nPDyn

Release v2.0.1+15.g1fdeaf1.dirty

Feb 21, 2022

Contents

nPDyn is a Python based API for analysis of neutron backscattering data.

The API aims at providing a lightweight, user-friendly and modular tool to process and analyze quasi-elastic neutron scattering (QENS) and fixed-window scans (FWS) obtained with backscattering spectroscopy.

nPDyn can be used in combination with other software for neutron data analysis such as Mantid. The API provides an interface to Mantid workspaces for that.

An important feature of nPDyn is the modelling interface, which is designed to be highly versatile and intuitive for multidimensional dataset with global and non-global parameters. The modelling in nPDyn is provided by builtin classes, `params.Parameters`, `model.Model` and `model.Component`. nPDyn provides also some helper functions to use lmfit as modelling backend. See *Fit data* for details.

Eventually, some plotting methods are available to examine processed data, model fitting and optimized parameters.

CHAPTER 1

Installation:

1.1 Unix and Windows

For installation within your python framework, use:

```
make install
```

or

```
python3 setup.py install
```

On Windows, the path to the GSL library can be provided using:

```
python.exe setup.py install --gsl-path="my/path/to/GSL/root/dir/"
```

CHAPTER 2

Getting started

The nPDyn API is organized around a `sample.Sample` class. This class inherits from the NumPy ndarray class with some extra features added, such as neutron scattering-specific attributes, binning, data correction algorithm, automatic error propagation and data fitting.

In a neutron backscattering experiment, there is not only the measurement of samples but also some calibration measurements like vanadium, empty cell and solvent signal (often D_2O). Some methods of the `sample.Sample` class can be used to perform normalization or absorption correction using the dataset corresponding to vanadium or empty cell, respectively. These calibration dataset can be used also in the *fit* function to automatically add a background or perform a convolution with the resolution function.

Details regarding importation of data are available in the *Import data* section of the documentation.

Importantly, nPDyn provides versatile tools for model building and fitting to the data. See the section *Fit data* for details.

Finally, a `plot.plot()` method is provided for easy visualisation of the data and the fit results.

CHAPTER 3

Documentation

3.1 Import data

nPDyn provides various ways to handle data.

The data importation routines are found in *nPDyn.dataParsers* module.

Sample data are stored in the `sample.Sample` class. Some useful information about the `sample.Sample` class and about different data importation routines can be found in the following.

3.1.1 Access the data values

Each imported data consists in a `sample.Sample` class. The class is essentially a NumPy ndarray with extra features specific to neutron scattering dataset. In addition, the class contains several methods for processing and fitting.

The specific attributes are the following:

- **filename**, the name of the file used to extract the data.
- **errors**, the errors associated with scattering data.
- **energies**, the energy transfers associated with the data.
- **time**, the experimental time.
- **wavelength**, the wavelength of the incoming neutrons.
- **name**, the name for the sample.
- **temperature**, the temperature(s) used experimentally.
- **concentration**, the concentration of the sample.
- **pressure**, the pressure used experimentally.
- **buffer**, a description of the buffer used experimentally.
- **q**, the values for the momentum transfer q.

- **beamline**, the name of the beamline used.
- **observable_name**, the name of the observable variable.

These attributes might be empty or not depending on the source file.

Note: The **errors** metadata is special as it is updated for various operations that are performed on the data array such as indexing or for the use of universal functions. For instance, indexing of the data will be performed on **errors** as well if its shape is the same as for the data. Also, addition, subtraction and other universal functions will lead to automatic error propagation. Some other metadata might change as well, like **q**, but only for the use of methods specific of the `Sample` class and not for methods inherited from numpy.

3.1.2 Raw data

Raw dataset, as generated on IN16B at the ILL, can be imported directly. The algorithm has several options allowing for detector grouping, unmirroring, integrating and summation of the scans.

See *in16b_qens_scans_reduction.IN16B_QENS* or *in16b_fws_scans_reduction.IN16B_FWS* for example.

To import raw data, the following can be used:

```
from nPDyn.dataParsers import IN16B_QENS, IN16B_FWS

# we can use a path to a folder or a list of strings
# here for FWS data where we only keep elastic scans
# and we choose the observable to be the temperature
sample = IN16B_FWS(
    'myDataFolder/',
    offset=0.0,
    observable='temperature'
)

# ...and here for a range of QENS data with .xml detector grouping file
sample = IN16B_QENS(
    'myDataFolder/scan01:scan10.nxs',
    detGroup='IN16B_detGroup.xml'
)
```

Different methods and properties of the dataset are accessible through this list, e.g., the momentum transfers using:

```
>>> sample.q
array([0.19102381, 0.29274028, 0.43543718, 0.56747019, 0.69687497,
0.82305221, 0.94541753, 1.0634042 , 1.17646584, 1.28407863,
1.38574439, 1.48099215, 1.5693807 , 1.65050083, 1.72397668,
1.78946811, 1.84667172, 1.89532256])
```

3.1.3 Nexus (hdf5) files

Nexus files as generated by Mantid can be read by nPDyn using the `dataParsers.mantidNexus.processNexus()` method.

The file will be assumed to be a Nexus file if the extension is '.nxs', hence the following:

```
from nPDyn.dataParsers import processNexus

sample = processNexus('mySample01.nxs', FWS=False)
```

will import all files using the Nexus file parser.

3.1.4 .inx files

Similarly to Nexus files, nPDyn can read '.inx' files as generated by the software SLAW available at the MLZ in Garching, Germany. The usage is essentially the same as for Nexus files:

```
from nPDyn.dataParsers import inxConvert

sample = inxConvert('mySample01.nxs', FWS=False)
```

3.2 Process data

nPDyn provides several data processing methods, which includes binning, normalization, scaling, empty cell correction, Paalman-Pings coefficient calculation and detector selection.

These are described below.

