MTPy Documentation

Release 1.01.01

Alisson Kirkby, Fei Zhang, Jared Peacock, Rakib Hassan, Jingming

May 13, 2019

Contents

Contents:

CHAPTER 1

Package Core

1.1 Module z

exception mtpy.core.z.**MT_Z_Error**

class mtpy.core.z.**ResPhase**(*z_array=None*, *z_err_array=None*, *freq=None*, ***kwargs*)
resistivity and phase container

Attributes

phase

phase_det

phase_det_err

phase_err

phase_err_xx

phase_err_xy

phase_err_yx

phase_err_yy

phase_xx

phase_xy

phase_yx

phase_yy

res_det

res_det_err

res_err_xx

res_err_xy

res_err_yx

res_err_yy

res_xx

res_xy

res_yx

res_yy

resistivity

resistivity_err

Methods

compute_resistivity_phase(self[, z_array, ...])	compute resistivity and phase from z and z_err
set_res_phase(self, res_array, phase_array, freq)	Set values for resistivity (res - in Ohm m) and phase (phase - in degrees), including error propagation.

compute_resistivity_phase (*self*, *z_array=None*, *z_err_array=None*, *freq=None*)
compute resistivity and phase from z and z_err

set_res_phase (*self*, *res_array*, *phase_array*, *freq*, *res_err_array=None*, *phase_err_array=None*)
Set values for resistivity (res - in Ohm m) and phase (phase - in degrees), including error propagation.

Parameters

- **res_array** (*np.ndarray (num_freq, 2, 2)*) – resistivity array in Ohm-m
- **phase_array** (*np.ndarray (num_freq, 2, 2)*) – phase array in degrees
- **freq** (*np.ndarray (num_freq)*) – frequency array in Hz
- **res_err_array** (*np.ndarray (num_freq, 2, 2)*) – resistivity error array in Ohm-m
- **phase_err_array** (*np.ndarray (num_freq, 2, 2)*) – phase error array in degrees

class mtpy.core.z.**Tipper** (*tipper_array=None*, *tipper_err_array=None*, *freq=None*)
Tipper class –> generates a Tipper-object.

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

Parameters

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

Attributes	Description
freq	array of frequencies corresponding to elements of z
rotation_angle	angle of which data is rotated by
tipper	tipper array
tipper_err	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

Attributes

amplitude

amplitude_err

angle_err

angle_imag

angle_real

freq

mag_err

mag_imag

mag_real

phase

phase_err

tipper

tipper_err

Methods

`compute_amp_phase`(self)	Sets attributes:
`compute_mag_direction`(self)	computes the magnitude and direction of the real and imaginary induction vectors.
`rotate`(self, alpha)	Rotate Tipper array.
`set_amp_phase`(self, r_array, phi_array)	Set values for amplitude(r) and argument (phi - in degrees).
`set_mag_direction`(self, mag_real, ang_real, ...)	computes the tipper from the magnitude and direction of the real and imaginary components.

compute_amp_phase (*self*)

Sets attributes:

- *amplitude*
- *phase*

- *amplitude_err*
- *phase_err*

values for resistivity are in in Ohm m and phase in degrees.

compute_mag_direction(*self*)
computes the magnitude and direction of the real and imaginary induction vectors.

rotate(*self*, *alpha*)
Rotate Tipper array.

Rotation angle must be given in degrees. All angles are referenced to geographic North=0, positive in clockwise direction. (Mathematically negative!)

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

Updates the attributes:

- *tipper*
- *tipper_err*
- *rotation_angle*

set_amp_phase(*self*, *r_array*, *phi_array*)
Set values for amplitude(r) and argument (phi - in degrees).

Updates the attributes:

- tipper
- tipper_err

set_mag_direction(*self*, *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

class mtpy.core.z.**Z**(*z_array=None*, *z_err_array=None*, *freq=None*)
Z class - generates an impedance tensor (Z) object.

Z is a complex array of the form (n_freq, 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_array** (*numpy.ndarray(n_freq, 2, 2)*) – array containing complex impedance values
- **z_err_array** (*numpy.ndarray(n_freq, 2, 2)*) – array containing error values (standard deviation) of impedance tensor elements
- **freq** (*np.ndarray(n_freq)*) – array of frequency values corresponding to impedance tensor elements.

Attributes	Description
freq	array of frequencies corresponding to elements of z
rotation_angle	angle of which data is rotated by
z	impedance tensor
z_err	estimated errors of impedance tensor
resistivity	apparent resisitivity estimated from z in Ohm-m
resistivity_err	apparent resisitivity error
phase	impedance phase (deg)
phase_err	error in impedance phase

Methods	Description
det	calculates determinant of z with errors
invariants	calculates the invariants of z
inverse	calculates the inverse of z
re-move_distortion	removes distortion given a distortion matrix
remove_ss	removes static shift by assumin Z = S * Z_0
norm	calculates the norm of Z
only1d	zeros diagonal components and computes the absolute valued mean of the off-diagonal components.
only2d	zeros diagonal components
res_phase	computes resistivity and phase
rotate	rotates z positive clockwise, angle assumes North is 0.
set_res_phase	recalculates z and z_err, needs attribute freq
skew	calculates the invariant skew (off diagonal trace)
trace	calculates the trace of z

Example

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

Attributes

`det` Return the determinant of Z

`det_err` Return the determinant of Z error

`freq` Frequencies for each impedance tensor element

`invariants` Return a dictionary of Z-invariants.

`inverse` Return the inverse of Z.

`norm` Return the 2-/Frobenius-norm of Z

`norm_err` Return the 2-/Frobenius-norm of Z error

`only_1d` Return Z in 1D form.

`only_2d` Return Z in 2D form.

phase

phase_det

phase_det_err

phase_err

phase_err_xx

phase_err_xy

phase_err_yx

phase_err_yy

phase_xx

phase_xy

phase_yx

phase_yy

res_det

res_det_err

res_err_xx

res_err_xy

res_err_yx

res_err_yy

res_xx

res_xy

res_yx

res_yy

resistivity

resistivity_err

`skew` Returns the skew of Z as defined by Z[0, 1] + Z[1, 0]

`skew_err` Returns the skew error of Z as defined by Z_err[0, 1] + Z_err[1, 0]

`trace` Return the trace of Z

`trace_err` Return the trace of Z

`z` Impedance tensor

z_err

Methods

`compute_resistivity_phase`(self[, z_array, ...])	compute resistivity and phase from z and z_err
`remove_distortion`(self, distortion_tensor[, ...])	Remove distortion D form an observed impedance tensor Z to obtain the uperturbed "correct" Z0: Z = D * Z0

Continued on next page

Table 3 – continued from previous page

remove_ss(self[, reduce_res_factor_x, . . .])	Remove the static shift by providing the respective correction factors for the resistivity in the x and y components.
rotate(self, alpha)	Rotate Z array by angle alpha.
set_res_phase(self, res_array, phase_array, freq)	Set values for resistivity (res - in Ohm m) and phase (phase - in degrees), including error propagation.

det
Return the determinant of Z

Returns det_Z

Return type np.ndarray(nfreq)

det_err
Return the determinant of Z error

Returns det_Z_err

Return type np.ndarray(nfreq)

freq
Frequencies for each impedance tensor element

Units are Hz.

invariants
Return a dictionary of Z-invariants.

inverse
Return the inverse of Z.

(no error propagtaion included yet)

norm
Return the 2-/Frobenius-norm of Z

Returns norm

Return type np.ndarray(nfreq)

norm_err
Return the 2-/Frobenius-norm of Z error

Returns norm_err

Return type np.ndarray(nfreq)

only_1d
Return Z in 1D form.

If Z 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 Z off-diagonal absolutes.

only_2d
Return Z in 2D form.

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

remove_distortion (*self*, *distortion_tensor*, *distortion_err_tensor=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_err_tensor** – default is None

Return type

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

returns impedance tensor with distorion removed

Return type

np.ndarray(num_freq, 2, 2, dtype='complex')

returns impedance tensor error after distortion is removed

Return type

np.ndarray(num_freq, 2, 2, dtype='complex')

Example

```
>>> import mtpy.core.z as mtz
>>> distortion = np.array([[1.2, .5],[.35, 2.1]])
>>> d, new_z, new_z_err = z_obj.remove_distortion(distortion)
```

remove_ss (*self*, *reduce_res_factor_x=1.0*, *reduce_res_factor_y=1.0*)

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

Returns static shift matrix,

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

Returns corrected Z

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!

rotate (*self*, *alpha*)

Rotate Z 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.

Updates the attributes

- *z*
- *z_err*
- *zrot*
- *resistivity*
- *phase*
- *resistivity_err*
- *phase_err*

skew

Returns the skew of Z as defined by Z[0, 1] + Z[1, 0]

Note: This is not the MT skew, but simply the linear algebra skew

Returns skew

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

skew_err

Returns the skew error of Z as defined by Z_err[0, 1] + Z_err[1, 0]

Note: This is not the MT skew, but simply the linear algebra skew

Returns skew_err

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

trace

Return the trace of Z

Returns Trace(z)

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

trace_err

Return the trace of Z

Returns Trace(z)

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

z

Impedance tensor

np.ndarray(nfreq, 2, 2)

`mtpy.core.z.`**`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_freq, 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')

1.2 Module TS

class `mtpy.core.ts.`**`MT_TS`**(***kwargs*)

MT time series object that will read/write data in different formats including hdf5, txt, miniseed.

The foundations are based on Pandas Python package.

The data are store in the variable ts, which is a pandas dataframe with the data in the column 'data'. This way the data can be indexed as a numpy array:

```
>>> MT_TS.ts['data'][0:256]
```

or

```
>>> MT_TS.ts.data[0:256]
```

Also, the data can be indexed by time (note needs to be exact time):

```
>>> MT_TS.ts['2017-05-04 12:32:00.0078125':'2017-05-05 12:35:00]
```

Input ts as a numpy.ndarray or Pandas DataFrame

Metadata	Description
azimuth	clockwise angle from coordinate system N (deg)
calibration_fn	file name for calibration data
component	component name ['ex' | 'ey' | 'hx' | 'hy' | 'hz']
coordinate_system	[geographic | geomagnetic]
datum	datum of geographic location ex. WGS84
declination	geomagnetic declination (deg)
dipole_length	length of dipole (m)
data_logger	data logger type
instrument_id	ID number of instrument for calibration
lat	latitude of station in decimal degrees
lon	longitude of station in decimal degrees
n_samples	number of samples in time series
sampling_rate	sampling rate in samples/second
start_time_epoch_sec	start time in epoch seconds
start_time_utc	start time in UTC
station	station name
units	units of time series

Note: Currently only supports hdf5 and text files

Method	Description
read_hdf5	read an hdf5 file
write_hdf5	write an hdf5 file
write_ascii_file	write an ascii file
read_ascii_file	read an ascii file

Example

```
>>> import mtpy.core.ts as ts
>>> import numpy as np
>>> mt_ts = ts.MT_TS()
>>> mt_ts.ts = np.random.randn(1024)
>>> mt_ts.station = 'test'
>>> mt_ts.lon = 30.00
>>> mt_ts.lat = -122.00
>>> mt_ts.component = 'HX'
>>> mt_ts.units = 'counts'
>>> mt_ts.write_hdf5(r"/home/test.h5")
```

Attributes

elev elevation in elevation units

lat Latitude in decimal degrees

lon Longitude in decimal degrees

n_samples number of samples

sampling_rate sampling rate in samples/second

start_time_epoch_sec start time in epoch seconds

start_time_utc start time in UTC given in time format

ts

Methods

apply_addaptive_notch_filter(self[, ...])	apply notch filter to the data that finds the peak around each frequency.
decimate(self[, dec_factor])	decimate the data by using scipy.signal.decimate
low_pass_filter(self[, low_pass_freq, ...])	low pass the data
plot_spectra(self[, spectra_type])	Plot spectra using the spectral type
read_ascii(self, fn_ascii)	Read an ascii format file with metadata
read_ascii_header(self, fn_ascii)	Read an ascii metadata
read_hdf5(self, fn_hdf5[, ...])	Read an hdf5 file with metadata using Pandas.
write_ascii_file(self, fn_ascii[, chunk_size])	Write an ascii format file with metadata
write_hdf5(self, fn_hdf5[, ...])	Write an hdf5 file with metadata using pandas to write the file.

apply_addaptive_notch_filter(*self*, *notches=None*, *notch_radius=0.5*, *freq_rad=0.5*, *rp=0.1*)
apply notch filter to the data that finds the peak around each frequency.

see mtpy.processing.filter.adaptive_notch_filter

Parameters **notch_dict** (*dictionary*) – dictionary of filter parameters. if an empty dictionary is input the filter looks for 60 Hz and harmonics to filter out.

decimate(*self*, *dec_factor=1*)
decimate the data by using scipy.signal.decimate

Parameters **dec_factor** (*int*) – decimation factor

- refills ts.data with decimated data and replaces sampling_rate

elev
elevation in elevation units

lat
Latitude in decimal degrees

lon
Longitude in decimal degrees

low_pass_filter(*self*, *low_pass_freq=15*, *cutoff_freq=55*)
low pass the data

Parameters

- **low_pass_freq** (*float*) – low pass corner in Hz

    - **cutoff_freq** (*float*) – cut off frequency in Hz

- filters ts.data

n_samples
number of samples

plot_spectra (*self, spectra_type='welch', **kwargs*)
Plot spectra using the spectral type

Note: Only spectral type supported is welch

Parameters **spectra_type** – ['welch']

Example

```
>>> ts_obj = mtts.MT_TS()
>>> ts_obj.read_hdf5(r"/home/MT/mt01.h5")
>>> ts_obj.plot_spectra()
```

read_ascii (*self, fn_ascii*)
Read an ascii format file with metadata

Parameters **fn_ascii** (*string*) – full path to ascii file

Example

```
>>> ts_obj.read_ascii(r"/home/ts/mt01.EX")
```

read_ascii_header (*self, fn_ascii*)
Read an ascii metadata

Parameters **fn_ascii** (*string*) – full path to ascii file

Example

```
>>> ts_obj.read_ascii_header(r"/home/ts/mt01.EX")
```

read_hdf5 (*self, fn_hdf5, compression_level=0, compression_lib='blosc'*)
Read an hdf5 file with metadata using Pandas.

Parameters

- **fn_hdf5** (*string*) – full path to hdf5 file, has .h5 extension
- **compression_level** (*int*) – compression level of file [0-9]
- **compression_lib** (*string*) – compression library *default* is blosc

Returns fn_hdf5

See also:

Pandas.HDf5Store

sampling_rate
sampling rate in samples/second

start_time_epoch_sec
start time in epoch seconds

start_time_utc
start time in UTC given in time format

write_ascii_file(*self*, *fn_ascii*, *chunk_size=4096*)
Write an ascii format file with metadata

Parameters

- **fn_ascii** (`string`) – full path to ascii file
- **chunk_size** (`int`) – read in file by chunks for efficiency

Example

```
>>> ts_obj.write_ascii_file(r"/home/ts/mt01.EX")
```

write_hdf5(*self*, *fn_hdf5*, *compression_level=0*, *compression_lib='blosc'*)
Write an hdf5 file with metadata using pandas to write the file.

Parameters

- **fn_hdf5** (`string`) – full path to hdf5 file, has .h5 extension
- **compression_level** (`int`) – compression level of file [0-9]
- **compression_lib** (`string`) – compression library *default* is blosc

Returns fn_hdf5

See also:

Pandas.HDf5Store

exception mtpy.core.ts.**MT_TS_Error**

class mtpy.core.ts.**Spectra**(***kwargs*)
compute spectra of time series

Methods

compute_spectra(self, data, spectra_type, . . .)	compute spectra according to input type
welch_method(self, data[, plot])	Compute the spectra using the Welch method, which is an average spectra of the data.

compute_spectra(*self*, *data*, *spectra_type*, ***kwargs*)
compute spectra according to input type

welch_method(*self*, *data*, *plot=True*, ***kwargs*)
Compute the spectra using the Welch method, which is an average spectra of the data. Computes short time window of length nperseg and averages them to reduce noise.

1.3 Module MT

class mtpy.core.mt.**Citation**(***kwargs*)
Information for a citation.

Holds the following information:

Attributes	Type	Explanation
author	string	Author names
title	string	Title of article, or publication
journal	string	Name of journal
doi	string	DOI number (doi:10.110/sf454)
year	int	year published

More attributes can be added by inputing a key word dictionary

```
>>> Citation(**{'volume':56, 'pages':'234--214'})
```

class mtpy.core.mt.**Copyright**(***kwargs*)

Information of copyright, mainly about how someone else can use these data. Be sure to read over the conditions_of_use.

Holds the following information:

Attributes	Type	Explanation
citation	Citation	citation of published work using these data
conditions_of_use	string	conditions of use of these data
release_status	string	release status [open | public | proprietary]

More attributes can be added by inputing a key word dictionary

```
>>> Copyright(**{'owner':'University of MT', 'contact':'Cagniard'})
```

class mtpy.core.mt.**DataQuality**(***kwargs*)

Information on data quality.

Holds the following information:

Attributes	Type	Explanation
comments	string	comments on data quality
good_from_period	float	minimum period data are good
good_to_period	float	maximum period data are good
rating	int	[1-5]; 1 = poor, 5 = excellent
warrning_comments	string	any comments on warnings in the data
warnings_flag	int	[0-#of warnings]

More attributes can be added by inputing a key word dictionary

>>>DataQuality(**{'time_series_comments':'Periodic Noise'})

class mtpy.core.mt.**FieldNotes**(***kwargs*)

Field note information.

Holds the following information:

Attributes	Type	Explanation
data_quality	DataQuality	notes on data quality
electrode	Instrument	type of electrode used
data_logger	Instrument	type of data logger
magnetometer	Instrument	type of magnetotmeter

More attributes can be added by inputing a key word dictionary

```
>>> FieldNotes(**{'electrode_ex':'Ag-AgCl 213', 'magnetometer_hx':'102'})
```

class mtpy.core.mt.**Instrument**(***kwargs*)

Information on an instrument that was used.

Holds the following information:

Attributes	Type	Explanation
id	string	serial number or id number of data logger
manufacturer	string	company whom makes the instrument
type	string	Broadband, long period, something else

More attributes can be added by inputing a key word dictionary

```
>>> Instrument(**{'ports':'5', 'gps':'time_stamped'})
```

class mtpy.core.mt.**Location**(***kwargs*)

location details

Attributes

easting

elevation

latitude

longitude

northing

Methods

project_location2ll(self)	project location coordinates into meters given the reference ellipsoid, for now that is constrained to WGS84
project_location2utm(self)	project location coordinates into meters given the reference ellipsoid, for now that is constrained to WGS84

project_location2ll(*self*)

project location coordinates into meters given the reference ellipsoid, for now that is constrained to WGS84

Returns East, North, Zone

project_location2utm(*self*)

project location coordinates into meters given the reference ellipsoid, for now that is constrained to WGS84

Returns East, North, Zone

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

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

- Site –> information on site details (lat, lon, name, etc)
- FieldNotes –> information on instruments, setup, etc.

- Copyright –> information on how the data can be used and citations
- Provenance –> where the data come from and how they are stored
- Processing –> how the data were processed.

The most used attributes are made available from MT, namely the following.

Attribute	Description
station	station name
lat	station latitude in decimal degrees
lon	station longitude in decimal degrees
elev	station elevation in meters
Z	mtpy.core.z.Z object for impedance tensor
Tipper	mtpy.core.z.Tipper object for tipper
pt	mtpy.analysis.pt.PhaseTensor for phase tensor
east	station location in UTM coordinates assuming WGS-84
north	station location in UTM coordinates assuming WGS-84
utm_zone	zone of UTM coordinates assuming WGS-84
rotation_angle	rotation angle of the data
fn	absolute path to the data file

Other information is contained with in the different class attributes. For instance survey name is in MT.Site.survey

Note:

- The best way to see what all the information is and where it is contained would be to write out a configuration file

```
>>> import mtpy.core.mt as mt
>>> mt_obj = mt.MT()
>>> mt_obj.write_cfg_file(r"/home/mt/generic.cfg")
```

- Currently EDI, XML, and j file are supported to read in information, and can write out EDI and XML formats. Will be extending to j and Egberts Z format.

Methods	Description
read_mt_file	read in a MT file [EDI | XML | j]
write_mt_file	write a MT file [EDI | XML]
read_cfg_file	read a configuration file
write_cfg_file	write a configuration file
remove_distortion	remove distortion following Bibby et al. [2005]
remove_static_shift	Shifts apparent resistivity curves up or down
interpolate	interpolates Z and T onto specified frequency array.

Examples

Read from an .edi File

```
>>> import mtpy.core.mt as mt
>>> mt_obj = mt.MT(r"/home/edi_files/s01.edi")
```

Remove Distortion

```
>>> 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

```
>>> new_z_obj = mt_obj.remove_static_shift(ss_x=.78, ss_y=1.1)
>>> # write a new edi file
>>> mt_obj.write_mt_file(new_fn=r"/home/edi_files/s01_ss.edi",
>>>                        new_Z=new_z)
>>> wrote file to: /home/edi_files/s01_ss.edi
```

Interpolate

```
>>> new_freq = np.logspace(-3, 3, num=24)
>>> new_z_obj, new_tipper_obj = mt_obj.interpolate(new_freq)
>>> mt_obj.write_mt_file(new_Z=new_z_obj, new_Tipper=new_tipper_obj)
>>> wrote file to: /home/edi_files/s01_RW.edi
```

Attributes

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

`Z` mtpy.core.z.Z object to hole impedance tensor

`east` easting (m)

`elev` Elevation

`fn` reference to original data file

`lat` Latitude

`lon` Longitude

`north` northing (m)

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

`rotation_angle` rotation angle in degrees from north

`station` station name

`utm_zone` utm zone

Methods

`interpolate`(self, new_freq_array[, . . .])	Interpolate the impedance tensor onto different frequencies
`plot_mt_response`(self, **kwargs)	Returns a mtpy.imaging.plotresponse.PlotResponse object
`read_cfg_file`(self, cfg_fn)	Read in a configuration file and populate attributes accordingly.
`read_mt_file`(self, fn[, file_type])	Read an MT response file.
`remove_distortion`(self[, num_freq])	remove distortion following Bibby et al.

Continued on next page

Table 7 – continued from previous page

remove_static_shift(self[, ss_x, ss_y])	Remove static shift from the apparent resistivity
write_cfg_file(self, cfg_fn)	Write a configuration file for the MT sections
write_mt_file(self[, save_dir, fn_basename, ...])	Write an mt file, the supported file types are EDI and XML.

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

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

east
easting (m)

elev
Elevation

fn
reference to original data file

interpolate (*self*, *new_freq_array*, *interp_type='slinear'*, *bounds_error=True*, *period_buffer=None*)
Interpolate the impedance tensor onto different frequencies

Parameters **new_freq_array** (*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.

Returns a new impedance object with the corresponding frequencies and components.

Return type *mtpy.core.z.Z*

Returns a new tipper object with the corresponding frequencies and components.

Return type *mtpy.core.z.Tipper*

Interpolate

```
>>> import mtpy.core.mt as mt
>>> edi_fn = r"/home/edi_files/mt_01.edi"
>>> mt_obj = mt.MT(edi_fn)
>>> # create a new frequency range to interpolate onto
>>> new_freq = np.logspace(-3, 3, 24)
>>> new_z_object, new_tipper_obj = mt_obj.interpolate(new_freq)
>>> mt_obj.write_mt_file(new_fn=r"/home/edi_files/mt_01_interp.edi",
>>> ...                     new_Z_obj=new_z_object,
>>> ...                     new_Tipper_obj=new_tipper_object)
```

lat
Latitude

lon
Longitude

north
northing (m)

plot_mt_response (*self*, ***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)
```

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

read_cfg_file (*self*, *cfg_fn*)
Read in a configuration file and populate attributes accordingly.

The configuration file should be in the form:

Site.Location.latitude = 46.5
Site.Location.longitude = 122.7
Site.Location.datum = 'WGS84'

Processing.Software.name = BIRRP
Processing.Software.version = 5.2.1

Provenance.Creator.name = L. Cagniard
Provenance.Submitter.name = I. Larionov

Parameters **cfg_fn** (*string*) – full path to configuration file

Note: The best way to make a configuration file would be to save a configuration file first from MT, then filling in the fields.

Make configuration file

```
>>> import mtpy.core.mt as mt
>>> mt_obj = mt.MT()
>>> mt_obj.write_cfg_file(r"/mt/generic_config.cfg")
```

Read in configuration file

```
>>> import mtpy.core.mt as mt
>>> mt_obj = mt.MT()
>>> mt_obj.read_cfg_file(r"/home/mt/survey_config.cfg")
```

read_mt_file (*self*, *fn*, *file_type=None*)
Read an MT response file.

Note: Currently only .edi, .xml, and .j files are supported

Parameters

- **fn** (*string*) – full path to input file
- **file_type** (*string*) – ['edi' | 'j' | 'xml' | . . .] if None, automatically detects file type by the extension.

Example

```
>>> import mtpy.core.mt as mt
>>> mt_obj = mt.MT()
>>> mt_obj.read_mt_file(r"/home/mt/mt01.xml")
```

remove_distortion (*self*, *num_freq=None*)

remove distortion following Bibby et al. [2005].

