *default is ‘km’
e_capsize	capsize of error bars
e_capthick	cap thickness of error bars
fig	matplotlib.figure instance for plot
fig_dpi	resolution in dots-per-inch for figure
fig_num	number of figure instance
fig_size	size of figure in inches [width, height]
font_size	size of axes tick labels, axes labels are +2
grid_alpha	transparency of grid
grid_color	color of grid
iter_te_fn	full path or list of .iter files for TE mode
iter_tm_fn	full path or list of .iter files for TM mode
lw	width of lines for model
model_fn	full path to model file
ms	marker size
mted	marker for TE data

continues on next page

Table 7 – continued from previous page

Attributes	Description
mtem	marker for TM data
mtmd	marker for TE model
mtmm	marker for TM model
phase_limits	(min, max) limits on phase in degrees
phase_major_ticks	spacing for major ticks in phase
phase_minor_ticks	spacing for minor ticks in phase
plot_yn	['y' | 'n'] plot on instantiation
res_limits	limits of resistivity in linear scale
resp_te_fn	full path or list of .resp files for TE mode
resp_tm_fn	full path or list of .iter files for TM mode
subplot_bottom	spacing of subplots from bottom of figure
subplot_hspace	height spacing between subplots
subplot_left	spacing of subplots from left of figure
subplot_right	spacing of subplots from right of figure
subplot_top	spacing of subplots from top of figure
subplot_wspace	width spacing between subplots
title_str	title of plot

plot()

plot data, response and model

redraw_plot()

redraw plot if parameters were changed

use this function if you updated some attributes and want to re-plot.

Example

```
>>> # change the color and marker of the xy components
>>> import mtpy.modeling.occam2d as occam2d
>>> ocd = occam2d.Occam2DData(r"/home/occam2d/Data.dat")
>>> p1 = ocd.plotAllResponses()
>>> #change line width
>>> p1.lw = 2
>>> p1.redraw_plot()
```

save_figure(*save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot='y'*)

save_plot will save the figure to save_fn.

Arguments:

save_fn

[string] full path to save figure to, can be input as * directory path -> the directory path to save to

in which the file will be saved as save_fn/station_name_PhaseTensor.file_format

- full path -> file will be save to the given path. If you use this option then the format will be assumed to be provided by the path

file_format

[[pdf | eps | jpg | png | svg]] file type of saved figure pdf,svg,eps…

orientation
[[landscape | portrait]] orientation in which the file will be saved *default* is portrait

fig_dpi
[int] The resolution in dots-per-inch the file will be saved. If None then the dpi will be that at which the figure was made. I don't think that it can be larger than dpi of the figure.

close_plot
[[y | n]]

- 'y' will close the plot after saving.
- 'n' will leave plot open

Example

```
>>> # to save plot as jpg
>>> import mtpy.modeling.occam2d as occam2d
>>> dfn = r"/home/occam2d/Inv1/data.dat"
>>> ocd = occam2d.Occam2DData(dfn)
>>> ps1 = ocd.plotPseudoSection()
>>> ps1.save_plot(r'/home/MT/figures', file_format='jpg')
```

update_plot(*fig*)

update any parameters that where changed using the built-in draw from canvas.

Use this if you change an of the .fig or axes properties

Example

```
>>> # to change the grid lines to only be on the major ticks
>>> import mtpy.modeling.occam2d as occam2d
>>> dfn = r"/home/occam2d/Inv1/data.dat"
>>> ocd = occam2d.Occam2DData(dfn)
>>> ps1 = ocd.plotAllResponses()
>>> [ax.grid(True, which='major') for ax in [ps1.axrte,ps1.axtep]]
>>> ps1.update_plot()
```

class mtpy.modeling.occam1d.**PlotOccam1DL2**(*dir_path*, *model_fn*, ***kwargs*)

Bases: *PlotBase*

plot L2 curve of iteration vs rms and roughness

Arguments:

rms_arr
[structured array with keys:]

- 'iteration' –> for iteration number (int)
- 'rms' –> for rms (float)
- 'roughness' –> for roughness (float)

Keywords/attributes	Description
ax1	matplotlib.axes instance for rms vs iteration
ax2	matplotlib.axes instance for roughness vs rms
fig	matplotlib.figure instance
fig_dpi	resolution of figure in dots-per-inch
fig_num	number of figure instance
fig_size	size of figure in inches (width, height)
font_size	size of axes tick labels, axes labels is +2
plot_yn	['y' | 'n'] 'y' –> to plot on instantiation 'n' –> to not plot on instantiation
rms_arr	structure np.array as described above
rms_color	color of rms marker and line
rms_lw	line width of rms line
rms_marker	marker for rms values
rms_marker_size	size of marker for rms values
rms_mean_color	color of mean line
rms_median_color	color of median line
rough_color	color of roughness line and marker
rough_font_size	font size for iteration number inside roughness marker
rough_lw	line width for roughness line
rough_marker	marker for roughness
rough_marker_size	size of marker for roughness
subplot_bottom	subplot spacing from bottom
subplot_left	subplot spacing from left
subplot_right	subplot spacing from right
subplot_top	subplot spacing from top

plot()

plot L2 curve

mtpy.modeling.occam2d module

class mtpy.modeling.occam2d.**Mesh**(*station_locations=None*, ***kwargs*)

Bases: `object`

deals only with the finite element mesh. Builds a finite element mesh based on given parameters defined below. The mesh reads in the station locations, finds the center and makes the relative location of the furthest left hand station 0. The mesh increases in depth logarithmically as required by the physics of MT. Also, the model extends horizontally and vertically with padding cells in order to fullfill the assumption of the forward operator that at the edges the structure is 1D. Stations are place on the horizontal nodes as required by Wannamaker's forward operator.

Mesh has the ability to create a mesh that incorporates topography given a elevation profile. It adds more cells to the mesh with thickness z1_layer. It then sets the values of the triangular elements according to the elevation value at that location. If the elevation covers less than 50% of the triangular cell, then the cell value is set to that of air

Note: Mesh is inhereted by Regularization, so the mesh can also be be built from there, same as the example below.

Arguments:

Key Words/Attrib	Description
air_key	letter associated with the value of air *default* is 0
air_value	value given to an air cell, *default* is 1E13
cell_width	width of cells with in station area in meters *default* is 100
eleva-tion_profile	elevation profile along the profile line. given as np.ndarray(nx, 2), where the elements are x_location, elevation. If elevation profile is given add_elevation is called automatically. *default* is None
mesh_fn	full path to mesh file.
mesh_values	letter values of each triangular mesh element if the cell is free value is ?
n_layers	number of vertical layers in mesh *default* is 90
num_x_pad_	number of horizontal padding cells outside the the station area that will increase in size by x_pad_multiplier. *default* is 7
num_x_pad_	number of horizonal padding cells just outside the station area with width cell_width. This is to extend the station area if needed. *default* is 2
num_z_pad_	number of vertical padding cells below z_target_depth down to z_bottom. *default* is 5
rel_station_lo	relative station locations within the mesh. The locations are relative to the center of the station area. *default* is None, filled later
save_path	full path to save mesh file to. *default* is current working directory.
sta-tion_location	location of stations in meters, can be on a relative grid or in UTM.
x_grid	location of horizontal grid nodes in meters
x_nodes	relative spacing between grid nodes
x_pad_multip	horizontal padding cells will increase by this multiple out to the edge of the grid. *default* is 1.5
z1_layer	thickness of the first layer in the model. Should be at least 1/4 of the first skin depth *default* is 10
z_bottom	bottom depth of the model (m). Needs to be large enough to be 1D at the edge. *default* is 200000.0
z_grid	location of vertical nodes in meters
z_nodes	relative distance between vertical nodes in meters
z_target_dept	depth to deepest target of interest. Below this depth cells will be padded to z_bottom

Methods	Description
add_elevation	adds elevation to the mesh given elevation profile.
build_mesh	builds the mesh given the attributes of Mesh. If elevation_profile is not None, add_elevation is called inside build_mesh
plot_mesh	plots the built mesh with station location.
read_mesh_fil	reads in an existing mesh file and populates the appropriate attributes.
write_mesh_fi	writes a mesh file to save_path

Example

```
>>> import mtpy.modeling.occam2d as occcam2d
>>> edipath = r"/home/mt/edi_files"
>>> slist = ['mt{0:03}'.format(ss) for ss in range(20)]
>>> ocd = occam2d.Data(edi_path=edipath, station_list=slist)
>>> ocd.save_path = r"/home/occam/Line1/Inv1"
```

(continues on next page)

(continued from previous page)

```
>>> ocd.write_data_file()
>>> ocm = occam2d.Mesh(ocd.station_locations)
>>> # add in elevation
>>> ocm.elevation_profile = ocd.elevation_profile
>>> # change number of layers
>>> ocm.n_layers = 110
>>> # change cell width in station area
>>> ocm.cell_width = 200
>>> ocm.build_mesh()
>>> ocm.plot_mesh()
>>> ocm.save_path = ocd.save_path
>>> ocm.write_mesh_file()
```

add_elevation(*elevation_profile=None*)

the elevation model needs to be in relative coordinates and be a numpy.ndarray(2, num_elevation_points) where the first column is the horizontal location and the second column is the elevation at that location.

If you have a elevation model use Profile to project the elevation information onto the profile line

To build the elevation I'm going to add the elevation to the top of the model which will add cells to the mesh. there might be a better way to do this, but this is the first attempt. So I'm going to assume that the first layer of the mesh without elevation is the minimum elevation and blocks will be added to max elevation at an increment according to z1_layer

Note: the elevation model should be symmetrical ie, starting at the first station and ending on the last station, so for now any elevation outside the station area will be ignored and set to the elevation of the station at the extremities. This is not ideal but works for now.

Arguments:

elevation_profile
[np.ndarray(2, num_elev_points)]

- 1st row is for profile location
- 2nd row is for elevation values

Computes:

mesh_values
[mesh values, setting anything above topography] to the key for air, which for Occam is '0'

build_mesh()

Build the finite element mesh given the parameters defined by the attributes of Mesh. Computes relative station locations by finding the center of the station area and setting the middle to 0. Mesh blocks are built by calculating the distance between stations and putting evenly spaced blocks between the stations being close to cell_width. This places a horizontal node at the station location. If the spacing between stations is smaller than cell_width, a horizontal node is placed between the stations to be sure the model has room to change between the station.

If elevation_profile is given, add_elevation is called to add topography into the mesh.

Populates attributes:

- mesh_values
- rel_station_locations
- x_grid
- x_nodes
- z_grid
- z_nodes

Example

:: >>> import mtpy.modeling.occam2d as occcam2d >>> edipath = r"/home/mt/edi_files" >>> slist = ['mt{0:03}'.format(ss) for ss in range(20)] >>> ocd = occam2d.Data(edi_path=edipath, station_list=slist) >>> ocd.save_path = r"/home/occam/Line1/Inv1" >>> ocd.write_data_file() >>> ocm = occam2d.Mesh(ocd.station_locations) >>> # add in elevation >>> ocm.elevation_profile = ocd.elevation_profile >>> # change number of layers >>> ocm.n_layers = 110 >>> # change cell width in station area >>> ocm.cell_width = 200 >>> ocm.build_mesh()

plot_mesh(***kwargs*)

Plot built mesh with station locations.

Key Words	Description
depth_scale	['km' | 'm'] scale of mesh plot. *default* is 'km'
fig_dpi	dots-per-inch resolution of the figure *default* is 300
fig_num	number of the figure instance *default* is 'Mesh'
fig_size	size of figure in inches (width, height) *default* is [5, 5]
fs	size of font of axis tick labels, axis labels are fs+2. *default* is 6
ls	['-' | '.' | ':'] line style of mesh lines *default* is '-'
marker	marker of stations *default* is r"$lacktriangledown$"
ms	size of marker in points. *default* is 5
plot_triangles	['y' | 'n'] to plot mesh triangles. *default* is 'n'

read_mesh_file(*mesh_fn*)

reads an occam2d 2D mesh file

Arguments:

mesh_fn
[string] full path to mesh file

Populates:

x_grid : array of horizontal locations of nodes (m)

x_nodes: array of horizontal node relative distances
(column locations (m))

z_grid : array of vertical node locations (m)

z_nodes
[array of vertical nodes] (row locations(m))

mesh_values : np.array of free parameters

To do:

incorporate fixed values

Example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> mg = occam2d.Mesh()
>>> mg.mesh_fn = r"/home/mt/occam/line1/Occam2Dmesh"
>>> mg.read_mesh_file()
```

write_mesh_file(*save_path=None*, *basename='Occam2DMesh'*)
Write a finite element mesh file.

Calls build_mesh if it already has not been called.

Arguments:

save_path
[string] directory path or full path to save file

basename
[string] basename of mesh file. *default* is 'Occam2DMesh'

Returns:

mesh_fn
[string] full path to mesh file

example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> edi_path = r"/home/mt/edi_files"
>>> profile = occam2d.Profile(edi_path)
>>> profile.plot_profile()
>>> mesh = occam2d.Mesh(profile.station_locations)
>>> mesh.build_mesh()
>>> mesh.write_mesh_file(save_path=r"/home/occam2d/Inv1")
```

class `mtpy.modeling.occam2d.`**`Occam2DData`**(*dataframe=None*, *center_point=None*, ***kwargs*)

Bases: `object`

Reads and writes data files and more.

Inherets Profile, so the intended use is to use Data to project stations onto a profile, then write the data file.

Model Modes	Description
1 or log_all	Log resistivity of TE and TM plus Tipper
2 or log_te_tip	Log resistivity of TE plus Tipper
3 or log_tm_tip	Log resistivity of TM plus Tipper
4 or log_te_tm	Log resistivity of TE and TM
5 or log_te	Log resistivity of TE
6 or log_tm	Log resistivity of TM
7 or all	TE, TM and Tipper
8 or te_tip	TE plus Tipper
9 or tm_tip	TM plus Tipper
10 or te_tm	TE and TM mode
11 or te	TE mode
12 or tm	TM mode
13 or tip	Only Tipper

Example Write Data File

```
>>> from mtpy.modeling.occam2d import Data
>>> occam_data_object = Data()
>>> occam_data_object.read_data_file(r"path/to/data/file.dat")
>>> occam_data_object.model_mode = 2
>>> occam_data_object.write_data_file(r"path/to/new/data/file_te.dat")
```

property data_filename

property dataframe

property frequencies

mask_from_datafile(*mask_datafn*)

reads a separate data file and applies mask from this data file. mask_datafn needs to have exactly the same frequencies, and station names must match exactly.

property n_data

property n_frequencies

property n_stations

property offsets

read_data_file(*data_fn=None*)

Read in an existing data file and populate appropriate attributes

- data
- data_list
- freq

- station_list
- station_locations

Arguments:

data_fn
[string] full path to data file *default* is None and set to save_path/fn_basename

Example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> ocd = occam2d.Data()
>>> ocd.read_data_file(r"/home/Occam2D/Line1/Inv1/Data.dat")
```

property stations

write_data_file(*data_fn=None*)

Write a data file.

Arguments:

data_fn
[string] full path to data file. *default* is save_path/fn_basename

If there data is None, then _fill_data is called to create a profile, rotate data and get all the necessary data. This way you can use write_data_file directly without going through the steps of projecting the stations, etc.

Example
:: >>> edipath = r"/home/mt/edi_files" >>> slst = ['mt{0:03}'.format(ss) for ss in range(1, 20)] >>> ocd = occam2d.Data(edi_path=edipath, station_list=slst) >>> ocd.save_path = r"/home/occam/line1/inv1" >>> ocd.write_data_file()

class mtpy.modeling.occam2d.**Occam2DModel**(*iter_fn=None*, *model_fn=None*, *mesh_fn=None*, ***kwargs*)

Bases: *Startup*

Read .iter file output by Occam2d. Builds the resistivity model from mesh and regularization files found from the .iter file. The resistivity model is an array(x_nodes, z_nodes) set on a regular grid, and the values of the model response are filled in according to the regularization grid. This allows for faster plotting.

Inherets Startup because they are basically the same object.

Argument:

iter_fn
[string] full path to .iter file to read. *default* is None.

model_fn
[string] full path to regularization file. *default* is None and found directly from the .iter file. Only input if the regularization is different from the file that is in the .iter file.

mesh_fn
[string] full path to mesh file. *default* is None Found directly from the model_fn file. Only input if the mesh is different from the file that is in the model file.

Key Words/Attributes	Description
data_fn	full path to data file
iter_fn	full path to .iter file
mesh_fn	full path to mesh file
mesh_x	np.ndarray(x_nodes, z_nodes) mesh grid for plotting
mesh_z	np.ndarray(x_nodes, z_nodes) mesh grid for plotting
model_values	model values from startup file
plot_x	nodes of mesh in horizontal direction
plot_z	nodes of mesh in vertical direction
res_model	np.ndarray(x_nodes, z_nodes) resistivity model values in linear scale

Methods	Description
build_mode	get the resistivity model from the .iter file in a regular grid according to the mesh file with resistivity values according to the model file
read_iter_fi	read .iter file and fill appropriate attributes
write_iter_1	write an .iter file incase you want to set it as the starting model or a priori model

Example
:: >>> model = occam2D.Model(r"/home/occam/line1/inv1/test_01.iter") >>> model.build_model()

build_model()

build the model from the mesh, regularization grid and model file

read_iter_file(*iter_fn=None*)

Read an iteration file.

Arguments:

iter_fn
[string] full path to iteration file if iterpath=None. If iterpath is input then iterfn is just the name of the file without the full path.

Returns:

Example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> itfn = r"/home/Occam2D/Line1/Inv1/Test_15.iter"
>>> ocm = occam2d.Model(itfn)
>>> ocm.read_iter_file()
```

write_iter_file(*iter_fn=None*)

write an iteration file if you need to for some reason, same as startup file

class mtpy.modeling.occam2d.**Regularization**(*station_locations=None*, ***kwargs*)

Bases: *Mesh*

Creates a regularization grid based on Mesh. Note that Mesh is inherited by Regularization, therefore the intended use is to build a mesh with the Regularization class.

The regularization grid is what Occam calculates the inverse model on. Setup is tricky and can be painful, as you can see it is not quite fully functional yet, as it cannot incorporate topography yet. It seems like you'd like to have the regularization setup so that your target depth is covered well, in that the regularization blocks to this depth are sufficiently small to resolve resistivity structure at that depth. Finally, you want the regularization to go to a half space at the bottom, basically one giant block.

Arguments:

station_locations
[np.ndarray(n_stations)] array of station locations along a profile line in meters.

Key Words/Attributes	Description
air_key	letter associated with the value of air *default* is 0
air_value	value given to an air cell, *default* is 1E13
binding_offset	offset from the right side of the furthest left hand model block in meters. The regularization grid is s
cell_width	width of cells with in station area in meters *default* is 100
description	description of the model for the model file. *default* is 'simple inversion'
elevation_profile	elevation profile along the profile line. given as np.ndarray(nx, 2), where the elements are x_location
mesh_fn	full path to mesh file.
mesh_values	letter values of each triangular mesh element if the cell is free value is ?
model_columns	
model_name	
model_rows	
min_block_width	[float] minimum model block width in meters, *default* is 2*cell_width
n_layers	number of vertical layers in mesh *default* is 90
num_free_param	[int] number of free parameters in the model. this is a tricky number to estimate apparently.
num_layers	[int] number of regularization layers.
num_x_pad_cells	number of horizontal padding cells outside the the station area that will increase in size by x_pad_mu
num_x_pad_small_cells	number of horizonal padding cells just outside the station area with width cell_width. This is to exter
num_z_pad_cells	number of vertical padding cells below z_target_depth down to z_bottom. *default* is 5
prejudice_fn	full path to prejudice file *default* is 'none'
reg_basename	basename of regularization file (model file) *default* is 'Occam2DModel'
reg_fn	full path to regularization file (model file) *default* is save_path/reg_basename
rel_station_locations	relative station locations within the mesh. The locations are relative to the center of the station area.
save_path	full path to save mesh and model file to. *default* is current working directory.
statics_fn	full path to static shift file Static shifts in occam may not work. *default* is 'none'
station_locations	location of stations in meters, can be on a relative grid or in UTM.
trigger	[float] multiplier to merge model blocks at depth. A higher number increases the number of model
x_grid	location of horizontal grid nodes in meters
x_nodes	relative spacing between grid nodes
x_pad_multiplier	horizontal padding cells will increase by this multiple out to the edge of the grid. *default* is 1.5
z1_layer	thickness of the first layer in the model. Should be at least 1/4 of the first skin depth *default* is 10
z_bottom	bottom depth of the model (m). Needs to be large enough to be 1D at the edge. *default* is 200000.0

Table 8 – continued from previous p

Key Words/Attributes	Description
z_grid	location of vertical nodes in meters
z_nodes	relative distance between vertical nodes in meters
z_target_depth	depth to deepest target of interest. Below this depth cells will be padded to z_bottom

Note: regularization does not work with topography yet. Having problems calculating the number of free parameters.

Example

```
>>> edipath = r"/home/mt/edi_files"
>>> profile = occam2d.Profile(edi_path=edi_path)
>>> profile.generate_profile()
>>> reg = occam2d.Regularization(profile.station_locations)
>>> reg.build_mesh()
>>> reg.build_regularization()
>>> reg.save_path = r"/home/occam2d/Line1/Inv1"
>>> reg.write_regularization_file()
```

build_regularization()

Builds larger boxes around existing mesh blocks for the regularization. As the model deepens the regularization boxes get larger.

The regularization boxes are merged mesh cells as prescribed by the Occam method.

get_num_free_params()

estimate the number of free parameters in model mesh.

I'm assuming that if there are any fixed parameters in the block, then that model block is assumed to be fixed. Not sure if this is right cause there is no documentation.

DOES NOT WORK YET

read_regularization_file(*reg_fn*)

Read in a regularization file and populate attributes:

- binding_offset
- mesh_fn
- model_columns
- model_rows
- prejudice_fn
- statics_fn

write_regularization_file(*reg_fn=None*, *reg_basename=None*, *statics_fn='none'*, *prejudice_fn='none'*, *save_path=None*)

Write a regularization file for input into occam.

Calls build_regularization if build_regularization has not already been called.

if reg_fn is None, then file is written to save_path/reg_basename

Arguments:

reg_fn
[string] full path to regularization file. *default* is None and file will be written to save_path/reg_basename

reg_basename
[string] basename of regularization file

statics_fn
[string] full path to static shift file .. note:: static shift does not always work in

occam2d.exe

prejudice_fn
[string] full path to prejudice file

save_path
[string] path to save regularization file. *default* is current working directory

class mtpy.modeling.occam2d.**Startup**(***kwargs*)

Bases: object

Reads and writes the startup file for Occam2D.

