ParSeq

Release 1.0.0

unknown

May 30, 2023

CONTENTS

1 Main features 3

2 Dependencies 5

3 Launch an example 7

4 Hosting and contact 9

5 User's Guide 11

5.1 Create analysis pipeline 11

5.1.1 Basic concepts and ideas 11

5.1.2 Prepare pipeline metadata and images 12

5.1.3 Make data nodes 12

5.1.4 Make data transformations 12

5.1.5 Make GUI widgets 12

5.1.6 Make fitting worker classes 12

5.1.7 Make data pipeline 13

5.1.8 Create test data tree 13

5.1.9 Create pipeline starter 13

5.1.10 Creating development versions of analysis application 13

5.2 Data nodes 13

5.3 Data model 14

5.4 Data transformations 16

5.5 User transformation widgets 17

5.6 Notes on usage of GUI 18

5.6.1 Load project 18

5.6.2 Docked node widgets 18

5.6.3 File tree and data formats, metadata 18

5.6.4 Data tree 19

5.6.5 Combine dialog 19

5.6.6 Plots 19

5.6.7 Fit widgets 19

5.6.8 Help panel 19

5.6.9 About dialog 20

5.6.10 Undo and redo lists 20

5.6.11 Save project, data and plot scripts 20

5.7 The MIT License 20

Python Module Index 21

Index 23

GitHub

PyPI

License

Author
Konstantin Klementiev (MAX IV Laboratory)

Package ParSeq is a python software library for **Par**allel execution of **Seq**uential data analysis. It implements a general analysis framework that consists of transformation nodes – intermediate stops along the data pipeline to visualize data, display status and provide user input – and transformations that connect the nodes. It provides an adjustable data model (supports grouping, renaming, moving and drag-and-drop), tunable data format definitions, plotters for 1D, 2D and 3D data, cross-data analysis routines and flexible widget work space suitable for single- and multi-screen computers. It also defines a structure to implement particular analysis pipelines as relatively lightweight Python packages.

ParSeq is intended for synchrotron based techniques, first of all spectroscopy.

A screenshot of a scanning XES analysis pipeline as an application example:

CHAPTER ONE

MAIN FEATURES

- ParSeq allows creating analysis pipelines as lightweight modules.
- Flexible use of screen area by detachable/dockable transformation nodes (parts of analysis pipeline).
- Two ways of acting from GUI onto multiple data: (a) simultaneous work with multiply selected data and (b) copying a specific parameter or a group of parameters from active data items to later selected data items.
- Undo and redo for most of treatment steps.
- Entering into the analysis pipeline at any node, not only at the head of the pipeline.
- Creation of cross-data combinations (e.g. averaging, RMS or PCA) and their propagation downstream the pipeline together with the parental data. The possibility of termination of the parental data at any selected downstream node.
- Parallel execution of data transformations with multiprocessing or multithreading (can be opted by the pipeline application).
- Optional curve fitting solvers, also executed in parallel for multiple data items.
- Informative error handling that provides alerts and stack traceback – the type and location of the occurred error.
- Export of the workflow into a project file. Export of data into various data formats with accompanied Python scripts that visualize the exported data for the user to tune their publication plots.
- ParSeq understands container files (presently only hdf5) and adds them to the system file tree as subfolders. The file tree, including hdf5 containers, is lazy loaded thus enabling big data collections.
- A web viewer widget near each analysis widget displays help pages generated from the analysis widget doc strings. The help pages are built by Sphinx at the startup time.
- The pipeline can be operated via scripts or GUI.
- Optional automatic loading of new data during a measurement time.

The mechanisms for creating nodes, transformations and curve fitting solvers, connecting them together and creating Qt widgets for the transformations and and curve fits are exemplified by separately installed analysis packages:

- ParSeq-XES-scan
- ParSeq-XES-dispersive
- ParSeq-XAS

CHAPTER TWO

DEPENDENCIES

- silx – for plotting and Qt imports
- sphinx – for building html documentation

CHAPTER

THREE

LAUNCH AN EXAMPLE

Either install ParSeq and a ParSeq pipeline application by their installers to the standard location or put them to any folder in their respective folders (`parseq` and e.g. `parseq_XES_scan`) and run the `*_start.py` module of the pipeline. You can try it with `--help` to explore the available options. An assumed usage pattern is to load a project `.pspj` file from GUI or from the starting command line.

