pyEvalData Documentation

Release 1.6.0

Daniel Schick, et.al.

Dec 27, 2023

CONTENTS

This is a Python module to read and evaluate experimental data. It can handle raw data from different sources such as spec, hdf5, NeXus files which are common data formats at synchrotrons, FELs, as well as in a growing number of laboratories. The evaluation provides common functionalities such as binning, error calculation, and advanced data manipulation via algebraic expressions as well as pre- and post-data-filters. Furthermore, advanced wrapper functions allow for plotting and fitting sequences of one or multiple scans in dependence of an external parameter.

A minimal code example would look like this:

```
import pyEvalData as ped
# define your data source
spec = ped.io.Spec(file_name='data.spec')
# initialize the evaluation
ev = ped.Evaluation(spec)
# define the x- and y-data
ev.xcol = 'motor1'
ev.clist = ['ct1', 'ct2', 'ct1/ct2']
# create a plot for scans 1-3
ev.plot_scans([1, 2, 3])
```

Please follow the user guide and examples for your first steps with `pyEvalData`.

CHAPTER

ONE

FEATURES

- reading of several pre-defined raw data formats
  - spec
  - hdf5
  - NeXus
  - user-defined text files
  - camera images (Dectris Pilatus, Princeton MTE, Greateyes, . . .)
  - composite sources
- easy implementation of new raw-data formats using an `interface class`
- common methods for plotting and fitting of experimental data, including:
  - data binning
  - error calculation
  - data manipulation via algebraic expressions
  - common data pre- and post-filters

CHAPTER

TWO

INSTALLATION

You can either install directly from pypi.org using the command

```
$ pip install pyEvalData
```

or if you want to work on the latest develop release you can clone `pyEvalData` from the main git repository:

```
$ git clone https://github.com/dschick/pyEvalData.git pyEvalData
```

To work in editable mode (source is only linked but not copied to the python site-packages), just do:

```
$ pip install -e ./pyEvalData
```

Or to do a normal install with

```
$ pip install ./pyEvalData
```

Optionally, you can also let pip install directly from the repository:

```
$ pip install git+https://github.com/dschick/pyEvalData.git
```

You can have the following optional installations to enable unit tests, as well as building the documentation:

```
$ pip install pyEvalData[testing]
$ pip install pyEvalData[documentation]
```

CHAPTER

THREE

CONTRIBUTE & SUPPORT

If you are having issues please let us know via the issue tracker.

You can contribute to the project via pull-requests following the GitHub flow concept.

CHAPTER

FOUR

LICENSE

The project is licensed under the MIT license.

4.1 User Guide

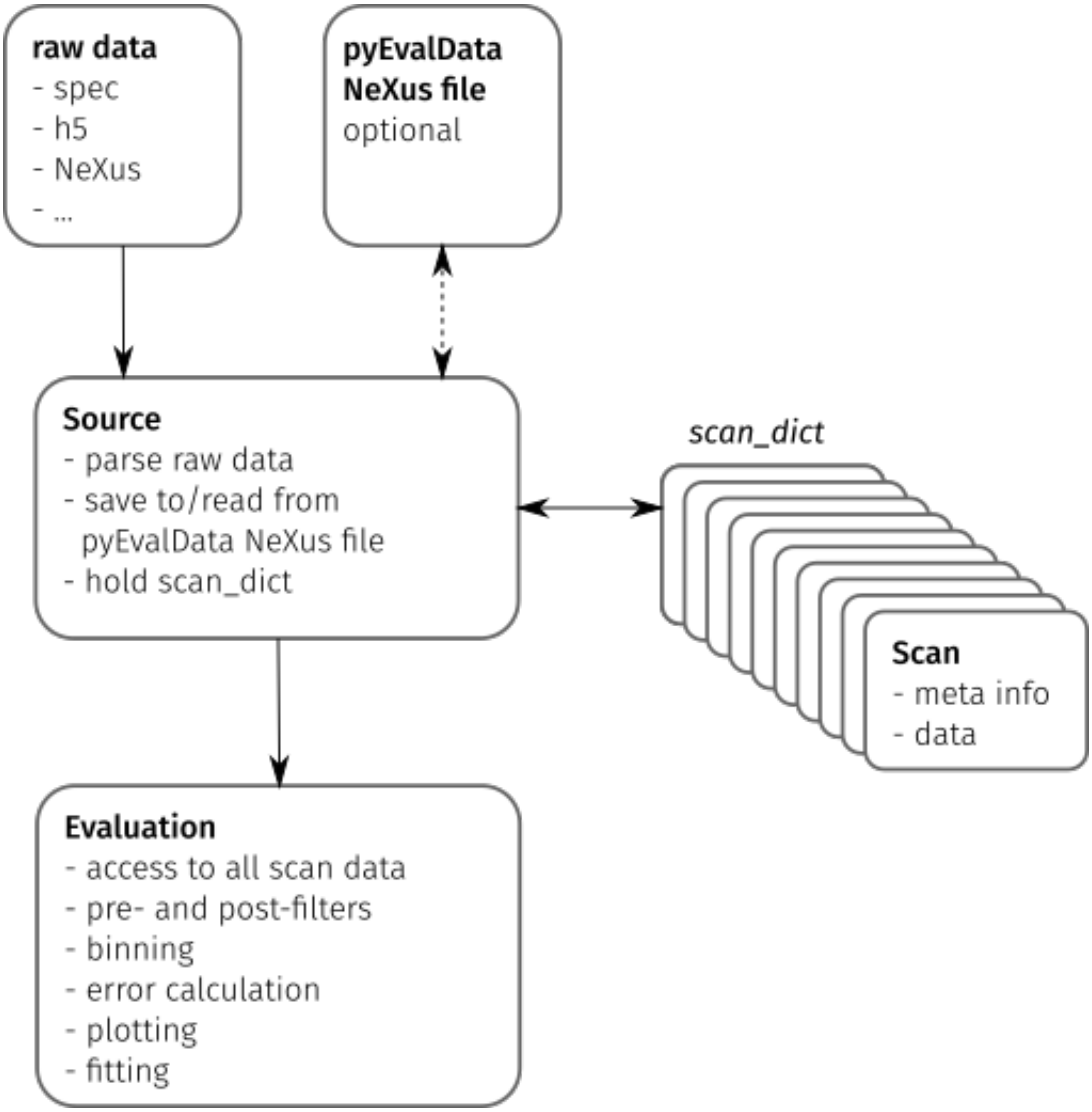
The idea of the `pyEvalData` module is to provide a simple but yet flexible way for reading and evaluating scientific data acquired at synchrotrons, FELs, or in the lab. It is written with the intend to reuse available code as much as possible and to simplify the access to complex data formats and very general evaluation and analysis procedures. Generally, students and scientists should focus on data interpretation rather than on scripting the same routines again and again.

Please read the upcoming section to understand the concept of reading and evaluating data with `pyEvalData` and how to extend/configure the module to your needs.

It is also strongly recommended to follow the *examples* which can also be run locally as `jupyter` notebooks. More details can be found in the *API documentation* of the individual sub-modules and classes.

4.1.1 General Concepts

The following figure illustrates the main components of the `pyEvalData` module and their interactions.



Raw Data

The starting point of most evaluations is any set of raw data files as generated by an experimental setup consisting of actuators such as motors and counters such as detectors, cameras, or sensors.

Typical data formats are human readable text files or compressed hdf5 or NeXus files. Cameras often use `tiff` or proprietary file formats.

Source

The `Source` class provides a common set of methods and attributes to read and store raw data. It further acts as an `interface` to implement the source-specific classes.

A `Source` class should be able to parse the raw data to detect all available scans e.g. in a data file or folder structure and to extract the scan's meta information such as a *scan number* or *scan command*. The actual data for a scan must be read by an independent method.

The scan meta information and possibly also the scan data are stored in a `Scan` object, which provides a general but flexible interface between the `Source` and `Evaluation` classes. All `Scan` objects of a raw data source are stored in the `scan_dict` attribute of the `Source` object.

The `pyEvalData` modules provides several build-in `Source` classes, e.g. for spec, hdf5, and NeXus files. It can be easily extended by the user as explained in the write your own `Source` section. It is highly appreciated if new `Source` plugins are shared with the community.

In a future release, it will be possible to join two or multiple `Source` classes to create a `CompositeSource` object. This will be helpful to read separate raw data sources originating from the same experiment. A typical example is a scan file, such as a spec file and a folder structure containing camera images, which are linked to the data points in the scan file.

pyEvalData NeXus file

A rather general feature of the `pyEvalData` module is the usage of a NeXus file for converting the raw data in a common, structured, fast, and compressed data format. If enabled, the user can benefit from:

- a single data file containing all raw data - easy portability
- high degree of compression - saves disk space
- fast data access - saves computational time
- common and well documented structure - easy access also by external tools

Evaluation

The `Evaluation` class requires any kind of `Source` on initialization. Hence it has access to all available scans in the `scan_dict`. In addition to all available meta information and raw data, it allows for defining additional `counters` by user-defined algebraic expression, which can also handle nested expressions.

Variable injection of pre- and post-filter for the raw and evaluated data, respectively, enables to simplify common procedures such as outlier removal, offset removal, or normalization. In a future release, the filters will be provided as dedicated objects inheriting from a base `Filter` class. It will be possible to concatenate multiple filters and again a set of common filters will be available in the `pyEvalData` module, while adding new user-defined filters is explained in the write your own `Filter` section.

Further features of the `Evaluation` is to handle the *averaging* of multiple scans. Here, the case of multiple datasets with different x-grids is a very common but yet complex scenario. The `Evaluation` class will **never** interpolate any data, because one should avoid the *generation* of data-points. Instead, the data will be always *binned* onto an automatically-generated or user-defined x-grid. The underlying algorithm take also care of the correct error-calculations and can handle error-propagation as well as Possion statistics as required for single-photon-counting data.

Finally, the evaluated data can be easily plotted as well as fitted based on the `matplotlib` and `lmfit` modules. As a very common task it is easily possible to do the plotting and fitting for a sequence of one or multiple scans in dependence of an external parameter, such as a temperature series or alike.

4.1.2 Write your own Source

All you need to do is to define your own class which inherits from the Source class of the pyEvalData module. You can do so also directly in your evaluation script following this example containing some pseudo-code.

```
import pyEvalData as ped

class MyDataSource(ped.io.Source):
    """MyDataSource

    Here you should copy and adapt the doctring from the ``Source`` class for
    proper documentation

    """
    def __init__(self, file_name, file_path='./', **kwargs):
        super().__init__(file_name, file_path, **kwargs)

    def parse_raw(self):
        """parse_raw

        Parse the raw source file/folder and populate the `scan_dict`.

        """
        raw_scans = parse_my_raw_data(self.file_name, self.file_path)
        for rs in raw_scans:
            # create the Scan from the meta information
            scan = ped.io.Scan(int(rs.nr),
                               cmd=rs.command,
                               date=rs.date,
                               time=rs.time,
                               int_time=rs.int_time,
                               header=rs.header,
                               init_mopo=rs.init_motor_pos)
            # store the scan in the scan_dict
            self.scan_dict[spec_scan.nr] = scan

            # check if the data needs to be read as well, if not it will be
            # read on demand later
            if self.read_all_data:
                self.read_scan_data(self.scan_dict[spec_scan.nr])

    def read_raw_scan_data(self, scan):
        """read_raw_scan_data

        Reads the data for a given scan object from raw source.

        Args:
            scan (Scan): scan object.

        """
        # read the actual data
        raw_scan_data = read_my_raw_scan_data(scan.number)
        # set the data of the scan object
```

(continues on next page)

(continued from previous page)

```
        scan.data = raw_scan_data
```

4.1.3 Write your own `Filter`

comming soon ...

4.2 Examples

4.2.1 Source

The `Source` class provides an interface and predefined methods for reading raw data from file. If `source.use_nexus == True` the raw data will be saved and read preferentially from a generated `NeXus` file, which results in a single data file with high compression and fast data access.

The `Source` class provides a `dict()` of all scans, called `scan_dict`. The items in this dictionary are instances of the `Scan` class, which defines the common data structure. However, the access to the scan data should **NOT** be done via the `scan_dict` directly, but instead for a deticated scan via `Source.get_scan_data(scan_number)` or for a list of scans via `Source.get_scan_list_data(scan_number_list)`. By this, all features of the `Source` class can be utilized.

One key concept of the `pyEvalData` is the idea, that `Scan` instances must only provide the meta information of the scan but not necessarily its data. By default, the scan data will only be read on request and is then stored in the `Scan` object for later use. It is possible, however, to read all data to each scan directly on first parsing by `Source.read_all_data = Ture`. On the other hand, if the data is allocating too much memory, it is possible to clear the data from the `Scan` object, directly after accessing it, via `Source.read_and_forget = True`. The flag `Source.update_before_read` enables parsing the raw source file to search and add new scans before accessing any other scans. By default the last `Scan` in the `scan_dict` will always be re-created in case new data was added. The flag `Source.force_overwrite` will force a full parsing of the raw source file and a complete overwrite of the `NeXus` file on every `update()`.
With these set of parameters the user is very flexible in how and if the raw data is read and stored and can adapt the `Source` behaviour for different situations such as *on-line* anaylsis during a beamtime or *post* analysis later at home.

The following examples are meant for introducing the low-level access to raw data via the `Source` layer. This can be helpful for direct access to raw data and integration into existing scripts and applications.
However, user will generally access the `Source` via the `Evaluation` class as described in the *upcoming examples*.

Setup

Here we do the necessary import for this example

```
import matplotlib.pyplot as plt
import numpy as np

import pyEvalData as ped
# define the path for the example data
example_data_path = '../../../example_data/'
```

SPEC

A very common data source are SPEC files from the original Certif spec application as well as from the open-source alternative Sardana. The `Spec` source relies on the great parser provided by xrayutilities.

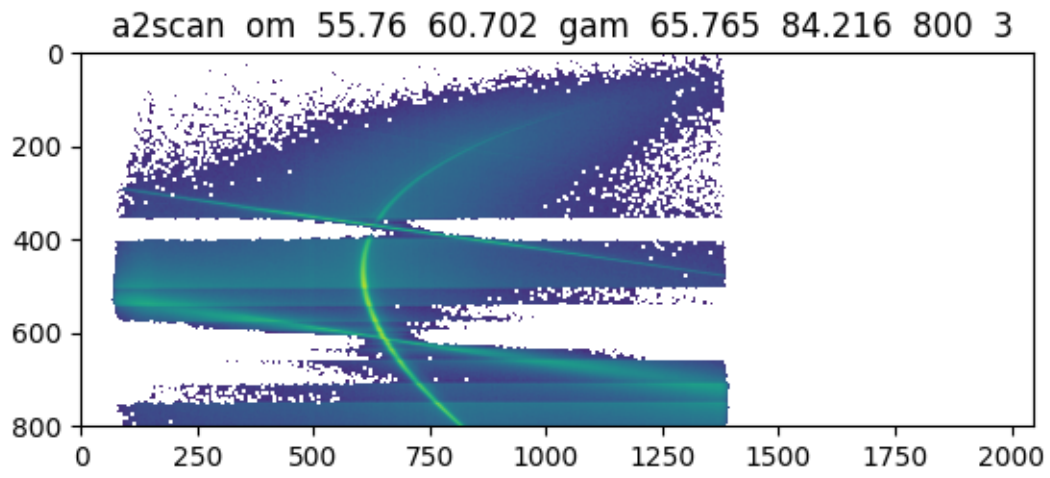
```
spec = ped.io.Spec(file_name='certif_xu.spec',
                   file_path=example_data_path,
                   use_nexus=True,
                   force_overwrite=False,
                   update_before_read=False,
                   read_and_forget=True,
                   nexus_file_path='./')
```

```
pyEvalData.io.source - INFO: Update source
```

```
pyEvalData.io.source - INFO: parse_nexus
```

As described above, the scan data and meta information should be accessed via `spec.get_scan_data(scan_number)`.
Here, `data` is returned as `numpy.recarray` and `meta` is a `dict()`.

```
data, meta = spec.get_scan_data(33)

plt.figure()
plt.imshow(np.log10(data['MCA']))
plt.title(meta['cmd'])
plt.show()
```



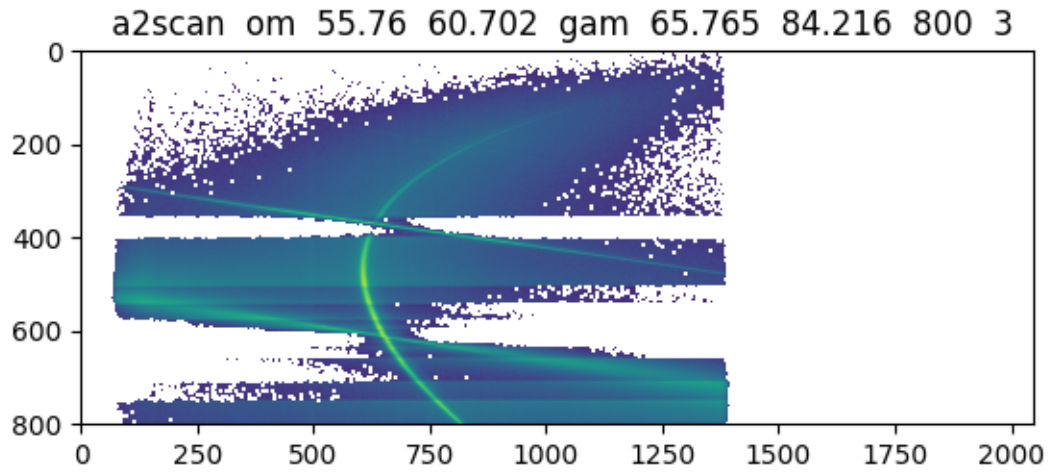
One can also directly work with the `Scan` objects, which provides nearly the same functionality as above but misses to automatically `read_and_forget` the data. Below, the example from above is reproduces with the `Scan`. Here one can also easily access the meta information as attributes of the `Scan` instance.

```
scan = spec.get_scan(33)

plt.figure()
plt.imshow(np.log10(scan.MCA))
```

(continues on next page)

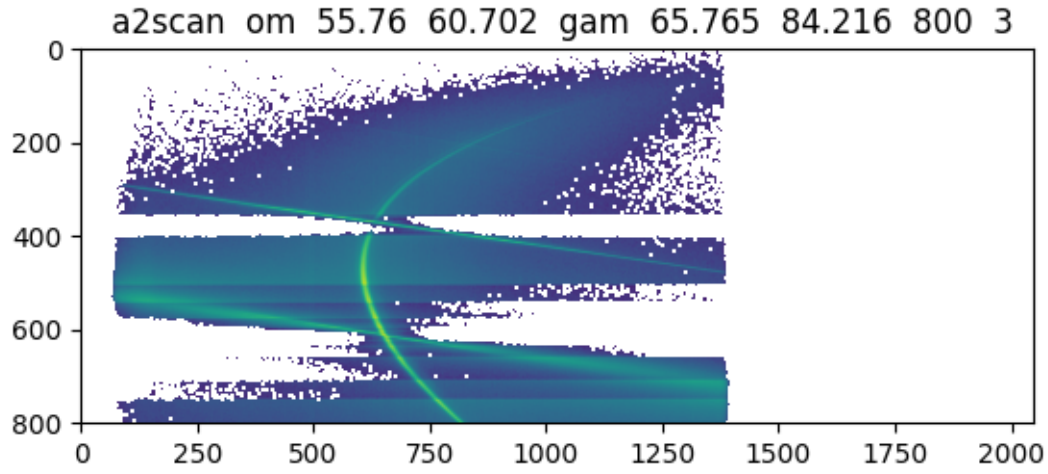
(continued from previous page)

```
plt.title(scan.cmd)
plt.show()
```



It is also possible to access the scan directly as attribute of the source which is fully aquivalent to the last example.

```
plt.figure()
plt.imshow(np.log10(spec.scan33.MCA))
plt.title(spec.scan33.cmd)
plt.show()
```



Sardana NeXus

The NeXus file format is a common standard in science for storing and exchanging data. It is based on hdf5 files with a hierarchical structure. As an example we read the NeXus files as created by Sardana. As we can directly access the data from the NeXus files, there is no need to enable additional NeXus export so one can set `source.use_nexus = False`.

