**Data Import**

Data import is the first step of any data processing pipeline. It is also the most complicated one since it depends on the initial data formatting of the considered raw file. FitLike is able to import two types of file: .sdf and .sef files. Both are Stelar©  data files corresponding to raw data (complex signal) and processed data respectively. All the functions presented here are stored in Data Controller/data import folder.

**.sdf files**

These files have two versions with two different data formatting. Briefly, both are organised with a header containing all the acquisition parameters followed by raw data. In version 1, couple of header/data are repeated for each zone acquired i.e. magnetization versus inversion time. In version 2, data are gathered by experiment thus dispersion curve are containing in one couple instead of N couples in version 1 (N is the number of zone, usually equal to the number of magnetic field applied). In any version, length of the header is variable depending on the sequence. Data are organised in column with variable number (time, real part, imag part…). The column’s names are indicated in version 2.

Because both versions are totally different two dedicated Matlab functions were written:

* [data, parameter] = readsdfv1(filename) for version 1
* [data, parameter] = readsdfv2(filename) for version 2

Also a ‘check version’ function is provided to check the version of the sdf file:

* ver = checkversion(filename)

In both functions, data are returned in a structure with three fields (time, real, imag). Each field is a 1xN cell array of double matrix where N corresponds to the number of experiment found in the file. In version 1 an experiment is defined by the succession (or not) of "ZONE" with identical data size (BS, NBLK), identical parameter structure (fieldnames) and identical sequence name (EXP field). See version 1 format for details. Parameter are 1XN ParamObj, see the container presentation for details.

*Note: For version 1 sdf file, the data columns are not specified. To overcome this issue, a custom format file is used that help the function to know the index of the time, real and imag column depending on the sequence and number of column. This format file or formatsettings.mat is created and updated by the getformatdlg() function and stored at the same hierarchical level in a ‘format’ folder.*

**.sef files**

These files correspond to Stelar processed files. It means that user has already applied data processing pipeline to the complex raw data signal using the Stelar software. Sef files do not contained any acquisition parameters and, to my knowledge, are organised as:

STELAR Export File

\_BRLX\_\_\_\_\_\_ \_T1\_\_\_\_\_\_\_\_ \_R1\_\_\_\_\_\_\_\_ \_\_\_\_%err\_\_\_ +-err(R1)\_\_ Zone File

3.0000e+01 1.8095e+00 5.5263e-01 4.4526e-01 2.4606e-03 7 file1.sef

2.2228e+01 1.6539e+00 6.0461e-01 7.2095e-01 4.3590e-03 8 file1.sef

….

I assumed here that sef files are always dispersion curve (R1 versus B0) but this assumption is probably wrong and dedicated importation function should be updated.

* data = readsef(filename)

Data are returned in a structure with three fields (x, y, dy) corresponding to the magnetic field, the relaxation rate and the relaxation error respectively.

**Import interface**

Import functions presented before are not adapted for fast importation task. Thus, a dedicated GUI function was developed to allow importation of sdf or sef files and sequence selections. This function should be updated if any other formats are considered.

After adding the FitLike path (run the ‘launcher’ for instance), open the interface with:

[filelist, sequence, data, parameter] = importData()

The following window will then appear:

