# README: matlab tools for EM time series

M. Becken

04/28/14

## Recent changes:

- before 08.05.2014
  The name of the afc files has been extended to include the actual time range the spectra cover. For very large data sets, it may be necessary to introduce a brake at some point to make the file size handable. To discriminate these output files, the time range is added in the central part of the filename.
- 08.05.2014
  - Introduced a possibility to access and change all public properties of an EMSpectra object through an EMTimeSeries object. This allows to modify settings for spectral estimation in batch mode. See section 3.4 for an example
  - Added simple first diff. prewhitening as an option to EMSpectra

## 1 Overview

The package contains matlab tools to read time series data, to compute spectra and to perform robust estimation of linear and bi-linear transfer functions. The code is largely organized in matlab classes. A matlab class is a convenient way to organize and store information on data sets. A class defintion includes properties - in the appearance similar to the fields of structures, and methods - functions which are exclusively available for the objects derived from that class. A variable derived from a class is called object.

Class properties can be discriminated into public properties, private properties and dependent properties. Public properties are properties whose value can be changed from the command line or within a matlab script or function. The syntax is the same as when assigning a value to a field of a struct. However, the format of the value must match the format in the classe definition. The general syntax to assign a value to a public property is

```
>> obj.publicproperty = <value>;
```

Public properties are used to control some output of the methods, for instance by defining a channel, a sample range, etc. to be plotted by some plot method.

In contrast, private properties do not accept assignments. Their values are set during the initialization of an object. For example, header information contained in a data file is meaningfully stored in private properties.

Note, that both public and private properties are given default values during the construction of an object, such that a default object conatins valid values.

Dependent properties are derived from other properties of the class. In fact, each dependent property is defined as a method (namely the get.method, a special function type) of other properties. Invoking a dependent property with the command

```
>> value = obj.publicproperty;
```

triggers a function call every time it is called. This has the great advantage that dependent properties are always up-to-date, when some of the other properties are changed. The dis-adavantage is that computations may be re-performed even if not necessary.

An example usage of a class is as follows;

`>> obj = ClassName(<filename>);` % generate an object of type ClassName to work with data in file <filename>

`>> obj.privateproperty_i = <value>;` % set some property to some value

`>> output = obj.dependentproperty_j;` % retrieve output dependentproperty_j that dependes on the value of some private properties, e.g. obj.privateproperty_i

`>> output = method_k(obj,varargin);` % invoke some method, e.g. plotting etc.

A special method is, among others, the display method. Typing the name of a variable on the command line ususally returns its value in the matlab command window. This behavior is actually defined by the display method. If a display method is defined, typing the name of the object will thus produce some formatted output on the command window.

The available classes in this package and their main purposes include

- `EDEs`: Read, resample, convert and plot EDE data for a single run
- `ADUs`: Read, resample and plot ADU data for a single run
- `EMSpectra`: Compute, write, read and plot spectra
- `EMRobustProcessing`: Robust linear and bi-linear transfer function estimation
- `EMTimeSeries`: Organize time series data for multiple stations, runs and sampling rates, and perform batch mode operations for time series and spectral estimation. Combines `EDEs`/`ADUs` and `EMSpectra`.
- `EMProc`: Organize spectral data for multiple stations, runs and sampling rates, and perform batch mode robust processing for local and interstation transfer functions. Combines `EMSpectra` and `EMRobustProcessing`.

# 2 Annotated examples

## 2.1 Reading ADU data

We first read the header information of the ADU data for one run, stored into a single or into multiple contigous files

```
>> pathname = 'D:\DCtrain\A02\ts\adc\ADU\meas_2014-03-21_11-26-00'; % path where the data reside
>> adu = ADUs; % generate default ADUs object
>> adu.name = 'A02'; % add information about the sitename
>> adu.run = '001'; % add information about the run label
>> adu = ADUs(adu,pathname); % read the header information into the ADUs object
```

Now, everything about the data is known, You can extract some information about the recording by invoking the display method:

```
>> adu
```

This will produce the following output in the command window:

```
+ Station details for site A02 - run 000:
System:  ADU07e 346
Latitude:  52.820953°
Longitude:  6.515902°
Sampling rate 512 Hz
Start or recording:  21-Mar-2014 11:26:00 + 000.0000 ms (1st sample)
Stop or recording:  23-Mar-2014 10:19:59 + 998.0469 ms (last sample)
Channel 1:  Ex dipole 63.6 (m)
Channel 2:  Ey dipole 65.6 (m)
Channel 3:  Bx MFS06e mag.  #450
Channel 4:  By MFS06e mag.  #451
```

So far, none of the actual time series have been read; only the header information. To read the data, you may want to look at a particular time window, at particular channels.

```
>> usetime = [2014 03 22 00 00 00 2014 03 22 12 00 00]; % we look at 12 hours, from midnight to noon
>> adu.usesmp = get_usesmp(adu,usetime); % it is more practical to provide this information in terms of samples rarther than in terms of times
>> adu.reftime = [2014 03 21 00 00 00]; % for plotting, and for some internal computations, we choose this as a reference time; it is close before the survey
>> adu.usech = [1 2]; % we just want to plot the first two channels
>> data = adu.dataphys; % Now we can read the data for the given time interval and the given channel. Here, we read the data rescaled to physical units.
```

Perhaps, we prefer to resample data for some reason before producing some output:

```
>> adu.resmpfreq = 8; % to resample to 8 Hz
>> data = adu.dataphys; % Now, we obtain the data resampled at 8 HZ.
```

Instead of assigning the data to a variable, we can directly invoke the plot method to look at the data:

```
>> hax = plot(adu,'time','relative h','units','physical','factor',[1 -1],'color','k');
```

In this example, the data are plotted in physical units, the time axis is in hours relative to reftime, and the second channel is multiplied by -1 before plotting. The call returns an array of axes handles. The produced graph is depicted as a black line in Figure 1.

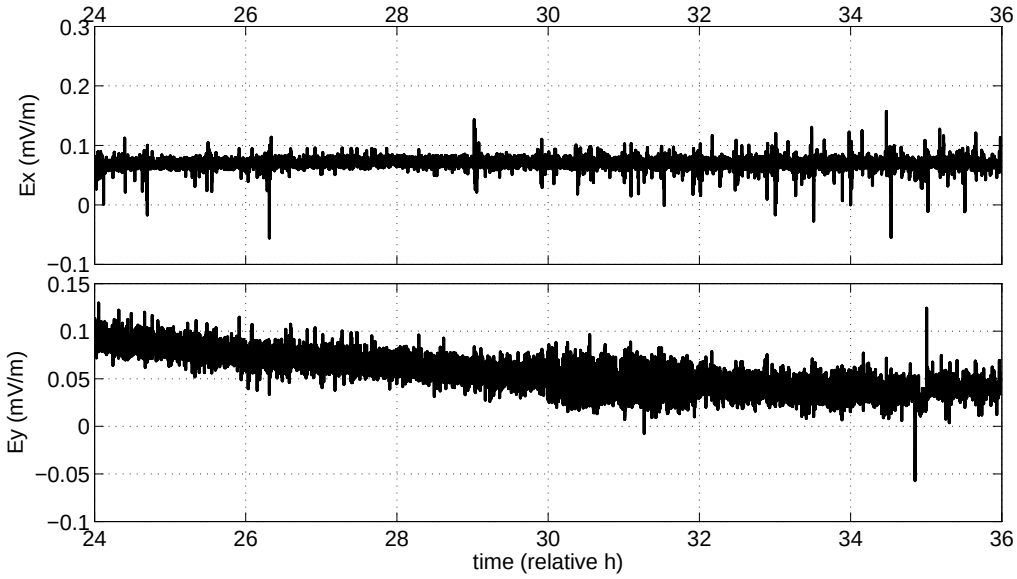


Figure 1: Time series recorded with an ADU system at 512 Hz and resampled to 8Hz. 12 hours are shown; the time axis is relative to the reference time.

## 2.2 Reading EDE data

The handling of EDE data is almost identical to that of ADU data. We first read the header information of the EDE data for one run, stored into a single or into multiple contigous files:

```
>> pathname = 'D:\DCtrain\A02\ts\adc\EDE\run2'; % path where the data reside
>> ede = EDEs; % generate default EDEs object
>> ede.name = 'A02'; % add information about the sitename
>> ede.run = '002'; % add information about the run label
>> ede = EDEs(ede,pathname); % read the header information into the EDEs object
```

Now, everything about the data is known. You can extract some information about the recording by invoking the display method:

```
>> ede
```

This will produce the following output in the command window:

```
+ Station details for site A02 - run 000:
System:  EDE 001
Latitude:  52.820939°
Longitude:  6.515931°
Sampling rate 500 Hz
Start or recording:  21-Mar-2014 17:09:59 + 998.0000 ms (1st sample)
Stop or recording:  22-Mar-2014 11:06:34 + 940.0000 ms (last sample)
PPS Delay:  80.30 us
Channel 1:  Ex dipole 63.6 (m)
Channel 2:  Ey dipole 65.6 (m)
```

Next, we want to plot the data for the same time window as we did before for the ADU data. The syntax is the same as before:

```
>> ede.usesmp = get_usesmp(ede,usetime);
>> ede.reftime = [2014 03 21 00 00 00];
>> ede.resmpfreq = 8;
>> hax = plot(ede,'time','relative h','units','physical','color','k','axes',hax);
```

Note that we provide the axes handles of the axes produced before to enable plotting into the same axes. The updated graph is depicted in Figure 2.

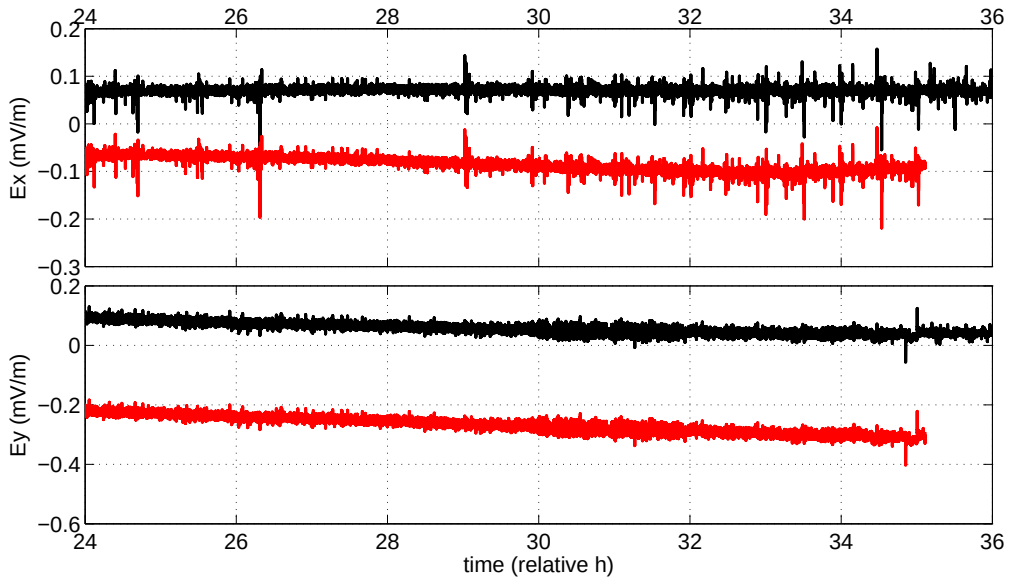


Figure 2: Time series recorded with an ADU system (black line) at 512 Hz and an EDE system (black line) at 500 Hz, both resampled to 8 Hz. 12 hours are shown; the time axis is relative to the reference time. Note that the ede recording was stopped before the the end of the plotted time interval.

## 2.3 Converting EDE data into ats format

EDE data can be converted into the ADU `*.ats` format. While the `*.ats` files are written with a complete header, the xml-file is not produced. Since the ADU and EDE work on different sampling rates, a resampling will typically be desired before format conversion. An example code is as follows:

```
>> pathname = 'D:\DCtrain\A02\ts\adc\EDE\run2'; % path where the data reside
>> ede = EDEs; % generate default EDEs object
>> ede.name = 'A02'; % add information about the sitename
>> ede.run = '002'; % add information about the run label
>> ede = EDEs(ede,pathname); % read the header information into the EDEs object
>> ede.usesmp = [1 ede.Nsmp]; % use all samples, but first and last samples will be omitted to make sure that we start (and stop) at a full second; Note that you can also extract a limited range of samples here. The *.ats filed will be corespondingly short
>> ede.usech = [1 2]; % output the two channels
>> ede.reftime = reftime; % not very relevant for this
>> ede.resmpfreq = 512; % do resampling, can be anything >= 1 (Hz)
>> ede.atsoutdir = {'D:\DCtrain\A02\ts\proc\EDE\run2'}; % this is the output directory for storing the *.ats-files ; the naming convention for *.ats files follows that used for the ADU data. If this property is not set, the default is to write the data into the same directory where the original EDE files reside
```

Now, we are ready to do the actual resmapling and conversion. This is simply triggered by the following command

```
>> atsfiles = ede.atsfile; % Invoking the atsfile property triggers exporting the data into *.ats format, with the names of the atsfiles returned in a cell array. The naming convention of the converted files follows that of the ADU system. In this example, the output file names are found from
>> atsfiles'
ans =
'D:\DCtrain\A02\ts\proc\EDE\run2\001_EDE_C01_R002_TEx_BL_512H.ats'
'D:\DCtrain\A02\ts\proc\EDE\run2\001_EDE_C02_R002_TEy_BL_512H.ats'
```

Note that the data are read, resampled and written seuqentially, so that there should be no memory problems, even with large data sets. To make sure that resampling operations to not introduce artefacts at the borders of each of the read intervals, we read actually some more data (exactly one second) before and after each time intervall, resample and cut out the relevant portion of the data.