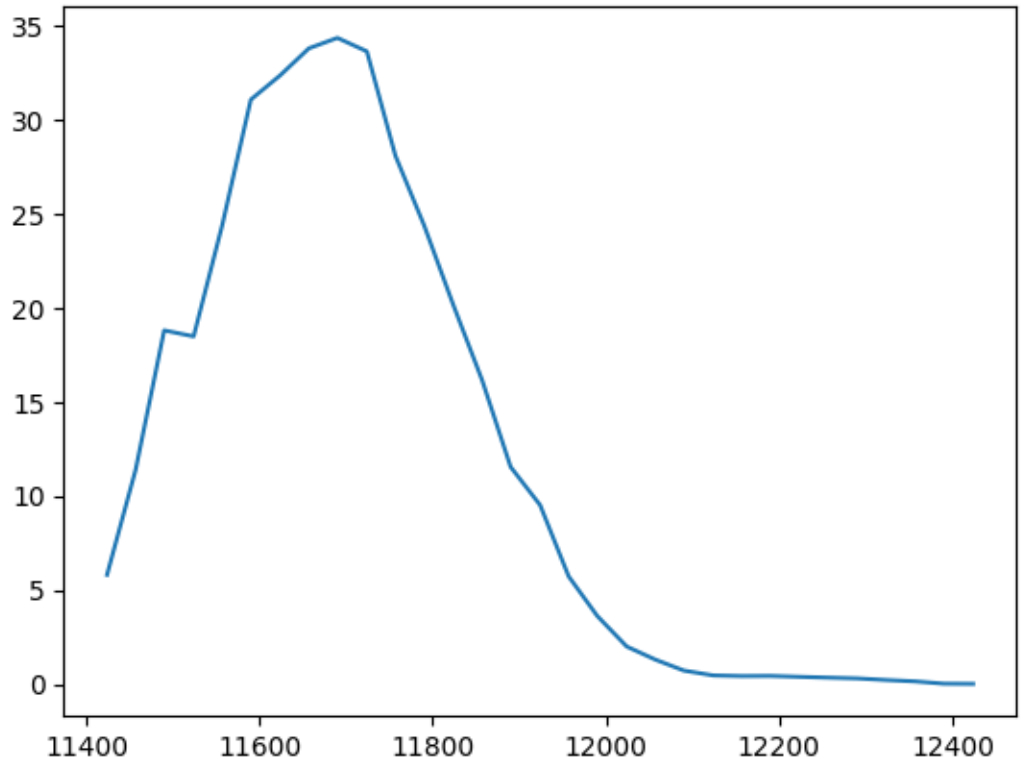
```
sarnxs = ped.io.SardanaNeXus(file_name='sardana_nexus.h5',
                             file_path=example_data_path,
                             use_nexus=False,
                             force_overwrite=False,
                             update_before_read=False,
                             read_and_forget=True)
```

```
pyEvalData.io.source - INFO: Update source
```

```
pyEvalData.io.source - INFO: parse_raw
```

```
plt.figure()
plt.plot(sarnxs.scan435.mhor, sarnxs.scan435.pmPump)
plt.show()
```

```
pyEvalData.io.source - INFO: read_raw_scan_data for scan #435
```



hdf5 - PAL FEL

hdf5 is also a very common data format. In this example the raw data was measured at PAL FEL in South Korea. The individual files are grouped in a single folder per scan. This is handled by the `fprint`-statement for the `file_name` is is automatically evaluated to provide the correct file and folder name for a given scan number.

```
pal = ped.io.PalH5(name='2020_12_Schick',
                   file_name='{0:07d}',
                   file_path=example_data_path+'/pal_fel',
                   use_nexus=False,
                   force_overwrite=False,
                   update_before_read=True,
                   read_and_forget=True,
                   nexus_file_path='./',
                   nexus_file_name='2020_12_Schick')
```

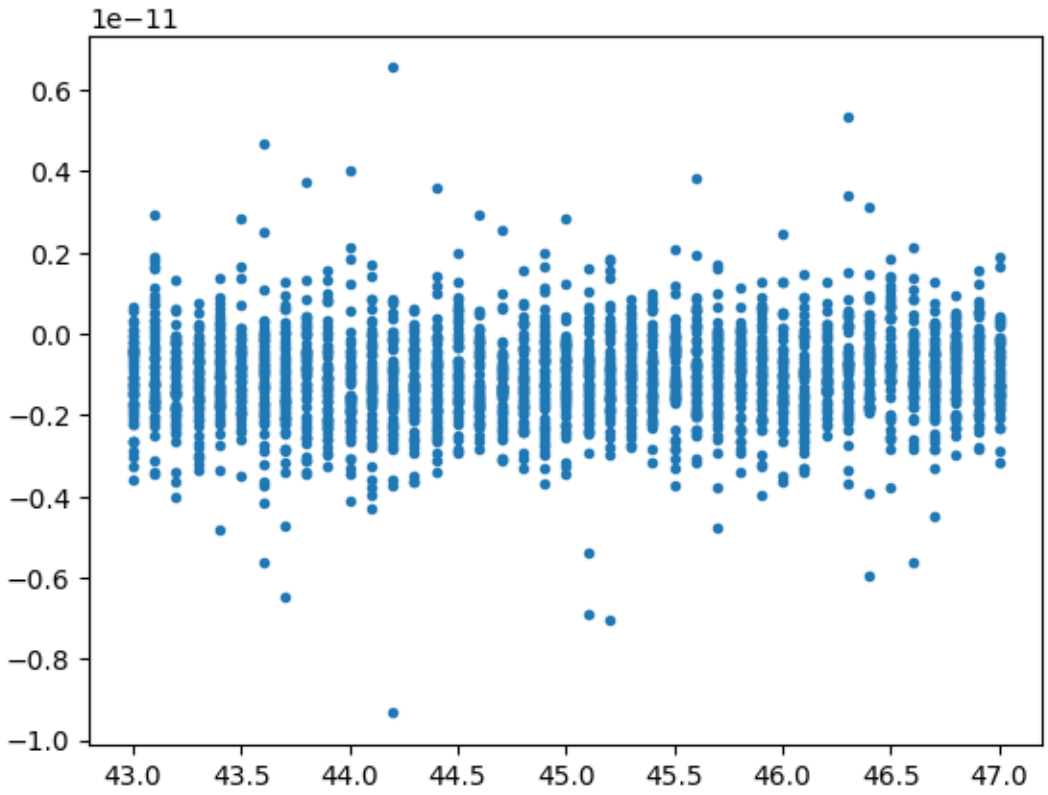
```
pyEvalData.io.source - INFO: Update source
```

```
pyEvalData.io.source - INFO: parse_raw
```

Again we have easy access to the raw data similar to the former examples.

```
plt.figure()
plt.plot(pal.scan9.th, pal.scan9.digi1_1, '.')
plt.show()
```

```
pyEvalData.io.source - INFO: read_raw_scan_data for scan #9
```



Composite Sources

idea for a future release

One specific idea of the `Source` class is to provide also composite models of pre-defined `Source` classes. An example is the `Spec` source for reading SPEC data files and a `Source` class to read camera images from a folder structure, that have been acquired simultaneously with the SPEC file. This would require to specify the two independent `Souce` objects and then add them to a single `CompositeSource`, e.g. by:

```
spec = ped.io.Spec(file_name, file_path)
ccd = ped.io.Pilatus(image_pattern, image_base_path)

comp_source = spec + ccd
```

After this step the workflow is the same as with all other `Source` classes. The concatenation should also allow for multiple items. Special care must be take for duplicate `motor` and `counter` names. This would require a `prefix` scheme similar to the `lmfit` package.

4.2.2 Evaluation

The `Evaluation` class is the actual interface for the user with the functionalities of the `pyEvalData` package. It provides access to the raw data via the `Source` object, given on initialization.
The main features of the `Evaluation` class are the definition of counter aliases as well as new counters by simple algebraic expressions. At the same time pre- and post-filters can be applied to the raw and evaluated data, respectively. Much efforts have been put into the binning, averaging, and error calculation of the raw data. In addition to the evaluation of a list of scans or scan sequence of one or multiple scans in dependence of an external paramter, the `Evaluation` class also provides high-level helper functions for plotting and fitting the according results.

Setup

Here we do the necessary import for this example

```
%load_ext autoreload
%autoreload 2

import matplotlib.pyplot as plt
import numpy as np

import pyEvalData as ped
# import lmfit for fitting
import lmfit as lf
# import some usefull fit functions
import ultrafastFitFunctions as ufff
# define the path for the example data
example_data_path = '../../../example_data/'
```

Source

Here we iitialize the `Source` for the current evaluation. It is based on raw data in a SPEC file which was generated by the open-source software Sardana.

```
spec = ped.io.Spec(file_name='sardana_spec.spec',
                   file_path=example_data_path,
                   use_nexus=True,
                   force_overwrite=False,
                   update_before_read=False,
                   read_and_forget=True)
```

```
pyEvalData.io.source - INFO: Update source
```

```
pyEvalData.io.source - INFO: parse_nexus
```

The `Evaluation` class

For the most basic example we just have to provide a `Source` on initialization:

```
ev = ped.Evaluation(spec)
```

Now it is possible to check the available attributes of the `Evaluation` object, which will be explained step-by-step in the upcominng sections.

```
print(ev.__doc__)
```

```
Evaluation

    Main class for evaluating data.
    The raw data is accessed via a ``Source`` object.
    The evaluation allows to bin data, calculate errors and propagate them.
    There is also an interface to ``lmfit`` for easy batch-fitting.

    Args:
        source (Source): raw data source.

    Attributes:
        log (logging.logger): logger instance from logging.
        clist (list[str]): list of counter names to evaluate.
        cdef (dict{str:str}): dict of predefined counter names and
            definitions.
        xcol (str): counter or motor for x-axis.
        t0 (float): approx. time zero for delay scans to determine the
            unpumped region of the data for normalization.
        custom_counters (list[str]): list of custom counters - default is []
        math_keys (list[str]): list of keywords which are evaluated as numpy functions
        statistic_type (str): 'gauss' for normal averaging, 'poisson' for counting␣
↪statistics
        propagate_errors (bool): propagate errors for dpendent counters.
```

Most of the attributes of the `Evaluation` class are well explained in the `docstring` above and described in more details below.
The `custom_counters` might be soon depricated.
The `t0` attribute is used for easy normalization to 1 by dividing the data by the mean of all values which are `xcol < t0`. This is typically useful for time-resolved delay scans but might be renamed for a more general approach in the future.
The `statistics_type` attribute allows to switch between *gaussian* statistics, which calculates the error from the standard derivation, and *poisson* statistics, whichs calculates the error from $1/\sqrt N$ with N being the total number of photons in the according bin.

Simple plot example

To plot data, the `Evlauation` objects does only need to know the `xcol` as horizontal axis as well a list of *counters* to plot, which is called `clist`.

First we can check the available scan numbers in the source:

```
spec.get_all_scan_numbers()
```

```
[1, 2, 3, 4, 5, 6]
```

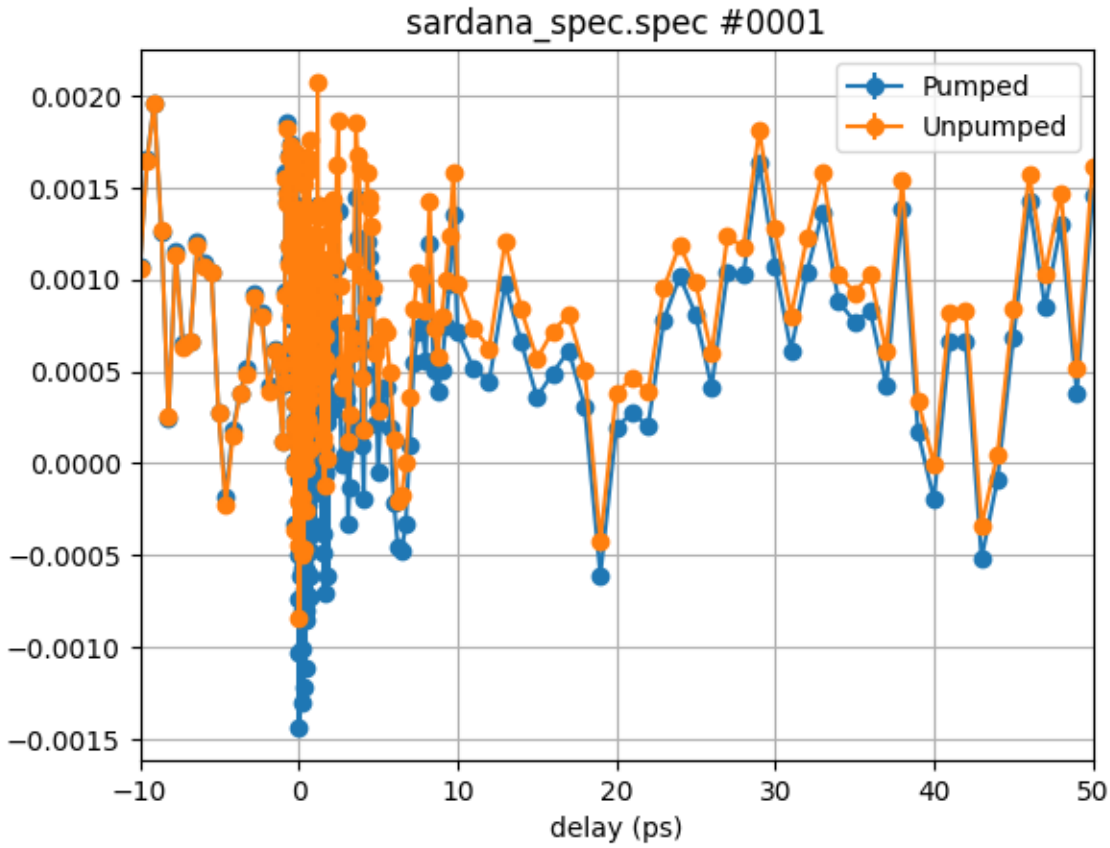
Now we can check for the available data for a specific scan

```
spec.scan1.data.dtype.names
```

```
('Diff',
 'DiffM',
 'Pt_No',
 'Pumped',
 'PumpedErr',
 'PumpedErrM',
 'PumpedM',
 'Rel',
 'RelM',
 'Unpumped',
 'UnpumpedErr',
 'UnpumpedErrM',
 'UnpumpedM',
 'chirp',
 'delay',
 'dt',
 'duration',
 'durationM',
 'envHumid',
 'envTemp',
 'freqTriggers',
 'magneticfield',
 'numTriggers',
 'numTriggersM',
 'thorlabsPM',
 'thorlabsPPM',
 'thorlabsPPMonitor')
```

So let's try to plot the counters `Pumped` and `Unpumped` vs the motor `delay` for scan #1:

```
ev.xcol = 'delay'
ev.clist = ['Pumped', 'Unpumped']

plt.figure()
ev.plot_scans([1])
plt.xlim(-10, 50)
plt.xlabel('delay (ps)')
plt.show()
```



Algebraic expressions

For now, we only see a lot of noise. So let's work a bit further on the data we plot. The experiment was an ultrafast MOKE measurement, which followed the polarization rotation of the probe pulse after a certain `delay` in respect to the pump pulse. Typically, this magnetic contrast is improved by subtracting the measured signal for two opposite magnetization directions of the sample, as the MOKE effect depends on the sample's magnetization.

In our example, we have two additional *counters* available, which contain the data for negative magnetic fields (*M* - minus): `PumpedM` and `UnpumpedM`
While the two former *counters* were acquired for positive fields.