CHAPTER

FOUR

HOSTING AND CONTACT

The ParSeq project is hosted on GitHub. Please use the project's Issues tab to get help or report an issue.

CHAPTER

FIVE

USER'S GUIDE

5.1 Create analysis pipeline

Consider *parseq_XES_scan* as an example for the development steps described below.

5.1.1 Basic concepts and ideas

An analysis pipeline consists of data nodes and transformations that connect the nodes. Data nodes define array names that will appear as attributes of data objects, e.g as `item.x`, with `x` being an attribute of a data object `item`. The array values will be read from files or calculated from other arrays. The pipeline can be used with or without GUI widgets. In the former case, the defined node arrays will appear in the node plotting: 1D, 2D or 3D (a stack of 2D plots).

Each transformation class defines a dictionary of transformation parameters and default values for them. It also defines a static method that calculates data arrays. The parameters will be attributed to each data object. The parameter values are supposed to be changed in user supplied GUI widgets. This change can be done simultaneously for one or several active data objects. Alternatively, any parameter can be copied to one or several later selected data.

Each transformation can optionally define the number of threads or processes that will start in parallel and run the transformation of several data items. The multiprocessing python API requires the main transformation method as a *static* or *class* method (not an instance method). Additionally, for the sake of data transfer in multiprocessing, all input and output arrays have to be added to `inArrays` and `outArrays` lists (attributes of the transformation class).

In the user-supplied GUI widgets, one for each data node, all interactive GUI elements should get registered using a few dedicated methods. The registration will enable automatic GUI update from the active data and will run transformations given the updated parameters, so no signal slots are typically required. The registration will also enable copying transformation parameters to other data by means of popup menus on each GUI element.

One or more curve fitting routines can optionally be defined per data node. Similarly to transformations, fitting solvers can run in parallel for several data items. Fitting parameters can be constrained or tied to other parameters.

The data model is a single object throughout the pipeline. Data can be rearranged by the user: ordered, renamed, grouped and removed. The data model tree is present in *each* data node, not as a single tree instance, with the idea also to serve as a plot legend that should always be close to the plot. User selection in the data model is common for all transformation nodes. For 1D data, the line color is the same in all data nodes. 1D data plotting can optionally be done for dynamically (via mouse selection) or statically (via check boxes) selected data. 2D and 3D data plotting is always done for one selected data object.

The transformation class docstrings will be built by ParSeq using Sphinx into an html file and will be displayed in a help panel close to the transformation widget.

5.1.2 Prepare pipeline metadata and images

Create a project directory for the pipeline. Create *__init__.py* file that defines metadata about the project. Note that user pipeline applications and ParSeq itself use the module *parseq.core.singletons* as a means to store global variables; the pipeline's *__init__.py* module defines a few of them. Together with the docstrings of the module, these metadata will appear in the About dialog.

Create *doc/_images* directory and put an application icon there. The pipeline transformations will have class docstrings that may also include images; those images should be located here, in *doc/_images*.

5.1.3 Make data nodes

To define a node class means to name all necessary arrays, define their roles, labels and units.

5.1.4 Make data transformations

Start making a transformation class with defining a dictionary *defaultParams* of default parameter values. Decide on using multiprocessing/multithreading by specifying *nThreads* or *nProcesses*. If any of these numbers is > 1 (the default values are both 1), specify two lists of array names: *inArrays* and *outArrays*. Define a static or a class method `Transform.run_main()`. Note, it can have a few signatures. Within the method, get the actual transformation parameters from the dictionary *data.transformParams* and the defined data arrays as attributes of *data*, e.g. `data.x`.

For expensive transformations, you should update the *progress* status.

For accessing arrays of other data objects, use a different signature of `Transform.run_main()` that contains the *allData* argument. Note that in this case multiprocessing is not possible.

5.1.5 Make GUI widgets

The widgets that control transformation parameters are descendants of `PropWidget`. The main methods of that class are `PropWidget.registerPropWidget()` and `PropWidget.registerPropGroup()`. They use the Qt signal/slot mechanism to update the corresponding transformation parameters; the user does not have to explicitly implement the reaction slots. Additionally, these methods enable copying transformation parameters to other data by means of popup menus, update the GUI upon selecting data objects in the data tree, start the corresponding transformation and operate undo and redo lists.