Note: Be sure to look at the Occam 2D documentation for description of all parameters

Key Words/Attributes	Description
data_fn	full path to data file
date_time	date and time the startup file was written
debug_level	[0 | 1 | 2] see occam documentation *default* is 1
description	brief description of inversion run *default* is 'startup created by mtpy'
diagonal_penalties	penalties on diagonal terms *default* is 0
format	Occam file format *default* is 'OCCAMITER_FLEX'
iteration	current iteration number *default* is 0
iterations_to_run	maximum number of iterations to run *default* is 20
lagrange_value	starting lagrange value *default* is 5
misfit_reached	[0 | 1] 0 if misfit has been reached, 1 if it has. *default* is 0
misfit_value	current misfit value. *default* is 1000
model_fn	full path to model file
model_limits	limits on model resistivity values *default* is None
model_value_step	limits on the step size of model values *default* is None
model_values	np.ndarray(num_free_params) of model values
param_count	number of free parameters in model
resistivity_start	starting resistivity value. If model_values is not given, then all values with in model_values array will be set to resistivity_start
roughness_type	[0 | 1 | 2] type of roughness *default* is 1
roughness_value	current roughness value. *default* is 1E10
save_path	directory path to save startup file to *default* is current working directory
startup_basename	basename of startup file name. *default* is Occam2DStartup
startup_fn	full path to startup file. *default* is save_path/startup_basename
stepsize_count	max number of iterations per step *default* is 8
target_misfit	target misfit value. *default* is 1.

Example

```
>>> startup = occam2d.Startup()
>>> startup.data_fn = ocd.data_fn
>>> startup.model_fn = profile.reg_fn
>>> startup.param_count = profile.num_free_params
>>> startup.save_path = r"/home/occam2d/Line1/Inv1"
```

write_startup_file(*startup_fn=None*, *save_path=None*, *startup_basename=None*)

Write a startup file based on the parameters of startup class. Default file name is save_path/startup_basename

Arguments:

startup_fn
[string] full path to startup file. *default* is None

save_path
[string] directory to save startup file. *default* is None

startup_basename
[string] basename of starup file. *default* is None

mtpy.modeling.occam2d_rewrite module

mtpy.modeling.occamtools module

mtpy.modeling.pek1d module

mtpy.modeling.pek1dclasses module

mtpy.modeling.pek2d module

mtpy.modeling.pek2dforward module

mtpy.modeling.structured_mesh_3d module

ModEM

Generate files for ModEM

revised by JP 2017 # revised by AK 2017 to bring across functionality from ak branch # revised by JP 2021 updating functionality and updating docs

`class mtpy.modeling.structured_mesh_3d.StructuredGrid3D`(*station_locations=None*, *center_point=None*, ***kwargs*)

Bases: `object`

make and read a FE mesh grid

The mesh assumes the coordinate system where:
x == North y == East z == + down

All dimensions are in meters.

The mesh is created by first making a regular grid around the station area, then padding cells are added that exponentially increase to the given extensions. Depth cell increase on a log10 scale to the desired depth, then padding cells are added that increase exponentially.

Parameters
`**station_object**` (*mtpy.modeling.modem.Stations object*) –

See also:

mtpy.modeling.modem.Stations

Examples

Example 1 –> create mesh first then data file

```
>>> import mtpy.modeling.modem as modem
>>> import os
>>> # 1) make a list of all .edi files that will be inverted for
>>> edi_path = r"/home/EDI_Files"
>>> edi_list = [os.path.join(edi_path, edi)
```

for edi in os.listdir(edi_path)

```
>>> ...          if edi.find('.edi') > 0]
>>> # 2) Make a Stations object
>>> stations_obj = modem.Stations()
>>> stations_obj.get_station_locations_from_edi(edi_list)
>>> # 3) make a grid from the stations themselves with 200m cell␣
↪spacing
>>> mmesh = modem.Model(station_obj)
>>> # change cell sizes
>>> mmesh.cell_size_east = 200,
>>> mmesh.cell_size_north = 200
>>> mmesh.ns_ext = 300000 # north-south extension
>>> mmesh.ew_ext = 200000 # east-west extension of model
>>> mmesh.make_mesh()
>>> # check to see if the mesh is what you think it should be
>>> msmesh.plot_mesh()
>>> # all is good write the mesh file
>>> msmesh.write_model_file(save_path=r"/home/modem/Inv1")
>>> # create data file
>>> md = modem.Data(edi_list, station_locations=mmesh.station_
↪locations)
>>> md.write_data_file(save_path=r"/home/modem/Inv1")
```

Example 2 –> Rotate Mesh

```
>>> mmesh.mesh_rotation_angle = 60
>>> mmesh.make_mesh()
```

Note: ModEM assumes all coordinates are relative to North and East, and does not accommodate mesh rotations, therefore, here the rotation is of the stations, which essentially does the same thing. You will need to rotate you data to align with the 'new' coordinate system.

Attributes	Description
_logger	python logging object that put messages in logging format defined in logging configure file, see MtPyLog more information
cell_number_ew	optional for user to specify the total number of sells on the east-west direction. *default* is None

continues on next page

Table 9 – continued from previous page

Attributes	Description
cell_number_ns	optional for user to specify the total number of sells on the north-south direction. *default* is None
cell_size_east	mesh block width in east direction *default* is 500
cell_size_north	mesh block width in north direction *default* is 500
grid_center	center of the mesh grid
grid_east	overall distance of grid nodes in east direction
grid_north	overall distance of grid nodes in north direction
grid_z	overall distance of grid nodes in z direction
model_fn	full path to initial file name
model_fn_basename	default name for the model file name
n_air_layers	number of air layers in the model. *default* is 0
n_layers	total number of vertical layers in model
nodes_east	relative distance between nodes in east direction
nodes_north	relative distance between nodes in north direction
nodes_z	relative distance between nodes in east direction
pad_east	number of cells for padding on E and W sides *default* is 7
pad_north	number of cells for padding on S and N sides *default* is 7
pad_num	number of cells with cell_size with outside of station area. *default* is 3
pad_method	method to use to create padding: extent1, extent2 - calculate based on ew_ext and ns_ext stretch - calculate based on pad_stretch factors
pad_stretch_h	multiplicative number for padding in horizontal direction.
pad_stretch_v	padding cells N & S will be pad_root_north**(x)
pad_z	number of cells for padding at bottom *default* is 4
ew_ext	E-W extension of model in meters
ns_ext	N-S extension of model in meters
res_scale	**scaling method of res, supports** ‘loge’ - for log e format ‘log’ or ‘log10’ - for log with base 10 ‘linear’ - linear scale *default* is ‘loge’
res_list	list of resistivity values for starting model
res_model	starting resistivity model
res_initial_value	resistivity initial value for the resistivity model *default* is 100
mesh_rotation_angle	Angle to rotate the grid to. Angle is measured positve clockwise assuming North is 0 and east is 90. *default* is None
save_path	path to save file to
sea_level	sea level in grid_z coordinates. *default* is 0
station_locations	location of stations
title	title in initial file
z1_layer	first layer thickness
z_bottom	absolute bottom of the model *default* is 300,000
z_target_depth	Depth of deepest target, *default* is 50,000

add_layers_to_mesh(*n_add_layers=None*, *layer_thickness=None*, *where='top'*)

Function to add constant thickness layers to the top or bottom of mesh. Note: It is assumed these layers are added before the topography. If you want to add topography layers, use function add_topography_to_model

Parameters

- **n_add_layers** – integer, number of layers to add
- **layer_thickness** – real value or list/array. Thickness of layers, defaults to z1 layer. Can provide a single value or a list/array containing multiple layer thicknesses.
- **where** – where to add, top or bottom

add_topography_from_data(*interp_method='nearest'*, *air_resistivity=1000000000000.0*, *topography_buffer=None*, *airlayer_type='log_up'*)

Wrapper around add_topography_to_model that allows creating a surface model from EDI data. The Data grid and station elevations will be used to make a 'surface' tuple that will be passed to add_topography_to_model so a surface model can be interpolated from it.

The surface tuple is of format (lon, lat, elev) containing station locations.

Parameters

- **data_object** (*mtpy.modeling.ModEM.data.Data*) – A ModEm data object that has been filled with data from EDI files.
- **interp_method** (*str, optional*) – Same as add_topography_to_model.
- **air_resistivity** (*float, optional*) – Same as add_topography_to_model.
- **topography_buffer** (*float*) – Same as add_topography_to_model.
- **airlayer_type** (*str, optional*) – Same as add_topography_to_model.

add_topography_to_model(*topography_file=None*, *surface=None*, *topography_array=None*, *interp_method='nearest'*, *air_resistivity=1000000000000.0*, *topography_buffer=None*, *airlayer_type='log_up'*, *max_elev=None*, *shift_east=0*, *shift_north=0*)

if air_layers is non-zero, will add topo: read in topograph file, make a surface model.

Call project_stations_on_topography in the end, which will re-write the .dat file.

If n_airlayers is zero, then cannot add topo data, only bathymetry is needed.

Parameters

- **topography_file** – file containing topography (arcgis ascii grid)
- **topography_array** – alternative to topography_file - array of elevation values on model grid
- **interp_method** – interpolation method for topography, 'nearest', 'linear', or 'cubic'
- **air_resistivity** – resistivity value to assign to air
- **topography_buffer** – buffer around stations to calculate minimum and maximum topography value to use for meshing
- **airlayer_type** – how to set air layer thickness - options are 'constant' for constant air layer thickness, or 'log', for logarithmically increasing air layer thickness upward

assign_resistivity_from_surface_data(*top_surface*, *bottom_surface*, *resistivity_value*)

assign resistivity value to all points above or below a surface requires the surface_dict attribute to exist and contain data for surface key (can get this information from ascii file using project_surface)

inputs surface_name = name of surface (must correspond to key in surface_dict) resistivity_value = value to assign where = 'above' or 'below' - assign resistivity above or below the

surface

convert_model_to_int(*res_list=None*)

convert resistivity values to integers according to resistivity list

Parameters
res_list (*list of floats*) – resistivity values in Ohm-m.

Returns
array of integers corresponding to the res_list

Return type
np.ndarray(dtype=int)

estimate_skin_depth(*apparent_resistivity*, *period*, *scale='km'*)

Estimate skin depth from apparent resistivity and period

Parameters

- **apparent_resistivity** (*TYPE*) – DESCRIPTION
- **period** (*TYPE*) – DESCRIPTION
- **scale** (*TYPE, optional*) – DESCRIPTION, defaults to "km"

Returns
DESCRIPTION

Return type
TYPE

from_gocad_sgrid(*sgrid_header_file*, *air_resistivity=1e+39*, *sea_resistivity=0.3*, *sgrid_positive_up=True*)

read a gocad sgrid file and put this info into a ModEM file. Note: can only deal with grids oriented N-S or E-W at this stage, with orthogonal coordinates

from_modem(*model_fn=None*)

read an initial file and return the pertinent information including grid positions in coordinates relative to the center point (0,0) and starting model.

Note that the way the model file is output, it seems is that the blocks are setup as

ModEM: WS: ———- —– 0—–> N_north 0———>N_east | | | | V V N_east N_north

Arguments:

model_fn : full path to initializing file.

Outputs:

nodes_north
[np.array(nx)] array of nodes in S –> N direction

nodes_east
[np.array(ny)] array of nodes in the W –> E direction

nodes_z
[np.array(nz)] array of nodes in vertical direction positive downwards

res_model
[dictionary] dictionary of the starting model with keys as layers

res_list
[list] list of resistivity values in the model

title
[string] title string

`from_ws3dinv`(*model_fn*)

read WS3DINV iteration model file.

Parameters
`model_fn` (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`from_ws3dinv_initial`(*initial_fn*)

read an initial file and return the pertinent information including grid positions in coordinates relative to the center point (0,0) and starting model.

Arguments:

initial_fn : full path to initializing file.

Outputs:

nodes_north
[np.array(nx)] array of nodes in S –> N direction

nodes_east
[np.array(ny)] array of nodes in the W –> E direction

nodes_z
[np.array(nz)] array of nodes in vertical direction positive downwards

res_model
[dictionary] dictionary of the starting model with keys as layers

res_list
[list] list of resistivity values in the model

title
[string] title string

`get_lower_left_corner`(*pad_east*, *pad_north*, *shift_east=0*, *shift_north=0*)

get the lower left corner in UTM coordinates for raster.

Parameters

- **pad_east** (*integer*) – number of padding cells to skip from outside in.
- **pad_north** (*integer*) – number of padding cells to skip from outside in.

Returns
Lower left hand corner

Return type
mtpy.core.MTLocation

`interpolate_elevation`(*surface_file=None*, *surface=None*, *get_surface_name=False*, *method='nearest'*, *fast=True*, *shift_north=0*, *shift_east=0*)

project a surface to the model grid and add resulting elevation data to a dictionary called surface_dict. Assumes the surface is in lat/long coordinates (wgs84)

returns nothing returned, but surface data are added to surface_dict under the key given by surface_name.

inputs choose to provide either surface_file (path to file) or surface (tuple). If both are provided then surface tuple takes priority.

surface elevations are positive up, and relative to sea level. surface file format is:

ncols 3601 nrows 3601 xllcorner -119.00013888889 (longitude of lower left) yllcorner 36.999861111111 (latitude of lower left) cellsize 0.00027777777777778 NODATA_value -9999 elevation data W –> E N | V S

Alternatively, provide a tuple with: (lon,lat,elevation) where elevation is a 2D array (shape (ny,nx)) containing elevation points (order S -> N, W -> E) and lon, lat are either 1D arrays containing list of longitudes and latitudes (in the case of a regular grid) or 2D arrays with same shape as elevation array containing longitude and latitude of each point.

other inputs: surface_epsg = epsg number of input surface, default is 4326 for lat/lon(wgs84) method = interpolation method. Default is 'nearest', if model grid is dense compared to surface points then choose 'linear' or 'cubic'

`interpolate_to_even_grid`(*cell_size*, *pad_north=None*, *pad_east=None*)

Interpolate the model onto an even grid for plotting as a raster or netCDF.

Parameters

- **cell_size** (*TYPE*) – DESCRIPTION
- **pad_north** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **pad_east** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

make_mesh(*verbose=True*)

create finite element mesh according to user-input parameters.

The mesh is built by:

1. Making a regular grid within the station area.

- Uses *cell_size_east* and *cell_size_north* for dimensions

2. Adding *pad_num* of cell_width cells outside of station area
3. Adding padding cells to given extension and number of padding cells. - *extent1* - stretch to a given distance with *pad_east* or

   pad_north number of cells.

   - *extent2* - stretch to a given distance with *pad_east* or

   pad_north number of cells.

   - *stretch* stretches from station area using

   pad_north and *pad_east* times *pad_stretch_h*

4. Making vertical cells starting with z1_layer increasing logarithmically (base 10) to z_target_depth and num_layers. - *default* creates a vertical mesh that increases

   logarithmically down. See *make_z_mesh*.

   - *custom* input your own vertical mesh.

5. Add vertical padding cells to desired extension.
6. Check to make sure none of the stations lie on a node. If they do then move the node by .02*cell_width

make_z_mesh(*n_layers=None*)

new version of make_z_mesh. make_z_mesh and M

property model_epsg

property model_fn

property model_parameters

get important model parameters to write to a file for documentation later.

property nodes_east

property nodes_north

property nodes_z

property plot_east

plot_mesh(***kwargs*)

Plot model mesh

Parameters
: **plot_topography** (*TYPE, optional*) – DESCRIPTION, defaults to False

Returns
: DESCRIPTION

Return type
: TYPE

property plot_north

property plot_z

property save_path

to_geosoft_xyz(*save_fn*, *pad_north=0*, *pad_east=0*, *pad_z=0*)

Write an XYZ file readable by Geosoft

All input units are in meters.

Parameters

- **save_fn** (*string or Path*) – full path to save file to
- **pad_north** (*int, optional*) – number of cells to cut from the north-south edges, defaults to 0
- **pad_east** (*int, optional*) – number of cells to cut from the east-west edges, defaults to 0
- **pad_z** (*int, optional*) – number of cells to cut from the bottom, defaults to 0

to_gocad_sgrid(*fn=None*, *origin=[0, 0, 0]*, *clip=0*, *no_data_value=-99999*)

write a model to gocad sgrid

optional inputs:

fn = filename to save to. File extension ('.sg') will be appended.
: default is the model name with extension removed

origin = real world [x,y,z] location of zero point in model grid clip = how much padding to clip off the edge of the model for export,

provide one integer value or list of 3 integers for x,y,z directions

no_data_value = no data value to put in sgrid

to_modem(*model_fn=None*, ***kwargs*)

will write an initial file for ModEM.

Note that x is assumed to be S –> N, y is assumed to be W –> E and z is positive downwards. This means that index [0, 0, 0] is the southwest corner of the first layer. Therefore if you build a model by hand the layer block will look as it should in map view.

Also, the xgrid, ygrid and zgrid are assumed to be the relative distance between neighboring nodes. This is needed because wsinv3d builds the model from the bottom SW corner assuming the cell width from the init file.

Key Word Arguments:

model_fn_basename
[string] basename to save file to *default* is ModEM_Model.ws file is saved at save_path/model_fn_basename

title
[string] Title that goes into the first line *default* is Model File written by MTpy.modeling.modem

res_starting_value
[float] starting model resistivity value, assumes a half space in Ohm-m *default* is 100 Ohm-m

res_scale
[['loge' | 'log' | 'log10' | 'linear']] scale of resistivity. In the ModEM code it converts everything to Loge, *default* is 'loge'

`to_netcdf`(*fn*, *pad_east=None*, *pad_north=None*, *metadata={}*)

create a netCDF file to read into GIS software

works about 50% of the time.

`to_raster`(*cell_size*, *pad_north=None*, *pad_east=None*, *save_path=None*, *depth_min=None*, *depth_max=None*, *rotation_angle=0*, *shift_north=0*, *shift_east=0*, *log10=True*, *verbose=True*)

write out each depth slice as a raster in UTM coordinates. Expecting a grid that is interoplated onto a regular grid of square cells with size *cell_size*.

Parameters

- **cell_size** (*float*) – square cell size (cell_size x cell_size) in meters.
- **pad_north** (*integer, optional*) – number of padding cells to skip from outside in, if None defaults to self.pad_north, defaults to None
- **pad_east** (*integer, optional*) – number of padding cells to skip from outside in if None defaults to self.pad_east, defaults to None
- **save_path** (*string or Path, optional*) – Path to save files to. If None use self.save_path, defaults to None
- **depth_min** (*float, optional*) – minimum depth to make raster for in meters, defaults to None which will use shallowest depth.
- **depth_max** (*float, optional*) – maximum depth to make raster for in meters, defaults to None which will use deepest depth.
- **rotation_angle** (*float, optional*) – Angle (degrees) to rotate the raster assuming clockwise positive rotation where North = 0, East = 90, defaults to 0
- **shift_north** (*float*) – shift north in meters
- **shift_east** (*float*) – shift east in meters

Raises
`ValueError` – If utm_epsg is not input.

Returns
list of file paths to rasters.

Return type
TYPE

`to_ubc`(*basename*)

Write a UBC .msh and .mod file

Parameters
`save_fn` (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

Note: not complete yet.

`to_vtk`(*vtk_save_path=None*, *vtk_fn_basename='ModEM_model_res'*, *geographic_coordinates=False*, *units='km'*, *coordinate_system='nez+'*, *label='resistivity'*)

Write a VTK file to plot in 3D rendering programs like Paraview

Parameters

- **vtk_save_path** (*string or Path, optional*) – directory to save vtk file to, defaults to None
- **vtk_fn_basename** (*string, optional*) – filename basename of vtk file, note that .vtr extension is automatically added, defaults to "ModEM_stations"
- **geographic_coordinates** (*boolean, optional*) – [True | False] True for geographic coordinates.
- **units** (*string, optional*) – Units of the spatial grid [km | m | ft], defaults to "km"

:type : string :param coordinate_system: coordinate system for the station, either the

normal MT right-hand coordinate system with z+ down or the sinister z- down [nez+ | enz-], defaults to nez+

Returns
full path to VTK file

Return type
Path

Write VTK file >>> model.write_vtk_file(vtk_fn_basename="modem_model")

Write VTK file in geographic coordinates with z+ up >>> model.write_vtk_station_file(vtk_fn_basename="modem_model", >>> … coordinate_system='enz-')

`to_winglink_out`(*save_fn*)

will write an .out file for LeapFrog.

Note that y is assumed to be S –> N, e is assumed to be W –> E and z is positive upwards. This means that index [0, 0, 0] is the southwest corner of the first layer.

Parameters
`save_fn` (*string or Path*) – full path to save file to

Returns
DESCRIPTION

Return type
TYPE

to_ws3dinv_intial(*initial_fn*, *res_list=None*)
write a WS3DINV inital model file.

to_xarray(***kwargs*)
put model in xarray format

to_xyres(*save_path=None*, *location_type='EN'*, *depth_index='all'*, *outfile_basename='DepthSlice'*, *log_res=False*, *pad_east=None*, *pad_north=None*)
write files containing depth slice data (x, y, res for each depth)

origin = x,y coordinate of zero point of ModEM_grid, or name of file
containing this info (full path or relative to model files)

save_path = path to save to, default is the model object save path location_type = 'EN' or 'LL' xy points saved as eastings/northings or

longitude/latitude, if 'LL' need to also provide model_epsg

model_epsg = epsg number that was used to project the model outfile_basename = string for basename for saving the depth slices. log_res = True/False - option to save resistivity values as log10

instead of linear

clip = number of cells to clip on each of the east/west and north/south edges

to_xyzres(*savefile=None*, *location_type='EN'*, *log_res=False*, *pad_east=None*, *pad_north=None*)
save a model file as a space delimited x y z res file

mtpy.modeling.winglink module

Created on Mon Aug 22 15:19:30 2011

deal with output files from winglink.