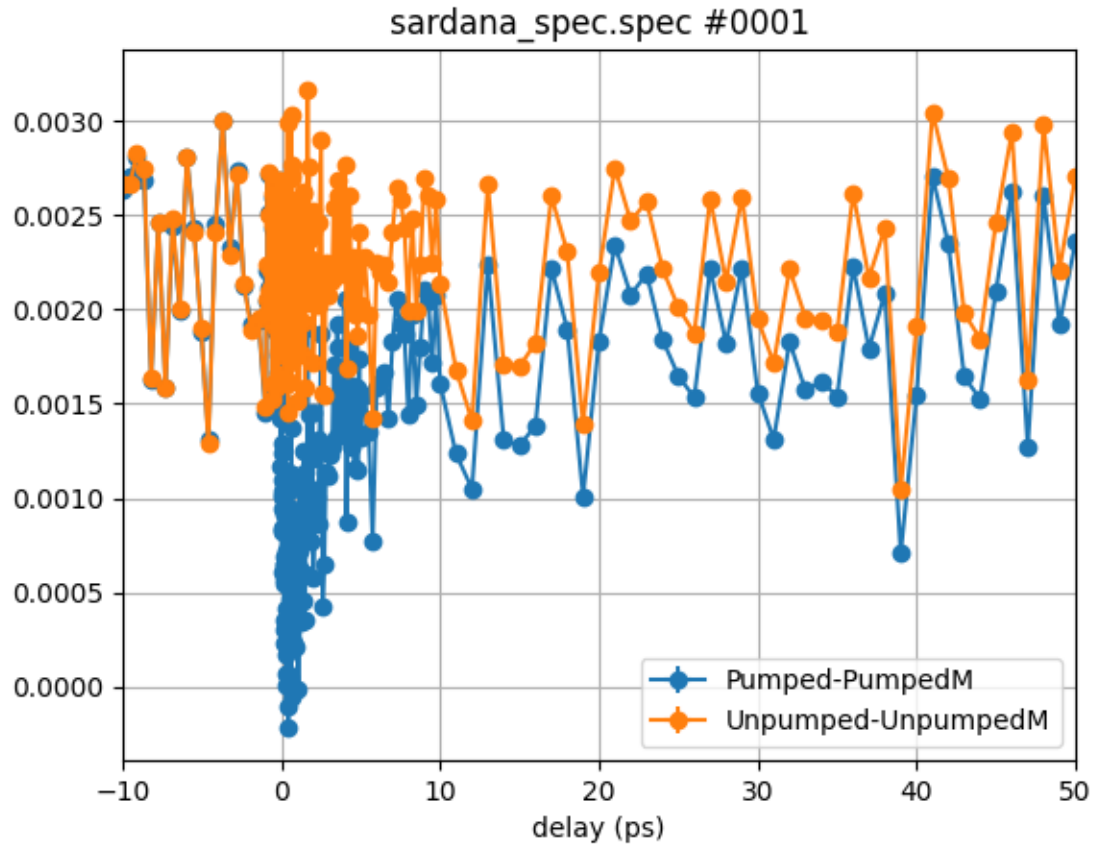
Let's plot the difference signal for the *pumped* and *unpumped* signals:

```
ev.xcol = 'delay'
ev.clist = ['Pumped-PumpedM', 'Unpumped-UnpumpedM']
```

(continues on next page)

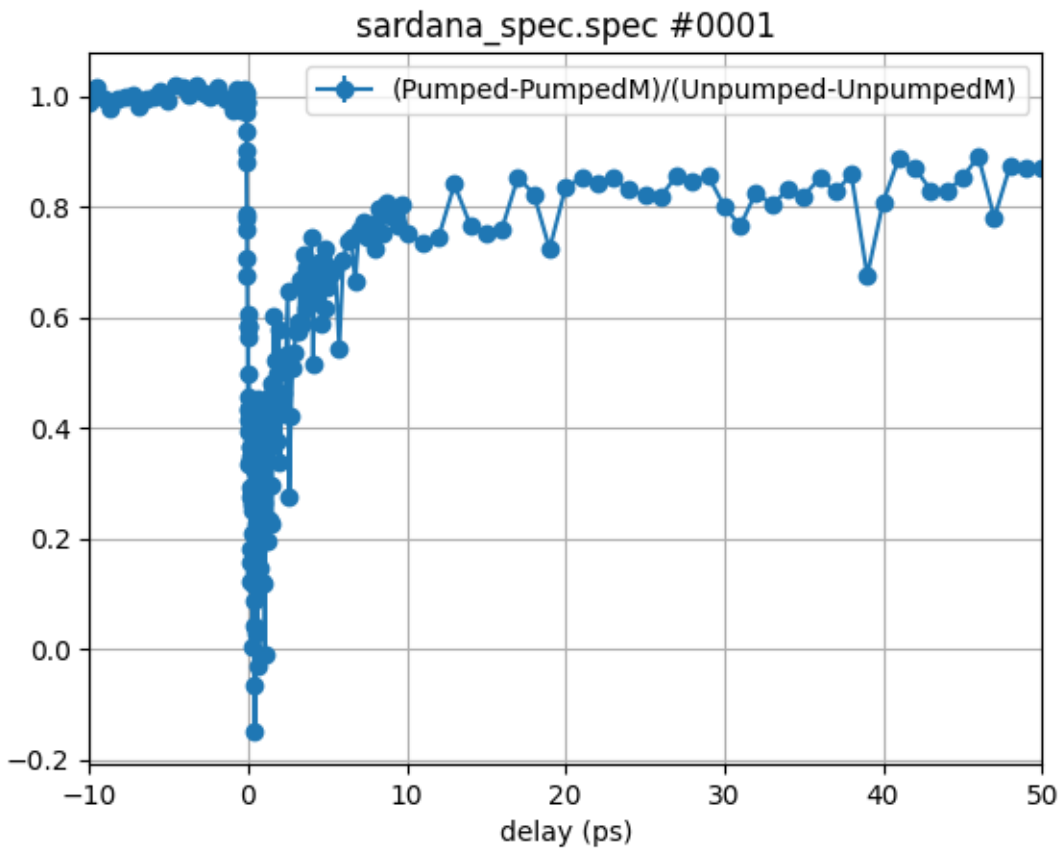
(continued from previous page)

```
plt.figure()
ev.plot_scans([1])
plt.xlim(-10, 50)
plt.xlabel('delay (ps)')
plt.show()
```



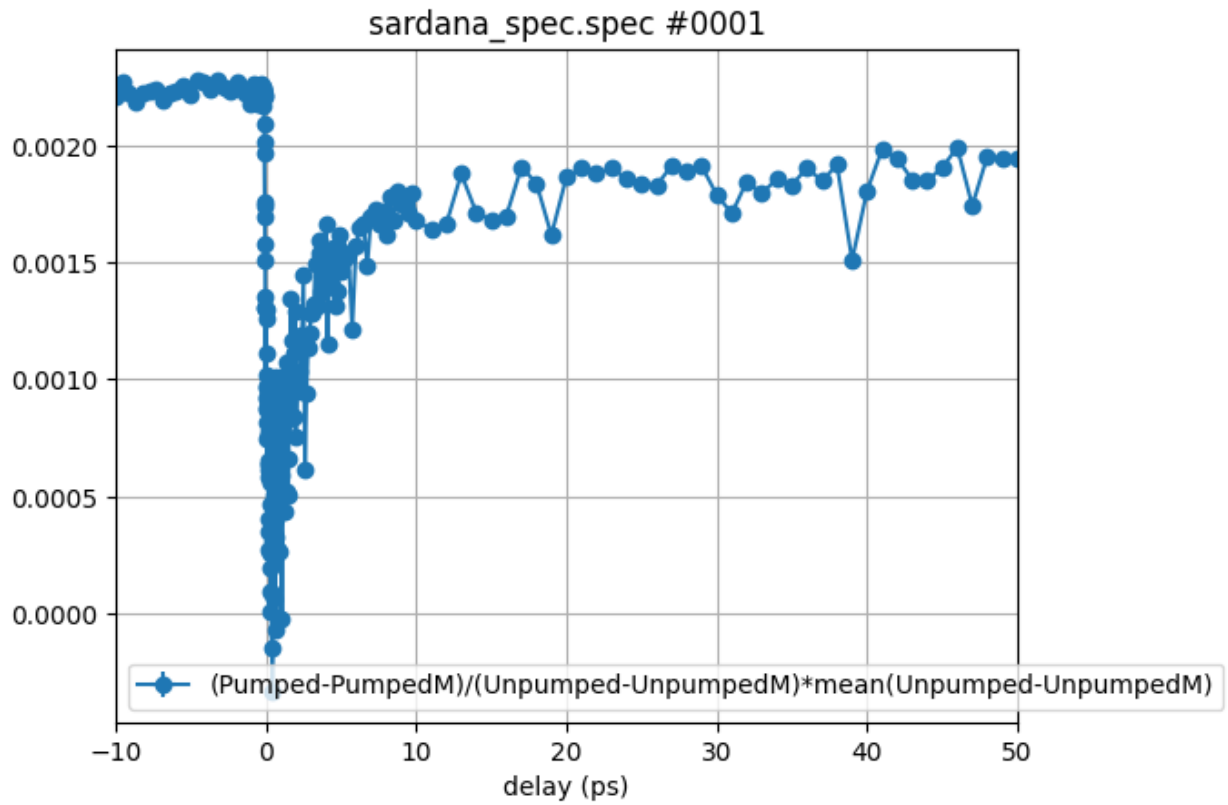
The new plot shows already much more dynamisc in the *pumped* vs. the *unpumped* signal. However, we can still improve that, by normalizing one by the other:

```
ev.xcol = 'delay'
ev.clist = ['(Pumped-PumpedM)/(Unpumped-UnpumpedM)']

plt.figure()
ev.plot_scans([1])
plt.xlim(-10, 50)
plt.xlabel('delay (ps)')
plt.show()
```



This does look much better but we lost the absolute value of the contrast. Let's simply multiply the trace with the average of the *unpumped* magnetic contrast:

```
ev.xcol = 'delay'
ev.clist = ['(Pumped-PumpedM)/(Unpumped-UnpumpedM)*mean(Unpumped-UnpumpedM)']

plt.figure()
ev.plot_scans([1])
plt.xlim(-10, 50)
plt.xlabel('delay (ps)')
plt.show()
```



So besides simple operations such as `+. -. *, /` we can also use some basic `numpy` functionalities. You can check the available functions by inspection of the attribute `math_keys`:

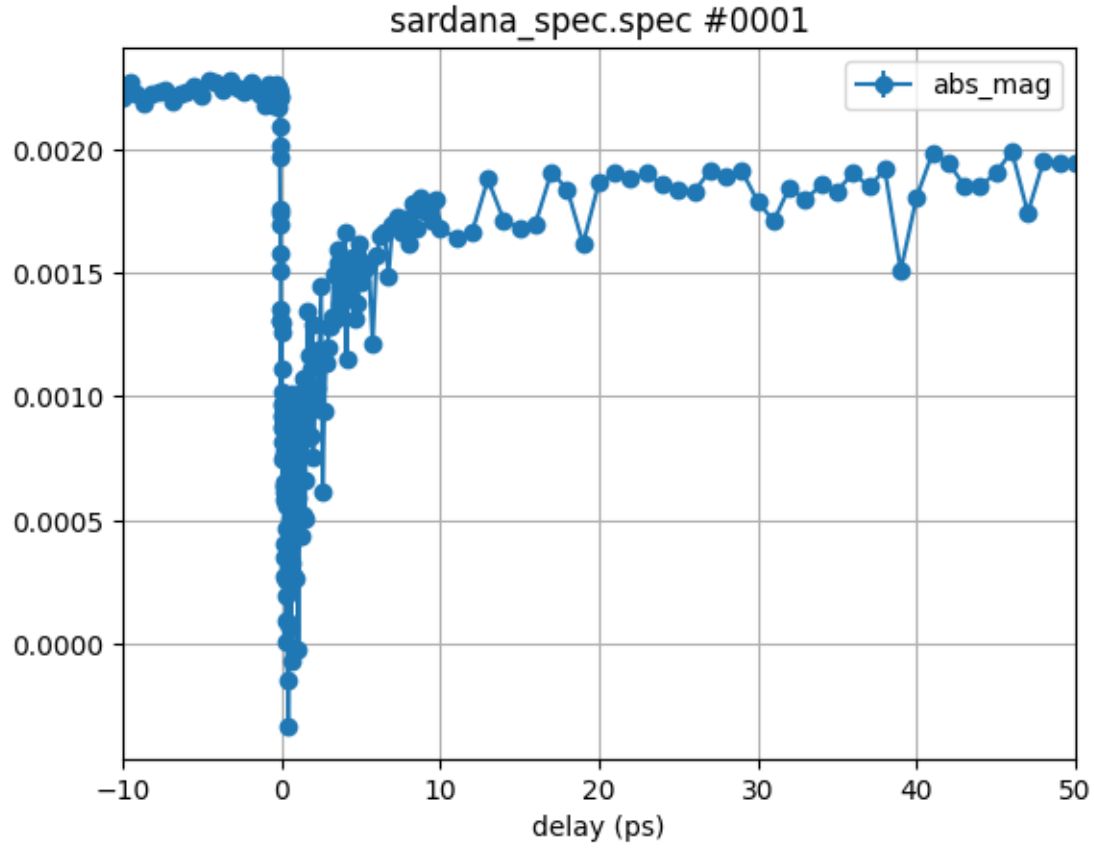
```
ev.math_keys
```

```
['mean',
 'sum',
 'diff',
 'max',
 'min',
 'round',
 'abs',
 'sin',
 'cos',
 'tan',
 'arcsin',
 'arccos',
 'arctan',
 'pi',
 'exp',
 'log',
 'log10',
 'sqrt',
 'sign']
```

But of course our current *counter* name is rather bulky. So lets define some aliases using the attribute `cdef`:

```
ev.cdef['pumped_mag'] = 'Pumped-PumpedM'
ev.cdef['unpumped_mag'] = 'Unpumped-UnpumpedM'
ev.cdef['rel_mag'] = 'pumped_mag/unpumped_mag'
ev.cdef['abs_mag'] = 'pumped_mag/unpumped_mag*mean(unpumped_mag)'
```

```
ev.xcol = 'delay'
ev.clist = ['abs_mag']

plt.figure()
ev.plot_scans([1])
plt.xlim(-10, 50)
plt.xlabel('delay (ps)')
plt.show()
```

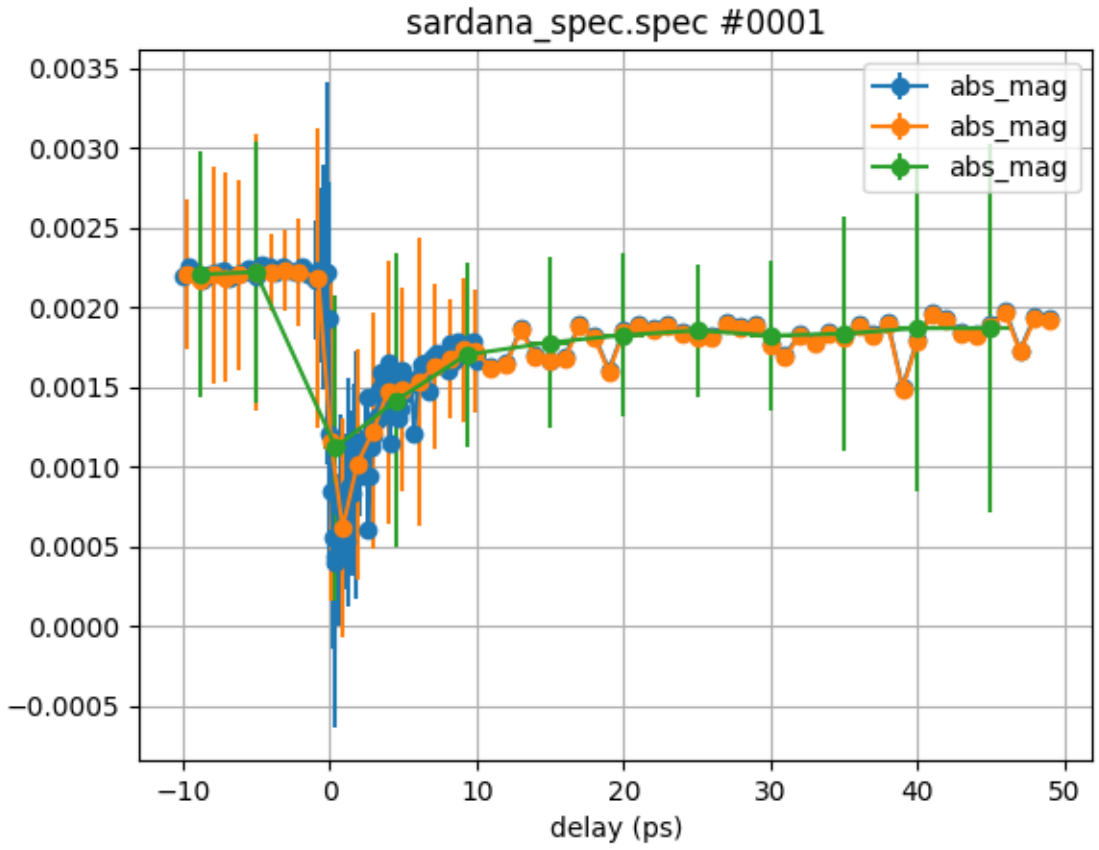


Binning

In many situations it is desireable to reduce the data density or to plot the data on a new grid. This can be easily achieved by the `xgrid` keyword of the `plot_scans` method.

Here we plot the same data as before on a three reduced grids with 0.1, 1, and 5 ps step width. Please note that the errorbars appear due to the averaging of multiple point in the bins of the grid. The errorbars are vertical and horizontal. We can also skip the `xlim` setting here, as our grid is in the same range as before.

```
ev.xcol = 'delay'
ev.clist = ['abs_mag']

plt.figure()
ev.plot_scans([1], xgrid=np.r_[-10:50:0.1])
ev.plot_scans([1], xgrid=np.r_[-10:50:1])
ev.plot_scans([1], xgrid=np.r_[-10:50:5])
plt.xlabel('delay (ps)')
plt.show()
```

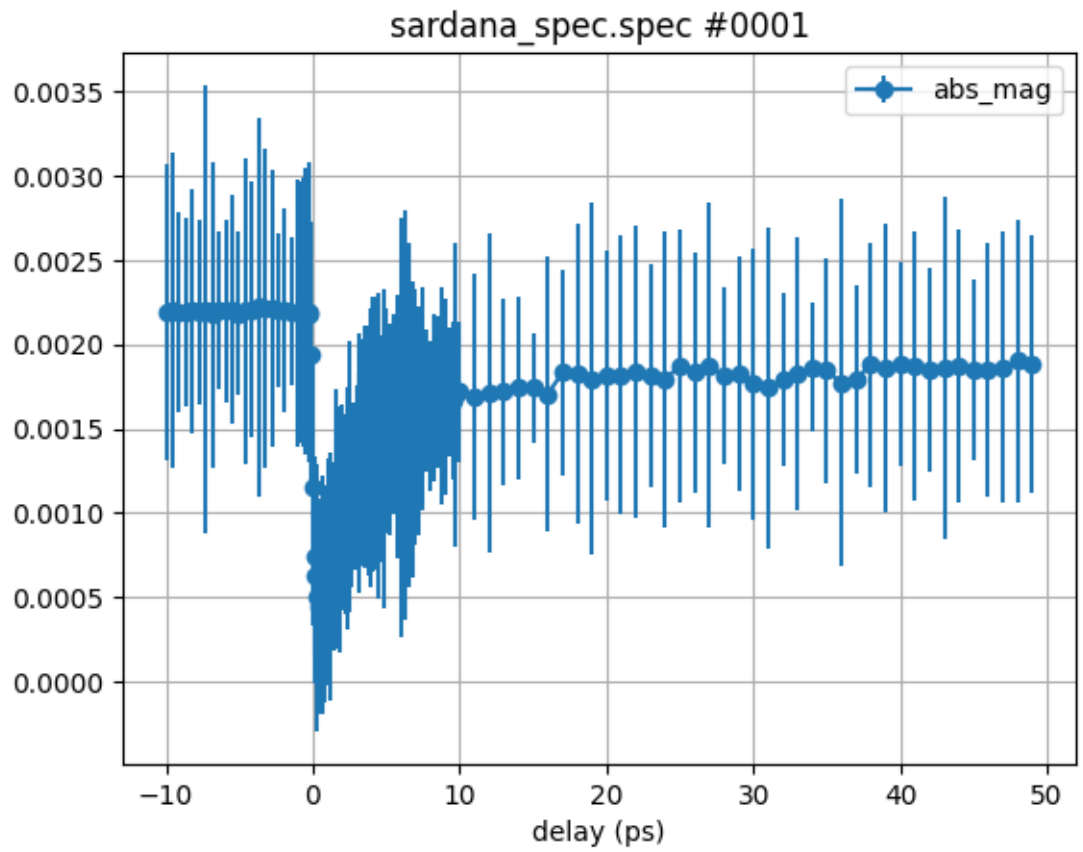


Averaging & error propagation

In order to improve statistics even further, scans are often repeated an averaged. This was also done for this experimental example and all scans #1-6 were done with the same settings.