3.2.1 Arithmetic operations

The *sample.Sample* is essentially a NumPy array, so arithmetic operations can be used as for any array.

```
>>> from nPDyn.dataParsers import processNexus
>>> sample1 = processNexus('mySample1.nxs')
>>> sample2 = processNexus('mySample2.nxs')
>>> corrected_sample = sample1 - 0.95 * sample2
```

Again, the errors are automatically propagated for most of the commonly used operators (addition, subtraction multiplication, division, exponentiation, logarithm, power, square, square root).

3.2.2 Binning

The dataset can be binned along any axis. This can be done using the method *sample.Sample.bin()*.

Here is an example code with quasi-elastic neutron scattering (QENS) data:

```
>>> from nPDyn.dataParsers import processNexus
>>> sample = processNexus('myData.nxs')
>>> sample.shape
(1, 18, 1004)
>>> # 1 observable, 18 detectors/q values and 1004 energy transfers
>>> sample = sample.bin(5, axis=2)  # bins of 5 points on the energy axis
>>> sample.shape
(1, 18, 200)
>>> sample.energies.shape
(200,)
```

3.2.3 Normalization

Normalization of data can be done by dividing by the integration of themselves, of vanadium or of data at low temperature.

The following:

```
>>> from nPDyn.dataParsers import processNexus
>>> sample = processNexus('myData.nxs')
>>> sample = sample.normalize()
```

will apply normalization using the integration of the 'sample' dataset.

Using

```
>>> from nPDyn.dataParsers import processNexus
>>> sample = processNexus('myData.nxs')
>>> vanadium = processNexus('vanadium.nxs')
>>> sample = sample.normalize(vanadium)
```

The signal of the vanadium will be integrated and used for normalization. If a fitted model exists for the vanadium, it will be used instead of the experimental data.

3.2.4 Background corrections

The correction of background, often using empty cell signal, can be done either using simple arithmetic operators or using the *sample.Sample.absorptionCorrection()* method.

For instance,

```
>>> from nPDyn.dataParsers import processNexus
>>> sample = processNexus('mySample.nxs')
>>> empty_cell = processNexus('empty_cell.nxs')
>>> sample = sample.absorptionCorrection(
...     empty_cell,
...     canScaling=0.95,
...     canType='tube',
...     useModel=False
... )
```

will computes the Paalman-Ping coefficient for a tubular sample holder, scale the empty cell data provided factor and apply the absorption correction to the dataset.

3.2.5 Selection of data range

The user will very likely want to restrain the analysis to a specific range of momentum transfers q or observable values. To this end, some self-explaining methods are provided to select a range based on values instead of indices:

```
>>> from nPDyn.dataParsers import processNexus
>>> sample = processNexus('mySample.nxs')
>>> sample.shape
(10, 18, 1004)
>>> sample = sample.get_q_range(0.3, 1.7)
>>> sample = sample.get_observable_range(280, 320)
>>> sample.shape
(4, 14, 1004)
```

3.3 Fit data

nPDyn relies on a builtin implementation to model and fit data, but provides also some methods to fit your data using lmfit as modelling and fitting backend.

In the following, we will introduce data modelling using two type of data and analysis, the first being the fit of quasi-elastic neutron scattering (QENS) measurement on a protein solution sample and the second the fit of elastic fixed-window scans (EFWS) of a protein powder sample.

The QENS data will be modelled using the following:

$$\mathrm{S}(\mathrm{q}, \hbar\omega) = \mathrm{R}(\mathrm{q}, \hbar\omega) \otimes \beta_\mathrm{q} \left[\alpha\mathcal{L}_\Gamma + (1-\alpha)\mathcal{L}_{\Gamma+\gamma}\right] + \beta_{\mathrm{D_2O}}\mathcal{L}_{\mathrm{D_2O}} \tag{3.1}$$

where q is the momentum transfer, $\hbar\omega$ the energy transfer, $\mathrm{R}(\mathrm{q}, \hbar\omega)$ is the resolution function (here a pseudo-Voigt profile), β_q a vector of scalars accounting for detector efficiency (one scalar for each q), α a scalar between 0 and 1, $\mathcal{L}_\Gamma$ a Lorentzian of accounting for center-of-mass diffusion with a explicit q-dependent width $\Gamma = \mathrm{D_s q^2}$, where $\mathrm{D_s}$ is the self-diffusion coefficient, $\mathcal{L}_{\Gamma+\gamma}$ is a Lorentzian accounting for internal dynamics with $\gamma = \frac{\mathrm{D_i q^2}}{1+\mathrm{D_i q^2}\tau}$ (see[1]) and $\beta_{\mathrm{D_2O}}\mathcal{L}_{\mathrm{D_2O}}$ accounting for the signal from the $\mathrm{D_2O}$.

The EFWS data will be modelled using a simple Gaussian to extract the mean-squared displacement (MSD) as a function of temperature:

$$\mathrm{S}(\mathrm{q}, 0) = \mathrm{e}^{-\frac{\mathrm{q^2 MSD}}{6}} \tag{3.2}$$

We use the sample data in the the test suite of nPDyn (from package root directory, use `cd nPDyn/tests/sample_data/` and we initiate our dataset using, for QENS:

```
>>> from nPDyn.dataParsers import processNexus
>>> import numpy as np
>>> qens = processNexus('lys_part_01_QENS_before_280K.nxs')
>>> vana = processNexus('vana_QENS_280K.nxs')
>>> ec = processNexus('empty_cell_QENS_280K.nxs')
>>> buffer = processNexus('D2O_QENS_280K.nxs')
>>> # Perform some data processing
>>> qens, vana, ec, buffer = (
... val.bin(5) for val in (qens, vana, ec, buffer)
... )
>>> qens, vana, buffer = (
... val - 0.95 * ec for val in (qens, vana, buffer)
... )
>>> qens, vana, ec, buffer = (
... val.get_q_range(0.4, 1.8) for val in (qens, vana, ec, buffer)
... )
>>> # Extract momentum transfers for modelling and make it 2D
>>> q = qens.q[:, None]
```

and for EFWS:

```
>>> from nPDyn.dataParsers import inxConvert
>>> efws = inxConvert.convert('D_syn_fibers_elastic_10to300K.inx', True)
>>> efws = efws.bin(5, 0)
>>> efws /= efws[:5].mean(0)
>>> efws = efws.get_q_range(0.2, 0.8)
```

[1] https://doi.org/10.1103/PhysRev.119.863

3.3.1 Using builtin model backend

The builtin modelling interface has been designed to be easy to use and adapted to the multi-dimensional dataset obtained with neutron backscattering spectroscopy and a mix of global and non-global parameters.

