**Thoughts and details on MT Data Processing Software – Gary Egbert**

I made a new graphical overview, which summarizes what is now implemented in EMTF (Egbert Fortran processing codes), informed also by efforts to develop processing in matlab. The upper (light blue) boxes represent the main “objects” that need to be implemented, and the main steps in routine processing. In reality the object structure may be more granular--that is probably a programming decision. Beneath the 4 main objects are the most important support objects. Most of these will be quite simple, but I think there are good reasons to make them distinct objects. The support objects are organized under the upper level object that they are first associated with--but in most cases they are (or at some point may be) used in other main objects. The lower row represents a few additional modules/objects which will be useful in practice. For example for a quick look at data quality, a user may run through the upper level, with some default setting of options, ending with a quick initial plot of results. If those look good, that may be it (especially for in-field processing). But if there are quality issues, the first step would be to plot the time-series, to get some idea of what the issues are. This may lead to various actions -- marking time series to omit some sections, modifying processing options, doing RR processing with several candidate sites, etc. Interactive tools (which would be embedded in the two boxes in the lower row) would facilitate this. All of the blocks here can be related to existing Fortran programs (or editable input/control files) in EMTF, and/or to matlab functions, scripts, or classes. I will use this breakdown into components in all of the following discussions, where I provide further details on the functionality, and some suggested general implementation requirements for each of these high-level objects.

A screenshot of a cell phone

Description automatically generated

Sidenote: MTpy (<https://github.com/MTgeophysics/mtpy>) is capable of displaying transfer functions for QC. Use of the covariance matrix and proper rotation of the errors would need implementation, but it is easily expandable. Plotting, simple analysis of strike and dimensionality, depth of penetration, simple bostick estimation, etc. are all available.

**Matlab Prototype (Prototype9-15-20.tar):** After starting to write this I decided to make simple matlab prototypes of the basic classes—actually writing programs helps clarify some issues for me. Ideally, we would have working copies of these for the programming contractor. The idea would be for these to define functionality, more than details of implementation. But it would give more precise targets in terms of what should be implemented and could help with testing and debugging. In fact, I have so far written most of the code, and debugged to the point where basic tests run—but do not really work yet. There are still bugs, and many features/scenarios that are untested. I do have working matlab programs that have similar functionality; this has mostly been about reorganizing and developing a relatively simple and clean set of routines that would be easier to understand (I think). At any rate, I believe that the prototype code provides a clearer idea of what I am suggesting immediately. I will continue debugging as time allows, and could provide updates to the contractor, and answer questions. I should also say that I am only implementing the critical components. Some additional features (and things like I/O, user control, etc.) may require input/specification from others.

**Note on timing:** We need to keep track of time throughout the processing. I believe there is a “datetime” class that can be used with Python, that can represent time to the microsecond. But there may be other IRIS preferences. I am going to assume that everything that needs to be tagged as to time can use a “datetime” to do this (this is what is done in the prototype). Every “survey” (set of sites to be processed together) will need a reference datetime which I call “clock\_zero”, which will come into play when we do the windowed FT, but it is best to have this attached to the time series also. (Sidenote: The python datetime class may be impractical due to memory requirements. Numpy’s datetime64 may be better option. Possible approaches are covered in <https://jakevdp.github.io/PythonDataScienceHandbook/03.11-working-with-time-series.html>.)

The following details may sometimes be repetitive, since this was edited several times and not completely cleaned up.

**Time Series (TTS) class:** The basics are pretty straightforward, and I have sketched an implementation: a TTS object has an array of data and key metadata. (Sidenote: The xarray package 1) reads from memory instead of RAM, so very efficient, 2) metadata is directly attached, 3) can be multidimensional and indexed by time, and 4) you can have a collection of datasets. This fits the requirement of metadata as well, as that can directly be passed on through the processing and therefore no other files are needed. This is implemented in MTH5 and MTpy.) Two points about the data: First, it is essential that TTS support multiple channels. If you want to use it with one channel, you can still do that! Second, we need to allow for burst sampling, where we want to store a series of data blocks. I think the best way to do this is to have an array (list, or whatever is appropriate to Python) of data blocks. In the simplest case of continuous sampling there would be one block. Each block would have a start time and the data. I would suggest allowing for small gaps marked by missing data) within any block. Note that this feature (allowing for multiple blocks) is not explicitly allowed for in my working Matlab codes, since I have been working with long period data, but is intrinsic to the new prototype. Thus, I have two basic classes: **TSbloc**k, which stores one block of time series (a series of contiguous data samples, with fixed sample rate dt, and time of first sample startTime.) Gaps within a block are denoted by a missing value code (in the prototype, always NaN – mostly hardcoded). Then the **TTS** object contains an array of **TSblock** objects. There are some basic filters, and other methods, like plus, copy, etc. defined in each of these classes (with the TTS versions mostly defined in terms of the corresponding TSblock methods).

