MT - How to ...
–
Help for MT data processing, analysing and modeling

Kent Inverarity, Jared Peacock, Lars Krieger*

October 2, 2014

*University of Adelaide – School of Earth and Environmental Sciences – Mawson Building – Adelaide

Introduction

This is a collection of guides (*How to ...*) for various steps in the processing of MT data. Most steps are based on the MTpy package.

October 2, 2014, LK

Contents

1 Structure

1.1

2 In the field

3 Raw-data-/time-series processing

3.1 Processing – step by step

The processing starts with the raw time series data files from the data loggers. Additionally, a configuration-file is needed. This contains all sort of meta data and further information. The configuration file is an ascii text file of the form

```
[station_name_1]

keyword1 = value1
keyword2 = value2
.
.
[station_name_2]
.
.
```

Few basic keywords are required:

```
- latitude (deg)
- longitude (deg)
- elevation (in meters)
- sampling_interval (in seconds)
- station_type (MT, E, B)
```

Depending on the type of station the following entries are required.
E-field recorded:

```
- E_logger_type (edl/elogger)
- E_logger_gain (factor/gain level)
- E_instrument_type (electrodes)
- E_instrument_amplification (applied amplification factor)
- E_Xaxis_azimuth (degrees)
- E_Xaxis_length (in meter)
- E_Yaxis_azimuth (degrees)
- E_Yaxis_length (in meter)
```

B-field recorded:

```
- B_logger_type (edl)
- B_logger_gain (factor/gain level)
- B_instrument_type (coil, fluxgate)
- B_instrument_amplification (applied amplification factor)
```

Further information can be provided using more keywords. The content of the configuration file will be stored as meta data in the final EDI file (if not restricted by the EDI format).

3.1.1 Concatenate time series recorded by EDLs - dayfiles

Functions for concatenation are in the *mtpy.utils.filehandling* module. The reference to the module is given here as `FH`. It is assumed that the files are ascii-formatted, and are named according to the EDL standard. This is
`<stationname><year><month><day><hour><minute><seconds>.<channel>`;
and the channels in question are *ex,ey,bx,by,bz*, case insensitive.

The data in the files are assumed to be either in single-column form (instrument counts), or in two-column form: *single column time stamp (e.g. epochs or datetime-string) - instrument counts*

1. Find the sampling rate
   You have to know the duration of your single files (e.g. 10 mins, 1 hour,...). Chose a file, of which you know that it contains complete information for the full duration. Then obtain the sampling period with the function
   `FH.get_sampling_interval_fromdatafile(filename, duration in seconds)`

2. Put all files of interest into one folder (preferably sorted by station).

3. For generating dayfiles call
   `FH.EDL_make_dayfiles(foldername, sampling , stationname = None)`
   This generates a subfolder called *dayfiles*. If the given files cannot be merged continuously, several files are created for the same day.

4. Output dayfiles have a single header line, which starts with the character #. The content of the line is
   stationname ; channel ; sampling interval ; time of first sample ; time of last sample.
   Time stamps are given in epoch seconds.

3.1.2 Concatenate time series recorded by EDLs - arbitrary length

. .

Decimate time series

Generally speaking, MT deals with the spectral analysis of time series. Therefore it is beneficial to process these time series in their full length. Unfortunately, computer hardware limitations (memory) sometimes do not allow to just read in a complete set of time series. One possibility to overcome this issue is a pre-processing step on the time series: "decimation". The data are downsampled by a given factor, so thhe number of samples is samll enough to be fully processed at once.

Although the name suggests it, the actual decimation factor does not have to be "10". The factor is to be chosen in a way that the loss of information is minimised. If for instance the sampling rate is 100Hz, but the instrument sensitivity is only sufficient in a range < 20Hz, a decimation with a factor of 5 effectively causes no harm to the data.

Be careful!!
It may seem that a decimation by a factor of n just means to take every n-th element of a time series. But unfortunately, this can result in spurious phase information. In order to avoid any side effects, a more sophisticated algorithm has to be used.