@author: jp

class mtpy.modeling.winglink.**PlotMisfitPseudoSection**(*data_fn*, *resp_fn*, ***kwargs*)
Bases: `object`

plot a pseudo section misfit of the data and response if given

Note: the output file from winglink does not contain errors, so to get a normalized error, you need to input the error for each component as a percent for resistivity and a value for phase and tipper. If you used the data errors, unfortunately, you have to input those as arrays.

wl_data_fn
[string] full path to output data file from winglink

key words	description
axmpte	matplotlib.axes instance for TE model phase
axmptm	matplotlib.axes instance for TM model phase
axmrte	matplotlib.axes instance for TE model app. res

continues on next page

Table 10 – continued from previous page

key words	description
axmrtm	matplotlib.axes instance for TM model app. res
axpte	matplotlib.axes instance for TE data phase
axptm	matplotlib.axes instance for TM data phase
axrte	matplotlib.axes instance for TE data app. res.
axrtm	matplotlib.axes instance for TM data app. res.
cb_pad	padding between colorbar and axes
cb_shrink	percentage to shrink the colorbar to
fig	matplotlib.figure instance
fig_dpi	resolution of figure in dots per inch
fig_num	number of figure instance
fig_size	size of figure in inches (width, height)
font_size	size of font in points
label_list	list to label plots
ml	factor to label stations if 2 every other station is labeled on the x-axis
period	np.array of periods to plot
phase_cmap	color map name of phase
phase_limits_te	limits for te phase in degrees (min, max)
phase_limits_tm	limits for tm phase in degrees (min, max)
plot_resp	['y' | 'n'] to plot response
plot_yn	['y' | 'n'] 'y' to plot on instantiation
res_cmap	color map name for resistivity
res_limits_te	limits for te resistivity in log scale (min, max)
res_limits_tm	limits for tm resistivity in log scale (min, max)
rp_list	list of dictionaries as made from read2Dresp
station_id	index to get station name (min, max)
station_list	station list got from rp_list
subplot_bottom	subplot spacing from bottom (relative coordinates)
subplot_hspace	vertical spacing between subplots
subplot_left	subplot spacing from left
subplot_right	subplot spacing from right
subplot_top	subplot spacing from top
subplot_wspace	horizontal spacing between subplots

Methods	Description
plot	plots a pseudo-section of apparent resistiviy and phase of data and model if given. called on instantiation if plot_yn is 'y'.
redraw_plot	call redraw_plot to redraw the figures, if one of the attributes has been changed
save_figu	saves the matplotlib.figure instance to desired location and format

Example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> ocd = occam2d.Occam2DData()
>>> rfile = r"/home/Occam2D/Line1/Inv1/Test_15.resp"
>>> ocd.data_fn = r"/home/Occam2D/Line1/Inv1/DataRW.dat"
>>> ps1 = ocd.plot2PseudoSection(resp_fn=rfile)
```

get_misfit()

compute misfit of MT response found from the model and the data.

Need to normalize correctly

plot()

plot pseudo section of data and response if given

redraw_plot()

redraw plot if parameters were changed

use this function if you updated some attributes and want to re-plot.

Example

```
>>> # change the color and marker of the xy components
>>> import mtpy.modeling.occam2d as occam2d
>>> ocd = occam2d.Occam2DData(r"/home/occam2d/Data.dat")
>>> p1 = ocd.plotPseudoSection()
>>> #change color of te markers to a gray-blue
>>> p1.res_cmap = 'seismic_r'
>>> p1.redraw_plot()
```

save_figure(*save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot='y'*)

save_plot will save the figure to save_fn.

Arguments:

save_fn
[string] full path to save figure to, can be input as * directory path -> the directory path to save to

in which the file will be saved as save_fn/station_name_PhaseTensor.file_format

- full path -> file will be save to the given path. If you use this option then the format will be assumed to be provided by the path

file_format
[[pdf | eps | jpg | png | svg]] file type of saved figure pdf,svg,eps...

orientation
[[landscape | portrait]] orientation in which the file will be saved *default* is portrait

fig_dpi
[int] The resolution in dots-per-inch the file will be saved. If None then the dpi will be that at which the figure was made. I don't think that it can be larger than dpi of the figure.

close_plot
[[y | n]]

- 'y' will close the plot after saving.
- 'n' will leave plot open

Example

```
>>> # to save plot as jpg
>>> import mtpy.modeling.occam2d as occam2d
>>> dfn = r"/home/occam2d/Inv1/data.dat"
>>> ocd = occam2d.Occam2DData(dfn)
>>> ps1 = ocd.plotPseudoSection()
>>> ps1.save_plot(r'/home/MT/figures', file_format='jpg')
```

update_plot()

update any parameters that where changed using the built-in draw from canvas.

Use this if you change an of the .fig or axes properties

Example

```
>>> # to change the grid lines to only be on the major ticks
>>> import mtpy.modeling.occam2d as occam2d
>>> dfn = r"/home/occam2d/Inv1/data.dat"
>>> ocd = occam2d.Occam2DData(dfn)
>>> ps1 = ocd.plotPseudoSection()
>>> [ax.grid(True, which='major') for ax in [ps1.axrte,ps1.axtep]]
>>> ps1.update_plot()
```

class mtpy.modeling.winglink.**PlotPseudoSection**(*wl_data_fn=None*, ***kwargs*)

Bases: object

plot a pseudo section of the data and response if given

wl_data_fn
[string] full path to winglink output data file.

key words	description
axmpte	matplotlib.axes instance for TE model phase
axmptm	matplotlib.axes instance for TM model phase
axmrte	matplotlib.axes instance for TE model app. res
axmrtm	matplotlib.axes instance for TM model app. res
axpte	matplotlib.axes instance for TE data phase
axptm	matplotlib.axes instance for TM data phase
axrte	matplotlib.axes instance for TE data app. res.
axrtm	matplotlib.axes instance for TM data app. res.
cb_pad	padding between colorbar and axes
cb_shrink	percentage to shrink the colorbar to
fig	matplotlib.figure instance
fig_dpi	resolution of figure in dots per inch
fig_num	number of figure instance
fig_size	size of figure in inches (width, height)
font_size	size of font in points
label_list	list to label plots
ml	factor to label stations if 2 every other station is labeled on the x-axis
period	np.array of periods to plot
phase_cmap	color map name of phase
phase_limits_te	limits for te phase in degrees (min, max)
phase_limits_tm	limits for tm phase in degrees (min, max)
plot_resp	['y' | 'n'] to plot response

continues on next page

Table 11 – continued from previous page

key words	description
plot_tipper	['y' | 'n'] to plot tipper
plot_yn	['y' | 'n'] 'y' to plot on instantiation
res_cmap	color map name for resistivity
res_limits_te	limits for te resistivity in log scale (min, max)
res_limits_tm	limits for tm resistivity in log scale (min, max)
rp_list	list of dictionaries as made from read2Dresp
station_id	index to get station name (min, max)
station_list	station list got from rp_list
subplot_bottom	subplot spacing from bottom (relative coordinates)
subplot_hspace	vertical spacing between subplots
subplot_left	subplot spacing from left
subplot_right	subplot spacing from right
subplot_top	subplot spacing from top
subplot_wspace	horizontal spacing between subplots

Methods	Description
plot	plots a pseudo-section of apparent resistiviy and phase of data and model if given. called on instantiation if plot_yn is 'y'.
redraw_plot	call redraw_plot to redraw the figures, if one of the attributes has been changed
save_figu	saves the matplotlib.figure instance to desired location and format

Example

```
>>> import mtpy.modeling.winglink as winglink
>>> d_fn = r"/home/winglink/Line1/Inv1/DataRW.txt"
>>> ps_plot = winglink.PlotPseudoSection(d_fn)
```

plot()

plot pseudo section of data and response if given

redraw_plot()

redraw plot if parameters were changed

use this function if you updated some attributes and want to re-plot.

Example

```
>>> # plot tipper and change station id
>>> import mtpy.modeling.winglink as winglink
>>> ps_plot = winglink.PlotPseudosection(wl_fn)
>>> ps_plot.plot_tipper = 'y'
>>> ps_plot.station_id = [2, 5]
>>> #label only every 3rd station
>>> ps_plot.ml = 3
>>> ps_plot.redraw_plot()
```

save_figure(*save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot='y'*)

save_plot will save the figure to save_fn.

Arguments:

save_fn
[string] full path to save figure to, can be input as * directory path -> the directory path to save to

in which the file will be saved as save_fn/station_name_PhaseTensor.file_format

- full path -> file will be save to the given path. If you use this option then the format will be assumed to be provided by the path

file_format
[[pdf | eps | jpg | png | svg]] file type of saved figure pdf,svg,eps...

orientation
[[landscape | portrait]] orientation in which the file will be saved *default* is portrait

fig_dpi
[int] The resolution in dots-per-inch the file will be saved. If None then the dpi will be that at which the figure was made. I don't think that it can be larger than dpi of the figure.

close_plot
[[y | n]]

- 'y' will close the plot after saving.
- 'n' will leave plot open

Example

```
>>> # to save plot as jpg
>>> ps_plot.save_plot(r'/home/MT/figures', file_format='jpg')
```

update_plot()

update any parameters that where changed using the built-in draw from canvas.

Use this if you change an of the .fig or axes properties

Example

```
>>> # to change the grid lines to only be on the major ticks
>>> [ax.grid(True, which='major') for ax in [ps_plot.axrte]]
>>> ps_plot.update_plot()
```

class mtpy.modeling.winglink.**PlotResponse**(*wl_data_fn=None*, *resp_fn=None*, ***kwargs*)

Bases: `object`

Helper class to deal with plotting the MT response and occam2d model.

Arguments:

data_fn
[string] full path to data file

resp_fn
[string or list] full path(s) to response file(s)

Attributes/key words	description
ax_list	list of matplotlib.axes instances for use with OccamPointPicker
color_mode	['color' | 'bw'] plot figures in color or black and white ('bw')
cted	color of Data TE marker and line
ctem	color of Model TE marker and line
ctewl	color of Winglink Model TE marker and line
ctmd	color of Data TM marker and line
ctmm	color of Model TM marker and line
ctmwl	color of Winglink Model TM marker and line
e_capsize	size of error bar caps in points
e_capthick	line thickness of error bar caps in points
err_list	list of line properties of error bars for use with OccamPointPicker
fig_dpi	figure resolution in dots-per-inch
fig_list	list of dictionaries with key words station –> station name fig –> matplotlib.figure instance axrte –> ma
fig_num	starting number of figure
fig_size	size of figure in inches (width, height)
font_size	size of axes ticklabel font in points
line_list	list of matplotlib.Line instances for use with OccamPointPicker
lw	line width of lines in points
ms	marker size in points
mted	marker for Data TE mode
mtem	marker for Model TE mode
mtewl	marker for Winglink Model TE
mtmd	marker for Data TM mode
mtmm	marker for Model TM mode
mtmwl	marker for Winglink TM mode
period	np.ndarray of periods to plot
phase_limits	limits on phase plots in degrees (min, max)
plot_model_error	['y' | 'n'] *default* is 'y' to plot model errors
plot_num	[1 | 2] 1 to plot both modes in a single plot 2 to plot modes in separate plots (default)
plot_tipper	['y' | 'n'] plot tipper data if desired
plot_type	['1' | station_list] '1' –> to plot all stations in different figures station_list –> to plot a few stations, giv
plot_yn	['y' | 'n'] 'y' –> to plot on instantiation 'n' –> to not plot on instantiation
res_limits	limits on resistivity plot in log scale (min, max)
rp_list	list of dictionaries from read2Ddata
station_list	station_list list of stations in rp_list
subplot_bottom	subplot spacing from bottom (relative coordinates)
subplot_hspace	vertical spacing between subplots
subplot_left	subplot spacing from left
subplot_right	subplot spacing from right
subplot_top	subplot spacing from top
subplot_wspace	horizontal spacing between subplots
wl_fn	Winglink file name (full path)

Methods	Description
plot	plots the apparent resistiviy and phase of data and model if given. called on instantiation if plot_yn is 'y'.
re-draw_plot	call redraw_plot to redraw the figures, if one of the attributes has been changed
save_figures	save all the matplotlib.figure instances in fig_list

Example

:: >>> data_fn = r"/home/occam/line1/inv1/OccamDataFile.dat" >>> resp_list = [r"/home/occam/line1/inv1/test_{0:02}".format(ii)

for ii in range(2, 8, 2)]

```
>>> pr_obj = occam2d.PlotResponse(data_fn, resp_list, plot_tipper='y')
```

plot()

plot the data and model response, if given, in individual plots.

redraw_plot()

redraw plot if parameters were changed

use this function if you updated some attributes and want to re-plot.

Example

```
>>> # change the color and marker of the xy components
>>> import mtpy.modeling.occam2d as occam2d
>>> ocd = occam2d.Occam2DData(r"/home/occam2d/Data.dat")
>>> p1 = ocd.plot2DResponses()
>>> #change color of te markers to a gray-blue
>>> p1.cted = (.5, .5, .7)
>>> p1.redraw_plot()
```

save_figures(*save_path*, *fig_fmt='pdf'*, *fig_dpi=None*, *close_fig='y'*)

save all the figure that are in self.fig_list

Example

```
>>> # change the color and marker of the xy components
>>> import mtpy.modeling.occam2d as occam2d
>>> ocd = occam2d.Occam2DData(r"/home/occam2d/Data.dat")
>>> p1 = ocd.plot2DResponses()
>>> p1.save_figures(r"/home/occam2d/Figures", fig_fmt='jpg')
```

exception mtpy.modeling.winglink.**WLInputError**

Bases: Exception

mtpy.modeling.winglink.**read_model_file**(*model_fn*)

readModelFile reads in the XYZ txt file output by Winglink.

Inputs:

modelfile = fullpath and filename to modelfile profiledirection = 'ew' for east-west predominantly, 'ns' for predominantly north-south. This gives column to fix

mtpy.modeling.winglink.**read_output_file**(*output_fn*)

Reads in an output file from winglink and returns the data in the form of a dictionary of structured arrays.

Arguments:

output_fn
[string] the full path to winglink outputfile

Returns:

wl_data
[dictionary with keys of station names] each station contains a structured array with keys * 'station' –> station name * 'period' –> periods to plot * 'te_res' –> TE resistivity in linear scale * 'tm_res' –> TM resistivity in linear scale * 'te_phase' –> TE phase in deg * 'tm_phase' –> TM phase in deg * 're_tip' –> real tipper amplitude. * 'im_tip' –> imaginary tipper amplitude * 'rms' –> RMS for the station * 'index' –> order from left to right of station number

Note: each data is an np.ndarray(2, num_periods) where the first index is the data and the second index is the model response

mtpy.modeling.winglinktools module

Created on Mon Aug 22 15:19:30 2011

@author: a1185872

mtpy.modeling.winglinktools.**plotResponses**(*outputfile*, *maxcol=8*, *plottype='all'*, ***kwargs*)

plotResponse will plot the responses modeled from winglink against the observed data.

Inputs:
outputfile = full path and filename to output file maxcol = maximum number of columns for the plot plottype = 'all' to plot all on the same plot

'1' to plot each respones in a different figure station to plot a single station or enter as a list of stations to plot a few stations [station1,station2]. Does not have to be verbatim but should have similar unique characters input pb01 for pb01cs in outputfile

Outputs:
None

mtpy.modeling.winglinktools.**readModelFile**(*modelfile*, *profiledirection='ew'*)

readModelFile reads in the XYZ txt file output by Winglink.

Inputs:
modelfile = fullpath and filename to modelfile profiledirection = 'ew' for east-west predominantly, 'ns' for

predominantly north-south. This gives column to fix

mtpy.modeling.winglinktools.**readOutputFile**(*outputfile*)

readOutputFile will read an output file from winglink and output data in the form of a dictionary.

Input:
outputfile = the full path and filename of outputfile

Output:

idict = dictionary with keys of station name
each idict[station name] is a dictionary with keys corresponding to modeled and observed responses:

'obsresxy','obsphasexy','modresxy','modphasexy','obsresyx', 'obsphaseyx','modresyx','modphaseyx','obshzres', 'obshzphase','modhzres','modhzphase','period'

rplst = list of dictionaries for each station with keywords:
'station' = station name 'offset' = relative offset, 'resxy' = TE resistivity and error as row 0 and 1 respectively, 'resyx'= TM resistivity and error as row 0 and 1 respectively, 'phasexy'= TE phase and error as row 0 and 1 respectively, 'phaseyx'= Tm phase and error as row 0 and 1 respectively, 'realtip'= Real Tipper and error as row 0 and 1 respectively, 'imagtip'= Imaginary Tipper and error as row 0 and 1 respectively

plst = periodlst as the median of all stations. stationlst = list of stations in order from profile title = list of parameters for plotting as [title,profile,inversiontype]

mtpy.modeling.ws3dinv module

Created on Tue Nov 7 11:42:53 2023

@author: jpeacock

class mtpy.modeling.ws3dinv.**WSData**(*mt_dataframe*, ***kwargs*)

Bases: object

Includes tools for reading and writing data files intended to be used with ws3dinv.

Example

```
>>> import mtpy.modeling.ws3dinv as ws
>>> import os
>>> edi_path = r"/home/EDI_Files"
>>> edi_list = [os.path.join(edi_path, edi) for edi in edi_path
>>> ...             if edi.find('.edi') > 0]
>>> # create an evenly space period list in log space
>>> p_list = np.logspace(np.log10(.001), np.log10(1000), 12)
>>> wsdata = ws.WSData(edi_list=edi_list, period_list=p_list,
>>> ...                     station_fn=r"/home/stations.txt")
>>> wsdata.write_data_file()
```

Attributes	Description
data	**numpy structured array with keys:** • *station* –> station name • ***east* –> relative eastern location in** grid • ***north* –> relative northern location in** grid • ***z_data* –> impedance tensor array with** shape (n_stations, n_freq, 4, dtype=complex) • ***z_data_err–> impedance tensor error without** error map applied • *z_err_map –> error map from data file
data_fn	full path to data file
edi_list	list of edi files used to make data file
n_z	[4 | 8] number of impedance tensor elements *default* is 8
ncol	number of columns in out file from winglink *default* is 5
period_list	list of periods to invert for
ptol	if periods in edi files don't match period_list then program looks for periods within ptol *defualt* is .15 or 15 percent
rotation_angle	Angle to rotate the data relative to north. Here the angle is measure clockwise from North, Assuming North is 0 and East is 90. Rotating data, and grid to align with regional geoelectric strike can improve the inversion. *default* is None
save_path	path to save the data file
station_fn	full path to station file written by WSStation
station_locations	**numpy structured array for station locations keys:** • *station* –> station name • ***east* –> relative eastern location in** grid • ***north* –> relative northern location in** grid if input a station file is written
station_east	relative locations of station in east direction
station_north	relative locations of station in north direction
station_names	names of stations
units	['mv' | 'else'] units of Z, needs to be mv for ws3dinv. *default* is 'mv'
wl_out_fn	Winglink .out file which describes a 3D grid
wl_site_fn	Wingling .sites file which gives station locations
z_data	impedance tensors of data with shape: (n_station, n_periods, 2, 2)
z_data_err	error of data impedance tensors with error map applied, shape (n_stations, n_periods, 2, 2)
z_err	[float | 'data'] 'data' to set errors as data errors or give a percent error to impedance tensor elements *default* is .05 or 5% if given as percent, ie. 5% then it is converted to .05.

Methods	Description
build_data	builds the data from .edi files
write_data_fil	writes a data file from attribute data. This way you can read in a data file, change some parameters and rewrite.
read_data_file	reads in a ws3dinv data file

property data_filename

property dataframe

get_n_stations()

get_period_df(*period*)

property period

read_data_file(*data_filename*)

read in data file

Arguments:

data_fn
[string] full path to data file

wl_sites_fn
[string] full path to sites file output by winglink. This is to match the station name with station number.

station_fn
[string] full path to station location file written by WSStation

Fills Attributes:

data
[structure np.ndarray] fills the attribute WSData.data with values

period_list
[np.ndarray()] fills the period list with values.

write_data_file(***kwargs*)

Writes a data file based on the attribute data

Key Word Arguments:

data_fn
[string] full path to data file name

save_path
[string] directory path to save data file, will be written as save_path/data_basename

data_basename
[string] basename of data file to be saved as save_path/data_basename *default* is WS-DataFile.dat

Note: if any of the data attributes have been reset, be sure to call build_data() before write_data_file.

class mtpy.modeling.ws3dinv.**WSStartup**(*data_fn=None*, *initial_fn=None*, ***kwargs*)

Bases: `object`

read and write startup files

Example

```
>>> import mtpy.modeling.ws3dinv as ws
>>> dfn = r"/home/MT/ws3dinv/Inv1/WSDataFile.dat"
>>> ifn = r"/home/MT/ws3dinv/Inv1/init3d"
>>> sws = ws.WSStartup(data_fn=dfn, initial_fn=ifn)
```

Attributes	Description
apriori_fn	full path to *a priori* model file *default* is 'default'
control_fn	full path to model index control file *default* is 'default'
data_fn	full path to data file
error_tol	error tolerance level *default* is 'default'
initial_fn	full path to initial model file
lagrange	starting lagrange multiplier *default* is 'default'
max_iter	max number of iterations *default* is 10
model_ls	model length scale *default* is 5 0.3 0.3 0.3
output_stem	output file name stem *default* is 'ws3dinv'
save_path	directory to save file to
startup_fn	full path to startup file
static_fn	full path to statics file *default* is 'default'
target_rms	target rms *default* is 1.0

read_startup_file(*startup_fn*)

read startup file fills attributes

property startup_fn

write_startup_file(*save_path*)

makes a startup file for WSINV3D.

class mtpy.modeling.ws3dinv.**WSStation**(*station_fn=None*, ***kwargs*)

Bases: `object`

read and write a station file where the locations are relative to the 3D mesh.

Attributes	Description
east	array of relative locations in east direction
elev	array of elevations for each station
names	array of station names
north	array of relative locations in north direction
station_fn	full path to station file
save_path	path to save file to

Methods	Description
read_station_file	reads in a station file
write_station_file	writes a station file
write_vtk_file	writes a vtk points file for station locations

from_wl_write_station_file(*sites_file*, *out_file*, *ncol=5*)

write a ws station file from the outputs of winglink

Arguments:

sites_fn
[string] full path to sites file output from winglink

out_fn
[string] full path to .out file output from winglink

ncol
[int] number of columns the data is in *default* is 5

read_station_file(*station_filename*)

read in station file written by write_station_file

Arguments:

station_fn
[string] full path to station file

Outputs:

east
[np.ndarray(n_stations)] relative station locations in east direction

north
[np.ndarray(n_stations)] relative station locations in north direction

elev
[np.ndarray(n_stations)] relative station locations in vertical direction

station_list
[list or np.ndarray(n_stations)] name of stations

property station_filename

write_station_file(*east=None*, *north=None*, *station_list=None*, *save_path=None*, *elev=None*)

write a station file to go with the data file.

the locations are on a relative grid where (0, 0, 0) is the center of the grid. Also, the stations are assumed to be in the center of the cell.

Arguments:

east
[np.ndarray(n_stations)] relative station locations in east direction

north
[np.ndarray(n_stations)] relative station locations in north direction

elev
[np.ndarray(n_stations)] relative station locations in vertical direction

station_list
[list or np.ndarray(n_stations)] name of stations

save_path
[string] directory or full path to save station file to if a directory the file will be saved as save_path/WS_Station_Locations.txt if save_path is none the current working directory is used as save_path

Outputs:

station_fn : full path to station file

write_vtk_file(*save_path*, *vtk_basename='VTKStations'*)
write a vtk file to plot stations

Arguments:

save_path
[string] directory to save file to. Will save as save_path/vtk_basename

vtk_basename
[string] base file name for vtk file, extension is automatically added.

Module contents

class mtpy.modeling.**StructuredGrid3D**(*station_locations=None*, *center_point=None*, ***kwargs*)
Bases: `object`

make and read a FE mesh grid

The mesh assumes the coordinate system where:
x == North y == East z == + down

All dimensions are in meters.

The mesh is created by first making a regular grid around the station area, then padding cells are added that exponentially increase to the given extensions. Depth cell increase on a log10 scale to the desired depth, then padding cells are added that increase exponentially.

Parameters
****station_object**** (*mtpy.modeling.modem.Stations object*) –

See also:

mtpy.modeling.modem.Stations

Examples

Example 1 –> create mesh first then data file