Because each transformation already has a set of default parameter values, these GUI widgets can gradually grow during the development time, without compromising the data pipeline functionality.

Provide docstrings in reStructuredText markup, they will be built by Sphinx and displayed near the corresponding widgets.

5.1.6 Make fitting worker classes

Similarly to a transformation class, a fitting class defines a dictionary *defaultParams*, defines multiprocessing/multithreading and a static or a class method `Fit.run_main()`.

5.1.7 Make data pipeline

This is a small module that instantiates the above nodes, transformations, fits and widgets and connects them together.

5.1.8 Create test data tree

Put a few data files in a local folder (i.e. *data*) and create a module that defines a function that loads the data into a *data tree*, defines suitable transformation parameters and launches the first transformation (the next ones will start automatically).

5.1.9 Create pipeline starter

The starter should understand command line arguments and prepare options for loading the test data and to run the pipeline with and without GUI.

5.1.10 Creating development versions of analysis application

Copy the whole folder of the application to the same level but with a different name, e.g. append a version suffix. In the import section in the start script change the import name to the above created folder name. Done.

5.2 Data nodes

Data nodes are intermediate stops along the data pipeline. Their main purposes are to visualize data, display transformation status, select parameters of data transformations, possibly stop or split data propagation or combine with other data. Each data node defines and operates arrays that get their values in the upstream part of the pipeline.

`class parseq.core.nodes.Node(`*widgetClass=None*`)`

Parental Node class. Must be subclassed to define the following class variables:

***name*: str**

The name of the node, also shown as a GUI tab and also is a section name in ini file. Must be unique in the pipeline.

***arrays*: OrderedDict of dicts**

Describes the arrays operated in this node. Note, this object only contains data description; the actual data arrays will be attributed to data objects (items). The keys of *arrays* are names of these arrays, the values are dictionaries that optionally contain the following kwargs:

***role*: str, default = '1D' (not plotted)**

The array's role. Can be 'x', 'y', 'yleft', 'yright', 'z' or '1D' for 1D arrays (one and only one x-axis array is required), '2D' for 2D plots and '3D' for stacked 2D images. '0D' values are listed in the data tree. 'optional' array is not required in data files and can be used as an auxiliary. Unless with a '0D' role, each array will appear in *data location* dialog to define its location.

***raw*: str, default = array name**

Can define an intermediate array at the pipeline head when the main array (the key in *arrays*) is supposed to be obtained by an after load transformation. The plotted array is still the main one. The idea of having a *raw* version of an array is in the possibility of creating a transformation that not only begins at the first node but can also *end* at the first node.

***qLabel*: str, default = array name**

Used in the GUI labels.

***plotLabel*: str, or list of str, default = qLabel**
Axis label for the GUI plot. For 2D or 3D plots the 2- or 3-list that corresponds to the plot axes. The list may contain keys from *arrays* and then the label and unit are taken from that dictionary (the entry of *arrays*) or, alternatively, the list elements themselves are axis labels. For 0D values, this parameter may hold a format string to be used with the format() method.

***qUnit*: str, default None**
Optional data unit to be displayed in the GUI.

plotUnit*: str, default = *qUnit
Attached to the plot label in parentheses. For example, for Å^-1: *qUnit* = u'Å^1' and *plotUnit* = r'Å$^{-1}$'.

***plotParams*: dict, default is *{}* that assumes thin solid lines**
Default parameters for plotting. Can have the following keys: *linewidth* (or *lw*), *style*, *symbol* and *symbolsize*. Note that color is set for a data item and is equal across the nodes, so it is set not here.

***checkShapes*: list of str**
Can be useful at data file reading. If given, the list contains keys of *arrays*. The corresponding arrays will be checked for equal shape. The names of multidimensional arrays can be ended by a slice. Example: *checkShapes = ['theta', 'i0', 'xes3D[0]'].*

***auxArrays*: list of lists**
Can be useful only for data export. Array names are grouped together so that the 1st element in a group is an x array and the others are y arrays. This grouping is respected only for the export of 1D data.