A screenshot of a cell phone

Description automatically generated

*Metadata*: I think that the metadata carried here can be a subset of what is in the archive (or distribution) metadata: basically, need some sort of site ID, start time, zero time, dt, channel azimuths (and tilts), system response information. But of course, we want to be able to put information such latitude and longitude, and perhaps many other things into the final TF output file, so it is best to carry these through all processing steps. It might be simplest to just carry the full metadata, perhaps as a structure embedded in the “Site Header” object. My idea is that basically this class has to read in whatever is available and set this into variables with names that subsequent codes can access—for data in the standard format this would be easy (and is all would be done in this project), but if someone else wanted to use the code for processing, they could write a subclass to ingest their format and set appropriate variables. Maxim Smirnov and I had a TDataHeader class that we used for many things; he has apparently modified to work more seamlessly with JSON. In prototype codes I created a series of Header objects (which so far only contain information that is essential to processing code functions). The headers are hierarchical: headers for data channels (**TCHannelHeader.m**, **TMagneticChannel.m**, **TElectricChannel.m**), then for sites (**TSiteHeader.m**; these contain an array of channel headers); and for “arrays” or groups of sites (**TArrayHeader.m**; these contain an array of site headers—either one or two for the routine processing considered here, but could be extended).

*Channel Response:* This somehow comes from the metadata, but I think it should be singled out and made into a distinct class. I think there should be one instance for each channel. The key method for this object would be a function that returns the complex response at a fixed frequency, (the number to multiply the data by to get the desired physical/practical units), or perhaps directly apply this to the frequency domain data. The class could be polymorphic, so for example if a table describing the full response were provided a very simple (table look up/interpolation) would be the implementation. But the calculation could be done in terms of poles and zeros, and multiple stages could be included, in a different subclass. But some program will have to get whatever is in the metadata and translate this into something that a generic program can interpret correctly (e.g., create Channel Response objects of the proper type, populate tables or parameters, based on metadata). We have to decide how general this should be (or not) for this development. I suggest tailoring to however system responses are stored in metadata. As a starting point I made a very simple table look-up class, but we might want more, even immediately.

*Marked TS Sections*: Sometimes we will want to mark sections of data to omit from processing. Bad data can be marked using the Matlab TSplot program, and I think this is a capability that we want to maintain. (Sidenote: xarray, numpy, and pandas can mask arrays such that unwanted data is not used, or the data array could be duplicated with masks to preserve the original.)

In principle, sections could be marked for other specialized uses. I think we should build this capability in from the start. Basically, there will be an array of marked sections; each section has two datetimes (start, end) and a list of channels that the mark applies to (e.g., if the Ex wire is cut, marks might apply to only that Ex.). Possibly multiple arrays of marks could be stored with the TS object—haven’t thought about this. (NOTE: this is supported in EMTF, but there are many little problems – stored in separate files (not with TS), ambiguous time information, etc.). I have not worked on a cleaned version of this yet.

*Methods:* In addition to methods to create the TS object, read data, and create/initialize Channel Response objects, a number of basic time series tools might be useful: decimation, merging several TS objects (different time segments, different sets of channels, filtering, spike removal, etc. Methods can be added later, but we should discuss what set of basic things we might want to develop now. Nothing much is absolutely necessary; lots of things might be useful. Since the actual TS is carried in a series of blocks, a few methods that I did write so far (decimate, high-pass filter, copy, zero, plus) are first implemented in the “TSblock” class, and the method in the upper level TS class just loops over the array of blocks.