Now the data should be readable with the ADUs class, e.g.

```
>> obj = ADUs('D:\DCtrain\A02\ts\proc\');
```

See the example codes that this really works.

## 2.4 Computing spectra

Spectrograms are computed in a cascade fashion similar to that used in Egbert's EMTF code. The default definition of the the cascade decimation scheme involves seven decimation levels, with sampling rates being reduced by a factor 4 between two adjacent decimation levels. For each decimation levels, a short window fourier transform is employed, with wondow lengths of 128 samples and window overlaps of 32 samples. The default window for windowing is a dpss window. Other options (**to be implemented**) include pre-whitening and delay filtering of the time series prior to fourier transformation. After spectral estimation, and before wrting the data to disk, the spectra are multiplied with the calibration functions for sensors and hardware. The spectrograms are computed for each channel separately. At present, it was more efficient (and convenient) to read the data for one channel completely before enetering the decimation scheme. If the time series is too long, you may run into memeory problems. This might be changed in future. Output format is `*.afc`. See the (**uncomplete**) format decription at the end of the documentation.

All parameters above are properties of the EMSpectra class, which is used to compute, store and read spectrograms. Modify these settings to your needs, and refer to the detailed class description of EMSpectra, given in the appendix. Here, I just provide a short example code to demonstrate the usage:

Let us use the ADU data from above:

```
>> pathname = 'D:\DCtrain\A02\ts\adc\ADU\meas_2014-03-21_11-26-00'; % path where the data reside
>> adu = ADUs;
>> adu.name = 'A02';
>> adu.run = '001';
```

```
>> adu = ADUs(adu,pathname);
>> adu.usesmp = [1 adu.Nsmp]; % Let us use all data
```

We do specify the channels to be used, which means that we want to use all channels. Now, let us create an EMSpectra object:

```
>> sp = EMSpectra;
>> sp.reftime = reftime; % Very importatnt to choose a common referernce time for the entire survey; this setting will internally overwrite the value of the reftime property in the ADUs/EDEs object
>> sp.source = {'D:\DCtrain\A02\fd'}; % this is the path where the *.afc file is written to. Do not be confused that the properties name is 'source' rather then destination. This is because the EMSpectra object is also used to access the data in the afc file; then this will be the source file :-). Note further that, after succesful computation of the spectra, the value of sp.source will be complemented with the output filename.
```

Otherwise we use the default settings. Calling the EMSpectra class definition with an ADUs/EDEs time series object as the second argument triggers the computation of the spectrograms; EMSpectra reads the data from the time series object, performs the spectral computation, calibration and write the data to disk.

```
>> sp = EMSpectra(sp,adu);
```

The naming convention is as self-explanatory and as follows:

```
>> sp.source
ans =
'D:\DCtrain\A02\fd\ADU\meas_2014-03-21_11-26-00\346_ADU_R001_TExEyBxBy_512H_Z20140321-000000.afc'
```

Note that we include the system and the reftime in the filename.

In the above, we have not provided any information about calibration data. In this case, theoretical transfer functions for coils, as well as for the EDE systems are built-in. Coil transfer functions can be computed if the code recognizes the sensor (currently basically the metronix coils) or the system (currently only the EDE). However, this can in the end not replace using individual sensor/system calibration files. If these are available, provide the path to these calibration data before computing the spectra, e.g.

```
>> sp.caldir = {'D:\DCtrain\s'};
>> sp = EMSpectra(sp,adu);
```

If the format of calibration files is recognized, they are read and used; otherwise the theoretical transfer functions are used. If nothing is recognized, e.g. unknown sensor, etc. then a unity transfer function is used for the respective channel.

Now, the EMSpectra object provides access to the fourier coeffcients, as would be an EMSpectra object that is initialized by reading an already existing *.afc file (next section).

*The EMSpectra class is designed to make no difference between different kinds of time series objects (e.g. ADUs or EDEs), so that the above example should work equally well with an EDEs object. Note that the resampled time series are used for spectral estimation, if resampling had been requested in the time series object. This would make it obsolete to perform format conversion of EDE into ADU data prior to spectral estimation. However, this has only been tested to the degree that the code produced some error, and the cause of the error has not yet been traced. Therefore, at the moment, the procedure would be to first resample the EDE data and convert them to *.ats format, and then perform the spectral computation on the correspondinf ADUs object (providing acces to either original ADU data or to converted EDE data).*

## 2.5 Reading spectra

The EMSpectra class is also used to read existing afc files. For example the above created file will can be read as follows:

```
>> filename = 'D:\DCtrain\A02\fd\ADU\meas_2014-03-21_11-26-00\...
346_ADU_R001_TExEyBxBy_512H_Z20140321-000000.afc';
>> sp = EMSpectra(filename);
```

The EMSpectra object should now be identical to the one created above.

*In practice, there is still some information missing in the header of the *.afc file such as the name of the calibration files and the calibration data.*

The display method outputs some basic information about the file:

```
>> sp % this should produce the following response:
+ Station details for site A02 - run 001:
Latitude:  52.820953°
Longitude:  6.515902°
Sampling rate:  512 Hz
First | last window:  21-Mar-2014 11:26:00 | 23-Mar-2014 10:19:59
Reference time:  21-Mar-2014 00:00:00
4 channels:  Ex, Ey, Bx, By
7 decimation levels:  512 Hz | 128 Hz | 32 Hz | 8 Hz | 2 Hz | 2 sec | 8 sec
```

## 2.6 Extracting fourier coefficients and plotting spectra

There are a number of switches to select particular channels, decimation levels, range of fourier coefficients and range frequencies to be read from file. Here, I provide an example, how to plot the autopower on one channel

First. load the data:

```
>> filename = 'D:\DCtrain\A02\fd\ADU\meas_2014-03-21_11-26-00\...
               346_ADU_R001_TExEyBxBy_512H_Z20140321-000000.afc';
>> sp = EMSpectra(filename);
```

We are interested to look at decimation level, say, 6

```
>> sp.usedec = 6;
```

Next, we specfiy a range of frequencies. Overall, with the current setting, we have 65 fourier coefficients at decimation level 6 (cf. `sp.Nf`), but want to truncate the first 4 coefficients.

```
>> sp.fcrange = [4 65];
```

We could also specify the range of sets (windows), but for now we decide to plot the entire data set. So we leave the sp.setrange property empty.

The plot method forms the autopower.

```
>> plot(sp,'channel','Ex','time','utc','clim',[-10 -2],'frequency','Hz');
```

The variable parameters request to plot Ex on an utc time axis, with the color axis in the range

Now, the output channel data can be read from file with these settings:

```
>> Y = sp.Y; % returns the complex fourier coeffcients for two channels, 61 coefficients and all sets.
```

Next, we wish to compare the spectra for the ADU and EDE recordings. We thus repeat the above procedure for both and extract the overlapping time windows.

```
filename = 'D:\DCtrain\A02\fd\EDE\run2\...
            001_EDE_R002_TExEy_512H_Z20140321-000000.afc'; % EDE spectra
spe = EMSpectra(filename);
spe.output = {'Ex' 'Ey'};
spe.usedec = 6;
filename = 'D:\DCtrain\A02\fd\ADU\meas_2014-03-21_11-26-00\...
            346_ADU_R001_TExEyBxBy_512H_Z20140321-000000.afc'; % ADU spectra
spa = EMSpectra(filename);
spa.output = {'Ex' 'Ey'};
spa.usedec = 6;
% Find the matching sets (time windows) with the intersect command by comparing the window index vectors relative to reftime
[a,b,c]=intersect(spa.wr,spe.wr);
spa.setrange = b;
```

```
spe.setrange = c;
spa.fcrange = [4 65];
spe.fcrange = [4 65];
plot(spa,'channel','Ex','time','utc','clim',[-10 -2],'frequency','Hz');
plot(spe,'channel','Ex','time','utc','clim',[-10 -2],'frequency','Hz');
```

This produces the Figures in Figs. 3 and 4.

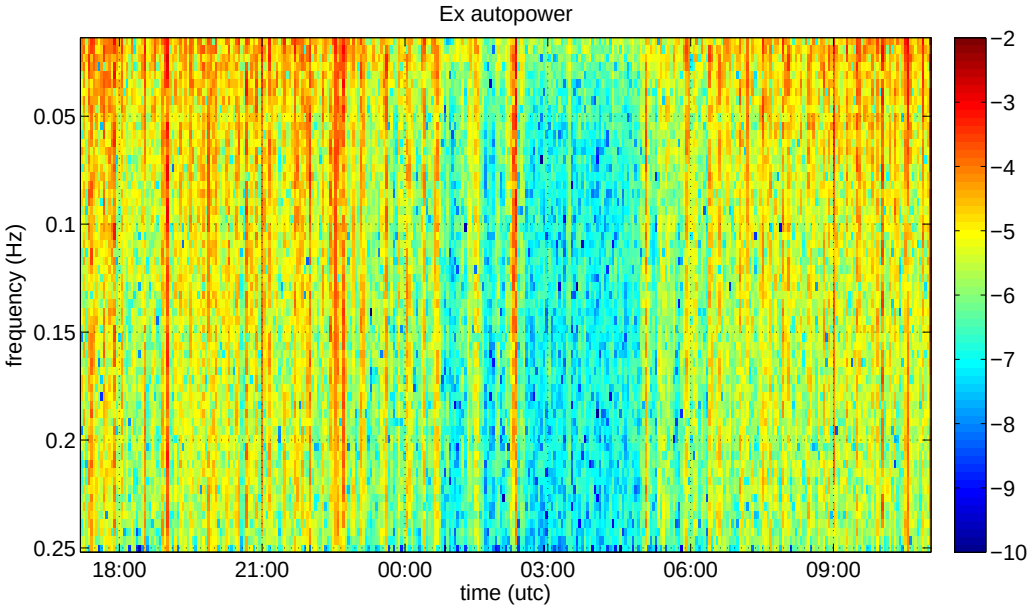


Figure 3: Ex-Spectrogram at decimation level 6 for the data recorded with an ADU system at 512 Hz. Only the portion of the data overlapping with one of the synchronous EDE recordings is shown.

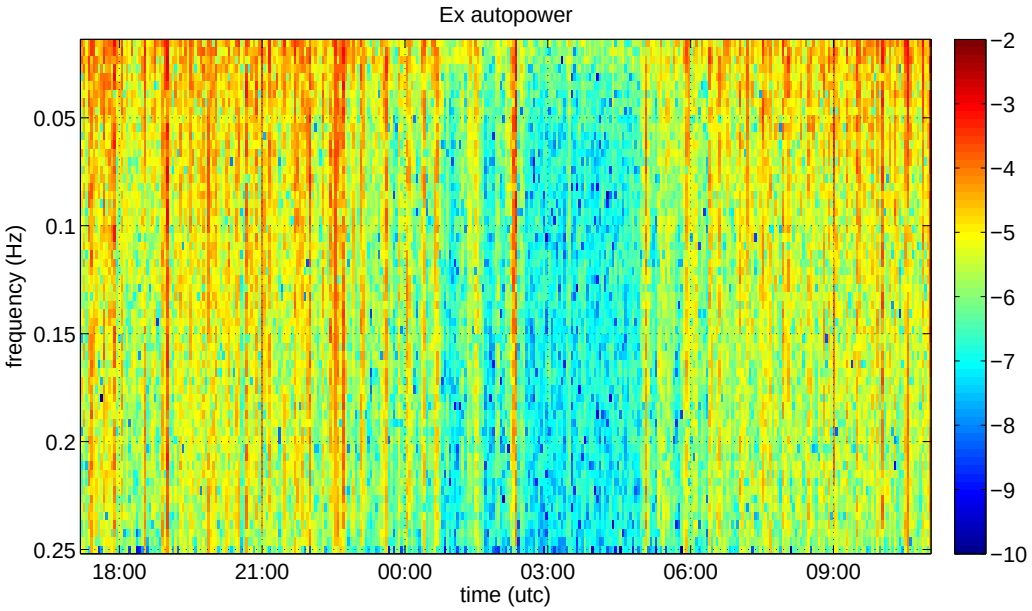


Figure 4: Ex-Spectrogram at decimation level 6 for the data recorded with an EDE system at 500 Hz resampled to 512 Hz and stored as in ats format. Only the portion of the data overlapping with the synchronous ADU recording is shown.

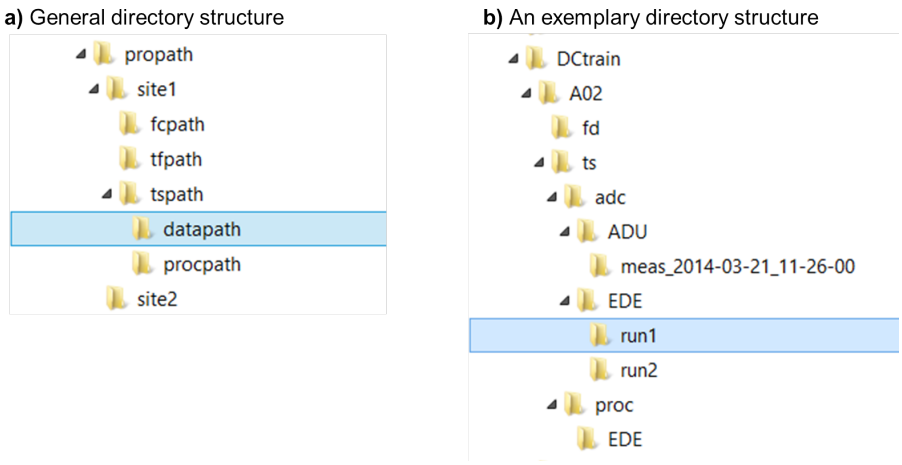


Figure 5: Directory structure used in the `EMTimeSeries` class. **a)** The general directory structure is encoded i public properties of an `EMTimeSeries` object.**b)** An example of a typical usage.