`__init__`(*widgetClass=None*)

Instantiates the node and optionally passes a Qt widget class of a user dialog that defines transformation parameters.

`get_prop`(*arrayName*, *prop*)

Returns the property *prop* for a given array name defined in this node. This method can be useful in creating the GUI part of a transformation node.

`is_between_nodes`(*nodeName1*, *nodeName2*, *node1in=True*, *node2in=True*)

Returns True if this transformation node is between the given two nodes in the sense of data propagation in the pipeline. This method can be useful in creating the GUI part of a transformation node.

***nodeName1* and *nodeName2*: Node**
nodeName2 can be None, the right end is infinite then.

***node1in* and *node2in*: bool**
define whether the interval is closed (when True) or open.

5.3 Data model

The ParSeq data model is a hierarchical tree model, where each element has zero or more children. If an element has zero children, it is called item and is a data container. If an element has at least one child, it is called group. All items and groups are instances of `Spectrum`.

The tree model can be manipulated in a script, and the following reference documentation explains how. Otherwise, most typically it is used within the ParSeq GUI, where the tree model feeds the model-view-controller software architecture of Qt, where the user does not have to know about the underlying objects and methods. See *Notes on usage of GUI*.

`class parseq.core.spectra.Spectrum`(*madeOf*, *parentItem=None*, *insertAt=None*, ***kwargs*)

This class is the main building block of the ParSeq data model and is either a group that contains other instances of `Spectrum` or an item (data container). All elements, except the root, have a parent referred to by *parentItem* field, and parents have their children in a list *childItems*. Only the root item is explicitly created by the constructor of `Spectrum`, and this is done in the module that defines the pipeline. All other tree elements are typically created by the parent's `insert_data()` or `insert_item()` methods.

`__init__`(*madeOf*, *parentItem=None*, *insertAt=None*, ***kwargs*)

madeOf
is either a file name, a callable, a list of other `Spectrum` instances (for making a combination) or a dictionary (for creating branches).

parentItem
is another `Spectrum` instance or None (for the tree root).

***insertAt*: int**
the position in *parentItem.childItems* list. If None, the spectrum is appended.

***kwargs*: dict**
defaults to the dictionary: dict(alias='auto', dataFormat={}, originNodeName=None, terminalNodeName=None, transformNames='each', copyTransformParams=True). The default *kwargs* can be changed in *parseq.core.singletons.dataRootItem.kwargs*, where *dataRootItem* is the root item of the data model that gets instantiated in the module that defines the pipeline.

***dataFormat*: dict**
is assumed to be an empty dict for a data group and must be non-empty for a data item. As a minimum, it defines the key *dataSource* and sets it to a list of hdf5 names (when for hdf5 data), column numbers or expressions of 'Col1', 'Col2' etc variables (when for column data). It may define 'conversionFactors' as a list of either floats or strings; a float is a multiplicative factor that converts to the node's array unit and a string is another unit that cannot be converted to the node's array unit, e.g. the node defines an array with a 'mA' unit while the data was measured with a 'count' unit. It may define 'metadata': a comma separated str of hdf5 attribute names that define metadata.

***originNodeName*, *terminalNodeName*: str**
The data propagation is between origin node and terminal node, both ends are included. If undefined, they default to the 0th node (the head of the pipeline) and the open end(s). If a node is between the origin node and the terminal node (in the data propagation sense) then the data is present in the node's data tree view as *alias* and is displayed in the plot.

transformNames
A list of transform names (or a single str 'each'). It defines whether a particular transform should be run for this data.

***copyTransformParams*: bool, default True.**
Controls the way the *transformParams* of the Spectrum are initialized: If False, they are copied from *defaultParams* of all transforms. If True, they are copied from the first selected spectrum when at least one is selected or otherwise from the ini file.

`find_data_item`(*alias=None*)

Finds the first data item with a given alias. Returns None if fails.

`get_items`(*alsoGroupHeads=False*)

Returns a list of all items in a given group, also included in all subgroups.

`insert_data`(*data*, *insertAt=None*, ***kwargs*)

This method inserts a tree-like structure *data* into the list of children. An example of *data*:

`data=["groupName", ["fName1.dat", "fName2.dat"]]` for a group with two items in it. All other key word parameters lumped into *kwargs* are the same as of `__init__()`.