The basic workflow is as follows:

1. Create a **Parameter** instance with parameters that can be scalar, 1D, 2D or any shaped arrays.
2. Create a **Model** instance that is initiated with the prviously created parameters.
3. Add several **Component** or other **Model** to this model. Each component is associated with a Python function, the arguments of which can be dynamically defined at the creation of the component using an expression as a string as shown below.
4. Fit your data!

For the QENS data, we first model the resolution function using a pseudo-voigt profile. To this end, we use the `builtins.modelPVoigt()` builtin model from nPDyn. The same is done for D_2O background using the `builtins.modelD2OBackground()` builtin model.

Simply use:

```
>>> from nPDyn.models.builtins import modelPVoigt
>>> from nPDyn.models.builtins import modelCalibratedD2O
>>> vana.fit(modelPVoigt(q, 'resolution'))
>>> buffer.fit(modelCalibratedD2O(q, temp=280))
```

With a little anticipation on this documentation, you can use the following to look at the fit result:

```
>>> from nPDyn.plot import plot
>>> plot(vana, buffer)
```

Create parameters

For the QENS sample, there are 6 parameters, namely β_q, α, D_s, D_i, τ, and β_{D_2O}.

We can thus create the **Parameters** instance:

```
>>> from nPDyn.models import Parameters
>>> pQENS = Parameters(
...     beta={'value': np.zeros_like(q) + 1, 'bounds': (0., np.inf)},
...     alpha={'value': 0.5, 'bounds': (0., 1)},
...     Ds={'value': 5, 'bounds': (0., 100)},
...     Di={'value': 20, 'bounds': (0., 100)},
...     tau={'value': 1, 'bounds': (0., np.inf)},
... )
```

For the EFWS sample, we only have the MSD and we use a slightly different way to instantiate the **Parameters** instance for demonstration purpose:

```
>>> from nPDyn.models import Model
>>> pEFWS = Parameters(msd=0.5)
>>> pEFWS.set('msd', bounds=(0., np.inf), fixed=False)
```

Instantiate a Model

Instantiating a **Model** is very straightforward, just use:

```
>>> modelQENS = Model(pQENS, 'QENS')  # for QENS data
>>> modelEFWS = Model(pEFWS, 'EFWS')  # for EFWS data
```

Add components

The `modelQENS` model should contain three components, or three lineshapes, as we can see in equation (3.1), namely a Lorentzian for center-of-mass diffusion, a Lorentzian for internal dynamics and the model we used for D_2O background. We can add them using:

```
>>> from nPDyn.models import Component
>>> from nPDyn.models.presets import lorentzian
>>> modelQENS.addComponent(Component(
...     'center-of-mass',
...     lorentzian,
...     scale='beta * alpha',  # will find the parameters values in pQENS
...     width='Ds * q**2',  # we will give q on the call to the fit method
...     center=0))  # we force the center to be at 0
...                 # (as it is given by the convolution with resolution)
>>> # we can add, subtract, multiply or divide a model using a Component or
>>> # another Model
>>> internal = Component(
...     'internal',
...     lorentzian,
...     scale='beta * (1 - alpha)',
...     width='Di * q**2 / (1 + Di * q**2 * tau)',
...     center=0)  # we force the center to be at 0
...                # (as it is given by the convolution with resolution)
>>> modelQENS += internal
>>> # for the D2O signal, we use a lambda function to include the scaling
>>> # note this can be done automatically with the 'bkgd' and
>>> # 'volume_fraction_bkgd' arguments of the fit function.
>>> modelQENS.addComponent(Component(
...     '$D_2O$',  # we can use LaTeX for the component and model names
...     lambda x, scale: scale * buffer.fit_best(x=x)[0],
...     scale=0.95,
...     skip_convolve=True))  # we do not want to convolve this
>>>                           # component with resolution
```

The `modelEFWS` model uses the momentum transfer q as independent variable, which will be passed later upon fitting and it contains only one component. Here, we use:

```
>>> from nPDyn.models.presets import gaussian
>>> modelEFWS.addComponent(Component(
...     'EISF',
...     lambda x, scale, msd: scale * np.exp(-x**2 * msd / 6)))
```

Fit data

The class *sample.Sample* provides a method to fit the data.

Here, we use it and write for QENS:

```
>>> qens.fit(
...     modelQENS,
```

(continues on next page)

(continued from previous page)

```
...     res=vana,
...     fit_method='basinhopping',
...     fit_kws={'niter': 10, 'disp': True}
... )
```

and for EFWS, where we set the independent variable to a column vector containing the momentum transfer q values:

```
>>> efws.fit(
...     modelEFWS,
...     x=efws.q[:, None]
... )
```

The fitted parameters can be saved in JSON format using (for the first observable):

```
>>> qens.params[0].writeParams(<'file_name'>)
```

Subsequently, the parameters can be imported using:

```
>>> qens.params[0].loadParams(<'file_name'>)
```

3.3.2 Using *lmfit* backend

In addition to the builtin model interface of nPDyn, the API also provides some helper functions to use the lmfit package. This package is more advanced and exhaustive than the builtin model interface but it is less adapted to multi-dimensional dataset with global and non-global parameters.