### 2.7 Computing single-site transfer functions

### 2.8 Computing inter-station transfer functions

## 3 EMTimeSeries: Operating with time series in batch mode

The `EMTimeSeries` object provides access to an entire survey consisting of multiples sites with multiple runs at multiple sampling rates. To achieve some degree of order, we require a particular directory structure as depicted inf Figure 5a. A project consists of a number of sites, and each site is organized in a directory structure with subdirectories for time series, spectra and transfer functions. Each of these subdirectories may be structured further. For instance, we usually store the original time series (as they come from the AD converter) in a subdirectory `./adc/system`, which also specifies the system type, e.g. `./adc/EDE`, and with further subdirectories to organize individual runs, e.g. `./adc/EDE/run*/` or `./adc/ADU/meas*/`. Processed data (e.g. resampled and converted data may be output) in a corresponding subdirectory `./proc/EDE/run*/` (only EDE data in this version). The above path definitions are encoded in public fields of the `EMTimeSeries` object and may be adopted to the individual needs. Fourier transformed data will be stored in the `./fc` subdirectory.

Internally, ADUs, EDEs and EMSpectra objects are initialized 'on the fly' and the appropriate options are set automatically. See section 3.4 for how to access and modify the default settings for an EMSpectra object.

### 3.1 Reading and plotting data and plotting runtimes

One of the main usage objectives of this class is to data quality control. A range of sites with multiple runs can therefore be read and accessed with a few lines of code. An example code is as follows:

```
%% Example EM Time series
>> propath = {'E:\DCtrain'};
>> reftime = [2014 03 21 0 0 0];
>> emts = EMTimeSeries(reftime,propath); % Initialize the object; Note that reftime and propath can not be changed any more.
>> emts = EMTimeSeries(emts,{'E02' 'E03' 'E04' 'E05' 'E06' 'E07'}); % Read the data from the sites specified in the cell array
>> plotruntimes(emts,'time','utc');% Here, we have an overview abot the runtimes of the imported stations (cf. Fig. )6
```

Next, we specify some of the parameters which we wish to plot: time range, channels, sampling rate. We shall also distinguish between some site which we denote as the local site, and simultaneous recordings, which we shalle denote as the basesites.

```
>> usetime = [2014 03 22 12 0 0 2014 03 22 13 0 0];
>> emts.usetime = usetime;
>> emts.usech = {'Ex' 'Ey'};
>> emts.lsname = {'E02'}; % this is to specfiy the local site
>> emts.lsrate = 500; % among all sampling schemes recorded at the local site, we pick the recordings at 500 Hz
>> emts.bsname = {'E03' 'E04' 'E05' 'E06' 'E07'}; % these are the base sites
>> emts.bsrate = [500]; % for the base sites, we also pick the recordings at 500 Hz but this is not necessary. We can provide a vector of sampling rates, e.g. [500 512]
>> emts.resmpfreq = 8; % for plotting, we will resample all data to 8Hz
>> plot(emts,'time','utc','color','r'); % This will plot the data, using a red color for the local site, and a choice of colors for all base sites (cf. Figure 7).
```

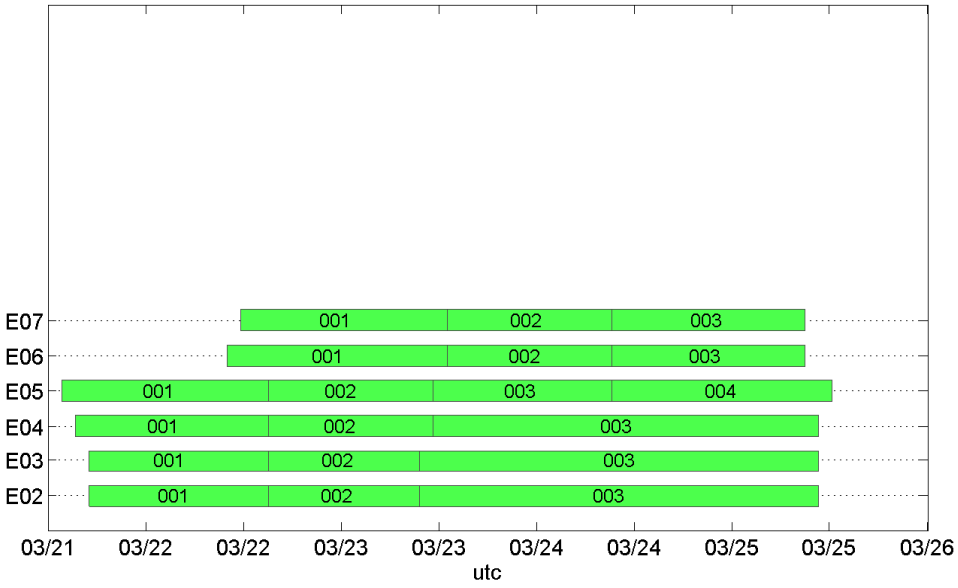


Figure 6: Run times for six sites. Run numbers are superposed on the bars. Time axis is labelled in utc time T

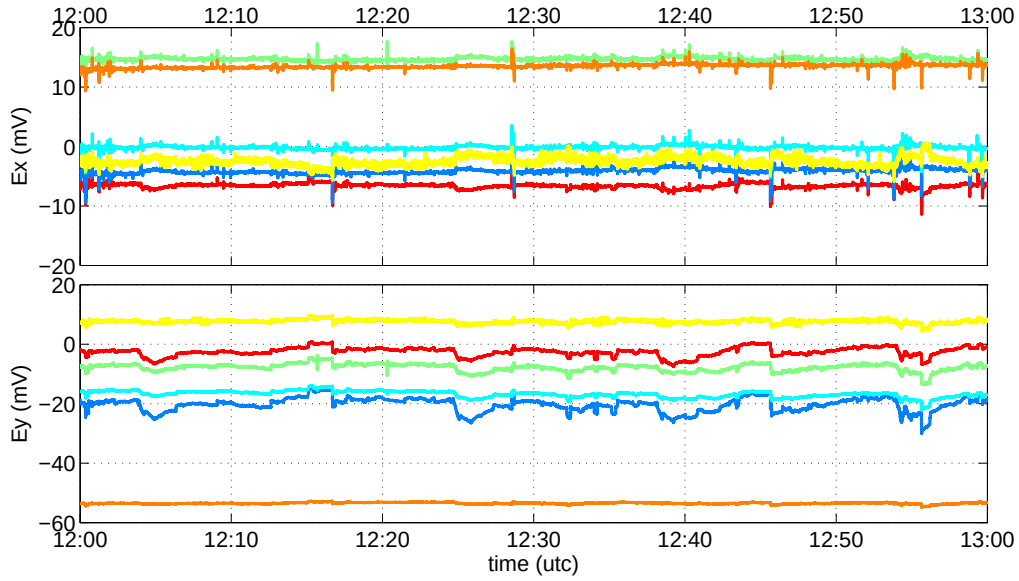


Figure 7: Time series recorded with the six different EDE systems at 500 Hz resampled to 8 Hz. 1 hour is shown; the time axis is in utc time.

## 3.2 Converting data

It may be necessary and convenient to resample and convert EDE data into *.ats format in batch mode. Here is an example code of how to accomplish this. In this example, all runs from the current local site are converted. If the usetime property is not set, then all data are used. Note that it is not necessary to define the base sites in this case. A simple loop over all stations would convert all sites. Note further that at present, data conversion will only work on ede data. Trying to apply this to adu data will most likely produce an error.

```
%% Example EM Time series:  convert local site data into ats format resampled to 512 Hz
>> propath = {'E:\DCtrain'};
>> reftime = [2014 03 21 0 0 0];
>> emts = EMTimeSeries(reftime,propath);
>> emts = EMTimeSeries(emts,{'E02' 'E03' 'E04' 'E05' 'E06' 'E07'});
>> emts.usech = {'Ex' 'Ey'};
>> emts.lsrate = 500;
>> emts.resmpfreq = 512;
>> emts.lsname = {'E02'};
>> atsfiles = emts.atsfiles;
```

The converted data will be stored in the subdirectory structure of obj.propath, e.g. or `./proc/EDE/run*/`. A simple for loop, such as

```
>> for is = 1:numel(emts.sites)
>>     emts.lsname = emts.sites(is);
>>     atsfiles = emts.atsfiles;
>>end
```

does the same sequentially for all stations.

## 3.3 Computing spectra and storing them to file

In a similar way, spectra can be computed and stored in batch mode.

```
%% Example EM Time series:  Compute spectra of converted EDE and ADU data
>> propath = {'E:\DCtrain'};
>> reftime = [2014 03 21 0 0 0];
>> emts = EMTimeSeries(reftime,propath);
>> emts.datapath = {'./proc/EDE/run*' './adc/ADU/meas*'};
>> emts = EMTimeSeries(emts,{'E02' 'E03' 'E04' 'E05' 'E06' 'E07'});
>> emts.usech = {'Ex' 'Ey'};
>> emts.lsrate = 512;
>> afcfiles = emts.afcfiles;
```

The converted data will be stored in the subdirectory structure of obj.propath, e.g. or `./fc/EDE/run*/`. A simple for loop, such as

```
>> for is = 1:numel(emts.sites)
>>    emts.lsname = emts.sites(is);
>>    afcfiles{is} = emts.afcfiles;
>>end
```

does the same sequentially for all stations.

Some data sets may be too large to be handled conveniently. To breal one large data set int a range of spectra files, invoke the usetime property as follows:

```
usetime = {[2014 3 21 00 00 00 2014 3 24 00 00 00] ...
  [2014 3 24 00 00 00 2014 3 27 00 00 00] ...
  [2014 3 27 00 00 00 2014 3 30 00 00 00] ...
  [2014 3 30 00 00 00 2014 4 02 00 00 00]};
for is = 1:numel(emts.sites)
   for iuse = 1:numel(usetime)
      emts.usetime = usetime{iuse};
      emts.lsname = emts.sites(is);
      afcfiles = emts.afcfiles;
   end
end
```

Here, we broke the survey time into pieces of three days. This will result in smaller file sizes, at the expense that at we may omit 1 or 2 spectra sets (time windows) at each breal and at each decimation level.

### 3.4 Changing default properties of `EMSpectra` objects

There is a possibility to pass values to an EMSpectra objects through an EMTimeSeries object via the EMTimeSeries property spdef. obj.spdef is either emtpy (default) or a struct with fields which are passed to an EMSpectra object. Therefore field names must be valid public properties of the EMSpectra class. If not, they are ignored (without issuing a warning). spdef may cintain an arbitrary number of valid fields. Here is an example, how to modify, for instance the decimation scheme for spectral computation.

```
%% Example EM Time series:  Compute spectra of ADU data with a modified decimation scheme
>> propath = {'E:\DCtrain'};
>> reftime = [2014 03 21 0 0 0];
>> emts = EMTimeSeries(reftime,propath);
>> emts.datapath = {'./proc/EDE/run*' './adc/ADU/meas*'};
>> emts = EMTimeSeries(emts,{'E02' 'E03' 'E04' 'E05' 'E06' 'E07'});
%% Example EM Time series:  Compute spectra of converted EDE and ADU data
>> spdef.Ndec = 5;
>> spdef.wlength = [512 512 512 128 128];
```

```
>> spdef.noverlap = [128 128 128 64 64];
>> spdef.prew = [-1 -1 -1 -1 -1];
>> ...
>> emts.spdef = spdef;
>> ...
```

# A EMProc: Operating with spectra files in batch mode

## A.1 Computing transfer functions

```
% Example code
>> projectname = {'G:\DCTrain'};
>> proc = EMProc(projectname);
>> proc.datapath = {'EDE\run*'};
```

Load all spectra files for some stations:

```
>> proc = EMProc(proc,{'E01' 'E03','E05','E06'});
```

We will compute electric interstation transfer functions; Define input and output channels accordingly:

```
>> proc.input = {'Ex' 'Ey'};
>> proc.output = {'Ex' 'Ey'};
```

We need a rule which fourier coefficients to combine into one estimation frequency. A default setup (bandsetup) is to use two target frequencies per decade. The averaging rules are automatically determined for the given window lengths/decimation levels etc.

```
>> proc.bandsetup = 'MT';
```

We take the input channels from the base site and the output channels from the local site: Define these and also define the original samplingrate to use

```
>> proc.lsrate = 512;
>> proc.bsrate = 512;
>> proc.bsname = {'E03'};
>> proc.lsname = {'E05'};
>> proc.usetime = []; % leave empty:  use all avaible data (default)
```

Next, we can pass some options to the EMRobustProcessing routine: These are combined in the procdef property.

```
>> procdef.avrange = [10 2]; % averaging domain Nfc x Nsets for smoothing of spectral matrics; this imparts on the coherency estimation, polarization props.  etc.
>> procdef.bicohthresg = {[0.9 1]}; % threshold fcs for which the coherency is estimated below 0.9
>> proc.procdef = procdef; %% compute transfer functions for all data tfs = proc.tf;
```

Now, we are ready to compute the transfer functions:

```
>> tf = proc.tf;
```

The output is shown in Figure 8a.

Instead of using all data, we may consider only a subset by setting the usetime property. An example is as follows:

```
>> proc.usetime = [2014 03 23 23 30 0 2014 03 24 00 00 0];
>> tf = proc.tf;
```

This produces the output in Figure 8b.