`insert_item`(*name*, *insertAt=None*, ***kwargs*)

This method inserts a data item *name* into the list of children. All other key word parameters lumped into *kwargs* are the same as of `__init__()` and additionally *configData* that can pass an instance *config.ConfigParser()* that contains a saved project.

5.4 Data transformations

Data transformations provide values for all the arrays defined in one transformation node (*fromNode*) given arrays defined in another transformation node (*toNode*). Each transformation defines a dictionary of transformation parameters; the values of these parameters are individual per data item. Each transformation in a pipeline requires subclassing from `Transform`.

`class parseq.core.transforms.`**`Transform`**(*fromNode*, *toNode*)

Parental Transform class. Must be subclassed to define the following class variables:

name: str name that must be unique within the pipeline.

defaultParams: dict of default transformation parameters for new data.

Transforms, if several are present, must be instantiated in the order of data flow.

The method `run_main()` must be declared either with @staticmethod or @classmethod decorator. A returned not None value indicates success.

nThreads or *nProcesses* can be > 1 to use threading or multiprocessing. If both are > 1, threading is used. If *nThreads* or *nProcesses* > 1, the lists *inArrays* and *outArrays* must be defined to send the operational arrays (those used in `run_main()`) over process-shared queues. The value can be an integer, 'all' or 'half' which refer to the hardware limit *multiprocessing.cpu_count()*.

progressTimeDelta, float, default 1.0 sec, a timeout delta to report on transformation progress. Only needed if `run_main()` is defined with a parameter *progress*.

`__init__`(*fromNode*, *toNode*)

fromNode and *toNode* are instances of `Node`. They may be the same object.

`classmethod` **`run_main`**(*data*)

Provides the actual functionality of the class. Other possible signatures:

run_main(cls, data, allData, progress)
run_main(cls, data, allData)
run_main(cls, data, progress)

data is a data item, instance of `Spectrum`.

allData and *progress* are both optional in the method's signature. The keyword names must be kept as given above if they are used and must be in this given order if both are present.

allData is a list of all data items living in the data model. If *allData* is needed, both *nThreads* or *nProcesses* must be set to 1.

progress is an object having a field *value*. A heavy transformation should periodically update this field, like this: `progress.value = 0.5` (means 50% completion). If used with GUI, progress will be visualized as an expanding colored background rectangle in the data tree. Quick transformations do not need progress reporting.

Should an error happen during the transformation, the error state will be notified in the ParSeq status bar and the traceback will be shown in the data item's tooltip in the data tree view.

Returns True when successful.

5.5 User transformation widgets

If the data pipeline is supposed to take user actions from GUI, each transformation node should have a dedicated Qt widget that sets relevant transformation parameters. ParSeq offers the base class `PropWidget` that reduces the task of creating a widget down to instantiating Qt control elements, putting them in a Qt layout and registering them. The docstrings of the user widget class will be built by ParSeq using Sphinx documentation system into an html file that will be displayed under the corresponding widget window or in a web browser.

User transformation widgets can profit from using silx library, as ParSeq already uses it heavily. It has many widgets that are internally integrated to plotting e.g. ROIs. A good first point of interaction with silx is its collection of examples.

`class parseq.gui.propWidget.`**`PropWidget`**`(`*`*args: Any`*`, `*`**kwargs: Any`*`)`

The base class for user transformation widgets and a few internal ParSeq widgets. The main idea of this class is to automatize a number of tasks: setting GUI from data, changing transformation parameters of data from GUI, copying parameters to other data, starting transformation and inserting user changes into undo and redo lists.

`__init__``(`*`parent=None`*`, `*`node=None`*`)`

node is the corresponding transformation node, instance of `Node`. This parental __init__() must be invoked in the derived classes at the top of their __init__() constructors. In the constructor of the derived class, the user should create control elements and register them by using the methods listed below.

`registerPropGroup``(`*`groupWidget`*`, `*`widgets`*`, `*`caption`*`)`

Registers a group widget (QGroupBox) that contains individual *widgets* , each with its own data properties. This group will appear in the copy popup menu.