We can simply average them by providing all scans of interest to the `plot_scans` method:

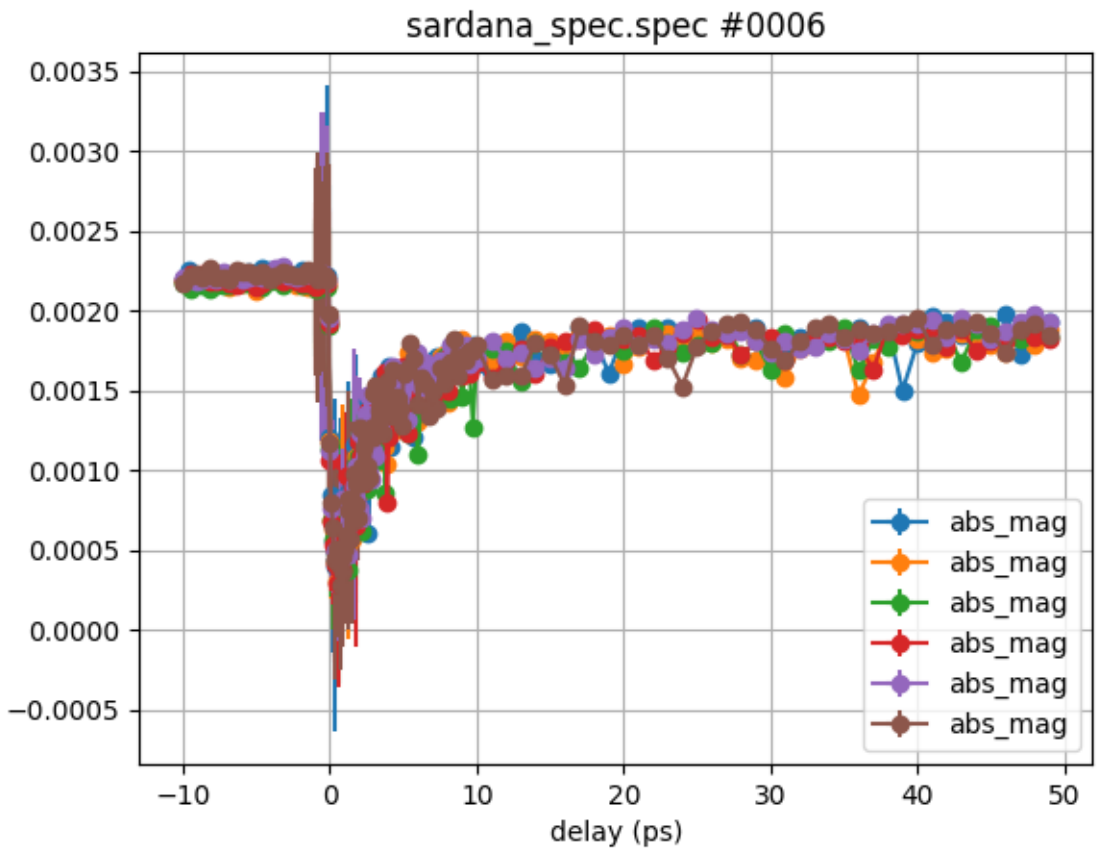
```
ev.xcol = 'delay'
ev.clist = ['abs_mag']

plt.figure()
ev.plot_scans([1, 2, 3, 4, 5, 6], xgrid=np.r_[-10:50:0.1])
plt.xlabel('delay (ps)')
plt.show()
```



Hm, somehow this did not really did the job, right? Altough the scattering of the circle symbols has decreased, the errorbars are much large as for the single scan before.

Let's check the individual scans to see what happened:

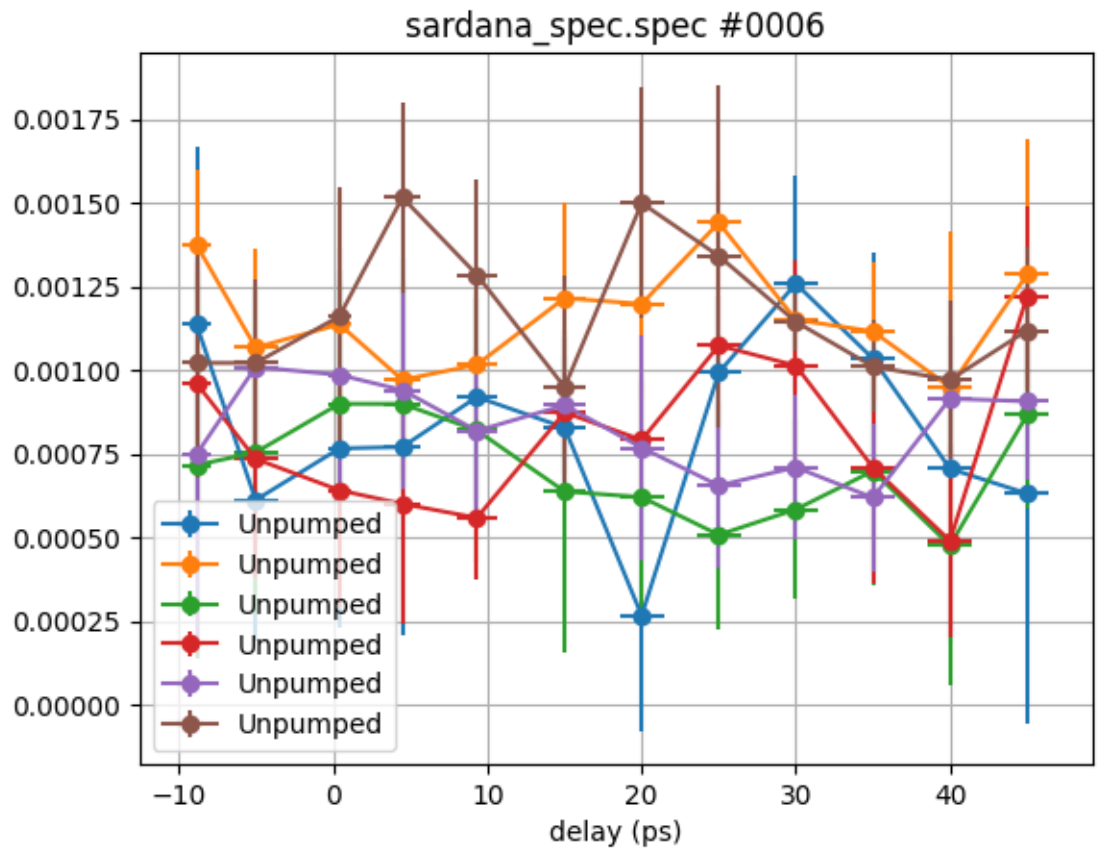
```
ev.xcol = 'delay'
ev.clist = ['abs_mag']

plt.figure()
ev.plot_scans([1], xgrid=np.r_[-10:50:0.1])
ev.plot_scans([2], xgrid=np.r_[-10:50:0.1])
ev.plot_scans([3], xgrid=np.r_[-10:50:0.1])
ev.plot_scans([4], xgrid=np.r_[-10:50:0.1])
ev.plot_scans([5], xgrid=np.r_[-10:50:0.1])
ev.plot_scans([6], xgrid=np.r_[-10:50:0.1])
plt.xlabel('delay (ps)')
plt.show()
```



Individually all scans look very much the same, with very small errorbars. So why do we get so large errorbars when we average them?

Let's go one more step back and plot the `Unpumped` signal for all scans with a large grid of 5 ps for clarity:

```
ev.xcol = 'delay'
ev.clist = ['Unpumped']

plt.figure()
ev.plot_scans([1], xgrid=np.r_[-10:50:5])
ev.plot_scans([2], xgrid=np.r_[-10:50:5])
ev.plot_scans([3], xgrid=np.r_[-10:50:5])
ev.plot_scans([4], xgrid=np.r_[-10:50:5])
ev.plot_scans([5], xgrid=np.r_[-10:50:5])
ev.plot_scans([6], xgrid=np.r_[-10:50:5])
plt.xlabel('delay (ps)')
plt.show()
```



We can observe a significant drift of the raw data which results in deviations that are not statistically distributed anymore.

This essentially means, that it makes a difference if we

1. evaluate the expression `abs_mag` for every scan individually and eventually average the resulting traces
2. first average the raw data (`Pumped, PumpedM, Unpumped, UnpumpedM`) and then calculate final trace for `abs_mag` using the averaged raw data. In the later case we need to carry out a proper error propagation to determine the errors for `abs_mag`.

The `Evaluation` class allows to switch between both cases by the attribute flag `propagate_errors` which is `True` by default and handles the error propagation automatically using the uncertainties package. For our last example we were follwoing option 2. as described above. Accoridngly, rather large errors from the drifiting of the raw signals were propagated.

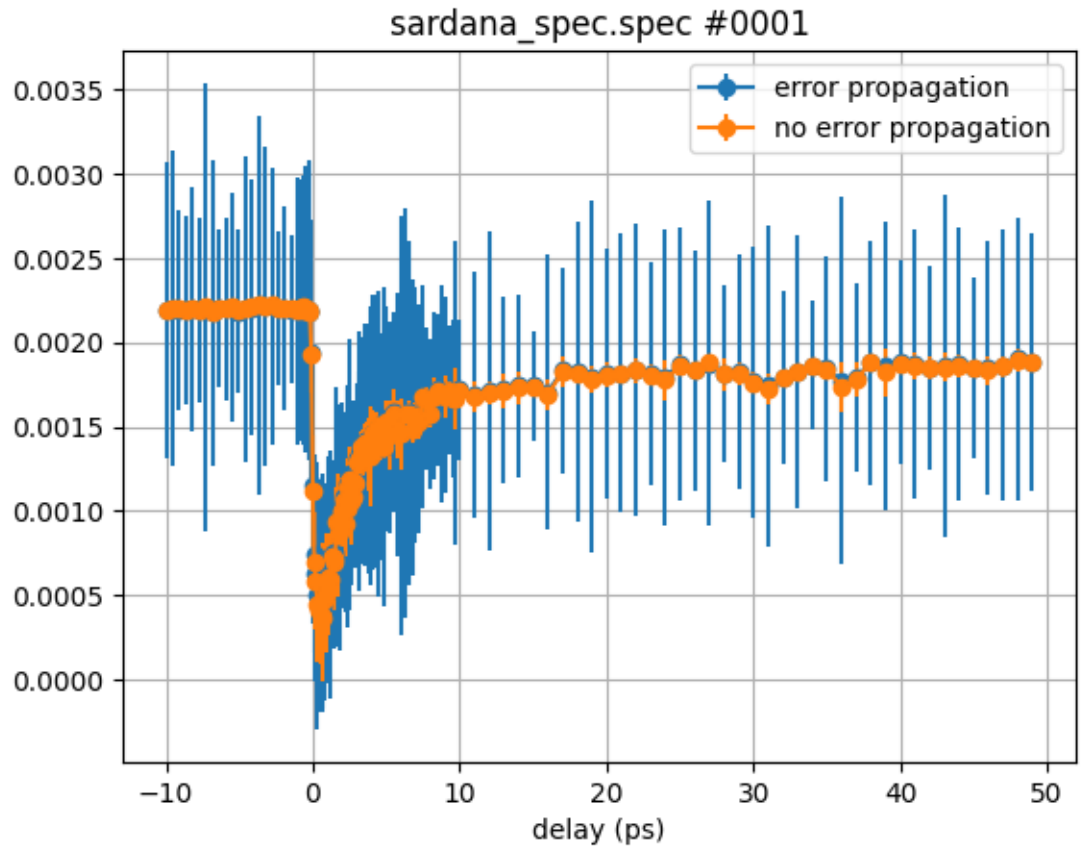
Now lets compare to option 1. without error propagation:

```
ev.xcol = 'delay'
ev.clist = ['abs_mag']

plt.figure()
ev.propagate_errors = True
ev.plot_scans([1, 2, 3, 4, 5, 6], xgrid=np.r_[-10:50:0.1])

ev.propagate_errors = False
ev.plot_scans([1, 2, 3, 4, 5, 6], xgrid=np.r_[-10:50:0.1])
plt.xlabel('delay (ps)')
```

(continues on next page)

(continued from previous page)

```
plt.legend(['error propagation', 'no error propagation'])
plt.show()
```



The application of the both options strongly depends on the type of noise and drifts of the acquired data.

`plot_scans()` options

Let's check all arguments of the `plot_scans()` method by simply calling:

```
help(ev.plot_scans)
```

```
Help on method plot_scans in module pyEvalData.evaluation:

plot_scans(scan_list, ylims=[], xlims=[], fig_size=[], xgrid=[], yerr='std', xerr='std',␣
↪norm2one=False, binning=True, label_text='', title_text='', skip_plot=False, grid_
↪on=True, ytext='', xtext='', fmt='-o') method of pyEvalData.evaluation.Evaluation␣
↪instance
    Plot a list of scans from the spec file.
    Various plot parameters are provided.
    The plotted data are returned.

    Args:
        scan_list (List[int])        : List of scan numbers.
```

(continues on next page)

(continued from previous page)

```
        ylims (Optional[ndarray])   : ylim for the plot.
        xlims (Optional[ndarray])   : xlim for the plot.
        fig_size (Optional[ndarray]) : Figure size of the figure.
        xgrid (Optional[ndarray])   : Grid to bin the data to -
                                      default in empty so use the
                                      x-axis of the first scan.
        yerr (Optional[ndarray])    : Type of the errors in y: [err, std, none]
                                      default is 'std'.
        xerr (Optional[ndarray])    : Type of the errors in x: [err, std, none]
                                      default is 'std'.
        norm2one (Optional[bool])   : Norm transient data to 1 for t < t0
                                      default is False.
        label_text (Optional[str])   : Label of the plot - default is none.
        title_text (Optional[str])   : Title of the figure - default is none.
        skip_plot (Optional[bool])   : Skip plotting, just return data
                                      default is False.
        grid_on (Optional[bool])     : Add grid to plot - default is True.
        ytext (Optional[str])       : y-Label of the plot - defaults is none.
        xtext (Optional[str])       : x-Label of the plot - defaults is none.
        fmt (Optional[str])         : format string of the plot - defaults is -o.

    Returns:
        y2plot (OrderedDict)    : y-data which was plotted.
        x2plot (ndarray)        : x-data which was plotted.
        yerr2plot (OrderedDict) : y-error which was plotted.
        xerr2plot (ndarray)     : x-error which was plotted.
        name (str)              : Name of the data set.
```

Most of the above arguments are *plotting options* and will be changed/simplified in a future release.
The `xerr` and `yerr` arguments allow to change the type of errorbars in `x` and `y` direction between *standard error, standard derivation* and *no error*.
The `norm2one` flag allows to normalize the data to 1 for all data which is before `Evaluation.t0` on the `xcol`.
The `skip_plot` option disables plotting at all and can be handy if only access to the return values is desired.
The returned data contains the `xcol` and according error as `ndarray` named `x2plot` and `xerr2plot`, while the according counters and errors from the `clist` are given as `OrderedDicts` `y2plot` and `yerr2plot`. The `keys` of these dictionaries correspond to the elements in the `clist`.

Scan sequences

Experimentally it is common to repeat similar scans while varying an external parameter, such as the sample environment (temperature, external fields, etc.).
For this common task, the `Evaluation` class provides a method named `plot_scan_sequence()` which wraps around the `plot_scans()` method.