```
>>> import mtpy.modeling.modem as modem
>>> import os
>>> # 1) make a list of all .edi files that will be inverted for
>>> edi_path = r"/home/EDI_Files"
>>> edi_list = [os.path.join(edi_path, edi)
```

for edi in os.listdir(edi_path)

```
>>> ...          if edi.find('.edi') > 0]
>>> # 2) Make a Stations object
>>> stations_obj = modem.Stations()
>>> stations_obj.get_station_locations_from_edi(edi_list)
>>> # 3) make a grid from the stations themselves with 200m cell␣
↪spacing
>>> mmesh = modem.Model(station_obj)
>>> # change cell sizes
>>> mmesh.cell_size_east = 200,
>>> mmesh.cell_size_north = 200
>>> mmesh.ns_ext = 300000 # north-south extension
>>> mmesh.ew_ext = 200000 # east-west extension of model
>>> mmesh.make_mesh()
>>> # check to see if the mesh is what you think it should be
>>> msmesh.plot_mesh()
>>> # all is good write the mesh file
>>> msmesh.write_model_file(save_path=r"/home/modem/Inv1")
>>> # create data file
>>> md = modem.Data(edi_list, station_locations=mmesh.station_
↪locations)
>>> md.write_data_file(save_path=r"/home/modem/Inv1")
```

Example 2 –> Rotate Mesh

```
>>> mmesh.mesh_rotation_angle = 60
>>> mmesh.make_mesh()
```

Note: ModEM assumes all coordinates are relative to North and East, and does not accommodate mesh rotations, therefore, here the rotation is of the stations, which essentially does the same thing. You will need to rotate you data to align with the 'new' coordinate system.

Attributes	Description
_logger	python logging object that put messages in logging format defined in logging configure file, see MtPyLog more information
cell_number_ew	optional for user to specify the total number of sells on the east-west direction. *default* is None

continues on next page

Table 13 – continued from previous page

Attributes	Description
cell_number_ns	optional for user to specify the total number of sells on the north-south direction. *default* is None
cell_size_east	mesh block width in east direction *default* is 500
cell_size_north	mesh block width in north direction *default* is 500
grid_center	center of the mesh grid
grid_east	overall distance of grid nodes in east direction
grid_north	overall distance of grid nodes in north direction
grid_z	overall distance of grid nodes in z direction
model_fn	full path to initial file name
model_fn_basename	default name for the model file name
n_air_layers	number of air layers in the model. *default* is 0
n_layers	total number of vertical layers in model
nodes_east	relative distance between nodes in east direction
nodes_north	relative distance between nodes in north direction
nodes_z	relative distance between nodes in east direction
pad_east	number of cells for padding on E and W sides *default* is 7
pad_north	number of cells for padding on S and N sides *default* is 7
pad_num	number of cells with cell_size with outside of station area. *default* is 3
pad_method	method to use to create padding: extent1, extent2 - calculate based on ew_ext and ns_ext stretch - calculate based on pad_stretch factors
pad_stretch_h	multiplicative number for padding in horizontal direction.
pad_stretch_v	padding cells N & S will be pad_root_north**(x)
pad_z	number of cells for padding at bottom *default* is 4
ew_ext	E-W extension of model in meters
ns_ext	N-S extension of model in meters
res_scale	**scaling method of res, supports** 'loge' - for log e format 'log' or 'log10' - for log with base 10 'linear' - linear scale *default* is 'loge'
res_list	list of resistivity values for starting model
res_model	starting resistivity model
res_initial_value	resistivity initial value for the resistivity model *default* is 100
mesh_rotation_angle	Angle to rotate the grid to. Angle is measured positve clockwise assuming North is 0 and east is 90. *default* is None
save_path	path to save file to
sea_level	sea level in grid_z coordinates. *default* is 0
station_locations	location of stations
title	title in initial file
z1_layer	first layer thickness
z_bottom	absolute bottom of the model *default* is 300,000
z_target_depth	Depth of deepest target, *default* is 50,000

add_layers_to_mesh(*n_add_layers=None*, *layer_thickness=None*, *where='top'*)

Function to add constant thickness layers to the top or bottom of mesh. Note: It is assumed these layers are added before the topography. If you want to add topography layers, use function add_topography_to_model

Parameters

- **n_add_layers** – integer, number of layers to add
- **layer_thickness** – real value or list/array. Thickness of layers, defaults to z1 layer. Can provide a single value or a list/array containing multiple layer thicknesses.
- **where** – where to add, top or bottom

add_topography_from_data(*interp_method='nearest'*, *air_resistivity=1000000000000.0*, *topography_buffer=None*, *airlayer_type='log_up'*)

Wrapper around add_topography_to_model that allows creating a surface model from EDI data. The Data grid and station elevations will be used to make a 'surface' tuple that will be passed to add_topography_to_model so a surface model can be interpolated from it.

The surface tuple is of format (lon, lat, elev) containing station locations.

Parameters

- **data_object** (*mtpy.modeling.ModEM.data.Data*) – A ModEm data object that has been filled with data from EDI files.
- **interp_method** (*str, optional*) – Same as add_topography_to_model.
- **air_resistivity** (*float, optional*) – Same as add_topography_to_model.
- **topography_buffer** (*float*) – Same as add_topography_to_model.
- **airlayer_type** (*str, optional*) – Same as add_topography_to_model.

add_topography_to_model(*topography_file=None*, *surface=None*, *topography_array=None*, *interp_method='nearest'*, *air_resistivity=1000000000000.0*, *topography_buffer=None*, *airlayer_type='log_up'*, *max_elev=None*, *shift_east=0*, *shift_north=0*)

if air_layers is non-zero, will add topo: read in topograph file, make a surface model.

Call project_stations_on_topography in the end, which will re-write the .dat file.

If n_airlayers is zero, then cannot add topo data, only bathymetry is needed.

Parameters

- **topography_file** – file containing topography (arcgis ascii grid)
- **topography_array** – alternative to topography_file - array of elevation values on model grid
- **interp_method** – interpolation method for topography, 'nearest', 'linear', or 'cubic'
- **air_resistivity** – resistivity value to assign to air
- **topography_buffer** – buffer around stations to calculate minimum and maximum topography value to use for meshing
- **airlayer_type** – how to set air layer thickness - options are 'constant' for constant air layer thickness, or 'log', for logarithmically increasing air layer thickness upward

assign_resistivity_from_surface_data(*top_surface*, *bottom_surface*, *resistivity_value*)

assign resistivity value to all points above or below a surface requires the surface_dict attribute to exist and contain data for surface key (can get this information from ascii file using project_surface)

inputs surface_name = name of surface (must correspond to key in surface_dict) resistivity_value = value to assign where = 'above' or 'below' - assign resistivity above or below the

surface

`convert_model_to_int`(*res_list=None*)

convert resistivity values to integers according to resistivity list

Parameters
`res_list` (*list of floats*) – resistivity values in Ohm-m.

Returns
array of integers corresponding to the res_list

Return type
np.ndarray(dtype=int)

`estimate_skin_depth`(*apparent_resistivity*, *period*, *scale='km'*)

Estimate skin depth from apparent resistivity and period

Parameters

- `apparent_resistivity` (*TYPE*) – DESCRIPTION
- `period` (*TYPE*) – DESCRIPTION
- `scale` (*TYPE, optional*) – DESCRIPTION, defaults to "km"

Returns
DESCRIPTION

Return type
TYPE

`from_gocad_sgrid`(*sgrid_header_file*, *air_resistivity=1e+39*, *sea_resistivity=0.3*, *sgrid_positive_up=True*)

read a gocad sgrid file and put this info into a ModEM file. Note: can only deal with grids oriented N-S or E-W at this stage, with orthogonal coordinates

`from_modem`(*model_fn=None*)

read an initial file and return the pertinent information including grid positions in coordinates relative to the center point (0,0) and starting model.

Note that the way the model file is output, it seems is that the blocks are setup as

ModEM: WS: ———- —– 0—–> N_north 0———>N_east | | | | V V N_east N_north

Arguments:

model_fn : full path to initializing file.

Outputs:

nodes_north
[np.array(nx)] array of nodes in S –> N direction

nodes_east
[np.array(ny)] array of nodes in the W –> E direction

nodes_z
[np.array(nz)] array of nodes in vertical direction positive downwards

res_model
[dictionary] dictionary of the starting model with keys as layers

res_list
[list] list of resistivity values in the model

title
[string] title string

`from_ws3dinv`(*model_fn*)

read WS3DINV iteration model file.

Parameters
`model_fn` (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`from_ws3dinv_initial`(*initial_fn*)

read an initial file and return the pertinent information including grid positions in coordinates relative to the center point (0,0) and starting model.

Arguments:

initial_fn : full path to initializing file.

Outputs:

nodes_north
[np.array(nx)] array of nodes in S –> N direction

nodes_east
[np.array(ny)] array of nodes in the W –> E direction

nodes_z
[np.array(nz)] array of nodes in vertical direction positive downwards

res_model
[dictionary] dictionary of the starting model with keys as layers

res_list
[list] list of resistivity values in the model

title
[string] title string

`get_lower_left_corner`(*pad_east*, *pad_north*, *shift_east=0*, *shift_north=0*)

get the lower left corner in UTM coordinates for raster.

Parameters

- **pad_east** (*integer*) – number of padding cells to skip from outside in.
- **pad_north** (*integer*) – number of padding cells to skip from outside in.

Returns
Lower left hand corner

Return type
mtpy.core.MTLocation

`interpolate_elevation`(*surface_file=None*, *surface=None*, *get_surface_name=False*, *method='nearest'*, *fast=True*, *shift_north=0*, *shift_east=0*)

project a surface to the model grid and add resulting elevation data to a dictionary called surface_dict. Assumes the surface is in lat/long coordinates (wgs84)

returns nothing returned, but surface data are added to surface_dict under the key given by surface_name.

inputs choose to provide either surface_file (path to file) or surface (tuple). If both are provided then surface tuple takes priority.

surface elevations are positive up, and relative to sea level. surface file format is:

ncols 3601 nrows 3601 xllcorner -119.00013888889 (longitude of lower left) yllcorner 36.999861111111 (latitude of lower left) cellsize 0.00027777777777778 NODATA_value -9999 elevation data W –> E N | V S

Alternatively, provide a tuple with: (lon,lat,elevation) where elevation is a 2D array (shape (ny,nx)) containing elevation points (order S -> N, W -> E) and lon, lat are either 1D arrays containing list of longitudes and latitudes (in the case of a regular grid) or 2D arrays with same shape as elevation array containing longitude and latitude of each point.

other inputs: surface_epsg = epsg number of input surface, default is 4326 for lat/lon(wgs84) method = interpolation method. Default is 'nearest', if model grid is dense compared to surface points then choose 'linear' or 'cubic'

`interpolate_to_even_grid`(*cell_size*, *pad_north=None*, *pad_east=None*)

Interpolate the model onto an even grid for plotting as a raster or netCDF.

Parameters

- **cell_size** (*TYPE*) – DESCRIPTION
- **pad_north** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **pad_east** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

make_mesh(*verbose=True*)

create finite element mesh according to user-input parameters.

The mesh is built by:

1. Making a regular grid within the station area.

- Uses *cell_size_east* and *cell_size_north* for dimensions

2. Adding *pad_num* of cell_width cells outside of station area
3. Adding padding cells to given extension and number of padding cells. - *extent1* - stretch to a given distance with *pad_east* or

   pad_north number of cells.

   - *extent2* - stretch to a given distance with *pad_east* or

   pad_north number of cells.

   - *stretch* stretches from station area using

   pad_north and *pad_east* times *pad_stretch_h*

4. Making vertical cells starting with z1_layer increasing logarithmically (base 10) to z_target_depth and num_layers. - *default* creates a vertical mesh that increases

   logarithmically down. See *make_z_mesh*.

   - *custom* input your own vertical mesh.

5. Add vertical padding cells to desired extension.
6. Check to make sure none of the stations lie on a node. If they do then move the node by .02*cell_width

make_z_mesh(*n_layers=None*)

new version of make_z_mesh. make_z_mesh and M

property model_epsg

property model_fn

property model_parameters

get important model parameters to write to a file for documentation later.

property nodes_east

property nodes_north

property nodes_z

property plot_east

`plot_mesh`(***kwargs*)

Plot model mesh

Parameters
`plot_topography` (*TYPE, optional*) – DESCRIPTION, defaults to False

Returns
DESCRIPTION

Return type
TYPE

`property plot_north`

`property plot_z`

`property save_path`

`to_geosoft_xyz`(*save_fn*, *pad_north=0*, *pad_east=0*, *pad_z=0*)

Write an XYZ file readable by Geosoft

All input units are in meters.

Parameters

- `save_fn` (*string or Path*) – full path to save file to
- `pad_north` (*int, optional*) – number of cells to cut from the north-south edges, defaults to 0
- `pad_east` (*int, optional*) – number of cells to cut from the east-west edges, defaults to 0
- `pad_z` (*int, optional*) – number of cells to cut from the bottom, defaults to 0

`to_gocad_sgrid`(*fn=None*, *origin=[0, 0, 0]*, *clip=0*, *no_data_value=-99999*)

write a model to gocad sgrid

optional inputs:

fn = filename to save to. File extension ('.sg') will be appended.
default is the model name with extension removed

origin = real world [x,y,z] location of zero point in model grid clip = how much padding to clip off the edge of the model for export,

provide one integer value or list of 3 integers for x,y,z directions

no_data_value = no data value to put in sgrid

`to_modem`(*model_fn=None*, ***kwargs*)

will write an initial file for ModEM.

Note that x is assumed to be S –> N, y is assumed to be W –> E and z is positive downwards. This means that index [0, 0, 0] is the southwest corner of the first layer. Therefore if you build a model by hand the layer block will look as it should in map view.

Also, the xgrid, ygrid and zgrid are assumed to be the relative distance between neighboring nodes. This is needed because wsinv3d builds the model from the bottom SW corner assuming the cell width from the init file.

Key Word Arguments:

model_fn_basename
[string] basename to save file to *default* is ModEM_Model.ws file is saved at save_path/model_fn_basename

title
[string] Title that goes into the first line *default* is Model File written by MTpy.modeling.modem

res_starting_value
[float] starting model resistivity value, assumes a half space in Ohm-m *default* is 100 Ohm-m

res_scale
[['loge' | 'log' | 'log10' | 'linear']] scale of resistivity. In the ModEM code it converts everything to Loge, *default* is 'loge'

to_netcdf(*fn*, *pad_east=None*, *pad_north=None*, *metadata={}*)

create a netCDF file to read into GIS software

works about 50% of the time.

to_raster(*cell_size*, *pad_north=None*, *pad_east=None*, *save_path=None*, *depth_min=None*, *depth_max=None*, *rotation_angle=0*, *shift_north=0*, *shift_east=0*, *log10=True*, *verbose=True*)

write out each depth slice as a raster in UTM coordinates. Expecting a grid that is interoplated onto a regular grid of square cells with size *cell_size*.

Parameters

- **cell_size** (*float*) – square cell size (cell_size x cell_size) in meters.
- **pad_north** (*integer, optional*) – number of padding cells to skip from outside in, if None defaults to self.pad_north, defaults to None
- **pad_east** (*integer, optional*) – number of padding cells to skip from outside in if None defaults to self.pad_east, defaults to None
- **save_path** (*string or Path, optional*) – Path to save files to. If None use self.save_path, defaults to None
- **depth_min** (*float, optional*) – minimum depth to make raster for in meters, defaults to None which will use shallowest depth.
- **depth_max** (*float, optional*) – maximum depth to make raster for in meters, defaults to None which will use deepest depth.
- **rotation_angle** (*float, optional*) – Angle (degrees) to rotate the raster assuming clockwise positive rotation where North = 0, East = 90, defaults to 0
- **shift_north** (*float*) – shift north in meters
- **shift_east** (*float*) – shift east in meters

Raises
ValueError – If utm_epsg is not input.

Returns
list of file paths to rasters.

Return type
TYPE

to_ubc(*basename*)

Write a UBC .msh and .mod file

Parameters
save_fn (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

Note: not complete yet.

---

to_vtk(*vtk_save_path=None*, *vtk_fn_basename='ModEM_model_res'*, *geographic_coordinates=False*, *units='km'*, *coordinate_system='nez+'*, *label='resistivity'*)

Write a VTK file to plot in 3D rendering programs like Paraview

Parameters

- **vtk_save_path** (*string or Path, optional*) – directory to save vtk file to, defaults to None
- **vtk_fn_basename** (*string, optional*) – filename basename of vtk file, note that .vtr extension is automatically added, defaults to "ModEM_stations"
- **geographic_coordinates** (*boolean, optional*) – [True | False] True for geographic coordinates.
- **units** (*string, optional*) – Units of the spatial grid [km | m | ft], defaults to "km"

:type : string :param coordinate_system: coordinate system for the station, either the

normal MT right-hand coordinate system with z+ down or the sinister z- down [nez+ | enz-], defaults to nez+

Returns
full path to VTK file

Return type
Path

Write VTK file >>> model.write_vtk_file(vtk_fn_basename="modem_model")

Write VTK file in geographic coordinates with z+ up >>> model.write_vtk_station_file(vtk_fn_basename="modem_model", >>> … coordinate_system='enz-')

to_winglink_out(*save_fn*)

will write an .out file for LeapFrog.

Note that y is assumed to be S –> N, e is assumed to be W –> E and z is positive upwards. This means that index [0, 0, 0] is the southwest corner of the first layer.

Parameters
save_fn (*string or Path*) – full path to save file to

Returns
DESCRIPTION

Return type
TYPE

to_ws3dinv_intial(*initial_fn*, *res_list=None*)
write a WS3DINV inital model file.

to_xarray(***kwargs*)
put model in xarray format

to_xyres(*save_path=None*, *location_type='EN'*, *depth_index='all'*, *outfile_basename='DepthSlice'*, *log_res=False*, *pad_east=None*, *pad_north=None*)
write files containing depth slice data (x, y, res for each depth)

origin = x,y coordinate of zero point of ModEM_grid, or name of file
containing this info (full path or relative to model files)

save_path = path to save to, default is the model object save path location_type = 'EN' or 'LL' xy points saved as eastings/northings or

longitude/latitude, if 'LL' need to also provide model_epsg

model_epsg = epsg number that was used to project the model outfile_basename = string for basename for saving the depth slices. log_res = True/False - option to save resistivity values as log10

instead of linear

clip = number of cells to clip on each of the east/west and north/south edges

to_xyzres(*savefile=None*, *location_type='EN'*, *log_res=False*, *pad_east=None*, *pad_north=None*)
save a model file as a space delimited x y z res file

mtpy.processing package

Submodules

mtpy.processing.birrp module

BIRRP

- deals with inputs and outputs from BIRRP

Created on Tue Sep 20 14:33:20 2016

@author: jrpeacock

exception `mtpy.processing.birrp.`**BIRRPParameterError**
Bases: `Exception`

class `mtpy.processing.birrp.`**BIRRPParameters**(*ilev=0*, ***kwargs*)
Bases: `object`

class to hold and produce the appropriate parameters given the input parameters.

from_dict(*birrp_dict*)
set birrp parameters from dict

read_config_file(*birrp_config_fn*)
read in a configuration file and fill in the appropriate parameters

to_dict()

get appropriate parameters

write_config_file(*save_fn*)

write a config file for birrp parameters

class mtpy.processing.birrp.**J2Edi**(***kwargs*)

Bases: object

Read in BIRRP out puts, in this case the .j file and convert that into an .edi file using the survey_config_fn parameters.

Key Word Arguments

birrp_dir
[string] full path to directory where birrp outputs are

station
[string] station name

survey_config_fn
[string] full path to survey configuration file with information on location and site setup must have a key that is the same as station.

birrp_config_fn
[string] full path to configuration file that was used to process with (all the birrp parameters used). If None is input, the file is searched for, if it is not found, the processing parameters are used from the .j file.

j_fn
[string] full path to j file. If none is input the .j file is searched for in birrp_dir.

Methods	Description
read_survey_config_fn	read in survey configuration file
get_birrp_config_fn	get the birrp_config_fn in birrp_dir
read_birrp_config_fn	read in birrp_config_fn
get_j_file	find .j file in birrp_dir
write_edi_file	write an .edi file fro all the provided information.

Example

```
>>> import mtpy.proceessing.birrp as birrp
>>> j2edi_obj = birrp.J_To_Edi()
>>> j2edi_obj.birrp_dir = r"/home/data/mt01/BF/256"
>>> j2edi_obj.station = 'mt01'
>>> j2edi_obj.survey_config_fn = r"/home/data/2016_survey.cfg"
>>> j2edi_obj.write_edi_file()
```

get_birrp_config_fn()

get birrp configuration file from birrp directory

get_j_file(*birrp_dir=None*)

get .j file output by birrp

read_birrp_config_fn(*birrp_config_fn=None*)

read in birrp configuration file

read_survey_config_fn(*survey_config_fn=None*)

read in survey configuration file and output into a useful dictionary

write_edi_file(*station=None*, *birrp_dir=None*, *survey_config_fn=None*, *birrp_config_fn=None*, *copy_path=None*)

Read in BIRRP out puts, in this case the .j file and convert that into an .edi file using the survey_config_fn parameters.

Parameters

- `**station**` (*string*) – name of station
- `**birrp_dir**` (*string*) – full path to output directory for BIRRP
- `**survey_config_fn**` (*string*) – full path to survey configuration file
- `**birrp_config_fn**` (*string*) – full path to birrp configuration file *default* is none and is looked for in the birrp_dir
- `**copy_path**` (*string*) – full path to directory to copy the edi file to

Outputs

edi_fn
[string] full path to edi file

The survey_config_fn is a file that has the structure:

[station]
b_instrument_amplification = 1 b_instrument_type = coil b_logger_gain = 1 b_logger_type = zen b_xaxis_azimuth = 0 b_yaxis_azimuth = 90 box = 26 date = 2015/06/09 e_instrument_amplification = 1 e_instrument_type = Ag-Agcl electrodes e_logger_gain = 1 e_logger_type = zen e_xaxis_azimuth = 0 e_xaxis_length = 100 e_yaxis_azimuth = 90 e_yaxis_length = 100 elevation = 2113.2 hx = 2274 hy = 2284 hz = 2254 lat = 37.7074236995 location = Earth lon = -118.999542099 network = USGS notes = Generic config file rr_box = 25 rr_date = 2015/06/09 rr_hx = 2334 rr_hy = 2324 rr_lat = 37.6909139779 rr_lon = -119.028707542 rr_station = 302 sampling_interval = all save_path = homemtdatasurvey_01mt_01 station = 300 station_type = mt

This file can be written using mtpy.utils.configfile:

```
>>> import mtpy.utils.configfile as mtcfg
>>> station_dict = {}
>>> station_dict['lat'] = 21.346
>>> station_dict['lon'] = 122.45654
>>> station_dict['elev'] = 123.43
>>> cfg_fn = r"\home\mtdata\survey_01"
>>> mtcfg.write_dict_to_configfile({station: station_dict}, cfg_fn)
```

class mtpy.processing.birrp.**ScriptFile**(*script_fn=None*, *fn_arr=None*, ***kwargs*)

Bases: *BIRRPParameters*

class to read and write script file

Arguments

fn_arr
[numpy.ndarray] numpy.ndarray([[block 1], [block 2]])

Note: [block n] is a numpy structured array with data type

Name	Description	Type
fn	file path/name	string
nread	number of points to read	int
nskip	number of points to skip	int
comp	component	[ex | ey | hx hy | hz]
calibration_fn	calibration file path/name	string
rr	a remote reference channel	[True | False]
rr_num	remote reference pair number	int
start	start time iso format	Timestamp
stop	stop time iso format	Timestamp
station	station name	string
sampling_rate	sampling rate	int

BIRRP Parameters

parameter	description
ilev	processing mode 0 for basic and 1 for advanced RR-2 stage
nout	Number of Output time series (2 or 3-> for BZ)
ninp	Number of input time series for E-field (1,2,3)
nref	Number of reference channels (2 for MT)
nrr	bounded remote reference (0) or 2 stage bounded influence (1)
tbw	Time bandwidth for Sepian sequence
deltat	Sampling rate (+) for (s), (-) for (Hz)
nfft	Length of FFT (should be even)
nsctinc	section increment divisor (2 to divide by half)
nsctmax	Number of windows used in FFT
nf1	1st frequency to extract from FFT window (>=3)
nfinc	frequency extraction increment
nfsect	number of frequencies to extract
mfft	AR filter factor, window divisor (2 for half)
uin	Quantile factor determination
ainlin	Residual rejection factor low end (usually 0)

continues on next page

Table 14 – continued from previous page

parameter	description
ainuin	Residual rejection factor high end (.95-.99)
c2threshb	Coherence threshold for magnetics (0 if undesired)
c2threshe	Coherence threshold for electrics (0 if undesired)
nz	**Threshold for Bz (0=separate from E, 1=E threshold,** 2=E and B) Input if 3 B components else None
c2thresh1	Squared coherence for Bz, input if NZ=0, Nout=3
perlo	longest period to apply coherence threshold over
perhi	shortes period to apply coherence threshold over
ofil	**Output file root(usually three letters, can add full** path)
nlev	Output files (0=Z; 1=Z,qq; 2=Z,qq,w; 3=Z,qq,w,d)
nprej	number of frequencies to reject
prej	frequencies to reject (+) for period, (-) for frequency
npcs	Number of independent data to be processed (1 for one segement)
nar	Prewhitening Filter (3< >15) or 0 if not desired',
imode	Output file mode (0=ascii; 1=binary; 2=headerless ascii; 3=ascii in TS mode',
jmode	**input file mode (0=user defined; 1=sconvert2tart time** YYYY-MM-DD HH:MM:SS)',
nread	Number of points to be read for each data set (if segments>1 -> npts1,npts2…)',
nfil	**Filter parameters (0=none; >0=input parameters;** <0=filename)
nskip	**Skip number of points in time series (0) if no skip,** (if segements >1 -> input1,input2…)',
nskipr	Number of points to skip over (0) if none, (if segements >1 -> input1,input2…)',
thetae	Rotation angles for electrics (relative to geomagnetic North)(N,E,rot)',
thetab	Rotation angles for magnetics (relative to geomagnetic North)(N,E,rot)',
thetar	Rotation angles for calculation (relative to geomagnetic North)(N,E,rot)'

Note: Currently only supports jmode = 0 and imode = 0

See also:

BIRRP Manual and publications by Chave and Thomson for more details on the parameters found at:

http://www.whoi.edu/science/AOPE/people/achave/Site/Next1.html

property comp_list

property deltat

make_fn_lines_block_00(*fn_arr*)

make lines for file in script file which includes

- nread
- filter_fn
- fn
- nskip

make_fn_lines_block_n(*fn_arr*)

make lines for file in script file which includes

- nread
- filter_fn
- fn
- nskip

property nout

property npcs

property nref

write_script_file(*script_fn=None*, *ofil=None*)

exception mtpy.processing.birrp.**ScriptFileError**

Bases: Exception

mtpy.processing.birrp.**run**(*birrp_exe*, *script_file*)

run a birrp script file from command line via python subprocess.

Parameters

- ****birrp_exe**** (*string*) – full path to the compiled birrp executable
- ****script_file**** (*string*) – full path to input script file following the guidelines of the BIRRP documentation.

Outputs

log_file.log : a log file of how BIRRP ran

See also:

BIRRP Manual and publications by Chave and Thomson for more details on the parameters found at:

http://www.whoi.edu/science/AOPE/people/achave/Site/Next1.html

mtpy.processing.filter module

mtpy/processing/filter.py

Functions for the frequency filtering of raw time series.

@UofA, 2013 (LK)

Revised 2017 JP

`mtpy.processing.filter.adaptive_notch_filter`(*bx*, *df*, *notches=[50, 100]*, *notchradius=.3*, *freqrad=.9*)

will apply a notch filter to the array bx by finding the nearest peak around the supplied notch locations. The filter is a zero-phase Chebyshev type 1 bandstop filter with minimal ripples.

Arguments:

bx
[np.ndarray(len_time_series)] time series to filter

df
[float] sampling frequency in Hz

notches : list of frequencies (Hz) to filter

notchradius
[float] radius of the notch in frequency domain (Hz)

freqrad
[float] radius to searching for peak about notch from notches

rp
[float] ripple of Chebyshev type 1 filter, lower numbers means less ripples

dbstop_limit
[float (in decibels)] limits the difference between the peak at the notch and surrounding spectra. Any difference above dbstop_limit will be filtered, anything less will not

Outputs:

bx
[np.ndarray(len_time_series)] filtered array

filtlst
[list] location of notches and power difference between peak of notch and average power.

..Example:

```
>>> import RemovePeriodicNoise_Kate as rmp
>>> # make a variable for the file to load in
>>> fn = r"/home/MT/mt01_20130101_000000.BX"
>>> # load in file, if the time series is not an ascii file
>>> # might need to add keywords to np.loadtxt or use another
>>> # method to read in the file
>>> bx = np.loadtxt(fn)
>>> # create a list of frequencies to filter out
>>> freq_notches = [50, 150, 200]
```

(continues on next page)

(continued from previous page)

```
>>> # filter data
>>> bx_filt, filt_lst = rmp.adaptiveNotchFilter(bx, df=100.
>>> ...                                         notches=freq_notches)
>>> #save the filtered data into a file
>>> np.savetxt(r"/home/MT/Filtered/mt01_20130101_000000.BX", bx_filt)
```

Notes:

Most of the time the default parameters work well, the only thing you need to change is the notches and perhaps the radius. I would test it out with a few time series to find the optimum parameters. Then make a loop over all you time series data. Something like

```
>>> import os
>>> dirpath = r"/home/MT"
>>> #make a director to save filtered time series
>>> save_path = r"/home/MT/Filtered"
>>> if not os.path.exists(save_path):
>>>     os.mkdir(save_path)
>>> for fn in os.listdir(dirpath):
>>>     bx = np.loadtxt(os.path.join(dirpath, fn)
>>>     bx_filt, filt_lst = rmp.adaptiveNotchFilter(bx, df=100.
>>>     ...                                         notches=freq_notches)
>>>     np.savetxt(os.path.join(save_path, fn), bx_filt)
```

`mtpy.processing.filter.`**`butter_bandpass`**(*lowcut*, *highcut*, *samplingrate*, *order=4*)

`mtpy.processing.filter.`**`butter_bandpass_filter`**(*data*, *lowcut*, *highcut*, *samplingrate*, *order=4*)

`mtpy.processing.filter.`**`low_pass`**(*f*, *low_pass_freq*, *cutoff_freq*, *sampling_rate*)

`mtpy.processing.filter.`**`remove_periodic_noise`**(*filename*, *dt*, *noiseperiods*, *save='n'*)

removePeriodicNoise will take a window of length noise period and compute the median of signal for as many windows that can fit within the data. This median window is convolved with a series of delta functions at each window location to create a noise time series. This is then subtracted from the data to get a 'noise free' time series.

Arguments:

filename
: [string (full path to file) or array] name of file to have periodic noise removed from can be an array

dt
: [float] time sample rate (s)

noiseperiods
: [list] a list of estimated periods with a range of values to look around [[noiseperiod1,df1]. . .] where df1 is a fraction value find the peak about noiseperiod1 must be less than 1. (0 is a good start, but if you're periodic noise drifts, might need to adjust df1 to .2 or something)

save
: [['y' | 'n']]

- **'y' to save file to:**
  os.path.join(os.path.dirname(filename), 'Filtered', fn)
- 'n' to return the filtered time series

Outputs:

bxnf
[np.ndarray] filtered time series

pn
[np.ndarray] periodic noise time series

fitlst
[list] list of peaks found in time series

..Example:

```
>>> import RemovePeriodicNoise_Kate as rmp
>>> # make a variable for the file to load in
>>> fn = r"/home/MT/mt01_20130101_000000.BX"
>>> # filter data assuming a 12 second period in noise and save data
>>> rmp.remove_periodic_noise(fn, 100., [[12,0]], save='y')
```

Notes:

Test out the periodic noise period at first to see if it drifts. Then loop over files