This is where the presets and builtin models in nPDyn come into play, to make it easier to use within the analysis workflow of neutron backscattering data.

The interface with lmfit relies on the *lmfit_presets.build_2D_model()* function.

We present here the analysis of QENS data using equation (3.1).

Build model

The function *lmfit_presets.build_2D_model()* uses a formatted string to build a 2D model where the words flanked by curly braces { } are considered as parameters.

The resolution function and the D_2O background signal can make use of the provided presets *lmfit_presets.pseudo_voigt()* and *lmfit_presets.calibratedD2O()*, we thus use:

```
>>> from nPDyn.lmfit.lmfit_presets import pseudo_voigt, calibratedD2O
>>> vana.fit(pseudo_voigt(q, prefix='res_'))
>>> buffer.fit(calibratedD2O(q, 0.95, 280, prefix='D2O_'))
>>> q = qens.q
```

To build the model for the protein sample, we use the function *lmfit_presets.build_2D_model()* to get the part inside square brackets in (3.1) and we will convolve with the resolution and add the D2O manually:

```
>>> from nPDyn.lmfit.lmfit_presets import build_2D_model
>>> # let us start with the formatted text for the center-of-mass term.
>>> comText = ("{beta} * {alpha} * {Ds} * {q}**2 / (np.pi * "
...            "(x**2 + ({Ds} * {q}**2)**2))")
>>> # same for the internal dynamics term
```

(continues on next page)

(continued from previous page)

```
>>> jumpDiffText = ("{beta} * (1 - {alpha}) * "
...                 "{Di} * {q}**2 / (1 + {Di} * {q}**2 * {tau}) / "
...                 "(np.pi * (x**2 + ({Di} * {q}**2 / "
...                 "(1 + {Di} * {q}**2 * {tau}))**2))")
>>> # now we build the components
>>> comModel = build_2D_model(
...     q,
...     'com',
...     comText,
...     paramGlobals=['alpha', 'Ds'],
...     bounds={
...         'beta': (0., np.inf),
...         'alpha': (0, 1),
...         'Ds': (0.01, np.inf)},  # non-zero min to avoid infinites
...     defVals={'alpha': 0.5,
...              'Ds': 5,
...              'beta': 1},
...     prefix='com_')
>>> jumpDiffModel = build_2D_model(
...     q,
...     'jumpDiff',
...     jumpDiffText,
...     paramGlobals=['alpha', 'Di', 'tau'],
...     bounds={
...         'beta': (0., np.inf),
...             'alpha': (0, 1),
...             'Di': (0.01, np.inf),  # non-zero min to avoid infinites
...             'tau': (0., np.inf)},
...         defVals={'beta': 1,
...                  'alpha': 0.5,
...                  'Di': 30,
...                  'tau': 10},
...         prefix='jd_')
>>> # and we assemble them
>>> model = comModel + jumpDiffModel
>>> # some parameters are the same for the two components,
>>> # so we set them equals using 'expr' hint
>>> model.set_param_hint('com_alpha', expr='jd_alpha')
>>> for i in range(q.size):
...     model.set_param_hint('com_beta_%i' % i, expr='jd_beta_%i' % i)
```

And finally, we add the D_2O signal with a scaling factor:

```
>>> # now we add the component for the D2O signal
>>> from nPDyn.lmfit.lmfit_presets import hline
>>> scale = hline(q, prefix='bD2O_')
>>> d2OModel = scale * qens.D2OData.model
>>> d2OModel.param_hints.update(qens.D2OData.getFixedOptParams(0))
>>> fitModel = model + d2OModel
```

Fit data

Data fitting can be done using the same functions as when using the builtin models. The `fit_method` and some other keywords are different and should correspond to the keywords expected in lmfit (see *lmfit* documentation for details).

Here, we can simply use:

```
>>> qens.fit(fitModel, res=vana)
```

to fit the data using *lmfit* default parameters.

3.3.3 References

3.4 Plot data

nPDyn provides a plot window for quasi-elastic neutron scattering (QENS) and elastic/inelastic fixed-window scans (E/IFWS) data.

It can be used as follows:

```
>>> from nPDyn.plot import plot
>>> plot(sample)
```

Using the result of the fitting procedure presented in the *Fit data* section, the data, the fitted model model and the parameters can be examined using the window as shown below:

3.5 API reference

3.5.1 Sample

Handle data associated with a sample.

class sample.**Sample**(*arr*, *errors=None*, ***kwargs*)

Handle the measured data along with metadata.

This class is a subclass of the numpy.ndarray class with additional methods and attributes that are specific to neutron backscattering experiments.

It can handle various operations such as addition and subtraction of sample data or numpy array, scaling by a scalar or an array, indexing, broadcasting, reshaping, binning, sliding average or data cleaning.

Parameters

- **input_arr** (*np.ndarray, list, tuple or scalar*) – Input array corresponding to sample scattering data.
- **kwargs** (*dict (optional)*) – Additional keyword arguments either for `np.asarray()` or for sample metadata. The metadata are:
  - **filename**, the name of the file used to extract the data.
  - **errors**, the errors associated with scattering data.
  - **energies**, the energy transfers associated with the data.
  - **time**, the experimental time.
  - **wavelength**, the wavelength of the incoming neutrons.
  - **name**, the name for the sample.
  - **temperature**, the temperature(s) used experimentally.