## A.2 Computing coherency

We keep the settings as above, and compute the coherency between the input and output channels for all fourier coefficients at decimation level

```
% Example code
```

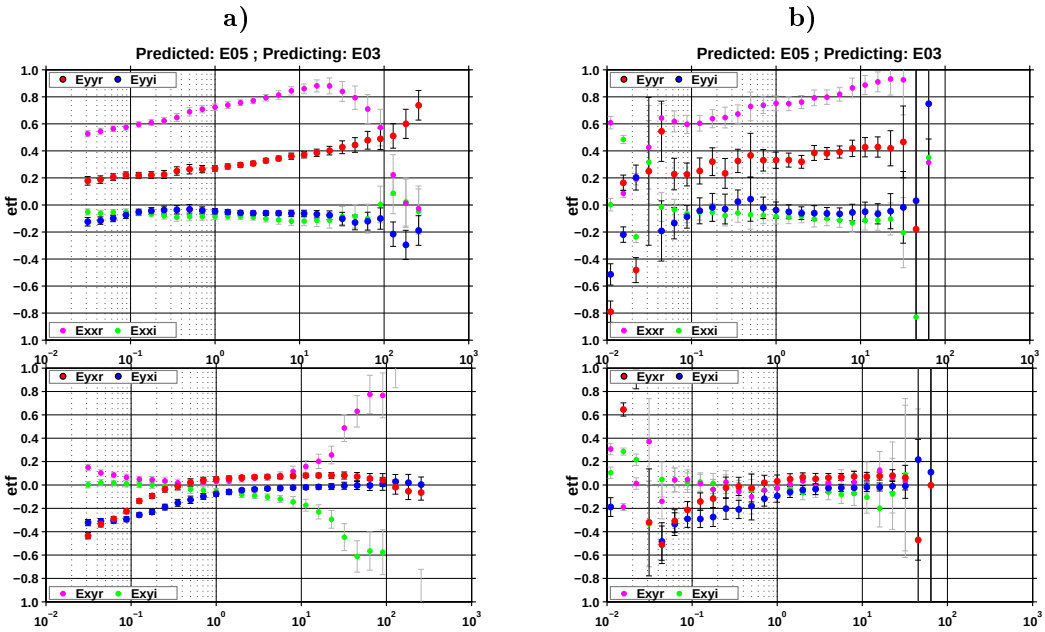


Figure 8: Transfer function estimates using **a)** one day of data **b)** half an hour of data.

```
>> proc.usetime = [2014 03 23 23 30 0 2014 03 24 00 00 0];
>> proc.usedec = 4;
>> proc.fcrange = [];
>> proc.input = {'Ex' 'Ey'};
>> proc.output = {'Ex'};
```

The coherency matrix is computed with the command

```
>> coh = proc.coh;
```

To produce the image in Figure 9, type

```
>> plotcoh(proc);
```

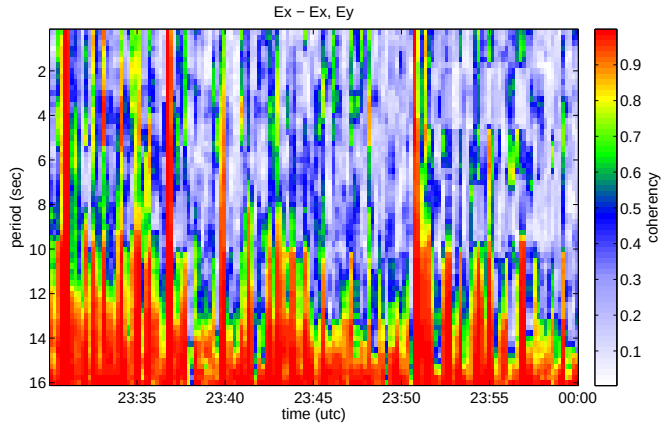


Figure 9: Bivariate coherency between local Ex and remote Ex and Ey channels for half an hour of data at decimation level.

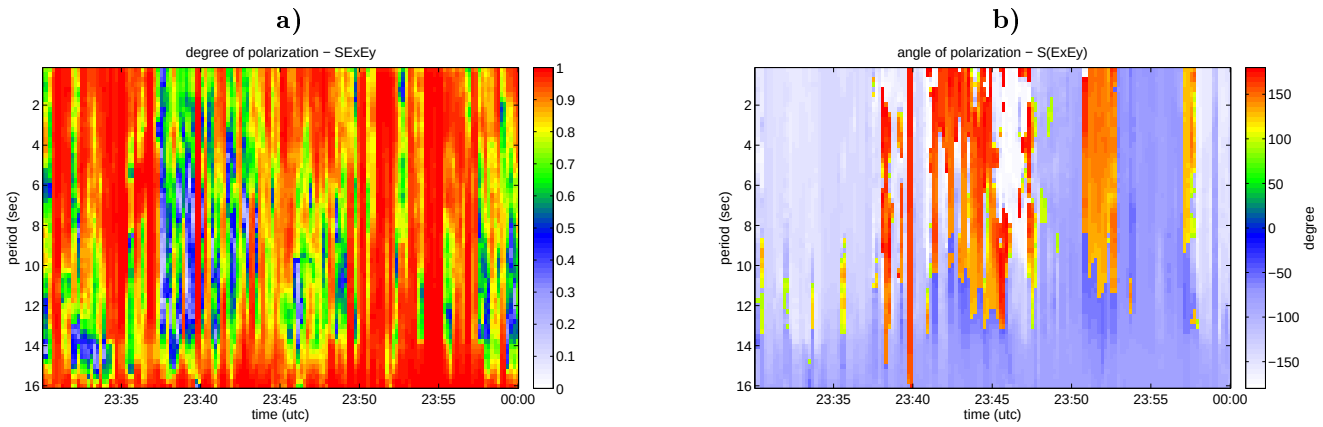


Figure 10: Polarization of the horizontal field at the base site, derived from an eigenvalue decomposition of the $2 \times 2$ spectral matrix. **a)** degree of polarization **b)** polarization of the field at the base site for half an hour of data at decimation level 4.

### A.3 Computing individual transfer functions

```
% Example code
```

...

### A.4 Computing polarization properties

We keep the settings above and compute the polarization characteristics with the command:

```
% Example code
>> pol = proc.pol;
```

To produce the image in Figure 10, type

```
>> plotpol(proc):
```

instead.

## B Class `EMTimeSeries`

An `EMTimeSeries` object is used to organize an entire survey constituting of multiple sites, each with multiple recordings in various sampling rates. The class provides the following functionality:

- Plot tuntimes for a given sampling frequency or a given array of sampling rates
- Plot time series for given channels, given time window, given local site and a range of synchronously recorded base sites. This can handle time series of different sampling rates. Resampling to a given sampling rate is possible.
- Batch mode for resampling and storing in ats format
- Batch mode for spectra computation

### B.1 Initializing the object

```
obj = EMTimeSeries(<reftime>,<propath>); % set global reference time and project path
obj = EMTimeSeries(obj,{'<site1>' '<site2>' ...}); % read data from site
```

## B.2 Public properties

`obj.propath` `{''};`

`obj.fcpath` `{'./fc'};`

`obj.tfpath` `{'./tf'};`

`obj.tspath` `{'./ts/'};`

`obj.datapath` `{'./adc/EDE/run*','./adc/ADU/meas*'}`

`obj.procpath=` `{'./proc'};`

`obj.calpath` `{'../s'};`

`obj.localsite` `{''};`

`obj.basesite` `{''};`

`obj.resmpfreq` `0;`

`obj.usetime` `[];`

`obj.usesites` `{'all'};`

`obj.usech` `{'all'} | {'Ex' 'Ey' 'Bx' ...}`

`obj.usesrates` `[]; % if empty, use all sampling rates; else provide array with numbers`

`obj.spdef` `[]; % settings for EMspectra object. if empty, default settings are used; otherwise you may pass a struct to spdef with fields whose names correspond to valid public properties of an EMSpectra object;`
`Example;`
`>> spdef.bandsetup = 'MT';`
`>> spdef.window = 'Hanning';`
`>> obj.spdef = spdef;`
`Then, these settings will be used for spectral estimation instead of the deault ones. See public properties of EMSpectra for a list of valid properties. Note that at the moment, the values in spdef are not tests for their validity ...`

`obj.debuglevel` `= 0 | 1 | 2;`

## B.3 Private properties

`obj.reftime` global survey reference time. This property is inherited to als timeseries objects (`EDEs, ADUs`) and remains therefore private.

`obj.sites` cell array with all station names

`obj.site` cell array collecting all time series objects (`EDEs, ADUs`). The cell is organized as follows:

## B.4 Dependent properties

`obj.lsind` index into local site. local time series objects are available from `obj.site{obj.lsind}`

`obj.lNruns` Number of runs (time series objects available) for local site

`obj.lsrates` sampling rates for each of the runs at the local site

`obj.lsrateind` index vector into runs of local site which match the requested sampling rate `obj.lsrate`. Time series objects of local site at this sampling rate are accessed with `obj.site{obj.lsind}(obj.lsrateind)` and returns a cell array with time series objects.

`obj.locasite` cell array with time series objects matching the current local site (`obj.lsname`) at the current sampling rate (`obj.lsrate`)

`obj.bsind` index vector into base sites. Base site time series objects for the `ibs`-st base site are available from `obj.site{obj.bsind{ibs}}`

`obj.bNruns` vector with number of runs (time series objects available) for base sites

`obj.bsrates` cell array of sampling rates at each base site and each run .

`obj.bsrateind` cell array of index vectors into runs of base site which match one the requested sampling rates in `obj.bsrate`. Time series objects of base site at this particular sampling rate are accessed with `obj.site{obj.bsind{ibs}}(obj.lsrateind{ibs})` and returns a cell array with time series objects.

`obj.basesite` cell array with time series objects matching one of the base sites (`obj.bsname`) at the current sampling rates (`obj.bsrate`)

`obj.atsfiles` triggers format conversion of local (`lsname`) EDE data into `*.ats` format for the given range of samples (derived from the `obj.usetime` property), the given channels `obj.usech` and the given sampling rate `obj.lsrate`. Data will be resampled, if `obj.resmpfreq` ist set (i.e. `> 0`)

## B.5 Methods

---

`varargout = display(obj);` Output some station details for the object

Example:

`>> obj` text output into the command window

`>> display(obj);` text output into the command window

`>> str = display(obj);` text output returned in a cell aray of strings (**not completed**)

---

`varargout = plotruntimes(obj,varargin)` Plot time series for the time segment given by `obj.usetime`. `varargin` is a variable sequence of keywords followed by a valid value, `varargout` returns an array of axes handles. The time series are plotted for the local site and all base sites.

Input arguments:

`'time', value,` appearance of time axis where `value` can be one out of
`value = 'relative smp';` samples relative to `obj.reftime`
`value = 'relative s';` seconds relative to `obj.reftime`
`value = 'relative h';` hours relative to `obj.reftime`
`value = 'relative d';` days relative to `obj.reftime`
`value = 'utc';` UTC time

Output arguments (**not implemented**):

`hax = plot(obj,varargin);` returns the handles to the axes.

---

`varagout = plot(obj,varargin}` Plot time series for the time segment given by `obj.usetime`. `varargin` is a variable sequence of keywords followed by a valid value, `varargout` returns an array of axes handles. The time series are plotted for the local site and all base sites.

Input arguments:

| | |
|---|---|
| `'time', value,` | appearance of time axis where `value` can be one out of<br>`value = 'relative smp';` samples relative to `obj.reftime`<br>`value = 'relative s';` seconds relative to `obj.reftime`<br>`value = 'relative h';` hours relative to `obj.reftime`<br>`value = 'relative d';` days relative to `obj.reftime`<br>`value = 'utc';` UTC time |
| `'units',value,` | appearance of time axis where `value` can be one out of<br>`value = 'int';` plots `obj.dataint` as integer numbers without any conversion to physical units<br>`value = 'mV';` plot `obj.data=obj.dataint*obj.lsb` in `mV`<br>`value = 'physical';` plot `obj.data/obj.dipole` in `mV/m` |
| `'elim',value,` | axis scaling for electric field plots, in the scale of `'units'`; Default is auto scaling |
| `'color', value` | color code for plotting, can be one of the valid matlab colors (e.g. 'r', 'k', 'b' ,...) or a [r g b] vector. |
| `'elim',` | `value`<br>`value = [min max];` scales the electric field axes |
| `'axes',` | `value`<br>`value = [hax1 hax2 ...];` plot the data into the existing axes `[hax1 hax2 ...]` |

Output arguments:

| | |
|---|---|
| `hax` | `= plot(obj,varargin);` returns the handles to each axes. Dimension of `hax` corresponds to dimension of `obj.usech` |

Examples:s

```
>> plot(obj);

>> plot(obj,'time','UTC');

>> plot(obj,'time','UTC','units','mV');

>> hax = plot(obj,'time','UTC','axes');

>> plot(obj,'time','UTC','axes',[hax1 hax2]);

>> plot(obj,'time','UTC','units','mV','elim',[-10 10],'axes',[hax1 hax2]);
```

# C Class `EMProc`

An `EMProc` object is used to .... The class provides the following functionality:

- Plot
- Batch mode for
- Batch mode for transfer function estimation

## C.1 Initializing the object

```
obj = EMProc(<propath>); % set global reference time and project path
obj = EMTimeSeries(obj,{'<site1>' '<site2>' ...}); % read data from site
```

## C.2 Public properties

`obj.propath` `{''};`

`obj.fcpath` `{'./fc'};`

`obj.tfpath` `{'./tf'};`

`obj.datapath` `{'./EDE/run*','./ADU/meas*'}`

`obj.lsname` `{};`

`obj.lsrate` `[];`

`obj.lsname` `{};`

`obj.lsrate` `[];`

`obj.refname` `{};`

`obj.input` `{'Bx' 'By'};`

`obj.input` `{'Ex' 'Ey'};`

`obj.refch` `{'Bx' 'By'};`

`obj.usedec` `{1};`

`obj.usetime` `[];`

`obj.fcrange` `[];`

`obj.procdef` `[]; % settings for EMRobustProcessing object. If empty, default settings are used;`
otherwise you may pass a struct to procdef with fields whose names correspond to
valid public properties of an EMRobustProcessing object;
Example;
>> procdef.reg = 'Mestimate';
>> procdef.cohthres = {[0.9 1]};
>> obj.procdef = procdef;
Then, these settings will be used for transfer function estimation instead of the
deault ones. See public properties of EMRobustProcessing for a list of valid properties
Note that at the moment, the values in procdef are not tests for their validity
...