```
>>> import os
>>> dirpath = r"/home/MT"
>>> for fn in os.listdir(dirpath):
>>>     rmp.remove_periodic_noise(fn, 100., [[12,0]], save='y')
```

`mtpy.processing.filter.`**`tukey`**(*window_length*, *alpha=0.2*)

The Tukey window, also known as the tapered cosine window, can be regarded as a cosine lobe of width alpha * N / 2 that is convolved with a rectangle window of width (1 - alpha / 2). At alpha = 0 it becomes rectangular, and at alpha = 1 it becomes a Hann window.

output

Reference

http://www.mathworks.com/access/helpdesk/help/toolbox/signal/tukeywin.html

`mtpy.processing.filter.`**`zero_pad`**(*input_array*, *power=2*, *pad_fill=0*)

pad the input array with pad_fill to the next power of power.

For faster fft computation pad the array to the next power of 2 with zeros

Arguments:

input_array
[np.ndarray (only 1-d arrays are supported at the] moment)

power
[[2 | 10]] power look for

pad_fill
[float or int] pad the array with this

Output:

pad_array : np.ndarray padded with pad_fill

mtpy.processing.tf module

mtpy/processing/tf.py

Functions for the time-frequency analysis of time series data.

Output can be visualised with the help of mtpy/imaging/spectrogram.py

JP, 2013

`mtpy.processing.tf.`**`dctrend`**`(`*f*`)`

dctrend(f) will remove a dc trend from the function f.

Arguments:

f
[np.ndarray()] array to remove dc trend from

Returns:

fdc
[np.ndarray()] array f with dc component removed

`mtpy.processing.tf.`**`decimate`**`(`*f*, *m*, *window_function='hanning'*`)`

resamples the data at the interval m so that the returned array is len(f)/m samples long

Arguments:

f
[np.ndarray] array to be decimated

m
[int] decimation factor

window_function
[windowing function to apply to the data] to make sure the is no Gibbs ringing or aliasing see scipy.signal.window for all the options

Returns:

fdec
[np.ndarray()] array f decimated by factor m

`mtpy.processing.tf.dwindow(window)`

Calculates the derivative of the given window. Used for reassignment methods

Arguments:

window
[np.ndarray] some sort of windowed array

Returns:

dwin
[np.ndarray] derivative of window

`mtpy.processing.tf.gausswin(window_len, alpha=2.5)`

gausswin will compute a gaussian window of length winlen with a variance of alpha

Arguments:

window_len: int
length of desired window

alpha
[float] 1/standard deviation of window, ie full width half max of window

Returns:

gauss_window
[np.array] gaussian window

`mtpy.processing.tf.modifiedb(fx, tstep=32, nfbins=1024, df=1.0, nh=255, beta=0.2)`

Calculates the modified b distribution as defined by cosh(n)^-2 beta for an array fx. Supposed to remove cross terms in the WVD.

Arguments:

fx
[list or np.ndarray] the function to have a spectrogram computed for for cross-correlation input as [fx1, fx2]

nh
[int (should be odd)] window length for each time step *default* is None and window is calculated automatically

tstep
[int] number of sample between short windows *default* is 2**5 = 32

df
[float] sampling frequency

nfbins
[int (should be power of 2 and equal or larger than nh)] number of frequency bins

beta
[float] smoothing coefficient ussully between [0, 1]

Returns:

tfarray
[np.ndarray(nfbins/2, len(fx)/tstep)] SPWVD spectrogram in units of amplitude

tlst
[np.array()] array of time instances for each window calculated

flst
[np.ndarray(nfbins/2)] frequency array containing only positive frequencies where the Fourier coeffients were calculated

`mtpy.processing.tf.normalize_L2(f)`

normalize_L2(f) returns the array f normalized by the L2 norm -> f/(sqrt(sum(abs(x_i)^2))).

Arguments

f
[np.ndarray()] array to be normalized

Returns:

fnorm
[np.ndarray()] array f normalized in L2 sense

`mtpy.processing.tf.padzeros(f, npad=None, pad_pattern=None)`

padzeros(f) returns a function that is padded with zeros to the next power of 2 for faster processing for fft or to length npad if given.

Arguments:

f
[np.ndarray(m, n)] array to pad

npad
[int] length to pad to if None finds next power of 2

pad_pattern
[int or float] pattern to pad with if None set zeros

Returns:

fpad
[np.ndarray(m, npad)] array f padded to length npad with pad_pattern

Example
:: >>> x_array = np.sin(np.arange(0, 2, .01)*np.pi/3) >>> print len(x_array) >>> x_array_pad = padzeros(x_array) >>> print len(x_array_pad)

`mtpy.processing.tf.reassigned_smethod`(*fx*, *nh=127*, *tstep=16*, *nfbins=512*, *df=1.0*, *alpha=4*, *thresh=0.01*, *L=5*)

Calulates the reassigned S-method as described by Djurovic[1999] by using the spectrogram to estimate the reassignment.

Arguments:

for cross-correlation input as [fx1, fx2]

L
[int (should be odd)] length of window for S-method calculation, higher numbers tend toward WVD

nh
[int (should be power of 2)] window length for each time step *default* is 2**8 = 256

alpha
[float] inverse of full-width half max of gaussian window, smaller numbers mean broader windows.

thresh
[float] threshold for reassignment, lower numbers more points reassigned, higer numbers less points reassigned

tstep
[int] number of sample between short windows *default* is 2**7 = 128

df
[float] sampling frequency

nfbins
[int (should be power of 2 and equal or larger than nh)] number of frequency bins

Returns:

tfarray
[np.ndarray(nfbins/2, len(fx)/tstep)] S-method spectrogram in units of amplitude

tlst
[np.array()] array of time instances for each window calculated

flst
[np.ndarray(nfbins/2)] frequency array containing only positive frequencies where the Fourier coeffients were calculated

sm
[np.ndarray(nfbins/2, len(fx)/tstep)] S-method spectrogram in units of amplitude

`mtpy.processing.tf.reassigned_stft`(*fx*, *nh=63*, *tstep=32*, *nfbins=1024*, *df=1.0*, *alpha=4*, *threshold=None*)

Computes the reassigned spectrogram by estimating the center of gravity of the signal and condensing dispersed energy back to that location. Works well for data with minimal noise and strong spectral structure.

Arguments:

fx
[np.ndarray] time series to be analyzed

nh
[int(should be odd)] length of gaussian window that is applied to the short time intervals *default* is 127

tstep
[int] time step for each window calculation *default* is 64

nfbins
[int (should be a power of 2 and larger or equal to nh] number of frequency bins to calculate, note result will be length nfbins/2 *default* is 1024

df
[float or int] sampling frequency (Hz)

alpha
[float] reciprocal of full width half max of gaussian window *default* is 4

threshold
[float] threshold value for reassignment If None the threshold is automatically calculated *default* is None

returns

np.ndarray(nfbins/2, len(fx)/tstep)
reassigned spectrogram in units of amplitude

tlst
[np.array()] array of time instances for each window calculated

flst
[np.ndarray(nfbins/2)] frequency array containing only positive frequencies where the Fourier coeffients were calculated

stft
[np.ndarray(nfbins/2, len(fx)/tstep)] standard spectrogram calculated from stft in units of amplitude

rtype
rtfarray

`mtpy.processing.tf.robust_smethod`(*fx*, *L=5*, *nh=128*, *tstep=32*, *nfbins=1024*, *df=1.0*, *robusttype='median'*, *sigmaL=None*, *alpha=0.325*)

Computes the robust Smethod via the robust spectrogram.

Arguments:

fx
: [list or np.ndarray] the function to have a spectrogram computed for for cross-correlation input as [fx1, fx2]

L
: [int (should be odd)] length of window for S-method calculation, higher numbers tend toward WVD

nh
: [int (should be power of 2)] window length for each time step *default* is 2**8 = 256

ng
: [int (should be odd)] length of smoothing window along frequency plane

tstep
: [int] number of sample between short windows *default* is 2**7 = 128

df
: [float] sampling frequency

nfbins
: [int (should be power of 2 and equal or larger than nh)] number of frequency bins

robusttype
: [['median' | 'L']] type of robust STFT to compute. *default* is 'median'

simgaL
: [float] full-width half max of gaussian window applied in frequency

Returns:

tfarray
: [np.ndarray(nfbins/2, len(fx)/tstep)] S-method spectrogram in units of amplitude

tlst
: [np.array()] array of time instances for each window calculated

flst
: [np.ndarray(nfbins/2)] frequency array containing only positive frequencies where the Fourier coeffients were calculated

pxx
: [np.ndarray(nfbins/2, len(fx)/tstep)] STFT spectrogram in units of amplitude

`mtpy.processing.tf.`**`robust_stft_L`**(*fx*, *alpha=0.325*, *nh=256*, *tstep=32*, *df=1.0*, *nfbins=1024*)

Calculates the robust spectrogram by estimating the vector median and summing terms estimated by alpha coefficients.

Arguments:

fx
[list or np.ndarray] the function to have a spectrogram computed for for cross-correlation input as [fx1, fx2]

alpha
[float] robust parameter [0,.5] -> 0 gives spectrogram, 0.5 gives median stft *default* is 0.325

nh
[int (should be power of 2)] window length for each time step *default* is 2**8 = 256

tstep
[int] number of sample between short windows *default* is 2**7 = 128

df
[float] sampling frequency

nfbins
[int (should be power of 2 and equal or larger than nh)] number of frequency bins

Returns:

tfarray
[np.ndarray(nfbins/2, len(fx)/tstep)] spectrogram in units of amplitude

tlst
[np.array()] array of time instances for each window calculated

flst
[np.ndarray(nfbins/2)] frequency array containing only positive frequencies where the Fourier coeffients were calculated

`mtpy.processing.tf.`**`robust_stft_median`**(*fx*, *nh=256*, *tstep=32*, *df=1.0*, *nfbins=1024*)

Calculates the robust spectrogram using the vector median simplification.

Arguments:

fx
[list or np.ndarray] the function to have a spectrogram computed for for cross-correlation input as [fx1, fx2]

nh
[int (should be power of 2)] window length for each time step *default* is 2**8 = 256

tstep
[int] number of sample between short windows *default* is 2**7 = 128

df
[float] sampling frequency (Hz)

nfbins
[int (should be power of 2 and equal or larger than nh)] number of frequency bins

Returns:

tfarray
[np.ndarray(nfbins/2, len(fx)/tstep)] spectrogram in units of amplitude

tlst
[np.array()] array of time instances for each window calculated

flst
[np.ndarray(nfbins/2)] frequency array containing only positive frequencies where the Fourier coeffients were calculated

`mtpy.processing.tf.`**`robust_wvd`**(*fx*, *nh=127*, *ng=15*, *tstep=16*, *nfbins=256*, *df=1.0*, *sigmat=None*, *sigmaf=None*)

Calculate the robust Wigner-Ville distribution for an array fx. Smoothed with Gaussians windows to get best localization.

Arguments:

fx
[list or np.ndarray] the function to have a spectrogram computed for for cross-correlation input as [fx1, fx2]

nh
[int (should be power of 2)] window length for each time step *default* is 2**8 = 256

tstep
[int] number of sample between short windows *default* is 2**7 = 128

ng
[int (should be odd)] length of smoothing window along frequency plane *default* is 2**4-1 = 15

df
[float] sampling frequency

nfbins
[int (should be power of 2 and equal or larger than nh)] number of frequency bins

sigmat
[float] std of window h, ie full width half max of gaussian *default* is None and sigmat is calculate automatically

sigmaf
[float] std of window g, ie full width half max of gaussian *default* is None and sigmaf is calculate automatically

Returns:

tfarray
[np.ndarray(nfbins/2, len(fx)/tstep)] spectrogram in units of amplitude

tlst
[np.array()] array of time instances for each window calculated

flst
[np.ndarray(nfbins/2)] frequency array containing only positive frequencies where the Fourier coeffients were calculated

`mtpy.processing.tf.sinc_filter`(*f*, *fcutoff=10.0*, *w=10.0*, *dt=0.001*)

Applies a sinc filter of width w to the function f by multipling in the frequency domain.

Arguments:

f
[np.ndarray()] array to filter

fcuttoff
[float] cutoff frequency of sinc filter

w
[float] length of filter

dt
[float] sampling rate in time (s)

Returns:

f_filt
[np.ndarray()] f with sinc filter applied

`mtpy.processing.tf.smethod`(*fx*, *L=11*, *nh=256*, *tstep=128*, *ng=1*, *df=1.0*, *nfbins=1024*, *sigmaL=None*)

Calculates the smethod by estimating the STFT first and computing the WV of window length L in the frequency domain.

For larger L more of WV estimation, if L=0 get back STFT

Arguments:

fx
[list or np.ndarray] the function to have a spectrogram computed for for cross-correlation input as [fx1, fx2]

L
[int (should be odd)] length of window for S-method calculation, higher numbers tend toward WVD

nh
[int (should be power of 2)] window length for each time step *default* is 2**8 = 256

ng
[int (should be odd)] length of smoothing window along frequency plane

tstep
[int] number of sample between short windows *default* is 2**7 = 128

df
[float] sampling frequency

nfbins
[int (should be power of 2 and equal or larger than nh)] number of frequency bins

Returns:

tfarray
[np.ndarray(nfbins/2, len(fx)/tstep)] S-method spectrogram in units of amplitude

tlst
[np.array()] array of time instances for each window calculated

flst
[np.ndarray(nfbins/2)] frequency array containing only positive frequencies where the Fourier coeffients were calculated

pxx
[np.ndarray(nfbins/2, len(fx)/tstep)] STFT spectrogram in units of amplitude

`mtpy.processing.tf.`**`specwv`**(*fx*, *tstep=32*, *nfbins=1024*, *nhs=256*, *nhwv=511*, *ngwv=7*, *df=1.0*)

Calculates the Wigner-Ville distribution mulitplied by the STFT windowed by the common gaussian window h for an array f. Handy for removing cross terms in the wvd.

Arguments:

fx
[list or np.ndarray] the function to have a spectrogram computed for

tstep
[int]

number of sample between short windows *default* is 2**7 = 128

nhs
[int (should be power of 2)] window length for each time step to caclulate STFT *default* is 2**8 = 256 and window is calculated automatically

nhwv
[int (should be odd)] length of smoothing window for each time step to calculate WVD. *default* is 2**9-1 = 511

ngwv
[int (should be odd)] length of frequency smoothing window for each time step to calculate WVD. *default* is 2**3-1 = 7

df
[float] sampling frequency

nfbins
[int (should be power of 2 and equal or larger than nh)] number of frequency bins

Returns:

tfarray
[np.ndarray(nfbins/2, len(fx)/tstep)] SPWVD spectrogram in units of amplitude

tlst
[np.array()] array of time instances for each window calculated

flst
[np.ndarray(nfbins/2)] frequency array containing only positive frequencies where the Fourier coeffients were calculated

`mtpy.processing.tf.`**`spwvd`**(*fx*, *tstep=32*, *nfbins=1024*, *df=1.0*, *nh=None*, *ng=None*, *sigmat=None*, *sigmaf=None*)

Calculates the smoothed pseudo Wigner-Ville distribution for an array fx. Smoothed with Gaussians windows to get best localization.

Can be input as [fx1, fx2] to compute cross spectra.

Arguments:

fx
[list or np.ndarray] the function to have a spectrogram computed for for cross-correlation input as [fx1, fx2]

nh
[int (should be odd)] window length for each time step *default* is None and window is calculated automatically

tstep
[int] number of sample between short windows *default* is 2**7 = 128

ng
[int (should be odd)] length of smoothing window along frequency plane

df
[float] sampling frequency

nfbins
[int (should be power of 2 and equal or larger than nh)] number of frequency bins

sigmat
[float] std of window h, ie full width half max of gaussian *default* is None and sigmat is calculate automatically

sigmaf
[float] std of window g, ie full width half max of gaussian *default* is None and sigmaf is calculate automatically

Returns:

tfarray
[np.ndarray(nfbins/2, len(fx)/tstep)] SPWVD spectrogram in units of amplitude

tlst
[np.array()] array of time instances for each window calculated

flst
[np.ndarray(nfbins/2)] frequency array containing only positive frequencies where the Fourier coeffients were calculated

`mtpy.processing.tf.stfbss`(*X*, *nsources=5*, *ng=31*, *nh=511*, *tstep=63*, *df=1.0*, *nfbins=1024*, *tftol=1e-08*, *L=7*, *normalize=True*, *tftype='spwvd'*, *alpha=0.38*)

btfssX,nsources=5,ng=25-1,nh=2**9-1,tstep=2**6-1,df=1.0,nfbins=2**10,**
tftol=1.E-8,normalize=True)

estimates sources using a blind source algorithm based on spatial time-frequency distributions. At the moment this algorithm uses the SPWVD to estimate TF distributions.

Arguments

X = m x n array of time series, where m is number of time series and n
is length of each time series

nsources = number of estimated sources ng = frequency window length nh = time window length tstep = time step increment df = sampling frequency (Hz) nfbins = number of frequencies tftol = tolerance for a time-frequency point to be estimated as a cross

term or as an auto term, the higher the number the more auto terms.

normalization = True or False, True to normalize, False if already
normalized

Returns
Se = estimated individual signals up to a permutation and scale Ae = estimated mixing matrix as X=A*S

`mtpy.processing.tf.stft`(*fx*, *nh=256*, *tstep=128*, *ng=1*, *df=1.0*, *nfbins=1024*)

calculate the spectrogam of the given function by calculating the fft of a window of length nh at each time instance with an interval of tstep. The frequency resolution is nfbins.

Can compute the cross STFT by inputting fx as [fx1, fx2]

Arguments:

fx
[list or np.ndarray] the function to have a spectrogram computed for for cross-correlation input as [fx1, fx2]

nh
[int (should be power of 2)] window length for each time step *default* is 2**8 = 256

tstep
[int] number of sample between short windows *default* is 2**7 = 128

ng
[int (should be odd)] length of smoothing window along frequency plane

df
[float] sampling frequency

nfbins
[int (should be power of 2 and equal or larger than nh)] number of frequency bins

Returns:

tfarray
[np.ndarray(nfbins/2, len(fx)/tstep)] spectrogram in units of amplitude

tlst
[np.array()] array of time instances for each window calculated

flst
[np.ndarray(nfbins/2)] frequency array containing only positive frequencies where the Fourier coeffients were calculated

`mtpy.processing.tf.wvd`(*fx*, *nh=255*, *tstep=32*, *nfbins=1024*, *df=1.0*)

calculates the Wigner-Ville distribution of f.

Can compute the cross spectra by inputting fx as [fx1,fx2]

Arguments:

fx
[list or np.ndarray] the function to have a spectrogram computed for for cross-correlation input as [fx1, fx2]

nh
[int (should be odd)] window length for each time step *default* is 2**8-1 = 255

tstep
[int] number of sample between short windows *default* is 2**7 = 128

df
[float] sampling frequency

nfbins
[int (should be power of 2 and equal or larger than nh)] number of frequency bins

Returns:

tfarray
[np.ndarray(nfbins/2, len(fx)/tstep)] spectrogram in units of amplitude

tlst
[np.array()] array of time instances for each window calculated

flst
[np.ndarray(nfbins/2)] frequency array containing only positive frequencies where the Fourier coeffients were calculated

`mtpy.processing.tf.`**`wvd_analytic_signal`**(*fx*)

Computes the analytic signal for WVVD as defined by J. M. O' Toole, M. Mesbah, and B. Boashash, (2008), "A New Discrete Analytic Signal for Reducing Aliasing in the

Discrete Wigner-Ville Distribution", IEEE Trans. on Signal Processing,

Argument:

fx
[np.ndarray()] signal to compute anlytic signal for with length N

Returns:

fxa
[np.ndarray()] analytic signal of fx with length 2*N

Module contents

mtpy.utils package

Submodules

mtpy.utils.array2raster module

mtpy.utils.basemap_tools module

`mtpy.utils.basemap_tools.`**`add_basemap_frame`**(*basemap*, *tick_interval=None*, *coastline_kwargs={}*, *states_kwargs={}*, *mlabels=[False, False, False, True]*, *plabels=[True, False, False, False]*)

add a standard map frame (lat/lon labels and tick marks, coastline and states) to basemap

Parameters

- **`tick_interval`** – tick interval in degrees
- **`coastline_kwargs`** – dictionary containing arguments to pass into the drawcoastlines function
- **`states_kwargs`** – dictionary containing arguments to pass into the drawstates function
- **`mlabels`** – where to place meridian (longitude) labels on plot (list containing True/False for [left,right,top,bottom])
- **`plabels`** – where to place parallels (latitudes) labels on plot (list containing True/False for [left,right,top,bottom])

`mtpy.utils.basemap_tools.`**`compute_lonlat0_from_modem_data`**(*stations_obj*)

compute lat0 and lon0 for creating a basemap, using data centre point in modem data file

`mtpy.utils.basemap_tools.`**`compute_map_extent_from_modem_data`**(*stations_obj*, *buffer=None*, *buffer_factor=0.1*)

compute extent for a plot from data extent from ModEM data file

Parameters

- **data_fn** – full path to modem data file
- **buffer** – optional argument; buffer in latitude/longitude (if not provided,

this is assumed to be a fraction of the maximum of the north-south or east-west extent) :param buffer_factor: fraction of north-south or east-west extent for buffer (if buffer not provided)

mtpy.utils.basemap_tools.**compute_tick_interval_from_map_extent**(*lonMin*, *lonMax*, *latMin*, *latMax*)
estimate an even tick interval based on map extent based on some sensible options

mtpy.utils.basemap_tools.**get_latlon_extents_from_modem_data**(*stations_obj*)

mtpy.utils.basemap_tools.**initialise_basemap**(*stations_obj*, *buffer=None*, ***basemap_kwargs*)
create a new basemap instance

mtpy.utils.basemap_tools.**plot_data**(*x*, *y*, *values*, *basemap=None*, *cbar=False*, ***param_dict*)
plot array data, either 1d or 2d

Parameters

- **x** – x position of points
- **y** – y position of points
- **values** – values to plot, if 1D, a scatter plot will be made, if 2D, a pcolormesh plot will be made
- **basemap** – supply a basemap, if None, data will be plotted on current axes
- **cbar** – True/False, whether or not to show a colorbar

mtpy.utils.calculator module

mtpy/utils/calculator.py

Helper functions for standard calculations, e.g. error propagation

@UofA, 2013 (LK)

mtpy.utils.calculator.**centre_point**(*xarray*, *yarray*)
get the centre point of arrays of x and y values

mtpy.utils.calculator.**compute_determinant_error**(*z_array*, *z_err_array*, *method='theoretical'*, *repeats=1000*)
compute the error of the determinant of z using a stochastic method seed random z arrays with a normal distribution around the input array

Parameters

- **z_array** – z (impedance) array containing real and imaginary values
- **z_err_array** – impedance error array containing real values, in MT we assume the real and imag errors are the same
- **method** – method to use, theoretical calculation or stochastic

Returns
error: array of real values with same shape as z_err_array representing the error in the determinant of Z

Returns
error_sqrt: array of real values with same shape as z_err_array representing the error in the (determinant of Z)**0.5

mtpy.utils.calculator.**get_period_list**(*period_min*, *period_max*, *periods_per_decade*, *include_outside_range=True*)

get a list of values (e.g. periods), evenly spaced in log space and including values on multiples of 10

Returns
numpy array containing list of values

Inputs
period_min = minimum period period_max = maximum period periods_per_decade = number of periods per decade include_outside_range = option whether to start and finish the period

list just inside or just outside the bounds specified by period_min and period_max default True

mtpy.utils.calculator.**invertmatrix_incl_errors**(*inmatrix*, *inmatrix_error=None*)

mtpy.utils.calculator.**make_log_increasing_array**(*z1_layer*, *target_depth*, *n_layers*, *increment_factor=0.999*)

create depth array with log increasing cells, down to target depth, inputs are z1_layer thickness, target depth, number of layers (n_layers)

mtpy.utils.calculator.**multiplymatrices_incl_errors**(*inmatrix1*, *inmatrix2*, *inmatrix1_error=None*, *inmatrix2_error=None*)

mtpy.utils.calculator.**nearest_index**(*val*, *array*)

find the index of the nearest value in the array :param val: the value to search for :param array: the array to search in

Returns
index: integer describing position of nearest value in array

mtpy.utils.calculator.**propagate_error_polar2rect**(*r*, *r_error*, *phi*, *phi_error*)

Find error estimations for the transformation from polar to cartesian coordinates.

Uncertainties in polar representation define a section of an annulus. Find the 4 corners of this section and additionally the outer boundary point, which is defined by phi = phi0, rho = rho0 + sigma rho. The cartesian "box" defining the uncertainties in x,y is the outer bound around the annulus section, defined by the four outermost points. So check the four corners as well as the outer boundary edge of the section to find the extrema in x znd y. These give you the sigma_x/y.

mtpy.utils.calculator.**propagate_error_rect2polar**(*x*, *x_error*, *y*, *y_error*)

mtpy.utils.calculator.**reorient_data2D**(*x_values*, *y_values*, *x_sensor_angle=0*, *y_sensor_angle=90*)

Re-orient time series data of a sensor pair, which has not been in default (x=0, y=90) orientation.

Input: - x-values - Numpy array - y-values - Numpy array Note: same length for both! - If not, the shorter length is taken

Optional: - Angle of the x-sensor - measured in degrees, clockwise from North (0) - Angle of the y-sensor - measured in degrees, clockwise from North (0)

Output: - corrected x-values (North) - corrected y-values (East)

mtpy.utils.calculator.**rhophi2z**(*rho*, *phi*, *freq*)

Convert impedance-style information given in Rho/Phi format into complex valued Z.

Input: rho - 2x2 array (real) - in Ohm m phi - 2x2 array (real) - in degrees freq - scalar - frequency in Hz

Output: Z - 2x2 array (complex)

mtpy.utils.calculator.**rotate_matrix_with_errors**(*in_matrix*, *angle*, *error=None*)

Rotate a matrix including errors clockwise given an angle in degrees.

Parameters

- **in_matrix** – A n x 2 x 2 matrix to rotate
- **angle** (*float*) – Angle to rotate by assuming clockwise positive from 0 = north
- **error** (*np.ndarray, optional*) – A n x 2 x 2 matrix of associated errors, defaults to None

Raises

MTex – If input array is incorrect

Returns

rotated matrix

Return type

np.ndarray

Returns

rotated matrix errors

Return type

np.ndarray

mtpy.utils.calculator.**rotate_vector_with_errors**(*in_vector*, *angle*, *error=None*)

Rotate a vector including errors clockwise given an angle in degrees.

Parameters

- **in_matrix** – A n x 1 x 2 vector to rotate
- **angle** (*float*) – Angle to rotate by assuming clockwise positive from 0 = north
- **error** (*np.ndarray, optional*) – A n x 1 x 2 vector of associated errors, defaults to None

Raises

MTex – If input array is incorrect

Returns

rotated vector

Return type

np.ndarray

Returns

rotated vector errors

Return type

np.ndarray

mtpy.utils.calculator.**roundsf**(*number*, *sf*)

round a number to a specified number of significant figures (sf)

mtpy.utils.calculator.**z_error2r_phi_error**(*z_real*, *z_imag*, *error*)

Error estimation from rectangular to polar coordinates.

By standard error propagation, relative error in resistivity is 2*relative error in z amplitude.

Uncertainty in phase (in degrees) is computed by defining a circle around the z vector in the complex plane. The uncertainty is the absolute angle between the vector to (x,y) and the vector between the origin and the tangent to the circle.

Returns
tuple containing relative error in resistivity, absolute error in phase

Inputs
z_real = real component of z (real number or array) z_imag = imaginary component of z (real number or array) error = absolute error in z (real number or array)

mtpy.utils.concatenate_input module

Description:
This script collates data from raw data files in a folder, within a time-range provided by the user and outputs corresponding .EX, .EY, .EZ, .BX, .BY and .BZ files in an output folder.

References:

CreationDate: 2017/10/23 Developer: rakib.hassan@ga.gov.au

Revision History:
LastUpdate: 2017/10/23 RH

class mtpy.utils.concatenate_input.**Data**(*dataPath*, *startDateTime=''*, *endDateTime=''*)

Bases: `object`

ouput(*prefix*, *outputPath*)

Parameters

- **prefix** – output file prefix
- **outputPath** – output folder

mtpy.utils.configfile module

Helper functions for the handling of configuration files (survey.cfg and BIRRP.cfg style).

@UofA, 2013 (LK)

mtpy.utils.configfile.**read_configfile**(*filename*)

Read a general config file and return the content as dictionary.

Config files without sections or only DEFAULT section -> return dictionary

Config files with sections -> return nested dictionary (main level keys are section heads)

Config files with sections as well as section-less entries -> return nested dictionary, which includes a top level 'DEFAULT' key

mtpy.utils.configfile.**read_survey_configfile**(*filename*)

Read in a survey configuration file and return a dictionary.

Input config file must contain station names as section headers!

The output dictionary keys are station names (capitalised), the values are (sub-)dictionaries. The configuration file must contain sections for all stations, each containing all mandatory keywords:

- latitude (deg)
- longitude (deg)
- elevation (in meters)
- sampling_interval (in seconds)

- station_type (MT, (Q)E, (Q)B)

Not mandatory, but recommended - declination (in degrees, positive to East) - this is set to '0.0', if omitted

Depending on the type of station the following entries are required.

E-field recorded:

- E_logger_type ('edl'/'elogger'/'qel')
- E_logger_gain (factor/gain-level)
- E_instrument_type ('electrodes'/'dipole')
- E_instrument_amplification (applied amplification factor)
- E_Xaxis_azimuth (degrees)
- E_Xaxis_length (in meters)
- E_Yaxis_azimuth (degrees)
- E_Yaxis_length (in meters)

B-field recorded:

- B_logger_type ('edl'/'qel_blogger')
- B_logger_gain (factor/gain level)
- B_instrument_type ('coil(s)', 'fluxgate')
- B_instrument_amplification (applied amplification factor)
- B_Xaxis_azimuth (degrees)
- B_Yaxis_azimuth (degrees)

A global section can be used to include parameters for all stations. The name of the section must be one of:

global/main/default/general

mtpy.utils.configfile.**read_survey_txt_file**(*survey_file*, *delimiter=None*)

read survey file and return a dictionary of dictionaries where the first nested dictionary is keyed by the station name. Each station dictionarly includes all the information input in the survey file with keywords verbatim as the headers in survey file, all lower case.

Must be included in survey file ================== == key word description ================= == station station name lat(itude) latitude (decimal degrees is best) long(itude) longitude (decimal degrees is best) elev(ation) elevation (in meters) ex/E_Xaxis_length dipole length in north direction (in meters) ey/E_Yaxis_length dipole length in east direction (in meters) E_Xaxis_azimuth orientaion of Ex (degrees) E_Yaxis_azimuth orientaion of Ey (degrees)

sampling_interval sampling interval in seconds hx coil number in north direction for calibration hy coil number in east direction for calibration hz coil number in vertical direction for calibration date date of deployment notes any notes that might help later station_type type of data collected (MT, E, B) declination declination in degrees (N = 0 and East = 90) ================= ==

Information on E-field data: ========================== === key word description ========================== === E_logger_type type of data logger used to record data E_logger_gain factor/gain level E_instrument_type type of electrodes used E_instrument_amplification applied amplification factor E_Xaxis_azimuth orientaion of Ex (degrees) E_Xaxis_length length of dipole for Ex (in meters) E_Yaxis_azimuth orientaion of Ey (degrees) E_Yaxis_length length of dipole for Ey (in meters) ========================== ==

***Information on B-field data*:**

survey_file : string (full path to file)

survey_lst
[list] list of dictionaries with key words the same as the headers in survey file, all lower case

`mtpy.utils.configfile.`**`write_config_from_survey_txt_file`**(*survey_file*, *save_name=None*, *delimiter='\t'*)

write a survey configuration file from a survey txt file

Arguments:

survey_file
[string] full path to survey text file. See read_survey_txt_file for the assumed header information.

save_name
[string] name to save file to. If save_name = None, then file saved as os.path.join(os.path.dirname(survey_file,

os.path.basename(survey_file).cfg)

Outputs:

cfg_fn
[string] full path to saved config file

`mtpy.utils.configfile.`**`write_dict_to_configfile`**(*dictionary*, *output_filename*)

Write a dictionary into a configuration file.

The dictionary can contain pure key-value pairs as well as a level-1 nested dictionary. In the first case, the entries are stored in a 'DEFAULT' section. In the latter case, the dictionary keys are taken as section heads and the sub-dictionaries key-value pairs fill up the respective section

mtpy.utils.convert_modem_data_to_geogrid module

mtpy.utils.edi_folders module

Description:
Find path to all the directories which contain a given type of files: .edi, .py .jpg, .pdf

How to Run:
python mtpy/utils/edi_folders.py . EDI python mtpy/utils/edi_folders.py /e/Data/ EDI 2 python mtpy/utils/edi_folders.py /e/Data/ PY

CreationDate: 26/11/2017 Developer: fei.zhang@ga.gov.au

Revision History:
LastUpdate: 26/11/2017 FZ started the first version

class mtpy.utils.edi_folders.**EdiFolders**(*startDir*, *edifiles_threshold=1*, *filetype='.edi'*)

Bases: object

find_edi_folders(*aStartDir*)

find directories containing the file of type self.filetype :param aStartDir: the directory to start from :return: a list of full path to folders of interest.

get_all_edi_files()

mtpy.utils.edi_folders.**recursive_glob**(*dirname*, *ext='*.edi'*)

Under the dirname recursively find all files with extension ext. Return a list of the full-path to the types of files of interest.

This function is useful to handle a nested directories of EDI files.

Parameters

- **dirname** – a single dir OR a list of dirs.
- **ext** – eg, ".edi", ".xml"

Returns

a list of path2files

mtpy.utils.exceptions module

Specific exceptions for MTpy.

@UofA, 2013 (LK)

exception mtpy.utils.exceptions.**MTTimeError**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_EDI**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_PT**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_Tipper**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_Z**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_config_file**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_edi_file**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_file_handling**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_float**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_input_arguments**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_module_import**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_occam**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_parameter_number**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_processing**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_ts_data**

Bases: Exception

exception mtpy.utils.exceptions.**MTpyError_value**

Bases: Exception

mtpy.utils.filehandling module

Helper functions for file handling.

The various functions deal with renaming, sorting, concatenation of time series, extraction of names and times from filenames, reading configuration files, ...

@UofA, 2013 (LK)

mtpy.utils.filehandling.**EDL_get_starttime_fromfilename**(*filename*)

Return starttime of data file in epoch seconds.

Starting time is determined by the filename. This has to be of the form 'somthing/*.stationname.ddmmyyHHMMSS.??'

mtpy.utils.filehandling.**EDL_get_stationname_fromfilename**(*filename*)

mtpy.utils.filehandling.**EDL_make_Nhour_files**(*n_hours*, *inputdir*, *sampling*, *stationname=None*, *outputdir=None*)

See 'EDL_make_dayfiles' for description and syntax.

Only difference: output files are blocks of (max) N hours, starting to count at midnight (00:00h) each day.

Conditions:

1. 24%%N = 0
2. input data files start on the hour marks

Not working yet!!

mtpy.utils.filehandling.**EDL_make_dayfiles**(*inputdir*, *sampling*, *stationname=None*, *outputdir=None*)

Concatenate ascii time series to dayfiles (calendar day, UTC reference).

Data can be within a single directory or a list of directories. However, the files in the directory(ies) 'inputdir' have to be for one station only, and named with a 2 character suffix, defining the channel!

If the time series are interrupted/discontinuous at some point, a new file will be started after that point, where the file index 'idx' is increased by 1. If no stationname is given, the leading non-datetime characters in the first filename are used.

Files are named as 'stationname_samplingrate_date_idx.channel' Stationname, channel, and sampling are written to a header line.

Output data consists of a single column float data array. The data are stored into one directory. If 'outputdir' is not specified, a subdirectory 'dayfiles' will be created witihn the current working directory.

Note: Midnight cannot be in the middle of a file, because only file starts are checked for a new day!!

`mtpy.utils.filehandling.get_filename(fn, save_path, fn_basename)`

Get file name from inputs

Parameters

- **fn** (*TYPE*) – DESCRIPTION
- **save_path** (*TYPE*) – DESCRIPTION
- **fn_basename** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`mtpy.utils.filehandling.get_pathlist(masterdir, search_stringlist=None, search_stringfile=None, start_dict={}, split='_', extension='', folder=False)`

get a list of files or folders by searching on a string contained in search_stringlist or alternatively search_stringfile

returns: dictionary containing search strings as keys and file/folder as values

masterdir - directory to search in search_stringlist = list containing string identifiers for files or folders,

e.g. k0101 will work for edifile k0101.edi or folder k0101.

search_stringfile = alternative to search_stringlist (need to provide one)
will get search_stringlist from a file, full path or make sure you are in the correct directory!

start_dict = starting dictionary to append to, default is an empty dict split = if no exact match is found, search string will be split using split

character, useful when matching up edi's to inversion directories that both contain additional characters

extension = file extension, e.g. '.edi'

`mtpy.utils.filehandling.get_sampling_interval_fromdatafile(filename, length=3600)`

Find sampling interval from data file.

Provide data file (purely numerical content) and total data length in seconds (default 3600). Data are read in by the Numpy 'loadtxt'-function, the lentgh of the data array yields the sampling interval.

Lines beginning with # are ignored.

`mtpy.utils.filehandling.get_ts_header_string(header_dictionary)`

Return a MTpy time series data file header string from a dictionary.

`mtpy.utils.filehandling.make_unique_filename(infn)`

`mtpy.utils.filehandling.make_unique_folder(wd, basename='run')`

make a folder that doesn't exist already.

`mtpy.utils.filehandling.read1columntext(textfile)`

read a list from a one column text file

`mtpy.utils.filehandling.read_2c2_file(filename)`

Read in BIRRP 2c2 coherence files and return 4 lists containing [period],[freq],[coh],[zcoh]. Note if any of the coherences are negative a value of 0 will be given to them.

`mtpy.utils.filehandling.read_data_header(fn_raw)`

Deprecated!!! USE

read_ts_header

INSTEAD

Read the header line of MTpy TS data files.

input

MTpy TS data file name

output

list of header elements: stationname, channel, sampling rate, starttime first sample, starttime last sample, unit, lat, lon, elevation

`mtpy.utils.filehandling.read_stationdatafile(textfile, read_duplicates=True)`

read a space delimited file containing station info of any sort - 3 columns: station x, y, … - to a dictionary - station:[x,y,…] textfile = full path to text file read_duplicates = True/False - if stations are listed more than once do you

want to read all information or just the first occurrence, default True

example: import mtpy.utils.filehandling as fh stationdict = fh.read_stationxyfile(textfile)

`mtpy.utils.filehandling.read_surface_ascii(ascii_fn)`

read in surface which is ascii format () unlike original function, returns numpy array of lon, lat, elev (no projections)

The ascii format is assumed to be: ncols 2743 nrows 2019 xllcorner 111.791666666667 (lon of lower left) yllcorner -45.341666666667 (lat of lower left) cellsize 0.016666666667 NODATA_value -9999 elevation data origin (0,0) is NW upper left. NW —————> E | | | | S

`mtpy.utils.filehandling.read_ts_file(mtdatafile)`

Read an MTpy TS data file and provide the content as tuple:

(station, channel,samplingrate,t_min,nsamples,unit,lat,lon,elev, data) If header information is incomplete, the tuple is filled up with 'None'

`mtpy.utils.filehandling.read_ts_header(tsfile)`

Read in the header line from MTpy timeseries data files.

Return header as dictionary. Return empty dict, if no header line was found.

`mtpy.utils.filehandling.reorient_files(lo_files, configfile, lo_stations=None, outdir=None)` mtpy.utils.filehandling.**sort_folder_list**(*wkdir*, *order_file*, *indices=[0, 9999]*, *delimiter=''*)

sort subfolders in wkdir according to order in order_file

wkdir = working directory containing subfolders order = full path to text file containing order.

needs to contain a string to search on that is the same length for each item in the list

indices = indices to search on; default take the whole string

returns a list of directories, in order.

mtpy.utils.filehandling.**validate_save_file**(*savepath=None*, *savefile=None*, *basename=None*, *prioritise_savefile=False*)

Return savepath, savefile and basename, ensuring they are internally consistent and populating missing fields from the others or using defaults.

Prioritises savepath and basename. I.e. if savepath, savefile and basename are all valid but inconsistent, savefile will be updated to reflect savepath and basename

Parameters

- **savepath** – directory to save to
- **savefile** – full file path to save to
- **basename** – base file name to save to

mtpy.utils.filehandling.**validate_ts_file**(*tsfile*)

Validate MTpy timeseries (TS) data file Return Boolean value True/False .

mtpy.utils.filehandling.**write_ts_file_from_tuple**(*outfile*, *ts_tuple*, *fmt='%.8e'*)

Write an MTpy TS data file, where the content is provided as tuple:

(station, channel,samplingrate,t_min,nsamples,unit,lat,lon,elev, data)

todo: needs tuple-validation

mtpy.utils.gis_tools module

GIS_TOOLS

This module contains tools to help project between coordinate systems. The module will first use GDAL if installed. If GDAL is not installed then pyproj is used. A test has been made for new versions of GDAL which swap the input lat and lon when using transferPoint, so the user should not have to worry about which version they have.

Main functions are:

- project_point_ll2utm
- project_point_utm2ll

These can take in a point or an array or list of points to project.

latitude and longitude can be input as:

- 'DD:mm:ss.ms'
- 'DD.decimal_degrees'
- float(DD.decimal_degrees)

Created on Fri Apr 14 14:47:48 2017 Revised: 5/2020 JP Revised: 10/2023 JP

@author: jrpeacock

exception mtpy.utils.gis_tools.**GISError**

Bases: Exception

mtpy.utils.gis_tools.**assert_elevation_value**(*elevation*)

make sure elevation is a floating point number

Parameters
elevation (*float or str*) – elevation as a float or string that can convert

mtpy.utils.gis_tools.**assert_lat_value**(*latitude*)

Make sure the latitude value is in decimal degrees, if not change it. And that the latitude is within -90 < lat > 90.

Parameters
latitude (*float or string*) – latitude in decimal degrees or other format

mtpy.utils.gis_tools.**assert_lon_value**(*longitude*)

Make sure the longitude value is in decimal degrees, if not change it. And that the latitude is within -180 < lat > 180.

Parameters
latitude (*float or string*) – longitude in decimal degrees or other format

mtpy.utils.gis_tools.**assert_minutes**(*minutes*)

mtpy.utils.gis_tools.**assert_seconds**(*seconds*)

mtpy.utils.gis_tools.**convert_position_float2str**(*position*)

Convert position float to a string in the format of DD:MM:SS.

Parameters
position (*float*) – decimal degrees of latitude or longitude

Returns
latitude or longitude in format of DD:MM:SS.ms

mtpy.utils.gis_tools.**convert_position_str2float**(*position_str*)

Convert a position string in the format of DD:MM:SS to decimal degrees

Parameters
position (*float*) – latitude or longitude om DD:MM:SS.ms

Returns
latitude or longitude as a float

mtpy.utils.gis_tools.**project_point**(*x*, *y*, *old_epsg*, *new_epsg*)

Transform point to new epsg

Parameters

- **x** (*TYPE*) – DESCRIPTION
- **y** (*TYPE*) – DESCRIPTION
- **old_epsg** (*TYPE*) – DESCRIPTION
- **new_epsg** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`mtpy.utils.gis_tools.`**`project_point_ll2utm`**(*lat*, *lon*, *datum='WGS84'*, *epsg=None*)

Project a point that is in latitude and longitude to the specified UTM coordinate system.

Parameters

- **latitude** (*[string | float]*) – latitude in ['DD:mm:ss.ms' | 'DD.decimal' | float]
- **longitude** (*[string | float]*) – longitude in ['DD:mm:ss.ms' | 'DD.decimal' | float]
- **datum** (*string*) – well known datum
- **epsg** (*[int | string]*) – EPSG number defining projection (see http://spatialreference.org/ref/ for moreinfo) Overrides utm_zone if both are provided

Returns

project point(s)

Return type

tuple if a single point, np.recarray if multiple points * tuple is (easting, northing,utm_zone) * recarray has attributes (easting, northing, utm_zone, elevation)

Single Point

