FitLike2

User manual

# Purpose

FitLike aims at facilitating the coarse analysis of relaxometry data obtained from the Stelar relaxometers. It allows importing raw data files, selecting how to process it at each step of the data processing steps and to fit the dispersion curve with models that can be added together and visualised separately. It also allows saving the data obtained and exporting relevant measurement for further statistical analyses. For further work with relaxometry data, one can also see fiteia (<http://fitteia.org/>).

FitLike is available on Gihub under the GNU GPL license at <https://github.com/ManuIdentiFY/FitLike2>.

# Basic concepts

FitLike re-uses the concept developed by Stelar for the data handling. Data acquisitions are organised by blocs, which correspond to one acquisition of several data points such as an FID or an echo train. Blocs are grouped by zones, which are used to determine T1 at a given field. Finally, zones are grouped into dispersions, which describe how T1 or R1 vary with the magnetic field.

Note that all the figures provided in by FitLike are in International Units System, as far as possible.

# Principles of operation

FitLike is made of three components:

- data storage: it contains all the classes required to store the data generated during use. These may be data blocs, zones or dispersions. These are passive classes, as far as possible.

- data controller: it contains the classes that perform the data processing and conversions. This may be the algorithms that convert blocs into zones, zones into dispersions, or fit the dispersion curves. It also regroups the fit models.

- data viewer: this contains all the code required for the GUI. The viewer classes do not process the data, but only disply them and help handling and controlling the data processing flow.

Relaxometry data processing is usually performed by stages. The first one consists in importing the raw data into blocs and to sort them by type of experiments. Then the blocs are converted into zones by selecting the appropriate algorithm, which depends on the type of acquisition. The zone objects can then be regrouped and processed into dispersion objects, once more by selecting the appropriate algorithm. Finally, the dispersion objects can be analysed by custom-made models that can be build from the GUI.

All the algorithm structures have been designed in such a way that it is easy to create your own processing class with relatively little knowledge of Matlab. Template classes have been written that can be copied and modified into your own algorithm, which should then appear from the GUI. If you write your own code, please consider sharing it via the Github website or email it to the developers to help the community.