**Fourier Transformation**: This implements the algorithm that I am suggesting. The basic idea is similar to what I use in my old dnff Fortran code: there are a series of “decimation levels” – at the first level short overlapping windows are windowed and Fourier transformed. At each subsequent level longer time windows are used, to get information about lower frequencies. The idea is to use the shortest time windows possible given the desired frequency, and spectral resolution. MT responses are very smooth in frequency, so low frequency resolution (short time windows) typically suffices. Keeping good time resolution can help in sorting out bad data in the frequency domain—a longer time window may mix good and bad segments. Of course, to go to lower frequencies, longer time windows are needed; hence the multiple decimation levels. I refer to the different levels as “decimation levels” because historically we decimated the TS before going to longer windows (so the number of points in each window stays the same, but the length of the window in seconds increases). However, it is of course not necessary to actually decimate to get lower frequencies, just have to use longer time windows. The basic steps at the outermost level (class **TSTFT.m** – STFT for “short time FT”) are as follows:

for idec=1:Ndec

HP filter TS – overwrite TS with the residual (LP filtered TS)

FT the HP filtered TS, using windows Win(idec)

Decimate LP filtered TS (residual)

end

Save final LP filtered TS as “residual”

(Sidenote: The FT and Decimation steps are candidates for being implemented with multiprocessing, for instance using the Python program Dask.)

A screenshot of a cell phone

Description automatically generated

Additional filtering steps (e.g., pre-whitening) might be added; filter corrections could be here or at an inner level. Note that all of these steps are executed block-by-block. NOTE: in my prototype I am using the residual (TS – HP(TS)) as the low-passed TS. To replicate what is in dnff (where the same frequency content is contained in two adjacent decimation levels) the LP filter would have to have to cutoff at a higher frequency than the HP filter, and so a separate LP filter step would have to be applied to the raw time series. An LP filter comparable to that used for dnff is implemented in the prototype, so this just requires minor reorganization of the above (pseudo-code) loop.

In addition to a method that implements this FT step, there is a method that can use the saved FCs and residual back to reconstruct the original TS. This has been implemented, but not yet tested in the protype. This feature has been used in an older matlab program and will probably be tested soon. This is not used in the routine processing that the development should focus on.

The next level down handles FT and filtering steps for one decimation level (**TFC1Dec.m**). These are executed block-by-block. The FCs saved are in a single array (for each decimation level), merging over all TS blocks. The FC array is thus of dimension FC(Nch, Nfc, Nseg). Nfc is the number of FCs saved out of the full set. Because of the cascade of filters there ends up being a narrow range of FCs that have significant amplitude—especially if there is no decimation—so we allow the range of FCs to be saved to be specified. Nseg is the number of overlapping time windows, concatenated over all blocks. We keep track of the range of segments that are derived from each block. A key idea is that absolute time is converted to “segment number”, where the first segment starts at clock\_zero and the window properties (in particular the shift between successive segments) define the start points for allowable sets. All of this is managed (quite cleanly now, but not in the old version) through the **TWindow** class. This class also is key to merging and aligning FCs for a series of runs (at one site) and for two (or more) sites. The resulting array of FCs provides the starting point for transfer function estimation, using either single station or remote reference. (These routines also can be used to create arrays of FCs for more than two sites, for multi-station processing methods, that are not part of the immediate development.) Details are discussed further under Transfer Function Estimation. Methods used for merging and aligning TFC1Dec objects into arrays needed for TF estimation are in TFC1Dec. Algorithms are a bit obtuse, because I was trying to be general and efficient.

***TWindow class***: For each decimation level there is a TWindow object. This defines the length of the window, overlaps, segment shifts, and the actual window coefficients. It also defines the HP (and LP) filter scheme. By keeping track of the clock\_zero and sampling interval for the decimation level (properties of the object) methods in this class can convert between the “local time” of set number and absolute time. Knowing also the startTime for a block, we can also convert between sample number within the block and real time, find start points of sets, assign numbers, etc. So basically, time is kept track of in this class. This is a property of the FC object so FC segment numbers can always be mapped to UT. Cleaning this up is probably the main thing I did with the prototype. Seems to work but needs further testing.

***TDecimate class***: This is pretty trivial.

**Transfer Function (TF) Estimation**: Inputs to this step are STFT objects for one or two sites, depending on whether processing is single station or remote reference. Multiple “runs” might be available for each site. Additional inputs include (1) a list of frequency bands for which estimates will be computed and (2) various options which control how the estimator works. In typical usage both of these would be standard, provided in a file, but different types of data would require different control files (e.g., frequency bands would depend on the sampling rate of the instrument—a wide band system that has multiple data streams, each with different sample rates, would require different frequency band files for each sampling band), and an experienced user might want to adjust various control parameters to try to improve processing results. So, these should be coded as inputs that allow flexible use of the code. Following is an overview of my Matlab prototype implementation.