Parameters **num_freq** (*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 (*self*, *ss_x=1.0*, *ss_y=1.0*)

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)
```

rotation_angle
rotation angle in degrees from north

station
station name

utm_zone
utm zone

write_cfg_file(*self*, *cfg_fn*)
Write a configuration file for the MT sections

Parameters **cfg_fn** (*string*) – full path to configuration file to write to

Return cfg_fn full path to configuration file

Rtype cfg_fn string

Write configuration file

```
>>> import mtpy.core.mt as mt
>>> mt_obj = mt.MT()
>>> mt_obj.read_mt_file(r"/home/mt/edi_files/mt01.edi")
>>> mt_obj.write_cfg_file(r"/home/mt/survey_config.cfg")
```

write_mt_file(*self*, *save_dir=None*, *fn_basename=None*, *file_type='edi'*, *new_Z_obj=None*, *new_Tipper_obj=None*, *longitude_format='LON'*, *latlon_format='dms'*)
Write an mt file, the supported file types are EDI and XML.

Parameters

- **save_dir** (*string*) – full path save directory
- **fn_basename** (*string*) – name of file with or without extension
- **file_type** (*string*) – ['edi' | 'xml']
- **new_Z_obj** (mtpy.core.z.Z) – new Z object
- **new_Tipper_obj** (mtpy.core.z.Tipper) – new Tipper object
- **longitude_format** (*string*) – whether to write longitude as LON or LONG. options are 'LON' or 'LONG', default 'LON'
- **latlon_format** (*string*) – format of latitude and longitude in output edi, degrees minutes seconds ('dms') or decimal degrees ('dd')

Returns full path to file

Return type string

Example

```
>>> mt_obj.write_mt_file(file_type='xml')
```

exception mtpy.core.mt.**MT_Error**

class mtpy.core.mt.**Person**(***kwargs*)
Information for a person

Holds the following information:

Attributes	Type	Explanation
email	string	email of person
name	string	name of person
organization	string	name of person's organization
organization_url	string	organizations web address

More attributes can be added by inputing a key word dictionary

```
>>> Person(**{'phone':'650-888-6666'})
```

class mtpy.core.mt.**Processing**(***kwargs*)

Information for a processing

Holds the following information:

Attributes	Type	Explanation
email	string	email of person
name	string	name of person
organization	string	name of person's organization
organization_url	string	organizations web address

More attributes can be added by inputing a key word dictionary

```
>>> Person(**{'phone':'888-867-5309'})
```

class mtpy.core.mt.**Provenance**(***kwargs*)

Information of the file history, how it was made

Holds the following information:

Attributes	Type	Explanation
creation_time	string	creation time of file YYYY-MM-DD,hh:mm:ss
creating_application	string	name of program creating the file
creator	Person	person whom created the file
submitter	Person	person whom is submitting file for archiving

More attributes can be added by inputing a key word dictionary

```
>>> Provenance(**{'archive':'IRIS', 'reprocessed_by':'grad_student'})
```

class mtpy.core.mt.**Site**(***kwargs*)

Information on the site, including location, id, etc.

Holds the following information:

Attributes	Type	Explanation
aqcuired_by	string	name of company or person whom aqcuired the data.
id	string	station name
Location	object Location	Holds location information, lat, lon, elev datum, easting, northing see Location class
start_date	string	YYYY-MM-DD start date of measurement
end_date	string	YYYY-MM-DD end date of measurement
year_collected	string	year data collected
survey	string	survey name
project	string	project name
run_list	string	list of measurment runs ex. [mt01a, mt01b]

More attributes can be added by inputing a key word dictionary

```
>>> Site(**{'state':'Nevada', 'Operator':'MTExperts'})
```

Attributes

year_collected

class mtpy.core.mt.**Software**(***kwargs*)
software

1.4 Module EDI

class mtpy.core.edi.**DataSection**(*edi_fn=None*, *edi_lines=None*)
DataSection contains the small metadata block that describes which channel is which. A typical block looks like:

```
>=MTSECT

    ex=1004.001
    ey=1005.001
    hx=1001.001
    hy=1002.001
    hz=1003.001
    nfreq=14
    sectid=par28ew
    nchan=None
    maxblks=None
```

Parameters **edi_fn** (*string*) – full path to .edi file to read in.

Methods

get_data_sect(self)	read in the data of the file, will detect if reading spectra or impedance.
read_data_sect(self[, data_sect_list])	read data section
write_data_sect(self[, data_sect_list, . . .])	write a data section

get_data_sect (*self*)
read in the data of the file, will detect if reading spectra or impedance.

read_data_sect (*self*, *data_sect_list=None*)
read data section

write_data_sect (*self*, *data_sect_list=None*, *over_dict=None*)
write a data section

class mtpy.core.edi.**DefineMeasurement** (*edi_fn=None*, *edi_lines=None*)
DefineMeasurement class holds information about the measurement. This includes how each channel was setup. The main block contains information on the reference location for the station. This is a bit of an archaic part and was meant for a multiple station .edi file. This section is also important if you did any forward modeling with Winglink cause it only gives the station location in this section. The other parts are how each channel was collected. An example define measurement section looks like:

```
>=DEFINEMEAS

    MAXCHAN=7
    MAXRUN=999
    MAXMEAS=9999
    UNITS=M
    REFTYPE=CART
    REFLAT=-30:12:49.4693
    REFLONG=139:47:50.87
    REFELEV=0

>HMEAS ID=1001.001 CHTYPE=HX X=0.0 Y=0.0 Z=0.0 AZM=0.0
>HMEAS ID=1002.001 CHTYPE=HY X=0.0 Y=0.0 Z=0.0 AZM=90.0
>HMEAS ID=1003.001 CHTYPE=HZ X=0.0 Y=0.0 Z=0.0 AZM=0.0
>EMEAS ID=1004.001 CHTYPE=EX X=0.0 Y=0.0 Z=0.0 X2=0.0 Y2=0.0
>EMEAS ID=1005.001 CHTYPE=EY X=0.0 Y=0.0 Z=0.0 X2=0.0 Y2=0.0
>HMEAS ID=1006.001 CHTYPE=HX X=0.0 Y=0.0 Z=0.0 AZM=0.0
>HMEAS ID=1007.001 CHTYPE=HY X=0.0 Y=0.0 Z=0.0 AZM=90.0
```

Parameters **edi_fn** (*string*) – full path to .edi file to read in.

Methods

get_measurement_dict(self)	get a dictionary for the xmeas parts
get_measurement_lists(self)	get measurement list including measurement setup
read_define_measurement(self[, measurement_list])	read the define measurment section of the edi file
write_define_measurement(self[, …])	write the define measurement block as a list of strings

get_measurement_dict (*self*)
get a dictionary for the xmeas parts

get_measurement_lists (*self*)
get measurement list including measurement setup

read_define_measurement (*self*, *measurement_list=None*)
read the define measurment section of the edi file

should be a list with lines for:

- maxchan
- maxmeas
- maxrun
- refelev
- reflat
- reflon
- reftype
- units
- **dictionaries for >XMEAS with keys:**
  - id
  - chtype
  - x
  - y
  - axm

  -acqchn

write_define_measurement(*self*, *measurement_list=None*, *longitude_format='LON'*, *latlon_format='dd'*)
write the define measurement block as a list of strings

class mtpy.core.edi.**EMeasurement**(***kwargs*)
EMeasurement contains metadata for an electric field measurement

Attributes	Description
id	Channel number
chtype	[EX | EY]
x	x (m) north from reference point (station) of one electrode of the dipole
y	y (m) east from reference point (station) of one electrode of the dipole
x2	x (m) north from reference point (station) of the other electrode of the dipole
y2	y (m) north from reference point (station) of the other electrode of the dipole
acqchan	name of the channel acquired usually same as chtype

Fill Metadata

```
>>> import mtpy.core.edi as mtedi
>>> e_dict = {'id': '1', 'chtype':'ex', 'x':0, 'y':0, 'x2':50, 'y2
↪':50}
>>> e_dict['acqchn'] = 'ex'
>>> emeas = mtedi.EMeasurement(**e_dict)
```

class mtpy.core.edi.**Edi**(*edi_fn=None*)
This class is for .edi files, mainly reading and writing. Has been tested on Winglink and Phoenix output .edi's, which are meant to follow the archaic EDI format put forward by SEG. Can read impedance, Tipper and/or spectra data.

The Edi class contains a class for each major section of the .edi file.

Frequency and components are ordered from highest to lowest frequency.

Parameters **edi_fn** (*string*) – full path to .edi file to be read in. *default* is None. If an .edi file is input, it is automatically read in and attributes of Edi are filled

Methods	Description
read_edi_file	Reads in an edi file and populates the associated classes and attributes.
write_edi_file	Writes an .edi file following the EDI format given the apporpriate attributes are filled. Writes out in impedance and Tipper format.
_read_data	Reads in the impedance and Tipper blocks, if the .edi file is in 'spectra' format, read_data converts the data to impedance and Tipper.
_read_mt	Reads impedance and tipper data from the appropriate blocks of the .edi file.
_read_spectra	Reads in spectra data and converts it to impedance and Tipper data.

Attributes	Description	default
Data_sect	DataSection class, contains basic information on the data collected and in whether the data is in impedance or spectra.	
Define_measurement	DefineMeasurement class, contains information on how the data was collected.	
edi_fn	full path to edi file read in	None
Header	Header class, contains metadata on where, when, and who collected the data	
Info	Information class, contains information on how the data was processed and how the transfer functions where estimated.	
Tipper	mtpy.core.z.Tipper class, contains the tipper data	
Z	mtpy.core.z.Z class, contains the impedance data	
_block_len	number of data in one line.	6
_data_header_str	header string for each of the data section	'>!****{0}****!'
_num_format	string format of data.	' 15.6e'
_t_labels	labels for tipper blocks	
_z_labels	labels for impedance blocks	

Change Latitude

```
>>> import mtpy.core.edi as mtedi
>>> edi_obj = mtedi.Edi(edi_fn=r"/home/mt/mt01.edi")
>>> # change the latitude
>>> edi_obj.header.lat = 45.7869
>>> new_edi_fn = edi_obj.write_edi_file()
```

Attributes

elev Elevation in elevation units

lat latitude in decimal degrees

lon longitude in decimal degrees

station station name

Methods

read_edi_file(self[, edi_fn])	Read in an edi file and fill attributes of each section's classes.

Continued on next page

Table 10 – continued from previous page

write_edi_file(self[, new_edi_fn, . . .])	Write a new edi file from either an existing .edi file or from data input by the user into the attributes of Edi.

elev
Elevation in elevation units

lat
latitude in decimal degrees

lon
longitude in decimal degrees

read_edi_file (*self*, *edi_fn=None*)
Read in an edi file and fill attributes of each section's classes. Including:

- Header
- Info
- Define_measurement
- Data_sect
- Z
- Tipper

Note: Automatically detects if data is in spectra format. All data read in is converted to impedance and Tipper.

Parameters **edi_fn** (*string*) – full path to .edi file to be read in *default* is None

Example

```
>>> import mtpy.core.Edi as mtedi
>>> edi_obj = mtedi.Edi()
>>> edi_obj.read_edi_file(edi_fn=r"/home/mt/mt01.edi")
```

station
station name

write_edi_file (*self*, *new_edi_fn=None*, *longitude_format='LON'*, *latlon_format='dms'*)
Write a new edi file from either an existing .edi file or from data input by the user into the attributes of Edi.

Parameters

- **new_edi_fn** (*string*) – full path to new edi file. *default* is None, which will write to the same file as the input .edi with as: r"/home/mt/mt01_1.edi"
- **longitude_format** (*string*) – whether to write longitude as LON or LONG. options are 'LON' or 'LONG', default 'LON'
- **latlon_format** (*string*) – format of latitude and longitude in output edi, degrees minutes seconds ('dms') or decimal degrees ('dd')

Returns full path to new edi file

Return type string

Example

```
>>> import mtpy.core.edi as mtedi
>>> edi_obj = mtedi.Edi(edi_fn=r"/home/mt/mt01/edi")
>>> edi_obj.Header.dataid = 'mt01_rr'
>>> n_edi_fn = edi_obj.write_edi_file()
```

class mtpy.core.edi.**HMeasurement**(***kwargs*)

HMeasurement contains metadata for a magnetic field measurement

Attributes	Description
id	Channel number
chtype	[HX | HY | HZ | RHX | RHY]
x	x (m) north from reference point (station)
y	y (m) east from reference point (station)
azm	angle of sensor relative to north = 0
acqchan	name of the channel acquired usually same as chtype

Fill Metadata

```
>>> import mtpy.core.edi as mtedi
>>> h_dict = {'id': '1', 'chtype':'hx', 'x':0, 'y':0, 'azm':0}
>>> h_dict['acqchn'] = 'hx'
>>> hmeas = mtedi.HMeasurement(**h_dict)
```

class mtpy.core.edi.**Header**(*edi_fn=None*, ***kwargs*)

Header class contains all the information in the header section of the .edi file. A typical header block looks like:

```
>HEAD

    ACQBY=None
    ACQDATE=None
    DATAID=par28ew
    ELEV=0.000
    EMPTY=1e+32
    FILEBY=WG3DForward
    FILEDATE=2016/04/11 19:37:37 UTC
    LAT=-30:12:49
    LOC=None
    LON=139:47:50
    PROGDATE=2002-04-22
    PROGVERS=WINGLINK EDI 1.0.22
    COORDINATE SYSTEM = GEOGRAPHIC NORTH
    DECLINATION = 10.0
```

Parameters **edi_fn** (*string*) – full path to .edi file to be read in. *default* is None. If an .edi file is input attributes of Header are filled.

Many of the attributes are needed in the .edi file. They are marked with a yes for 'In .edi'

Methods	Description
get_header_list	get header lines from edi file
read_header	read in header information from header_lines
write_header	write header lines, returns a list of lines to write

Read Header

```
>>> import mtpy.core.edi as mtedi
>>> header_obj = mtedi.Header(edi_fn=r"/home/mt/mt01.edi")
```

Methods

get_header_list(self)	Get the header information from the .edi file in the form of a list, where each item is a line in the header section.
read_header(self[, header_list])	read a header information from either edi file or a list of lines containing header information.
write_header(self[, header_list, ...])	Write header information to a list of lines.

get_header_list (*self*)
Get the header information from the .edi file in the form of a list, where each item is a line in the header section.

read_header (*self, header_list=None*)
read a header information from either edi file or a list of lines containing header information.

Parameters header_list (*list*) – should be read from an .edi file or input as ['key_01=value_01', 'key_02=value_02']

Input header_list

```
>>> h_list = ['lat=36.7898', 'lon=120.73532', 'elev=120.0', ...
>>>           'dataid=mt01']
>>> import mtpy.core.edi as mtedi
>>> header = mtedi.Header()
>>> header.read_header(h_list)
```

write_header (*self, header_list=None, longitude_format='LON', latlon_format='dms'*)

Write header information to a list of lines.

param header_list should be read from an .edi file or input as ['key_01=value_01', 'key_02=value_02']

type header_list list

param longitude_format whether to write longitude as LON or LONG. options are 'LON' or 'LONG', default 'LON'

type longitude_format string

param latlon_format format of latitude and longitude in output edi, degrees minutes seconds ('dms') or decimal degrees ('dd')

type latlon_format string

returns header_lines list of lines containing header information will be of the form:

```
['>HEAD
```

', ' key_01=value_01

'] if None is input then reads from input .edi file or uses attribute information to write metadata.

class mtpy.core.edi.**Information**(*edi_fn=None*, *edi_lines=None*)

Contain, read, and write info section of .edi file

not much to really do here, but just keep it in the same format that it is read in as, except if it is in phoenix format then split the two paragraphs up so they are sequential.

Methods

get_info_list(self)	get a list of lines from the info section
read_info(self[, info_list])	read information section of the .edi file
write_info(self[, info_list])	write out information

get_info_list(*self*)

get a list of lines from the info section

read_info(*self*, *info_list=None*)

read information section of the .edi file

write_info(*self*, *info_list=None*)

write out information

1.5 Module EDI_Collection

Description: To compute and encapsulate the properties of a set of EDI files

Author: fei.zhang@ga.gov.au

CreateDate: 2017-04-20

class mtpy.core.edi_collection.**EdiCollection**(*edilist=None*, *mt_objs=None*, *outdir=None*, *ptol=0.05*)

A super class to encapsulate the properties pertinent to a set of EDI files

Parameters

- **edilist** – a list of edifiles with full path, for read-only
- **outdir** – computed result to be stored in outdir
- **ptol** – period tolerance considered as equal, default 0.05 means 5 percent

The ptol parameter controls what freqs/periods are grouped together: 10 percent may result more double counting of freq/period data than 5 pct. (eg: MT_Datasets/WPJ_EDI)

Methods

create_measurement_csv(self, dest_dir[, ...])	create csv file from the data of EDI files: IMPEDANCE, APPARENT RESISTIVITIES AND PHASES see also utils/shapefiles_creator.py
create_mt_station_gdf(self[, outshpfile])	create station location geopandas dataframe, and output to shape file
create_phase_tensor_csv(self, dest_dir[, ...])	create phase tensor ellipse and tipper properties.

Continued on next page

Table 13 – continued from previous page

create_phase_tensor_csv_with_image(*args, ...)	Using PlotPhaseTensorMaps class to generate csv file of phase tensor attributes, etc.
display_on_basemap(self)	display MT stations which are in stored in geopandas dataframe in a base map.
display_on_image(self)	display/overlay the MT properties on a background geo-referenced map image
export_edi_files(self, dest_dir[, ...])	export edi files.
get_bounding_box(self[, epsgcode])	compute bounding box
get_min_max_distance(self)	get the min and max distance between all possible pairs of stations.
get_period_occurance(self, aper)	For a given aperiod, compute its occurance frequencies among the stations/edi :param aper: a float value of the period :return:
get_periods_by_stats(self[, percentage])	check the presence of each period in all edi files, keep a list of periods which are at least percentage present :return: a list of periods which are present in at least percentage edi files
get_phase_tensor_tippers(self, period[, ...])	For a given MT period (s) value, compute the phase tensor and tippers etc.
get_station_utmzones_stats(self)	A simple method to find what UTM zones these (edi files) MT stations belong to are they in a single UTM zone, which corresponds to a unique EPSG code? or do they belong to multiple UTM zones?
get_stations_distances_stats(self)	get the min max statistics of the distances between stations.
plot_stations(self[, savefile, showfig])	Visualise the geopandas df of MT stations
select_periods(self[, show, period_list, ...])	Use edi_collection to analyse the whole set of EDI files
show_obj(self[, dest_dir])	test call object's methods and show it's properties

create_measurement_csv (*self, dest_dir, period_list=None, interpolate=True*)
create csv file from the data of EDI files: IMPEDANCE, APPARENT RESISTIVITIES AND PHASES see also utils/shapefiles_creator.py

Parameters

- **dest_dir** – output directory
- **period_list** – list of periods; default=None, in which data for all available frequencies are output
- **interpolate** – Boolean to indicate whether to interpolate data onto given period_list

Returns csvfname

create_mt_station_gdf (*self, outshpfile=None*)
create station location geopandas dataframe, and output to shape file

Parameters **outshpfile** – output file

Returns gdf

create_phase_tensor_csv (*self, dest_dir, period_list=None, interpolate=True, file_name='phase_tensor.csv'*)
create phase tensor ellipse and tipper properties. Implementation based on mtpy.utils.shapefiles_creator.ShapeFilesCreator.create_csv_files

Parameters

- **dest_dir** – output directory
- **period_list** – list of periods; default=None, in which data for all available frequencies are output
- **interpolate** – Boolean to indicate whether to interpolate data onto given period_list
- **file_name** – output file name

Returns pt_dict

create_phase_tensor_csv_with_image(**args*, ***kwargs*)
Using PlotPhaseTensorMaps class to generate csv file of phase tensor attributes, etc. Only for comparison. This method is more expensive because it will create plot object first.

Returns

display_on_basemap(*self*)
display MT stations which are in stored in geopandas dataframe in a base map.

Returns plot object

display_on_image(*self*)
display/overlay the MT properties on a background geo-referenced map image

Returns plot object

export_edi_files(*self*, *dest_dir*, *period_list=None*, *interpolate=True*, *period_buffer=None*, *longitude_format='LON'*)
export edi files. :param dest_dir: output directory :param period_list: list of periods; default=None, in which data for all available

frequencies are output

Parameters

- **interpolate** – Boolean to indicate whether to interpolate data onto given period_list; otherwise a period_list is obtained from get_periods_by_stats()
- **file_name** – output file name
- **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

Returns

get_bounding_box(*self*, *epsgcode=None*)
compute bounding box

Returns bounding box in given proj coord system

get_min_max_distance(*self*)
get the min and max distance between all possible pairs of stations.

Returns min_dist, max_dist

get_period_occurance(*self*, *aper*)
For a given aperiod, compute its occurance frequencies among the stations/edi :param aper: a float value of the period :return:

get_periods_by_stats(*self*, *percentage=10.0*)
check the presence of each period in all edi files, keep a list of periods which are at least percentage present :return: a list of periods which are present in at least percentage edi files

get_phase_tensor_tippers (*self*, *period*, *interpolate=True*)
For a given MT period (s) value, compute the phase tensor and tippers etc.

Parameters

- **period** – MT_period (s)
- **interpolate** – Boolean to indicate whether to interpolate on to the given period

Returns dictionary pt_dict_list

pt_dict keys ['station', 'freq', 'lon', 'lat', 'phi_min', 'phi_max', 'azimuth', 'skew', 'n_skew', 'elliptic',
'tip_mag_re', 'tip_mag_im', 'tip_ang_re', 'tip_ang_im']

get_station_utmzones_stats (*self*)
A simple method to find what UTM zones these (edi files) MT stations belong to are they in a single UTM zone, which corresponds to a unique EPSG code? or do they belong to multiple UTM zones?

Returns a_dict like {UTMZone:Number_of_MT_sites}

get_stations_distances_stats (*self*)
get the min max statistics of the distances between stations. useful for determining the ellipses tipper sizes etc

Returns dict={}

plot_stations (*self*, *savefile=None*, *showfig=True*)
Visualise the geopandas df of MT stations

Parameters

- **savefile** –
- **showfig** –

Returns

select_periods (*self*, *show=True*, *period_list=None*, *percentage=10.0*)
Use edi_collection to analyse the whole set of EDI files

Parameters

- **show** – True or false
- **period_list** –
- **percentage** –

Returns select_period_list

show_obj (*self*, *dest_dir=None*)
test call object's methods and show it's properties

Returns

mtpy.core.edi_collection.**is_num_in_seq** (*anum*, *aseq*, *atol=0.0001*)
check if anum is in a sequence by a small tolerance

Parameters

- **anum** – a number to be checked
- **aseq** – a sequence or a list of values
- **atol** – absolute tolerance

Returns True | False

1.6 Module XML

Note: This module is written to align with the tools written by Anna Kelbert <akelbert@usgs.gov>

class mtpy.core.mt_xml.**MT_XML**(***kwargs*)

Class to read and write MT information from XML format. This tries to follow the format put forward by Anna Kelbert for archiving MT response data.

A configuration file can be read in that might make it easier to write multiple files for the same survey.

See also:

mtpy.core.mt_xml.XML_Config

Attributes	Description
Z	object of type mtpy.core.z.Z
Tipper	object of type mtpy.core.z.Tipper

Note: All other attributes are of the same name and of type XML_element, where attributes are name, value and attr. Attr contains any tag information. This is left this way so that mtpy.core.mt.MT can read in the information. **Use mtpy.core.mt.MT for conversion between data formats.**

Methods	Description
read_cfg_file	Read a configuration file in the format of XML_Config
read_xml_file	Read an xml file
write_xml_file	Write an xml file

Example :: >>> import mtpy.core.mt_xml as mtxml >>> x = mtxml.read_xml_file(r"/home/mt_data/mt01.xml") >>> x.read_cfg_file(r"/home/mt_data/survey_xml.cfg") >>> x.write_xml_file(r"/home/mt_data/xml/mt01.xml")

Attributes

Tipper get Tipper information

Z get z information

Methods

read_cfg_file(self[, cfg_fn])	Read in a cfg file making all key = value pairs attribures of XML_Config.
read_xml_file(self, xml_fn)	read in an xml file and set attributes appropriately.
write_cfg_file(self[, cfg_fn])	Write out configuration file in the style of: parent.attribute = value
write_xml_file(self, xml_fn[, cfg_fn])	write xml from edi

Tipper

get Tipper information

Z

get z information

read_xml_file (*self*, *xml_fn*)
read in an xml file and set attributes appropriately.

write_xml_file (*self*, *xml_fn*, *cfg_fn=None*)
write xml from edi

exception mtpy.core.mt_xml.**MT_XML_Error**

class mtpy.core.mt_xml.**XML_Config** (***kwargs*)
Class to deal with configuration files for xml.

Includes all the important information for the station and how data was processed.

Key Information includes:

Name	Purpose
ProductID	Station name
ExternalUrl	External URL to link to data
Notes	Any important information on station, data collection.
TimeSeriesArchived	Information on Archiving time series including URL.
Image	A location to an image of the station or the MT response.

- **ProductID –> station name**
  - ExternalUrl –> external url to link to data
  - Notes –> any

Methods

read_cfg_file(self[, cfg_fn])	Read in a cfg file making all key = value pairs attribures of XML_Config.
write_cfg_file(self[, cfg_fn])	Write out configuration file in the style of: parent.attribute = value

read_cfg_file (*self*, *cfg_fn=None*)
Read in a cfg file making all key = value pairs attribures of XML_Config. Being sure all new attributes are XML_element objects.

The assumed structure of the xml.cfg file is similar to: ''# XML Configuration File MTpy

Attachement.Description = Original file use to produce XML Attachment.Filename = None

Copyright.Citation.Authors = None Copyright.Citation.DOI = None Copyright.Citation.Journal = None Copyright.Citation.Title = None Copyright.Citation.Volume = None Copyright.Citation.Year = None

PeriodRange(max=0)(min=0) = None''

where the heirarchy of information is separated by a . and if the information has attribures they are in the name with (key=value) syntax.

write_cfg_file (*self*, *cfg_fn=None*)
Write out configuration file in the style of: parent.attribute = value

class mtpy.core.mt_xml.**XML_element** (*name*, *attr*, *value*, ***kwargs*)

Basically an ET element. The key components are

- 'name' –> name of the element
- 'attr' –> attribute information of the element
- 'value' –> value of the element

Used the property function here to be sure that these 3 cannot be set through the common k.value = 10, just in case there are similar names in the xml file. This seemed to be the safest to avoid those cases.

Attributes

attr

name

value

1.7 Module JFile

class mtpy.core.jfile.**JFile**(*j_fn=None*)

be able to read and write a j-file

Methods

read_header(self[, j_lines])	Parsing the header lines of a j-file to extract processing information.
read_j_file(self[, j_fn])	read_j_file will read in a *.j file output by BIRRP (better than reading lots of *.<k>r<l>.rf files)
read_metadata(self[, j_lines, j_fn])	read in the metadata of the station, or information of station logistics like: lat, lon, elevation

read_header(*self, j_lines=None*)

Parsing the header lines of a j-file to extract processing information.

Input: - j-file as list of lines (output of readlines())

Output: - Dictionary with all parameters found

read_j_file(*self, j_fn=None*)

read_j_file will read in a *.j file output by BIRRP (better than reading lots of *.<k>r<l>.rf files)

Input: j-filename

Output: 4-tuple - periods : N-array - Z_array : 2-tuple - values and errors - tipper_array : 2-tuple - values and errors - processing_dict : parsed processing parameters from j-file header

read_metadata(*self, j_lines=None, j_fn=None*)

read in the metadata of the station, or information of station logistics like: lat, lon, elevation

Not really needed for a birrp output since all values are nan's

CHAPTER 2

Package Analysis

2.1 Module Distortion

mtpy/analysis/distortion.py

Contains functions for the determination of (galvanic) 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 constraining 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.

Algorithm Details: Finding the distortion of a Z array. Using the phase tensor so, Z arrays are transformed into PTs first), following Bibby et al. 2005.

First, try to find periods that indicate 1D. From them determine D incl. the g-factor by calculatiing a weighted mean. The g is assumed in order to cater for the missing 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 regional 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 get 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.g. 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.analysis.distortion.`**`find_1d_distortion`**(*z_object*, *include_non1d=False*)

find 1D distortion tensor from z object

ONly use the 1D part of the Z to determine D. Treat all frequencies as 1D, if "include_non1d = True".

`mtpy.analysis.distortion.`**`find_2d_distortion`**`(`*z_object*, *include_non2d=False*`)`

find 2D distortion tensor from z object

ONly use the 2D part of the Z to determine D. Treat all frequencies as 2D, if "include_non2d = True".

`mtpy.analysis.distortion.`**`find_distortion`**`(`*z_object*, *g='det'*, *num_freq=None*, *lo_dims=None*`)`

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]

g [['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, meaning all frequencies are used

lo_dims [list] list of dimensions for each frequency *default* is None, meaning calculated from data

Returns

distortion [np.ndarray(2, 2)]

distortion array all real values

distortion_err [np.ndarray(2, 2)] distortion error array

Examples

Estimate Distortion

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

`mtpy.analysis.distortion.`**`remove_distortion`**`(`*z_array=None*, *z_object=None*, *num_freq=None*, *g='det'*`)`

remove distortion from an impedance tensor using the method outlined by Bibby et al., [2005].

Parameters

z_array [np.ndarray((nf, 2, 2))]

numpy array of impedance tensor *default* is None

z_object [mtpy.core.z object] *default* is None

num_freq [int] number of frequecies to look for distortion *default* is None, meaning look over all frequencies

g [['det' | '01' | '10]] type of distortion to look for *default* is 'det'

Returns

distortion [np.ndarray (2, 2)]

distortion array

new_z_obj [mtpy.core.z] z object with distortion removed and error calculated

Examples

Remove Distortion

```
>>> import mtpy.analysis.distortion as distortion
>>> d, new_z = distortion.remove_distortion(z_object=z_obj)
```

2.2 Module Geometry

mtpy/mtpy/analysis/geometry.py

Contains classes and functions for handling geometry analysis of impedance tensors:

dimensionality, strike directions, alphas/skews/...

- 1d - 2d : excentricity of ellipses
- 2d - 3d : skew < threshold (to be given as argument)
- strike: frequency - depending angle (incl. 90degree ambiguity)

@UofA, 2013(LK)

Edited by JP, 2016

`mtpy.analysis.geometry.`**`dimensionality`**(*z_array=None, z_object=None, pt_array=None, pt_object=None, skew_threshold=5, eccentricity_threshold=0.1*)

Esitmate dimensionality of an impedance tensor, frequency by frequency.

Dimensionality is estimated from the phase tensor given the threshold criteria on the skew angle and eccentricity following Bibby et al., 2005 and Booker, 2014.

Returns

****dimensions**** [np.ndarray(nf, dtype=int)] an array of dimesions for each frequency the values are [1 | 2 | 3]

Examples

Estimate Dimesions

```
>>> import mtpy.analysis.geometry as geometry
>>> dim = geometry.dimensionality(z_object=z_obj,
>>>                               skew_threshold=3)
```

`mtpy.analysis.geometry.`**`eccentricity`**(*z_array=None, z_object=None, pt_array=None, pt_object=None*)

Estimate eccentricy of a given impedance or phase tensor object

Returns

****eccentricity**** [np.ndarray(nf)] **eccentricity_err** : np.ndarray(nf)

Examples

Estimate Dimesions

```
>>> import mtpy.analysis.geometry as geometry
>>> ec, ec_err= geometry.eccentricity(z_object=z_obj)
```

mtpy.analysis.geometry.**strike_angle**(*z_array=None*, *z_object=None*, *pt_array=None*, *pt_object=None*, *skew_threshold=5*, *eccentricity_threshold=0.1*)

Estimate strike angle from 2D parts of the impedance tensor given the skew and eccentricity thresholds

Returns

****strike**** [np.ndarray(nf)] an array of strike angles in degrees for each frequency assuming 0 is north, and e is 90. There is a 90 degree ambiguity in the angle.

Examples

Estimate Dimesions