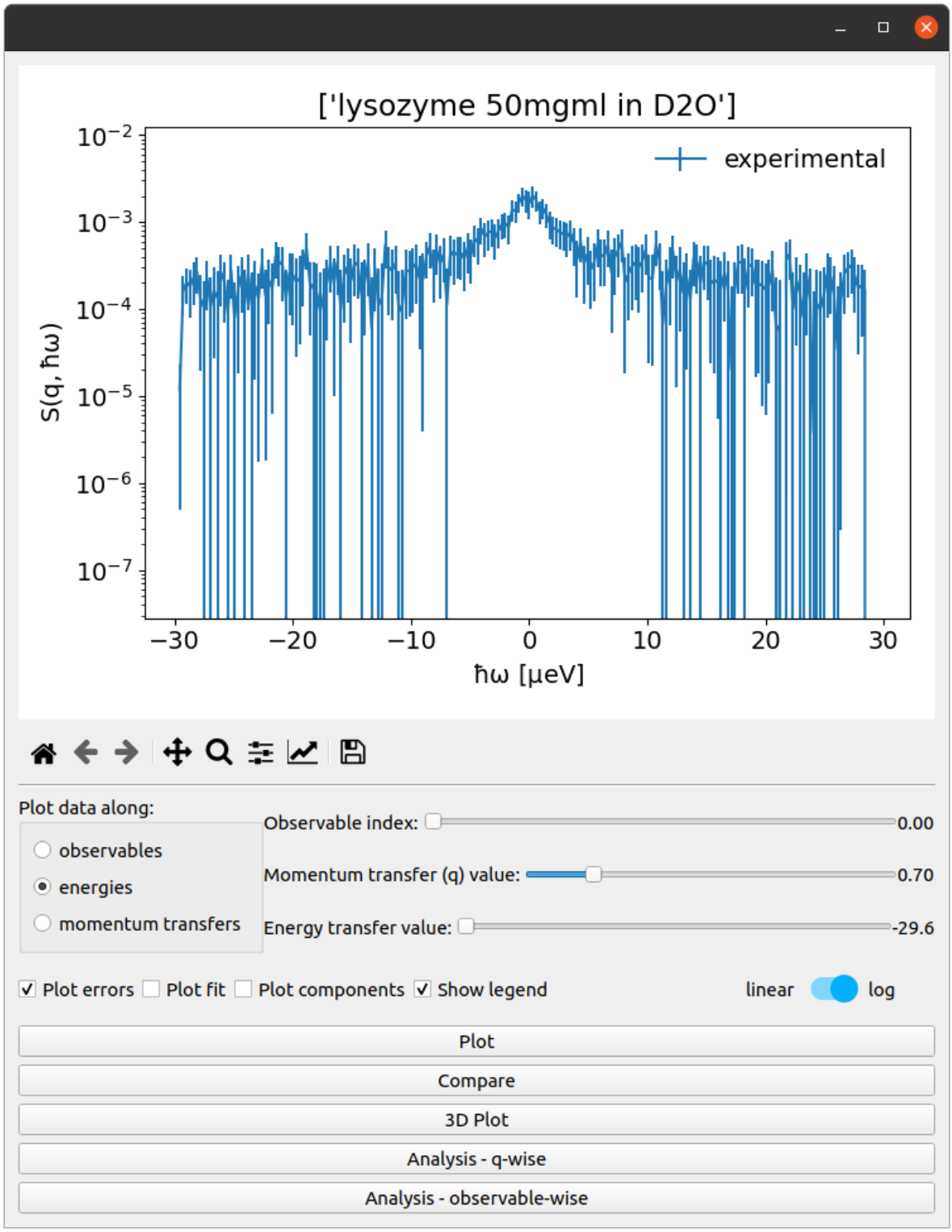

Fig. 1: The experimental data are plotted alone with their errors for the selected observable and momentum transfer q value.