First, we have to define the `scan_sequence` as a nested list to be correctly parsed by the `plot_scan_sequence()` method. For that, we use the day time of the scans as an external parameter. We can access such meta information directly from the `Source` object as follows:

```
print(spec.scan1.time)
print(spec.scan2.time)
print(spec.scan3.time)
print(spec.scan4.time)
```

(continues on next page)

(continued from previous page)

```
print(spec.scan5.time)
print(spec.scan6.time)
```

```
21:28:49
00:05:09
02:27:56
05:03:44
07:39:54
```

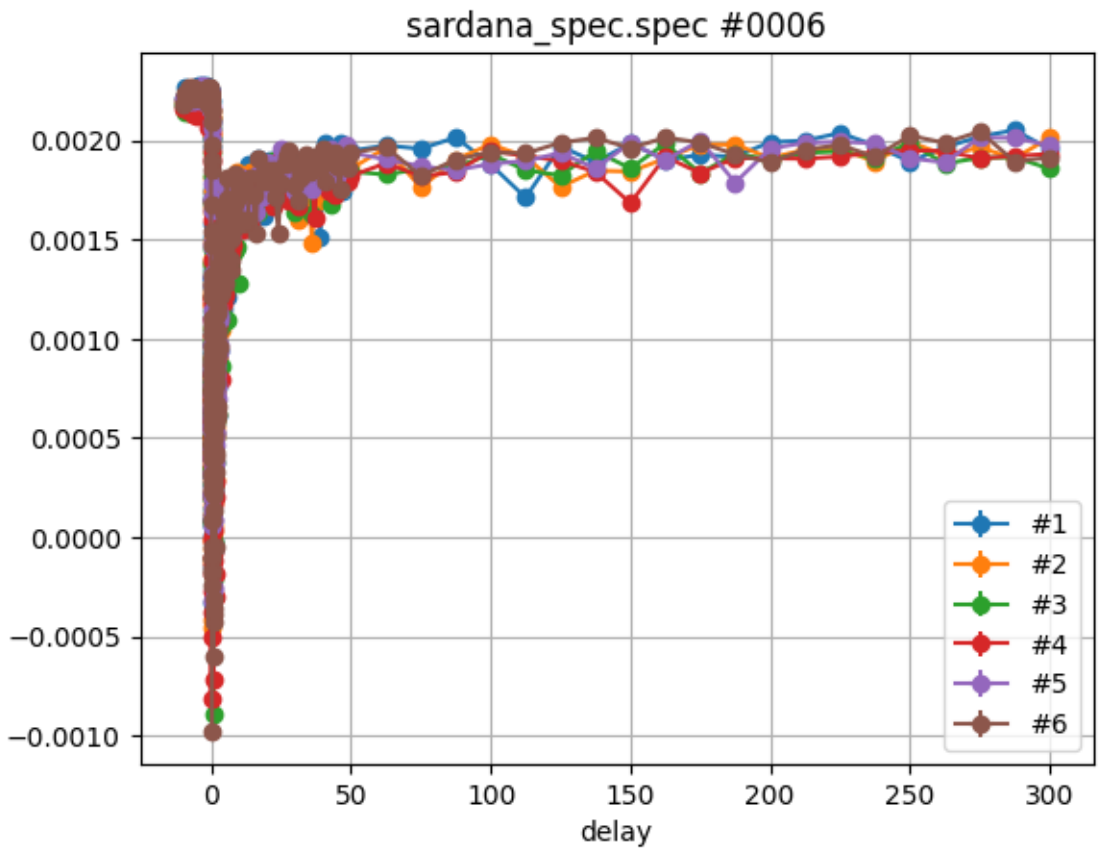
```
10:16:11
```

Now we create the `scan_sequence` as a `list`, which contains one or multiple entries. Each entry can be a `list` or `tuple` which contains two elements: The scan list containing one or multiple scan numbers, and a string or number describing the external parameter.

```
scan_sequence = [
    # ([scan numbers], parameter)
    ([1], spec.scan1.time),  # first entry
    ([2], spec.scan2.time),
    ([3], spec.scan3.time),
    ([4], spec.scan4.time),
    ([5], spec.scan5.time),
    ([6], spec.scan6.time),  # last entry
]
```

The minimum example below does not differ too much from plotting all six scans manually by the `plot_scans()` method:

```
plt.figure()
ev.plot_scan_sequence(scan_sequence)
plt.show()
```



Obviously, the legend did not take the scan time into account. Let's check the documentation for some details:

```
help(ev.plot_scan_sequence)
```

```
Help on method plot_scan_sequence in module pyEvalData.evaluation:

plot_scan_sequence(scan_sequence, ylims=[], xlims=[], fig_size=[], xgrid=[], yerr='std',␣
↪xerr='std', norm2one=False, binning=True, sequence_type='', label_text='', title_text='
↪', skip_plot=False, grid_on=True, ytext='', xtext='', fmt='-o') method of pyEvalData.
↪evaluation.Evaluation instance
    Plot a list of scans from the spec file.
    Various plot parameters are provided.
    The plotted data are returned.

    Args:
        scan_sequence (List[
            List/Tuple[List[int],
            int/str]])              : Sequence of scan lists and parameters.
        ylims (Optional[ndarray])   : ylim for the plot.
        xlims (Optional[ndarray])   : xlim for the plot.
        fig_size (Optional[ndarray]) : Figure size of the figure.
        xgrid (Optional[ndarray])   : Grid to bin the data to -
                                      default in empty so use the
                                      x-axis of the first scan.
        yerr (Optional[ndarray])    : Type of the errors in y: [err, std, none]
```

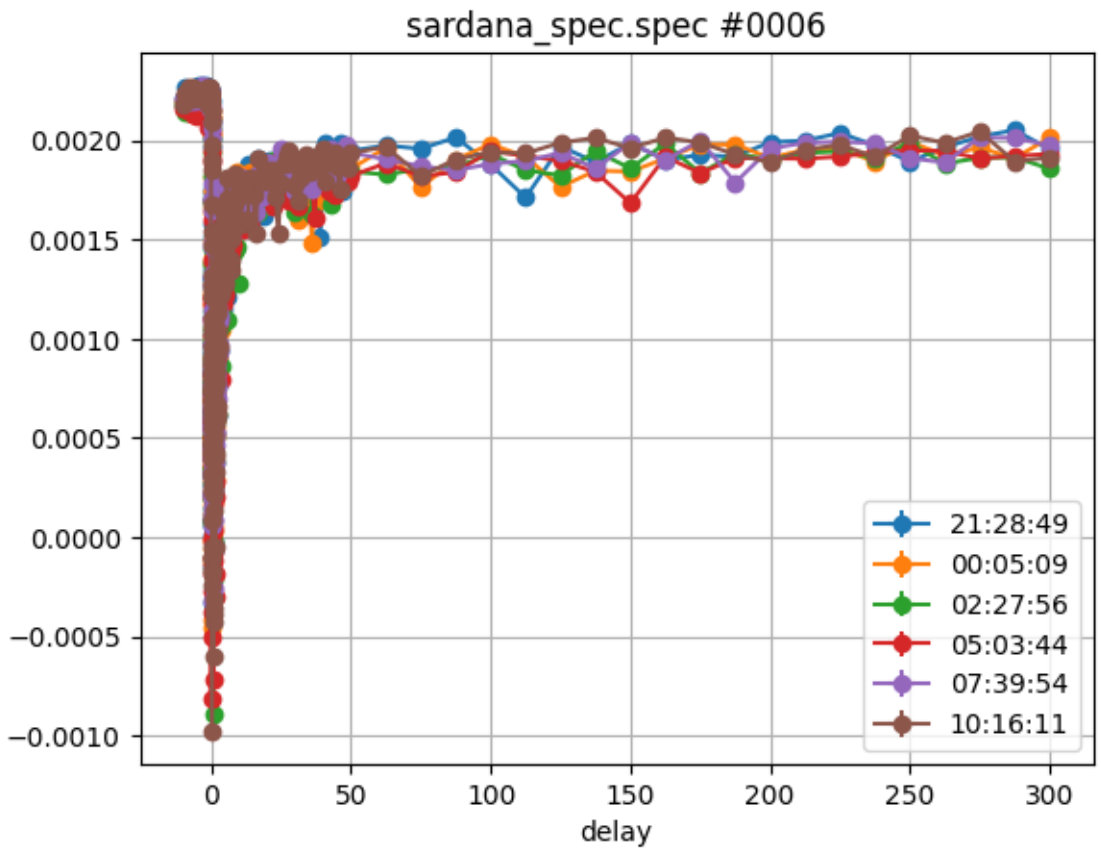
(continues on next page)

(continued from previous page)

```
                                    default is 'std'.
        xerr (Optional[ndarray])    : Type of the errors in x: [err, std, none]
                                    default is 'std'.
        norm2one (Optional[bool])   : Norm transient data to 1 for t < t0
                                    default is False.
        sequence_type (Optional[str]): Type of the sequence: [fluence, delay,
                                    energy, theta, position, voltage, none,
                                    text] - default is enumeration.
        label_text (Optional[str])   : Label of the plot - default is none.
        title_text (Optional[str])   : Title of the figure - default is none.
        skip_plot (Optional[bool])   : Skip plotting, just return data
                                    default is False.
        grid_on (Optional[bool])     : Add grid to plot - default is True.
        ytext (Optional[str])       : y-Label of the plot - defaults is none.
        xtext (Optional[str])       : x-Label of the plot - defaults is none.
        fmt (Optional[str])         : format string of the plot - defaults is -o.

    Returns:
        sequence_data (OrderedDict) : Dictionary of the averaged scan data.
        parameters (List[str, float]) : Parameters of the sequence.
        names (List[str])          : List of names of each data set.
        label_texts (List[str])     : List of labels for each data set.
```

In the docstring we can find many arguments already known from the `plot_scans()` method. In order to fix the legend labels we need to tell the method about the `sequence_type` as an argument. Otherwise it will enumerate the scans by default. In our case we provided a `text` type label:

```
plt.figure()
sequence_data, parameters, names, label_texts = \
    ev.plot_scan_sequence(scan_sequence, sequence_type='text')
plt.show()
```



In the example above we also catched the return values. Here the `parameters` correspond exactly to the data we provided in the `scan_sequence` while the `label_texts` are the formatted string as written in the legend. The `names` correspond to the auto-generated name of each scan as given by the `plot_scans()` method.

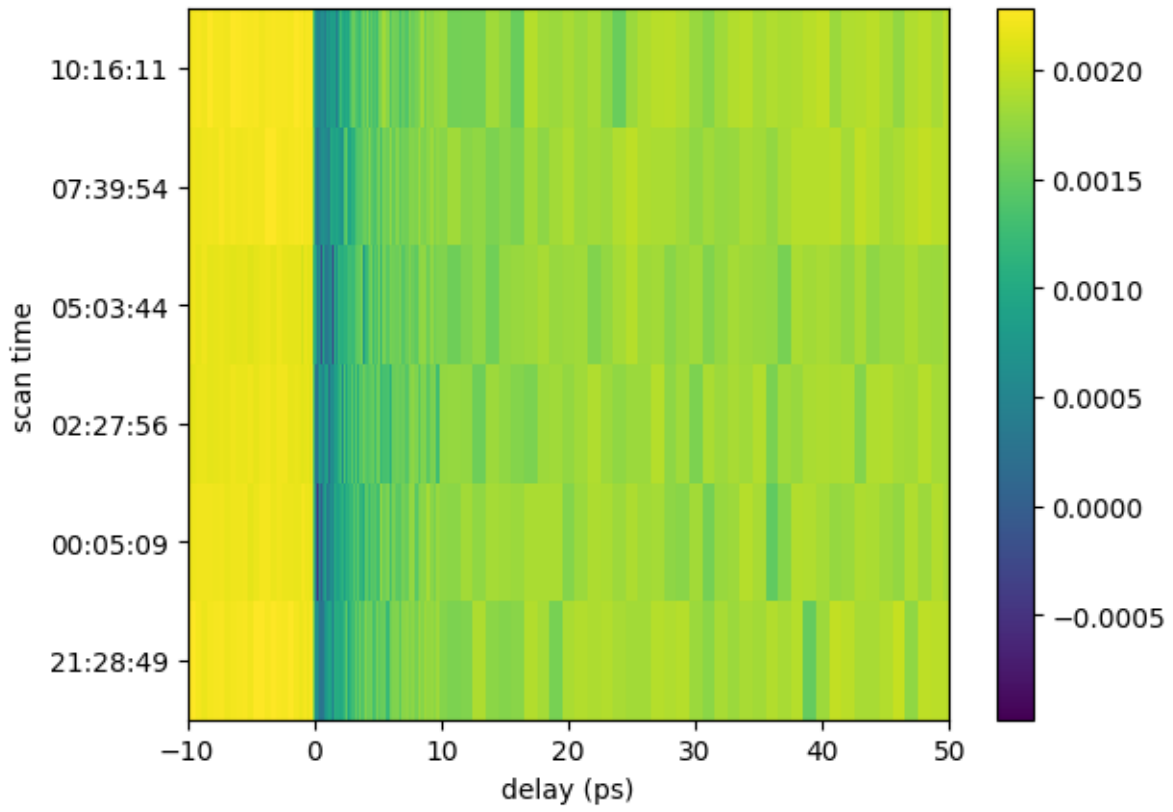
The actual `sequence_data` is again an `OrderedDict` where the `keys` are given by the strings in the `xcol` and `clist` attributes of the `Evaluation` object. Each value for a given `key` is a `list` of `ndarrays` that hold the data for every parameter.

```
print(sequence_data.keys())
```

```
odict_keys(['delay', 'delayErr', 'abs_mag', 'abs_magErr'])
```

Let's make an 2D plot from the `scan_sequence`:

```
x = sequence_data['delay'][0]
y = np.arange(6)
z = sequence_data['abs_mag']

plt.figure()
plt.pcolormesh(x, y, z, shading='auto')
plt.xlim(-10, 50)
plt.xlabel('delay (ps)')
plt.yticks(y, label_texts, rotation='horizontal')
plt.ylabel('scan time')
plt.colorbar()
plt.show()
```



Fit scan sequences

Finally, we want to fit the `scan_sequence` and extract the according fit parameters for every trace. Here we use the `fit_scan_sequence()` method. So let's check the documentation first:

```
help(ev.fit_scan_sequence)
```

```
Help on method fit_scan_sequence in module pyEvalData.evaluation:

fit_scan_sequence(scan_sequence, mod, pars, ylims=[], xlims=[], fig_size=[], xgrid=[],␣
↪yerr='std', xerr='std', norm2one=False, binning=True, sequence_type='', label_text='',␣
↪title_text='', ytext='', xtext='', select='', fit_report=0, show_single=False,␣
↪weights=False, fit_method='leastsq', offset_t0=False, plot_separate=False, grid_
↪on=True, last_res_as_par=False, sequence_data=[], fmt='o') method of pyEvalData.
↪evaluation.Evaluation instance
    Fit, plot, and return the data of a scan sequence.

    Args:
        scan_sequence (List[
            List/Tuple[List[int],
            int/str]])              : Sequence of scan lists and parameters.
        mod (Model[lmfit])          : lmfit model for fitting the data.
        pars (Parameters[lmfit])    : lmfit parameters for fitting the data.
        ylims (Optional[ndarray])   : ylim for the plot.
        xlims (Optional[ndarray])   : xlim for the plot.
```

(continues on next page)

(continued from previous page)

```
        fig_size (Optional[ndarray]) : Figure size of the figure.
        xgrid (Optional[ndarray])   : Grid to bin the data to -
                                      default in empty so use the
                                      x-axis of the first scan.
        yerr (Optional[ndarray])    : Type of the errors in y: [err, std, none]
                                      default is 'std'.
        xerr (Optional[ndarray])    : Type of the errors in x: [err, std, none]
                                      default is 'std'.
        norm2one (Optional[bool])   : Norm transient data to 1 for t < t0
                                      default is False.
        sequence_type (Optional[str]): Type of the sequence: [fluence, delay,
                                      energy, theta] - default is fluence.
        label_text (Optional[str])   : Label of the plot - default is none.
        title_text (Optional[str])   : Title of the figure - default is none.
        ytext (Optional[str])       : y-Label of the plot - defaults is none.
        xtext (Optional[str])       : x-Label of the plot - defaults is none.
        select (Optional[str])      : String to evaluate as select statement
                                      for the fit region - default is none
        fit_report (Optional[int])   : Set the fit reporting level:
                                      [0: none, 1: basic, 2: full]
                                      default 0.
        show_single (Optional[bool]) : Plot each fit seperately - default False.
        weights (Optional[bool])    : Use weights for fitting - default False.
        fit_method (Optional[str])   : Method to use for fitting; refer to
                                      lmfit - default is 'leastsq'.
        offset_t0 (Optional[bool])   : Offset time scans by the fitted
                                      t0 parameter - default False.
        plot_separate (Optional[bool]):A single plot for each counter
                                      default False.
        grid_on (Optional[bool])     : Add grid to plot - default is True.
        last_res_as_par (Optional[bool]): Use the last fit result as start
                                       values for next fit - default is False.
        sequence_data (Optional[ndarray]): actual exp. data are externally given.
                                          default is empty
        fmt (Optional[str])         : format string of the plot - defaults is -o.


    Returns:
        res (Dict[ndarray])        : Fit results.
        parameters (ndarray)       : Parameters of the sequence.
        sequence_data (OrderedDict) : Dictionary of the averaged scan data.equenceData
```

Again we find a lot of previously defined arguments which we are already familiar with.
For the fitting, we need to provide first of all a proper fitting model `mod` and the accroding fit parameters `pars` with *initial* and *boundray* conditions. Here we rely on the `lmfit` package. So please dive into its great documentation before continuing here.

In order to describe our data best, we would like to use a double-exponential function for the initial decrease and subsequent increase of the magnetization. Moreover, we have to take into account the *step-like* behaviour before and after the exciation at *delay=0* ps as well as the temporal resoultion of our setup as mimiced by a convolution with a gaussian function.

Such rather complex fitting function is provided by the ultrafastFitFunctions package which as been already imported in the Setup.

```
help(ufff.doubleDecayConvScale)
```

```
Help on function doubleDecayConvScale in module ultrafastFitFunctions.dynamics:

doubleDecayConvScale(x, mu, tau1, tau2, A, q, alpha, sigS, sigH, I0)
```

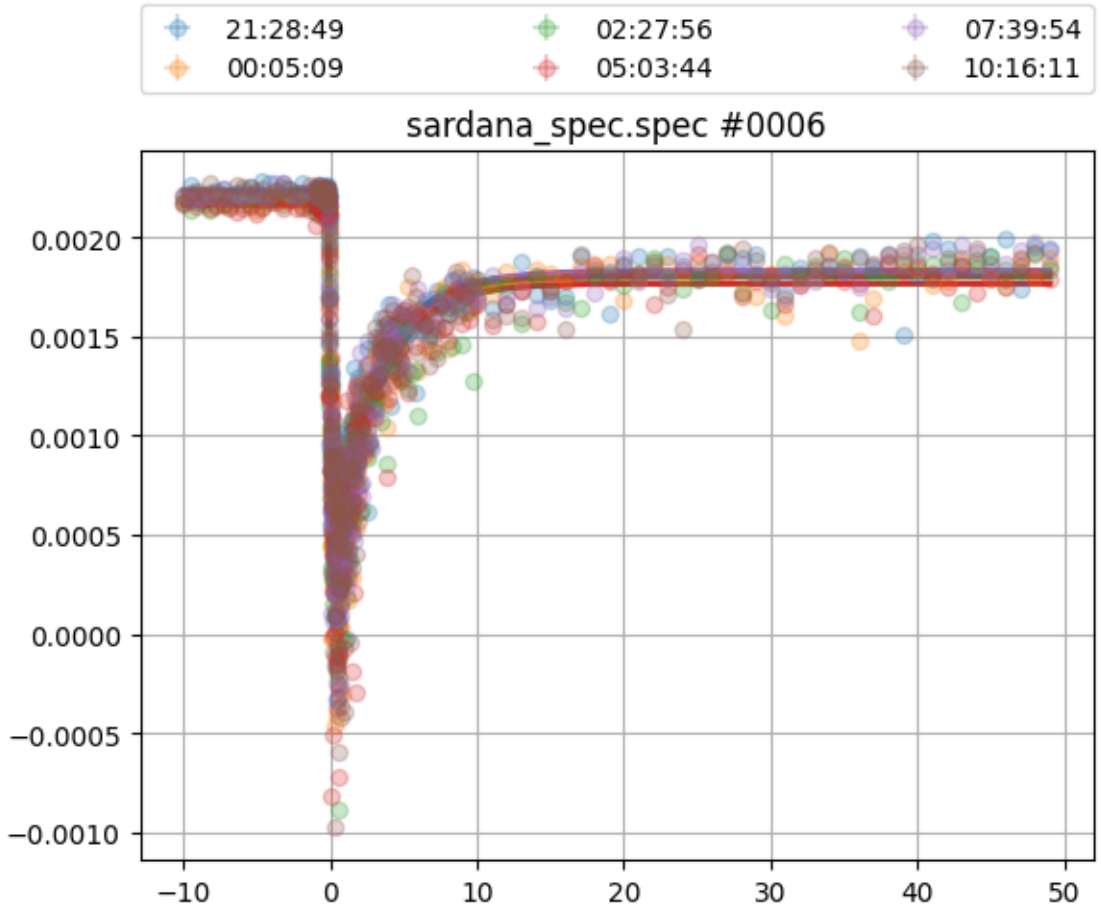
The documentation for the fitting fucntions are hopefully comming soon :)