```
>>> import mtpy.analysis.geometry as geometry
>>> strike = geometry.strike_angle(z_object=z_obj,
>>>                                skew_threshold=3)
```

2.3 Module Phase Tensor

Following Caldwell et al, 2004

Residual Phase Tensor following Heise et al., [2008]

@UofA, 2013 (LK)

Revised by Peacock, 2016

class mtpy.analysis.pt.**PhaseTensor**(*pt_array=None*, *pt_err_array=None*, *z_array=None*, *z_err_array=None*, *z_object=None*, *freq=None*, *pt_rot=0.0*)

PhaseTensor class - generates a Phase Tensor (PT) 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,...

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

PTxx: (0,0) - PTxy: (0,1) - PTyx: (1,0) - PTyy: (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).

Attributes	Description
freq	array of frequencies associated with elements of impedance tensor.
pt	phase tensor array
pt_err	phase tensor error
z	impedance tensor
z_err	impedance error
rotation_angle	rotation angle in degrees

Attributes

`alpha` Return the principal axis angle (strike) of PT in degrees (incl.

alpha_err

`azimuth` Returns the azimuth angle related to geoelectric strike in degrees

azimuth_err

`beta` Return the 3D-dimensionality angle Beta of PT in degrees (incl.

beta_err

`det` Return the determinant of PT (incl.

det_err

`ellipticity` Returns the ellipticity of the phase tensor, related to dimesionality

ellipticity_err

`freq` freq array

`invariants` Return a dictionary of PT-invariants.

only1d

only2d

`phimax` Return the angle Phi_max of PT (incl.

phimax_err

`phimin` Return the angle Phi_min of PT (incl.

phimin_err

`pt` Phase tensor array

`pt_err` Phase tensor error array, must be same shape as pt

`skew` Return the skew of PT (incl.

skew_err

`trace` Return the trace of PT (incl.

trace_err

Methods

`rotate`(self, alpha)	Rotate PT array.

Continued on next page

Table 1 – continued from previous page

set_z_object(self, z_object)	Read in Z object and convert information into PhaseTensor object attributes.

alpha

Return the principal axis angle (strike) of PT in degrees (incl. uncertainties).

Output: - Alpha - Numpy array - Error of Alpha - Numpy array

azimuth
Returns the azimuth angle related to geoelectric strike in degrees including uncertainties

beta
Return the 3D-dimensionality angle Beta of PT in degrees (incl. uncertainties).

Output: - Beta - Numpy array - Error of Beta - Numpy array

det
Return the determinant of PT (incl. uncertainties).

Output: - Det(PT) - Numpy array - Error of Det(PT) - Numpy array

ellipticity
Returns the ellipticity of the phase tensor, related to dimesionality

freq
freq array

invariants
Return a dictionary of PT-invariants.

Contains: trace, skew, det, phimax, phimin, beta

phimax
Return the angle Phi_max of PT (incl. uncertainties).

Phi_max is calculated according to Bibby et al. 2005: Phi_max = Pi2 + Pi1

Output: - Phi_max - Numpy array - Error of Phi_max - Numpy array

phimin
Return the angle Phi_min of PT (incl. uncertainties).

Phi_min is calculated according to Bibby et al. 2005: Phi_min = Pi2 - Pi1

Output: - Phi_min - Numpy array - Error of Phi_min - Numpy array

pt
Phase tensor array

pt_err
Phase tensor error array, must be same shape as pt

rotate (*self*, *alpha*)
Rotate PT array. Change the rotation angles attribute respectively.

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.

set_z_object (*self*, *z_object*)
Read in Z object and convert information into PhaseTensor object attributes.

skew
Return the skew of PT (incl. uncertainties).

Output: - Skew(PT) - Numpy array - Error of Skew(PT) - Numpy array

trace
Return the trace of PT (incl. uncertainties).

Output: - Trace(PT) - Numpy array - Error of Trace(PT) - Numpy array

class mtpy.analysis.pt.**ResidualPhaseTensor** (*pt_object1=None*, *pt_object2=None*, *residual-type='heise'*)
PhaseTensor class - generates a Phase Tensor (PT) object DeltaPhi

DeltaPhi = 1 - Phi1^-1*Phi2

Methods

compute_residual_pt(self, pt_o1, pt_o2)	Read in two instance of the MTpy PhaseTensor class.
read_pts(self, pt1, pt2[, pt1err, pt2err])	Read two PT arrays and calculate the ResPT array (incl.
set_rpt(self, rpt_array)	Set the attribute 'rpt' (ResidualPhaseTensor array).
set_rpt_err(self, rpt_err_array)	Set the attribute 'rpt_err' (ResidualPhaseTensor-error array).

compute_residual_pt (*self*, *pt_o1*, *pt_o2*)
Read in two instance of the MTpy PhaseTensor class.

Update attributes: rpt, rpt_err, _pt1, _pt2, _pt1err, _pt2err

read_pts (*self*, *pt1*, *pt2*, *pt1err=None*, *pt2err=None*)
Read two PT arrays and calculate the ResPT array (incl. uncertainties).

Input: - 2x PT array

Optional: - 2x pt_error array

set_rpt (*self*, *rpt_array*)
Set the attribute 'rpt' (ResidualPhaseTensor array).

Input: ResPT array

Test for shape, but no test for consistency!

set_rpt_err (*self*, *rpt_err_array*)
Set the attribute 'rpt_err' (ResidualPhaseTensor-error array).

Input: ResPT-error array

Test for shape, but no test for consistency!

mtpy.analysis.pt.**edi_file2pt** (*filename*)
Calculate Phase Tensor from Edi-file (incl. uncertainties)

Input: - Edi-file : full path to the Edi-file

Return: - PT object

mtpy.analysis.pt.**z2pt** (*z_array*, *z_err_array=None*)
Calculate Phase Tensor from Z array (incl. uncertainties)

Input: - Z : 2x2 complex valued Numpy array

Optional: - Z-error : 2x2 real valued Numpy array

Return: - PT : 2x2 real valued Numpy array - PT-error : 2x2 real valued Numpy array

`mtpy.analysis.pt.`**`z_object2pt`**(*z_object*)

Calculate Phase Tensor from Z object (incl. uncertainties)

Input: - Z-object : instance of the MTpy Z class

Return: - PT object

2.4 Module Static Shift

module for estimating static shift

Created on Mon Aug 19 10:06:21 2013

@author: jpeacock

`mtpy.analysis.staticshift.`**`estimate_static_spatial_median`**(*edi_fn*, *radius=1000.0*, *num_freq=20*, *freq_skip=4*, *shift_tol=0.15*)

Remove static shift from a station using a spatial median filter. This will look at all the edi files in the same directory as edi_fn and find those station within the given radius (meters). Then it will find the medain static shift for the x and y modes and remove it, given that it is larger than the shift tolerance away from 1. A new edi file will be written in a new folder called SS.

Returns

****shift_corrections**** [(float, float)] static shift corrections for x and y modes

`mtpy.analysis.staticshift.`**`remove_static_shift_spatial_filter`**(*edi_fn*, *radius=1000*, *num_freq=20*, *freq_skip=4*, *shift_tol=0.15*, *plot=False*)

Remove static shift from a station using a spatial median filter. This will look at all the edi files in the same directory as edi_fn and find those station within the given radius (meters). Then it will find the medain static shift for the x and y modes and remove it, given that it is larger than the shift tolerance away from 1. A new edi file will be written in a new folder called SS.

Returns

****new_edi_fn_ss**** [string]

new path to the edi file with static shift removed

shift_corrections [(float, float)] static shift corrections for x and y modes

plot_obj [mtplot.plot_multiple_mt_responses object] If plot is True a plot_obj is returned If plot is False None is returned

2.5 Module Z Invariants

Created on Wed May 08 09:40:42 2013

Interpreted from matlab code written by Stephan Thiel 2005

@author: jpeacock

class `mtpy.analysis.zinvariants.`**`Zinvariants`**(*z_object=None*, *z_array=None*, *z_err_array=None*, *freq=None*, *rot_z=0*)

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

Attributes

****inv1**** [real off diaganol part normalizing factor] **inv2** : imaginary off diaganol normalizing factor

inv3 : real anisotropy factor (range from [0,1])

inv4 : imaginary anisotropy factor (range from [0,1])

inv5 : suggests electric field twist

inv6 : suggests in phase small scale distortion

inv7 : suggests 3D structure

strike : strike angle (deg) assuming positive clockwise 0=N

strike_err : strike angle error (deg)

q : dependent variable suggesting dimensionality

Methods

compute_invariants(self)	Computes the invariants according to Weaver et al., [2000, 2003]
rotate(self, rot_z)	Rotates the impedance tensor by the angle rot_z clockwise positive assuming 0 is North
set_freq(self, freq)	set the freq array, needs to be the same length at z
set_z(self, z_array)	set the z array.
set_z_err(self, z_err_array)	set the z_err array.

`compute_invariants`(*self*)

Computes the invariants according to Weaver et al., [2000, 2003]

Mostly used to plot Mohr's circles

In a 1D case: rho = mu (inv1**2+inv2**2)/w & phi = arctan(inv2/inv1)

Sets the invariants as attributes: inv1 : real off diaganol part normalizing factor

inv2 : imaginary off diaganol normalizing factor

inv3 : real anisotropy factor (range from [0,1])

inv4 : imaginary anisotropy factor (range from [0,1])

inv5 : suggests electric field twist

inv6 : suggests in phase small scale distortion

inv7 : suggests 3D structure

strike : strike angle (deg) assuming positive clockwise 0=N

strike_err : strike angle error (deg)

q : dependent variable suggesting dimensionality

rotate (*self*, *rot_z*)
Rotates the impedance tensor by the angle rot_z clockwise positive assuming 0 is North

set_freq (*self*, *freq*)
set the freq array, needs to be the same length at z

set_z (*self*, *z_array*)
set the z array.

If the shape changes or the freq are changed need to input those as well.

set_z_err (*self*, *z_err_array*)
set the z_err array.

If the shape changes or the freq are changed need to input those as well.

CHAPTER 3

Package Modeling

3.1 Module ModEM

exception mtpy.modeling.modem.**ModEMError**

exception mtpy.modeling.modem.**DataError**
Raise for ModEM Data class specific exceptions

class mtpy.modeling.modem.**Stations**(***kwargs*)
station locations class

..note:: If the survey steps across multiple UTM zones, then a distance will be added to the stations to place them in the correct location. This distance is _utm_grid_size_north and _utm_grid_size_east. You should these parameters to place the locations in the proper spot as grid distances and overlaps change over the globe. **This is not implemented yet**

Attributes

center_point calculate the center point from the given station locations

east

elev

lat

lon

north

rel_east

rel_north

station

utm_zone

Methods

`calculate_rel_locations`(self[, shift_east, ...])	put station in a coordinate system relative to (shift_east, shift_north) (+) shift right or up (-) shift left or down
`check_utm_crossing`(self)	If the stations cross utm zones, then estimate distance by computing distance on a sphere.
`get_station_locations`(self, input_list)	get station locations from a list of edi files
`rotate_stations`(self, rotation_angle)	Rotate stations assuming N is 0

calculate_rel_locations (*self*, *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

center_point
calculate the center point from the given station locations

Returns

****center_location**** [np.ndarray] structured array of length 1 dtype includes (east, north, zone, lat, lon)

check_utm_crossing (*self*)
If the stations cross utm zones, then estimate distance by computing distance on a sphere.

get_station_locations (*self*, *input_list*)
get station locations from a list of edi files

Returns

- **fills station_locations array**

rotate_stations (*self*, *rotation_angle*)
Rotate stations assuming N is 0

Returns

- **refils rel_east and rel_north in station_locations. Does this** because you will still need the original locations for plotting later.

class mtpy.modeling.modem.**Data** (*edi_list=None*, ***kwargs*)
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

Attributes

rotation_angle Rotate data assuming N=0, E=90

station_locations location of stations

Methods

`center_stations`(self, model_fn[, data_fn])	Center station locations to the middle of cells, might be useful for topography.

Continued on next page

Table 2 – continued from previous page

change_data_elevation(self, model_fn[, ...])	At each station in the data file rewrite the elevation, so the station is on the surface, not floating in air.
compute_inv_error(self)	compute the error from the given parameters
compute_phase_tensor(self, datfile, outdir)	Compute the phase tensors from a ModEM dat file :param datfile: path2/file.dat :return: path2csv created by this method
convert_modem_to_ws(self[, data_fn, ...])	convert a ModEM data file to WS format.
convert_ws3dinv_data_file(self, ws_data_fn)	convert a ws3dinv data file into ModEM format
fill_data_array(self[, new_edi_dir, ...])	fill the data array from mt_dict
filter_periods(mt_obj, per_array)	Select the periods of the mt_obj that are in per_array.
get_header_string(error_type, error_value, ...)	reset the header sring for file
get_mt_dict(self)	get mt_dict from edi file list
get_parameters(self)	get important parameters for documentation
get_period_list(self)	make a period list to invert for
get_relative_station_locations(self)	get station locations from edi files
project_stations_on_topography(self, ...[, ...])	This method is used in add_topography().
read_data_file(self[, data_fn, center_utm])	Read ModEM data file
write_data_file(self[, save_path, ...])	write data file for ModEM will save file as save_path/fn_basename
write_vtk_station_file(self[, ...])	write a vtk file for station locations.

center_stations (*self*, *model_fn*, *data_fn=None*)
Center station locations to the middle of cells, might be useful for topography.

Returns

****new_data_fn**** [string] full path to new data file

change_data_elevation (*self*, *model_fn*, *data_fn=None*, *res_air=1000000000000.0*)
At each station in the data file rewrite the elevation, so the station is on the surface, not floating in air.

compute_inv_error (*self*)
compute the error from the given parameters

compute_phase_tensor (*self*, *datfile*, *outdir*)
Compute the phase tensors from a ModEM dat file :param datfile: path2/file.dat :return: path2csv created by this method

convert_modem_to_ws (*self, data_fn=None, ws_data_fn=None, error_map=[1, 1, 1, 1]*)
convert a ModEM data file to WS format.

convert_ws3dinv_data_file (*self*, *ws_data_fn*, *station_fn=None*, *save_path=None*, *fn_basename=None*)
convert a ws3dinv data file into ModEM format

fill_data_array (*self*, *new_edi_dir=None*, *use_original_freq=False*, *longitude_format='LON'*)
fill the data array from mt_dict

static filter_periods (*mt_obj*, *per_array*)
Select the periods of the mt_obj that are in per_array. used to do original freq inversion.

Parameters

- **mt_obj** –

- **per_array** –

Returns array of selected periods (subset) of the mt_obj

static get_header_string (*error_type*, *error_value*, *rotation_angle*)
reset the header sring for file

get_mt_dict (*self*)
get mt_dict from edi file list

get_parameters (*self*)
get important parameters for documentation

get_period_list (*self*)
make a period list to invert for

get_relative_station_locations (*self*)
get station locations from edi files

project_stations_on_topography (*self*, *model_object*, *air_resistivity=1000000000000.0*)
This method is used in add_topography(). It will Re-write the data file to change the elevation column. And update covariance mask according topo elevation model. :param model_object: :param air_resistivity: :return:

read_data_file (*self*, *data_fn=None*, *center_utm=None*)
Read ModEM data file

inputs: data_fn = full path to data file name center_utm = option to provide real world coordinates of the center of

the grid for putting the data and model back into utm/grid coordinates, format [east_0, north_0, z_0]

Fills attributes:

- data_array
- period_list
- mt_dict

rotation_angle
Rotate data assuming N=0, E=90

station_locations
location of stations

write_data_file (*self*, *save_path=None*, *fn_basename=None*, *rotation_angle=None*, *compute_error=True*, *fill=True*, *elevation=False*, *use_original_freq=False*, *longitude_format='LON'*)
write data file for ModEM will save file as save_path/fn_basename

write_vtk_station_file (*self*, *vtk_save_path=None*, *vtk_fn_basename='ModEM_stations'*)
write a vtk file for station locations. For now this in relative coordinates.

class mtpy.modeling.modem.**Model** (*stations_object=None*, *data_object=None*, ***kwargs*)
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.

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 MtPy-Log more information

Continued on next page

Table 3 – continued from previous page

Attributes	Description
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
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

Continued on next page

Table 3 – continued from previous page

Attributes	Description
z_target_depth	Depth of deepest target, *default* is 50,000

Attributes

nodes_east

nodes_north

nodes_z

Methods

add_layers_to_mesh(self[, n_add_layers, ...])	Function to add constant thickness layers to the top or bottom of mesh.
add_topography_to_model2(self[, ...])	if air_layers is non-zero, will add topo: read in topograph file, make a surface model.
assign_resistivity_from_surfacedata(self, ...)	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)
get_parameters(self)	get important model parameters to write to a file for documentation later.
interpolate_elevation2(self[, surfacefile, ...])	project a surface to the model grid and add resulting elevation data to a dictionary called surface_dict.
make_mesh(self)	create finite element mesh according to user-input parameters.
make_z_mesh_new(self)	new version of make_z_mesh.
plot_mesh(self[, east_limits, north_limits, ...])	Plot the mesh to show model grid
plot_mesh_xy(self)	# add mesh grid lines in xy plan north-east map :return:
plot_mesh_xz(self)	display the mesh in North-Depth aspect :return:
plot_topography(self)	display topography elevation data together with station locations on a cell-index N-E map :return:
read_gocad_sgrid_file(self, sgrid_header_file)	read a gocad sgrid file and put this info into a ModEM file.
read_model_file(self[, model_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.
read_ws_model_file(self, ws_model_fn)	reads in a WS3INV3D model file
write_gocad_sgrid_file(self[, fn, origin, ...])	write a model to gocad sgrid
write_model_file(self, **kwargs)	will write an initial file for ModEM.
write_vtk_file(self[, vtk_save_path, ...])	write a vtk file to view in Paraview or other
write_xyres(self[, location_type, origin, ...])	write files containing depth slice data (x, y, res for each depth)

print_mesh_params	
print_model_file_summary	

add_layers_to_mesh (*self*, *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_model2

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_to_model2 (*self*, *topographyfile=None*, *topographyarray=None*, *interp_method='nearest'*, *air_resistivity=1000000000000.0*, *topography_buffer=None*, *airlayer_type='log_up'*)

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

- **topographyfile** – file containing topography (arcgis ascii grid)
- **topographyarray** – alternative to topographyfile - 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_surfacedata (*self*, *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 surfacename = 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

get_parameters (*self*)

get important model parameters to write to a file for documentation later.

interpolate_elevation2 (*self*, *surfacefile=None*, *surface=None*, *surfacename=None*, *method='nearest'*)

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 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'

make_mesh (*self*)
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_new (*self*)
new version of make_z_mesh. make_z_mesh and M

plot_mesh (*self*, *east_limits=None*, *north_limits=None*, *z_limits=None*, ***kwargs*)
Plot the mesh to show model grid

plot_mesh_xy (*self*)
add mesh grid lines in xy plan north-east map :return:

plot_mesh_xz (*self*)
display the mesh in North-Depth aspect :return:

plot_topography (*self*)
display topography elevation data together with station locations on a cell-index N-E map :return:

read_gocad_sgrid_file (*self*, *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 (*self*, *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

read_ws_model_file (*self*, *ws_model_fn*)
reads in a WS3INV3D model file

write_gocad_sgrid_file (*self, 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 (*self, **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.

write_vtk_file (*self, vtk_save_path=None, vtk_fn_basename='ModEM_model_res'*)
write a vtk file to view in Paraview or other

write_xyres (*self, location_type='EN', origin=[0, 0], model_epsg=None, depth_index='all', savepath=None, outfile_basename='DepthSlice', log_res=False, model_utm_zone=None, 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)

savepath = 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

class mtpy.modeling.modem.**Residual** (***kwargs*)
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	

Methods

calculate_residual_from_data(self[, . . .])	created by ak on 26/09/2017
write_rms_to_file(self)	write rms station data to file

get_rms	
read_residual_file	

calculate_residual_from_data (*self*, *data_fn=None*, *resp_fn=None*, *save_fn_basename=None*)

created by ak on 26/09/2017

Parameters

- **data_fn** –
- **resp_fn** –

Returns

write_rms_to_file (*self*)

write rms station data to file

class mtpy.modeling.modem.**ControlInv** (***kwargs*)

read and write control file for how the inversion starts and how it is run

Methods

read_control_file(self[, control_fn])	read in a control file
write_control_file(self[, control_fn, . . .])	write control file

read_control_file (*self*, *control_fn=None*)

read in a control file

write_control_file (*self*, *control_fn=None*, *save_path=None*, *fn_basename=None*)

write control file

class mtpy.modeling.modem.**ControlFwd** (***kwargs*)

read and write control file for

This file controls how the inversion starts and how it is run

Methods

read_control_file(self[, control_fn])	read in a control file
write_control_file(self[, control_fn, . . .])	write control file

read_control_file (*self*, *control_fn=None*)

read in a control file

write_control_file (*self*, *control_fn=None*, *save_path=None*, *fn_basename=None*)

write control file

class mtpy.modeling.modem.**Covariance** (*grid_dimensions=None*, ***kwargs*)

read and write covariance files

Methods

read_cov_file(self, cov_fn)	read a covariance file
write_cov_vtk_file(self, cov_vtk_fn[, . . .])	write a vtk file of the covariance to match things up
write_covariance_file(self[, cov_fn, . . .])	write a covariance file

get_parameters	

read_cov_file (*self*, *cov_fn*)
read a covariance file

write_cov_vtk_file (*self*, *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 (*self*, *cov_fn=None*, *save_path=None*, *cov_fn_basename=None*, *model_fn=None*, *sea_water=0.3*, *air=1000000000000.0*)
write a covariance file

class mtpy.modeling.modem.**ModEMConfig** (***kwargs*)
read and write configuration files for how each inversion is run

Methods

add_dict(self[, fn, obj])	add dictionary based on file name or object
write_config_file(self[, save_dir, . . .])	write a config file based on provided information

add_dict (*self*, *fn=None*, *obj=None*)
add dictionary based on file name or object

write_config_file (*self*, *save_dir=None*, *config_fn_basename='ModEM_inv.cfg'*)
write a config file based on provided information

class mtpy.modeling.modem.**ModelManipulator** (*model_fn=None*, *data_fn=None*, ***kwargs*)
will plot a model from wsinv3d or init file so the user can manipulate the resistivity values relatively easily. At the moment only plotted in map view.

Example :: >>> import mtpy.modeling.ws3dinv as ws >>> initial_fn = r"/home/MT/ws3dinv/Inv1/WSInitialFile" >>> mm = ws.WSModelManipulator(initial_fn=initial_fn)

Buttons	Description
'='	increase depth to next vertical node (deeper)
'-'	decrease depth to next vertical node (shallower)
'q'	quit the plot, rewrites initial file when pressed
'a'	copies the above horizontal layer to the present layer
'b'	copies the below horizonal layer to present layer
'u'	undo previous change

Attributes	Description
ax1	matplotlib.axes instance for mesh plot of the model
ax2	matplotlib.axes instance of colorbar
cb	matplotlib.colorbar instance for colorbar

Continued on next page

Table 10 – continued from previous page

Attributes	Description
cid_depth	matplotlib.canvas.connect for depth
cmap	matplotlib.colormap instance
cmax	maximum value of resistivity for colorbar. (linear)
cmin	minimum value of resistivity for colorbar (linear)
data_fn	full path fo data file
depth_index	integer value of depth slice for plotting
dpi	resolution of figure in dots-per-inch
dscale	depth scaling, computed internally
east_line_xlist	list of east mesh lines for faster plotting
east_line_ylist	list of east mesh lines for faster plotting
fdict	dictionary of font properties
fig	matplotlib.figure instance
fig_num	number of figure instance
fig_size	size of figure in inches
font_size	size of font in points
grid_east	location of east nodes in relative coordinates
grid_north	location of north nodes in relative coordinates
grid_z	location of vertical nodes in relative coordinates
initial_fn	full path to initial file
m_height	mean height of horizontal cells
m_width	mean width of horizontal cells
map_scale	[‘m’ | ‘km’] scale of map
mesh_east	np.meshgrid of east, north
mesh_north	np.meshgrid of east, north
mesh_plot	matplotlib.axes.pcolormesh instance
model_fn	full path to model file
new_initial_fn	full path to new initial file
nodes_east	spacing between east nodes
nodes_north	spacing between north nodes
nodes_z	spacing between vertical nodes
north_line_xlist	list of coordinates of north nodes for faster plotting
north_line_ylist	list of coordinates of north nodes for faster plotting
plot_yn	[‘y’ | ‘n’] plot on instantiation
radio_res	matplotlib.widget.radio instance for change resistivity
rect_selector	matplotlib.widget.rect_selector
res	np.ndarray(nx, ny, nz) for model in linear resistivity
res_copy	copy of res for undo
res_dict	dictionary of segmented resistivity values
res_list	list of resistivity values for model linear scale
res_model	np.ndarray(nx, ny, nz) of resistivity values from res_list (linear scale)
res_model_int	np.ndarray(nx, ny, nz) of integer values corresponding to res_list for initial model
res_value	current resistivty value of radio_res
save_path	path to save initial file to
station_east	station locations in east direction
station_north	station locations in north direction
xlimits	limits of plot in e-w direction
ylimits	limits of plot in n-s direction

Attributes

nodes_east

nodes_north

nodes_z

Methods

add_layers_to_mesh(self[, n_add_layers, ...])	Function to add constant thickness layers to the top or bottom of mesh.
add_topography_to_model2(self[, ...])	if air_layers is non-zero, will add topo: read in topograph file, make a surface model.
assign_resistivity_from_surfacedata(self, ...)	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)
change_model_res(self, xchange, ychange)	change resistivity values of resistivity model
get_model(self)	reads in initial file or model file and set attributes:
get_parameters(self)	get important model parameters to write to a file for documentation later.
interpolate_elevation2(self[, surfacefile, ...])	project a surface to the model grid and add resulting elevation data to a dictionary called surface_dict.
make_mesh(self)	create finite element mesh according to user-input parameters.
make_z_mesh_new(self)	new version of make_z_mesh.
plot(self)	plots the model with:
plot_mesh(self[, east_limits, north_limits, ...])	Plot the mesh to show model grid
plot_mesh_xy(self)	# add mesh grid lines in xy plan north-east map :return:
plot_mesh_xz(self)	display the mesh in North-Depth aspect :return:
plot_topography(self)	display topography elevation data together with station locations on a cell-index N-E map :return:
read_gocad_sgrid_file(self, sgrid_header_file)	read a gocad sgrid file and put this info into a ModEM file.
read_model_file(self[, model_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.
read_ws_model_file(self, ws_model_fn)	reads in a WS3INV3D model file
rect_onselect(self, eclick, erelease)	on selecting a rectangle change the colors to the resistivity values
redraw_plot(self)	redraws the plot
rewrite_model_file(self[, model_fn, ...])	write an initial file for wsinv3d from the model created.
set_res_list(self, res_list)	on setting res_list also set the res_dict to correspond
set_res_value(self, val)	
write_gocad_sgrid_file(self[, fn, origin, ...])	write a model to gocad sgrid
write_model_file(self, **kwargs)	will write an initial file for ModEM.
write_vtk_file(self[, vtk_save_path, ...])	write a vtk file to view in Paraview or other
write_xyres(self[, location_type, origin, ...])	write files containing depth slice data (x, y, res for each depth)

print_mesh_params	
print_model_file_summary	

change_model_res (*self*, *xchange*, *ychange*)
change resistivity values of resistivity model

get_model (*self*)

reads in initial file or model file and set attributes: -resmodel -northrid -eastrid -zgrid -res_list if initial file

plot (*self*)

plots the model with: -a radio dial for depth slice -radio dial for resistivity value

rect_onselect (*self*, *eclick*, *erelease*)
on selecting a rectangle change the colors to the resistivity values

redraw_plot (*self*)
redraws the plot

rewrite_model_file (*self*, *model_fn=None*, *save_path=None*, *model_fn_basename=None*)
write an initial file for wsinv3d from the model created.

set_res_list (*self*, *res_list*)
on setting res_list also set the res_dict to correspond

class mtpy.modeling.modem.**PlotResponse** (*data_fn=None*, *resp_fn=None*, ***kwargs*)
plot data and response

Plots the real and imaginary impedance and induction vector if present.

Example