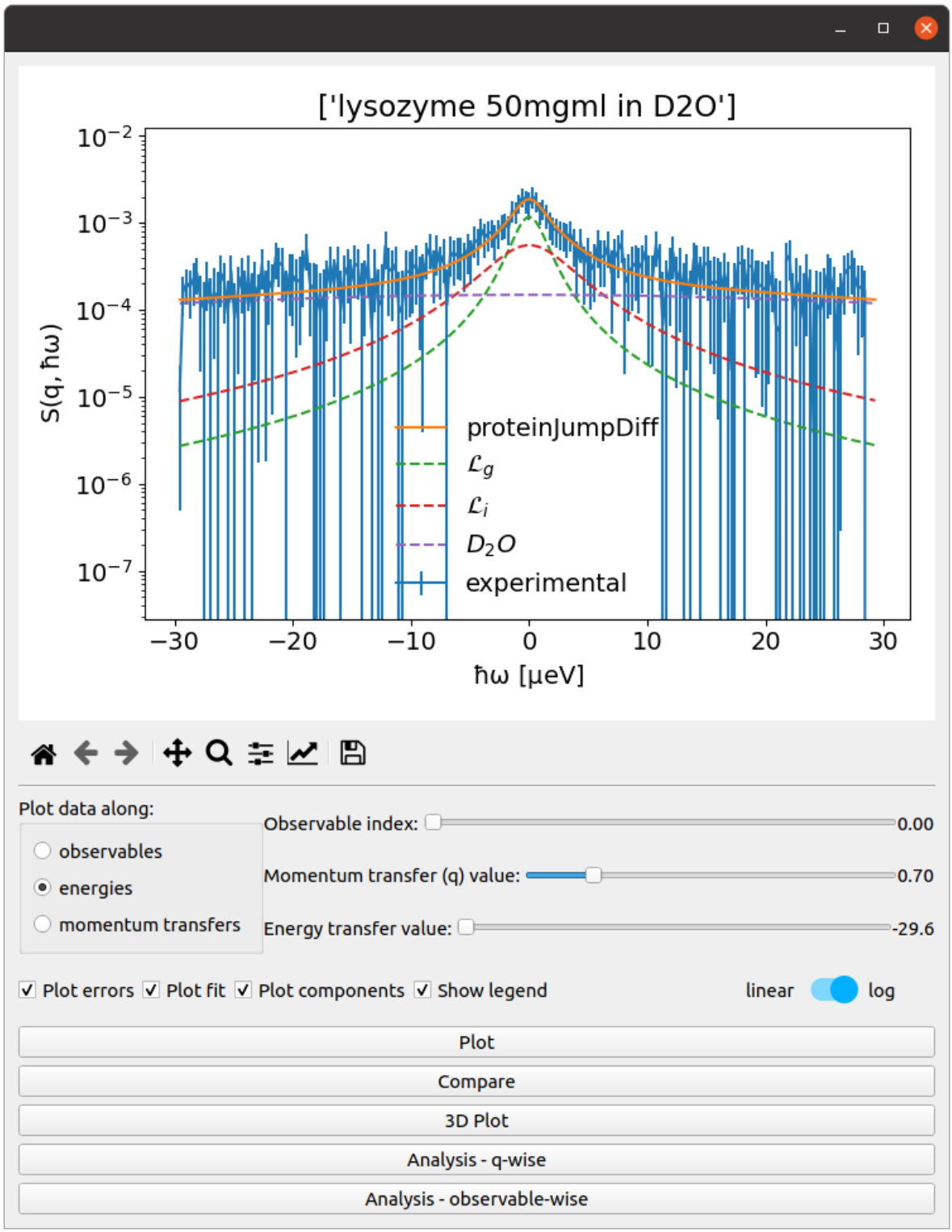

Fig. 2: Here, the fitted model and its components are added by clicking on the associated checkboxes.

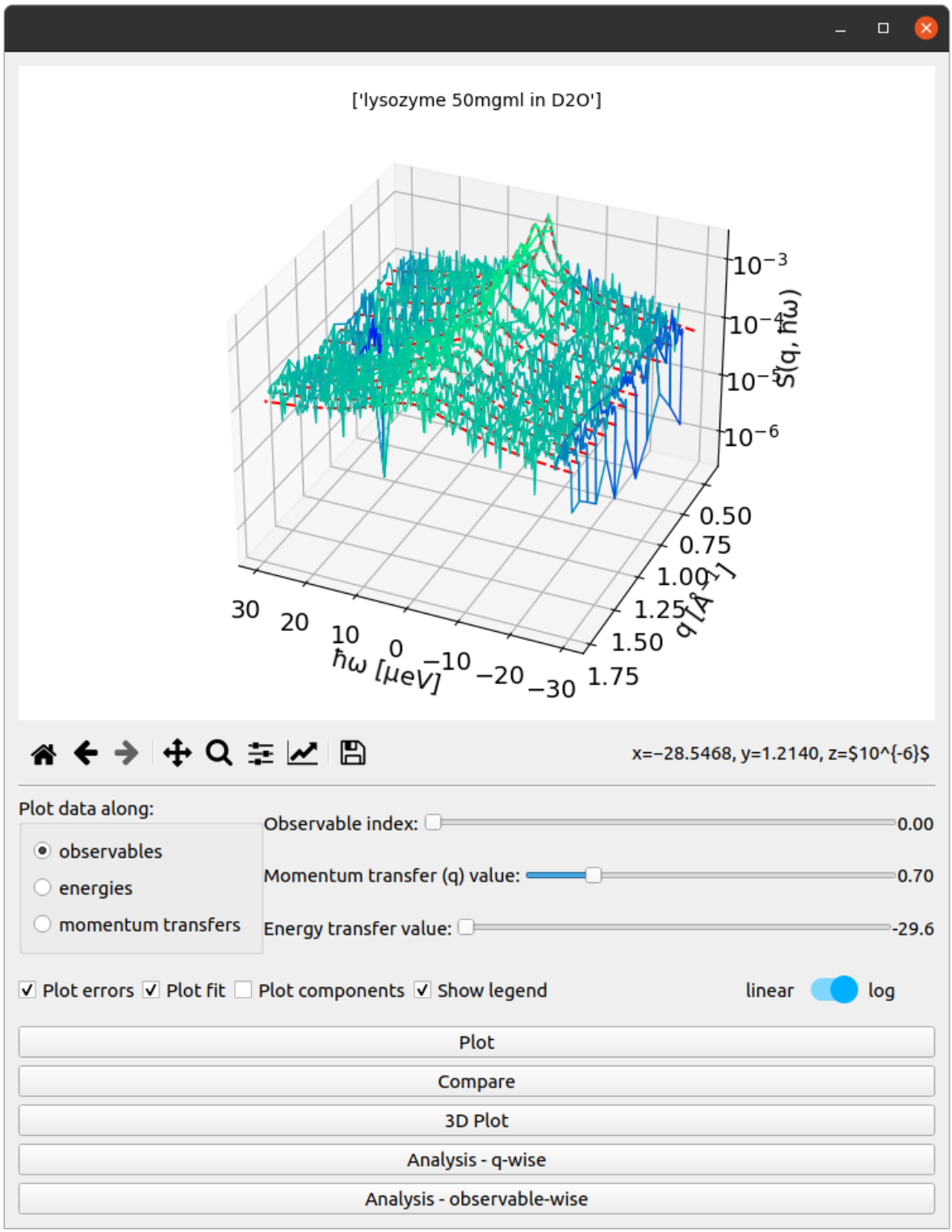

Fig. 3: An 3D view of all spectra is available by clicking on the '3D plot' button.