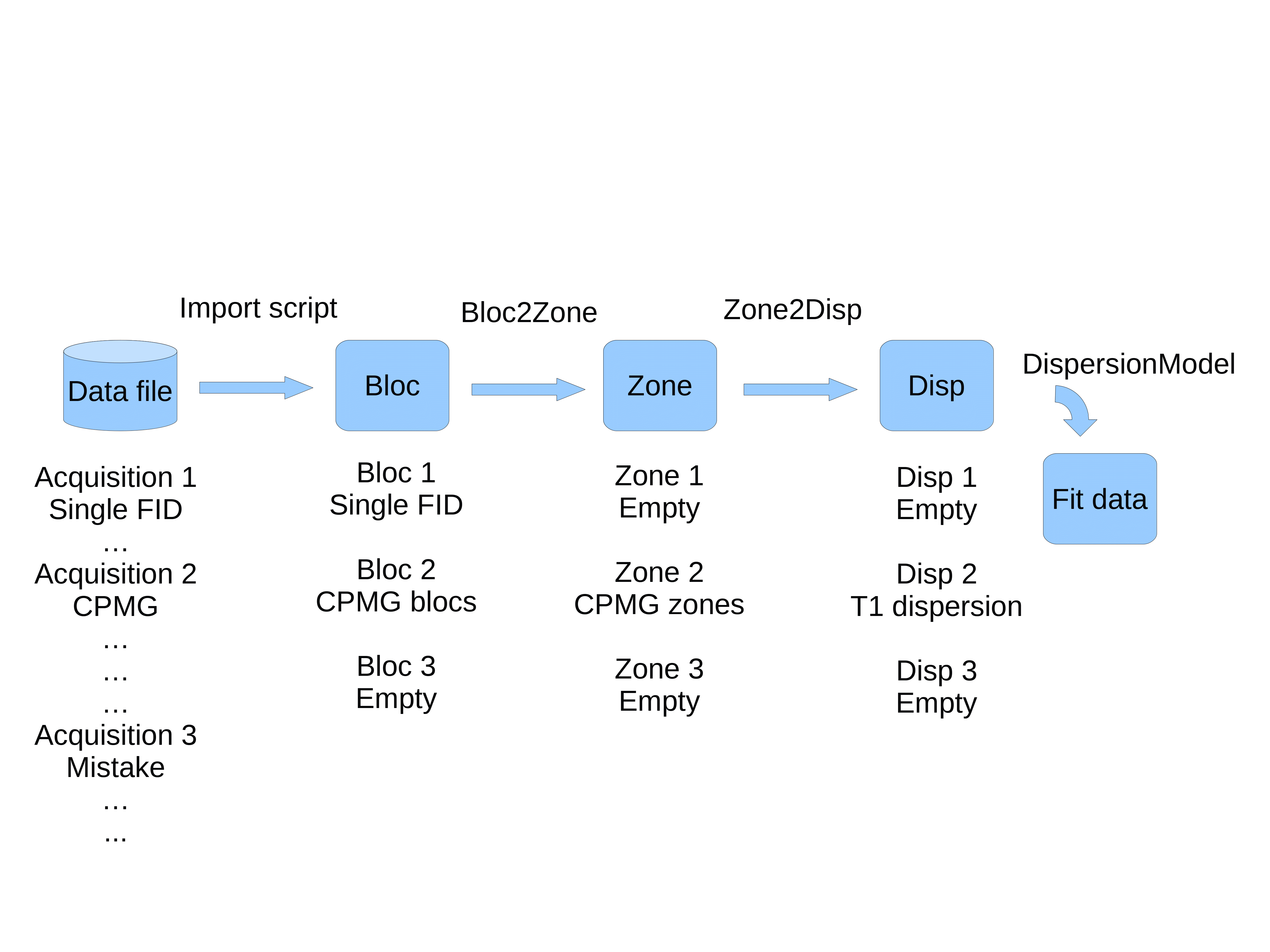
## Data structure

The data classes are organised as follow:

- DataUnit: parent class at the origin of all the other data classes. It handles the basic operations and defines how the data is stored. For simplicity of coding, any data object stores a 1-dimensional dataset in the fields x and y. The error on y is provided in the field dy, and a mask is also stored in the field mask (note that masked points are associated with the value 0). DataUnit objects also have a legendTag field that stores strings to be used for legend in plots. legendTag strings automatically concatenate as the object is being processed. Finally, DataUnit objects have a dedicated field that contains the processing object to be used on the data, so that data processing can be done by batch while insuring individual parametrisation.

- Bloc, Zone and Dispersion: each one of these classes refer to the corresponding dataset. They are derived from DataUnit and contain class-specific methods.

- ParamObj: this is the parent class for the handling of parameters from the raw data files. The format of the parameters changes with the version of the EVO console so that different sub-classes are required to handle them correctly (ParamV1 and ParamV2). This also allows importing other types of data files.