`obj.debuglevel` `= 0 | 1 | 2;`

## C.3 Private properties

`obj.sites` cell array with all station names

`obj.site` cell array collecting all `EMSpectra` objects.

## C.4 Dependent properties

`obj.lsind` index into local site. Local `EMSpectra` objects are available from `obj.site{obj.lsind}`

`obj.lNruns` Number of runs (`EMSpectra` objects available) for local site

`obj.lsrates` sampling rates for each of the runs at the local site

`obj.lsrateind` index vector into runs of local site which match the requested sampling rate `obj.lsrate`.
`EMSpectra` objects of the local site at this sampling rate are accessed with `obj.site{obj.lsind}(obj.lsra`
and returns a cell array with `EMSpectra` objects.

| | |
|---|---|
| `obj.locasite` | cell array with time series objects matching the current local site (`obj.lsname`) at the current sampling rate (`obj.lsrate`) |
| `obj.bsind` | index vector into base sites. Base site `EMSpectra` objects for the `ibs`-st base site are available from `obj.site{obj.bsind{ibs}}` |
| `obj.bNruns` | vector with number of runs (`EMSpectra` objects available) for base sites |
| `obj.bsrates` | cell array of sampling rates at each base site and each run . |
| `obj.bsrateind` | cell array of index vectors into runs of base site which match one the requested sampling rates in `obj.bsrate`. `EMSpectra` objects of base site at this particular sampling rate are accessed with `obj.site{obj.bsind{ibs}}(obj.lsrateind{ibs})` and returns a cell array `EMSpectra` objects. |
| `obj.basesite` | cell array with time series objects matching one of the base sites (`obj.bsname`) at the current sampling rates (`obj.bsrate`) |
| `obj.tf` | triggers estimation of transfer functions for the curent settings. `tf` is a struct with the following fields:<br>... |
| `obj.coh` | univariate or bivariate coherency, depending on the number of the input channels |
| `obj.tfs` | Transfer functions estimated for each pixel of the smoothed spectral matrics. The Dimension is `Nfc x Nsets x (Ninput*Noutput)` for the current decimation level `usedec`, `fcrange` and `setrange, obj.input, obj.output.` |
| `obj.pol` | Polarization characetristics of the 2 input channels `obj.input`, formed from the eigenvalue decomposition of the `2x2` smoothed spectral matrix of the input channels. `pol.deg` is the degree of polarization and `pol.or` is the polarization direction (should be positive clockwise from north). The dimension of these fields is `Nfc x Nsets` for the current decimation level `usedec`, `fcrange` and `setrange` |

## C.5 Methods

**(these methods are not yet implemented with full options!)**

---

`varargout = display(obj);`

Example:

| | |
|---|---|
| `>> obj` | text output into the command window |
| `>> display(obj);` | text output into the command window |
| `>> str = display(obj);` | text output returned in a cell aray of strings |

---

`varargout = plottfs(obj,varargin)` Plot individual transfer functions

Output arguments :

`hax = plottfs(obj,varargin);` returns the handles to the axes.

---

`varargout = plotpol(obj,varargin)` Plot degree of polarization and polarization angle

Output arguments :

`hax = plottfs(obj,varargin);` returns the handles to the axes.

---

`varargout = plotcoh(obj,varargin)` Plot coherencies between each input and output channels. `if numel(input)==1`, univariate coherencies are computed; else `if numel(input)==2`, bivariate coherencies are computed. Only data within the time window given by `obj.usetime` are used; if `obj.usetime` is empty, all sets are used. `varargin` is a variable sequence of keywords followed by a valid value, `varargout` returns an array of axes handles.

# D Class EDEs

An `EDEs` object is used to access EDE data `*.mtd/*.txt`-format. The data reside in one directory corresponding to one run.

**2do:**

- lsb field must be a vector with one lsb value for each channel, to be consistent with the adu.
- include decimation
- include interrupted recordings as multiple runs

## D.1 Initializing the object

### D.1.1 Reading `*.mtd/*.txt` data

The object can be initialized with the commands

```
>> obj = EDEs; % returns an EDEs object with default properties
>> obj = EDEs('obj','pathname'); % populates an existing EDEs object
>> obj = EDEs('pathname'); % creates a new EDEs object and populates it
```

Here, `pathname` is the name of a valid path which contains a continous recording (run) stored in multiple pairs of `*.mtd/*.txt` files.
During the construction of the object, only the header information is read from the files and stored in the objects' properties.

## D.2 Public properties

| | |
|---|---|
| `obj.source` | path to the data directory of the run |
| `obj.name` | station name |
| `obj.system` | type of recording system |
| `obj.systemSN` | serial number of recording system |
| `obj.run` | run number |
| `obj.lat` | Latitude (deg) |
| `obj.lon` | Longitude (deg) |
| `obj.alt` | Altitude (m), usually set to zero |

| | |
|---|---|
| `obj.lsb` | conversion factor of `int32` to mV. `lsb = (5/2^32)*1000;` |
| `obj.Nch` | number of channels. |
| `obj.chnames` | channels names. `chnames = {'Ex' 'Ey'};` |
| `obj.dipole` | length of electric dipoles. For two channels, `dipole=[l1 l2];` |
| `obj.orient` | orientation of dipole/sensor |
| `obj.tilt` | tilt of sensor; ususally zero for electric fields |
| `obj.sens_sn` | serial number of sensors. `sens_sn = {[N S] [E W] ...}` |
| `obj.sens_name` | name of sensor, e.g. . `sens_name = {'AgAgCl' 'AgAgCl' ...}` |
| `obj.srate` | sampling rate |
| `obj.Nsmp` | total number of samples, taken over all data files |
| `obj.starttime` | starttime of the recording (1st sample) `[yyyy mm dd hh mm ss]` |
| `obj.starttimems` | milli-seconds of the 1st sample |
| `obj.ppsdelay` | Delay of pps signal; i.e. all samples are actually later by `ppsdelay.` |
| `obj.reftime` | reference time being used for plotting etc. It is intende to use this as the time to which all time windows for spectral estimation will refer |
| `obj.Nfiles` | Number of data files |
| `obj.mtdfiles` | `*.mtd` datafiles |
| `obj.hdfiles` | `*.txt` headerfiles |
| `obj.startstopfile` | `Nfiles x 14` array containing the times of the first and last sample in each file. The first 7 columns correspond to the date vector of the first sample (including milliseconds in the 7th column), and the last 7 columns contain the date vector of the last sample in the file (again including the milliseconds in the 14th column) |
| `obj.Nsmpfile` | `Nfiles x 2` array containing the indices of the first and last sample contained in each file relative to the overall first sample |
| `obj.usech` | channels to be used for whatever (e.g. for plotting) |
| `obj.usesmp` | sample range (relative to first sample of the current run) to be used for whatever (e.g. for plotting). To calculate the sample range for a given time range, use the method `usesmp = get_usesmp(obj,usetime);` |
| `obj.dec` | decimate data by this factor. Must be an integer. |
| `obj.resmpfreq` | resample data to this frequency; must be larger than 1 Hz |
| `obj.atsoutdir` | Output directory for converted `*.ats` files. Default is `atsoutdir = {''}` and means that the `*.ats` files will be written to the `ede` data directory given by `obj.source`. |
| `obj.afcoutdir` | Output directory for fourie coefficient files `*.afc` files. Default is `afcoutdir = {''}` and means that the `*.afc` files will be written to the `ede` data directory given by `obj.source`. (not implemented) |
| `obj.debuglevel` | Degree of information returned on the command line.<br>`debuglevel = 0;` only error messages;<br>`debuglevel = 1; (default)` some information;<br>`debuglevel = 2;` information and not so important warnings |

## D.3 Dependent properties

| | |
|---|---|
| `obj.starttimestr` | Starttime as a string |
| `obj.stoptimestr` | |
| `obj.usetime` | time range being to be used; derived from usesmp (**not implemented**) |
| `obj.usefiles` | `*.mtd` files which contain the data defined by `usesmp` |
| `obj.dataint` | integer data as stored in the file; these data are not being resampled, because this would result in floating point numbers. |
| `obj.dataproc` | resampled data, if obj.resmpfreq is set; otherwise the same as dataint. |
| `obj.data` | `dataproc` multiplied by the `obj.lsb` value to obtain `mV`. |
| `obj.dataphys` | `data` divided by the `obj.dipole` length, or multiplied by the coil sensitivity to obtain `mV/m` or `mV/nT` |
| `obj.smp` | sample vector for each sample (e.g sample axis for plotting) as defined by `usesmp`, but relative to startt of recording |
| `obj.rsmp` | sample vector for each sample (e.g sample axis for plotting) as defined by `usesmp`, but relative to `reftime` |
| `obj.trs` | seconds vector for each sample (e.g time axis for plotting) as defined by `usesmp`, but relative to `reftime` |
| `obj.trh` | hours vector for each sample (e.g time axis for plotting) as defined by `usesmp`, but relative to `reftime` |
| `obj.trd` | days vector for each sample (e.g time axis for plotting) as defined by `usesmp`, but relative to `reftime` |
| `obj.utc` | utc time vector (matlab datenum format) for each sample (e.g time axis for plotting) as defined by `usesmp`, but relative to `reftime` |
| `obj.usesmpr` | used samples for resampled data, derived from `usesmp` and `resmpfreq` |
| `obj.Nsmpr` | Number of samples of the resampled data, derived from `usesmpr` |
| `obj.smpr` | sample vector for each sample of the resampled data (e.g sample axis for plotting) as defined by `usesmpr`, relative to start of recording |
| `obj.rsmpr` | sample vector for each sample of the resampled data (e.g sample axis for plotting) as defined by `usesmpr` relative to to `reftime` |
| `obj.trsr` | seconds vector for each sample of the resampled data (e.g time axis for plotting) as defined by `usesmpr` relative to `reftime` |
| `obj.trhr` | hours vector for each sample of the resampled data (e.g time axis for plotting) as defined by `usesmpr` relative to `reftime` |
| `obj.trdr` | days vector for each sample of the resampled data (e.g time axis for plotting) as defined by `usesmpr` relative to `reftime` |
| `obj.utcr` | utc time vector for each sample of the resampled data (e.g time axis for plotting) as defined by `usesmpr` relative to `reftime` |
| `obj.atsheader` | returns a struct with all the information contained in the header of an `*.ats` file |
| `obj.atsfile` | invoking the `atsfile` property ats triggers an export of the data into `*.ats` format. `atsfile` is a cell array fith the filenames of the written `*.ats` files. Note that the data are written for the channels contained in `obj.usech`, for the sample range given by `obj.usesmp`, and resampled to `obj.resmplfreq`. The default output directory is the directory of the `*.mtd/*.txt` ede data files, but the output path can be changed by changing the value of `obj.atsoutdir`. |

## D.4 Methods

---

`varargout = display(obj);` Output some station details for the object-

Example:

`>> obj` text output into the command window

`>> display(obj);` text output into the command window

`>> str = display(obj);` text output returned in a cell aray of strings

---

`usesmp = get_usesmp(obj,usetime);` Calculate sample indices corresponding to a given time interval. Provide a datevector of type `usetime = [<from> <to>];` where `<from>` and `<to>` are variables in format `[yyyy mm dd hh mm ss]`. The first extracted sample from the file is exactly recorded at the time `<from>`, whereas the last sample is exactly recorded at the time `<to> - 1/obj.srate` (i.e. one sample less before the `<to>` time!). If the `<from>` is earlier in time than the actual start of the recording (i.e. `obj.starttime+obj.starttimems/1000`), then the start of the recording will be used instead. If `<to>` is later than the end of the recording (`obj.stoptime+obj.stoptimems/1000`), then the end of recording will be used instead.

Example:

```
>> usesmp = get_usesmp(obj,[2014 3 17 10 00 00 2014 3 17 20 00 00]);
```

---

`varagout = plot(obj,varargin}` Plot time series for the time segment given by `obj.usesmp`. `varargin` is a variable sequence of keywords followed by a valid value, `varargout` returns an array of axes handles.

Input arguments:

`'time', value,` appearance of time axis where `value` can be one out of

| | |
|---|---|
| `value = 'smp';` | samples relative to the beginning of the recording |
| `value = 'relative smp';` | samples relative to `obj.reftime` |
| `value = 'relative s';` | seconds relative to `obj.reftime` |
| `value = 'relative h';` | hours relative to `obj.reftime` |
| `value = 'relative d';` | days relative to `obj.reftime` |
| `value = 'utc';` | UTC time |

`'units',value,` appearance of time axis where `value` can be one out of

| | |
|---|---|
| `value = 'int';` | plots `obj.dataint` as integer numbers without any conversion to physical units |
| `value = 'mV';` | plot `obj.data=obj.dataint*obj.lsb` in `mV` |
| `value = 'physical';` | plot `obj.data/obj.dipole` in `mV/m` |

`'elim',value,` axis scaling for electric field plots, in the scale of `'units';` Default is auto scaling

`'color', value` color code for plotting, can be one of the valid matlab colors (e.g. 'r', 'k', 'b' ,...) or a [r g b] vector.