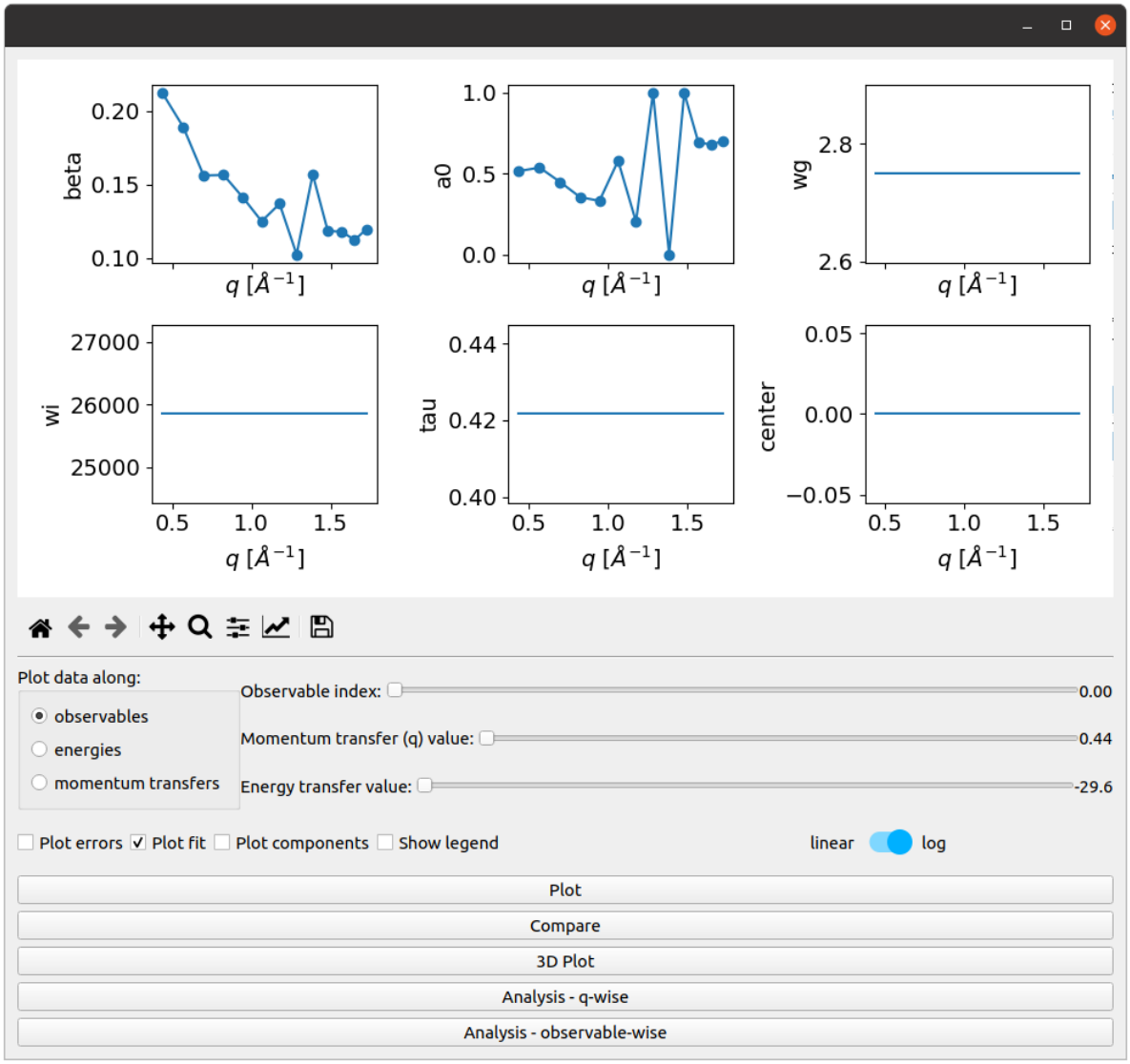

Fig. 4: The optimized parameters can be plotted by clicking on the 'Analysis' button. The global parameters (which are unique for all q-values) are represented by a single horizontal line.