```
>>> gis_tools.project_point_ll2utm('-34:17:57.99', '149.2010301')
```

(702562.6911014864, 6202448.5654573515, '55H')

Multiple Points

```
>>> lat = np.arange(20, 40, 5)
>>> lon = np.arange(-110, -90, 5)
>>> gis_tools.project_point_ll2utm(lat, lon, datum='NAD27')
```

rec.array([(-23546.69921068, 2219176.82320353, 0., '13R'),

(500000. , 2764789.91224626, 0., '13R'), (982556.42985037, 3329149.98905941, 0., '13R'), (1414124.6019547 , 3918877.48599922, 0., '13R')],

dtype=[('easting', '<f8'), ('northing', '<f8'),
('elev', '<f8'), ('utm_zone', '<U3')])

`mtpy.utils.gis_tools.`**`project_point_utm2ll`**(*easting*, *northing*, *utm_epsg*, *datum_epsg=4326*)

Project a point that is in UTM to the specified geographic coordinate system.

Parameters

- **easting** (*float*) – easting in meters
- **northing** (*float*) – northing in meters
- **datum** (*string*) – well known datum
- **utm_zone** (*[string | int]*) – utm_zone {0-9}{0-9}{C-X} or {+, -}{0-9}{0-9}
- **epsg** (*[int | string]*) – EPSG number defining projection (see http://spatialreference.org/ref/ for moreinfo) Overrides utm_zone if both are provided

Returns

project point(s)

Return type
tuple if a single point, np.recarray if multiple points * tuple is (easting, northing,utm_zone) * recarray has attributes (easting, northing, utm_zone, elevation)

Single Point

```
>>> gis_tools.project_point_utm2ll(670804.18810336,
```

... 4429474.30215206, ... datum='WGS84', ... utm_zone='11T', ... epsg=26711) (40.000087, -114.999128)

Multiple Points

```
>>> gis_tools.project_point_utm2ll([670804.18810336, 680200],
```

... [4429474.30215206, 4330200], ... datum='WGS84', utm_zone='11T', ... epsg=26711) rec.array([(40.000087, -114.999128), (39.104208, -114.916058)],

dtype=[('latitude', '<f8'), ('longitude', '<f8')])

`mtpy.utils.gis_tools.`**`validate_input_values`**(*values*, *location_type=None*)

make sure the input values for lat, lon, easting, northing will be an numpy array with a float data type

can input a string as a comma separated list

Parameters
values (*[float | string | list | numpy.ndarray]*) – values to project, can be given as: * float * string of a single value or a comma separate string '34.2, 34.5' * list of floats or string * numpy.ndarray

Returns
array of floats

Return type
numpy.ndarray(dtype=float)

mtpy.utils.matplotlib_utils module

`mtpy.utils.matplotlib_utils.`**`gen_hist_bins`**(*uniq_period_list*)

`mtpy.utils.matplotlib_utils.`**`get_next_fig_num`**()

mtpy.utils.mtpy_decorator module

class `mtpy.utils.mtpy_decorator.`**`deprecated`**(*reason*)

Bases: `object`

Description:
used to mark functions, methods and classes deprecated, and prints warning message when it called decorators based on https://stackoverflow.com/a/40301488

Usage:
todo: write usage

Author: YingzhiGou Date: 20/06/2017

mtpy.utils.plot_rms_iterations module

`mtpy.utils.plot_rms_iterations.`**`concatenate_log_files`**(*directory*)

Any file of the pattern '*.log' will be included. The files are sorted alphanumerically and this is the order they will be concatenated in. It is up to the user to ensure that the files are named correctly to achieve the desired order.

Parameters
directory (*str*) –

`mtpy.utils.plot_rms_iterations.`**`plot`**(*metric*, *values*, *x_start=0*, *x_end=None*, *x_interval=1*, *y_start=None*, *y_end=None*, *y_interval=None*, *fig_width=1900*, *fig_height=1200*, *dpi=100*, *minor_ticks=True*)

`mtpy.utils.plot_rms_iterations.`**`read`**(*logfile*)

Get a sequence of values from a ModEM logfile. Each type of value present in the logfile is collected and ordered by iteration.

Parameters
path (*str*) – Path to the logfile to be read.

Returns

dict of str, float: A dictionary containing lists of metric
values.

mtpy.utils.sensor_orientation_correction module

Created on Fri Oct 7 23:28:58 2022

@author: jpeacock

`mtpy.utils.sensor_orientation_correction.`**`correct4sensor_orientation`**(*Z_prime*, *Bx=0*, *By=90*, *Ex=0*, *Ey=90*, *Z_prime_error=None*)

Correct a Z-array for wrong orientation of the sensors.

Assume, E' is measured by sensors orientated with the angles
E'x: a E'y: b

Assume, B' is measured by sensors orientated with the angles
B'x: c B'y: d

With those data, one obtained the impedance tensor Z':
E' = Z' * B'

Now we define change-of-basis matrices T,U so that
E = T * E' B = U * B'

=> T contains the expression of the E'-basis in terms of E (the standard basis) and U contains the expression of the B'-basis in terms of B (the standard basis) The respective expressions for E'x-basis vector and E'y-basis vector are the columns of T. The respective expressions for B'x-basis vector and B'y-basis vector are the columns of U.

We obtain the impedance tensor in default coordinates as:

E' = Z' * B' => T^(-1) * E = Z' * U^(-1) * B
=> E = T * Z' * U^(-1) * B => Z = T * Z' * U^(-1)

Parameters

- **Z_prime** – impedance tensor to be adjusted
- **Bx** (*float (angle in degrees)*) – orientation of Bx relative to geographic north (0) *default* is 0
- **By** –
- **Ex** (*float (angle in degrees)*) – orientation of Ex relative to geographic north (0) *default* is 0
- **Ey** (*float (angle in degrees)*) – orientation of Ey relative to geographic north (0) *default* is 90
- **Z_prime_error** (*np.ndarray(Z_prime.shape)*) – impedance tensor error (std) *default* is None

Dtype Z_prime
np.ndarray(num_frequency, 2, 2, dtype='complex')

Returns
adjusted impedance tensor

Return type
np.ndarray(Z_prime.shape, dtype='complex')

Returns
impedance tensor standard deviation in default orientation

Return type
np.ndarray(Z_prime.shape, dtype='real')

mtpy.utils.shapefiles module

mtpy.utils.shapefiles_creator module

Module contents

Got rid of the GDAL check because have moved geographic operations to use pyproj. There are still some functions that use GDAL, but those are for raster and shapefile making. Therefore the check is not needed.

Created on Tue Sep 5 14:35:54 2023

@author: jpeacock

Submodules

mtpy.mtpy_globals module

Description:
keep all the mtpy global params constants in this module.

Author: fei.zhang@ga.gov.au

FZ Last Updated: 2017-12-04 JP (2021-01-18) updated to use Path and get relative path locations.

Module contents

MTpy

class mtpy.**MT**(*fn=None*, ***kwargs*)

Bases: TF, *MTLocation*

Basic MT container to hold all information necessary for a MT station including the following parameters.

Impedance and Tipper element nomenclature is E/H therefore the first letter represents the output channels and the second letter represents the input channels.

For exampe for an input of Hx and an output of Ey the impedance tensor element is Zyx.

property Tipper

mtpy.core.z.Tipper object to hold tipper information

property Z

mtpy.core.z.Z object to hold impedance tensor

add_model_error(*comp=[]*, *z_value=5*, *t_value=0.05*, *periods=None*)

Add error to a station's components for given period range

Parameters

- **station** (*string or list of strings*) – name of station(s) to add error to
- **comp** – list of components to add data to, valid components are

zxx, zxy, zyx, zyy, tx, ty :type comp: string or list of strings :param periods: the period range to add to, if None all periods, otherwise enter as a tuple as (minimum, maximum) period in seconds :type periods: tuple (minimum, maxmum) :return: data array with added errors :rtype: np.ndarray

```
>>> d = Data()
>>> d.read_data_file(r"example/data.dat")
>>> d.data = d.add_error("mt01", comp=["zxx", "zxy", "tx"], z_value=7, t_
↪value=.05)
```

add_white_noise(*value*, *inplace=True*)

Add white noise to the data, useful for synthetic tests.

Parameters

- **value** (*TYPE*) – DESCRIPTION
- **inplace** (*TYPE, optional*) – DESCRIPTION, defaults to True

Returns

DESCRIPTION

Return type

TYPE

clone_empty()

copy metadata but not the transfer function estimates

compute_model_t_errors(*error_value=0.02*, *error_type='absolute'*, *floor=False*)

Compute mode errors based on the error type

key	definition
percent	error_value * t
absolute	error_value

Parameters

- **error_value** (*TYPE, optional*) – DESCRIPTION, defaults to .02
- **error_type** (*TYPE, optional*) – DESCRIPTION, defaults to "absolute"
- **floor** (*TYPE, optional*) – DESCRIPTION, defaults to True

Returns
DESCRIPTION

Return type
TYPE

compute_model_z_errors(*error_value=5*, *error_type='geometric_mean'*, *floor=True*)

Compute mode errors based on the error type

key	definition
egbert	error_value * sqrt(Zxy * Zyx)
geometric_mean	error_value * sqrt(Zxy * Zyx)
arithmetic_mean	error_value * (Zxy + Zyx) / 2
mean_od	error_value * (Zxy + Zyx) / 2
off_diagonals	zxx_error == zxy_error, zyx_error == zyy_error
median	error_value * median(z)
eigen	error_value * mean(eigen(z))
percent	error_value * z
absolute	error_value

Parameters

- **error_value** (*TYPE, optional*) – DESCRIPTION, defaults to 5
- **error_type** (*TYPE, optional*) – DESCRIPTION, defaults to "geometric_mean"
- **floor** (*TYPE, optional*) – DESCRIPTION, defaults to True

Returns
DESCRIPTION

Return type
TYPE

copy()

property ex_metadata

EX metadata

property ey_metadata

EY metadata

find_flipped_phase()

identify if the off-diagonal components are flipped from traditional quadrants. xy should be in the 1st quadarant (0-90 deg) and yx should be in the 3rd quadrant (-180 to -90 deg)

Returns
a dictionary of components with a bool for flipped or not if flipped return value is True

Return type
dict

flip_phase(*zxx=False*, *zxy=False*, *zyx=False*, *zyy=False*, *tzx=False*, *tzy=False*, *inplace=False*)

Flip the phase of a station in case its plotting in the wrong quadrant

Parameters

- **station** (*string or list*) – name(s) of station to flip phase
- **station** – station name or list of station names
- **zxx** (*TYPE, optional*) – Z_xx, defaults to False
- **zxy** (*TYPE, optional*) – Z_xy, defaults to False
- **zyy** (*TYPE, optional*) – Z_yx, defaults to False
- **zyx** (*TYPE, optional*) – Z_yy, defaults to False
- **tx** (*TYPE, optional*) – T_zx, defaults to False
- **ty** (*TYPE, optional*) – T_zy, defaults to False

Returns
new_data

Return type
np.ndarray

Returns
new mt_dict with components removed

Return type
dictionary

from_dataframe(*mt_df*)

fill transfer function attributes from a dataframe for a single station

Parameters
df (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

property hx_metadata

HX metadata

property hy_metadata

HY metadata

property hz_metadata

HZ metadata

interpolate(*new_period*, *method='slinear'*, *bounds_error=True*, *f_type='period'*, *z_log_space=False*, ***kwargs*)

Interpolate the impedance tensor onto different frequencies

Parameters

- **new_period** (*np.ndarray*) – a 1-d array of frequencies to interpolate on to. Must be with in the bounds of the existing frequency range, anything outside and an error will occur.
- **method** (*string, optional*) – method to interpolate by, defaults to "cubic"
- **bounds_error** (*boolean, optional*) – check for if input frequencies are within the original frequencies, defaults to True
- **f_type** (*string, defaults to 'period'*) – frequency type can be ['frequency' | 'period']
- ****kwargs** – key word arguments for *interp*

Raises

`ValueError` – If input frequencies are out of bounds

Returns

New MT object with interpolated values.

Return type

`mtpy.core.MT`

Note: 'cubic' seems to work the best, the 'slinear' seems to do the same as 'linear' when using the *interp* in xarray.

Interpolate over frequency

```
>>> mt_obj = MT()
>>> new_frequency = np.logspace(-3, 3, 20)
>>> new_mt_obj = mt_obj.interpolate(new_frequency, f_type="frequency
↪")
```

plot_depth_of_penetration(***kwargs*)

Plot Depth of Penetration estimated from Niblett-Bostick estimation

Parameters

****kwargs** – DESCRIPTION

Returns

DESCRIPTION

Return type

TYPE

plot_mt_response(***kwargs*)

Returns a mtpy.imaging.plotresponse.PlotResponse object

Plot Response