`'elim', value`

`value = [min max];` scales the electric field axes

'factor', value
: value = [fac1 fac2 ...]; multiplies each channel by a factor. Dimension of value must match the dimension of obj.usech

'axes', value
: value = [hax1 hax2 ...]; plot the data into the existing axes [hax1 hax2 ...]

Output arguments:

hax
: = plot(obj,varargin); returns the handles to each axes. Dimension of hax corresponds to dimension of obj.usech

Examples:

```
>> plot(obj);
>> plot(obj,'time','UTC');
>> plot(obj,'time','UTC','units','mV');
>> hax = plot(obj,'time','UTC','axes');
>> plot(obj,'time','UTC','axes',[hax1 hax2]);
>> plot(obj,'time','UTC','units','mV','elim',[-10 10],'factor',[1 -1],'axes',[hax1 hax2]);
```

# E Class ADU

An ADU object is used to access ADU data in *.ats-format. The data reside in one directory corresponding to one run, typically of name meas*.

**2do:**

- include output of resampled data
- include decimation
- include interrupted recordings as multiple runs, or recrodings at different sampling rates
- xml files are not read

## E.1 Initializing the object

### E.1.1 Reading *.ats data

The object can be initialized with the commands

```
>> obj = ADUs; % returns an ADUs object with default properties
>> obj = ADUs('obj','pathname'); % populates an existing ADUs object
>> obj = ADUs('pathname'); % creates a new ADUs object and populates it
```

Here, pathname is the name of a valid path which contains a continous recording (run) stored in multiple *.ats files.

During the construction of the object, only the header information is read from the files and stored in the objects' properties.

## E.2 Public properties

| | |
|---|---|
| `obj.source` | path to the data directory of the run |
| `obj.name` | station name |
| `obj.system` | type of recording system |
| `obj.systemSN` | serial number of recording system |
| `obj.run` | run number |
| `obj.lat` | Latitude (deg) |
| `obj.lon` | Longitude (deg) |
| `obj.alt` | Altitude (m), usually set to zero |
| `obj.lsb` | conversion factor of `int32` to mV. `lsb = (5/2^32)*1000;` |
| `obj.Nch` | number of channels. |
| `obj.chnames` | channels names. `chnames = {'Ex' 'Ey'};` |
| `obj.dipole` | length of electric dipoles. For two channels, `dipole=[l1 l2];` |
| `obj.orient` | orientation of dipole/sensor |
| `obj.tilt` | tilt of sensor; ususally zero for electric fields |
| `obj.sens_sn` | serial number of sensors. `sens_sn = {[N S] [E W] ...}` |
| `obj.sens_name` | name of sensor, e.g. . `sens_name = {'AgAgCl' 'AgAgCl' ...}` |
| `obj.srate` | sampling rate |
| `obj.Nsmp` | total number of samples, taken over all data files |
| `obj.starttime` | starttime of the recording (1st sample) `[yyyy mm dd hh mm ss]` |
| `obj.starttimems` | milli-seconds of the 1st sample |
| `obj.ppsdelay` | Delay of pps signal; i.e. all samples are actually later by `ppsdelay.` |
| `obj.reftime` | reference time being used for plotting etc. It is intende to use this as the time to which all time windows for spectral estimation will refer |
| `obj.Nfiles` | Number of data files |
| `obj.mtdfiles` | `*.mtd` datafiles |
| `obj.hdfiles` | `*.txt` headerfiles |
| `obj.startstopfile` | `Nfiles x 14` array containing the times of the first and last sample in each file. The first 7 columns correspond to the date vector of the first sample (including milliseconds in the 7th column), and the last 7 columns contain the date vector of the last sample in the file (again including the milliseconds in the 14th column) |
| `obj.Nsmpfile` | `Nfiles x 2` array containing the indices of the first and last sample contained in each file relative to the overall first sample |
| `obj.usech` | channels to be used for whatever (e.g. for plotting) |
| `obj.usesmp` | sample range (relative to first sample of the current run) to be used for whatever (e.g. for plotting). To calculate the sample range for a given time range, use the method `usesmp = get_usesmp(obj,usetime);` |

`obj.dec` decimate data by this factor. Must be an integer.

`obj.resmpfreq` resample data to this frequency; must be larger than 1 Hz

`obj.atsoutdir` Output directory for converted `*.ats` files. Default is `atsoutdir = {''}` and means that the `*.ats` files will be written to the `ede` data directory given by `obj.source`.

`obj.afcoutdir` Output directory for fourie coefficient files `*.afc` files. Default is `afcoutdir = {''}` and means that the `*.afc` files will be written to the `ede` data directory given by `obj.source`. (not implemented)

`obj.debuglevel` Degree of information returned on the command line.
`debuglevel = 0;` only error messages;
`debuglevel = 1; (default)` some information;
`debuglevel = 2;` information and not so important warnings

## E.3 Dependent properties

`obj.starttimestr` Starttime as a string

`obj.stoptimestr`

`obj.usetime` time range being to be used; derived from usesmp

`obj.usefiles` `*.mtd` files which contain the data defined by `usesmp`

`obj.dataint` integer data as stored in the file; these data are not being resampled, because this would result in floating point numbers.

`obj.dataproc` resampled data, if obj.resmpfreq is set; otherwise the same as dataint.

`obj.data` `dataproc` multiplied by the `obj.lsb` value to obtain `mV.`

`obj.dataphys` `data` divided by the `obj.dipole` length, or multiplied by the coil sensitivity to obtain `mV/m` or `mV/nT`

`obj.smp` sample vector for each sample (e.g sample axis for plotting) as defined by `usesmp`, but relative to start of recording

`obj.rsmp` sample vector for each sample (e.g sample axis for plotting) as defined by `usesmp`, but relative to `reftime`

`obj.trs` seconds vector for each sample (e.g time axis for plotting) as defined by `usesmp`, but relative to `reftime`

`obj.trh` hours vector for each sample (e.g time axis for plotting) as defined by `usesmp`, but relative to `reftime`

`obj.trd` days vector for each sample (e.g time axis for plotting) as defined by `usesmp`, but relative to `reftime`

`obj.utc` utc time vector (matlab datenum format) for each sample (e.g time axis for plotting) as defined by `usesmp`, but relative to `reftime`

`obj.usesmpr` used samples for resampled data, derived from `usesmp` and `resmpfreq`

`obj.Nsmpr` Number of samples of the resampled data, derived from `usesmpr`

`obj.smpr` sample vector for each sample of the resampled data (e.g sample axis for plotting) as defined by `usesmpr`, relative to start of recording

`obj.rsmpr` sample vector for each sample of the resampled data (e.g sample axis for plotting) as defined by `usesmpr` relative to to `reftime`

| | |
|---|---|
| `obj.trsr` | seconds vector for each sample of the resampled data (e.g time axis for plotting) as defined by `usesmpr` relative to `reftime` |
| `obj.trhr` | hours vector for each sample of the resampled data (e.g time axis for plotting) as defined by `usesmpr` relative to `reftime` |
| `obj.trdr` | days vector for each sample of the resampled data (e.g time axis for plotting) as defined by `usesmpr` relative to `reftime` |
| `obj.utcr` | utc time vector for each sample of the resampled data (e.g time axis for plotting) as defined by `usesmpr` relative to `reftime` |
| `obj.atsheader` | returns a struct with all the information contained in the header of an `*.ats` file |
| `obj.atsfile` | invoking the `atsfile` property ats triggers an export of the data into `*.ats` format. `atsfile` is a cell array fith the filenames of the written `*.ats` files. Note that the data are written for the channels contained in `obj.usech`, for the sample range given by `obj.usesmp`, and resampled to `obj.resmplfreq`. The default output directory is the directory of the `*.mtd/*.txt` ede data files, but the output path can be changed by changing the value of `obj.atsoutdir.` |

## E.4 Methods

---

`varargout = display(obj);` Output some station details for the object-

Example:

| | |
|---|---|
| `>> obj` | text output into the command window |
| `>> display(obj);` | text output into the command window |
| `>> str = display(obj);` | text output returned in a cell aray of strings |

---

`usesmp = get_usesmp(obj,usetime);` Calculate sample indices corresponding to a given time interval. Provide a datevector of type `usetime = [<from> <to>];` where `<from>` and `<to>` are variables in format `[yyyy mm dd hh mm ss]`. The first extracted sample from the file is exactly recorded at the time `<from>`, whereas the last sample is exactly recorded at the time `<to> - 1/obj.srate` (i.e. one sample less before the `<to>` time!). If the `<from>` is earlier in time than the actual start of the recording (i.e. `obj.starttime+obj.starttimems/1000`), then the start of the recording will be used instead. If `<to>` is later than the end of the recording (`obj.stoptime+obj.stoptimems/1000`), then the end of recording will be used instead.

Example:

```
>> usesmp = get_usesmp(obj,[2014 3 17 10 00 00 2014 3 17 20 00 00]);
```

---

`varagout = plot(obj,varargin}` Plot time series for the time segment given by `obj.usesmp`. `varargin` is a variable sequence of keywords followed by a valid value, `varargout` returns an array of axes handles.

Input arguments:

| | |
|---|---|
| `'time', value,` | appearance of time axis where `value` can be one out of<br>`value = 'smp';` samples relative to the beginning of the recording<br>`value = 'relative smp';` samples relative to `obj.reftime`<br>`value = 'relative s';` seconds relative to `obj.reftime`<br>`value = 'relative h';` hours relative to `obj.reftime`<br>`value = 'relative d';` days relative to `obj.reftime`<br>`value = 'utc';` UTC time |
| `'units',value,` | appearance of time axis where `value` can be one out of<br>`value = 'int';` plots `obj.dataint` as integer numbers without any conversion to physical units<br>`value = 'mV';` plot `obj.data=obj.dataint*obj.lsb` in `mV`<br>`value = 'physical';` plot `obj.data/obj.dipole` in `mV/m`<br><br>`'elim',value,` axis scaling for electric field plots, in the scale of `'units'`; Default is auto scaling |
| `'color', value` | color code for plotting, can be one of the valid matlab colors (e.g. 'r', 'k', 'b' ,...) or a [r g b] vector. |
| `'elim',` | `value`<br>`value = [min max];` scales the electric field axes |
| `'factor',` | `value`<br>`value = [fac1 fac2 ...];` multiplies each channel by a factor. Dimension of `value` must match the dimension of `obj.usech` |
| `'axes',` | `value`<br>`value = [hax1 hax2 ...];` plot the data into the existing axes `[hax1 hax2 ...]` |

Output arguments:

| | |
|---|---|
| `hax` | `= plot(obj,varargin);` returns the handles to each axes. Dimension of `hax` corresponds to dimension of `obj.usech` |

Examples:s

```
>> plot(obj);
>> plot(obj,'time','UTC');
>> plot(obj,'time','UTC','units','mV');
>> hax = plot(obj,'time','UTC','axes');
>> plot(obj,'time','UTC','axes',[hax1 hax2]);
>> plot(obj,'time','UTC','units','mV','elim',[-10 10],'factor',[1 -1],'axes',[hax1 hax2]);
```

# F Class `EMSpectra`

An `EMSpectra` object is used to compute spectrograms or access the data stored in one fourier coefficient file `*.afc`.

## F.1 Initializing the object

The object can be initialized with the following commands:

```
>> obj = EMSpectra; % returns a default EMSpectra object.
>> obj = EMSpectra('filename.afc'); % reads a *.afc file named 'filename.afc'
>> obj = EMSpectra(obj,ede); % computes a spectrogram from an ede object and stores the output to an *.afc
>> obj = EMSpectra(obj,adu); % computes a spectrogram from an adu object and stores the output to an *.afc
```

`obj` contains the header information of the `*.afc` file and allows acces to its contents.

### F.1.1 Computation and storage of spectrograms

Spectrograms are computed in a cascade fashion with a windowed fourier transformation: The scheme involves the steps a) decimation, b) segmenting into overlapping windows c) multiplication with a taper d) fourier transformation e) multiplication with calibration functions. These steps are usually applied to short windows - yielding a narrow frequency band - and repeated for various decimation levels. The decimation scheme is defined by the properties `Ndec` and `decimate`, where `decimate` is a vector of dimension `Ndec` and defines the factors by which the time series are decimated from one level to the next. A typical choice for MT processing is for instance to use windows of 128 samples length, which overlap by 32 samples, and which will be decimated at consecutive factors of 4 at, say, seven levels, including the original sampling rate (=decimation level 1). In this case, we define

```
>> obj.Ndec     = 7;
>> obj.decimate = [1 4 4 4 4 4 4];
>> obj.wlength  = [128 128 128 128 128 128 128];
>> obj.noverlap = [32 32 32 32 32 32 32];
```

Let the original sampling rate be `512 Hz`. Then, the sampling rates of the decimated time series will then be `[512 128 32 8 2 1/2 1/8 1/32]`. Because the time series is real, we store only half of the fourier coefficients for each window, plus the DC component for completeness. In the above example, this would correspond to 65 coefficients for each time window at each decimation level, where the first entry is the DC component, and the remaing 64 coefficients span the frequency domain covered by the window.