```
>>> import mtpy.modeling.modem as modem
>>> dfn = r"/home/MT/ModEM/Inv1/DataFile.dat"
>>> rfn = r"/home/MT/ModEM/Inv1/Test_resp_000.dat"
>>> mrp = modem.PlotResponse(data_fn=dfn, resp_fn=rfn)
>>> # plot only the TE and TM modes
>>> mrp.plot_component = 2
>>> mrp.redraw_plot()
```

Attributes	Description
color_mode	['color' | 'bw'] color or black and white plots
cted	color for data Z_XX and Z_XY mode
ctem	color for model Z_XX and Z_XY mode
ctmd	color for data Z_YX and Z_YY mode
ctmm	color for model Z_YX and Z_YY mode
data_fn	full path to data file
data_object	WSResponse instance
e_capsize	cap size of error bars in points (*default* is .5)
e_capthick	cap thickness of error bars in points (*default* is 1)
fig_dpi	resolution of figure in dots-per-inch (300)
fig_list	list of matplotlib.figure instances for plots
fig_size	size of figure in inches (*default* is [6, 6])
font_size	size of font for tick labels, axes labels are font_size+2 (*default* is 7)
legend_border_axes_pad	padding between legend box and axes

Table 12 – continued from prev

Attributes	Description
legend_border_pad	padding between border of legend and symbols
legend_handle_text_pad	padding between text labels and symbols of legend
legend_label_spacing	padding between labels
legend_loc	location of legend
legend_marker_scale	scale of symbols in legend
lw	line width data curves (*default* is .5)
ms	size of markers (*default* is 1.5)
lw_r	line width response curves (*default* is .5)
ms_r	size of markers response curves (*default* is 1.5)
mted	marker for data Z_XX and Z_XY mode
mtem	marker for model Z_XX and Z_XY mode
mtmd	marker for data Z_YX and Z_YY mode
mtmm	marker for model Z_YX and Z_YY mode
phase_limits	limits of phase
plot_component	[2 | 4] 2 for TE and TM or 4 for all components
plot_style	[1 | 2] 1 to plot each mode in a seperate subplot and 2 to plot xx, xy and yx, yy in same plots
plot_type	['1' | list of station name] '1' to plot all stations in data file or input a list of station names to plot if statio
plot_z	[True | False] *default* is True to plot impedance, False for plotting resistivity and phase
plot_yn	['n' | 'y'] to plot on instantiation
res_limits	limits of resistivity in linear scale
resp_fn	full path to response file
resp_object	WSResponse object for resp_fn, or list of WSResponse objects if resp_fn is a list of response files
station_fn	full path to station file written by WSStation
subplot_bottom	space between axes and bottom of figure
subplot_hspace	space between subplots in vertical direction
subplot_left	space between axes and left of figure
subplot_right	space between axes and right of figure
subplot_top	space between axes and top of figure
subplot_wspace	space between subplots in horizontal direction

Methods

`redraw_plot`(self)	redraw plot if parameters were changed
`save_figure`(self, save_fn[, file_format, ...])	save_plot will save the figure to save_fn.

plot	

redraw_plot (*self*)

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
```

(continues on next page)

(continued from previous page)

```
>>> p1.lw = 2
>>> p1.redraw_plot()
```

save_figure (*self, save_fn, file_format='pdf', orientation='portrait', fig_dpi=None, close_fig='y'*)
save_plot will save the figure to save_fn.

class mtpy.modeling.modem.**PlotSlices** (*model_fn, data_fn=None, **kwargs*)

- Plot all cartesian axis-aligned slices and be able to scroll through the model
- Extract arbitrary profiles (e.g. along a seismic line) from a model

Example

```
>>> import mtpy.modeling.modem as modem
>>> mfn = r"/home/modem/Inv1/Modular_NLCG_100.rho"
>>> dfn = r"/home/modem/Inv1/ModEM_data.dat"
>>> pds = ws.PlotSlices(model_fn=mfn, data_fn=dfn)
```

Buttons	Description
'e'	moves n-s slice east by one model block
'w'	moves n-s slice west by one model block
'n'	moves e-w slice north by one model block
'm'	moves e-w slice south by one model block
'd'	moves depth slice down by one model block
'u'	moves depth slice up by one model block

Attributes	Description
ax_en	matplotlib.axes instance for depth slice map view
ax_ez	matplotlib.axes instance for e-w slice
ax_map	matplotlib.axes instance for location map
ax_nz	matplotlib.axes instance for n-s slice
climits	(min , max) color limits on resistivity in log scale. *default* is (0, 4)
cmap	name of color map for resisitiviy. *default* is 'jet_r'
data_fn	full path to data file name
draw_colorbar	show colorbar on exported plot; default True
dscale	scaling parameter depending on map_scale
east_line_xlist	list of line nodes of east grid for faster plotting
east_line_ylist	list of line nodes of east grid for faster plotting
ew_limits	(min, max) limits of e-w in map_scale units *default* is None and scales to station area
fig	matplotlib.figure instance for figure
fig_aspect	aspect ratio of plots. *default* is 1
fig_dpi	resolution of figure in dots-per-inch *default* is 300
fig_num	figure instance number
fig_size	[width, height] of figure window. *default* is [6,6]
font_dict	dictionary of font keywords, internally created
font_size	size of ticklables in points, axes labes are font_size+2. *default* is 4
grid_east	relative location of grid nodes in e-w direction in map_scale units
grid_north	relative location of grid nodes in n-s direction in map_scale units
grid_z	relative location of grid nodes in z direction in map_scale units
index_east	index value of grid_east being plotted

Continued on next page

Table 14 – continued from previous page

Attributes	Description
index_north	index value of grid_north being plotted
index_vertical	index value of grid_z being plotted
initial_fn	full path to initial file
key_press	matplotlib.canvas.connect instance
map_scale	['m' | 'km'] scale of map. *default* is km
mesh_east	np.meshgrid(grid_east, grid_north)[0]
mesh_en_east	np.meshgrid(grid_east, grid_north)[0]
mesh_en_north	np.meshgrid(grid_east, grid_north)[1]
mesh_ez_east	np.meshgrid(grid_east, grid_z)[0]
mesh_ez_vertical	np.meshgrid(grid_east, grid_z)[1]
mesh_north	np.meshgrid(grid_east, grid_north)[1]
mesh_nz_north	np.meshgrid(grid_north, grid_z)[0]
mesh_nz_vertical	np.meshgrid(grid_north, grid_z)[1]
model_fn	full path to model file
ms	size of station markers in points. *default* is 2
nodes_east	relative distance betwen nodes in e-w direction in map_scale units
nodes_north	relative distance betwen nodes in n-s direction in map_scale units
nodes_z	relative distance betwen nodes in z direction in map_scale units
north_line_xlist	list of line nodes north grid for faster plotting
north_line_ylist	list of line nodes north grid for faster plotting
ns_limits	(min, max) limits of plots in n-s direction *default* is None, set veiwing area to station area
plot_yn	['y' | 'n'] 'y' to plot on instantiation *default* is 'y'
plot_stations	default False
plot_grid	show grid on exported plot; default False
res_model	np.ndarray(n_north, n_east, n_vertical) of model resistivity values in linear scale
save_format	exported format; default png
save_path	path to save exported plots to; default current working folder
station_color	color of station marker. *default* is black
station_dict_east	location of stations for each east grid row
station_dict_north	location of stations for each north grid row
station_east	location of stations in east direction
station_fn	full path to station file
station_font_color	color of station label
station_font_pad	padding between station marker and label
station_font_rotation	angle of station label
station_font_size	font size of station label
station_font_weight	weight of font for station label
station_id	[min, max] index values for station labels
station_marker	station marker
station_names	name of stations
station_north	location of stations in north direction
subplot_bottom	distance between axes and bottom of figure window
subplot_hspace	distance between subplots in vertical direction
subplot_left	distance between axes and left of figure window
subplot_right	distance between axes and right of figure window
subplot_top	distance between axes and top of figure window
subplot_wspace	distance between subplots in horizontal direction
title	title of plot
xminorticks	location of xminorticks

Continued on next page

Table 14 – continued from previous page

Attributes	Description
yminorticks	location of yminorticks
z_limits	(min, max) limits in vertical direction,

Methods

`export_slices`(self[, plane, indexlist, ...])	Plot Slices
`get_slice`(self[, option, coords, nsteps, ...])	**param option** can be either of 'STA', 'XY' or 'XYZ'. For 'STA' or 'XY', a vertical
`get_station_grid_locations`(self)	get the grid line on which a station resides for plotting
`on_key_press`(self, event)	on a key press change the slices
`plot`(self)	plot:
`read_files`(self)	read in the files to get appropriate information
`redraw_plot`(self)	redraw plot if parameters were changed
`save_figure`(self[, save_fn, fig_dpi, ...])	save_figure will save the figure to save_fn.

export_slices (*self*, *plane='N-E'*, *indexlist=[]*, *station_buffer=200*, *save=True*)

Plot Slices

Parameters

- **plane** – must be either 'N-E', 'N-Z' or 'E-Z'
- **indexlist** – must be a list or 1d numpy array of indices
- **station_buffer** – spatial buffer (in metres) used around grid locations for selecting stations to be projected and plotted on profiles. Ignored if .plot_stations is set to False.

Returns [figlist, savepaths]. A list containing (1) lists of Figure objects, for further manipulation (2) corresponding paths for saving them to disk

get_slice (*self*, *option='STA'*, *coords=[]*, *nsteps=-1*, *nn=1*, *p=4*, *absolute_query_locations=False*, *extrapolate=True*)

Parameters

- **option** – can be either of 'STA', 'XY' or 'XYZ'. For 'STA' or 'XY', a vertical profile is returned based on station coordinates or arbitrary XY coordinates, respectively. For 'XYZ', interpolated values at those coordinates are returned
- **coords** – Numpy array of shape (np, 2) for option='XY' or of shape (np, 3) for option='XYZ', where np is the number of coordinates. Not used for option='STA', in which case a vertical profile is created based on station locations.
- **nsteps** – When option is set to 'STA' or 'XY', nsteps can be used to create a regular grid along the profile in the horizontal direction. By default, when nsteps=-1, the horizontal grid points are defined by station locations or values in the XY array. This parameter is ignored for option='XYZ'
- **nn** – Number of neighbours to use for interpolation. Nearest neighbour interpolation is returned when nn=1 (default). When nn>1, inverse distance weighted interpolation is

returned. See link below for more details: https://en.wikipedia.org/wiki/Inverse_distance_weighting

- **p** – Power parameter, which determines the relative influence of near and far neighbours during interpolation. For p<=3, causes interpolated values to be dominated by points far away. Larger values of p assign greater influence to values near the interpolated point.
- **absolute_query_locations** – if True, query locations are shifted to be centered on the center of station locations; default False, in which case the function treats query locations as relative coordinates. For option='STA', this parameter is ignored, since station locations are internally treated as relative coordinates
- **extrapolate** – Extrapolates values (default), which can be particularly useful for extracting values at nodes, since the field values are given for cell-centres.

Returns

1: when option is 'STA' or 'XY' gd, gz, gv : where gd, gz and gv are 2D grids of distance (along profile), depth and interpolated values, respectively. The shape of the 2D grids depend on the number of stations or the number of xy coordinates provided, for options 'STA' or 'XY', respectively, the number of vertical model grid points and whether regular gridding in the horizontal direction was enabled with nsteps>-1.

2: when option is 'XYZ' gv : list of interpolated values of shape (np)

get_station_grid_locations (*self*)
get the grid line on which a station resides for plotting

on_key_press (*self*, *event*)
on a key press change the slices

plot (*self*)

plot: east vs. vertical, north vs. vertical, east vs. north

read_files (*self*)
read in the files to get appropriate information

redraw_plot (*self*)
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 (*self*, *save_fn=None*, *fig_dpi=None*, *file_format='pdf'*, *orientation='landscape'*, *close_fig='y'*)
save_figure will save the figure to save_fn.

class mtpy.modeling.modem.**PlotRMSMaps** (*residual_fn*, ***kwargs*)
plots the RMS as (data-model)/(error) in map view for all components of the data file. Gets this infomration from the .res file output by ModEM.

Methods

plot(self)	plot rms in map view
plot_loop(self[, fig_format])	loop over all periods and save figures accordingly
save_figure(self[, save_path, . . .])	save figure in the desired format

read_residual_fn	
redraw_plot	

plot (*self*)
plot rms in map view

plot_loop (*self*, *fig_format='png'*)
loop over all periods and save figures accordingly

save_figure (*self*, *save_path=None*, *save_fn_basename=None*, *save_fig_dpi=None*, *fig_format='png'*, *fig_close=True*)
save figure in the desired format

Generate files for ModEM

revised by JP 2017 # revised by AK 2017 to bring across functionality from ak branch

class mtpy.modeling.modem.plot_response.**PlotResponse** (*data_fn=None*, *resp_fn=None*, ***kwargs*)

plot data and response

Plots the real and imaginary impedance and induction vector if present.

Example

```
>>> import mtpy.modeling.modem as modem
>>> dfn = r"/home/MT/ModEM/Inv1/DataFile.dat"
>>> rfn = r"/home/MT/ModEM/Inv1/Test_resp_000.dat"
>>> mrp = modem.PlotResponse(data_fn=dfn, resp_fn=rfn)
>>> # plot only the TE and TM modes
>>> mrp.plot_component = 2
>>> mrp.redraw_plot()
```

Attributes	Description
color_mode	['color' | 'bw'] color or black and white plots
cted	color for data Z_XX and Z_XY mode
ctem	color for model Z_XX and Z_XY mode
ctmd	color for data Z_YX and Z_YY mode
ctmm	color for model Z_YX and Z_YY mode
data_fn	full path to data file
data_object	WSResponse instance
e_capsize	cap size of error bars in points (*default* is .5)
e_capthick	cap thickness of error bars in points (*default* is 1)
fig_dpi	resolution of figure in dots-per-inch (300)
fig_list	list of matplotlib.figure instances for plots
fig_size	size of figure in inches (*default* is [6, 6])
font_size	size of font for tick labels, axes labels are font_size+2 (*default* is 7)
legend_border_axes_pad	padding between legend box and axes

Table 17 – continued from prev

Attributes	Description
legend_border_pad	padding between border of legend and symbols
legend_handle_text_pad	padding between text labels and symbols of legend
legend_label_spacing	padding between labels
legend_loc	location of legend
legend_marker_scale	scale of symbols in legend
lw	line width data curves (*default* is .5)
ms	size of markers (*default* is 1.5)
lw_r	line width response curves (*default* is .5)
ms_r	size of markers response curves (*default* is 1.5)
mted	marker for data Z_XX and Z_XY mode
mtem	marker for model Z_XX and Z_XY mode
mtmd	marker for data Z_YX and Z_YY mode
mtmm	marker for model Z_YX and Z_YY mode
phase_limits	limits of phase
plot_component	[2 | 4] 2 for TE and TM or 4 for all components
plot_style	[1 | 2] 1 to plot each mode in a seperate subplot and 2 to plot xx, xy and yx, yy in same plots
plot_type	['1' | list of station name] '1' to plot all stations in data file or input a list of station names to plot if statio
plot_z	[True | False] *default* is True to plot impedance, False for plotting resistivity and phase
plot_yn	['n' | 'y'] to plot on instantiation
res_limits	limits of resistivity in linear scale
resp_fn	full path to response file
resp_object	WSResponse object for resp_fn, or list of WSResponse objects if resp_fn is a list of response files
station_fn	full path to station file written by WSStation
subplot_bottom	space between axes and bottom of figure
subplot_hspace	space between subplots in vertical direction
subplot_left	space between axes and left of figure
subplot_right	space between axes and right of figure
subplot_top	space between axes and top of figure
subplot_wspace	space between subplots in horizontal direction

Methods

`redraw_plot`(self)	redraw plot if parameters were changed
`save_figure`(self, save_fn[, file_format, ...])	save_plot will save the figure to save_fn.

plot	

redraw_plot (*self*)

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
```

(continues on next page)

(continued from previous page)

```
>>> p1.lw = 2
>>> p1.redraw_plot()
```

save_figure (*self*, *save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_fig='y'*)
save_plot will save the figure to save_fn.

Generate files for ModEM

revised by JP 2017 # revised by AK 2017 to bring across functionality from ak branch

class mtpy.modeling.modem.plot_slices.**PlotSlices** (*model_fn*, *data_fn=None*, ***kwargs*)

- Plot all cartesian axis-aligned slices and be able to scroll through the model
- Extract arbitrary profiles (e.g. along a seismic line) from a model

Example

```
>>> import mtpy.modeling.modem as modem
>>> mfn = r"/home/modem/Inv1/Modular_NLCG_100.rho"
>>> dfn = r"/home/modem/Inv1/ModEM_data.dat"
>>> pds = ws.PlotSlices(model_fn=mfn, data_fn=dfn)
```

Buttons	Description
'e'	moves n-s slice east by one model block
'w'	moves n-s slice west by one model block
'n'	moves e-w slice north by one model block
'm'	moves e-w slice south by one model block
'd'	moves depth slice down by one model block
'u'	moves depth slice up by one model block

Attributes	Description
ax_en	matplotlib.axes instance for depth slice map view
ax_ez	matplotlib.axes instance for e-w slice
ax_map	matplotlib.axes instance for location map
ax_nz	matplotlib.axes instance for n-s slice
climits	(min , max) color limits on resistivity in log scale. *default* is (0, 4)
cmap	name of color map for resisitiviy. *default* is 'jet_r'
data_fn	full path to data file name
draw_colorbar	show colorbar on exported plot; default True
dscale	scaling parameter depending on map_scale
east_line_xlist	list of line nodes of east grid for faster plotting
east_line_ylist	list of line nodes of east grid for faster plotting
ew_limits	(min, max) limits of e-w in map_scale units *default* is None and scales to station area
fig	matplotlib.figure instance for figure
fig_aspect	aspect ratio of plots. *default* is 1
fig_dpi	resolution of figure in dots-per-inch *default* is 300
fig_num	figure instance number
fig_size	[width, height] of figure window. *default* is [6,6]
font_dict	dictionary of font keywords, internally created
font_size	size of ticklables in points, axes labes are font_size+2. *default* is 4
grid_east	relative location of grid nodes in e-w direction in map_scale units

Continued on next page

Table 19 – continued from previous page

Attributes	Description
grid_north	relative location of grid nodes in n-s direction in map_scale units
grid_z	relative location of grid nodes in z direction in map_scale units
index_east	index value of grid_east being plotted
index_north	index value of grid_north being plotted
index_vertical	index value of grid_z being plotted
initial_fn	full path to initial file
key_press	matplotlib.canvas.connect instance
map_scale	['m' | 'km'] scale of map. *default* is km
mesh_east	np.meshgrid(grid_east, grid_north)[0]
mesh_en_east	np.meshgrid(grid_east, grid_north)[0]
mesh_en_north	np.meshgrid(grid_east, grid_north)[1]
mesh_ez_east	np.meshgrid(grid_east, grid_z)[0]
mesh_ez_vertical	np.meshgrid(grid_east, grid_z)[1]
mesh_north	np.meshgrid(grid_east, grid_north)[1]
mesh_nz_north	np.meshgrid(grid_north, grid_z)[0]
mesh_nz_vertical	np.meshgrid(grid_north, grid_z)[1]
model_fn	full path to model file
ms	size of station markers in points. *default* is 2
nodes_east	relative distance betwen nodes in e-w direction in map_scale units
nodes_north	relative distance betwen nodes in n-s direction in map_scale units
nodes_z	relative distance betwen nodes in z direction in map_scale units
north_line_xlist	list of line nodes north grid for faster plotting
north_line_ylist	list of line nodes north grid for faster plotting
ns_limits	(min, max) limits of plots in n-s direction *default* is None, set veiwing area to station area
plot_yn	['y' | 'n'] 'y' to plot on instantiation *default* is 'y'
plot_stations	default False
plot_grid	show grid on exported plot; default False
res_model	np.ndarray(n_north, n_east, n_vertical) of model resistivity values in linear scale
save_format	exported format; default png
save_path	path to save exported plots to; default current working folder
station_color	color of station marker. *default* is black
station_dict_east	location of stations for each east grid row
station_dict_north	location of stations for each north grid row
station_east	location of stations in east direction
station_fn	full path to station file
station_font_color	color of station label
station_font_pad	padding between station marker and label
station_font_rotation	angle of station label
station_font_size	font size of station label
station_font_weight	weight of font for station label
station_id	[min, max] index values for station labels
station_marker	station marker
station_names	name of stations
station_north	location of stations in north direction
subplot_bottom	distance between axes and bottom of figure window
subplot_hspace	distance between subplots in vertical direction
subplot_left	distance between axes and left of figure window
subplot_right	distance between axes and right of figure window
subplot_top	distance between axes and top of figure window

Continued on next page

Table 19 – continued from previous page

Attributes	Description
subplot_wspace	distance between subplots in horizontal direction
title	title of plot
xminorticks	location of xminorticks
yminorticks	location of yminorticks
z_limits	(min, max) limits in vertical direction,

Methods

export_slices(self[, plane, indexlist, ...])	Plot Slices
get_slice(self[, option, coords, nsteps, ...])	**param option** can be either of 'STA', 'XY' or 'XYZ'. For 'STA' or 'XY', a vertical
get_station_grid_locations(self)	get the grid line on which a station resides for plotting
on_key_press(self, event)	on a key press change the slices
plot(self)	plot:
read_files(self)	read in the files to get appropriate information
redraw_plot(self)	redraw plot if parameters were changed
save_figure(self[, save_fn, fig_dpi, ...])	save_figure will save the figure to save_fn.

export_slices(*self*, *plane='N-E'*, *indexlist=[]*, *station_buffer=200*, *save=True*)
Plot Slices

Parameters

- **plane** – must be either 'N-E', 'N-Z' or 'E-Z'
- **indexlist** – must be a list or 1d numpy array of indices
- **station_buffer** – spatial buffer (in metres) used around grid locations for selecting stations to be projected and plotted on profiles. Ignored if .plot_stations is set to False.

Returns [figlist, savepaths]. A list containing (1) lists of Figure objects, for further manipulation (2) corresponding paths for saving them to disk

get_slice(*self*, *option='STA'*, *coords=[]*, *nsteps=-1*, *nn=1*, *p=4*, *absolute_query_locations=False*, *extrapolate=True*)

Parameters

- **option** – can be either of 'STA', 'XY' or 'XYZ'. For 'STA' or 'XY', a vertical profile is returned based on station coordinates or arbitrary XY coordinates, respectively. For 'XYZ', interpolated values at those coordinates are returned
- **coords** – Numpy array of shape (np, 2) for option='XY' or of shape (np, 3) for option='XYZ', where np is the number of coordinates. Not used for option='STA', in which case a vertical profile is created based on station locations.
- **nsteps** – When option is set to 'STA' or 'XY', nsteps can be used to create a regular grid along the profile in the horizontal direction. By default, when nsteps=-1, the horizontal grid points are defined by station locations or values in the XY array. This parameter is ignored for option='XYZ'

- **nn** – Number of neighbours to use for interpolation. Nearest neighbour interpolation is returned when nn=1 (default). When nn>1, inverse distance weighted interpolation is returned. See link below for more details: https://en.wikipedia.org/wiki/Inverse_distance_weighting
- **p** – Power parameter, which determines the relative influence of near and far neighbours during interpolation. For p<=3, causes interpolated values to be dominated by points far away. Larger values of p assign greater influence to values near the interpolated point.
- **absolute_query_locations** – if True, query locations are shifted to be centered on the center of station locations; default False, in which case the function treats query locations as relative coordinates. For option='STA', this parameter is ignored, since station locations are internally treated as relative coordinates
- **extrapolate** – Extrapolates values (default), which can be particularly useful for extracting values at nodes, since the field values are given for cell-centres.

Returns

1: when option is 'STA' or 'XY' gd, gz, gv : where gd, gz and gv are 2D grids of distance (along profile), depth and interpolated values, respectively. The shape of the 2D grids depend on the number of stations or the number of xy coordinates provided, for options 'STA' or 'XY', respectively, the number of vertical model grid points and whether regular gridding in the horizontal direction was enabled with nsteps>-1.

2: when option is 'XYZ' gv : list of interpolated values of shape (np)

get_station_grid_locations (*self*)
get the grid line on which a station resides for plotting

on_key_press (*self*, *event*)
on a key press change the slices

plot (*self*)

plot: east vs. vertical, north vs. vertical, east vs. north

read_files (*self*)
read in the files to get appropriate information

redraw_plot (*self*)
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 (*self*, *save_fn=None*, *fig_dpi=None*, *file_format='pdf'*, *orientation='landscape'*, *close_fig='y'*)
save_figure will save the figure to save_fn.

Create Phase Tensor Map from the ModEM's output Resistivity model

class mtpy.modeling.modem.phase_tensor_maps.**PlotPTMaps**(*data_fn=None*, *resp_fn=None*, *model_fn=None*, ***kwargs*)

Plot phase tensor maps including residual pt if response file is input.

Plot only data for one period

```
>>> import mtpy.modeling.ws3dinv as ws
>>> dfn = r"/home/MT/ws3dinv/Inv1/WSDataFile.dat"
>>> ptm = ws.PlotPTMaps(data_fn=dfn, plot_period_list=[0])
```

Plot data and model response

```
>>> import mtpy.modeling.ws3dinv as ws
>>> dfn = r"/home/MT/ws3dinv/Inv1/WSDataFile.dat"
>>> rfn = r"/home/MT/ws3dinv/Inv1/Test_resp.00"
>>> mfn = r"/home/MT/ws3dinv/Inv1/Test_model.00"
>>> ptm = ws.PlotPTMaps(data_fn=dfn, resp_fn=rfn, model_fn=mfn,
>>> ...                 plot_period_list=[0])
>>> # adjust colorbar
>>> ptm.cb_res_pad = 1.25
>>> ptm.redraw_plot()
```

Methods

get_period_attributes(self, periodIdx, key)	Returns, for a given period, a list of attribute values for key (e.g.
plot(self[, period, save2file])	Plot phase tensor maps for data and or response, each figure is of a different period.
plot_on_axes(self, ax, m, periodIdx[, …])	Plots phase tensors for a given period index.
redraw_plot(self)	redraw plot if parameters were changed
save_figure(self[, save_path, fig_dpi, …])	save_figure will save the figure to save_fn.
write_pt_data_to_gmt(self[, period, epsg, …])	write data to plot phase tensor ellipses in gmt.

write_pt_data_to_text	

get_period_attributes(*self*, *periodIdx*, *key*, *ptarray='data'*)

Returns, for a given period, a list of attribute values for key (e.g. skew, phimax, etc.).

Parameters

- **periodIdx** – index of period; print out _plot_period for periods available
- **key** – attribute key
- **ptarray** – name of data-array to access for retrieving attributes; can be either 'data', 'resp' or 'resid'

Returns numpy array of attribute values

plot(*self*, *period=0*, *save2file=None*, ***kwargs*)

Plot phase tensor maps for data and or response, each figure is of a different period. If response is input a third column is added which is the residual phase tensor showing where the model is not fitting the data well. The data is plotted in km.

Args: period: the period index to plot, default=0

Returns:

plot_on_axes(*self, ax, m, periodIdx, ptarray='data', ellipse_size_factor=10000, cvals=None, map_scale='m', centre_shift=[0, 0], plot_tipper='n', tipper_size_factor=100000.0, **kwargs*)

Plots phase tensors for a given period index.

Parameters

- **ax** – plot axis
- **m** – basemap instance
- **periodIdx** – period index
- **ptarray** – name of data-array to access for retrieving attributes; can be either 'data', 'resp' or 'resid'
- **ellipse_size_factor** – factor to control ellipse size
- **cvals** – list of colour values for colouring each ellipse; must be of the same length as the number of tuples for each period
- **map_scale** – map length scale
- **kwargs** – list of relevant matplotlib arguments (e.g. zorder, alpha, etc.)
- **plot_tipper** – string ('n', 'yr', 'yi', or 'yri') to plot no tipper, real only, imaginary only, or both
- **tipper_size_factor** – scaling factor for tipper vectors

redraw_plot(*self*)

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(*self, save_path=None, fig_dpi=None, file_format='pdf', orientation='landscape', close_fig='y'*)

save_figure will save the figure to save_fn.

write_pt_data_to_gmt(*self, period=None, epsg=None, savepath='.', center_utm=None, colorby='phimin', attribute='data', clim=None*)

write data to plot phase tensor ellipses in gmt. saves a gmt script and text file containing ellipse data

provide: period to plot (seconds) epsg for the projection the model was projected to (google "epsg your_projection_name" and you will find it) centre_utm - utm coordinates for centre position of model, if not

provided, script will try and extract it from data file

colorby - what to colour the ellipses by, 'phimin', 'phimax', or 'skew' attribute - attribute to plot 'data', 'resp', or 'resid' for data, response or residuals

Generate files for ModEM

revised by JP 2017 # revised by AK 2017 to bring across functionality from ak branch

class mtpy.modeling.modem.plot_rms_maps.**PlotRMSMaps**(*residual_fn*, ***kwargs*)
plots the RMS as (data-model)/(error) in map view for all components of the data file. Gets this infomration from the .res file output by ModEM.

Methods

plot(self)	plot rms in map view
plot_loop(self[, fig_format])	loop over all periods and save figures accordingly
save_figure(self[, save_path, ...])	save figure in the desired format

read_residual_fn	
redraw_plot	

plot(*self*)
plot rms in map view

plot_loop(*self*, *fig_format='png'*)
loop over all periods and save figures accordingly

save_figure(*self*, *save_path=None*, *save_fn_basename=None*, *save_fig_dpi=None*, *fig_format='png'*, *fig_close=True*)
save figure in the desired format

3.2 Module Occam 1D

- **Wrapper class to interact with Occam1D written by Kerry Keys at Scripps**

  adapted from the method of Constable et al., [1987].

  - This class only deals with the MT functionality of the Fortran code, so it can make the input files for computing the 1D MT response of an input model and or data. It can also read the output and plot them in a useful way.
  - Note that when you run the inversion code, the convergence is quite quick, within the first few iterations, so have a look at the L2 cure to decide which iteration to plot, otherwise if you look at iterations long after convergence the models will be unreliable.
  - Key, K., 2009, 1D inversion of multicomponent, multi-frequency marine CSEM data: Methodology and synthetic studies for resolving thin resistive layers: Geophysics, 74, F9–F20.
  - The original paper describing the Occam's inversion approach is:
  - Constable, S. C., R. L. Parker, and C. G. Constable, 1987, Occam's inversion — A practical algorithm for generating smooth models from electromagnetic sounding data, Geophysics, 52 (03), 289–300.

  Intended Use