## Data controller

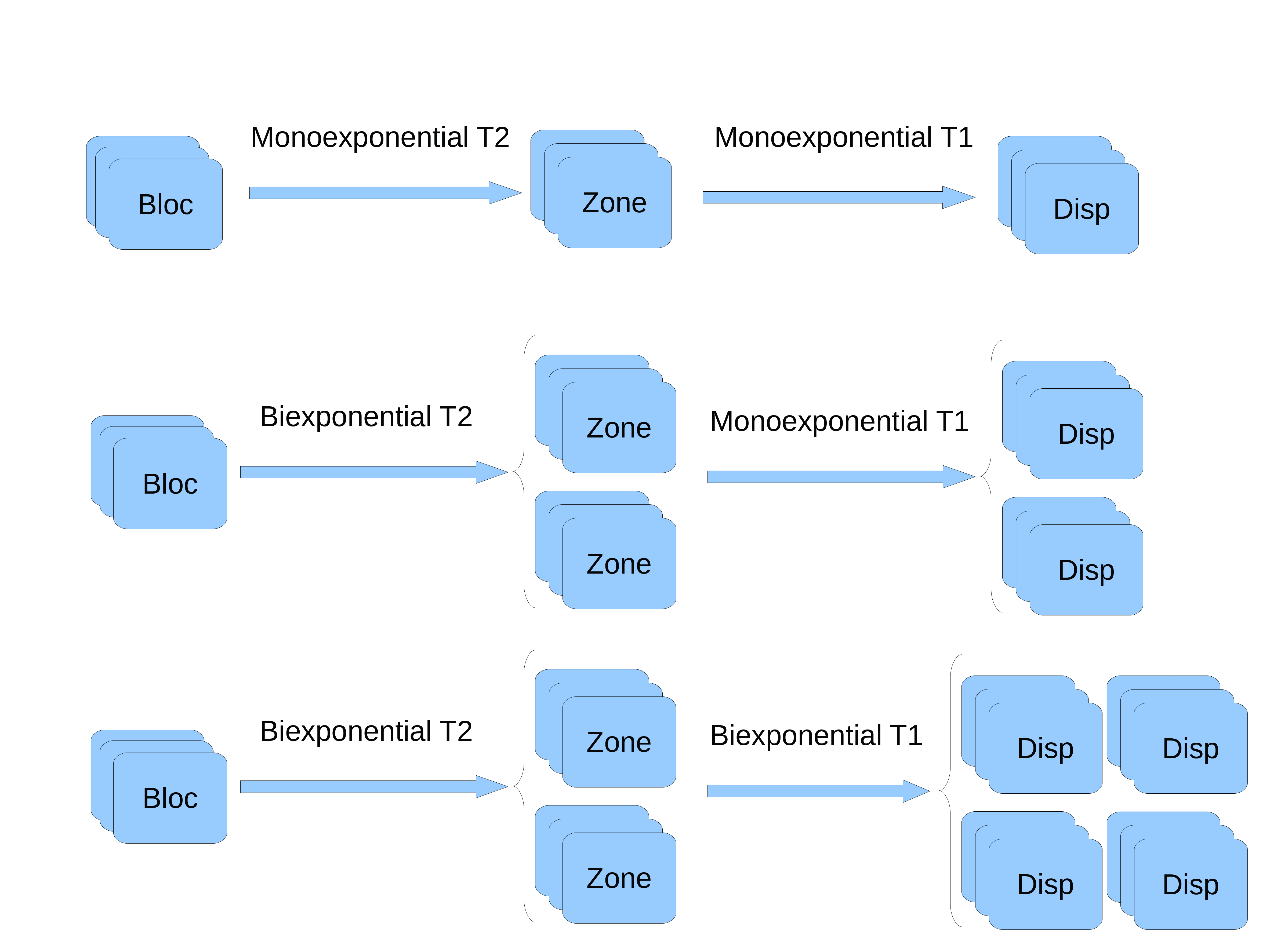
This component contains two types of classes: data import and data processing.

### Data import

This regroups the import function required to read the stelar data files (or other types of files). It also contains various tools needed to facilitate the import tasks.

### Data processing

This regroups all the functions needed to transform objects from one class to another as well as to perform curve fitting operations. It also contains the list of models and dispersion fitting algorithms. The processing of DataUnit object provides an array of DataUnit of the next level. For instance, once can process an array of Bloc objects with a Bloc2Zone algorithm to obtain a new array of Zone objects. The number of objects depends on the algorithm: if the model provided contains two outputs, then there will be twice as many Zone objects as there were Blocs in the input. In such a case it is important to provide a meaningful legend in the processing algorithm so that the Zone objects are labelled in a way that makes sense to the user. For instance, a bi-exponential T1 fit algorithm may have the field legendTag as {'Long T2 component','Short T2 component'} so that each Bloc object will provide a first Zone object with the legendTag field set to 'Long T2 component' and a second Zone object with 'Short T2 component'. The data stored in each Zone object can then be determined by the boundaries set to each parameter of the Biexponential model.

The processing algorithms are designed to use arrays of objects to facilitate data processing. Any further development must take care to remain compatible with this.

A similar approach is taken for multiple data processing in parallel. It is possible to process the same DataUnit with several algorithms, for instance if one wants to compare several processing pipelines. To do this, one can assign the list of processing objects (array of ProcessDataUnit) to the field processingMethod of the DataUnit object. Note that this should be done using the function assignProcessingFunction:

assignProcessingFunction(list of DataUnit objects, list of ProcessDataUnit objects to assign)

The processing is then performed using processData:

new DataUnit object = processData(list of DataUnit objects)

This will run all the algorithms assigned to all the DataUnit objects and will produce as many children objects.

Example:

% read the content of a sdf version 2 file:

[dataContent, parameter] = readsdfv2('C1-LAD.sdf');

% turn it into a Bloc object (this may be a list of blocs, depending on the content of the file, here we suppose that length(bloc) = 2):

bloc = makeComplexBloc(dataContent,parameter);

% create several data processing methods to generate the Zone objects:

b2z\_1 = ProcessAverageAbs;

b2z\_2 = ProcessPhasedMagnitude;

% assign the processing methods to the Bloc object:

assignProcessingFunction(bloc,[b2z\_1 b2z\_2]);

% Now generate the Zone objects (this generates 4 objects, 2 for each of the initial Bloc objects):

zone = processData(bloc);

% we can also use multiexponential algorithms:

z2d\_1 = Monoexp;

z2d\_2 = BiexponentialT1;

% Again, we just have to assign the algorithms and to launch the process

assignProcessingFunction(zone,[ z2d\_1 z2d\_2]);

disp = processData(zone);

% the result is a list of 12 Dispersion objects, as each Zone object provides one monoexponential and two biexponential dispersions.