For later processing purposes (e.g. remote reference, inter-station transfer functions, etc) it is very important to perform the windowing of the time series for an entire survey on a common time axis to ensure that windows overlap exactly. This time axis is defined relative to a global survey time, the `reftime`. It is thus **very important to use the same `reftime` and the same decimation scheme** for data that shall be combined in fourier domain. Note that the EMspectra property `obj.reftime` will overwrite the `reftime` property in the respective time series object; it is therefore not required to set this property explicitly in the `EDEs` and `ADUs` (and any other) objects being transformed.

Note: The data to transform are the resampled data, if the resampling option is activated in the `EDEs` and `ADUs` (and any other) objects, i.e. if in the `ede.resmpfreq>0` or `adu.resmpfreq > 0`; otherwise, the original data are used. Only the time segment corresponding to the `usesmp` property and the channels defined by `usech` are extracted. Example:

```
>> usetime = [2014 03 21 18 00 00 2014 03 22 00 00 00]; % we look at 6 hours, from midnight to
noon
>> reftime = [2014 03 21 00 00 00]; % survey reference time
>> adu = ADUs(<pathname>);
>> adu.usesmp = get_usesmp(adu,usetime); % this returns the samples corresponding to usetime
>> adu.resmpfreq = 128; % resampling of the data to 128 Hz
>> sp = EMSpectra;
>> sp.source = {'D:\DCtrain\A02\fd'}; % output directory, where the *.afc
>> sp.reftime = reftime;
>> sp = EMSpectra(sp,adu);
```

### F.1.2 Reading spectrograms

Existing spectra are accessed by the command

```
>> sp = EMSpectra('filename.afc'); % reads a *.afc file named 'filename.afc'
```

Band set up definitons (which fourier coefficients are to be averaged for a single target frequency) are defined in `@EMSpectra\private\sp_defaultbs.m`. Modify this function to your needs. The default produces averaging rules for logarithimcally equidistant target frequencies with two target frequencies per octave.

## F.2 Public properties

Additional properties must be defined to extract the data subsets of interest. These include the range of frequencies, the range of sets (time windows), the decimation level and the channels. These properties are listed below; default values are given.

| | |
|---|---|
| `obj.source` | `{'filename.afc'}` |
| `obj.name` | `'000'` |
| `obj.run` | `'001'` |
| `obj.lat` | `0` |
| `obj.lon` | `0` |
| `obj.alt` | `0` |
| `obj.reftime` | `[1970 1 1 0 0 0];` |
| `obj.caldir` | `{''};` |
| `obj.Ndec` | `7;` |
| `obj.decimate` | `[1 4 4 4 4 4 4];` |
| `obj.wlength` | `[128 128 128 128 128 128 128];` |
| `obj.noverlap` | `[32 32 32 32 32 32 32];` |
| `obj.prew` | `[0 0 0 0 0 0 0];` |
| `obj.window` | `'dpss';` % window type |
| `obj.timebandwidth` | `5/2;` % |
| `obj.Nk` | `1;` % number of tapers for each decimation level |
| `obj.delayfilt` | `0;` |
| `obj.bandsetup` | `'MT'` % default band setup. Band set up definitons (which fourier coefficients are to be averaged for a single target frequency) are defined in `@EMSpectra\private\sp_defaultbs.m`. Modify this function to your needs. The default produces averaging rules for logarithimcally equidistant target frequencies with two target frequencies per octave. |
| `obj.bsfc` | `{};` |
| `obj.bsfcenter` | `{};` |
| `obj.usedec` | Intdex to define the current decimation level to be used. Default value is `1`. |
| `obj.fcrange` | Two-component index vector to define First and last index of the range of fourier coefficients. Default is `[]` and returns all frequencies. To return a subset only, set<br>`>> obj.fcrange = [<frst> <last>];` |
| `obj.setrange` | N-component index vector to define the range of sets (time windows). Default is `[]` and returns all sets. To return a subset only, set<br>`>> obj.setrange = [<indexvector>];` |
| `obj.input` | Cell array to define the input channels. Default is `{'Bx' 'By'}`. Note that channels are identified by their given name. Multiple channels as well as a single channels is allowed. One possible example is<br>`>> obj.input = {'Ex'};`<br>when the input channel is stored under the name `'Ex'`. |
| `obj.output` | Cell array to define the input channels. Default is `{'Bz'}`. Multiple output channels are allowed. |
| `obj.debuglevel` | Output some debugging information. Default value is `1`. |

The particular way of channel grouping into input and output channels is motivated by the later processing objectiv, where we will be interested to estimate (multi-)linear transfer functions $\mathbf{T}$ between some input channels $\mathbf{X}$ and output channels $\mathbf{Y}$.

## F.3 Private properties

`obj.global_headerlength 1024;`

`obj.channel_headerlength 1024;`

`obj.srate` sampling rate of the undecimated data

`obj.smprange` range of samples of the undecimated data relative to `reftime` corresponding to the time window given by `usetime` resp. `usesmp`

`obj.Nch` `0;` number of channels

`obj.tssource` `{''};` source directory for time series

`obj.chnames` `{''};`

`obj.chtypes` `{''};`

`obj.sens_name` `{''};`

`obj.sens_sn` `{''};`

`obj.calfile` `{''};`

`obj.caldata` `{''};`

`obj.srate` `0;` sampling rate of the undecimated data

`obj.Nsets` `0;` number of sets (windows) for each decimation level

`obj.Nf` `0;` number of requencies for each decimation level

`obj.W` `[1 2];` index of first and last window for each decimaimation level

`obj.T` `[0 0 0];` central time of first window in seconds since reftime and time % spacing between adjacent windows for each decimation level

`obj.F` `[1 1 0];` first and last frequency

## F.4 Dependent properties

`obj.sratedec` Sampling rates for each decimation levels

`obj.Nfc` `total number of fourier coefficients for each decimation level`

`obj.Y` Fourier coefficients of output channels at the given decimation level for the range of frequencies, sets and channels defined above. Invoking this dependent property triggers data reading from the file. The dimension of `Y` is `[Nfc,Nsets,Nk,Noutputch]`. For one output channel and a single taper spectrum, this reduces to `[Nfc,Nsets]`.

`obj.maskY` Logical mask for the same subset of coefficients, read from the fourier coefficient file.

`obj.X` Fourier coefficients of input channels. The dimension of `X` is `[Nfc,Nsets,Nk,Ninputch]`. For one input channel and a single taper spectrum, this reduces to `[Nfc,Nsets]`.

`obj.maskX` Logical mask for the same subset of coefficients, read from the fourier coefficient file.

`obj.F` Three-component vector containing the first and the last frequency (in Hz) of fourier coefficients as well as the spacing of frequencies for the current decimation level.

`obj.f` Vector of frequencies for the current decimation level (usedec) and the current frequency subset (fcrange)

`obj.w` Vector of of set indices relative to first set in file for current range of windows (setrange)

| | |
|---|---|
| `obj.wr` | Vector of of set indices relative to reftime the current range of windows (setrange) |
| `obj.trs` | central time of sets (time windows) for current selection of sets, relative to reftime, in seconds |
| `obj.trh` | central time of sets (time windows) for current selection of sets, relative to reftime, in hours |
| `obj.trd` | central time of sets (time windows) for current selection of sets, relative to reftime, in days |
| `obj.utc` | central time of sets (time windows) for current selection of sets, in utc time; use datestr to convert to readable format |

### F.5 Methods

2do: move plotting functions into EMRobustProcesing

add plotting of power spectra

need to check time axis

## G Class `EMRobustProcessing`

`EMRobustProcessing` performs multi-linear robust regression on fourier coefficients or spectra.

**2DO:** at the moment, the first argument is the frequency vector. remove this. add time axis. These two vectors are only needed for plotting.

### G.1 Initializing the object

The object is initialized with the command

```
>> obj = EMRobustProcessing(Y,X);
```

where `Y` and `X` are output and input channels of the linear system. Here, we understand that the values of `Y` and `X` are fourier coefficients, and that all values of `Y` and `X` are independent realizations of one random variable. In standard EM applications, we average over a range of frequencies to estimate the response function at one central (target frequency). Thus, the corresponding fourier coefficients can be averaged together. Useful band setup defintions are contained in the `bs` property of an `EMSpectra` object.

The data arrays `Y` and `X` are of dimension `[Nfc,Nsets,Nk,Nch]`, where `Nfc` is the number of fourier coefficients, `Nsets` the number of sets (time widnows in our application), `Nk` the number of tapers (for multi-taper spectral estimations), and `Nch` is the number of output and input channels, respectively. Hence the dimension of `Y` and `X` must be equal except for the number of channels, which can be different for the input and output. Note that it is only meaningful to combine synchronous sets (time windows). To combine multiple, overlapping recordings of the input and output channels (which is the standard for interstation transfer functions), the data must be synchronized (intersected) before.

A data mask can be provided during the initialization of the object. Let `useY` and `useX` be logical masks of the same dimension as `Y` and `X`, respectively, the command

```
>> obj = EMRobustProcessing(Y,X,useY,uesX);
```

sets these fields. `useY` and `useX` are public fields, i.e. these fields can be set also after intializing the object, with the commands

```
>> obj.useY=useY;
```

```
>> obj.useX=useX;
```

In addition to input and output channels, reference channels can be defined. The object is then initialized with the command

```
>> obj = EMRobustProcessing(Y,X,Xr);
```

where `Xr` are the corresponding data of the reference and which is the same dimension as `X`. A corresponding initialization that includes masks is

```
>> obj = EMRobustProcessing(Y,X,Xr,useY,useX,useXr);
```

The class supports additional asming or selection schemes based on the coherency of the channels. These masks will be overlayed with the `use*` mask with an logical `AND`. I.e., let the masking determined by some criteria be stored in, say, `maskY`, then only those realizations are used, for which

`useY & maskY`

is `TRUE`.

Other properties that are set during the initialization of the object are

| | |
|---|---|
| `obj.Nfc` | number of fourier coefs. |
| `obj.Nsets` | number of sets |
| `obj.Nk` | number of slepian sequences / or tapers |
| `obj.Noutput` | number of output channels |
| `obj.Ninput` | number of input channels |
| `obj.Nrefs` | number of of reference channels |

Although these fields are public, they are automatically initialized and should not be changed.

## G.2 Public properties

Public properties are as follows:

| | |
|---|---|
| `obj.f` | frequencies. Default is `f = [];` (property is optional and only used for plotting) |
| `obj.t` | central times of sets (time windows) in seconds relative to the start of the first window. Default is `t = [];` (property is optional and only used for plotting) |
| `obj.input` | cell array with names of input channels. Default is `input = {};` (property is optional and only used for plotting) |
| `obj.output` | cell array with names of output channels. . Default is `output = {};` (property is optional and only used for plotting) |
| `obj.Y` | output channel data (see above) |
| `obj.X` | input channel data |
| `obj.Xr` | reference channel data |
| `obj.useY` | predefined masking for output channels (see above) |
| `obj.useX` | predefined masking for input channels |
| `obj.useXr` | predefined masking for reference channels |
| `obj.huber` | Huber constant for robust regression. Default is `huber=1.5;` *See Methods for details on the regression.* |
| `obj.reg` | Perform regression on fourier coefficients directly `'fc'` or on spectra `'spectra'`. Default is `reg='fc';` *See Methods for details on the regression.* |
| `obj.smooth` | Method for spectral smoothing. Default is `smooth='runav';` Options are<br>`'none'` no smoothing<br>`'smoothn'` uses the smoothing function `smoothn` from Garcia, 2010; it requires the *MATLAB Image Toolbox*. The `smoothn` method interpretes the `avrange` as the preset smoothing factor. If `avrange = []`, the smoothing factor is to be determined automatically. Type `>> help smoothn` for details and more options. May be very slow for large data sets.<br>`'runav'` (2D running average) |

`obj.avrange` Averaging domain / smoothing factor for smoothing of spectra. Default is `avrange=[2 5]` and is valid for the `runav` option. For this option, the vector contains the number of adjacent fourier coefficients and sets (time windows) which are being averaged to smooth the spectra. For the `smoothn` option, `avrange` is a single number, which is interpreted as the smoothing factor for `smoothn`.

`obj.bicohthresg` `{[min max],[],...}` Global bivariate coherency thresholds, applied to individual pixels of the `Nfc x Nsets` bivariate coherency image. Minimum and maximum coherencies are defined separately for each channel of the output channels. Default is empty;

`obj.bicohthresf` `{[min max],[],...}` Noutput global bivariate coherency thresholds, applied to time averaged coherencies to pull out individual frequencies of the `Nfc x Nsets` bivariate coherency image. Minimum and maximum coherencies are defined separately for each channel of the output channels. Default is empty;

`obj.bicohthrest` `{[min max],[],...}` Noutput global bivariate coherency thresholds, applied to frequency-averaged coherencies to pull out individual sets (time windows) of the `Nfc x Nsets` bivariate coherency image. Minimum and maximum coherencies are defined separately for each channel of the output channels. Default is empty;

`obj.unicohthresg` `{[min max],[],...}` Noutput global univariate coherency thresholds, applied to individual pixels of the `Nfc x Nsets` univariate coherency image. Minimum and maximum coherencies are defined separately for each channel of the output channels. Default is empty;

`obj.unicohthresf` `{[min max],[],...}` Noutput global univariate coherency thresholds, applied to time-averaged coherencies to pull out individual frequencies of the `Nfc x Nsets` univariate coherency image. Minimum and maximum coherencies are defined separately for each channel of the output channels. Default is empty;

`obj.unicohthrest` `{[min max],[],...}` Noutput global univariate coherency thresholds, applied to frequwncy-averaged coherencies to pull out individual sets (time windows) of the `Nfc x Nsets` univariate coherency image. Minimum and maximum coherencies are defined separately for each channel of the output channels. Default is empty;

`obj.tfthres` `{[zr+1i*zi, radius, iinput],[],...}` include points within cirle drawn by radius around point zr+1i*zi in the complex plane. iinput indexes the input channel, i.e. let the two output channels be Ex and Ey, then Zyx is masked with the following syntax `{[],[zr+1i*zi, radius, 1],` where 1 refers to Hx

`obj.debuglevel` Output some debugging information. Default value is 1.