```
>>> import mtpy.modeling.occam1d as occam1d
>>> #--> make a data file
```

(continues on next page)

(continued from previous page)

```
>>> 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
↪')
>>> #--> make a model file
>>> m1 = occam1d.Model()
>>> m1.write_model_file(save_path=d1.save_path, target_depth=15000)
>>> #--> make a startup file
>>> s1 = occam1d.Startup()
>>> s1.data_fn = d1.data_fn
>>> s1.model_fn = m1.model_fn
>>> s1.save_path = m1.save_path
>>> s1.write_startup_file()
>>> #--> run occam1d from python
>>> occam_path = r"/home/occam1d/Occam1D_executable"
>>> occam1d.Run(s1.startup_fn, occam_path, mode='TE')
>>> #--plot the L2 curve
>>> l2 = occam1d.PlotL2(d1.save_path, m1.model_fn)
>>> #--> see that iteration 7 is the optimum model to plot
>>> p1 = occam1d.Plot1DResponse()
>>> p1.data_te_fn = d1.data_fn
>>> p1.model_fn = m1.model_fn
>>> p1.iter_te_fn = r"/home/occam1d/mt01/TE/TE_7.iter"
>>> p1.resp_te_fn = r"/home/occam1d/mt01/TE/TE_7.resp"
>>> p1.plot()
```

@author: J. Peacock (Oct. 2013)

class mtpy.modeling.occam1d.**Data**(*data_fn=None*, ***kwargs*)

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')
```

Methods

read_data_file(self[, data_fn])	reads a 1D data file
read_resp_file(self[, resp_fn, data_fn])	read response file
write_data_file(self[, rp_tuple, edi_file, . . .])	make1Ddatafile will write a data file for Occam1D

read_data_file(*self*, *data_fn=None*)
reads a 1D data file

read_resp_file(*self*, *resp_fn=None*, *data_fn=None*)

read response file

resp_fn : full path to response file

data_fn : full path to data file

write_data_file(*self*, *rp_tuple=None*, *edi_file=None*, *save_path=None*, *mode='det'*, *res_err='data'*, *phase_err='data'*, *thetar=0*, *res_errorfloor=0.0*, *phase_errorfloor=0.0*, *z_errorfloor=0.0*, *remove_outofquadrant=False*)
make1Ddatafile will write a data file for Occam1D

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.**Model**(*model_fn=None*, ***kwargs*)

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")
```

Methods

read_iter_file(self[, iter_fn, model_fn])	read an 1D iteration file
read_model_file(self[, model_fn])	will read in model 1D file
write_model_file(self[, save_path])	Makes a 1D model file for Occam1D.

read_iter_file (*self*, *iter_fn=None*, *model_fn=None*)
read an 1D iteration file

read_model_file (*self*, *model_fn=None*)
will read in model 1D file

write_model_file (*self*, *save_path=None*, ***kwargs*)
Makes a 1D model file for Occam1D.

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*)
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

Continued on next page

Table 25 – continued from previous page

Attributes	Description
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
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

Methods

`plot`(self)	plot data, response and model
`redraw_plot`(self)	redraw plot if parameters were changed
`save_figure`(self, save_fn[, file_format, ...])	save_plot will save the figure to save_fn.
`update_plot`(self, fig)	update any parameters that where changed using the built-in draw from canvas.

plot (*self*)
plot data, response and model

redraw_plot (*self*)
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 (*self, save_fn, file_format='pdf', orientation='portrait', fig_dpi=None, close_plot='y'*)
save_plot will save the figure to save_fn.

update_plot (*self, 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.**PlotL2** (*dir_path, model_fn, **kwargs*)
plot L2 curve of iteration vs rms and roughness

Methods

plot(self)	plot L2 curve
redraw_plot(self)	redraw plot if parameters were changed
save_figure(self, save_fn[, file_format, ...])	save_plot will save the figure to save_fn.
update_plot(self)	update any parameters that where changed using the built-in draw from canvas.

plot (*self*)
plot L2 curve

redraw_plot (*self*)
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")
```

(continues on next page)

(continued from previous page)

```
>>> p1 = ocd.plotAllResponses()
>>> #change line width
>>> p1.lw = 2
>>> p1.redraw_plot()
```

save_figure (*self*, *save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_fig='y'*)
save_plot will save the figure to save_fn.

update_plot (*self*)
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.**Run** (*startup_fn=None*, *occam_path=None*, ***kwargs*)
run occam 1d from python given the correct files and location of occam1d executable

Methods

run_occam1d	

class mtpy.modeling.occam1d.**Startup** (*data_fn=None*, *model_fn=None*, ***kwargs*)
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

Methods

`read_startup_file`(self, startup_fn)	reads in a 1D input file
`write_startup_file`(self[, save_path])	Make a 1D input file for Occam 1D

read_startup_file(*self*, *startup_fn*)
reads in a 1D input file

inputfn : full path to input file

write_startup_file(*self*, *save_path=None*, ***kwargs*)
Make a 1D input file for Occam 1D

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

mtpy.modeling.occam1d.**build_run**()
build input files and run a suite of models in series (pretty quick so won't bother parallelise)

run Occam1d on each set of inputs. Occam is run twice. First to get the lowest possible misfit. we then set the target rms to a factor (default 1.05) times the minimum rms achieved and run to get the smoothest model.

author: Alison Kirkby (2016)

mtpy.modeling.occam1d.**divide_inputs**(*work_to_do*, *size*)
divide list of inputs into chunks to send to each processor

mtpy.modeling.occam1d.**generate_inputfiles**(***input_parameters*)
generate all the input files to run occam1d, return the path and the startup files to run.

author: Alison Kirkby (2016)

mtpy.modeling.occam1d.**get_strike**(*mt_object*, *fmin*, *fmax*, *strike_approx=0*)
get the strike from the z array, choosing the strike angle that is closest to the azimuth of the PT ellipse (PT strike).

if there is not strike available from the z array use the PT strike.

`mtpy.modeling.occam1d.`**`parse_arguments`**(*arguments*)
takes list of command line arguments obtained by passing in sys.argv reads these and returns a parser object

author: Alison Kirkby (2016)

`mtpy.modeling.occam1d.`**`update_inputs`**()
update input parameters from command line

author: Alison Kirkby (2016)

3.3 Module Occam 2D

Spin-off from 'occamtools' (Created August 2011, re-written August 2013)

Tools for Occam2D

authors: JP/LK

Classes:

- Data
- Model
- Setup
- Run
- Plot
- Mask

Functions:

- getdatetime
- makestartfiles
- writemeshfile
- writemodelfile
- writestartupfile
- read_datafile
- get_model_setup
- blocks_elements_setup

class `mtpy.modeling.occam2d_rewrite.`**`Data`**(*edi_path=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
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

data [is a list of dictioinaries containing the data for each station.]

keys include:

- 'station' – name of station
- 'offset' – profile line offset
- 'te_res' – TE resisitivity in linear scale
- 'tm_res' – TM resistivity in linear scale
- 'te_phase' – TE phase in degrees
- 'tm_phase' – TM phase in degrees in first quadrant
- 're_tip' – real part of tipper along profile
- 'im_tip' – imaginary part of tipper along profile

each key is a np.ndarray(2, num_freq) index 0 is for data index 1 is for error

Key Words/Attributes	Desctription
_data_header	header line in data file
_data_string	full data string
_profile_generated	[True | False] True if profile has already been generated.
_rotate_to_strike	[True | False] True to rotate data to strike angle. *default* is True
data	list of dictionaries of data for each station. see above
data_fn	full path to data file
data_list	list of lines to write to data file
edi_list	list of mtpy.core.mt instances for each .edi file read
edi_path	directory path where .edi files are
edi_type	['z' | 'spectra'] for .edi format
elevation_model	model elevation np.ndarray(east, north, elevation) in meters
elevation_profile	elevation along profile np.ndarray (x, elev) (m)
fn_basename	data file basename *default* is OccamDataFile.dat
freq	list of frequencies to use for the inversion
freq_max	max frequency to use in inversion. *default* is None
freq_min	min frequency to use in inversion. *default* is None
freq_num	number of frequencies to use in inversion

Continued o

Table 29 – continued from previous page

Key Words/Attributes	Desctription
geoelectric_strike	geoelectric strike angle assuming N = 0, E = 90
masked_data	similar to data, but any masked points are now 0
mode_dict	dictionary of model modes to chose from
model_mode	model mode to use for inversion, see above
num_edi	number of stations to invert for
occam_dict	dictionary of occam parameters to use internally
occam_format	occam format of data file. *default* is OCCAM2MTDATA_1.0
phase_te_err	percent error in phase for TE mode. *default* is 5
phase_tm_err	percent error in phase for TM mode. *default* is 5
profile_angle	angle of profile line realtive to N = 0, E = 90
profile_line	m, b coefficients for mx+b definition of profile line
res_te_err	percent error in resistivity for TE mode. *default* is 10
res_tm_err	percent error in resistivity for TM mode. *default* is 10
error_type	'floor' or 'absolute' - option to set error as floor (i.e. maximum of the data error and a specified value) or a s
save_path	directory to save files to
station_list	list of station for inversion
station_locations	station locations along profile line
tipper_err	percent error in tipper. *default* is 5
title	title in data file.

Methods	Description
_fill_data	fills the data array that is described above
_get_data_list	gets the lines to write to data file
_get_frequencies	gets frequency list to invert for
get_profile_origin	get profile origin in UTM coordinates
mask_points	masks points in data picked from plot_mask_points
plot_mask_points	plots data responses to interactively mask data points.
plot_resonse	plots data/model responses, returns PlotResponse data type.
read_data_file	read in existing data file and fill appropriate attributes.
write_data_file	write a data file according to Data attributes

Example Write Data File :: >>> import mtpy.modeling.occam2d as occam2d >>> 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) >>> # model just the tm mode and tipper >>> ocd.model_mode = 3 >>> ocd.save_path = r"/home/occam/Line1/Inv1" >>> ocd.write_data_file() >>> # mask points >>> ocd.plot_mask_points() >>> ocd.mask_points()

Methods

`generate_profile`(self)	Generate linear profile by regression of station locations.
`get_profile_origin`(self)	get the origin of the profile in real world coordinates
`mask_from_datafile`(self, mask_datafn)	reads a separate data file and applies mask from this data file.
`mask_points`(self, maskpoints_obj)	mask points and rewrite the data file
`plot_mask_points`(self[, data_fn, marker, . . .])	An interactive plotting tool to mask points an add errorbars

Continued on next page

Table 30 – continued from previous page

plot_profile(self, **kwargs)	Plot the projected profile line along with original station locations to make sure the line projected is correct.
plot_response(self, **kwargs)	plot data and model responses as apparent resistivity, phase and tipper.
project_elevation(self[, elevation_model])	projects elevation data into the profile
read_data_file(self[, data_fn])	Read in an existing data file and populate appropriate attributes
write_data_file(self[, data_fn])	Write a data file.

get_profile_origin(*self*)
get the origin of the profile in real world coordinates

Author: Alison Kirkby (2013)

NEED TO ADAPT THIS TO THE CURRENT SETUP.

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.

mask_points(*self*, *maskpoints_obj*)
mask points and rewrite the data file

NEED TO REDO THIS TO FIT THE CURRENT SETUP

plot_mask_points(*self*, *data_fn=None*, *marker='h'*, *res_err_inc=0.25*, *phase_err_inc=0.05*)
An interactive plotting tool to mask points an add errorbars

plot_response(*self*, ***kwargs*)
plot data and model responses as apparent resistivity, phase and tipper. See PlotResponse for key words.

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

write_data_file(*self*, *data_fn=None*)
Write a data file.

class mtpy.modeling.occam2d_rewrite.**Mask**(*edi_path=None*, ***kwargs*)
Allow masking of points from data file (effectively commenting them out, so the process is reversable). Inheriting from Data class.

Methods

generate_profile(self)	Generate linear profile by regression of station locations.
get_profile_origin(self)	get the origin of the profile in real world coordinates

Continued on next page

Table 31 – continued from previous page

`mask_from_datafile`(self, mask_datafn)	reads a separate data file and applies mask from this data file.
`mask_points`(self, maskpoints_obj)	mask points and rewrite the data file
`plot_mask_points`(self[, data_fn, marker, ...])	An interactive plotting tool to mask points an add errorbars
`plot_profile`(self, **kwargs)	Plot the projected profile line along with original station locations to make sure the line projected is correct.
`plot_response`(self, **kwargs)	plot data and model responses as apparent resistivity, phase and tipper.
`project_elevation`(self[, elevation_model])	projects elevation data into the profile
`read_data_file`(self[, data_fn])	Read in an existing data file and populate appropriate attributes
`write_data_file`(self[, data_fn])	Write a data file.

class `mtpy.modeling.occam2d_rewrite.`**`Mesh`**(*station_locations=None*, ***kwargs*)

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.

Methods

`add_elevation`(self[, elevation_profile])	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.
`build_mesh`(self)	Build the finite element mesh given the parameters defined by the attributes of Mesh.
`plot_mesh`(self, **kwargs)	Plot built mesh with station locations.
`read_mesh_file`(self, mesh_fn)	reads an occam2d 2D mesh file
`write_mesh_file`(self[, save_path, basename])	Write a finite element mesh file.

`add_elevation`(*self*, *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.

build_mesh(*self*)

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(*self*, ***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 (*self*, *mesh_fn*)
reads an occam2d 2D mesh file

write_mesh_file (*self*, *save_path=None*, *basename='Occam2DMesh'*)
Write a finite element mesh file.

Calls build_mesh if it already has not been called.

class mtpy.modeling.occam2d_rewrite.**Model** (*iter_fn=None*, *model_fn=None*, *mesh_fn=None*, ***kwargs*)
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.

Methods

build_model(self)	build the model from the mesh, regularization grid and model file
read_iter_file(self[, iter_fn])	Read an iteration file.
write_iter_file(self[, iter_fn])	write an iteration file if you need to for some reason, same as startup file
write_startup_file(self[, startup_fn, ...])	Write a startup file based on the parameters of startup class.

build_model (*self*)
build the model from the mesh, regularization grid and model file

read_iter_file (*self*, *iter_fn=None*)
Read an iteration file.

write_iter_file (*self*, *iter_fn=None*)
write an iteration file if you need to for some reason, same as startup file

exception mtpy.modeling.occam2d_rewrite.**OccamInputError**

class mtpy.modeling.occam2d_rewrite.**OccamPointPicker** (*ax_list*, *line_list*, *err_list*, *res_err_inc=0.05*, *phase_err_inc=0.02*, *marker='h'*)
This class helps the user interactively pick points to mask and add error bars.

Methods

__call__(self, event)	When the function is called the mouse events will be recorder for picking points to mask or change error bars.
inAxes(self, event)	gets the axes that the mouse is currently in.
inFigure(self, event)	gets the figure number that the mouse is in
on_close(self, event)	close the figure with a 'q' key event and disconnect the event ids

inAxes (*self*, *event*)
gets the axes that the mouse is currently in.

event: is a type axes_enter_event

inFigure (*self*, *event*)
gets the figure number that the mouse is in

on_close (*self*, *event*)
close the figure with a 'q' key event and disconnect the event ids

class mtpy.modeling.occam2d_rewrite.**PlotL2** (*iter_fn*, ***kwargs*)
Plot L2 curve of iteration vs rms and rms vs roughness.

Need to only input an .iter file, will read all similar .iter files to get the rms, iteration number and roughness of all similar .iter files.

Methods

plot(self)	plot L2 curve
redraw_plot(self)	redraw plot if parameters were changed
save_figure(self, save_fn[, file_format, ...])	save_plot will save the figure to save_fn.
update_plot(self)	update any parameters that where changed using the built-in draw from canvas.

plot (*self*)
plot L2 curve

redraw_plot (*self*)
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 (*self*, *save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_fig='y'*)
save_plot will save the figure to save_fn.

update_plot (*self*)
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.occam2d_rewrite.**PlotMisfitPseudoSection**(*data_fn*, *resp_fn*, ***kwargs*)

plot a pseudo section of the data and response if given

Methods

get_misfit(self)	compute misfit of MT response found from the model and the data.
plot(self)	plot pseudo section of data and response if given
redraw_plot(self)	redraw plot if parameters were changed
save_figure(self, save_fn[, file_format, ...])	save_plot will save the figure to save_fn.
update_plot(self)	update any parameters that where changed using the built-in draw from canvas.

get_misfit(*self*)

compute misfit of MT response found from the model and the data.

Need to normalize correctly

plot(*self*)

plot pseudo section of data and response if given

redraw_plot(*self*)

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(*self*, *save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot='y'*)

save_plot will save the figure to save_fn.

update_plot(*self*)

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.occam2d_rewrite.**PlotModel**(*iter_fn=None*, *data_fn=None*, ***kwargs*)

plot the 2D model found by Occam2D. The model is displayed as a meshgrid instead of model bricks. This speeds things up considerably.

Inherets the Model class to take advantage of the attributes and methods already coded.

Methods

`build_model`(self)	build the model from the mesh, regularization grid and model file
`plot`(self)	plotModel will plot the model output by occam2d in the iteration file.
`read_iter_file`(self[, iter_fn])	Read an iteration file.
`redraw_plot`(self)	redraw plot if parameters were changed
`save_figure`(self, save_fn[, file_format, . . .])	save_plot will save the figure to save_fn.
`update_plot`(self)	update any parameters that where changed using the built-in draw from canvas.
`write_iter_file`(self[, iter_fn])	write an iteration file if you need to for some reason, same as startup file
`write_startup_file`(self[, startup_fn, . . .])	Write a startup file based on the parameters of startup class.

plot (*self*)
plotModel will plot the model output by occam2d in the iteration file.

Example

```
>>> import mtpy.modeling.occam2d as occam2d
>>> itfn = r"/home/Occam2D/Line1/Inv1/Test_15.iter"
>>> model_plot = occam2d.PlotModel(itfn)
>>> model_plot.ms = 20
>>> model_plot.ylimits = (0,.350)
>>> model_plot.yscale = 'm'
>>> model_plot.spad = .10
>>> model_plot.ypad = .125
>>> model_plot.xpad = .025
>>> model_plot.climits = (0,2.5)
>>> model_plot.aspect = 'equal'
>>> model_plot.redraw_plot()
```

redraw_plot (*self*)
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 (*self, save_fn, file_format='pdf', orientation='portrait', fig_dpi=None, close_fig='y'*)
save_plot will save the figure to save_fn.

update_plot (*self*)
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.occam2d_rewrite.**PlotPseudoSection** (*data_fn*, *resp_fn=None*, ***kwargs*)

plot a pseudo section of the data and response if given

Methods

plot(self)	plot pseudo section of data and response if given
redraw_plot(self)	redraw plot if parameters were changed
save_figure(self, save_fn[, file_format, . . .])	save_plot will save the figure to save_fn.
update_plot(self)	update any parameters that where changed using the built-in draw from canvas.

plot (*self*)
plot pseudo section of data and response if given

redraw_plot (*self*)
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 (*self*, *save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot='y'*)
save_plot will save the figure to save_fn.

update_plot (*self*)
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
```

(continues on next page)

(continued from previous page)

```
>>> 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.occam2d_rewrite.**PlotResponse**(*data_fn*, *resp_fn=None*, ***kwargs*)

Helper class to deal with plotting the MT response and occam2d model.

Methods

plot(self)	plot the data and model response, if given, in individual plots.
redraw_plot(self)	redraw plot if parameters were changed
save_figures(self, save_path[, fig_fmt, ...])	save all the figure that are in self.fig_list

plot(*self*)

plot the data and model response, if given, in individual plots.

redraw_plot(*self*)

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(*self*, *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')
```

class mtpy.modeling.occam2d_rewrite.**Profile**(*edi_path=None*, *edi_list=[]*, ***kwargs*)

Takes data from .edi files to create a profile line for 2D modeling. Can project the stations onto a profile that is perpendicular to strike or a given profile direction.

If _rotate_to_strike is True, the impedance tensor and tipper are rotated to align with the geoelectric strike angle.

If _rotate_to_strike is True and geoelectric_strike is not given, then it is calculated using the phase tensor. First, 2D sections are estimated from the impedance tensor then the strike is estimated from the phase tensor azimuth + skew. This angle is then used to project the stations perpendicular to the strike angle.

If you want to project onto an angle not perpendicular to strike, give profile_angle and set _rotate_to_strike to False. This will project the impedance tensor and tipper to be perpendicular with the profile_angle.

Methods

generate_profile(self)	Generate linear profile by regression of station locations.
plot_profile(self, **kwargs)	Plot the projected profile line along with original station locations to make sure the line projected is correct.
project_elevation(self[, elevation_model])	projects elevation data into the profile

generate_profile (*self*)

Generate linear profile by regression of station locations.

If profile_angle is not None, then station are projected onto that line. Else, the a geoelectric strike is calculated from the data and the stations are projected onto an angle perpendicular to the estimated strike direction. If _rotate_to_strike is True, the impedance tensor and Tipper data are rotated to align with strike. Else, data is not rotated to strike.

To project stations onto a given line, set profile_angle and _rotate_to_strike to False. This will project the stations onto profile_angle and rotate the impedance tensor and tipper to be perpendicular to the profile_angle.

plot_profile (*self*, ***kwargs*)

Plot the projected profile line along with original station locations to make sure the line projected is correct.

Key Words	Description
fig_dpi	dots-per-inch resolution of figure *default* is 300
fig_num	number if figure instance *default* is 'Projected Profile'
fig_size	size of figure in inches (width, height) *default* is [5, 5]
fs	[float] font size in points of axes tick labels axes labels are fs+2 *default* is 6
lc	[string | (r, g, b)]color of profile line (see matplotlib.line for options) *default* is 'b' – blue
lw	float, width of profile line in points *default* is 1
marker	[string] marker for stations (see matplotlib.pyplot.plot) for options
mc	[string | (r, g, b)] color of projected stations. *default* is 'k' – black
ms	[float] size of station marker *default* is 5
station_id	[min, max] index values for station labels *default* is None

Example :: >>> edipath = r"/home/mt/edi_files" >>> pr = occam2d.Profile(edi_path=edipath) >>> pr.generate_profile() >>> # set station labels to only be from 1st to 4th index >>> # of station name >>> pr.plot_profile(station_id=[0,4])

project_elevation (*self*, *elevation_model=None*)

projects elevation data into the profile

class mtpy.modeling.occam2d_rewrite.**Regularization** (*station_locations=None*, ***kwargs*)

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.

Methods

`add_elevation`(self[, elevation_profile])	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.
`build_mesh`(self)	Build the finite element mesh given the parameters defined by the attributes of Mesh.
`build_regularization`(self)	Builds larger boxes around existing mesh blocks for the regularization.
`get_num_free_params`(self)	estimate the number of free parameters in model mesh.
`plot_mesh`(self, **kwargs)	Plot built mesh with station locations.
`read_mesh_file`(self, mesh_fn)	reads an occam2d 2D mesh file
`read_regularization_file`(self, reg_fn)	Read in a regularization file and populate attributes:
`write_mesh_file`(self[, save_path, basename])	Write a finite element mesh file.
`write_regularization_file`(self[, reg_fn, …])	Write a regularization file for input into occam.

build_regularization(*self*)

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(*self*)

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(*self*, *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(*self*, *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

class mtpy.modeling.occam2d_rewrite.**Response**(*resp_fn=None*, ***kwargs*)

Reads .resp file output by Occam. Similar structure to Data.data.

If resp_fn is given in the initialization of Response, read_response_file is called.

Methods

read_response_file(self[, resp_fn])	read in response file and put into a list of dictionaries similar to Data

read_response_file(*self*, *resp_fn=None*)

read in response file and put into a list of dictionaries similar to Data

class mtpy.modeling.occam2d_rewrite.**Run**

Run Occam2D by system call.

Future plan: implement Occam in Python and call it from here directly.

class mtpy.modeling.occam2d_rewrite.**Startup**(***kwargs*)

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'
diago-nal_penalties	penalties on diagonal terms *default* is 0
format	Occam file format *default* is 'OCCAMITER_FLEX'
iteration	current iteration number *default* is 0
itera-tions_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_steps	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
rough-ness_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"
```

Methods

write_startup_file(self[, startup_fn, ...])	Write a startup file based on the parameters of startup class.

write_startup_file (*self*, *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

3.4 Module Winglink

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*)

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.

Methods

get_misfit(self)	compute misfit of MT response found from the model and the data.
plot(self)	plot pseudo section of data and response if given
redraw_plot(self)	redraw plot if parameters were changed
save_figure(self, save_fn[, file_format, . . .])	save_plot will save the figure to save_fn.
update_plot(self)	update any parameters that where changed using the built-in draw from canvas.

get_misfit(*self*)

compute misfit of MT response found from the model and the data.

Need to normalize correctly

plot(*self*)

plot pseudo section of data and response if given

redraw_plot(*self*)

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(*self*, *save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot='y'*)

save_plot will save the figure to save_fn.

update_plot(*self*)

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*)

plot a pseudo section of the data and response if given

Methods

plot(self)	plot pseudo section of data and response if given
redraw_plot(self)	redraw plot if parameters were changed
save_figure(self, save_fn[, file_format, ...])	save_plot will save the figure to save_fn.
update_plot(self)	update any parameters that where changed using the built-in draw from canvas.

plot(*self*)
plot pseudo section of data and response if given

redraw_plot(*self*)
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(*self*, *save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot='y'*)
save_plot will save the figure to save_fn.

update_plot(*self*)
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*)
Helper class to deal with plotting the MT response and occam2d model.

Methods

plot(self)	plot the data and model response, if given, in individual plots.
redraw_plot(self)	redraw plot if parameters were changed
save_figures(self, save_path[, fig_fmt, ...])	save all the figure that are in self.fig_list

plot (*self*)
plot the data and model response, if given, in individual plots.

redraw_plot (*self*)
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 (*self, 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**

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.

3.5 Module WS3DINV

- **Deals with input and output files for ws3dinv written by:** Siripunvaraporn, W.; Egbert, G.; Lenbury, Y. & Uyeshima, M. Three-dimensional magnetotelluric inversion: data-space method Physics of The Earth and Planetary Interiors, 2005, 150, 3-14 * Dependencies: matplotlib 1.3.x, numpy 1.7.x, scipy 0.13

  and evtk if vtk files want to be written.