`registerPropWidget``(`*`widgets`*`, `*`caption`*`, `*`prop`*`, `*`**kw`*`)`

Registers one or more widgets and connects them to one or more transformation parameters.

widget: a sequence of widgets or a single widget.

caption: str, will appear in the popup menu in its "apply to" part.

prop: str or a sequence of str, transformation parameter name(s).

Optional key words (in *kw*):

convertType: a Python type or a list of types, same length as *prop*, that is applied to the widget value.

hideEmpty: bool, applicable to edit GUI elements. If True and the *prop* is None or an empty str, the edit element is not visible.

emptyMeans: a value that is assigned to *prop* when the edit element is empty.

copyValue: a single value or a list of length of *prop*. When copy *prop* to other data items, this specific value can be copied. If *copyValue* is a list of length of *prop*, it can mix specific values and a str 'from data' that signals that the corresponding prop is taken from the actual data transformation parameter.

transformNames list of str The transforms to run after the given widgets have changed. Defaults to the names in *self.node.transformsIn*.

dataItems a list of data items None (the transformation parameter will be applied to selected items) or 'all' (applied to all items).

`registerStatusLabel`(*widget*, *prop*, ***kw*)

Registers a status widget (typically QLabel) that gets updated when the transformation has been completed. The widget must have *setData()* or *setText()* or *setValue()* method.

Optional key words:

hideEmpty: same as in `registerPropWidget()`.

textFormat: format specification, as in Python's format() function e.g. '.4f' .

5.6 Notes on usage of GUI

5.6.1 Load project

To start testing a GUI, load a test project, typically located in *saved* directory. The "Load project" dialog has a preview panel that displays all node plots in the project, just browse over them. The initial visible plot displays the transformation node that was active when the project was saved.

5.6.2 Docked node widgets

With the idea of flexible usage of screen area, the node widgets were made detachable and dockable into the main ParSeq window. To do this, drag a node widget by its caption bar. To dock it back, hover it over the main window or use the dock button at the right end of the caption bar.

The state of each node widget (docked or floating) and its floating geometry is saved in ini file and project files.

5.6.3 File tree and data formats, metadata

The file tree is by default visible only in the pipeline head node(s). If needed, make it visible/hidden by the vertical button of the leftmost splitter widget "files & containers".

When you click on an entry in the data tree, the corresponding file or hdf5 entry will get highlighted in the starting transformation node widget. When you browse the file tree, the highlight color is green if this entry can be loaded, i.e. the data format fields in the data format widget are defined. To define the fields (array names), one can highlight one or several hdf5 datasets and use popup menu commands. Note that you can use hdf5 data sets from various hdf5 data groups or even hdf5 data files, not necessarily from one data group when you load one data item.

For column files, one should define the file header and expressions of variables *Col0*, *Col1* etc. for data fields (arrays). If an array definition is just a column, one can reduce it to the ordinal number of that column, so type *0* instead of *Col0*.

Metadata can be composed of string hdf5 fields or for column files they are copied from the header. Metadata are displayed in a panel below the plot in each node.

The format fields in the "data format" dialog can be saved into an ini file (.parseq/formats.ini) and later restored from it using the popup menu when right-clicked on a data file.

The tabs of "data format" dialog have some help text in their tooltips.

5.6.4 Data tree

The data tree can be populated from the file tree by using the popup menu or by a drag-and-drop action.

The newly loaded data get their set of transformation parameters from the first previously selected data item. If no items have been previously loaded, the parameters are read from the ini file. If the ini file does not exist yet, the parameters get their values from *defaultParams* defined in each transformation class.

Use the popup menu or corresponding keyboard shortcuts to rearrange the data tree.

The Qt tooltip on each data entry provides data path, shape and size. If an error occurs during a transformation, the tooltip also contains the last exception traceback.

In 1D transformation nodes, one can change the data visibility mode by clicking on the "eye" header section. Try these modes while selecting different data entries or groups.

Line properties of the selected data items can be set from the popup menu or by clicking on the data column header. New data get their line properties from the previously active data. The first data get their plot settings from the optional *plotParams* of node's arrays.

5.6.5 Combine dialog

A limited number of combination functions acting on several selected data items can be performed via the "combine" dialog that can be found under the data tree widget.

5.6.6 Plots

All types of plots implemented in ParSeq are taken from silx library. Find more about their functionality here.