Now lets create the model and parameters:

```
mod = lf.Model(ufff.doubleDecayConvScale)
pars = lf.Parameters()

pars.add('mu', value=0)
pars.add('tau1', value=0.2)
pars.add('tau2', value=10)
pars.add('A', value=0.5)
pars.add('q', value=1)
pars.add('alpha', value=1, vary=False)
pars.add('sigS', value=0.05, vary=False)
pars.add('sigH', value=0, vary=False)
pars.add('I0', value=0.002)
```

We are not going too much in to detail of the fitting function, so let's do the actual fit. For that, we limit the data again on a reduced grid given by the `xgrid` argument and provide the correct `sequence_type` for the label generation.

```
plt.figure()
ev.fit_scan_sequence(scan_sequence, mod, pars, xgrid=np.r_[-10:50:0.01],
                     sequence_type='text')
plt.show()
```

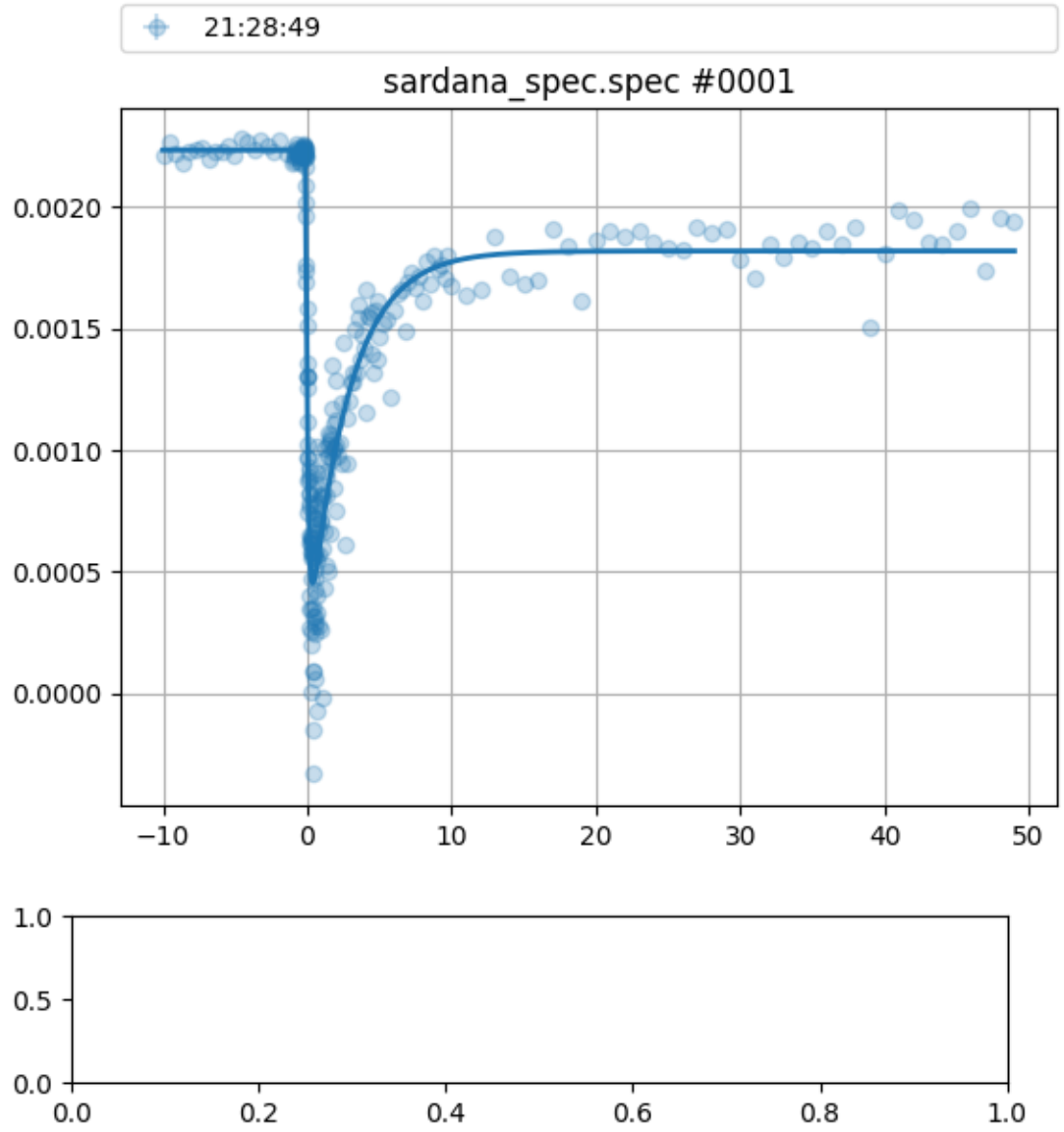


The results does already look very good, but lets access some more information:

```
plt.figure()
res, parameters, sequence_data = ev.fit_scan_sequence(scan_sequence, mod, pars,
                                                      xgrid=np.r_[-10:50:0.01],
                                                      sequence_type='text',
                                                      show_single=True,
                                                      fit_report=1)
plt.show()
```

```
========== Parameter: 21:28:49 ===============

---------- abs_mag: --------------
          mu:  -1.1282e-01
        tau1:  1.4375e-01
        tau2:  2.7683e+00
           A:  1.8572e-01
           q:  5.1436e+00
       alpha:  1.0000e+00
        sigS:  5.0000e-02
        sigH:  0.0000e+00
          I0:  2.2348e-03
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[31], line 2
      1 plt.figure()
----> 2 res, parameters, sequence_data = ev.fit_scan_sequence(scan_sequence, mod, pars,
      3                                                      xgrid=np.r_[-10:50:0.01],
      4                                                      sequence_type='text',
      5                                                      show_single=True,
      6                                                      fit_report=1)
      7 plt.show()

File ~/checkouts/readthedocs.org/user_builds/pyevaldata/envs/stable/lib/python3.9/site-
↪packages/pyEvalData/evaluation.py:1219, in Evaluation.fit_scan_sequence(self, scan_
↪sequence, mod, pars, ylims, xlims, fig_size, xgrid, yerr, xerr, norm2one, binning,␣
↪sequence_type, label_text, title_text, ytext, xtext, select, fit_report, show_single,␣
↪weights, fit_method, offset_t0, plot_separate, grid_on, last_res_as_par, sequence_data,
↪ fmt)
   1216 gs = mpl.gridspec.GridSpec(
   1217     2, 1, height_ratios=[1, 3], hspace=0.1)
   1218 ax1 = plt.subplot(gs[0])
-> 1219 markerline, stemlines, baseline = plt.stem(
   1220     x2plot-offsetX, out.residual, markerfmt=' ',
   1221     use_line_collection=True)
   1222 plt.setp(stemlines, 'color',
   1223          plot[0].get_color(), 'linewidth', 2, alpha=0.5)
   1224 plt.setp(baseline, 'color', 'k', 'linewidth', 0)

TypeError: stem() got an unexpected keyword argument 'use_line_collection'
```



Access fit results

The results of the fits are given in the `res` dictionary. Here the keys correspond again to the elements in the `clist`:

```
print(res.keys())
```

For every counter in the `clist` we have a nested dictionary with all best values and errrors for the individual fit parameters. Moreover, we can access some general parameters as the *center of mass* (`CoM`) or *integral* (`int`), as well as the `fit` objects themselve.

```
print(res['abs_mag'].keys())
```

So let's plot the decay amplitude `A` for the different parameters in the `scan_sequence`:

```
plt.figure()
plt.errorbar(parameters, res['abs_mag']['A'], yerr=res['abs_mag']['AErr'], fmt='-o')
plt.xlabel('scan time')
plt.ylabel('decay amplitude')
plt.show()
```

Filters

```
# to be done
```

4.3 API Documentation

4.3.1 io

Classes:

Scan(number, **kwargs)	Representation of a scan which holds the relevant data and meta information.
Source(file_name[, file_path])	Class of default source implementation.
Spec(file_name, file_path, **kwargs)	Source implementation for SPEC files.
SardanaNeXus(file_name, file_path, **kwargs)	Source implementation for Sardana NeXus files.
PalH5(name, file_name, file_path, **kwargs)	Source implementation for PalH5 folder/files.

class pyEvalData.io.**Scan**(*number*, ***kwargs*)

Bases: object

Representation of a scan which holds the relevant data and meta information.

Parameters **number** (*uint*) – number of the scan.

Keyword Arguments

- **cmd** (*str*) – scan command.
- **user** (*str*) – scan user.
- **date** (*str*) – scan date.
- **time** (*str*) – scan time.
- **int_time** (*float*) – integration time.
- **init_mopo** (*dict(float)*) – initial motor position.
- **header** (*str*) – full scan header.

Attributes

- **log** (*logging.logger*) – logger instance from logging.
- **number** (*uint*) – number of the scan.
- **meta** (*dict*) – meta data dictionary.
- **data** (*ndarray[float]*) – data recarray.

Methods:

index_data()	Check the dimensions of the data recarray elements and remember the names for scaler, 1d, and 2d data columns.
get_scalar_data()	Returns only scalar data from the data recarray.
get_oned_data()	Returns only 1d data from the data recarray.
get_twod_data()	Returns only 2d data from the data recarray.
clear_data()	Clears the data to save memory.

index_data()
Check the dimensions of the data recarray elements and remember the names for scaler, 1d, and 2d data columns.

get_scalar_data()
Returns only scalar data from the data recarray.

Returns *data (ndarray[float])* – scalar data.

get_oned_data()
Returns only 1d data from the data recarray.

Returns *data (ndarray[float])* – 1d data.

get_twod_data()
Returns only 2d data from the data recarray.

Returns *data (ndarray[float])* – 2d data.

clear_data()
Clears the data to save memory.

class pyEvalData.io.**Source**(*file_name*, *file_path='./'*, ***kwargs*)
Bases: object

Class of default source implementation.

Parameters

- **file_name** (*str*) – file name including extension, can include regex pattern.
- **file_path** (*str, optional*) – file path - defaults to *./*.

Keyword Arguments

- **start_scan_number** (*uint*) – start of scan numbers to parse.
- **stop_scan_number** (*uint*) – stop of scan numbers to parse. This number is included.
- **nexus_file_name** (*str*) – name for generated nexus file.
- **nexus_file_name_postfix** (*str*) – postfix for nexus file name.
- **nexus_file_path** (*str*) – path for generated nexus file.
- **read_all_data** (*bool*) – read all data on parsing. If false, data will be read only on demand.
- **read_and_forget** (*bool*) – clear data after read to save memory.
- **update_before_read** (*bool*) – always update from source before reading scan data.
- **use_nexus** (*bool*) – use nexus file to join/compress raw data.
- **force_overwrite** (*bool*) – forced re-read of raw source and re-generated of nexus file.

Attributes

- **log** (*logging.logger*) – logger instance from logging.
- **name** (*str*) – name of the source
- **scan_dict** (*dict(scan)*) – dict of scan objects with key being the scan number.
- **start_scan_number** (*uint*) – start of scan numbers to parse.
- **stop_scan_number** (*uint*) – stop of scan numbers to parse. This number is included.
- **file_name** (*str*) – file name including extension, can include regex pattern.
- **file_path** (*str, optional*) – file path - defaults to `./`.
- **nexus_file_name** (*str*) – name for generated nexus file.
- **nexus_file_name_postfix** (*str*) – postfix for nexus file name.
- **nexus_file_path** (*str*) – path for generated nexus file.
- **nexus_file_exists** (*bool*) – if nexus file exists.
- **read_all_data** (*bool*) – read all data on parsing.
- **read_and_forget** (*bool*) – clear data after read to save memory.
- **update_before_read** (*bool*) – always update from source before reading scan data.
- **use_nexus** (*bool*) – use nexus file to join/compress raw data.
- **force_overwrite** (*bool*) – forced re-read of raw source and re-generated of nexus file.

Methods:

update([scan_number_list])	update the `scan_dict` either from the raw source file/folder or from the nexus file.
parse_raw()	Parse the raw source file/folder and populate the *scan_dict*.
parse_nexus()	Parse the nexus file and populate the *scan_dict*.
check_nexus_file_exists()	Check if the nexus file is present and set *self.nexus_file_exists*.
get_last_scan_number()	Return the number of the last scan in the *scan_dict*.
get_all_scan_numbers()	Return the all scan number from the *scan_dict*.
get_scan(scan_number[, read_data, ...])	Returns a scan object from the scan dict determined by the scan_number.
get_scan_list(scan_number_list[, read_data])	Returns a list of scan object from the *scan_dict* determined by the list of scan_number.
get_scan_data(scan_number)	Returns data and meta information from a scan object from the *scan_dict* determined by the scan_number.
get_scan_list_data(scan_number_list)	Returns data and meta information for a list of scan objects from the *scan_dict* determined by the scan_numbers.
read_scan_data(scan)	Reads the data for a given scan object.
read_raw_scan_data(scan)	Reads the data for a given scan object from raw source.
read_nexus_scan_data(scan)	Reads the data for a given scan object from the nexus file.
clear_scan_data(scan)	Clear the data for a given scan object.

continues on next page

Table 3 – continued from previous page

read_all_scan_data()	Reads the data for all scan objects in the *scan_dict* from source.
clear_all_scan_data()	Clears the data for all scan objects in the *scan_dict*.
save_scan_to_nexus(scan[, nxs_file])	Saves a scan to the nexus file.
save_all_scans_to_nexus()	Saves all scan objects in the *scan_dict* to the nexus file.
get_nexus_file([mode])	Return the file handle to the NeXus file in a given `mode`.

update(*scan_number_list=[]*)
update the `scan_dict` either from the raw source file/folder or from the nexus file. The optional `scan_number_list` runs the update only if required for the included scan.

Attributes **scan_number_list** (*list[int]*) – explicit list of scans

parse_raw()
Parse the raw source file/folder and populate the *scan_dict*.

parse_nexus()
Parse the nexus file and populate the *scan_dict*.

check_nexus_file_exists()
Check if the nexus file is present and set *self.nexus_file_exists*.

get_last_scan_number()
Return the number of the last scan in the *scan_dict*. If the *scan_dict* is empty return 0.

get_all_scan_numbers()
Return the all scan number from the *scan_dict*.

get_scan(*scan_number*, *read_data=True*, *dismiss_update=False*)
Returns a scan object from the scan dict determined by the scan_number.

Parameters

- **scan_number** (*uint*) – number of the scan.
- **read_data** (*bool, optional*) – read data from source. Defaults to *False*.
- **dismiss_update** (*bool, optional*) – Dismiss update even if set as object attribute. Defaults to *False*.

Returns *scan (Scan)* – scan object.

get_scan_list(*scan_number_list*, *read_data=True*)
Returns a list of scan object from the *scan_dict* determined by the list of scan_number.

Parameters

- **scan_number_list** (*list(uint)*) – list of numbers of the scan.
- **read_data** (*bool, optional*) – read data from source. Defaults to *False*.

Returns *scans (list(Scan))* – list of scan object.

get_scan_data(*scan_number*)
Returns data and meta information from a scan object from the *scan_dict* determined by the scan_number.

Parameters **scan_number** (*uint*) – number of the scan.

Returns *data (numpy.recarray[float])* – scan data. meta (dict()): scan meta information.

get_scan_list_data(*scan_number_list*)
Returns data and meta information for a list of scan objects from the *scan_dict* determined by the scan_numbers.

Parameters **scan_number_list** (*list(uint)*) – list of numbers of the scan.

Returns *data (list(numpy.recarray[float]))* – list of scan data. meta (list(dict())): list scan meta information.

read_scan_data(*scan*)
Reads the data for a given scan object.

Parameters **scan** (Scan) – scan object.

read_raw_scan_data(*scan*)
Reads the data for a given scan object from raw source.

Parameters **scan** (Scan) – scan object.

read_nexus_scan_data(*scan*)
Reads the data for a given scan object from the nexus file.

Parameters **scan** (Scan) – scan object.

clear_scan_data(*scan*)
Clear the data for a given scan object.

Parameters **scan** (Scan) – scan object.

read_all_scan_data()
Reads the data for all scan objects in the *scan_dict* from source.

clear_all_scan_data()
Clears the data for all scan objects in the *scan_dict*.

save_scan_to_nexus(*scan*, *nxs_file=''*)
Saves a scan to the nexus file.

save_all_scans_to_nexus()
Saves all scan objects in the *scan_dict* to the nexus file.

get_nexus_file(*mode='rw'*)
Return the file handle to the NeXus file in a given `mode`.

Parameters **mode** (*str, optional*) – file mode. defaults to 'rw'.

Returns *nxs_file (NXFile)* – file handle to NeXus file.

class pyEvalData.io.**Spec**(*file_name*, *file_path*, ***kwargs*)
Bases: *pyEvalData.io.source.Source*

Source implementation for SPEC files.

Parameters

- **file_name** (*str*) – file name including extension, can include regex pattern.
- **file_path** (*str, optional*) – file path - defaults to `./`.

Keyword Arguments

- **start_scan_number** (*uint*) – start of scan numbers to parse.
- **stop_scan_number** (*uint*) – stop of scan numbers to parse. This number is included.
- **nexus_file_name** (*str*) – name for generated nexus file.

- **nexus_file_name_postfix** (*str*) – postfix for nexus file name.
- **nexus_file_path** (*str*) – path for generated nexus file.
- **read_all_data** (*bool*) – read all data on parsing. If false, data will be read only on demand.
- **read_and_forget** (*bool*) – clear data after read to save memory.
- **update_before_read** (*bool*) – always update from source before reading scan data.
- **use_nexus** (*bool*) – use nexus file to join/compress raw data.
- **force_overwrite** (*bool*) – forced re-read of raw source and re-generated of nexus file.

Attributes

- **log** (*logging.logger*) – logger instance from logging.
- **name** (*str*) – name of the source
- **scan_dict** (*dict(scan)*) – dict of scan objects with key being the scan number.
- **start_scan_number** (*uint*) – start of scan numbers to parse.
- **stop_scan_number** (*uint*) – stop of scan numbers to parse. This number is included.
- **file_name** (*str*) – file name including extension, can include regex pattern.
- **file_path** (*str, optional*) – file path - defaults to `./`.
- **nexus_file_name** (*str*) – name for generated nexus file.
- **nexus_file_name_postfix** (*str*) – postfix for nexus file name.
- **nexus_file_path** (*str*) – path for generated nexus file.
- **nexus_file_exists** (*bool*) – if nexus file exists.
- **read_all_data** (*bool*) – read all data on parsing.
- **read_and_forget** (*bool*) – clear data after read to save memory.
- **update_before_read** (*bool*) – always update from source before reading scan data.
- **use_nexus** (*bool*) – use nexus file to join/compress raw data.
- **force_overwrite** (*bool*) – forced re-read of raw source and re-generated of nexus file.