The intended use or workflow is something like this for getting started:

Making input files

```
>>> import mtpy.modeling.ws3dinv as ws
>>> 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 edi_path
>>> ...             if edi.find('.edi') > 0]
>>> #2) make a grid from the stations themselves with 200m cell spacing
>>> wsmesh = ws.WSMesh(edi_list=edi_list, cell_size_east=200,
>>> ...                   cell_size_north=200)
>>> wsmesh.make_mesh()
>>> # check to see if the mesh is what you think it should be
>>> wsmesh.plot_mesh()
>>> # all is good write the mesh file
>>> wsmesh.write_initial_file(save_path=r"/home/ws3dinv/Inv1")
>>> # note this will write a file with relative station locations
>>> #change the starting model to be different than a halfspace
>>> mm = ws.WS3DModelManipulator(initial_fn=wsmesh.initial_fn)
>>> # an interactive gui will pop up to change the resistivity model
>>> #once finished write a new initial file
>>> mm.rewrite_initial_file()
>>> #3) write data file
>>> wsdata = ws.WSData(edi_list=edi_list, station_fn=wsmesh.station_fn)
>>> wsdata.write_data_file()
>>> #4) plot mt response to make sure everything looks ok
>>> rp = ws.PlotResponse(data_fn=wsdata.data_fn)
>>> #5) make startup file
>>> sws = ws.WSStartup(data_fn=wsdata.data_fn, initial_fn=mm.new_initial_
↪fn)
```

checking the model and response

```
>>> mfn = r"/home/ws3dinv/Inv1/test_model.01"
>>> dfn = r"/home/ws3dinv/Inv1/WSDataFile.dat"
>>> rfn = r"/home/ws3dinv/Inv1/test_resp.01"
>>> sfn = r"/home/ws3dinv/Inv1/WS_Sation_Locations.txt"
>>> # plot the data vs. model response
>>> rp = ws.PlotResponse(data_fn=dfn, resp_fn=rfn, station_fn=sfn)
>>> # plot model slices where you can interactively step through
>>> ds = ws.PlotSlices(model_fn=mfn, station_fn=sfn)
>>> # plot phase tensor ellipses on top of depth slices
>>> ptm = ws.PlotPTMaps(data_fn=dfn, resp_fn=rfn, model_fn=mfn)
>>> #write files for 3D visualization in Paraview or Mayavi
>>> ws.write_vtk_files(mfn, sfn, r"/home/ParaviewFiles")
```

Created on Sun Aug 25 18:41:15 2013

@author: jpeacock-pr

class mtpy.modeling.ws3dinv.**PlotDepthSlice**(*model_fn=None*, *data_fn=None*, *station_fn=None*, *initial_fn=None*, ***kwargs*)

Plots depth slices of resistivity model

Example

```
>>> import mtpy.modeling.ws3dinv as ws
>>> mfn = r"/home/MT/ws3dinv/Inv1/Test_model.00"
>>> sfn = r"/home/MT/ws3dinv/Inv1/WSStationLocations.txt"
>>> # plot just first layer to check the formatting
>>> pds = ws.PlotDepthSlice(model_fn=mfn, station_fn=sfn,
>>> ...                        depth_index=0, save_plots='n')
>>> #move color bar up
>>> pds.cb_location
>>> (0.64500000000000002, 0.14999999999999997, 0.3, 0.025)
>>> pds.cb_location = (.645, .175, .3, .025)
>>> pds.redraw_plot()
>>> #looks good now plot all depth slices and save them to a folder
>>> pds.save_path = r"/home/MT/ws3dinv/Inv1/DepthSlices"
>>> pds.depth_index = None
>>> pds.save_plots = 'y'
>>> pds.redraw_plot()
```

Attributes	Description
cb_location	location of color bar (x, y, width, height) *default* is None, automatically locates
cb_orientation	['vertical' | 'horizontal'] *default* is horizontal
cb_pad	padding between axes and colorbar *default* is None
cb_shrink	percentage to shrink colorbar by *default* is None
climits	(min, max) of resistivity color on log scale *default* is (0, 4)
cmap	name of color map *default* is 'jet_r'
data_fn	full path to data file
depth_index	integer value of depth slice index, shallowest layer is 0
dscale	scaling parameter depending on map_scale
ew_limits	(min, max) plot limits in e-w direction in map_scale units. *default* is None, sets viewing area to the station area
fig_aspect	aspect ratio of plot. *default* is 1
fig_dpi	resolution of figure in dots-per-inch. *default* is 300
fig_list	list of matplotlib.figure instances for each depth slice
fig_size	[width, height] in inches of figure size *default* is [6, 6]
font_size	size of ticklabel font in points, labels are font_size+2. *default* is 7
grid_east	relative location of grid nodes in e-w direction in map_scale units
grid_north	relative location of grid nodes in n-s direction in map_scale units
grid_z	relative location of grid nodes in z direction in map_scale units
initial_fn	full path to initial file
map_scale	['km' | 'm'] distance units of map. *default* is km
mesh_east	np.meshgrid(grid_east, grid_north, indexing='ij')
mesh_north	np.meshgrid(grid_east, grid_north, indexing='ij')
model_fn	full path to model file
nodes_east	relative distance betwen nodes in e-w direction in map_scale units
nodes_north	relative distance betwen nodes in n-s direction in map_scale units
nodes_z	relative distance betwen nodes in z direction in map_scale units
ns_limits	(min, max) plot limits in n-s direction in map_scale units. *default* is None, sets viewing area to the station area
plot_grid	['y' | 'n'] 'y' to plot mesh grid lines. *default* is 'n'
plot_yn	['y' | 'n'] 'y' to plot on instantiation
res_model	np.ndarray(n_north, n_east, n_vertical) of model resistivity values in linear scale
save_path	path to save figures to
save_plots	['y' | 'n'] 'y' to save depth slices to save_path
station_east	location of stations in east direction in map_scale units
station_fn	full path to station locations file

Continued on next page

Table 47 – continued from previous page

Attributes	Description
station_names	station names
station_north	location of station in north direction in map_scale units
subplot_bottom	distance between axes and bottom of figure window
subplot_left	distance between axes and left of figure window
subplot_right	distance between axes and right of figure window
subplot_top	distance between axes and top of figure window
title	titiel of plot *default* is depth of slice
xminorticks	location of xminorticks
yminorticks	location of yminorticks

Methods

`plot`(self)	plot depth slices
`read_files`(self)	read in the files to get appropriate information
`redraw_plot`(self)	redraw plot if parameters were changed
`update_plot`(self, fig)	update any parameters that where changed using the built-in draw from canvas.

plot (*self*)
plot depth slices

read_files (*self*)
read in the files to get appropriate information

redraw_plot (*self*)
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()
```

update_plot (*self, 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.ws3dinv.**PlotPTMaps**(*data_fn=None*, *resp_fn=None*, *station_fn=None*, *model_fn=None*, *initial_fn=None*, ***kwargs*)

Plot phase tensor maps including residual pt if response file is input.

Plot only data for one period

```
>>> import mtpy.modeling.ws3dinv as ws
>>> dfn = r"/home/MT/ws3dinv/Inv1/WSDataFile.dat"
>>> ptm = ws.PlotPTMaps(data_fn=dfn, plot_period_list=[0])
```

Plot data and model response

```
>>> import mtpy.modeling.ws3dinv as ws
>>> dfn = r"/home/MT/ws3dinv/Inv1/WSDataFile.dat"
>>> rfn = r"/home/MT/ws3dinv/Inv1/Test_resp.00"
>>> mfn = r"/home/MT/ws3dinv/Inv1/Test_model.00"
>>> ptm = ws.PlotPTMaps(data_fn=dfn, resp_fn=rfn, model_fn=mfn,
>>> ...                 plot_period_list=[0])
>>> # adjust colorbar
>>> ptm.cb_res_pad = 1.25
>>> ptm.redraw_plot()
```

Attributes	Description
cb_pt_pad	percentage from top of axes to place pt color bar. *default* is .90
cb_res_pad	percentage from bottom of axes to place resistivity color bar. *default* is 1.2
cb_residual_tick_step	tick step for residual pt. *default* is 3
cb_tick_step	tick step for phase tensor color bar, *default* is 45
data	np.ndarray(n_station, n_periods, 2, 2) impedance tensors for station data
data_fn	full path to data fle
dscale	scaling parameter depending on map_scale
ellipse_cmap	color map for pt ellipses. *default* is mt_bl2gr2rd
ellipse_colorby	**['skew' | 'skew_seg' | 'phimin' | 'phimax'|** 'phidet' | 'ellipticity'] parameter to color ellipses by. *default* is 'phimin'
ellipse_range	(min, max, step) min and max of colormap, need to input step if plotting skew_seg
ellipse_size	relative size of ellipses in map_scale
ew_limits	limits of plot in e-w direction in map_scale units. *default* is None, scales to station area
fig_aspect	aspect of figure. *default* is 1
fig_dpi	resolution in dots-per-inch. *default* is 300
fig_list	list of matplotlib.figure instances for each figure plotted.
fig_size	[width, height] in inches of figure window *default* is [6, 6]
font_size	font size of ticklabels, axes labels are font_size+2. *default* is 7
grid_east	relative location of grid nodes in e-w direction in map_scale units

Continued on next page

Table 49 – continued from previous page

Attributes	Description
grid_north	relative location of grid nodes in n-s direction in map_scale units
grid_z	relative location of grid nodes in z direction in map_scale units
initial_fn	full path to initial file
map_scale	['km' | 'm'] distance units of map. *default* is km
mesh_east	np.meshgrid(grid_east, grid_north, indexing='ij')
mesh_north	np.meshgrid(grid_east, grid_north, indexing='ij')
model_fn	full path to model file
nodes_east	relative distance betwen nodes in e-w direction in map_scale units
nodes_north	relative distance betwen nodes in n-s direction in map_scale units
nodes_z	relative distance betwen nodes in z direction in map_scale units
ns_limits	(min, max) limits of plot in n-s direction *default* is None, viewing area is station area
pad_east	padding from extreme stations in east direction
pad_north	padding from extreme stations in north direction
period_list	list of periods from data
plot_grid	['y' | 'n'] 'y' to plot grid lines *default* is 'n'
plot_period_list	list of period index values to plot *default* is None
plot_yn	['y' | 'n'] 'y' to plot on instantiation *default* is 'y'
res_cmap	colormap for resisitivity values. *default* is 'jet_r'
res_limits	(min, max) resistivity limits in log scale *default* is (0, 4)
res_model	np.ndarray(n_north, n_east, n_vertical) of model resistivity values in linear scale
residual_cmap	color map for pt residuals. *default* is 'mt_wh2or'
resp	np.ndarray(n_stations, n_periods, 2, 2) impedance tensors for model response
resp_fn	full path to response file
save_path	directory to save figures to
save_plots	['y' | 'n'] 'y' to save plots to save_path
station_east	location of stations in east direction in map_scale units
station_fn	full path to station locations file
station_names	station names
station_north	location of station in north direction in map_scale units
subplot_bottom	distance between axes and bottom of figure window
subplot_left	distance between axes and left of figure window
subplot_right	distance between axes and right of figure window
subplot_top	distance between axes and top of figure window
title	titiel of plot *default* is depth of slice
xminorticks	location of xminorticks
yminorticks	location of yminorticks

Methods

`plot`(self)	plot phase tensor maps for data and or response, each figure is of a different period.
`redraw_plot`(self)	redraw plot if parameters were changed
`save_figure`(self[, save_path, fig_dpi, ...])	save_figure will save the figure to save_fn.

plot (*self*)

plot phase tensor maps for data and or response, each figure is of a different period. If response is input a third column is added which is the residual phase tensor showing where the model is not fitting the data well. The data is plotted in km in units of ohm-m.

Inputs: data_fn = full path to data file resp_fn = full path to response file, if none just plots data sites_fn = full path to sites file periodlst = indicies of periods you want to plot esize = size of ellipses as:

0 = phase tensor ellipse 1 = phase tensor residual 2 = resistivity tensor ellipse 3 = resistivity tensor residual

ecolor = 'phimin' for coloring with phimin or 'beta' for beta coloring colormm = list of min and max coloring for plot, list as follows:

0 = phase tensor min and max for ecolor in degrees 1 = phase tensor residual min and max [0,1] 2 = resistivity tensor coloring as resistivity on log scale 3 = resistivity tensor residual coloring as resistivity on

linear scale

xpad = padding of map from stations at extremities (km) units = 'mv' to convert to Ohm-m dpi = dots per inch of figure

redraw_plot (*self*)

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 (*self*, *save_path=None*, *fig_dpi=None*, *file_format='pdf'*, *orientation='landscape'*, *close_fig='y'*)

save_figure will save the figure to save_fn.

class mtpy.modeling.ws3dinv.**PlotResponse** (*data_fn=None*, *resp_fn=None*, *station_fn=None*, ***kwargs*)

plot data and response

Example

```
>>> import mtpy.modeling.ws3dinv as ws
>>> dfn = r"/home/MT/ws3dinv/Inv1/WSDataFile.dat"
>>> rfn = r"/home/MT/ws3dinv/Inv1/Test_resp.00"
>>> sfn = r"/home/MT/ws3dinv/Inv1/WSStationLocations.txt"
```

(continues on next page)

(continued from previous page)

```
>>> wsrp = ws.PlotResponse(data_fn=dfn, resp_fn=rfn, station_fn=sfn)
>>> # plot only the TE and TM modes
>>> wsrp.plot_component = 2
>>> wsrp.redraw_plot()
```

Attributes	Description
color_mode	['color' | 'bw'] color or black and white plots
cted	color for data TE mode
ctem	color for data TM mode
ctmd	color for model TE mode
ctmm	color for model TM mode
data_fn	full path to data file
data_object	WSResponse instance
e_capsize	cap size of error bars in points (*default* is .5)
e_capthick	cap thickness of error bars in points (*default* is 1)
fig_dpi	resolution of figure in dots-per-inch (300)
fig_list	list of matplotlib.figure instances for plots
fig_size	size of figure in inches (*default* is [6, 6])
font_size	size of font for tick labels, axes labels are font_size+2 (*default* is 7)
legend_border_axes_pad	padding between legend box and axes
legend_border_pad	padding between border of legend and symbols
legend_handle_text_pad	padding between text labels and symbols of legend
legend_label_spacing	padding between labels
legend_loc	location of legend
legend_marker_scale	scale of symbols in legend
lw	line width response curves (*default* is .5)
ms	size of markers (*default* is 1.5)
mted	marker for data TE mode
mtem	marker for data TM mode
mtmd	marker for model TE mode
mtmm	marker for model TM mode
phase_limits	limits of phase
plot_component	[2 | 4] 2 for TE and TM or 4 for all components
plot_style	[1 | 2] 1 to plot each mode in a seperate subplot and 2 to plot xx, xy and yx, yy in same plots
plot_type	['1' | list of station name] '1' to plot all stations in data file or input a list of station names to plot if statio
plot_z	[True | False] *default* is True to plot impedance, False for plotting resistivity and phase
plot_yn	['n' | 'y'] to plot on instantiation
res_limits	limits of resistivity in linear scale
resp_fn	full path to response file
resp_object	WSResponse object for resp_fn, or list of WSResponse objects if resp_fn is a list of response files
station_fn	full path to station file written by WSStation
subplot_bottom	space between axes and bottom of figure
subplot_hspace	space between subplots in vertical direction
subplot_left	space between axes and left of figure
subplot_right	space between axes and right of figure
subplot_top	space between axes and top of figure
subplot_wspace	space between subplots in horizontal direction

Methods

plot(self)	
plot_errorbar(self, ax, period, data, error, ...)	convinience function to make an error bar instance
redraw_plot(self)	redraw plot if parameters were changed
save_figure(self, save_fn[, file_format, ...])	save_plot will save the figure to save_fn.
update_plot(self)	update any parameters that where changed using the built-in draw from canvas.

plot (*self*)

plot_errorbar (*self*, *ax*, *period*, *data*, *error*, *color*, *marker*)
convinience function to make an error bar instance

redraw_plot (*self*)
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 (*self*, *save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_fig='y'*)
save_plot will save the figure to save_fn.

update_plot (*self*)
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.ws3dinv.**PlotSlices** (*model_fn*, *data_fn=None*, *station_fn=None*, *initial_fn=None*, ***kwargs*)
plot all slices and be able to scroll through the model

Example

```
>>> import mtpy.modeling.ws3dinv as ws
>>> mfn = r"/home/MT/ws3dinv/Inv1/Test_model.00"
>>> sfn = r"/home/MT/ws3dinv/Inv1/WSStationLocations.txt"
>>> # plot just first layer to check the formating
>>> pds = ws.PlotSlices(model_fn=mfn, station_fn=sfn)
```

Buttons	Description
‘e’	moves n-s slice east by one model block
‘w’	moves n-s slice west by one model block
‘n’	moves e-w slice north by one model block
‘m’	moves e-w slice south by one model block
‘d’	moves depth slice down by one model block
‘u’	moves depth slice up by one model block

Attributes	Description
ax_en	matplotlib.axes instance for depth slice map view
ax_ez	matplotlib.axes instance for e-w slice
ax_map	matplotlib.axes instance for location map
ax_nz	matplotlib.axes instance for n-s slice
climits	(min , max) color limits on resistivity in log scale. *default* is (0, 4)
cmap	name of color map for resisitiviy. *default* is ‘jet_r’
data_fn	full path to data file name
dscale	scaling parameter depending on map_scale
east_line_xlist	list of line nodes of east grid for faster plotting
east_line_ylist	list of line nodes of east grid for faster plotting
ew_limits	(min, max) limits of e-w in map_scale units *default* is None and scales to station area
fig	matplotlib.figure instance for figure
fig_aspect	aspect ratio of plots. *default* is 1
fig_dpi	resolution of figure in dots-per-inch *default* is 300
fig_num	figure instance number
fig_size	[width, height] of figure window. *default* is [6,6]
font_dict	dictionary of font keywords, internally created
font_size	size of ticklables in points, axes labes are font_size+2. *default* is 7
grid_east	relative location of grid nodes in e-w direction in map_scale units
grid_north	relative location of grid nodes in n-s direction in map_scale units
grid_z	relative location of grid nodes in z direction in map_scale units
index_east	index value of grid_east being plotted
index_north	index value of grid_north being plotted
index_vertical	index value of grid_z being plotted
initial_fn	full path to initial file
key_press	matplotlib.canvas.connect instance
map_scale	[‘m’ | ‘km’] scale of map. *default* is km
mesh_east	np.meshgrid(grid_east, grid_north)[0]
mesh_en_east	np.meshgrid(grid_east, grid_north)[0]
mesh_en_north	np.meshgrid(grid_east, grid_north)[1]
mesh_ez_east	np.meshgrid(grid_east, grid_z)[0]
mesh_ez_vertical	np.meshgrid(grid_east, grid_z)[1]
mesh_north	np.meshgrid(grid_east, grid_north)[1]
mesh_nz_north	np.meshgrid(grid_north, grid_z)[0]
mesh_nz_vertical	np.meshgrid(grid_north, grid_z)[1]
model_fn	full path to model file
ms	size of station markers in points. *default* is 2
nodes_east	relative distance betwen nodes in e-w direction in map_scale units
nodes_north	relative distance betwen nodes in n-s direction in map_scale units
nodes_z	relative distance betwen nodes in z direction in map_scale units
north_line_xlist	list of line nodes north grid for faster plotting

Continued on next page

Table 53 – continued from previous page

Attributes	Description
north_line_ylist	list of line nodes north grid for faster plotting
ns_limits	(min, max) limits of plots in n-s direction *default* is None, set veiwing area to station area
plot_yn	['y' | 'n'] 'y' to plot on instantiation *default* is 'y'
res_model	np.ndarray(n_north, n_east, n_vertical) of model resistivity values in linear scale
station_color	color of station marker. *default* is black
station_dict_east	location of stations for each east grid row
station_dict_north	location of stations for each north grid row
station_east	location of stations in east direction
station_fn	full path to station file
station_font_color	color of station label
station_font_pad	padding between station marker and label
station_font_rotation	angle of station label
station_font_size	font size of station label
station_font_weight	weight of font for station label
station_id	[min, max] index values for station labels
station_marker	station marker
station_names	name of stations
station_north	location of stations in north direction
subplot_bottom	distance between axes and bottom of figure window
subplot_hspace	distance between subplots in vertical direction
subplot_left	distance between axes and left of figure window
subplot_right	distance between axes and right of figure window
subplot_top	distance between axes and top of figure window
subplot_wspace	distance between subplots in horizontal direction
title	title of plot
z_limits	(min, max) limits in vertical direction,

Methods

get_station_grid_locations(self)	get the grid line on which a station resides for plotting
on_key_press(self, event)	on a key press change the slices
plot(self)	plot:
read_files(self)	read in the files to get appropriate information
redraw_plot(self)	redraw plot if parameters were changed
save_figure(self[, save_fn, fig_dpi, ...])	save_figure will save the figure to save_fn.

get_station_grid_locations (*self*)
get the grid line on which a station resides for plotting

on_key_press (*self*, *event*)
on a key press change the slices

plot (*self*)

plot: east vs. vertical, north vs. vertical, east vs. north

read_files (*self*)
read in the files to get appropriate information

redraw_plot (*self*)

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(*self*, *save_fn=None*, *fig_dpi=None*, *file_format='pdf'*, *orientation='landscape'*, *close_fig='y'*)
save_figure will save the figure to save_fn.

class mtpy.modeling.ws3dinv.**WSData**(***kwargs*)
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_file	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

Methods

build_data(self)	Builds the data from .edi files to be written into a data file
compute_errors(self)	compute the errors from the given attributes
read_data_file(self[, data_fn, wl_sites_fn, ...])	read in data file
write_data_file(self, **kwargs)	Writes a data file based on the attribute data

build_data (*self*)
Builds the data from .edi files to be written into a data file

Need to call this if any parameters have been reset to write a correct data file.

compute_errors (*self*)
compute the errors from the given attributes

read_data_file (*self*, *data_fn=None*, *wl_sites_fn=None*, *station_fn=None*)
read in data file

write_data_file (*self*, ***kwargs*)
Writes a data file based on the attribute data

exception mtpy.modeling.ws3dinv.**WSInputError**

class mtpy.modeling.ws3dinv.**WSMesh** (*edi_list=None*, ***kwargs*)
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.

Example

```
>>> import mtpy.modeling.ws3dinv as ws
>>> 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 edi_path
>>> ...           if edi.find('.edi') > 0]
>>> #2) make a grid from the stations themselves with 200m cell␣
↪spacing
>>> wsmesh = ws.WSMesh(edi_list=edi_list, cell_size_east=200,
>>> ...                cell_size_north=200)
>>> wsmesh.make_mesh()
>>> # check to see if the mesh is what you think it should be
>>> wsmesh.plot_mesh()
>>> # all is good write the mesh file
>>> wsmesh.write_initial_file(save_path=r"/home/ws3dinv/Inv1")
```

Attributes	Description
cell_size_east	mesh block width in east direction *default* is 500
cell_size_north	mesh block width in north direction *default* is 500
edi_list	list of .edi files to invert for
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
initial_fn	full path to initial file name
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 5
pad_north	number of cells for padding on S and N sides *default* is 5
pad_root_east	padding cells E & W will be pad_root_east**(x)
pad_root_north	padding cells N & S will be pad_root_north**(x)
pad_z	number of cells for padding at bottom *default* is 5
res_list	list of resistivity values for starting model
res_model	starting resistivity model
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
station_fn	full path to station file
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

Methods	Description
make_mesh	makes a mesh from the given specifications
plot_mesh	plots mesh to make sure everything is good
write_initial_file	writes an initial model file that includes the mesh

Methods

`convert_model_to_int`(self)	convert the resistivity model that is in ohm-m to integer values corresponding to res_list
`make_mesh`(self)	create finite element mesh according to parameters set.
`plot_mesh`(self[, east_limits, north_limits, . . .])	
`read_initial_file`(self, 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.
`write_initial_file`(self, **kwargs)	will write an initial file for wsinv3d.

convert_model_to_int(*self*)

convert the resistivity model that is in ohm-m to integer values corresponding to res_list

make_mesh (*self*)
create finite element mesh according to parameters set.

The mesh is built by first finding the center of the station area. Then cells are added in the north and east direction with width cell_size_east and cell_size_north to the extremeties of the station area. Padding cells are then added to extend the model to reduce edge effects. The number of cells are pad_east and pad_north and the increase in size is by pad_root_east and pad_root_north. The station locations are then computed as the center of the nearest cell as required by the code.

The vertical cells are built to increase in size exponentially with depth. The first cell depth is first_layer_thickness and should be about 1/10th the shortest skin depth. The layers then increase on a log scale to z_target_depth. Then the model is padded with pad_z number of cells to extend the depth of the model.

padding = np.round(cell_size_east*pad_root_eastnp.arange(start=.5,** stop=3,
step=3./pad_east))+west

plot_mesh (*self*, *east_limits=None*, *north_limits=None*, *z_limits=None*, ***kwargs*)

read_initial_file (*self*, *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.

write_initial_file (*self*, ***kwargs*)
will write an initial file for wsinv3d.

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.

class mtpy.modeling.ws3dinv.**WSModel** (*model_fn=None*)
Reads in model file and fills necessary attributes.

Example