## G.3 Dependent Properties

`obj.maskY` logical mask for otput channels, made dependent on above thresholds. `maskY` is overlaid with `useY` (with logical `AND`: `maskY = maskY & useY`). Note that for mathematical operations, `maskY` must be converted to double precision, e.g. by `maskY*1.0`. Dimension of `maskY` is the same as the dimension of `Y`.

`obj.maskX` same as bove, but for input channels `X`.

`obj.maskXr` same as bove, but for reference channels `X`.

`obj.Ym` `Y` with masked entries in `maskY` set to `NAN`

`obj.Xm` `X` with masked entries in `maskX` set to `NAN`

`obj.Xrm` `Xr` with masked entries in `maskXr` set to `NAN`

`obj.YmN` `Ym`, reshaped to `N x Noutput` matrix, with `N = Nfc x Nsets x Nk`

`obj.XmN` `Ym`, reshaped to `N x Ninput` matrix, with `N = Nfc x Nsets x Nk`

`obj.XrmN` `Ym`, reshaped to `N x Nrefs` matrix, with `N = Nfc x Nsets x Nk`

`obj.YY` Spectral matrix of output channels `Y`. For `Noutput=1`, this reduces to `Y.*conj(Y)`. For multiple output channels, the spectral matrix is computed as follows. Let $\mathbf{Y}_{1,\ldots,N}$ be the $N$ channels, then the spectral matrix is given by

$$\mathbf{Y}\mathbf{Y}^* = \begin{bmatrix} \mathbf{Y}_1\mathbf{Y}_1^* & \mathbf{Y}_1\mathbf{Y}_2^* & \cdots & \mathbf{Y}_1\mathbf{Y}_N^* \\ \mathbf{Y}_2\mathbf{Y}_1^* & \mathbf{Y}_2\mathbf{Y}_2^* & & \\ \vdots & & \ddots & \\ \mathbf{Y}_N\mathbf{Y}_1^* & & & \mathbf{Y}_N\mathbf{Y}_N^* \end{bmatrix} .$$

and stored in the four-dimensional array `YY` with dimensions `Nfc x Nsets x Nk x Noutput^2`, where the order of storage for the last dimension is row-wise:

$$\mathbf{Y}\mathbf{Y}^* = \begin{bmatrix} \mathbf{Y}_1\mathbf{Y}_1^* & \mathbf{Y}_1\mathbf{Y}_2^* & \cdots & \mathbf{Y}_1\mathbf{Y}_N^* & \mathbf{Y}_2\mathbf{Y}_1^* & \mathbf{Y}_2\mathbf{Y}_2^* & \cdots & \mathbf{Y}_N\mathbf{Y}_1^* & \cdots & \mathbf{Y}_N\mathbf{Y}_N^* \end{bmatrix} .$$

The spectral combination $\mathbf{Y}_i\mathbf{Y}_j^*$ of channels $i$, $j$ are thus indexed in the fourth dimension with `j + (i-1) x Noutput`, and in particular for the autopowers $\mathbf{Y}_i\mathbf{Y}_i^*$, the index reads `i + (i-1) x Noutput.`

`obj.XX` Spectral matrix of input channels `X`. Otherwise same as for `YY.`

`obj.XXr` Spectral matrix of input channels `Xr`. Otherwise same as for `XX.`

`obj.YX` Cross-spectral matrix between output channels `Y` and input channels `X.` For `Noutput=1` and `Ninput=1,` this reduces to `Y.*conj(X).` For multiple output and input channels, the spectral matrix is computed as follows. Let $\mathbf{Y}_{1,\ldots,N}$ be the $N$ output channels and $\mathbf{X}_{1,\ldots,M}$ be the $M$ input channels , then the $N \times M$ cross-spectral matrix is given by

$$\mathbf{Y}\mathbf{X}^* = \begin{bmatrix} \mathbf{Y}_1\mathbf{X}_1^* & \mathbf{Y}_1\mathbf{X}_2^* & \cdots & \mathbf{Y}_1\mathbf{X}_M^* \\ \mathbf{Y}_2\mathbf{X}_1^* & \mathbf{Y}_2\mathbf{X}_2^* & & \\ \vdots & & \ddots & \\ \mathbf{Y}_N\mathbf{X}_1^* & & & \mathbf{Y}_N\mathbf{X}_M^* \end{bmatrix} .$$

and stored in the four-dimensional array `YX` with dimensions `Nfc x Nsets x Nk x (Noutput x Ninput)`, where the order of storage for the last dimension is row-wise:

$$\mathbf{Y}\mathbf{X}^* = \begin{bmatrix} \mathbf{Y}_1\mathbf{X}_1^* & \mathbf{Y}_1\mathbf{X}_2^* & \cdots & \mathbf{Y}_1\mathbf{X}_M^* & \mathbf{Y}_2\mathbf{X}_1^* & \mathbf{Y}_2\mathbf{X}_2^* & \cdots & \mathbf{Y}_N\mathbf{X}_1^* & \cdots & \mathbf{Y}_N\mathbf{X}_M^* \end{bmatrix} .$$

The spectral combination $\mathbf{Y}_i\mathbf{X}_j^*$ of channels $i$, $j$ are thus indexed in the fourth dimension with `j + (i-1) x Noutput.`

`obj.XY` Cross-spectral matrix between input channels `X` and output channels `Y.` For `Ninput=1` and `output=1,` this reduces to `X.*conj(Y).` For multiple input and output channels, the spectral matrix is computed as follows. Let $\mathbf{X}_{1,\ldots,M}$ be the $M$ input channels and $\mathbf{Y}_{1,\ldots,N}$ be the $N$ output channels , then the $M \times N$ cross-spectral matrix is given by

$$\mathbf{X}\mathbf{Y}^* = \begin{bmatrix} \mathbf{X}_1\mathbf{Y}_1^* & \mathbf{X}_1\mathbf{Y}_2^* & \cdots & \mathbf{X}_1\mathbf{Y}_N^* \\ \mathbf{X}_2\mathbf{Y}_1^* & \mathbf{X}_2\mathbf{Y}_2^* & & \\ \vdots & & \ddots & \\ \mathbf{X}_M\mathbf{Y}_1^* & & & \mathbf{X}_M\mathbf{Y}_N^* \end{bmatrix} .$$

and stored in the four-dimensional array `YX` with dimensions `Nfc x Nsets x Nk x (Ninput x Noutput)`, where the order of storage for the last dimension is row-wise:

$$\mathbf{X}\mathbf{Y}^* = \begin{bmatrix} \mathbf{X}_1\mathbf{Y}_1^* & \mathbf{X}_1\mathbf{Y}_2^* & \cdots & \mathbf{X}_1\mathbf{Y}_N^* & \mathbf{X}_2\mathbf{Y}_1^* & \mathbf{X}_2\mathbf{Y}_2^* & \cdots & \mathbf{X}_M\mathbf{Y}_1^* & \cdots & \mathbf{X}_M\mathbf{Y}_N^* \end{bmatrix} .$$

The spectral combination $\mathbf{X}_i\mathbf{Y}_j^*$ of channels $i$, $j$ are thus indexed in the fourth dimension with `j + (i-1) x Ninput.`

`obj.YYs` smoothed version of spectral matrix `YY`. Smoothing options are set in the properties `obj.smooth` and `obj.avrange.`

`obj.XXs` smoothed version of spectral matrix `XX`.

`obj.XXrs` smoothed version of spectral matrix `XXr`.

`obj.YXs` smoothed version of spectral matrix `YX`.

`obj.XYs` smoothed version of spectral matrix `XY`.

`obj.YYm` smoothed version of spectral matrix `YYs` with masked data set to `NAN.`

`obj.XXm` smoothed version of spectral matrix `XXs` with masked data set to `NAN.`

`obj.XXrm` smoothed version of spectral matrix `XXrs` with masked data set to `NAN`

`obj.YXm` smoothed version of spectral matrix `YXs` with masked data set to `NAN`

`obj.XYm` smoothed version of spectral matrix `XYs` with masked data set to `NAN`

`obj.YYmN` `YYm` reshaped to `N x (Noutput x Noutput)` matrix, with `N = Nfc x Nsets x Nk`

`obj.XXmN` `XXm` reshaped to `N x (Ninput x Ninput)` matrix, with `N = Nfc x Nsets x Nk`

`obj.XXrmN` `XXrm` reshaped to `N x (Ninput x Ninput)` matrix, with `N = Nfc x Nsets x Nk`

`obj.YXmN` `YXm` reshaped to `N x (Noutput x Ninput)` matrix, with `N = Nfc x Nsets x Nk`

`obj.XYmN` `XYm` reshaped to `N x (Ninput x Noutput)` matrix, with `N = Nfc x Nsets x Nk`

`obj.unicoh` univariate coherencies; only available for `Ninput = Noutput = 1.` ***Must be generalized!***

`obj.bicoh` variate coherencies only available for `Ninput = 2;` Dimension of `bicoh` is `Nfc x Nsets` for `Noutput = 1` and `Nfc x Nsets x Noutput`, where the order of the third dimension corresponds to the order of the output channels.

`obj.tfs` lsq transfer functions for each estimate of the smoothed spectra

`obj.leverage` ***not yet implemented;*** should come along with `tfs`.

`obj.resid` ***not yet implemented;*** should come along with `tfs`.

### G.4 Methods

`[Z,Zse] = computetf(obj);` Robust transfer function estimation.
`reg = 'fc';` The function performs a robust M-estimate of transfer functions an iterative least squares ***Must be generalized!***
`reg = 'spectra';`

## H Format of `*.mtd/*.txt` files

## I Format of `*.ats` files

## J Format of `*.afc` files

Fourier coefficients for multiple channels and multiple decimation levels are stored in a common binary `*.afc` file. The file is organized in header sections and data blocks. A global header contains information about the station, the number of channels and the decimation scheme, and channel headers conatin channel-specific information. The format is as follows:

| | |
|---|---|
| `Global header` | starts at bof; length of this header is the first number in file |
| `1 x int16` | `global_headerlength` length of global header |
| `1 x int16` | `channel_headerlength` length of channel header |
| `1 x int16` | `Nch`, number of channels |
| `1 x float32` | sampling rate |
| `1 x float32` | `reftime` of survey, use `datevec` to convert |
| `1 x int16` | `Ndec`, number of decimation levels |
| `Ndec x int16` | `decimate`, decimation factors |
| `Ndec x int16` | `wlength`, window lengths in number of samples |
| `Ndec x int16` | `noverlap`, overlap of adjacent windows in number of samples |
| `Ndec x int16` | `prew`, prewhitening option |
| `4 x char*1` | window type, can be `'hann', 'hamm', 'dpss'` , 'rect' |
| `1 x int16` | `Nk`, number of tapers |
| `1 x int16` | `timebandwidth`, for dpss windows |
| `1 x float32` | delay filter frequency |
| `1 x int16` | `Ndnts` number of characters in diecrtiry name of time series |
| `Ndnts x char*1` | `tssource` directory name of time series |

The following block `Ndec` times, one for each decimation level:

*`for idec = 1:Ndec`*

| | |
|---|---|
| `3 x int32` | `[Nfc Nsets Nk]` size of data in the current decimation level |
| `2 x int32` | `[W1 WN]` window ids of first and last window with respect to reftime; |
| `3 x float32` | `[T1 TN dT]` central time of first window in seconds since reftime, central time of last window in seconds since reftime, and time interval between to adjacent windows |
| `3 x float32` | `[F1 FN dF]` frequency of first fourier coefficient, frequency of last fourier coefficient, frequency increment |

*`end`*

`<reserved>`

---

| | |
|---|---|
| `channel 1` | starts at `global_headerlength+1` bytes; |
| `2 x char*1` | channel name of 1st channel, e.g. `'Ex'`, ... |
| `2 x char*1` | channel type, e.g. `'Ex'`, ... |

`<reserved>`

---

`data block for channel 1` starts at `global_headerlength+global_headerlength+1` bytes; the folllowing block is repeated for each decimation level, and if there are more than one taper (i.e. multi-taper spectra), the data block containing the fourier coefficients is repeated for each taper.

*for idec = 1:Ndec*

*for ik = 1:Nk*

`Nfc x Nsets x float32` real part of fourier coefficients at level `idec` and for taper `ik`

`Nfc x Nsets x float32` imaginary part of fourier coefficients

*end*

`Nfc x Nsets x uint8` binary mask

*end*

---

| | |
|---|---|
| `channel ich` | starts at `global_headerlength+(ich-1)*channel_headerlength+(ich-1)*Nfc*9+1` bytes; `Nfc` is the total number of Fcs, summed over all decimation levels `Ndec`. This number multiplies by `9 byte`, four for each real and imaginary part and `1 byte` for a binary mask |
| `2 x char*1` | channel name of 2nd channel, e.g. `'Ey'`, ... |
| `2 x char*1` | channel type, e.g. `'Ey'`, ... |
| `1 x int16` | `Nfn,` length of file name of time series |
| `Nfn x char*1` | filename of time series |

---

`data block for channel ic` starts at `global_headerlength+ich*channel_headerlength+ (ich-1)*(Nfc*8+Nfc/kd)` bytes;

...

**References:**

Garcia, D., **2010**. Robust smoothing of gridded data in one and higher dimensions with missing values. Computational Statistics & Data Analysis. *http://www.biomecardio.com/pageshtm/publi/csda10.pdf*