Methods:

parse_raw()	Parse the raw source file/folder and populate the *scan_dict*.
read_raw_scan_data(scan)	Reads the data for a given scan object from raw source.
check_nexus_file_exists()	Check if the nexus file is present and set *self.nexus_file_exists*.
clear_all_scan_data()	Clears the data for all scan objects in the *scan_dict*.
clear_scan_data(scan)	Clear the data for a given scan object.
get_all_scan_numbers()	Return the all scan number from the *scan_dict*.
get_last_scan_number()	Return the number of the last scan in the *scan_dict*.
get_nexus_file([mode])	Return the file handle to the NeXus file in a given `mode`.
get_scan(scan_number[, read_data, ...])	Returns a scan object from the scan dict determined by the scan_number.

continues on next page

Table 4 – continued from previous page

get_scan_data(scan_number)	Returns data and meta information from a scan object from the *scan_dict* determined by the scan_number.
get_scan_list(scan_number_list[, read_data])	Returns a list of scan object from the *scan_dict* determined by the list of scan_number.
get_scan_list_data(scan_number_list)	Returns data and meta information for a list of scan objects from the *scan_dict* determined by the scan_numbers.
parse_nexus()	Parse the nexus file and populate the *scan_dict*.
read_all_scan_data()	Reads the data for all scan objects in the *scan_dict* from source.
read_nexus_scan_data(scan)	Reads the data for a given scan object from the nexus file.
read_scan_data(scan)	Reads the data for a given scan object.
save_all_scans_to_nexus()	Saves all scan objects in the *scan_dict* to the nexus file.
save_scan_to_nexus(scan[, nxs_file])	Saves a scan to the nexus file.
update([scan_number_list])	update the `scan_dict` either from the raw source file/folder or from the nexus file.

parse_raw()
Parse the raw source file/folder and populate the *scan_dict*.

read_raw_scan_data(*scan*)
Reads the data for a given scan object from raw source.

Parameters **scan** (Scan) – scan object.

check_nexus_file_exists()
Check if the nexus file is present and set *self.nexus_file_exists*.

clear_all_scan_data()
Clears the data for all scan objects in the *scan_dict*.

clear_scan_data(*scan*)
Clear the data for a given scan object.

Parameters **scan** (Scan) – scan object.

get_all_scan_numbers()
Return the all scan number from the *scan_dict*.

get_last_scan_number()
Return the number of the last scan in the *scan_dict*. If the *scan_dict* is empty return 0.

get_nexus_file(*mode='rw'*)
Return the file handle to the NeXus file in a given `mode`.

Parameters **mode** (*str, optional*) – file mode. defaults to 'rw'.

Returns *nxs_file (NXFile)* – file handle to NeXus file.

get_scan(*scan_number*, *read_data=True*, *dismiss_update=False*)
Returns a scan object from the scan dict determined by the scan_number.

Parameters

- **scan_number** (*uint*) – number of the scan.
- **read_data** (*bool, optional*) – read data from source. Defaults to *False*.

- **dismiss_update** (*bool, optional*) – Dismiss update even if set as object attribute. Defaults to *False*.

Returns *scan (Scan)* – scan object.

get_scan_data(*scan_number*)
Returns data and meta information from a scan object from the *scan_dict* determined by the scan_number.

Parameters **scan_number** (*uint*) – number of the scan.

Returns *data (numpy.recarray[float])* – scan data. meta (dict()): scan meta information.

get_scan_list(*scan_number_list*, *read_data=True*)
Returns a list of scan object from the *scan_dict* determined by the list of scan_number.

Parameters

- **scan_number_list** (*list(uint)*) – list of numbers of the scan.
- **read_data** (*bool, optional*) – read data from source. Defaults to *False*.

Returns *scans (list(Scan))* – list of scan object.

get_scan_list_data(*scan_number_list*)
Returns data and meta information for a list of scan objects from the *scan_dict* determined by the scan_numbers.

Parameters **scan_number_list** (*list(uint)*) – list of numbers of the scan.

Returns *data (list(numpy.recarray[float]))* – list of scan data. meta (list(dict())): list scan meta information.

parse_nexus()
Parse the nexus file and populate the *scan_dict*.

read_all_scan_data()
Reads the data for all scan objects in the *scan_dict* from source.

read_nexus_scan_data(*scan*)
Reads the data for a given scan object from the nexus file.

Parameters **scan** (Scan) – scan object.

read_scan_data(*scan*)
Reads the data for a given scan object.

Parameters **scan** (Scan) – scan object.

save_all_scans_to_nexus()
Saves all scan objects in the *scan_dict* to the nexus file.

save_scan_to_nexus(*scan*, *nxs_file=''*)
Saves a scan to the nexus file.

update(*scan_number_list=[]*)
update the `scan_dict` either from the raw source file/folder or from the nexus file. The optional `scan_number_list` runs the update only if required for the included scan.

Attributes **scan_number_list** (*list[int]*) – explicit list of scans

class pyEvalData.io.**SardanaNeXus**(*file_name*, *file_path*, ***kwargs*)
Bases: *pyEvalData.io.source.Source*

Source implementation for Sardana NeXus files.

Parameters

- **file_name** (*str*) – file name including extension, can include regex pattern.
- **file_path** (*str, optional*) – file path - defaults to `./`.

Keyword Arguments

- **start_scan_number** (*uint*) – start of scan numbers to parse.
- **stop_scan_number** (*uint*) – stop of scan numbers to parse. This number is included.
- **nexus_file_name** (*str*) – name for generated nexus file.
- **nexus_file_name_postfix** (*str*) – postfix for nexus file name.
- **nexus_file_path** (*str*) – path for generated nexus file.
- **read_all_data** (*bool*) – read all data on parsing. If false, data will be read only on demand.
- **read_and_forget** (*bool*) – clear data after read to save memory.
- **update_before_read** (*bool*) – always update from source before reading scan data.
- **use_nexus** (*bool*) – use nexus file to join/compress raw data.
- **force_overwrite** (*bool*) – forced re-read of raw source and re-generated of nexus file.

Attributes

- **log** (*logging.logger*) – logger instance from logging.
- **name** (*str*) – name of the source
- **scan_dict** (*dict(scan)*) – dict of scan objects with key being the scan number.
- **start_scan_number** (*uint*) – start of scan numbers to parse.
- **stop_scan_number** (*uint*) – stop of scan numbers to parse. This number is included.
- **file_name** (*str*) – file name including extension, can include regex pattern.
- **file_path** (*str, optional*) – file path - defaults to `./`.
- **nexus_file_name** (*str*) – name for generated nexus file.
- **nexus_file_name_postfix** (*str*) – postfix for nexus file name.
- **nexus_file_path** (*str*) – path for generated nexus file.
- **nexus_file_exists** (*bool*) – if nexus file exists.
- **read_all_data** (*bool*) – read all data on parsing.
- **read_and_forget** (*bool*) – clear data after read to save memory.
- **update_before_read** (*bool*) – always update from source before reading scan data.
- **use_nexus** (*bool*) – use nexus file to join/compress raw data.
- **force_overwrite** (*bool*) – forced re-read of raw source and re-generated of nexus file.

Methods:

parse_raw()	Parse the Sardana NeXus file and populate the *scan_dict*.
read_raw_scan_data(scan)	Reads the data for a given scan object from Sardana NeXus file.
check_nexus_file_exists()	Check if the nexus file is present and set *self.nexus_file_exists*.

continues on next page

Table 5 – continued from previous page

clear_all_scan_data()	Clears the data for all scan objects in the *scan_dict*.
clear_scan_data(scan)	Clear the data for a given scan object.
get_all_scan_numbers()	Return the all scan number from the *scan_dict*.
get_last_scan_number()	Return the number of the last scan in the *scan_dict*.
get_nexus_file([mode])	Return the file handle to the NeXus file in a given `mode`.
get_scan(scan_number[, read_data, ...])	Returns a scan object from the scan dict determined by the scan_number.
get_scan_data(scan_number)	Returns data and meta information from a scan object from the *scan_dict* determined by the scan_number.
get_scan_list(scan_number_list[, read_data])	Returns a list of scan object from the *scan_dict* determined by the list of scan_number.
get_scan_list_data(scan_number_list)	Returns data and meta information for a list of scan objects from the *scan_dict* determined by the scan_numbers.
parse_nexus()	Parse the nexus file and populate the *scan_dict*.
read_all_scan_data()	Reads the data for all scan objects in the *scan_dict* from source.
read_nexus_scan_data(scan)	Reads the data for a given scan object from the nexus file.
read_scan_data(scan)	Reads the data for a given scan object.
save_all_scans_to_nexus()	Saves all scan objects in the *scan_dict* to the nexus file.
save_scan_to_nexus(scan[, nxs_file])	Saves a scan to the nexus file.
update([scan_number_list])	update the `scan_dict` either from the raw source file/folder or from the nexus file.

parse_raw()
Parse the Sardana NeXus file and populate the *scan_dict*.

read_raw_scan_data(*scan*)
Reads the data for a given scan object from Sardana NeXus file.

Parameters **scan** (Scan) – scan object.

check_nexus_file_exists()
Check if the nexus file is present and set *self.nexus_file_exists*.

clear_all_scan_data()
Clears the data for all scan objects in the *scan_dict*.

clear_scan_data(*scan*)
Clear the data for a given scan object.

Parameters **scan** (Scan) – scan object.

get_all_scan_numbers()
Return the all scan number from the *scan_dict*.

get_last_scan_number()
Return the number of the last scan in the *scan_dict*. If the *scan_dict* is empty return 0.

get_nexus_file(*mode='rw'*)
Return the file handle to the NeXus file in a given `mode`.

Parameters **mode** (*str, optional*) – file mode. defaults to 'rw'.

Returns *nxs_file (NXFile)* – file handle to NeXus file.

get_scan(*scan_number*, *read_data=True*, *dismiss_update=False*)
Returns a scan object from the scan dict determined by the scan_number.

Parameters

- **scan_number** (*uint*) – number of the scan.
- **read_data** (*bool, optional*) – read data from source. Defaults to *False*.
- **dismiss_update** (*bool, optional*) – Dismiss update even if set as object attribute. Defaults to *False*.

Returns *scan (Scan)* – scan object.

get_scan_data(*scan_number*)
Returns data and meta information from a scan object from the *scan_dict* determined by the scan_number.

Parameters **scan_number** (*uint*) – number of the scan.

Returns *data (numpy.recarray[float])* – scan data. meta (dict()): scan meta information.

get_scan_list(*scan_number_list*, *read_data=True*)
Returns a list of scan object from the *scan_dict* determined by the list of scan_number.

Parameters

- **scan_number_list** (*list(uint)*) – list of numbers of the scan.
- **read_data** (*bool, optional*) – read data from source. Defaults to *False*.

Returns *scans (list(Scan))* – list of scan object.

get_scan_list_data(*scan_number_list*)
Returns data and meta information for a list of scan objects from the *scan_dict* determined by the scan_numbers.

Parameters **scan_number_list** (*list(uint)*) – list of numbers of the scan.

Returns *data (list(numpy.recarray[float]))* – list of scan data. meta (list(dict())): list scan meta information.

parse_nexus()
Parse the nexus file and populate the *scan_dict*.

read_all_scan_data()
Reads the data for all scan objects in the *scan_dict* from source.

read_nexus_scan_data(*scan*)
Reads the data for a given scan object from the nexus file.

Parameters **scan** (Scan) – scan object.

read_scan_data(*scan*)
Reads the data for a given scan object.

Parameters **scan** (Scan) – scan object.

save_all_scans_to_nexus()
Saves all scan objects in the *scan_dict* to the nexus file.

save_scan_to_nexus(*scan*, *nxs_file=''*)
Saves a scan to the nexus file.

update(*scan_number_list=[]*)
update the `scan_dict` either from the raw source file/folder or from the nexus file. The optional `scan_number_list` runs the update only if required for the included scan.

Attributes **scan_number_list** (*list[int]*) – explicit list of scans

class pyEvalData.io.**PalH5**(*name*, *file_name*, *file_path*, ***kwargs*)

Bases: *pyEvalData.io.source.Source*

Source implementation for PalH5 folder/files.

Parameters

- **name** (*str*) – name of the source
- **file_name** (*str*) – file name including extension, can include regex pattern.
- **file_path** (*str, optional*) – file path - defaults to `./`.

Keyword Arguments

- **start_scan_number** (*uint*) – start of scan numbers to parse.
- **stop_scan_number** (*uint*) – stop of scan numbers to parse. This number is included.
- **nexus_file_name** (*str*) – name for generated nexus file.
- **nexus_file_name_postfix** (*str*) – postfix for nexus file name.
- **nexus_file_path** (*str*) – path for generated nexus file.
- **read_all_data** (*bool*) – read all data on parsing. If false, data will be read only on demand.
- **read_and_forget** (*bool*) – clear data after read to save memory.
- **update_before_read** (*bool*) – always update from source before reading scan data.
- **use_nexus** (*bool*) – use nexus file to join/compress raw data.
- **force_overwrite** (*bool*) – forced re-read of raw source and re-generated of nexus file.

Attributes

- **log** (*logging.logger*) – logger instance from logging.
- **name** (*str*) – name of the source
- **scan_dict** (*dict(scan)*) – dict of scan objects with key being the scan number.
- **start_scan_number** (*uint*) – start of scan numbers to parse.
- **stop_scan_number** (*uint*) – stop of scan numbers to parse. This number is included.
- **file_name** (*str*) – file name including extension, can include regex pattern.
- **file_path** (*str, optional*) – file path - defaults to `./`.
- **nexus_file_name** (*str*) – name for generated nexus file.
- **nexus_file_name_postfix** (*str*) – postfix for nexus file name.
- **nexus_file_path** (*str*) – path for generated nexus file.
- **nexus_file_exists** (*bool*) – if nexus file exists.
- **read_all_data** (*bool*) – read all data on parsing.
- **read_and_forget** (*bool*) – clear data after read to save memory.
- **update_before_read** (*bool*) – always update from source before reading scan data.
- **use_nexus** (*bool*) – use nexus file to join/compress raw data.
- **force_overwrite** (*bool*) – forced re-read of raw source and re-generated of nexus file.

Methods:

parse_raw()	Parse the PalH5 folder and populate the *scan_dict*.
read_raw_scan_data(scan)	Reads the data for a given scan object from Sardana NeXus file.
check_nexus_file_exists()	Check if the nexus file is present and set *self.nexus_file_exists*.
clear_all_scan_data()	Clears the data for all scan objects in the *scan_dict*.
clear_scan_data(scan)	Clear the data for a given scan object.
get_all_scan_numbers()	Return the all scan number from the *scan_dict*.
get_last_scan_number()	Return the number of the last scan in the *scan_dict*.
get_nexus_file([mode])	Return the file handle to the NeXus file in a given `mode`.
get_scan(scan_number[, read_data, ...])	Returns a scan object from the scan dict determined by the scan_number.
get_scan_data(scan_number)	Returns data and meta information from a scan object from the *scan_dict* determined by the scan_number.
get_scan_list(scan_number_list[, read_data])	Returns a list of scan object from the *scan_dict* determined by the list of scan_number.
get_scan_list_data(scan_number_list)	Returns data and meta information for a list of scan objects from the *scan_dict* determined by the scan_numbers.
parse_nexus()	Parse the nexus file and populate the *scan_dict*.
read_all_scan_data()	Reads the data for all scan objects in the *scan_dict* from source.
read_nexus_scan_data(scan)	Reads the data for a given scan object from the nexus file.
read_scan_data(scan)	Reads the data for a given scan object.
save_all_scans_to_nexus()	Saves all scan objects in the *scan_dict* to the nexus file.
save_scan_to_nexus(scan[, nxs_file])	Saves a scan to the nexus file.
update([scan_number_list])	update the `scan_dict` either from the raw source file/folder or from the nexus file.

parse_raw()
Parse the PalH5 folder and populate the *scan_dict*.

read_raw_scan_data(*scan*)
Reads the data for a given scan object from Sardana NeXus file.

Parameters **scan** (Scan) – scan object.

check_nexus_file_exists()
Check if the nexus file is present and set *self.nexus_file_exists*.

clear_all_scan_data()
Clears the data for all scan objects in the *scan_dict*.

clear_scan_data(*scan*)
Clear the data for a given scan object.

Parameters **scan** (Scan) – scan object.

get_all_scan_numbers()
Return the all scan number from the *scan_dict*.

get_last_scan_number()
Return the number of the last scan in the *scan_dict*. If the *scan_dict* is empty return 0.

get_nexus_file(*mode='rw'*)
Return the file handle to the NeXus file in a given `mode`.

Parameters **mode** (*str, optional*) – file mode. defaults to 'rw'.

Returns *nxs_file (NXFile)* – file handle to NeXus file.

get_scan(*scan_number*, *read_data=True*, *dismiss_update=False*)
Returns a scan object from the scan dict determined by the scan_number.

Parameters

- **scan_number** (*uint*) – number of the scan.
- **read_data** (*bool, optional*) – read data from source. Defaults to *False*.
- **dismiss_update** (*bool, optional*) – Dismiss update even if set as object attribute. Defaults to *False*.

Returns *scan (Scan)* – scan object.

get_scan_data(*scan_number*)
Returns data and meta information from a scan object from the *scan_dict* determined by the scan_number.

Parameters **scan_number** (*uint*) – number of the scan.

Returns *data (numpy.recarray[float])* – scan data. meta (dict()): scan meta information.

get_scan_list(*scan_number_list*, *read_data=True*)
Returns a list of scan object from the *scan_dict* determined by the list of scan_number.

Parameters

- **scan_number_list** (*list(uint)*) – list of numbers of the scan.
- **read_data** (*bool, optional*) – read data from source. Defaults to *False*.

Returns *scans (list(Scan))* – list of scan object.

get_scan_list_data(*scan_number_list*)
Returns data and meta information for a list of scan objects from the *scan_dict* determined by the scan_numbers.

Parameters **scan_number_list** (*list(uint)*) – list of numbers of the scan.

Returns *data (list(numpy.recarray[float]))* – list of scan data. meta (list(dict())): list scan meta information.

parse_nexus()
Parse the nexus file and populate the *scan_dict*.

read_all_scan_data()
Reads the data for all scan objects in the *scan_dict* from source.

read_nexus_scan_data(*scan*)
Reads the data for a given scan object from the nexus file.