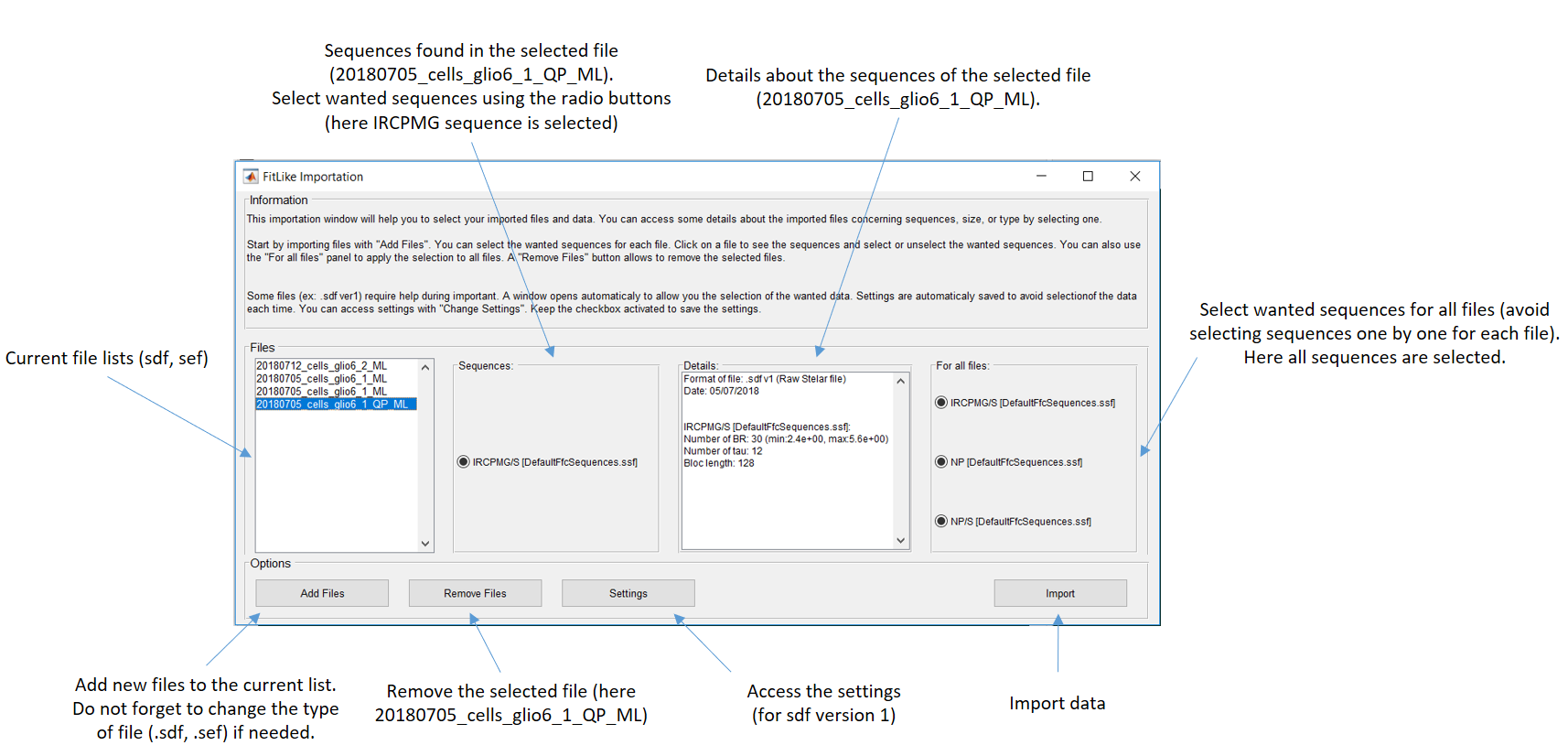


Figure 1: Importation window from the importData() function

If you click on “Settings” you can delete previous format settings to read sdf version 1 files. Modification is not possible since it could produce errors later. Thus if you set a wrong format setting for a specific sequence/number of column you need to delete it and reset it when format window appears (see below).