Since this approach does not keep the relationship between the objects from the data structure, a parent and children approach is used and each DataUnit is automatically linked to the others during processing. The parent field contains the list of objects that have been used to derive the data unit (usually just one) and the children contains all the other DataUnits derived from the algorithms list in the processingMethod’ field. This poses a complication when erasing datasets, since items listed in these fields must also be removed when one DataUnit is deleted. This is solved by calling the function remove:

objList = remove(objList,indexes);

This command removes the objects indexed at the position provided in a list of DataUnit objects, and makes sure that all parents objects are updated to unlink to the deleted objects. If no indexes are provided, then all the objects in the list are unlinked.

Example:

% using the data list obtained previously, with 12 dispersion objects:

disp = remove(disp,[1 4]);

% now disp is an array with 12 objects, and zone(1) and zone(2) only have 2 children each.

Having so many objects also makes it difficult to get the legend right for each of them. To make this task easier, the function collectLegend generates a legend for a DataUnit object, using the default legend provided by the algorithms at each stage of the processing.

Example:

% ProcessAverageAbs gives the default legend ‘Average magnitude’, while ‘Monoexp’ gives ‘Monoexponential T1’. Therefore an object processed with both gives the two of them:

collectLegend(disp(5))

% gives ‘Monoexponential T1, Average magnitude’.

% Multi-component algorithms have several tags so that one can differentiate between them:

collectLegend(disp(1))

% returns ‘Long T1 component, Average magnitude’, while:

collectLegend(disp(2))

% returns Short T1 component, Average magnitude

Datasets of the same type can be merged into a new DataUnit by using the function merge:

mergedUnit = merge(dataList);

This simply creates a new DataUnit object of the same type as dataList and places the list of objects to be merged into the field subUnitList. When this field is populated, the fields x, y, dy and mask are re-directed seamlessly to point towards the list of objects. No additional coding is required, read and write operations automatically update the list of objects. This allows un-merging the dataset easily while keeping the modifications:

dataList = unMerge(mergedUnit);

## Fitting dispersion data

Once the dispersion data is obtained, the fitting procedures can start. There are three elements used to that purpose: the dispersion object, which contains the data, the fit engine, which contains the algorithm used to perform the fit, and the model, which contains the equation or procedure used to generate the model curve. One can build a complete model by associating several elementary models (14N quadrupolar peaks, Lorentzian profile, power law,…) into a single model that is the sum of all of them. This is done by listing these elementary models as an array of objects into the fit engine object, using the field subModel. This operation can be done automatically using the function addModel, which updates the final model at the same time (field model).

Example:

% Creating a new fit engine for the data we created:

LorPeak = DispersionLsqCurveFit;

% This object should contain a model that is the sum of a Lorentzian contribution and a series of quadrupolar peaks. This is done as follows:

LorPeak = addModel(LorPeak,[Lorentzian QPFriesBelorizkyNormalised]);

% Starting the fitting procedure is done the same way as before, by assigning the processing object and launching the process. Here we only assign the model to the monoexponential fits (supposing we did not remove any of the dispersion objects):

assignProcessingFunction(disp(1:3:end), LorPeak);

% assigning the model in this way also runs the function ‘evaluateStartPoint’ from the fit engine, which estimates the start point using the data from the dispersion object. This estimation function can either be written in the fit engine object, which can estimate a start point using an overall approach, or in the model objects, which let each component evaluate its own start point if a function ‘evaluateStartPoint’ is provided.

% another model may be added to the other dispersion data sets:

Powerlaw = DispersionLsqCurveFit;

Powerlaw = addModel(Powerlaw, PowerLawTwoSegment);

assignProcessingFunction(disp([2:3:end 3:3:end]), Powerlaw);

% Then the data can be processed:

exp = processData(disp);

% this creates a new object with the fit results, but it also updates the fit engine objects with the best values so that one can plot the fits.