```
>>> mfn = r"/home/ws3dinv/test_model.00"
>>> wsmodel = ws.WSModel(mfn)
>>> wsmodel.write_vtk_file(r"/home/ParaviewFiles")
```

Attributes	Description
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
iteration_number	iteration number of the inversion
lagrange	lagrange multiplier
model_fn	full path to model file
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
res_model	starting resistivity model
rms	root mean squared error of data and model

Methods	Description
read_model_file	read model file and fill attributes
write_vtk_file	write a vtk structured grid file for resistivity model

Methods

read_model_file(self)	read in a model file as x-north, y-east, z-positive down

write_vtk_file	

read_model_file(*self*)
read in a model file as x-north, y-east, z-positive down

write_vtk_file(*self*, *save_fn*)

class mtpy.modeling.ws3dinv.**WSModelManipulator**(*model_fn=None*, *initial_fn=None*, *data_fn=None*, ***kwargs*)
will plot a model from wsinv3d or init file so the user can manipulate the resistivity values relatively easily. At the moment only plotted in map view.

Example :: >>> import mtpy.modeling.ws3dinv as ws >>> initial_fn = r"/home/MT/ws3dinv/Inv1/WSInitialFile" >>> mm = ws.WSModelManipulator(initial_fn=initial_fn)

Buttons	Description
'='	increase depth to next vertical node (deeper)
'-'	decrease depth to next vertical node (shallower)
'q'	quit the plot, rewrites initial file when pressed
'a'	copies the above horizontal layer to the present layer
'b'	copies the below horizonal layer to present layer
'u'	undo previous change

Attributes	Description
ax1	matplotlib.axes instance for mesh plot of the model
ax2	matplotlib.axes instance of colorbar
cb	matplotlib.colorbar instance for colorbar
cid_depth	matplotlib.canvas.connect for depth
cmap	matplotlib.colormap instance
cmax	maximum value of resistivity for colorbar. (linear)
cmin	minimum value of resistivity for colorbar (linear)
data_fn	full path fo data file
depth_index	integer value of depth slice for plotting
dpi	resolution of figure in dots-per-inch
dscale	depth scaling, computed internally
east_line_xlist	list of east mesh lines for faster plotting
east_line_ylist	list of east mesh lines for faster plotting
fdict	dictionary of font properties
fig	matplotlib.figure instance

Continued on next page

Table 58 – continued from previous page

Attributes	Description
fig_num	number of figure instance
fig_size	size of figure in inches
font_size	size of font in points
grid_east	location of east nodes in relative coordinates
grid_north	location of north nodes in relative coordinates
grid_z	location of vertical nodes in relative coordinates
initial_fn	full path to initial file
m_height	mean height of horizontal cells
m_width	mean width of horizontal cells
map_scale	['m' | 'km'] scale of map
mesh_east	np.meshgrid of east, north
mesh_north	np.meshgrid of east, north
mesh_plot	matplotlib.axes.pcolormesh instance
model_fn	full path to model file
new_initial_fn	full path to new initial file
nodes_east	spacing between east nodes
nodes_north	spacing between north nodes
nodes_z	spacing between vertical nodes
north_line_xlist	list of coordinates of north nodes for faster plotting
north_line_ylist	list of coordinates of north nodes for faster plotting
plot_yn	['y' | 'n'] plot on instantiation
radio_res	matplotlib.widget.radio instance for change resistivity
rect_selector	matplotlib.widget.rect_selector
res	np.ndarray(nx, ny, nz) for model in linear resistivity
res_copy	copy of res for undo
res_dict	dictionary of segmented resistivity values
res_list	list of resistivity values for model linear scale
res_model	np.ndarray(nx, ny, nz) of resistivity values from res_list (linear scale)
res_model_int	np.ndarray(nx, ny, nz) of integer values corresponding to res_list for initial model
res_value	current resistivty value of radio_res
save_path	path to save initial file to
station_east	station locations in east direction
station_north	station locations in north direction
xlimits	limits of plot in e-w direction
ylimits	limits of plot in n-s direction

Methods

`change_model_res`(self, xchange, ychange)	change resistivity values of resistivity model
`convert_model_to_int`(self)	convert the resistivity model that is in ohm-m to integer values corresponding to res_list
`convert_res_to_model`(self, res_array)	converts an output model into an array of segmented valued according to res_list.
`plot`(self)	plots the model with:
`read_file`(self)	reads in initial file or model file and set attributes:
`rect_onselect`(self, eclick, erelease)	on selecting a rectangle change the colors to the resistivity values
`redraw_plot`(self)	redraws the plot

Continued on next page

Table 59 – continued from previous page

rewrite_initial_file(self[, save_path])	write an initial file for wsinv3d from the model created.
set_res_list(self, res_list)	on setting res_list also set the res_dict to correspond

set_res_value	

change_model_res (*self*, *xchange*, *ychange*)
change resistivity values of resistivity model

convert_model_to_int (*self*)
convert the resistivity model that is in ohm-m to integer values corresponding to res_list

convert_res_to_model (*self*, *res_array*)
converts an output model into an array of segmented valued according to res_list.

output is an array of segemented resistivity values in ohm-m (linear)

plot (*self*)

plots the model with: -a radio dial for depth slice -radio dial for resistivity value

read_file (*self*)

reads in initial file or model file and set attributes: -resmodel -northrid -eastrid -zgrid -res_list if initial file

rect_onselect (*self*, *eclick*, *erelease*)
on selecting a rectangle change the colors to the resistivity values

redraw_plot (*self*)
redraws the plot

rewrite_initial_file (*self*, *save_path=None*)
write an initial file for wsinv3d from the model created.

set_res_list (*self*, *res_list*)
on setting res_list also set the res_dict to correspond

class mtpy.modeling.ws3dinv.**WSResponse** (*resp_fn=None*, *station_fn=None*, *wl_station_fn=None*)
class to deal with .resp file output by ws3dinv

Attributes	Description
n_z	number of vertical layers
period_list	list of periods inverted for
resp	**np.ndarray structured with keys:** • *station* –> station name • ***east* –> relative eastern location in** grid • ***north* –> relative northern location in** grid • ***z_resp* –> impedance tensor array** of response with shape (n_stations, n_freq, 4, dtype=complex) • *z_resp_err–> response impedance tensor error
resp_fn	full path to response file
station_east	location of stations in east direction
station_fn	full path to station file written by WSStation
station_names	names of stations
station_north	location of stations in north direction
units	['mv' | 'other'] units of impedance tensor
wl_sites_fn	full path to .sites file from Winglink
z_resp	impedance tensors of response with shape (n_stations, n_periods, 2, 2)
z_resp_err	impedance tensors errors of response with shape (n_stations, n_periods, 2, 2) (zeros)

Methods	Description
read_resp_file	read response file and fill attributes

Methods

read_resp_file(self[, resp_fn, wl_sites_fn, ...])	read in data file

read_resp_file(*self*, *resp_fn=None*, *wl_sites_fn=None*, *station_fn=None*)
read in data file

class mtpy.modeling.ws3dinv.**WSStartup**(*data_fn=None*, *initial_fn=None*, ***kwargs*)
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

Methods

read_startup_file(self[, startup_fn])	read startup file fills attributes
write_startup_file(self)	makes a startup file for WSINV3D.

read_startup_file(*self*, *startup_fn=None*)
read startup file fills attributes

write_startup_file(*self*)
makes a startup file for WSINV3D.

class mtpy.modeling.ws3dinv.**WSStation**(*station_fn=None*, ***kwargs*)
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

Methods

from_wl_write_station_file(self, sites_file, ...)	write a ws station file from the outputs of winglink
read_station_file(self[, station_fn])	read in station file written by write_station_file

Continued on next page

Table 62 – continued from previous page

write_station_file(self[, east, north, . . .])	write a station file to go with the data file.
write_vtk_file(self, save_path[, vtk_basename])	write a vtk file to plot stations

from_wl_write_station_file(*self*, *sites_file*, *out_file*, *ncol=5*)
write a ws station file from the outputs of winglink

read_station_file(*self*, *station_fn=None*)
read in station file written by write_station_file

write_station_file(*self*, *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.

write_vtk_file(*self*, *save_path*, *vtk_basename='VTKStations'*)
write a vtk file to plot stations

mtpy.modeling.ws3dinv.**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.modeling.ws3dinv.**computeMemoryUsage**(*nx*, *ny*, *nz*, *n_stations*, *n_zelements*, *n_period*)
compute the memory usage of a model

mtpy.modeling.ws3dinv.**estimate_skin_depth**(*res_model*, *grid_z*, *period*, *dscale=1000*)
estimate the skin depth from the resistivity model assuming that

delta_skin ~ 500 * sqrt(rho_a*T)

mtpy.modeling.ws3dinv.**write_vtk_files**(*model_fn*, *station_fn*, *save_path*)
writes vtk files

mtpy.modeling.ws3dinv.**write_vtk_res_model**(*res_model*, *grid_north*, *grid_east*, *grid_z*, *save_fn*)
Write a vtk file for resistivity as a structured grid to be read into paraview or mayavi

Doesn't work properly under windows

adds extension automatically

mtpy.modeling.ws3dinv.**write_vtk_stations**(*station_north*, *station_east*, *save_fn*, *station_z=None*)
Write a vtk file as points to be read into paraview or mayavi

Doesn't work properly under windows

adds extension automatically

CHAPTER 4

Package Imaging

4.1 Penetration Depth

Description: For a given input edi file, plot the Penetration Depth vs all the periods (1/freq). Or input a directory of edi multi-files (*.edi), the program will loop to plot the penetration depth profile for each edi.

Author: fei.zhang@ga.gov.au Date: 2017-01-23

`mtpy.imaging.penetration_depth1d.`**`plot_edi_dir`**(*edi_path, rholist=['zxy', 'zyx', 'det'], fig_dpi=400, savefile=None*)

plot edi files from the input directory edi_path

`mtpy.imaging.penetration_depth1d.`**`plot_edi_file`**(*edifile, rholist=['zxy', 'zyx', 'det'], savefile=None, fig_dpi=400*)

Plot the input edi_file Args:

edi_file: path2edifile rholist: a list of the rho to be used. savefile: path2savefig, not save if None

Returns:

Description: With an input edi_file_folder and a list of period index, generate a profile using occam2d module, then plot the Penetration Depth profile at the given periods vs the stations locations.

Usage: python mtpy/imaging/penetration_depth2d.py /path2/edi_files_dir/ period_index_list python mtpy/imaging/penetration_depth2d.py.py examples/data/edi2/ 0 1 10 20 30 40

Author: fei.zhang@ga.gov.au Date: 2017-01-23

`mtpy.imaging.penetration_depth2d.`**`barplot_multi_station_penentration_depth`**(*edifiles_dir, per_index=0, zcomponent='det'*)

A simple bar chart plot of the penetration depth across multiple edi files (stations), at the given (frequency) per_index. No profile-projection is done in this funciton. :param edifiles_dir: a list of edi files, or a dir of edi :param per_index: an integer smaller than the number of MT frequencies in the edi files. :return:

Description: Given a set of EDI files plot the Penetration Depth vs the station_location. Note that the values of periods within10% tolerance (ptol=0.1) are considered as equal. Setting a smaller value for ptol(=0.05) may result less MT sites data included.

Usage: python mtpy/imaging/penetration_depth3d.py /path2/edi_files_dir/ period_index

Author: fei.zhang@ga.gov.au Date: 2017-01-23

`mtpy.imaging.penetration_depth3d.`**`create_csv_file`**(*edi_dir*, *outputcsv=None*, *zcomponent='det'*)

Loop over all edi files, and create a csv file with columns: lat, lon, pendepth0, pendepth1, ... :param edi_dir: path_to_edifiles_dir :param zcomponent: det | zxy | zyx :param outputcsv: path2output.csv file :return:

`mtpy.imaging.penetration_depth3d.`**`create_shapefile`**(*edi_dir*, *outputfile=None*, *zcomponent='det'*)

create a shapefile for station, penetration_depths :param edi_dir: :param outputfile: :param zcomponent: :return:

`mtpy.imaging.penetration_depth3d.`**`get_index2`**(**args*, ***kwargs*)

Mapping of lat lon to a grid :param lat: :param lon: :param ref_lon: :param ref_lat: :param pixelsize: :return:

`mtpy.imaging.penetration_depth3d.`**`get_penetration_depths_from_edi_file`**(*edifile*, *rholist=['det']*)

Compute the penetration depths of an edi file :param edifile: input edifile :param rholist: flag the method to compute penetration depth: det zxy zyx :return: a tuple:(station_lat, statoin_lon, periods_list, pendepth_list)

`mtpy.imaging.penetration_depth3d.`**`plot_bar3d_depth`**(*edifiles*, *per_index*, *whichrho='det'*)

plot 3D bar of penetration depths For a given freq/period index of a set of edifiles/dir, the station,periods, pendepth,(lat, lon) are extracted the geo-bounding box calculated, and the mapping from stations to grids is constructed and plotted.

Parameters

- **`whichrho`** – z component either 'det', 'zxy' or 'zyx'
- **`edifiles`** – an edi_dir or list of edi_files
- **`per_index`** – period index number 0,1,2

Returns

`mtpy.imaging.penetration_depth3d.`**`plot_latlon_depth_profile`**(*edi_dir*, *period*, *zcomponent='det'*, *showfig=True*, *savefig=True*, *savepath=None*, *fig_dpi=400*, *fontsize=14*, *file_format='png'*, *ptol=0.1*)

MT penetration depth profile in lat-lon coordinates with pixelsize = 0.002 :param savefig: :param showfig: :param edi_dir: :param period: :param zcomponent: :return:

`mtpy.imaging.penetration_depth3d.`**`reverse_colourmap`**(**args*, ***kwargs*)

In: cmap, name Out: my_cmap_r

Explanation: http://stackoverflow.com/questions/3279560/invert-colormap-in-matplotlib

Description: This file defines imaging functions for penetration. The plotting function are extracted and implemented in plot() of each class from penetration_depth1D.py, penetration_depth2D.py and penetration_depth3D.py

Usage: see descriptions of each clases

Author: YingzhiGou Date: 20/06/2017

class mtpy.imaging.penetration.**Depth1D**(*edis=None*, *rholist=set(['det', 'zxy', 'zyx'])*)
Description: For a given input MT object, plot the Penetration Depth vs all the periods (1/freq).

Attributes

data the data (mt objects) that are to be plotted

fig matplotlib fig object

Methods

close(self)	close the figure :return:
show(self[, block])	display the image :return:

export_image	
get_data	
get_figure	
plot	
set_data	
set_rholist	

class mtpy.imaging.penetration.**Depth2D**(*data=None*, *period_index_list=None*, *rho='det'*)
With a list of MT object and a list of period index, generate a profile using occam2d module, then plot the Penetration Depth profile at the given periods vs the stations locations.

Attributes

data the data (mt objects) that are to be plotted

fig matplotlib fig object

Methods

close(self)	close the figure :return:
show(self[, block])	display the image :return:

export_image	
get_data	
get_figure	
plot	
set_data	
set_period_index_list	
set_rho	

class mtpy.imaging.penetration.**Depth3D**(*edis=None*, *period=None*, *rho='det'*, *ptol=0.1*)
For a set of EDI files (input as a list of MT objects), plot the Penetration Depth vs the station_location, for a given period value or index Note that the values of periods within tolerance (ptol=0.1) are considered as equal. Setting a smaller value for ptol may result less MT sites data included.

Attributes

data the data (mt objects) that are to be plotted

fig matplotlib fig object

Methods

close(self)	close the figure :return:
show(self[, block])	display the image :return:

export_image	
get_data	
get_figure	
get_period_fmt	
plot	
set_data	
set_period	
set_rho	

exception mtpy.imaging.penetration.**ZComponentError**(**args*, ***kwargs*)

mtpy.imaging.penetration.**check_period_values**(*period_list*, *ptol=0.1*)
check if all the values are equal in the input list :param period_list: a list of period :param ptol=0.1 # 1% percentage tolerance of period values considered as equal :return: True/False

mtpy.imaging.penetration.**get_bounding_box**(*latlons*)
get min max lat lon from the list of lat-lon-pairs points

mtpy.imaging.penetration.**get_index**(*lat*, *lon*, *minlat*, *minlon*, *pixelsize*, *offset=0*)
compute the grid index from the lat lon float value :param lat: float lat :param lon: float lon :param minlat: min lat at low left corner :param minlon: min long at left :param pixelsize: pixel size in lat long degree :param offset: a shift of grid index. should be =0. :return: a paire of integer

mtpy.imaging.penetration.**get_penetration_depth**(*mt_obj_list*, *per_index*, *whichrho='det'*)
compute the penetration depth of mt_obj at the given period_index, and using whichrho option :param per_index: the index of periods 0, 1, … :param mt_obj_list: list of edi file paths or mt objects :param whichrho: det, zxy, or zyx :return:

mtpy.imaging.penetration.**get_penetration_depth_generic**(*edi_file_list*, *period_sec*, *whichrho='det'*, *ptol=0.1*)
This is a more generic and useful function to compute the penetration depths of a list of edi files at given period_sec (seconds). No assumption is made about the edi files period list. A tolerance of 10% is used to identify the relevant edi files which contain the period of interest.

Parameters

- **ptol** – freq error/tolerance, need to be consistent with phase_tensor_map.py, default is 0.1
- **edi_file_list** – edi file list of mt object list
- **period_sec** – the float number value of the period in second: 0.1, …20.0
- **whichrho** –

Returns tuple of (stations, periods, penetrationdepth, lat-lons-pairs)

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

class mtpy.imaging.plot_resphase_maps.**PlotResPhaseMaps**(***kwargs*)

Plots apparent resistivity and phase in map view from a list of edi files

Methods

plot(self, freq, type, vmin, vmax[, ...])	**param freq** plot frequency

plot(*self, freq, type, vmin, vmax, extrapolation_buffer_degrees=1, regular_grid_nx=100, regular_grid_ny=100, nn=7, p=4, show_stations=True, show_station_names=False, save_path='/home/docs/checkouts/readthedocs.org/user_builds/mtpy2/checkouts/stable/docs/source', file_ext='png', cmap='rainbow', show=True*)

Parameters

- **freq** – plot frequency
- **type** – plot type; can be either 'res' or 'phase'
- **vmin** – minimum value used in color-mapping
- **vmax** – maximum value used in color-mapping
- **extrapolation_buffer_degrees** – extrapolation buffer in degrees
- **regular_grid_nx** – number of longitudinal grid points to use during interpolation
- **regular_grid_ny** – number of latitudinal grid points to use during interpolation
- **nn** – number of nearest neighbours to use in inverse distance weighted interpolation
- **p** – power parameter in inverse distance weighted interpolation
- **save_path** – path where plot is saved
- **file_ext** – file extension
- **show** – boolean to toggle display of plot

Returns fig object

4.2 Module Plot Phase Tensor Maps

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:

class mtpy.imaging.phase_tensor_maps.**PlotPhaseTensorMaps**(***kwargs*)

Plots phase tensor ellipses in map view from a list of edi files

Attributes

rot_z rotation angle(s)

Methods

export_params_to_file(self[, save_path])	write text files for all the phase tensor parameters.
plot(self[, fig, save_path, show, raster_dict])	Plots the phase tensor map.
redraw_plot(self)	use this function if you updated some attributes and want to re-plot.
save_figure(self, save_fn[, file_format, . . .])	save_plot will save the figure to save_fn.
update_plot(self)	update any parameters that where changed using the built-in draw from canvas.

export_params_to_file(*self*, *save_path=None*)
write text files for all the phase tensor parameters. :param save_path: string path to save files into. File naming pattern is like save_path/PhaseTensorTipper_Params_freq.csv/table **Files Content **

*station *lon *lat *phi_min *phi_max *skew *ellipticity *azimuth *tipper_mag_real *tipper_ang_real *tipper_mag_imag *tipper_ang_imag

Returns path2savedfile

plot(*self*, *fig=None*, *save_path=None*, *show=True*, *raster_dict={'cbar_title': 'Arbitrary units', 'lats': [], 'levels': 50, 'cbar_position': None, 'cmap': 'rainbow', 'vals': [], 'lons': []}*)
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: Plotting of raster data is currently only supported when mapscale='deg'.

This parameter is a dictionary of parameters for plotting raster data, on top of which phase tensor data are plotted. 'lons', 'lats' and 'vals' are one dimensional lists (or numpy arrays) for longitudes, latitudes and corresponding values, respectively. 'levels', 'cmap' and 'cbar_title' are the number of levels to be used in the colormap, the colormap and its title, respectively.

redraw_plot(*self*)
use this function if you updated some attributes and want to re-plot.

rot_z
rotation angle(s)

save_figure(*self*, *save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot='y'*)
save_plot will save the figure to save_fn.

update_plot(*self*)
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

4.3 Module PlotPhaseTensorPseudoSection

Created on Thu May 30 18:10:55 2013

@author: jpeacock-pr

class mtpy.imaging.phase_tensor_pseudosection.**PlotPhaseTensorPseudoSection**(***kwargs*)
PlotPhaseTensorPseudoSection will plot the phase tensor ellipses in a pseudo section format

Attributes

rot_z rotation angle(s)

Methods

plot(self[, show])	plots the phase tensor pseudo section.
redraw_plot(self)	use this function if you updated some attributes and want to re-plot.
save_figure(self, save_fn[, file_format, . . .])	save_plot will save the figure to save_fn.
save_figure2(self, save_fn[, file_format, . . .])	save_plot will save the figure to save_fn.
update_plot(self)	update any parameters that where changed using the built-in draw from canvas.
writeTextFiles(self[, save_path, ptol])	This will write text files for all the phase tensor parameters

plot (*self*, *show=True*)
plots the phase tensor pseudo section. See class doc string for more details.

redraw_plot (*self*)
use this function if you updated some attributes and want to re-plot.

Example

```
>>> # change ellipse size and color map to be segmented for skew
>>> pt1.ellipse_size = 5
>>> pt1.ellipse_colorby = 'beta_seg'
>>> pt1.ellipse_cmap = 'mt_seg_bl2wh2rd'
>>> pt1.ellipse_range = (-9, 9, 3)
>>> pt1.redraw_plot()
```

rot_z
rotation angle(s)

save_figure (*self*, *save_fn*, *file_format='png'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot='y'*)
save_plot will save the figure to save_fn.

save_figure2 (*self*, *save_fn*, *file_format='jpg'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot='y'*)
save_plot will save the figure to save_fn.

update_plot (*self*)
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
>>> pt1.ax.grid(True, which='major', color=(.5,.5,.5))
>>> pt1.update_plot()
```

writeTextFiles (*self*, *save_path=None*, *ptol=0.1*)
This will write text files for all the phase tensor parameters

4.4 Module MTPlot

Provides

1. Different plotting options to represent the MT response.
2. Ability to create text files of the plots for further analysis
3. Class object that contains all the important information for an MT station.

Functions	Description
plot_mt_response	plots resistivity and phase for a single station Options include tipper, strike and skew.
plot_multiple_mt_responses	plots multiple stations at once with options of plotting in single figure, all in one figure as subplots or all in one plot for direct comparison.
plot_pt	plots the phase tensor ellipses and parameters in one plot including strike angle, minimum and maximum phase, skew angle and ellipticity
plot_pt_pseudosection	plots a pseudo section of phase tensor ellipses assuming the stations are along a profile line. Options to plot induction arrows.
plot_mt_map	plots phase tensor ellipses in map view for a single frequency. Options to plot induction arrows.
plot_strike	plots strike angle estimated from the invariants of the impedance tensor defined by Weaver et al. [2000,2003], strike angle from the phase tensor and option to plot strike estimated from the induction arrows.
plot_residual_pt_maps	plots the residual phase tensor between two surveys in map view.
plot_residual_pt_ps	plots the residual phase tensor between two surveys as a pseudo section.

All plot function return plot classes where the important properties are made attributes which can be manipulated by the user. All classes have been written with the basic input being edi files. This was assumed to be the standard MT response file, but turns out to be not as widely used as thought. So the inputs can be other arrays and class objects (see MTplot doc string for details). If you have a data file format you can create a class using the objects in mtpy.core to create an input, otherwise contact us and we can try to build something.

A typical use might be loading in all the .edi files in and plotting them in different modes, like apparent resistivity and phase, phase tensor pseudo section and strike angle.

Example

```
>>> import mtpy.imaging.mtplot as mtplot
>>> import os
>>> import matplotlib.pyplot as plt
>>> edipath = r"/home/MT/EDIfiles"
>>> #--> create a list of full paths to the edi files
>>> edilst = [os.path.join(edipath,edi) for edi in os.listdir(edipath)
>>> ...         if edi.find('.edi')>0]
>>> #--> plot apparent resisitivity, phase and induction arrows
>>> rpm = mtplot.plot_multiple_mt_responses(fn_lst=edilst, plot_style='1
↪',
>>> ...                                      plot_tipper='yr')
>>> #--> close all the plots after done looking at them
>>> plt.close('all')
>>> #--> plot phase tensor pseudo section with induction arrows
>>> pts = mtplot.plot_pt_pseudosection(fn_lst=edilst,
>>> ...                                 plot_tipper='yr')
>>> #--> write out the phase tensor parameter values to files
>>> pts.export_pt_params_to_file()
>>> #--> change coloring scheme to color by skew and a segmented colormap
>>> pts.ellipse_colorby = 'skew_seg'
```

(continues on next page)

(continued from previous page)

```
>>> pts.ellipse_cmap = 'mt_seg_bl2wh2rd'
>>> pts.ellipse_range = (-9, 9, 3)
>>> pts.redraw_plot()
```

Authors Lars Krieger, Jared Peacock, and Kent Invariarty

Version 0.0.1 of 2013

`mtpy.imaging.mtplot.`**`plot_mt_response`**(***kwargs*)

Plots Resistivity and phase for the different modes of the MT response. At the moment is 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.

`mtpy.imaging.mtplot.`**`plot_multiple_mt_responses`**(***kwargs*)

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.

expecting only one type of input –> can be: fn_list : list of filenames to plot

z_object_list : list of mtpy.core.z.Z objects

res_object_list : list of mtpy.imaging.mtplot.ResPhase objects

tipper_object_list : list of mtpy.imaging.mtplot.Tipper objects

mt_object_list : list of mtpy.imaging.mtplot.MTplot objects

`mtpy.imaging.mtplot.`**`plot_pt`**(***kwargs*)

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.

`mtpy.imaging.mtplot.`**`plot_pt_map`**(***kwargs*)

Plots phase tensor ellipses in map view from a list of edi files

`mtpy.imaging.mtplot.`**`plot_pt_pseudosection`**(***kwargs*)

PlotPhaseTensorPseudoSection will plot the phase tensor ellipses in a pseudo section format

`mtpy.imaging.mtplot.`**`plot_residual_pt_maps`**(*fn_list1*, *fn_list2*, ***kwargs*)

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.

`mtpy.imaging.mtplot.`**`plot_residual_pt_ps`**(*fn_list1*, *fn_list2*, ***kwargs*)

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.

`mtpy.imaging.mtplot.`**`plot_resphase_pseudosection`**(***kwargs*)

plot a resistivity and phase pseudo section for different components

Need to input one of the following lists:

`mtpy.imaging.mtplot.`**`plot_station_locations`**(***kwargs*)

plot station locations in map view.

Need to input one of the following lists:

`mtpy.imaging.mtplot.`**`plot_strike`**(***kwargs*)

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.

Attributes

-axhinv matplotlib.axes instance for invariant strike

-axhpt	matplotlib.axes instance for phase tensor strike
-axhtip	matplotlib.axes instance for tipper strike
-barinv	matplotlib.axes.bar instance for invariant strike
-barpt	matplotlib.axes.bar instance for pt strike

-bartr	matplotlib.axes.bar instance for tipper strike
-bin_width	width of histogram bins in degrees
-fig	matplotlib.figure instance of plot
-fig_dpi	dots-per-inch resolution of figure
-fig_num	number of figure being plotted
-fig_size	size of figure in inches
-fold	boolean to fold angles to range from [0,180] or [0,360]
-font_size	font size of axes tick labels
-mt_list	list of mtplot.MTplot instances containing all the important information for each station
-period_tolerance	tolerance to look for periods being plotted
-plot_range	range of periods to plot
-plot_tipper	string to tell program to plot induction arrows
-plot_type	string to tell program how to plot strike angles
-plot_yn	plot strike on instance creation
-pt_error_floor	error floor to plot phase tensor strike, anything above this error will not be plotted
-text_pad	padding between text and rose diagram
-text_size	font size of text labeling the mode of the histogram
-title_dict	title dictionary

Methods

-plot plots the pseudo section	**-redraw_plot** on call redraws the plot from scratch -save_figure saves figure to a file of given format -update_plot updates the plot while still active -export_pt_params_to_file writes parameters of the phase tensor and tipper to text files.

Plots the resistivity and phase for different modes and components

Created on Thu May 30 16:54:08 2013

@author: jpeacock-pr

class mtpy.imaging.plotresponse.**PlotResponse**(***kwargs*)

Plots Resistivity and phase for the different modes of the MT response. At the moment is 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.

Attributes

plot_pt string to plot phase tensor ellipses

plot_skew string to plot skew

plot_strike string to plot strike

plot_tipper string to plot tipper

Methods

plot(self)	plotResPhase(filename,fig_num) will plot the apparent resistivity and phase for a single station.
redraw_plot(self)	use this function if you updated some attributes and want to re-plot.
save_plot(self, save_fn[, file_format, ...])	save_plot will save the figure to save_fn.
update_plot(self)	update any parameters that where changed using the built-in draw from canvas.

plot(*self*)

plotResPhase(filename,fig_num) will plot the apparent resistivity and phase for a single station.

plot_pt

string to plot phase tensor ellipses

plot_skew

string to plot skew

plot_strike

string to plot strike

plot_tipper
string to plot tipper

redraw_plot (*self*)
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.imaging.mtplottools as mtplot
>>> p1 = mtplot.PlotResPhase(r'/home/MT/mt01.edi')
>>> p1.xy_color = (.5,.5,.9)
>>> p1.xy_marker = '*'
>>> p1.redraw_plot()
```

save_plot (*self, save_fn, file_format='pdf', orientation='portrait', fig_dpi=None, close_plot='y'*)
save_plot will save the figure to save_fn.

update_plot (*self*)
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.imaging.mtplottools as mtplot
>>> p1 = mtplot.PlotResPhase(r'/home/MT/mt01.edi')
>>> [ax.grid(True, which='major') for ax in [p1.axr,p1.axp]]
>>> p1.update_plot()
```

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.plotnresponses.**PlotMultipleResponses** (***kwargs*)
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.

expecting only one type of input –> can be: fn_list : list of filenames to plot

z_object_list : list of mtpy.core.z.Z objects

res_object_list : list of mtpy.imaging.mtplot.ResPhase objects

tipper_object_list : list of mtpy.imaging.mtplot.Tipper objects

mt_object_list : list of mtpy.imaging.mtplot.MTplot objects

Attributes

plot_pt string to plot phase tensor ellipses

plot_skew string to plot skew

plot_strike string to plot strike

plot_tipper string to plot tipper

rot_z rotation angle(s)

Methods

plot(self[, show])	plot the apparent resistivity and phase
redraw_plot(self)	use this function if you updated some attributes and want to re-plot.
update_plot(self)	update any parameters that where changed using the built-in draw from canvas.

plot (*self*, *show=True*)
plot the apparent resistivity and phase

plot_pt
string to plot phase tensor ellipses

plot_skew
string to plot skew

plot_strike
string to plot strike

plot_tipper
string to plot tipper

redraw_plot (*self*)
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.imaging.mtplottools as mtplot
>>> p1 = mtplot.PlotResPhase(r'/home/MT/mt01.edi')
>>> p1.xy_color = (.5,.5,.9)
>>> p1.xy_marker = '*'
>>> p1.redraw_plot()
```

rot_z
rotation angle(s)

update_plot (*self*)
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.imaging.mtplottools as mtplot
>>> p1 = mtplot.PlotResPhase(r'/home/MT/mt01.edi')
>>> [ax.grid(True, which='major') for ax in [p1.axr,p1.axp]]
>>> p1.update_plot()
```

Created on Thu May 30 18:28:24 2013

@author: jpeacock-pr

class mtpy.imaging.plotstrike.**PlotStrike** (***kwargs*)
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.