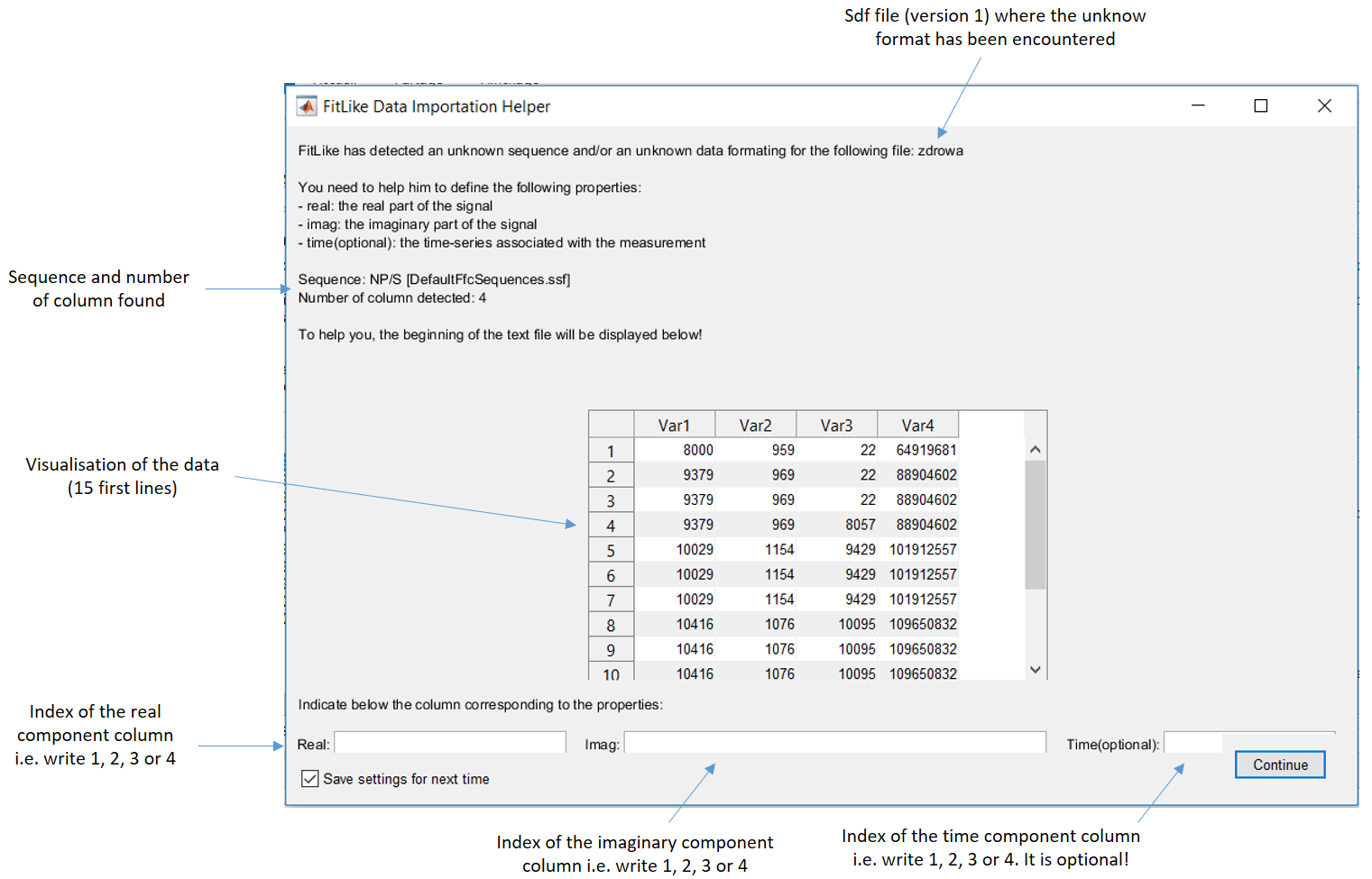


Figure 2: Format window that opens automaticaly if unknow format (i.e. sequence/number of column) is found in sdf file version 1 only.

Output of importData() function are not formatted to apply FitLike functions (processing, fitting,…). To obtain array of RelaxObj (file container) you need to use the following function just after importData():

relaxObj = format\_import\_data(filelist, sequence, data, parameter)

This function will return 1xN RelaxObj where N is the number of experiments found in the files. Again, if M files are imported, N will always be N >= M.

*Note: Like the importData() function, the format\_import\_data() function is based on the previous description of format (sdf, sef). If you want to import different type of format, you will need to update this function too.*

**Data Export**

Data export is defined here as the exportation of RelaxObj/Data Unit objects in a formatted .txt files. Two types of export are available:

* export\_data(relaxObj): allow to export dispersion data from array of RelaxObj objects
* export\_fit(dataObj): allow to export model data (coefficient, error) from array of DataUnit objects

Output (formatted .txt files) can be imported in Excel using **‘,’ delimiter only.** See the functions to have information about the format. Functions can be found in Data Controller/Data export folder.

**Data Viewer**

This part is dedicated to the presentation of the different visual components (i.e. figure), their functionality and architecture. FitLike was made to allow further development and the architecture chosen for each component try to reflect this purpose.