A screenshot of a cell phone

Description automatically generated

***TSTFTarray class***: Although this can be generalized, to start it should suffice to consider only frequencies from a single “decimation level” for any single band. To keep this simple a frequency band is then be specified by the decimation level (iDec, say) and the band limits (iFreq1, iFreq2), where iFreq1, iFreq2 are the frequency numbers from the FT at that decimation level (if T is the length of the window used in the STFT at this decimation level (in seconds), frequency #n is n/T Hz. The two integers specify the range of frequencies to be averaged over in the TF estimation. The first step in the processing is to read in STFT objects for all runs/sites (function ReadTranMTcfg.m – this just reads the same file that is used by the fortran code tranmt.f). This information is used to create the instance of the TSTFT class, which loads and stores all of the STFT objects, and to load the list of frequency/decimation level bands that estimates should be computed for.

Then, looping over bands, the appropriate range of frequencies, for the appropriate decimation level, is extracted from these objects, and merged/aligned over runs, then over sites. The result can be stored naturally as a TFC1Dec object—i.e., an array of FCs and metadata for multiple channels (and multiple blocks), for a single decimation level, together with metadata. Thus, a series of methods are defined in the TFC1Dec class that (1) extract the subset of frequencies (iFreq1, iFre2) for the band (TFC1Dec.extractBand); (2) combine FCs and segment numbers for all runs at each site (TFC1Dec.mergeRuns); (3) combine data from all sites (assuming there is more than one, for remote reference; TFC1Dec.mergeSites). The result is an array of FCs for all channels from all runs, and all sites, aligned with respect to time, but only for the range of frequencies in the estimation band. This TFC1Dec object is stored in the TSTFTarray object. The implementation suggested is general (should work for any number of sites). If any site in the input list has data for a segment (time window) this segment is included in the array. If no data are available for some channels (typically from all channels from a site) the array contains a missing value code (NaN in the Matlab code) for this segment. The array is organized as a series of Blocks, just as for the input objects. Each Block contains a contiguous series of segment numbers, with gaps (for all sites) resulting in new Blocks. The key data for TF estimation produced here is the FC array, FC(Nch,Nfc,Nseg) where Nch is the total number of channels (for RR estimation starting from two 5 channel MT sites, Nch=10), Nfc = Ifreq2-Ifreq1+1 is the number of frequencies in the averaging band. And Nseg is the total number of time segments where any data are available for either of the two sites. Obviously, for single site processing Nch=5 (and there is no merging of sites). As implemented, this class also contains a final method, which extracts the local H and E channels, and the reference (R) channels when the remote reference approach is used. Header classes (mentioned above under TS) are used to define and keep track of channel types/sites needed for this.

***TTF class***: For storing transfer function estimates, with errors, for a series of frequency bands.

***Regression Classes:*** These define the actual estimators, which are always based on some form of multivariate regression. We give as an example an abstract class (TRegression), with a series of subclasses TRME, TRME\_RR, TSiegel. TRME is the regression M-estimate, appropriate for single site processing; TRME\_RR is the generalization of this to RR. These are the basic work-horses that would be used for standard processing. Inputs to these regression classes are the arrays of input variables (H), output variables (E), and for RR reference variables (R). There are a number of parameters that control implementation details, which are provided through the iteration control object (IterControl). TSiegel is an example of a different robust estimator, based on the approach of Smirnov. The implementation here does not support RR but this could easily be added. The main point of including this is to demonstrate that alternative estimation schemes could easily be added, for example the scheme based on maximum likelihood for the stable distribution suggested by Chave. The first two subclasses should be implemented as a starting point, with the idea that others could extend to allow alternative estimators to be used if desired.

***Weighting:*** There are other schemes for defining weights which can be input to the regression estimators. Leverage control is an example, implemented (following what is in tranmt) via function EdfWts.m. Coherence weighting is another example, that has proven somewhat useful in the routine processing of EarthScope TA LPMT data (not yet implemented in matlab). Other schemes (based on polarization, uniformity of magnetic fields from two sites, and more) are possible. Again, the idea would be to make these as independent add-on modules that could be easily inserted into the processing stream.

***MTTFdriver:*** This is a simple Matlab script provided as an example of how the transfer function estimation steps fit together, following the steps outlined above, and illustrating the use of the various classes. The starting point is a simple input file listing FC files (assumed to be stored as STFT objects, stored in mat files), and the end point is a TTF object, containing the TF estimates and error covariances.