The functions dealing with the decimation of time series can be found in `mtpy.`

3.1.3 Time series data calibration

The conversion of time series from lists of raw *instrument-output/-counts* into time series of data values with an actual physical meaning and approprite units is called *calibration*. The functions for the calibration are contained in the module `mtpy.processing.calibration` (here `Cal` in short).

The calibration depends on the instrument as well as on the respective data logger. The given time series are multiplied by various factors, which are unique for each kind of instrument, logger, and their software setup. Additionally, possible spurious offsets can be removed.

The basic unit for the magnetic flux density is Tesla ($[\mathbf{B}] = T$), the unit of the electric field is Volt per meter ($[\mathbf{E}] = \frac{\mathrm{V}}{\mathrm{m}}$). For obtaining more convenient numbers, these units are often scaled e.g. to nT or $\frac{\mu \mathrm{V}}{\mathrm{m}}$

Using the EDL or "elogger" datalogger, the time series values are given in microvolts μV. The amplification factors of the instruments as well as the gain factor of the data logger are given in the configuration files. The basic function `Cal.calibrate()` carries out a calibration for one given data array. In order to calibrate a data file, use `Cal.calibrate_file()` instead. For calibrating several time series at once, follow these steps:

1. Put all raw time series files into one folder.
2. Setup a proper configuration file, containing the necessary information on all included stations.
3. Call the wrapper script
   `calibratefiles.py <directory name> <configuration file> [optional: <output directory>]`

The result is a collection of calibrated data files located in the output directory. If no explicit output directory is given, a new subdirectory "calibrated" is generated within the input directory.

The units are microvolts per meter ($\mu V/m$) for the electric field and nanoteslas (nT) for the magnetic field.

The calibrated files have a single header line, starting with the common commenting symbol "#". The header line contains the fields
`stationname, channel, unit of data, sampling interval, timestamp first sample , timestamp last sample, latitude, longitude, elevation`
Hence from this point on all basic information are present, which are necessary for continuing the MT processing, even if the configuration file should be lost.

In order to be able to distiguish between latitude and longitude values, the standard position format is applied: latitude values are given as 2-digit numbers $\pm dd.ddddd$, whereas longitudes are in the 3-digit form $\pm ddd.ddddd$. The positions are given to 5 decimal digits, this means an accuracy of < 2m.

3.2 E field logger data

4 Modelling (1-/2-/3-D)

4.1 ModEM

4.1.1 ModEM: input files

The code was built for the inputs to be .EDI files, so this is where we will start. You should have a directory where all your .EDI files exist, at least all the ones that you want to model.

There a few ways to begin. You can either make the data file first, then the mesh or the other way around. Both methods will be covered below.

We assume as ModEM does and that:

- **x == North**
- **y == East**
- **z == positive down**
- All spatial dimensions are in meters.
- **Note:** If your data set spans more than one UTM zone, then station locations may look a little funny or slightly off. This is because I have added a general fudge factor to account for grid changes as this is the simplest method I came up with. So you can change these fudge factors which are `_utm_grid_size_north` and `_utm_grid_size_east` (both part of `modem.Data` and `modem.Model` classes). I suggest plotting the stations in latitude and longitude for a reference and then change the parameters so the station locations match. If someone wants to come up with a better idea, then that is why this is open source.

Creating a Mesh and Data file

```
:Example --> 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 grid from the stations themselves with 200m cell spacing
    >>> mmesh = modem.Model(edi_list=edi_list, cell_size_east=200,
                            cell_size_north=200)
    >>> 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)
>>> # load in the station locations from the model mesh
>>> # use copy so you don't over write anything if you change something
>>> md.station_locations = mmesh.station_locations.copy()
>>> md.write_data_file(save_path=r"/home/modem/Inv1")
```

```
:Example --> create data file first then model 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) create data file
    >>> md = modem.Data(edi_list)
    >>> md.write_data_file(save_path=r"/home/modem/Inv1")
    >>> #3) make a grid from the stations themselves with 200m cell spacing
    >>> mmesh = modem.Model(edi_list=edi_list, cell_size_east=200,
                            cell_size_north=200,
                            station_locations=md.station_locations.copy())
    >>> 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")
```