Parameters **scan** (Scan) – scan object.

read_scan_data(*scan*)
Reads the data for a given scan object.

Parameters **scan** (Scan) – scan object.

save_all_scans_to_nexus()
Saves all scan objects in the *scan_dict* to the nexus file.

save_scan_to_nexus(*scan*, *nxs_file=''*)
Saves a scan to the nexus file.

update(*scan_number_list=[]*)
update the `scan_dict` either from the raw source file/folder or from the nexus file. The optional `scan_number_list` runs the update only if required for the included scan.

Attributes **scan_number_list** (*list[int]*) – explicit list of scans

4.3.2 evaluation

Classes:

Evaluation(source)	Main class for evaluating data.

class pyEvalData.evaluation.**Evaluation**(*source*)
Bases: `object`

Main class for evaluating data. The raw data is accessed via a `Source` object. The evaluation allows to bin data, calculate errors and propagate them. There is also an interface to `lmfit` for easy batch-fitting.

Parameters **source** (*Source*) – raw data source.

Attributes

- **log** (*logging.logger*) – logger instance from logging.
- **clist** (*list[str]*) – list of counter names to evaluate.
- **cdef (dict{str** – str}): dict of predefined counter names and definitions.
- **xcol** (*str*) – counter or motor for x-axis.
- **t0** (*float*) – approx. time zero for delay scans to determine the unpumped region of the data for normalization.
- **custom_counters** (*list[str]*) – list of custom counters - default is []
- **math_keys** (*list[str]*) – list of keywords which are evaluated as numpy functions
- **statistic_type** (*str*) – 'gauss' for normal averaging, 'poisson' for counting statistics
- **propagate_errors** (*bool*) – propagate errors for dpendent counters.

Methods:

get_clist()	Returns a list of counters as defined by the user.
traverse_counters(clist[, source_cols])	Traverse all counters and replace all predefined counter definitions.
resolve_counter_name(col_name[, source_cols])	Replace all predefined counter definitions in a given counter name.
col_string_to_eval_string(col_string[, ...])	Use regular expressions in order to generate an evaluateable string from the counter string in order to append the new counter to the spec data.
add_custom_counters(spec_data, scan_num, ...)	Add custom counters to the spec data array.

continues on next page

Table 8 – continued from previous page

filter_data(data) **param data** DESCRIPTION.	
get_scan_data(scan_num) **param scan_num** DESCRIPTION.	
get_scan_list_data(scan_list) **param scan_num** DESCRIPTION.	
avg_N_bin_scans(scan_list[, xgrid, binning])	Averages data defined by the counter list, clist, onto an optional xgrid.
plot_scans(scan_list[, ylims, xlims, ...])	Plot a list of scans from the spec file.
plot_mesh_scan(scan_num[, skip_plot, ...])	Plot a single mesh scan from the spec file.
plot_scan_sequence(scan_sequence[, ylims, ...])	Plot a list of scans from the spec file.
export_scan_sequence(scan_sequence, path, ...)	Exports spec data for each scan list in the sequence as individual file.
fit_scans(scans, mod, pars[, ylims, xlims, ...])	Fit, plot, and return the data of scans.
fit_scan_sequence(scan_sequence, mod, pars)	Fit, plot, and return the data of a scan sequence.
get_last_fig_number()	Return the last figure number of all opened figures for plotting data in the same figure during for-loops.
get_next_fig_number()	Return the number of the next available figure.

get_clist()
Returns a list of counters as defined by the user. If the counters where defined in a `dict` it will be converted to a `list` for backwards compatibility.

Returns *clist (list[str])* – list of counter names to evaluate.

traverse_counters(*clist*, *source_cols=''*)
Traverse all counters and replace all predefined counter definitions. Returns also a list of the included source counters for error propagation.

Parameters

- **clist** (*list[str]*) – Initial counter list.
- **source_cols** (*list[str], optional*) – counters in the raw source data.

Returns

(tuple) –

- *resolved_counters (list[str])* - resolved counters.
- *source_counters (list[str])* - all source counters in the resolved counters.

resolve_counter_name(*col_name*, *source_cols=''*)
Replace all predefined counter definitions in a given counter name. The function works recursively.

Parameters

- **col_name** (*str*) – initial counter string.
- **source_cols** (*list[str], optional*) – columns in the source data.

Returns

(tuple) –

- *col_string (str)* - resolved counter string.

- *source_counters (list[str])* - source counters in the col_string

col_string_to_eval_string(*col_string*, *array_name='spec_data'*)

Use regular expressions in order to generate an evaluateable string from the counter string in order to append the new counter to the spec data.

Parameters

- **col_string** (*str*) – Definition of the counter.
- **mode** (*int*) – Flag for different modes

Returns

eval_string (str) –

Evaluateable string to add the new counter to the spec data.

add_custom_counters(*spec_data*, *scan_num*, *source_counters*)

Add custom counters to the spec data array. This is a stub for child classes.

Parameters

- **spec_data** (*ndarray*) – Data array from the spec scan.
- **scan_num** (*int*) – Scan number of the spec scan.
- **list** (*source_counters*) – List of the source counters and custom counters from the clist and xcol.

Returns *spec_data (ndarray)* – Updated data array from the spec scan.

filter_data(*data*)

Parameters **data** (*TYPE*) – DESCRIPTION.

Returns *TYPE* – DESCRIPTION.

get_scan_data(*scan_num*)

Parameters **scan_num** (*TYPE*) – DESCRIPTION.

Returns *TYPE* – DESCRIPTION.

get_scan_list_data(*scan_list*)

Parameters **scan_num** (*TYPE*) – DESCRIPTION.

Returns *TYPE* – DESCRIPTION.

avg_N_bin_scans(*scan_list*, *xgrid=array([], dtype=float64)*, *binning=True*)

Averages data defined by the counter list, clist, onto an optional xgrid. If no xgrid is given the x-axis data of the first scan in the list is used instead.

Parameters

- **scan_list** (*List[int]*) – List of scan numbers.
- **xgrid** (*Optional[ndarray]*) – Grid to bin the data to - default in empty so use the x-axis of the first scan.

Returns *avg_data (ndarray)* – Averaged data for the scan list. std_data (ndarray) : Standart derivation of the data for the scan list. err_data (ndarray) : Error of the data for the scan list. name (str) : Name of the data set.

plot_scans(*scan_list*, *ylims=[]*, *xlims=[]*, *fig_size=[]*, *xgrid=[]*, *yerr='std'*, *xerr='std'*, *norm2one=False*, *binning=True*, *label_text=''*, *title_text=''*, *skip_plot=False*, *grid_on=True*, *ytext=''*, *xtext=''*, *fmt='-o'*)

Plot a list of scans from the spec file. Various plot parameters are provided. The plotted data are returned.

Parameters

- **scan_list** (*List[int]*) – List of scan numbers.
- **ylims** (*Optional[ndarray]*) – ylim for the plot.
- **xlims** (*Optional[ndarray]*) – xlim for the plot.
- **fig_size** (*Optional[ndarray]*) – Figure size of the figure.
- **xgrid** (*Optional[ndarray]*) – Grid to bin the data to - default in empty so use the x-axis of the first scan.
- **yerr** (*Optional[ndarray]*) – Type of the errors in y: [err, std, none] default is 'std'.
- **xerr** (*Optional[ndarray]*) – Type of the errors in x: [err, std, none] default is 'std'.
- **norm2one** (*Optional[bool]*) – Norm transient data to 1 for t < t0 default is False.
- **label_text** (*Optional[str]*) – Label of the plot - default is none.
- **title_text** (*Optional[str]*) – Title of the figure - default is none.
- **skip_plot** (*Optional[bool]*) – Skip plotting, just return data default is False.
- **grid_on** (*Optional[bool]*) – Add grid to plot - default is True.
- **ytext** (*Optional[str]*) – y-Label of the plot - defaults is none.
- **xtext** (*Optional[str]*) – x-Label of the plot - defaults is none.
- **fmt** (*Optional[str]*) – format string of the plot - defaults is -o.

Returns *y2plot (OrderedDict)* – y-data which was plotted. x2plot (ndarray) : x-data which was plotted. yerr2plot (OrderedDict) : y-error which was plotted. xerr2plot (ndarray) : x-error which was plotted. name (str) : Name of the data set.

plot_mesh_scan(*scan_num*, *skip_plot=False*, *grid_on=False*, *ytext=''*, *xtext=''*, *levels=20*, *cbar=True*)

Plot a single mesh scan from the spec file. Various plot parameters are provided. The plotted data are returned.

Parameters

- **scan_num** (*int*) – Scan number of the spec scan.
- **skip_plot** (*Optional[bool]*) – Skip plotting, just return data default is False.
- **grid_on** (*Optional[bool]*) – Add grid to plot - default is False.
- **ytext** (*Optional[str]*) – y-Label of the plot - defaults is none.
- **xtext** (*Optional[str]*) – x-Label of the plot - defaults is none.
- **levels** (*Optional[int]*) – levels of contour plot - defaults is 20.
- **cbar** (*Optional[bool]*) – Add colorbar to plot - default is True.

Returns *xx, yy, zz* – x,y,z data which was plotted

plot_scan_sequence(*scan_sequence*, *ylims=[]*, *xlims=[]*, *fig_size=[]*, *xgrid=[]*, *yerr='std'*, *xerr='std'*, *norm2one=False*, *binning=True*, *sequence_type=''*, *label_text=''*, *title_text=''*, *skip_plot=False*, *grid_on=True*, *ytext=''*, *xtext=''*, *fmt='-o'*)

Plot a list of scans from the spec file. Various plot parameters are provided. The plotted data are returned.

Parameters

- **(List[** (*scan_sequence*) – List/Tuple[List[int], int/str]]) : Sequence of scan lists and parameters.
- **ylims** (*Optional[ndarray]*) – ylim for the plot.
- **xlims** (*Optional[ndarray]*) – xlim for the plot.
- **fig_size** (*Optional[ndarray]*) – Figure size of the figure.
- **xgrid** (*Optional[ndarray]*) – Grid to bin the data to - default in empty so use the x-axis of the first scan.
- **yerr** (*Optional[ndarray]*) – Type of the errors in y: [err, std, none] default is 'std'.
- **xerr** (*Optional[ndarray]*) – Type of the errors in x: [err, std, none] default is 'std'.
- **norm2one** (*Optional[bool]*) – Norm transient data to 1 for t < t0 default is False.
- **sequence_type** (*Optional[str]*) – Type of the sequence: [fluence, delay, energy, theta, position, voltage, none, text] - default is enumeration.
- **label_text** (*Optional[str]*) – Label of the plot - default is none.
- **title_text** (*Optional[str]*) – Title of the figure - default is none.
- **skip_plot** (*Optional[bool]*) – Skip plotting, just return data default is False.
- **grid_on** (*Optional[bool]*) – Add grid to plot - default is True.
- **ytext** (*Optional[str]*) – y-Label of the plot - defaults is none.
- **xtext** (*Optional[str]*) – x-Label of the plot - defaults is none.
- **fmt** (*Optional[str]*) – format string of the plot - defaults is -o.

Returns *sequence_data (OrderedDict)* – Dictionary of the averaged scan data. parameters (List[str, float]) : Parameters of the sequence. names (List[str]) : List of names of each data set. label_texts (List[str]) : List of labels for each data set.

export_scan_sequence(*scan_sequence*, *path*, *fileName*, *yerr='std'*, *xerr='std'*, *xgrid=[]*, *norm2one=False*, *binning=True*)

Exports spec data for each scan list in the sequence as individual file.

Parameters

- **(List[** (*scan_sequence*) – List/Tuple[List[int], int/str]]) : Sequence of scan lists and parameters.
- **path** (*str*) – Path of the file to export to.
- **fileName** (*str*) – Name of the file to export to.
- **yerr** (*Optional[ndarray]*) – Type of the errors in y: [err, std, none] default is 'std'.
- **xerr** (*Optional[ndarray]*) – Type of the errors in x: [err, std, none] default is 'std'.
- **xgrid** (*Optional[ndarray]*) – Grid to bin the data to - default in empty so use the x-axis of the first scan.
- **norm2one** (*Optional[bool]*) – Norm transient data to 1 for t < t0 default is False.

fit_scans(*scans*, *mod*, *pars*, *ylims=[]*, *xlims=[]*, *fig_size=[]*, *xgrid=[]*, *yerr='std'*, *xerr='std'*, *norm2one=False*, *binning=True*, *sequence_type='text'*, *label_text=''*, *title_text=''*, *ytext=''*, *xtext=''*, *select=''*, *fit_report=0*, *show_single=False*, *weights=False*, *fit_method='leastsq'*, *offset_t0=False*, *plot_separate=False*, *grid_on=True*, *fmt='o'*)

Fit, plot, and return the data of scans.

This is just a wrapper for the fit_scan_sequence method

fit_scan_sequence(*scan_sequence*, *mod*, *pars*, *ylims=[]*, *xlims=[]*, *fig_size=[]*, *xgrid=[]*, *yerr='std'*, *xerr='std'*, *norm2one=False*, *binning=True*, *sequence_type=''*, *label_text=''*, *title_text=''*, *ytext=''*, *xtext=''*, *select=''*, *fit_report=0*, *show_single=False*, *weights=False*, *fit_method='leastsq'*, *offset_t0=False*, *plot_separate=False*, *grid_on=True*, *last_res_as_par=False*, *sequence_data=[]*, *fmt='o'*)

Fit, plot, and return the data of a scan sequence.

Parameters

- **(List[** (*scan_sequence*) – List/Tuple[List[int], int/str]]) : Sequence of scan lists and parameters.
- **mod** (*Model[lmfit]*) – lmfit model for fitting the data.
- **pars** (*Parameters[lmfit]*) – lmfit parameters for fitting the data.
- **ylims** (*Optional[ndarray]*) – ylim for the plot.
- **xlims** (*Optional[ndarray]*) – xlim for the plot.
- **fig_size** (*Optional[ndarray]*) – Figure size of the figure.
- **xgrid** (*Optional[ndarray]*) – Grid to bin the data to - default in empty so use the x-axis of the first scan.
- **yerr** (*Optional[ndarray]*) – Type of the errors in y: [err, std, none] default is 'std'.
- **xerr** (*Optional[ndarray]*) – Type of the errors in x: [err, std, none] default is 'std'.
- **norm2one** (*Optional[bool]*) – Norm transient data to 1 for t < t0 default is False.
- **sequence_type** (*Optional[str]*) – Type of the sequence: [fluence, delay, energy, theta] - default is fluence.
- **label_text** (*Optional[str]*) – Label of the plot - default is none.
- **title_text** (*Optional[str]*) – Title of the figure - default is none.
- **ytext** (*Optional[str]*) – y-Label of the plot - defaults is none.
- **xtext** (*Optional[str]*) – x-Label of the plot - defaults is none.
- **select** (*Optional[str]*) – String to evaluate as select statement for the fit region - default is none
- **fit_report** (*Optional[int]*) – Set the fit reporting level: [0: none, 1: basic, 2: full] default 0.
- **show_single** (*Optional[bool]*) – Plot each fit seperately - default False.
- **weights** (*Optional[bool]*) – Use weights for fitting - default False.
- **fit_method** (*Optional[str]*) – Method to use for fitting; refer to lmfit - default is 'leastsq'.
- **offset_t0** (*Optional[bool]*) – Offset time scans by the fitted t0 parameter - default False.

- **plot_separate** (*Optional[bool]*) – A single plot for each counter default False.
- **grid_on** (*Optional[bool]*) – Add grid to plot - default is True.
- **last_res_as_par** (*Optional[bool]*) – Use the last fit result as start values for next fit - default is False.
- **sequence_data** (*Optional[ndarray]*) – actual exp. data are externally given. default is empty
- **fmt** (*Optional[str]*) – format string of the plot - defaults is -o.

Returns *res (Dict[ndarray])* – Fit results. parameters (ndarray) : Parameters of the sequence. sequence_data (OrderedDict) : Dictionary of the averaged scan data.equenceData

get_last_fig_number()

Return the last figure number of all opened figures for plotting data in the same figure during for-loops.

Returns *fig_number (int)* – last figure number of all opened figures.

get_next_fig_number()

Return the number of the next available figure.

Returns *next_fig_number (int)* – next figure number of all opened figures.

4.3.3 helpers

Functions:

edges4grid(grid)	Returns the edges for a given grid vector as well as the corresponding width of these bins.
bin_data(y, x, X[, statistic])	This is a wrapper around scipy's binned_statistic.

pyEvalData.helpers.**edges4grid**(*grid*)

Returns the edges for a given grid vector as well as the corresponding width of these bins.

The `grid` is NOT altered - on purpose! So even if the `grid` is not *unique* there will be bins of width 0.

Be also aware of the hanling of the first and last bin, as they will contain values which will lay outside of the original `grid`.

grid x x x x x x x

edges | | | | | | | | |

binwidth <—> <—> <—> <—> <—> <—> <—> <—>

Attributes **grid** (*ndarray[float]*) – array of grid points.

Returns

(tuple) –

- *edges (ndarray[float])* - array of edges.
- *binwidth (ndarray[float])* - array of bin widths.

pyEvalData.helpers.**bin_data**(*y*, *x*, *X*, *statistic='mean'*)

This is a wrapper around scipy's binned_statistic. In the first step possbile masked elements from the input arrays *x* and *y* are removed. The same applies for the new grid array *X* which is also sorted and made unique.

In a second step the edges for the new grid are calculated by `edges4grid` and used to calculate the new binned values *Y* by using `scipy.stats.binned_statistsic`.

The type of *statistic* can be chosen. In case of *sum* Poisson statistics are applied to calculate the standard deviation of the binned values *Y*. Also errors due to the horizontal binning are calculated and returned. All return values contain only elements with according non-zero bins.

Parameters

- **y** (*ndarray[float]*) – input y array.
- **x** (*ndarray[float]*) – input x array.
- **X** (*ndarray[float]*) – new grid array.
- **statistic** (*str, optional*) – type of statistics used for scipy's binned_statistic - default is mean.

Returns

(tuple) –

- *Y (ndarray[float])* - binned Y data without zero-bins.
- *X (ndarray[float])* - new X grid array.
- *Yerr (ndarray[float])* - Error for Y, according to statistic.
- *Xerr (ndarray[float])* - Error for Y, according to statistic.
- *Ystd (ndarray[float])* - Std for Y, according to statistic.
- *Xstd (ndarray[float])* - Std for X, according to statistic.
- *edges (ndarray[float])* - Edges of binned data.
- *bins (ndarray[float])* - Indices of the bins.
- *n (ndarray[float])* - Number of values per given bin.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

I

M

P

R

S

T

U