Bear in mind that if several items have the same alias, silx displays only one of them, so make sure aliases are unique. Parseq will try to append a numbered suffix to the alias if the added data have the same file name. Aliases can always be changed by the user.

In 1D plotting window, clicking on a curve will select the corresponding data item in the data tree widget. Auxiliary curves can be added in user-defined transformation widgets by specifying a method *extraPlot()*. The curves should have their *legend* property defined in the following format: the data item alias followed by a dot followed by a sub-name. If this convention is followed, the curves become clickable, which will select the corresponding data item in the data tree. Selected data items are plotted on top of unselected items.

5.6.7 Fit widgets

If one or more fit widgets were specified for a given node, they appear in separate QSplitter under the node's plot. In the initial view, the splitters are collapsed.

5.6.8 Help panel

The help panel under transformation widgets is hidden by default and can be made visible by clicking on the small button "help" at the very bottom of the main window. Alternatively, it can be opened in the system browser.

5.6.9 About dialog

The about dialog displays the connectivity between the pipeline nodes in a dynamically created svg graph. If a fit is defined in a node, it is also displayed here.

5.6.10 Undo and redo lists

When a transformation parameter has been changed or a data item has been deleted, this action is inserted into the undo list. Clicking on the big undo button will revert the last action and put it into the redo list (not for the undelete operation). Any undo action can also be executed separately, not necessarily in the reverse order, by using the drop down-menu. Note, the undo entry for a delete operation will keep the reference to the deleted item, so to clean up the memory, this entry should be individually removed from the undo list.

5.6.11 Save project, data and plot scripts

The present data tree and all transformation parameters of all data items can be saved into a project file that has an ini text file structure. Simultaneously, data arrays defined in each node can optionally be exported as a few chosen data types. Note that data arrays will be exported only for the currently selected data items, not for all data. Two types of data plotting scripts can also be saved. These scripts will plot the exported data and are provided with the idea to help the user adjust their publication quality graphs.

In the saved project, file path to each data item is saved in two versions: as an absolute path and as a relative path in respect to the project location. When the project is copied together with the files to a new location, the project should be directly loadable. When copied from a GPFS location at a beamline, this may not work, and the relative paths have to be manually edited by a Search/Replace operation in a text file editor.

5.7 The MIT License

PYTHON MODULE INDEX

p

parseq.core, 11
parseq.core.nodes, 13
parseq.core.spectra, 14
parseq.core.transforms, 16
parseq.gui, 18
parseq.gui.propWidget, 17

INDEX

Symbols

`__init__()` (*parseq.core.nodes.Node method*), 14
`__init__()` (*parseq.core.spectra.Spectrum method*), 15
`__init__()` (*parseq.core.transforms.Transform method*), 16
`__init__()` (*parseq.gui.propWidget.PropWidget method*), 17

F

`find_data_item()` (*parseq.core.spectra.Spectrum method*), 15

G

`get_items()` (*parseq.core.spectra.Spectrum method*), 15
`get_prop()` (*parseq.core.nodes.Node method*), 14

I

`insert_data()` (*parseq.core.spectra.Spectrum method*), 15
`insert_item()` (*parseq.core.spectra.Spectrum method*), 16
`is_between_nodes()` (*parseq.core.nodes.Node method*), 14

M

module
    parseq.core, 11
    parseq.core.nodes, 13
    parseq.core.spectra, 14
    parseq.core.transforms, 16
    parseq.gui, 18
    parseq.gui.propWidget, 17

N

`Node` (*class in parseq.core.nodes*), 13

P

parseq.core
    module, 11
parseq.core.nodes
    module, 13
parseq.core.spectra
    module, 14
parseq.core.transforms
    module, 16
parseq.gui
    module, 18
parseq.gui.propWidget
    module, 17
`PropWidget` (*class in parseq.gui.propWidget*), 17

R

`registerPropGroup()` (*parseq.gui.propWidget.PropWidget method*), 17
`registerPropWidget()` (*parseq.gui.propWidget.PropWidget method*), 17
`registerStatusLabel()` (*parseq.gui.propWidget.PropWidget method*), 18
`run_main()` (*parseq.core.transforms.Transform class method*), 16

S

`Spectrum` (*class in parseq.core.spectra*), 14

T

`Transform` (*class in parseq.core.transforms*), 16