Rotate Mesh

If you want to rotate the mesh, this can be a bit of a mind game because ModEM assumes the grid is oriented North and East. Therefore, you need to rotate the station locations to the angle at which you want to rotate. Now you also need to rotate you data to this new angle as well so that the off-diagonal components are parallel with the grid. The mesh will look a little funny and can be irritating as it usually makes the grid larger. So you may choose a different option than the one I've outlined here.

```
:Example --> Rotate Mesh:

    >>> mmesh.mesh_rotation_angle = 60
```

```
        >>> mmesh.make_mesh()
        >>> mmesh.write_model_file(save_path=r"/home/modem/Inv1")
        >>> # rotate data
        >>> md.station_locations = mmesh.station_locations.copy()
        >>> md.rotation_angle = 60
        >>> md.write_data_file(save_path=r"/home/modem/Inv1")
```

If you already have a data file and would like to create a mesh then you can do the following:

```
:Example --> Rotate Mesh:

        >>> md = modem.Data()
        >>> md.read_data_file(r"/home/modem/Inv1/ModEM_Data.dat"
        >>> # make a grid from the stations themselves with 200m cell spacing
        >>> mmesh = modem.Model(edi_list=edi_list, cell_size_east=200,
                                cell_size_north=200,
                                station_locations=md.station_locations.copy())
        >>> 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")
```

Control Files

There are two control files that you need to make in order for ModEM to run. That is a control file for how the forward modeling code runs and a control file for how the inversion runs.

To write the control file for the inversion:

```
:Example --> make control.inv :

>>> cntrl_inv = modem.Control_Inv()
>>> # change some parameters
>>> cntrl_inv.rms_target = 1.5
>>> cntrl_inv.max_iterations = 200
>>> cntrl_inv.lambda_initial = 100
>>> # write file
>>> cntrl_inv.write_control_file(save_path=r"/home/modem/Inv1")
```

If you already have a control file but want to change some parameters:

```
:Example --> make control.inv :

>>> cntrl_inv = modem.Control_Inv()
>>> cntrl_inv.read_control_file(r"/home/modem/Inv1/control.inv")
>>> # change some parameters
```

```
>>> cntrl_inv.rms_target = 1.5
>>> cntrl_inv.max_iterations = 200
>>> cntrl_inv.lambda_initial = 100
>>> # write file
>>> cntrl_inv.write_control_file()
```

To write the control file for the forward model:

```
:Example --> make control.fwd :

>>> cntrl_fwd = modem.Control_Fwd()
>>> # change some parameters
>>> cntrl_fwd.max_num_div_calls = 10
>>> cntrl_fwd.max_num_div_iters = 75
>>> # write file
>>> cntrl_fwd.write_control_file(save_path=r"/home/modem/Inv1")
```

Covariance File

The covariance is an important input in ModEM and can change the model drastically depending on the parameters. A covariance file can be written by:

```
:Example --> make covariance.cov:

>>> cov = modem.Covarianc()
>>> # if you know the grid dimensions (Nx, Ny, Nz)
>>> cov.grid_dimensions = (30, 20, 25)
>>> # if you made the mesh with modem.Model
>>> cov.grid_dimensions = (mmesh.grid_north.shape[0],
   mmesh.grid_east.shape[0],
   mmesh.grid_z.shape[0])
>>> # change the covariance in value [0,1]
>>> cov.smoothing_east = 0.4
>>> cov.smoothing_north = 0.3
>>> # if you have an array of masked values
>>> # must have same shape as grid dimensions
>>> cov.mask_arr = mask_array.copy()
>>> # write file
>>> cov.write_covariance_file(save_path=r"/home/modem/Inv1")
```

5 Visualisation

6 Helpful information, tools, and scripts