**File Manager**

FileManager is the class that creates and control the file visualisation window in FitLike. It is composed of three main parts:

* A file tree that allows the selection and the edition of files (you can edit dataset, sequence or filename)
* Three data tree (in a tab object) that allows the visualisation and selection of the data contain in the selected file
* A console that throw messages from FitLike software

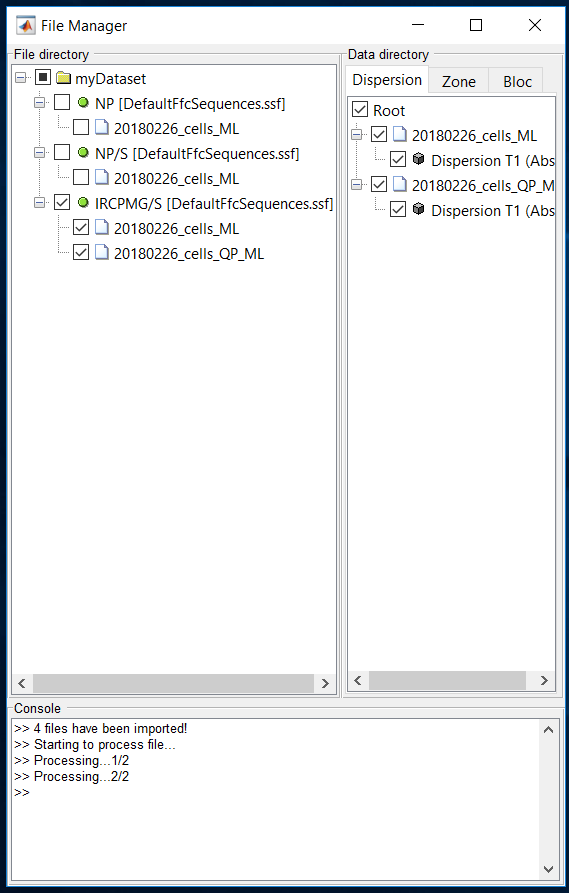


Figure 3: File Manager window. On the left, the file tree allows to select/edit files. On the right, tab/tree allows to visualise data contains in the selected files. At the bottom, a console throws messages.

**TreeManager object**

One of the main component of File Manager is the tree object. For the FitLike requirements I create a custom TreeManager class derived from the excellent submission of Robyn Jackey CheckBoxTree (FEX). This TreeManager object (checkbox tree) implements different methods that can be reuse for other applications:

* DragDrop(this,~,event): define conditions for drag and drop function based on the Value property and throw event after dropping if tree has changed. Briefly, TreeManager defines a valid\_target property where source’s node Value has specific target’s node Value. This property is used to define the possible drag and drop actions.
* onCheckboxClicked(tObj,~), onButtonUp(tObj,e) and flag = isMultipleCall(): redefine callbacks when user click on the nodes. If user clicks on specific checkbox, clicked checkbox node is returned. If it is the node itself (i.e. the text), user can edit the text (Value property) of the clicked node.
* hChildren = addNode(hParent, name, icon, type, varargin): add a new node to a parent node. If it exists a child node with the same Value property, this function returns this child else it creates a new node.
* hChildren = findobj(hParent, field, value): return all the nodes that have a field equals to value input (equivalent of findobj() in Matlab).
* hNodes = stackNodes(hNodes, new\_order, newNode) allows to reorder the nodes and/or insert a new node at a given position.

You can use TreeManager without any other class (except Robyn Jackey classes) by replacing FileManager input by anything else (0, ‘1’…). See the CheckboxTree properties for other input and Robyn Jackey examples for tree uses.