When processed, the results of the fitting algorithm are stored in the fit engine object, at fitEngine.model.bestValue and fitEngine.model.errorBar. These values are also updated into each of the sub-model components individually.

Note that manually modifying the field bestValue from the model field also updates the sub-model list. This can be useful to visualise how the individual components behave when manually changing the model parameters.

When creating a model, one may want to visualise additional components that would be difficult to separate into independent models. This is possible to do by listing equations into the field visualisationFunction from the DispersionModel objects. Once the model is fitted and the best values are found, these functions can be estimated using the function estimateVisualisationFunction.

Example

% This function comes handy for power law dispersions. The PowerLawTwoSegment script has the following input:

model.visualisationFunction = {'dl\*f.^v1', ...

'(dl\*f\_trans1^(v1-v2))\*f^v2'};

% these correspond to the two individual components of the model. Once the model has been assigned to a dataset and fitted, the components can be evaluated by using:

y = evaluateVisualisationFunction(modelObj,x);

% this provides a matrix y with the size [length(x), number of equations in visualisationFunction]

# Data viewer

This component contains all the classes required to display the data and to interact with the GUI.

TO BE COMPLETED

# Description of the Matlab objects and function

## ParamObj

This class is designed to contain some parameters in a structure (field paramList) and to facilitate some file-dependent operations such as generating time axes.

## Data import function

[dataArray, parametersArray] = readsdfvX(filename)

At a low level, importing experimental data is done using the functions readsdfv1 or readsdfv2 depending on the version of the sdf file. These function need the file name of the data file to be openened (from fopen) and return two elements, an array of data and an array of parameter objects. These can then be used to generate an array of custom-made bloc objects.

Example:

[data,param]= readsdfv2('tube16.1\_All\_P.sdf');

'data' and 'param' are cell arrays with the content of the file selected.

## DataUnit

This is the class of object from which are derived all the objects that contain some data, i.e. Bloc, Zone and Disp. It contains the following fields:

x: array of doubles, stores the values used to generate the X-axis of the plot (time, Bevo,...)

y: array of doubles, stores the values used to generate the Y-axis of th plot ('R1','fid',...)

dy: array of doubles, stores the values used to generate the error bars on Y

xLabel: string of characters displayed along with the X axis in TeX format ('Time (s}, ...)

yLabel: string of characters displayed along with the Y axis in TeX format ('R\_1 (s^{-1}, ...)

mask: array of booleans to mask the fields above, 1 when the data is retained, 0 when removed

parameter: ParamObj object containing the parameters associated with the data

legendTag: contains a string of characters that is used to label the dataset in the legends.

One Bloc object contains all the blocs generated by one experiment, so that they should have the same pulse sequence. When creating a bloc, the data is organised by array of objects so that batch operations are facilitated by the use of cellfun and arrayfun (see Matlab documentation). Bloc objects can be created from the outputs of the readsdfvX functions quite easily because they accept cells as inputs, in which case they create arrays of objects using each element of the cell arrays provided.

The function makeComplexBloc facilitates the creation of Bloc lists from the outputs of the file readers.

Examples:

blocList = makeComplexBloc(data,param);

blocList is an array of Bloc objects with the same size as 'data' or 'param'.

## ProcessDataUnit

Bloc, Zone and Disp objets can be used to generate new data. Zone object are used to make Disp objects, and Blocs are used to make Zone (or Disp, too, is a proper class is defined). These processes can be modified to use different algorithms or parameters, so this process has been encapsulated into an object of type ProcessDataUnit.

ProcessDataUnit objects transform one type of object into another:

Bloc2Zone: takes an array of Blocs as an input and provides an array of Zones

Zone2Disp: same as above, but from Zone to Disp object

Bloc2Bloc: filter class that acts on a Bloc object

etc...

The fields for this objects are as follows:

functionName: string of chars, name of the model as appearing in the figure legend

labelY: string of char, labels to be used to generate the Y-axis labels

labelX: string of char, labels to be used to generate the X-axis labels

Other classes derived from this one have specific fields and methods. For instance, Bloc2Zone objects derive from ProcessDataUnit and have the additional method makeZone, which generate the Zone object from the Bloc object. Zone2Disp objects have the method makeDisp, which does a similar job.

For instance, ProcessAverageAbs is a class that generate Zone objects by using the average of the magnitude data from the Bloc objects,so it derives from Bloc2Zone.

Example:

b2z = ProcessAverageAbs;

zoneList = makeZone(b2z,blocList);