```
>>> mt_obj = mt.MT(edi_file)
>>> pr = mt.plot_mt_response()
>>> # if you need more info on plot_mt_response
>>> help(pr)
```

plot_phase_tensor(***kwargs*)

Returns
DESCRIPTION

Return type
TYPE

property pt

mtpy.analysis.pt.PhaseTensor object to hold phase tensor

remove_component(*zxx=False*, *zxy=False*, *zyy=False*, *zyx=False*, *tzx=False*, *tzy=False*, *inplace=False*)

Remove a component for a given station(s)

Parameters

- **station** (*string or list*) – station name or list of station names
- **zxx** (*TYPE, optional*) – Z_xx, defaults to False
- **zxy** (*TYPE, optional*) – Z_xy, defaults to False
- **zyy** (*TYPE, optional*) – Z_yx, defaults to False
- **zyx** (*TYPE, optional*) – Z_yy, defaults to False
- **tx** (*TYPE, optional*) – T_zx, defaults to False
- **ty** (*TYPE, optional*) – T_zy, defaults to False

Returns
new data array with components removed

Return type
np.ndarray

Returns
new mt_dict with components removed

Return type
dictionary

```
>>> d = Data()
>>> d.read_data_file(r"example/data.dat")
>>> d.data, d.mt_dict = d.remove_component("mt01", zxx=True, tx=True)
```

remove_distortion(*n_frequencies=None*, *comp='det'*, *only_2d=False*, *inplace=False*)

remove distortion following Bibby et al. [2005].

Parameters
n_frequencies (*int*) – number of frequencies to look for distortion from the highest frequency

Returns
Distortion matrix

Return type
np.ndarray(2, 2, dtype=real)

Returns
Z with distortion removed

Return type
mtpy.core.z.Z

Remove distortion and write new .edi file

```
>>> import mtpy.core.mt as mt
>>> mt1 = mt.MT(fn=r"/home/mt/edi_files/mt01.edi")
>>> D, new_z = mt1.remove_distortion()
>>> mt1.write_mt_file(new_fn=r"/home/mt/edi_files/mt01_dr.edi",
↪       >>>                    new_Z=new_z)
```

remove_static_shift(*ss_x=1.0*, *ss_y=1.0*, *inplace=False*)

Remove static shift from the apparent resistivity

Assume the original observed tensor Z is built by a static shift S and an unperturbated "correct" Z0 :

- Z = S * Z0

therefore the correct Z will be :

- Z0 = S^(-1) * Z

Parameters

- **ss_x** (*float*) – correction factor for x component
- **ss_y** (*float*) – correction factor for y component

Returns
new Z object with static shift removed

Return type
mtpy.core.z.Z

Note: The factors are in resistivity scale, so the entries of the matrix "S" need to be given by their square-roots!

Remove Static Shift

```
>>> import mtpy.core.mt as mt
>>> mt_obj = mt.MT(r"/home/mt/mt01.edi")
>>> new_z_obj = mt.remove_static_shift(ss_x=.5, ss_y=1.2)
>>> mt_obj.write_mt_file(new_fn=r"/home/mt/mt01_ss.edi",
>>> ...                    new_Z_obj=new_z_obj)
```

rotate(*theta_r*, *inplace=True*)

Rotate the data in degrees assuming North is 0 measuring clockwise positive to East as 90.

Parameters

- **theta_r** (*TYPE*) – DESCRIPTION

- **inplace** (*TYPE, optional*) – DESCRIPTION, defaults to True

Returns
DESCRIPTION

Return type
TYPE

property rotation_angle
rotation angle in degrees from north

property rrhx_metadata
RRHX metadata

property rrhy_metadata
RRHY metadata

to_dataframe(*utm_crs=None*, *cols=None*)
Create a dataframe from the transfer function for use with plotting and modeling.

Parameters
utm_crs (string, int, pyproj.CRS) – the utm zone to project station to, could be a name, pyproj.CRS, EPSG number, or anything that pyproj.CRS can intake.

to_occam1d(*data_filename=None*, *mode='det'*)
write an Occam1DData data file

Parameters

- **data_filename** (*string or Path*) – path to write file, if None returns Occam1DData object.
- **mode** (*string, optional*) – ['te', 'tm', 'det', 'tez', 'tmz', 'detz'], defaults to "det"

Returns
Occam1DData object

Return type
mtpy.modeling.occam1d.Occam1DData

Example

```
>>> mt_object = MT()
>>> mt_object.read(r"/path/to/tranfer_function/file")
>>> mt_object.compute_model_z_error()
>>> occam_data = mt_object.to_occam1d(data_filename=r"/path/to/data_
↪file.dat")
```

to_simpeg_1d(*mode='det'*, ***kwargs*)
helper method to run a 1D inversion using Simpeg

default is smooth parameters

To run sharp inversion

```
>>> mt_object.to_simpeg_1d({"p_s": 2, "p_z": 0, "use_irls": True})
```

To run sharp inversion adn compact

```
>>> mt_object.to_simpeg_1d({"p_s": 0, "p_z": 0, "use_irls": True})
```

Parameters
**kwargs – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

class mtpy.**MTCollection**(*working_directory=None*)

Bases: object

Collection of transfer functions

The main working variable is *MTCollection.dataframe* which is a property that returns either the *master dataframe* that contains all the TF's in the MTH5 file, or the *working_dataframe* which is a dataframe that has been queried in some way. Therefore all the user has to do is set the working directory as a subset of the master_dataframe

Example

```
>>> mc = MTCollection()
>>> mc.open_collection(filename="path/to/example/mth5.h5")
>>> mc.working_dataframe = mc.master_dataframe.iloc[0:5]
```

add_tf(*transfer_function*, *new_survey=None*, *tf_id_extra=None*)

transfer_function could be a transfer function object, a file name, a list of either.

Parameters

- **transfer_function** (*list, tuple, array,* MTData, MT) – transfer function object
- **new_survey** (*str, optional*) – new survey name, defaults to None
- **tf_id_extra** (*string, optional*) – additional text onto existing 'tf_id', defaults to None

Returns
DESCRIPTION

Return type
TYPE

apply_bbox(*lon_min*, *lon_max*, *lat_min*, *lat_max*)

Return pandas.DataFrame of station within bounding box

Parameters

- **longitude_min** (*float*) – Minimum longitude
- **longitude_max** (*float*) – Maximum longitude
- **latitude_min** (*float*) – Minimum latitude
- **latitude_max** (*float*) – Maximum longitude

Returns
Only stations within the given bounding box

Return type
pandas.DataFrame

average_stations(*cell_size_m*, *bounding_box=None*, *count=1*, *n_periods=48*, *new_file=True*)

Average nearby stations to make it easier to invert

Parameters

- **cell_size_m** (*TYPE*) – DESCRIPTION
- **bounding_box** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **save_dir** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

check_for_duplicates(*locate='location'*, *sig_figs=6*)

Check for duplicate station locations in a MT DataFrame

Parameters
dataframe (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

close_collection()

close mth5

Returns
DESCRIPTION

Return type
TYPE

property dataframe

This property returns the working dataframe or master dataframe if the working dataframe is None.

Returns
DESCRIPTION

Return type
TYPE

from_mt_data(*mt_data*, *new_survey=None*, *tf_id_extra=None*)

Add data from a MTData object to an MTH5 collection.

Can use 'new_survey' to create a new survey to load to.

Can use 'tf_id_extra' to add a string onto the existing 'tf_id', useful if data have been edited or manipulated in some way. For example could set 'tf_id_extra' = 'rotated' for rotated data. This will help you organize the tf's for each station.

Parameters

- **mt_data** (*mtpy.core.mt_data.MTData*) – MTData object
- **new_survey** (*str, optional*) – new survey name, defaults to None

- **tf_id_extra** (*string, optional*) – additional text onto existing 'tf_id', defaults to None

Raises
IOError – If an MTH5 is not writable raises

get_tf(*tf_id*, *survey=None*)
Get transfer function

Parameters
tf_id (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

has_data()

static **make_file_list**(*mt_path*, *file_types=['edi']*)
Get a list of MT file from a given path

Parameters
mt_path – full path to where the MT transfer functions are stored

or a list of paths :type mt_path: string or `pathlib.Path` or list

Parameters
file_types (*list*) – List of file types to look for given their extension

Currently available file types are or will be:

- edi - EDI files
- zmm - EMTF output file
- j - BIRRP output file
- avg - Zonge output file

property **master_dataframe**
This is the full summary of all transfer functions in the MTH5 file. It is a property because if a user adds TF's then the master_df will be automatically updated. the tranformation is quick for now.

property **mth5_filename**

open_collection(*filename=None*, *basename=None*, *working_directory=None*, *mode='a'*)
Initialize an mth5

Parameters

- **basename** (*TYPE, optional*) – DESCRIPTION, defaults to "mt_collection"
- **working_directory** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

plot_mt_response(*tf_id*, *survey=None*, ***kwargs*)

Parameters

- **tf_id** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

if input as list, tuple, np.ndarray, pd.series assuming first column is tf_id, and if needed the second column should be the survey id for that tf.

plot_penetration_depth_1d(*tf_id*, *survey=None*, ***kwargs*)

Plot 1D penetration depth based on the Niblett-Bostick transformation

Note that data is rotated to estimated strike previous to estimation and strike angles are interpreted for data points that are 3D.

See also:

`mtpy.analysis.niblettbostick.calculate_depth_of_investigation`

Parameters

- **tf_object** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_penetration_depth_map(*mt_data=None*, ***kwargs*)

Plot Penetration depth in map view for a single period

See also:

mtpy.imaging.PlotPenetrationDepthMap

Parameters
mt_data (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_phase_tensor(*tf_id*, *survey=None*, ***kwargs*)

plot phase tensor elements

Parameters

- **tf_id** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_phase_tensor_map(*mt_data=None*, ***kwargs*)

Plot Phase tensor maps for transfer functions in the working_dataframe

See also:

mtpy.imaging.PlotPhaseTensorMaps

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_phase_tensor_pseudosection(*mt_data=None*, ***kwargs*)

Plot a pseudo section of phase tensor ellipses and induction vectors if specified

See also:

mtpy.imaging.PlotPhaseTensorPseudosection

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_residual_phase_tensor(*mt_data_01*, *mt_data_02*, *plot_type='map'*, ***kwargs*)

Parameters

- **mt_data_01** (*TYPE*) – DESCRIPTION
- **mt_data_02** (*TYPE*) – DESCRIPTION
- **plot_type** (*TYPE, optional*) – DESCRIPTION, defaults to "map"
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_resistivity_phase_maps(*mt_data=None*, ***kwargs*)

Plot apparent resistivity and/or impedance phase maps from the working dataframe

See also:

mtpy.imaging.PlotResPhaseMaps

Parameters
`**kwargs` – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_resistivity_phase_pseudosections(*mt_data=None*, ***kwargs*)
Plot resistivity and phase in a pseudosection along a profile line

Parameters

- **mt_data** (*TYPE, optional*) – DESCRIPTION, defaults to None
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_stations(*map_epsg=4326*, *bounding_box=None*, ***kwargs*)
plot stations

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_strike(*mt_data=None*, ***kwargs*)
Plot strike angle

See also:

mtpy.imaging.PlotStrike

to_geo_df(*bounding_box=None*, *epsg=4326*)
Make a geopandas dataframe for easier GIS manipulation

to_mt_data(*bounding_box=None*, ***kwargs*)
Get a list of transfer functions

Parameters

- **tf_ids** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **bounding_box** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

to_shp(*filename*, *bounding_box=None*, *epsg=4326*)

Parameters

- **filename** (*TYPE*) – DESCRIPTION
- **bounding_box** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **epsg** (*TYPE, optional*) – DESCRIPTION, defaults to 4326

Returns
DESCRIPTION

Return type
TYPE

property working_directory

class mtpy.**MTData**(*mt_list=None*, ***kwargs*)

Bases: `OrderedDict`, *MTStations*

Collection of MT objects as an OrderedDict where keys are formatted as *survey_id.station_id*. Has all functionality of an OrderedDict for example can iterate of *.keys()*, *.values()* or *.items*. Values are a list of MT objects.

Inherits `mtpyt.core.MTStations` to deal with geographic locations of stations.

Is not optimized yet for speed, works fine for smaller surveys, but for large can be slow. Might try using a dataframe as the base.

add_station(*mt_object*, *survey=None*, *compute_relative_location=True*, *interpolate_periods=None*, *compute_model_error=False*)

Add a MT object

Parameters

- **mt_object** (*mtpy.MT*) – MT object for a single station
- **survey** (*str, optional*) – new survey name, defaults to None
- **compute_relative_location** (*bool, optional*) – Compute relative location, can be slow if adding single stations in a loop. If looping over station set to False and compute at the end, defaults to True
- **interpolate_periods** (*np.array, optional*) – periods to interpolate onto, defaults to None

add_tf(*tf*, ***kwargs*)

Add a MT object

Parameters

- **mt_object** (*mtpy.MT*) – MT object for a single station
- **survey** (*str, optional*) – new survey name, defaults to None
- **compute_relative_location** (*bool, optional*) – Compute relative location, can be slow if adding single stations in a loop. If looping over station set to False and compute at the end, defaults to True
- **interpolate_periods** (*np.array, optional*) – periods to interpolate onto, defaults to None

add_white_noise(*value*, *inplace=True*)

Add white noise to the data, useful for synthetic tests.

Parameters

- **value** (*TYPE*) – DESCRIPTION
- **inplace** (*TYPE, optional*) – DESCRIPTION, defaults to True

Returns
DESCRIPTION

Return type
TYPE

clone_empty()

Return a copy of MTData excluding MT objects.

Returns
Copy of MTData object excluding MT objects

Return type
mtpy.MTData

compute_model_errors(*z_error_value=None*, *z_error_type=None*, *z_floor=None*, *t_error_value=None*, *t_error_type=None*, *t_floor=None*)

Compute mode errors based on the error type

key	definition
egbert	error_value * sqrt(Zxy * Zyx)
geometric_mean	error_value * sqrt(Zxy * Zyx)
arithmetic_mean	error_value * (Zxy + Zyx) / 2
mean_od	error_value * (Zxy + Zyx) / 2
off_diagonals	zxx_err == zxy_err, zyx_err == zyy_err
median	error_value * median(z)
eigen	error_value * mean(eigen(z))
percent	error_value * z
absolute	error_value

Parameters

- **z_error_value** (*TYPE, optional*) – DESCRIPTION, defaults to 5
- **z_error_type** (*TYPE, optional*) – DESCRIPTION, defaults to "geometric_mean"
- **z_floor** (*TYPE, optional*) – DESCRIPTION, defaults to True
- **t_error_value** (*TYPE, optional*) – DESCRIPTION, defaults to 0.02
- **t_error_type** (*TYPE, optional*) – DESCRIPTION, defaults to "absolute"
- **t_floor** (*TYPE, optional*) – DESCRIPTION, defaults to True

:param : DESCRIPTION :type : TYPE :return: DESCRIPTION :rtype: TYPE

copy()

Deep copy of original MTData object

Parameters
memo (*TYPE*) – DESCRIPTION

Returns
Deep copy of original MTData

Return type
mtpy.MTData

estimate_spatial_static_shift(*station_key*, *radius*, *period_min*, *period_max*, *radius_units='m'*, *shift_tolerance=0.15*)

Estimate static shift for a station by estimating the median resistivity values for nearby stations within a radius given. Can set the period range to estimate the resistivity values.

Parameters

- **station_key** (*TYPE*) – DESCRIPTION
- **radius** (*TYPE*) – DESCRIPTION
- **period_min** (*TYPE*) – DESCRIPTION
- **period_max** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

estimate_starting_rho()

Estimate starting resistivity from the data. Creates a plot of the mean and median apparent resistivity values.

Returns
array of the median rho per period

Return type
np.ndarray(n_periods)

Returns
array of the mean rho per period

Return type
np.ndarray(n_periods)

```
>>> d = Data()
>>> d.read_data_file(r"example/data.dat")
>>> rho_median, rho_mean = d.estimate_starting_rho()
```

from_dataframe(*df*)

Create an dictionary of MT objects from a dataframe

Parameters
df (*pandas.DataFrame*) – dataframe of mt data

Returns
DESCRIPTION

Return type
TYPE

from_modem(*data_filename*, *survey='data'*, ***kwargs*)

read in a modem data file

Parameters

- **data_filename** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

from_modem_data(*data_filename*, *survey='data'*, ***kwargs*)

Parameters

- **data_filename** (*TYPE*) – DESCRIPTION
- **file_type** (*TYPE, optional*) – DESCRIPTION, defaults to "data"
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

from_occam2d(*data_filename*, *file_type='data'*, ***kwargs*)

read in occam data from a 2D data file *.dat

Read data file and plot

```
>>> from mtpy import MTData
>>> md = MTData()
>>> md.from_occam2d_data(f"/path/to/data/file.dat")
>>> plot_stations = md.plot_stations(model_locations=True)
```

Read response file

```
>>> md.from_occam2d_data(f"/path/to/response/file.dat")
```

Note: When reading in a response file the survey will be called model. So now you can have the data and model response in the same object.

from_occam2d_data(*data_filename*, *file_type='data'*, ***kwargs*)

get_nearby_stations(*station_key*, *radius*, *radius_units='m'*)

get stations close to a given station

Parameters

- **station_key** (*TYPE*) – DESCRIPTION
- **radius** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`get_periods()`

get all unique periods

Returns
DESCRIPTION

Return type
TYPE

`get_profile`(*x1*, *y1*, *x2*, *y2*, *radius*)

Get stations along a profile line given the (x1, y1) and (x2, y2) coordinates within a given radius (in meters).

These can be in (longitude, latitude) or (easting, northing). The calculation is done in UTM, therefore a UTM CRS must be input

Parameters

- **x1** (*TYPE*) – DESCRIPTION
- **y1** (*TYPE*) – DESCRIPTION
- **x2** (*TYPE*) – DESCRIPTION
- **y2** (*TYPE*) – DESCRIPTION
- **radius** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

`get_station`(*station_id=None*, *survey_id=None*, *station_key=None*)

if 'station_key' is None, tries to find key from *station_id* and 'survey_id' using MTData._get_station_key()

Parameters

- **station_key** (*str, optional*) – full station key {survey_id}.{station_id}, defaults to None
- **station_id** (*str, optional*) – station ID, defaults to None
- **survey_id** (*str, optional*) – survey ID, defaults to None

Raises
KeyError – If cannot find station_key

Returns
MT object

Return type
mtpy.MT

`get_subset`(*station_list*)

get a subset of the data from a list of stations, could be station_id or station_keys

Parameters
station_list (*list*) – list of station keys as {survey_id}.{station_id}

Returns
Returns just those stations within station_list

Return type
`mtpy.MTData`

get_survey(*survey_id*)
Get all MT objects that belong to the 'survey_id' from the data set.

Parameters
survey_id (*str*) – survey ID

Returns
MTData object including only those with the desired 'survey_id'

Return type
`mtpy.MTData`

interpolate(*new_periods*, *f_type='period'*, *inplace=True*)
Interpolate onto common period range

Parameters

- **new_periods** (*TYPE*) – DESCRIPTION
- **f_type** (*string, defaults to 'period'*) – frequency type can be ['frequency' | 'period']

Returns
DESCRIPTION

Return type
TYPE

property mt_list
List of MT objects :rtype: list

Type
return

property n_stations
number of stations in MT data

plot_mt_response(*station_key=None*, *station_id=None*, *survey_id=None*, ***kwargs*)

Parameters

- **tf_id** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

if input as list, tuple, np.ndarray, pd.series assuming first column is tf_id, and if needed the second column should be the survey id for that tf.

plot_penetration_depth_1d(*station_key=None*, *station_id=None*, *survey_id=None*, ***kwargs*)
Plot 1D penetration depth based on the Niblett-Bostick transformation

Note that data is rotated to estimated strike previous to estimation and strike angles are interpreted for data points that are 3D.

See also:

`mtpy.analysis.niblettbostick.calculate_depth_of_investigation`

Parameters

- **tf_object** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_penetration_depth_map(***kwargs*)

Plot Penetration depth in map view for a single period

See also:

mtpy.imaging.PlotPenetrationDepthMap

Parameters
mt_data (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_phase_tensor(*station_key=None*, *station_id=None*, *survey_id=None*, ***kwargs*)

plot phase tensor elements

Parameters

- **tf_id** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_phase_tensor_map(***kwargs*)

Plot Phase tensor maps for transfer functions in the working_dataframe

See also:

mtpy.imaging.PlotPhaseTensorMaps

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_phase_tensor_pseudosection(*mt_data=None*, ***kwargs*)

Plot a pseudo section of phase tensor ellipses and induction vectors if specified

See also:

`mtpy.imaging.PlotPhaseTensorPseudosection`

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_residual_phase_tensor_maps(*survey_01*, *survey_02*, ***kwargs*)

Parameters

- **survey_01** (*TYPE*) – DESCRIPTION
- **survey_02** (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_resistivity_phase_maps(***kwargs*)

Plot apparent resistivity and/or impedance phase maps from the working dataframe

See also:

mtpy.imaging.PlotResPhaseMaps

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_resistivity_phase_pseudosections(***kwargs*)

Plot resistivity and phase in a pseudosection along a profile line

Parameters

- **mt_data** (*TYPE, optional*) – DESCRIPTION, defaults to None
- ****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_stations(*map_epsg=4326*, *bounding_box=None*, *model_locations=False*, ***kwargs*)

plot stations

Parameters
****kwargs** – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

plot_strike(***kwargs*)

Plot strike angle

See also:

mtpy.imaging.PlotStrike

remove_station(*station_id*, *survey_id=None*)

remove a station from the dictionary based on the key

Parameters

- **station_id** (*str*) – station ID
- **survey_id** (*str*) – survey ID

rotate(*rotation_angle*, *inplace=True*)

rotate the data by the given angle assuming positive clockwise with north = 0, east = 90.

Parameters
rotation_angle (*TYPE*) – DESCRIPTION

Returns
DESCRIPTION

Return type
TYPE

property survey_ids

Survey IDs for all MT objects

Returns
list of survey IDs

Return type
list

to_dataframe(*utm_crs=None*, *cols=None*)

Parameters

- **utm_crs** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **cols** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns
DESCRIPTION

Return type
TYPE

`to_geo_df`(*model_locations=False*)

Make a geopandas dataframe for easier GIS manipulation

`to_modem`(*data_filename=None*, ***kwargs*)

Create a modem data file

Parameters

- **data_filename** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns

DESCRIPTION

Return type

TYPE

`to_modem_data`(*data_filename=None*, ***kwargs*)

`to_occam2d`(*data_filename=None*, ***kwargs*)

write an Occam2D data file

Parameters

- **data_filename** (*TYPE*) – DESCRIPTION
- ****kwargs** – DESCRIPTION

Returns

DESCRIPTION

Return type

TYPE

`to_occam2d_data`(*data_filename=None*, ***kwargs*)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

t

INDEX

A

B

C

D

E

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z