**FileManager object**

FileManager has specific behaviours that will be described below:

* Files (RelaxObj) are automatically sorted according to the property dataset, sequence and filename. The organisation is hierarchical: dataset property is the first node layer, sequence the second and filename (=file) the third.
* Clicking on a file node’s text (not the checkbox) allow to edit the selected node. If dataset or sequence are modified, all the files that shared this property value will be updated.
* Clicking on a file checkbox fires the data visualisation in the visible data tree. If nothing appears it means that the type of data (bloc, zone, dispersion) is missing in the selected files. If the selected nodes in the file tree is a sequence or a dataset, all the children nodes are selected.
* Clicking on a data checkbox fires an event ‘DataSelected’. This event is used by FitLike to know the selected data and display them.
* Changing the data tree visible (clicking on another tab) fires an event ‘TreeHasChanged’ and automatically update the data type displayed. For instance, if Zone tree is selected, zone data of the selected file are displayed.
* Drag and drop is implemented in file tree. However, a specific node need to be dropped at the same level or +1 level (for instance, files can be dropped in sequence or file, not in dataset). If this action changes the file property (if file is dropped in another sequence), an event is fired to update the object property.

To speed up data access, RelaxObj handles are stored in the UserData property of the file’s node (in file tree and data tree). Also, invisible nodes i.e. nodes in masked tabs are automatically deleted.

As it will be described below, FileManager is a stand-alone component that can work out of FitLike software. This particularity changes the behaviour of certain functions:

* Edit file(s) update automatically the RelaxObj handles without passing by FitLike
* Delete file(s) delete only the nodes and the RelaxObj handles, not the RelaxObj itself
* Console messages are displayed in the command window instead of in the console itself
* Drag and drop actions that change file(s) property (file dropped inside another dataset, sequence) update automatically the RelaxObj handles.

Please see the script example in Documentation/FitLike offline/FileManager\_offline.mat to have an overview of the different functions. Main idea is to allow fast visualisation and modification of files while using FitLike in command line.

**Display Manager**

DisplayManager is a class that creates and control data visualisation in FitLike. It is mainly composed of a tab group object that allows custom tab design. Zoom and pan from the built-in toolbar were kept for axis control. Like FileManager, DisplayManager is a stand-alone component and can be called out of FitLike. See the script example in Documentation/FitLike offline/DisplayManager\_offline.mat.

**Tab system**

DisplayManager handles a custom tab system. Tabs are created with the built-in function uitabgroup(). Because it is not possible to custom directly the built-in tab (uitab function) custom tabs are built from Container object. Then by assigning the uitab() as Parent of the Container object custom class components can be inserted in the tab system.

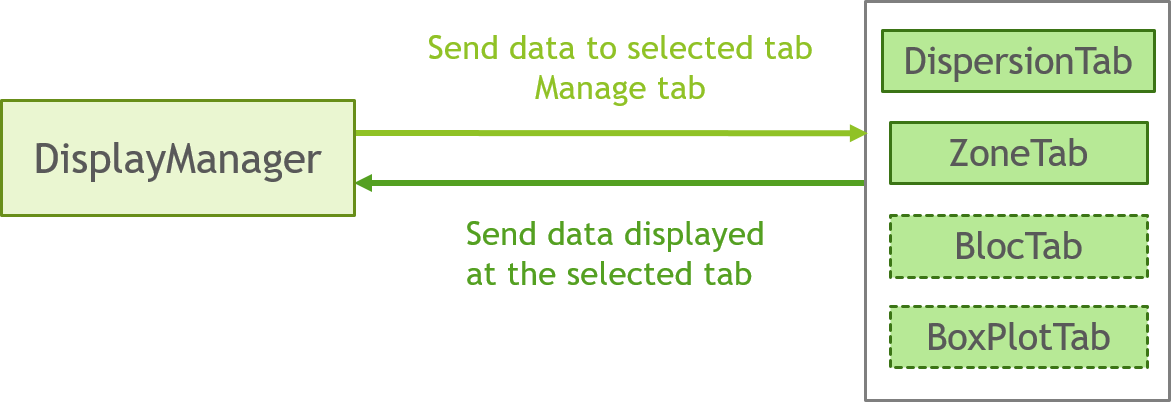
*Note: Inserting components in uitab() can be achieved with simpler method. However, to keep the custom object as a class (and thus using property/method) the object need to be created inside a Container.*

Figure 4: Custom tab system. DisplayManager contains different custom classes in its tab group object (DispersionTab, EmptyTab…). DisplayManager handles data access and the creation/deletion of custom tabs.