Attributes

-axhinv matplotlib.axes instance for invariant strike

-axhpt	matplotlib.axes instance for phase tensor strike
-axhtip	matplotlib.axes instance for tipper strike
-barinv	matplotlib.axes.bar instance for invariant strike
-barpt	matplotlib.axes.bar instance for pt strike
-bartr	matplotlib.axes.bar instance for tipper strike
-bin_width	width of histogram bins in degrees
-fig	matplotlib.figure instance of plot
-fig_dpi	dots-per-inch resolution of figure
-fig_num	number of figure being plotted
-fig_size	size of figure in inches
-fold	boolean to fold angles to range from [0,180] or [0,360]
-font_size	font size of axes tick labels
-mt_list	list of mtplot.MTplot instances containing all the important information for each station
-period_tolerance	tolerance to look for periods being plotted
-plot_range	range of periods to plot
-plot_tipper	string to tell program to plot induction arrows
-plot_type	string to tell program how to plot strike angles
-plot_yn	plot strike on instance creation
-pt_error_floor	error floor to plot phase tensor strike, anything above this error will not be plotted
-text_pad	padding between text and rose diagram
-text_size	font size of text labeling the mode of the histogram
-title_dict	title dictionary

Methods

-plot plots the pseudo section	**-redraw_plot** on call redraws the plot from scratch -save_figure saves figure to a file of given format -update_plot updates the plot while still active -export_pt_params_to_file writes parameters of the phase tensor and tipper to text files.

redraw_plot (*self*)

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.imaging.mtplottools as mtplot
>>> p1 = mtplot.PlotResPhase(r'/home/MT/mt01.edi')
>>> p1.xy_color = (.5,.5,.9)
>>> p1.xy_marker = '*'
>>> p1.redraw_plot()
```

rot_z

rotation angle(s)

save_plot (*self, save_fn, file_format='pdf', orientation='portrait', fig_dpi=None, close_plot='y'*)

save_plot will save the figure to save_fn.

Examples

Example

```
>>> # to save plot as jpg
>>> import mtpy.imaging.mtplottools as mtplot
>>> p1 = mtplot.PlotPhaseTensorMaps(edilist,freqspot=10)
>>> p1.save_plot(r'/home/MT', file_format='jpg')
```

'Figure saved to /home/MT/PTMaps/PTmap_phimin_10Hz.jpg'

update_plot (*self*)

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.imaging.mtplottools as mtplot
>>> p1 = mtplot.PlotResPhase(r'/home/MT/mt01.edi')
>>> [ax.grid(True, which='major') for ax in [p1.axr,p1.axp]]
>>> p1.update_plot()
```

writeTextFiles(*self*, *save_path=None*)

Saves the strike information as a text file.

Created on Thu May 30 18:28:24 2013

@author: jpeacock-pr

class mtpy.imaging.plotstrike2d.**PlotStrike2D**(***kwargs*)

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.

Attributes

rot_z rotation angle(s)

Methods

redraw_plot(self)	use this function if you updated some attributes and want to re-plot.
save_plot(self, save_fn[, file_format, ...])	save_plot will save the figure to save_fn.
update_plot(self)	update any parameters that where changed using the built-in draw from canvas.
writeTextFiles(self[, save_path])	Saves the strike information as a text file.

plot	

redraw_plot(*self*)

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.imaging.mtplottools as mtplot
>>> p1 = mtplot.PlotResPhase(r'/home/MT/mt01.edi')
>>> p1.xy_color = (.5,.5,.9)
>>> p1.xy_marker = '*'
>>> p1.redraw_plot()
```

rot_z

rotation angle(s)

save_plot (*self*, *save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot='y'*)
save_plot will save the figure to save_fn.

update_plot (*self*)
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.imaging.mtplottools as mtplot
>>> p1 = mtplot.PlotResPhase(r'/home/MT/mt01.edi')
>>> [ax.grid(True, which='major') for ax in [p1.axr,p1.axp]]
>>> p1.update_plot()
```

writeTextFiles (*self*, *save_path=None*)
Saves the strike information as a text file.

4.5 Plot MT Response

4.5.1 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*)
Plots Resistivity and phase for the different modes of the MT response. At the moment is 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.

Attributes

period plot period

Methods

plot(self[, show])	plotResPhase(filename,fig_num) will plot the apparent resistivity and phase for a single station.
redraw_plot(self)	use this function if you updated some attributes and want to re-plot.
save_plot(self, save_fn[, file_format, . . .])	save_plot will save the figure to save_fn.
update_plot(self)	update any parameters that where changed using the built-in draw from canvas.

period
plot period

plot (*self*, *show=True*)
plotResPhase(filename,fig_num) will plot the apparent resistivity and phase for a single station.

redraw_plot (*self*)
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.imaging.mtplottools as mtplot
>>> p1 = mtplot.PlotResPhase(r'/home/MT/mt01.edi')
>>> p1.xy_color = (.5,.5,.9)
>>> p1.xy_marker = '*'
>>> p1.redraw_plot()
```

save_plot (*self*, *save_fn*, *file_format='pdf'*, *orientation='portrait'*, *fig_dpi=None*, *close_plot='y'*)
save_plot will save the figure to save_fn.

update_plot (*self*)
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.imaging.mtplottools as mtplot
>>> p1 = mtplot.PlotResPhase(r'/home/MT/mt01.edi')
>>> [ax.grid(True, which='major') for ax in [p1.axr,p1.axp]]
>>> p1.update_plot()
```

4.6 Visualization of Models

class mtpy.imaging.plot_depth_slice.**PlotDepthSlice** (*model_fn=None*, *data_fn=None*, ***kwargs*)
Plots depth slices of resistivity model (file.rho)

Example

```
>>> import mtpy.modeling.ws3dinv as ws
>>> mfn = r"/home/MT/ws3dinv/Inv1/Test_model.00"
>>> sfn = r"/home/MT/ws3dinv/Inv1/WSStationLocations.txt"
```

(continues on next page)

(continued from previous page)

```
>>> # plot just first layer to check the formatting
>>> pds = ws.PlotDepthSlice(model_fn=mfn, station_fn=sfn,
>>> ...                         depth_index=0, save_plots='n')
>>> #move color bar up
>>> pds.cb_location
>>> (0.64500000000000002, 0.14999999999999997, 0.3, 0.025)
>>> pds.cb_location = (.645, .175, .3, .025)
>>> pds.redraw_plot()
>>> #looks good now plot all depth slices and save them to a folder
>>> pds.save_path = r"/home/MT/ws3dinv/Inv1/DepthSlices"
>>> pds.depth_index = None
>>> pds.save_plots = 'y'
>>> pds.redraw_plot()
```

Attributes	Description
cb_location	location of color bar (x, y, width, height) *default* is None, automatically locates
cb_orientation	['vertical' | 'horizontal'] *default* is horizontal
cb_pad	padding between axes and colorbar *default* is None
cb_shrink	percentage to shrink colorbar by *default* is None
climits	(min, max) of resistivity color on log scale *default* is (0, 4)
cmap	name of color map *default* is 'jet_r'
data_fn	full path to data file
depth_index	integer value of depth slice index, shallowest layer is 0
dscale	scaling parameter depending on map_scale
ew_limits	(min, max) plot limits in e-w direction in map_scale units. *default* is None, sets viewing area to the station area
fig_aspect	aspect ratio of plot. *default* is 1
fig_dpi	resolution of figure in dots-per-inch. *default* is 300
fig_list	list of matplotlib.figure instances for each depth slice
fig_size	[width, height] in inches of figure size *default* is [6, 6]
font_size	size of ticklabel font in points, labels are font_size+2. *default* is 7
grid_east	relative location of grid nodes in e-w direction in map_scale units
grid_north	relative location of grid nodes in n-s direction in map_scale units
grid_z	relative location of grid nodes in z direction in map_scale units
initial_fn	full path to initial file
map_scale	['km' | 'm'] distance units of map. *default* is km
mesh_east	np.meshgrid(grid_east, grid_north, indexing='ij')
mesh_north	np.meshgrid(grid_east, grid_north, indexing='ij')
model_fn	full path to model file
nodes_east	relative distance betwen nodes in e-w direction in map_scale units
nodes_north	relative distance betwen nodes in n-s direction in map_scale units
nodes_z	relative distance betwen nodes in z direction in map_scale units
ns_limits	(min, max) plot limits in n-s direction in map_scale units. *default* is None, sets viewing area to the station area
plot_grid	['y' | 'n'] 'y' to plot mesh grid lines. *default* is 'n'
plot_yn	['y' | 'n'] 'y' to plot on instantiation
res_model	np.ndarray(n_north, n_east, n_vertical) of model resistivity values in linear scale
save_path	path to save figures to
save_plots	['y' | 'n'] 'y' to save depth slices to save_path
station_east	location of stations in east direction in map_scale units
station_fn	full path to station locations file
station_names	station names
station_north	location of station in north direction in map_scale units

Continued on next page

Table 11 – continued from previous page

Attributes	Description
subplot_bottom	distance between axes and bottom of figure window
subplot_left	distance between axes and left of figure window
subplot_right	distance between axes and right of figure window
subplot_top	distance between axes and top of figure window
title	titiel of plot *default* is depth of slice
xminorticks	location of xminorticks
yminorticks	location of yminorticks

Methods

`plot`(self[, ind])	plot the depth slice ind-th
`redraw_plot`(self)	redraw plot if parameters were changed use this function if you updated some attributes and want to re-plot.

plot (*self*, *ind=1*)
plot the depth slice ind-th

redraw_plot (*self*)
redraw plot if parameters were changed use this function if you updated some attributes and want to re-plot.

CHAPTER 5

Package utils

5.1 Shapefile Creator

Description: Create shape files for Phase Tensor Ellipses, Tipper Real/Imag. export the phase tensor map and tippers into jpeg/png images

CreationDate: 2017-03-06 Developer: fei.zhang@ga.gov.au

Revision History: LastUpdate: 10/11/2017 FZ fix bugs after the big merge LastUpdate: 20/11/2017 change from freq to period filenames, allow to specify a period LastUpdate: 30/10/2018 combine ellipses and tippers together, refactorings

class mtpy.utils.shapefiles_creator.**ShapeFilesCreator**(*edifile_list*, *outdir*, *orig_crs={'init': 'epsg:4283'}*)

Extend the EdiCollection parent class, create phase tensor and tipper shapefiles for a list of edifiles

Parameters

- **edifile_list** – [path2edi,. . .]
- **outdir** – path2output dir, where the shp file will be written.
- **= {'init'** (*orig_crs*) – 'epsg:4283' } # GDA94

Methods

create_measurement_csv(self, dest_dir[, . . .])	create csv file from the data of EDI files: IMPEDANCE, APPARENT RESISTIVITIES AND PHASES see also utils/shapefiles_creator.py
create_mt_station_gdf(self[, outshpfile])	create station location geopandas dataframe, and output to shape file

Continued on next page

Table 1 – continued from previous page

`create_phase_tensor_csv`(self, dest_dir[, ...])	create phase tensor ellipse and tipper properties.
`create_phase_tensor_csv_with_image`(*args, ...)	Using PlotPhaseTensorMaps class to generate csv file of phase tensor attributes, etc.
`create_phase_tensor_shp`(self, period[, ...])	create phase tensor ellipses shape file correspond to a MT period :return: (geopdf_obj, path_to_shapefile)
`create_tipper_imag_shp`(self, period[, ...])	create imagery tipper lines shapefile from a csv file The shapefile consists of lines without arrow.
`create_tipper_real_shp`(self, period[, ...])	create real tipper lines shapefile from a csv file The shapefile consists of lines without arrow.
`display_on_basemap`(self)	display MT stations which are in stored in geopandas dataframe in a base map.
`display_on_image`(self)	display/overlay the MT properties on a background geo-referenced map image
`export_edi_files`(self, dest_dir[, ...])	export edi files.
`get_bounding_box`(self[, epsgcode])	compute bounding box
`get_min_max_distance`(self)	get the min and max distance between all possible pairs of stations.
`get_period_occurance`(self, aper)	For a given aperiod, compute its occurance frequencies among the stations/edi :param aper: a float value of the period :return:
`get_periods_by_stats`(self[, percentage])	check the presence of each period in all edi files, keep a list of periods which are at least percentage present :return: a list of periods which are present in at least percentage edi files
`get_phase_tensor_tippers`(self, period[, ...])	For a given MT period (s) value, compute the phase tensor and tippers etc.
`get_station_utmzones_stats`(self)	A simple method to find what UTM zones these (edi files) MT stations belong to are they in a single UTM zone, which corresponds to a unique EPSG code? or do they belong to multiple UTM zones?
`get_stations_distances_stats`(self)	get the min max statistics of the distances between stations.
`plot_stations`(self[, savefile, showfig])	Visualise the geopandas df of MT stations
`select_periods`(self[, show, period_list, ...])	Use edi_collection to analyse the whole set of EDI files
`show_obj`(self[, dest_dir])	test call object's methods and show it's properties

create_phase_tensor_shp(*self*, *period*, *ellipsize=None*, *target_epsg_code=4283*, *export_fig=False*)

create phase tensor ellipses shape file correspond to a MT period :return: (geopdf_obj, path_to_shapefile)

create_tipper_imag_shp(*self*, *period*, *line_length=None*, *target_epsg_code=4283*, *export_fig=False*)

create imagery tipper lines shapefile from a csv file The shapefile consists of lines without arrow. User can use GIS software such as ArcGIS to display and add an arrow at each line's end line_length is how long will be the line, auto-calculatable :return:(geopdf_obj, path_to_shapefile)

create_tipper_real_shp(*self*, *period*, *line_length=None*, *target_epsg_code=4283*, *export_fig=False*)

create real tipper lines shapefile from a csv file The shapefile consists of lines without arrow. User can use GIS software such as ArcGIS to display and add an arrow at each line's end line_length is how long will be the line, auto-calculatable `mtpy.utils.shapefiles_creator.`**`create_ellipse_shp_from_csv`**(*csvfile*, *esize=0.03*, *target_epsg_code=4283*)

create phase tensor ellipse geometry from a csv file. This function needs csv file as its input. :param csvfile: a csvfile with full path :param esize: ellipse size, defaut 0.03 is about 3KM in the max ellipse rad :return: a geopandas dataframe

`mtpy.utils.shapefiles_creator.`**`create_tipper_imag_shp_from_csv`**(*csvfile*, *line_length=0.03*, *target_epsg_code=4283*)

create imagery tipper lines shape from a csv file. this function needs csv file as input. The shape is a line without arrow. Must use a GIS software such as ArcGIS to display and add an arrow at each line's end line_length=4 how long will be the line (arrow) return: a geopandas dataframe object for further processing.

`mtpy.utils.shapefiles_creator.`**`create_tipper_real_shp_from_csv`**(*csvfile*, *line_length=0.03*, *target_epsg_code=4283*)

create tipper lines shape from a csv file. This function needs csv file as its input. The shape is a line without arrow. Must use a GIS software such as ArcGIS to display and add an arrow at each line's end line_length=4 how long will be the line (arrow) return: a geopandas dataframe object for further processing.

`mtpy.utils.shapefiles_creator.`**`export_geopdf_to_image`**(*geopdf*, *bbox*, *jpg_file_name*, *target_epsg_code=None*, *colorby=None*, *colormap=None*, *showfig=False*)

Export a geopandas dataframe to a jpe_file, with optionally a new epsg projection. :param geopdf: a geopandas dataframe :param bbox: This param ensures that we can set a consistent display area defined by a dict with 4 keys

[MinLat, MinLon, MaxLat, MaxLon], cover all ground stations, not just this period-dependent geopdf

Parameters

- **`jpg_file_name`** (*`output`*) – path2jpeg
- **`target_epsg_code`** – 4326 etc
- **`showfig`** – If True, then display fig on screen.

Returns

`mtpy.utils.shapefiles_creator.`**`plot_phase_tensor_ellipses_and_tippers`**(*edi_dir*, *outfile=None*, *iperiod=0*)

plot phase tensor ellipses and tipers into one figure. :param edi_dir: edi directory :param outfile: save figure to output file :param iperiod: the index of periods :return: saved figure file

`mtpy.utils.shapefiles_creator.`**`process_csv_folder`**(*csv_folder*, *bbox_dict*, *target_epsg_code=4283*)

process all *.csv files in a dir, ude target_epsg_code=4283 GDA94 as default. This function uses csv-files folder as its input. :param csv_folder: :return:

Create shape files for phase tensor ellipses. https://pcjericks.github.io/py-gdalogr-cookbook/vector_layers.html#create-a-new-shapefile-and-add-data

Created on Sun Apr 13 12:32:16 2014

@author: jrpeacock

class mtpy.utils.shapefiles.**PTShapeFile**(*edi_list=None*, *proj='WGS84'*, *esize=0.03*, ***kwargs*)

write shape file for GIS plotting programs

key words/attributes	Description
edi_list	list of edi files, full paths
ellipse_size	size of normalized ellipse in map scale *default* is .01
mt_obj_list	list of mt.MT objects *default* is None, filled if edi_list is given
plot_period	list or value of period to convert to shape file *default* is None, which will write a file for every period in the edi files
ptol	tolerance to look for given periods *default* is .05
pt_dict	dictionary with keys of plot_period. Each dictionary key is a structured array containing the important information for the phase tensor.
projection	projection of coordinates see EPSG for all options *default* is WSG84 in lat and lon
save_path	path to save files to *default* is current working directory.

Methods	Description
_get_plot_period	get a list of all frequencies possible from input files
_get_pt_array	get phase tensors from input files and put the information into a structured array
write_shape_files	write shape files based on attributes of class

- This will project the data into UTM WSG84

  Example :: >>> edipath = r"/home/edi_files_rotated_to_geographic_north" >>> edilist = [os.path.join(edipath, edi) for edi in os.listdir(edipath) if edi.find('.edi')>0] >>> pts = PTShapeFile(edilist, save_path=r"/home/gis") >>> pts.write_shape_files()

- To project into another datum, set the projection attribute

  Example :: >>> pts = PTShapeFile(edilist, save_path=r"/home/gis") >>> pts.projection = 'NAD27' >>> pts.write_shape_files()

  Attributes

  rotation_angle rotation angle of Z and Tipper

Methods

write_data_pt_shape_files_modem(self, ...[, ...])	write pt files from a modem data file.
write_residual_pt_shape_files_modem(self, ...)	write residual pt shape files from ModEM output
write_resp_pt_shape_files_modem(self, ...[, ...])	write pt files from a modem response file where ellipses are normalized by the data file.
write_shape_files(self[, periods])	write shape file from given attributes https://pcjericks.github.io/py-gdalogr-cookbook/vector_layers.html #create-a-new-shapefile-and-add-data

rotation_angle
rotation angle of Z and Tipper

write_data_pt_shape_files_modem(*self*, *modem_data_fn*, *rotation_angle=0.0*)
write pt files from a modem data file.

write_residual_pt_shape_files_modem(*self*, *modem_data_fn*, *modem_resp_fn*, *rotation_angle=0.0*, *normalize='1'*)
write residual pt shape files from ModEM output

normalize ['1' | 'all']

- **'1' to normalize the ellipse by itself, all ellipses are** normalized to phimax, thus one axis is of length 1*ellipse_size
- 'all' to normalize each period by the largest phimax

write_resp_pt_shape_files_modem(*self*, *modem_data_fn*, *modem_resp_fn*, *rotation_angle=0.0*)
write pt files from a modem response file where ellipses are normalized by the data file.

write_shape_files(*self*, *periods=None*)
write shape file from given attributes https://pcjericks.github.io/py-gdalogr-cookbook/vector_layers.html #create-a-new-shapefile-and-add-data

class mtpy.utils.shapefiles.**TipperShapeFile**(*edi_list=None*, ***kwargs*)
write shape file for GIS plotting programs.

currently only writes the real induction vectors.

key words/attributes	Description
arrow_direction	[1 | -1] 1 for Weise convention –> point toward conductors. *default* is 1 (-1 is not supported yet)
arrow_head_height	height of arrow head in map units *default* is .002
arrow_head_width	width of arrow head in map units *default* is .001
arrow_lw	width of arrow in map units *default* is .0005
arrow_size	size of normalized arrow length in map units *default* is .01
edi_list	list of edi files, full paths
mt_obj_list	list of mt.MT objects *default* is None, filled if edi_list is given
plot_period	list or value of period to convert to shape file *default* is None, which will write a file for every period in the edi files
ptol	tolerance to look for given periods *default* is .05
pt_dict	dictionary with keys of plot_period. Each dictionary key is a structured array containing the important information for the phase tensor.
projection	projection of coordinates see EPSG for all options *default* is WSG84
save_path	path to save files to *default* is current working directory.

Methods	Description
_get_plot_period	get a list of all possible frequencies from data
_get_tip_array	get Tipper information from data and put into a structured array for easy manipulation
write_real_shape_files	write real induction arrow shape files
write_imag_shape_files	write imaginary induction arrow shape files

Example :: >>> edipath = r"/home/edi_files_rotated_to_geographic_north" >>> edilist = [os.path.join(edipath, edi) for edi in os.listdir(edipath) if edi.find('.edi')>0] >>> tipshp = TipperShapeFile(edilist, save_path=r"/home/gis") >>> tipshp.arrow_head_height = .005 >>> tipshp.arrow_lw = .0001 >>> tipshp.arrow_size = .05 >>> tipshp.write_shape_files()

Attributes

rotation_angle rotation angle of Z and Tipper

Methods

write_imag_shape_files(self)	write shape file from given attributes
write_real_shape_files(self)	write shape file from given attributes
write_tip_shape_files_modem(self, modem_data_fn)	write tip files from a modem data file.
write_tip_shape_files_modem_residual(self, ...)	write residual tipper files for modem

rotation_angle
rotation angle of Z and Tipper

write_imag_shape_files(*self*)
write shape file from given attributes

write_real_shape_files(*self*)
write shape file from given attributes

write_tip_shape_files_modem(*self, modem_data_fn, rotation_angle=0.0*)
write tip files from a modem data file.

write_tip_shape_files_modem_residual(*self, modem_data_fn, modem_resp_fn, rotation_angle*)
write residual tipper files for modem

mtpy.utils.shapefiles.**create_phase_tensor_shpfiles**(*edi_dir, save_dir, proj='WGS84', ellipse_size=1000, every_site=1, period_list=None*)
generate shape file for a folder of edi files, and save the shape files a dir. :param edi_dir: :param save_dir: :param proj: defult is WGS84-UTM, with ellipse_size=1000 meters :param ellipse_size: the size of ellipse: 100-5000, try them out to suit your needs :param every_site: by default every MT station will be output, but user can sample down with 2, 3,.. :return:

mtpy.utils.shapefiles.**create_tipper_shpfiles**(*edipath, save_dir*)
Create Tipper (induction arrows real and imaginary) shape files :param edipath: :param save_dir: :return:

mtpy.utils.shapefiles.**modem_to_shapefiles**(*mfndat, save_dir*)
create shape file representaiotn for ModEM model :param mfndat: path2Modular_NLCG_110.dat :param save_dir: path2outshp :return:

mtpy.utils.shapefiles.**reproject_layer**(*in_shape_file, out_shape_file=None, out_proj='WGS84'*)
reproject coordinates into a different coordinate system

5.2 GIS Tools

Created on Fri Apr 14 14:47:48 2017

@author: jrpeacock

exception mtpy.utils.gis_tools.**GIS_ERROR**

mtpy.utils.gis_tools.**assert_elevation_value**(*elevation*)
make sure elevation is a floating point number

mtpy.utils.gis_tools.**assert_lat_value**(*latitude*)
make sure latitude is in decimal degrees

mtpy.utils.gis_tools.**assert_lon_value**(*longitude*)
make sure longitude is in decimal degrees

mtpy.utils.gis_tools.**convert_position_float2str**(*position*)
convert position float to a string in the format of DD:MM:SS

Returns

****position_str**** [string] 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

Returns

****position**** [float] latitude or longitude in decimal degrees

mtpy.utils.gis_tools.**epsg_project**(*x*, *y*, *epsg_from*, *epsg_to*)
project some xy points using the pyproj modules

mtpy.utils.gis_tools.**get_epsg**(*latitude*, *longitude*)
get epsg code for the utm projection (WGS84 datum) of a given latitude and longitude pair

mtpy.utils.gis_tools.**get_utm_string_from_sr**(**args*, ***kwargs*)
return utm zone string from spatial reference instance

mtpy.utils.gis_tools.**get_utm_zone**(*latitude*, *longitude*)
Get utm zone from a given latitude and longitude

mtpy.utils.gis_tools.**ll_to_utm**(**args*, ***kwargs*)
converts lat/long to UTM coords. Equations from USGS Bulletin 1532 East Longitudes are positive, West longitudes are negative. North latitudes are positive, South latitudes are negative Lat and Long are in decimal degrees Written by Chuck Gantz- chuck.gantz@globalstar.com

Outputs: UTMzone, easting, northing

mtpy.utils.gis_tools.**project_point_ll2utm**(*lat*, *lon*, *datum='WGS84'*, *utm_zone=None*, *epsg=None*)
Project a point that is in Lat, Lon (will be converted to decimal degrees) into UTM coordinates.

mtpy.utils.gis_tools.**project_point_utm2ll**(*easting*, *northing*, *utm_zone*, *datum='WGS84'*, *epsg=None*)
Project a point that is in Lat, Lon (will be converted to decimal degrees) into UTM coordinates.

mtpy.utils.gis_tools.**project_points_ll2utm**(*lat*, *lon*, *datum='WGS84'*, *utm_zone=None*, *epsg=None*)
Project a list of points that is in Lat, Lon (will be converted to decimal degrees) into UTM coordinates.

mtpy.utils.gis_tools.**utm_to_ll**(**args*, ***kwargs*)
converts UTM coords to lat/long. Equations from USGS Bulletin 1532 East Longitudes are positive, West longitudes are negative. North latitudes are positive, South latitudes are negative Lat and Long are in decimal degrees. Written by Chuck Gantz- chuck.gantz@globalstar.com Converted to Python by Russ Nelson <nelson@crynwr.com>

Outputs: Lat,Lon

`mtpy.utils.gis_tools.`**`utm_wgs84_conv`**(*lat*, *lon*)
Bidirectional UTM-WGS84 converter https://github.com/Turbo87/utm/blob/master/utm/conversion.py :param lat: :param lon: :return: tuple(e, n, zone, lett)

`mtpy.utils.gis_tools.`**`utm_zone_to_epsg`**(*zone_number*, *is_northern*)
get epsg code (WGS84 datum) for a given utm zone

5.3 Other Tools

`class` `mtpy.utils.decorator.`**`deprecated`**(*reason*)

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

Methods

__call__	

Created on Wed Oct 25 09:35:31 2017

@author: Alison Kirkby

functions to assist with mesh generation

`mtpy.utils.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.utils.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.

Returns

****padding**** [np.ndarray] array of padding cells for one side

`mtpy.utils.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.utils.mesh_tools.`**`get_padding_from_stretch`**(*cell_width*, *pad_stretch*, *num_cells*)
get padding cells using pad stretch factor

`mtpy.utils.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.utils.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.utils.mesh_tools.`**`interpolate_elevation_to_grid`**(*grid_east*, *grid_north*, *epsg=None*, *utm_zone=None*, *surfacefile=None*, *surface=None*, *method='linear'*, *fast=True*)

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.utils.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.utils.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

A more Pythonic way of logging: Define a class MtPyLog to wrap the python logging module; Use a (optional) configuration file (yaml, ini, json) to configure the logging, It will return a logger object with the user-provided config setting. see also: http://www.cdotson.com/2015/11/python-logging-best-practices/

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Bibliography

[Rf5ecdd4b8de8-1] Changes these values to change what is written to edi file

[R5ea4377773dd-1] Each channel with have its own define measurement and depending on whether it is an E or H channel the metadata will be different. the #### correspond to the channel number.

[R5ea4377773dd-2] Internally everything is converted to decimal degrees. Output is written as HH:MM:SS.ss so Winglink can read them in.

[R5ea4377773dd-3] If you want to change what metadata is written into the .edi file change the items in _header_keys. Default attributes are:

- maxchan
- maxrun
- maxmeas
- reflat
- reflon
- refelev
- reftype
- units

[R60960842fb28-1] Internally everything is converted to decimal degrees. Output is written as HH:MM:SS.ss so Winglink can read them in.

[R60960842fb28-2] If you want to change what metadata is written into the .edi file change the items in _header_keys. Default attributes are:

- acqby
- acqdate
- coordinate_system
- dataid
- declination
- elev

- fileby
- lat
- loc
- lon
- filedate
- empty
- progdate
- progvers

Python Module Index

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W

X

Z