DisplayManager controls the tab system with two methods:

* addTab(this) adds a new empty tab (EmptyTab object) to the uitabgroup. It also adds a context menu to allow the deletion of the tab (‘Close tab’ menu object) through the setUImenu(this) method. The function is triggered by the selection of the specific tab ‘+’ (EmptyTabPlus object).
* removeTab(this) removes the selected tab. This function is triggered by the context menu created with the tab.

DisplayManager can send data to display (or remove) to the selected tab using:

* addPlot(this, hData, idxZone) adds data to display in the selected tab. Data are defines by hData (a DataUnit array) and idxZone (a vector). IdxZone indicates the zone index to show. If all the zones need to be display for a given hData(i) then idxZone(i) = NaN
* removePlot(this, hData, idxZone) removes data from the selected tab. Data to remove are defines by hData (a DataUnit array) and idxZone (a vector).

If the selected tab is an EmptyTab object, DisplayManager replaces this last by the appropriate one (DispersionTab for dispersion data, ZoneTab for zone data…).

DisplayManager can return some properties of the selected tab using:

* [hData, idxZone] = getData(this) return the data stored in the selected tab. hData is an array of DataUnit object while idxZone is a vector of same size.
* leg = getLegend(this) return the current legend of the selected tab. Output will be empty if no legend is displayed.

**EmptyTab**

The tab group object is used for the integration of complex object components that allow the visualisation of specific data (dispersion, zone, bloc, …) with specific tools (show error, show residual…). These complex object components are EmptyTab class or derived classes. EmptyTab derived from Container class object. EmptyTab creates a simple container with axis and defines the required properties for data display:

* hData : handle to the data array object (Dispersion, Zone…)
* idxZone: vector indicating the data to read (zone) in hData. If all zone need to be selected for a given hData(i) then idxZone(i) = NaN
* hGroup: structure storing the handle to the plot object (line, errorbar…)
* PlotSpec: structure storing the plot specification (marker, color…)
* inputType: char indicating the input data type accepted by the tab

Also EmptyTab defines important methods for derived classes:

* [hData, idxZone] = getData(this) return the handle of the data stored in the tab with their corresponding zone index.
* leg = getLegend(this) return the legend of the tab
* plotID = getPlotID(this, hData, idxZone) return plot ID of the data stored or the ID of a given data handle/zone index couple.
* idxZone = getIdxZone(hPlot) return the zone index of a given plot object. Zone index is recovered from the plot ID stored in the UserData property of the plot object.

**DispersionTab**

DispersionTab is a more complex class derived from EmptyTab that allows the visualisation of dispersion data. A lot of functionality are available with this component but only general principles will be described here. See the DispersionTab class and user documentation for details.

Dispersion data are imported through the addPlot() method that initialise different important features for data visualisation:

* Data handle and zone index are stored in hData and idxZone, duplicates are then avoided by comparing new input with previous data stored. Also storing data allows to recover plot objects after deletion.
* Plot specifications (color, marker) are set during importation and store in the PlotSpec structure allowing to recover same graphical aspect even when plot objects are deleted. Specific rules are applied:
  + Dispersion data from same files have the same color
  + Dispersion data from same files have different markers
* Different listeners are set to handle dynamic update of the dispersion data:
  + Data properties (x,y,dy,mask,processingMethod) are updated in updateData() method
  + Data deletion is fired by the DataDeletion event and achieved in deletePlot() method
  + Data meta-data (xLabel, yLabel) are updated in setLabel() method
* Also legend can be updated at this step if a new input dispersion data comes from the same file than a previous stored data. In this case, legend inputs of these data are updated to specify the displayName and the filename for both.

Most of the property related to the data (hData, idxZone, hGroup, PlotSpec) are ordered i.e. a given hData(i) has its plot specification in PlotSpec(i) and its graphical handles in hGroup(i). This help to find, modify or delete objects faster.

Also to avoid multiple methods, display methods (showData, showError, showFit…) are used as callback of pushbutton objects and as methods call with addPlot().

The way DispersionTab is built has the advantage to be general and thus can be used to derived ZoneTab object without adding a lot of different methods.

**Processing Manager**

ProcessingManager