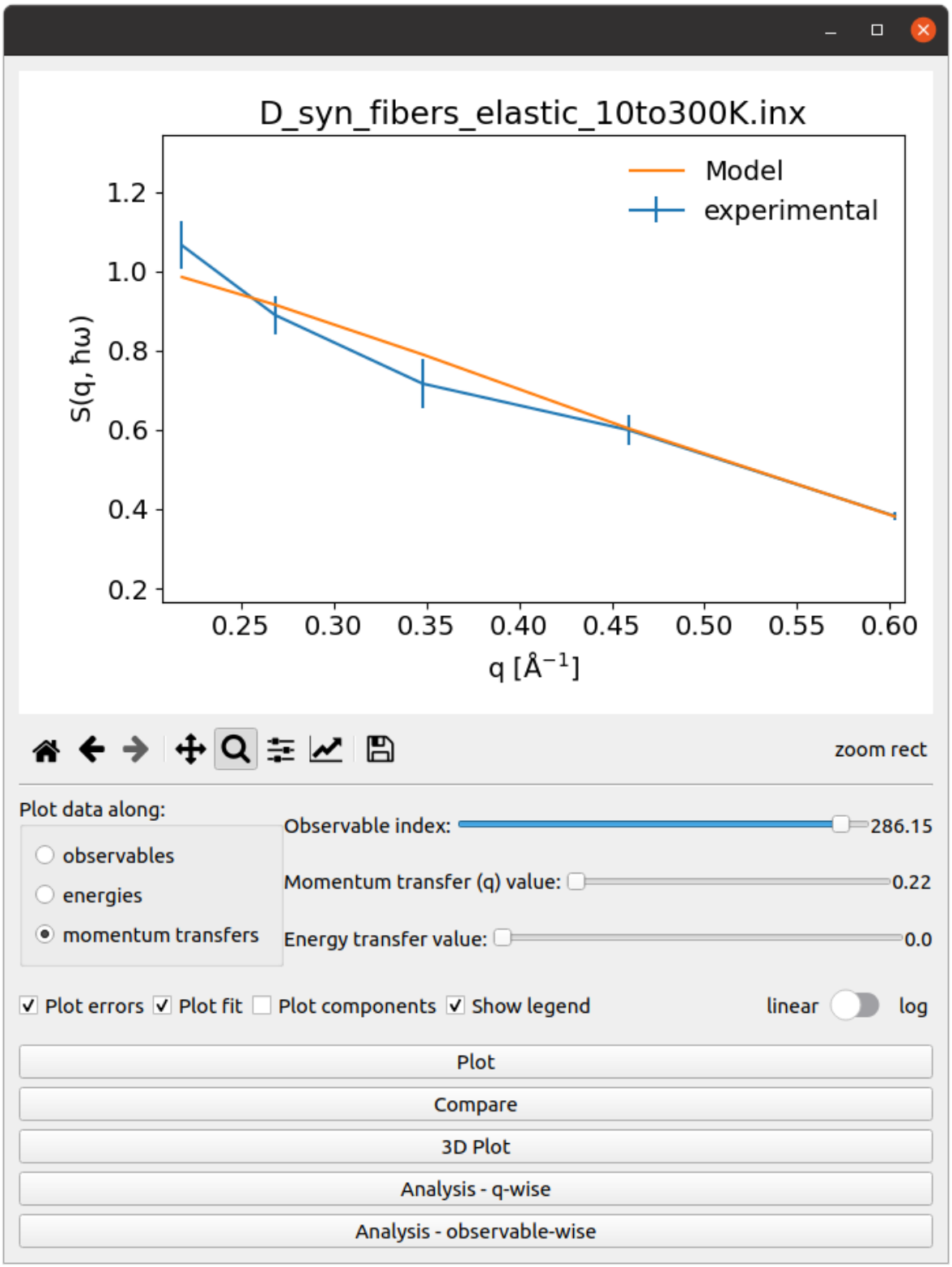

Fig. 5: The data are plotted along the momentum-transfer q-values. The fitted model, which is used to extract the mean-squared displacement is added.

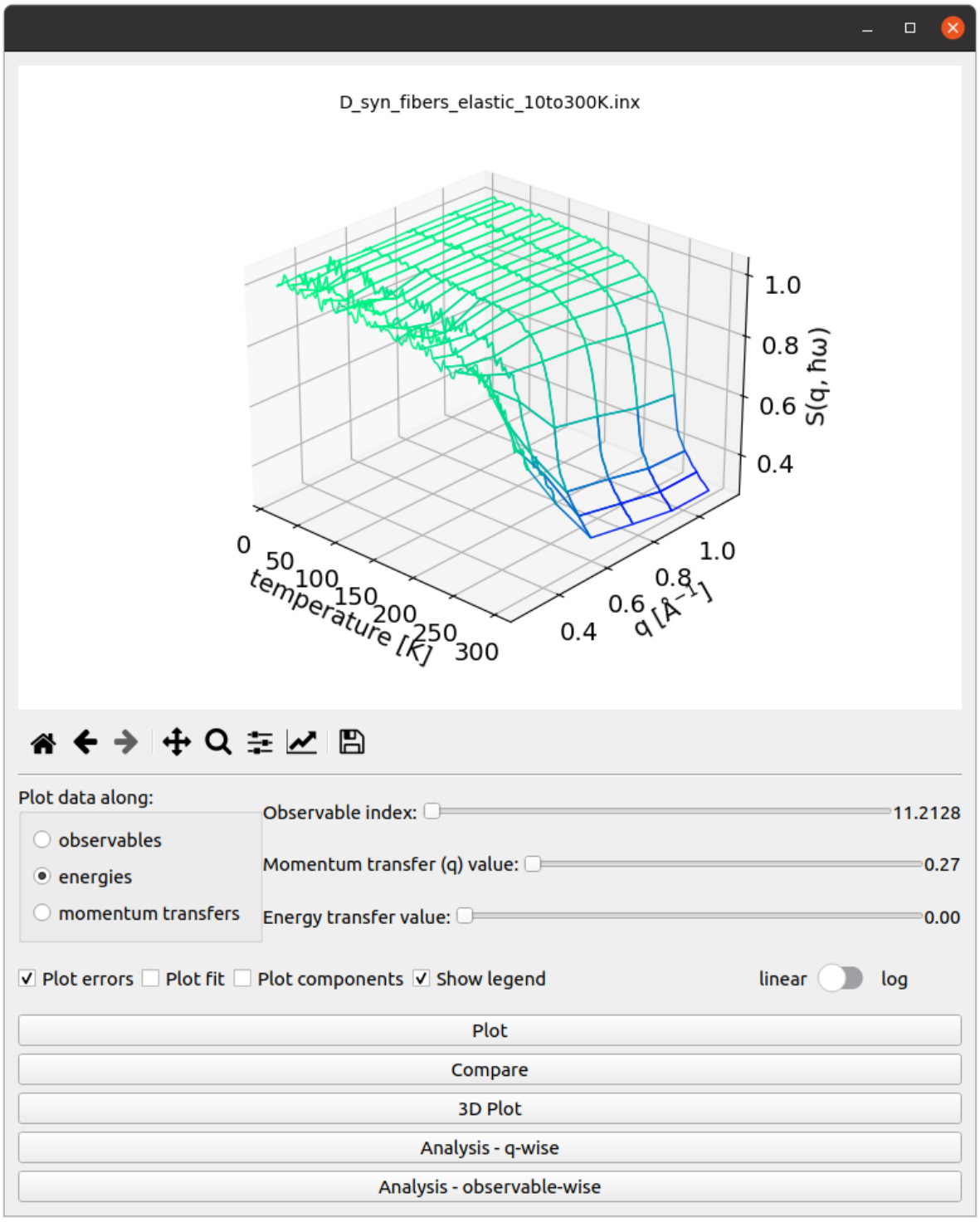

Fig. 6: The whole dataset can be plotted using the '3D plot' button.

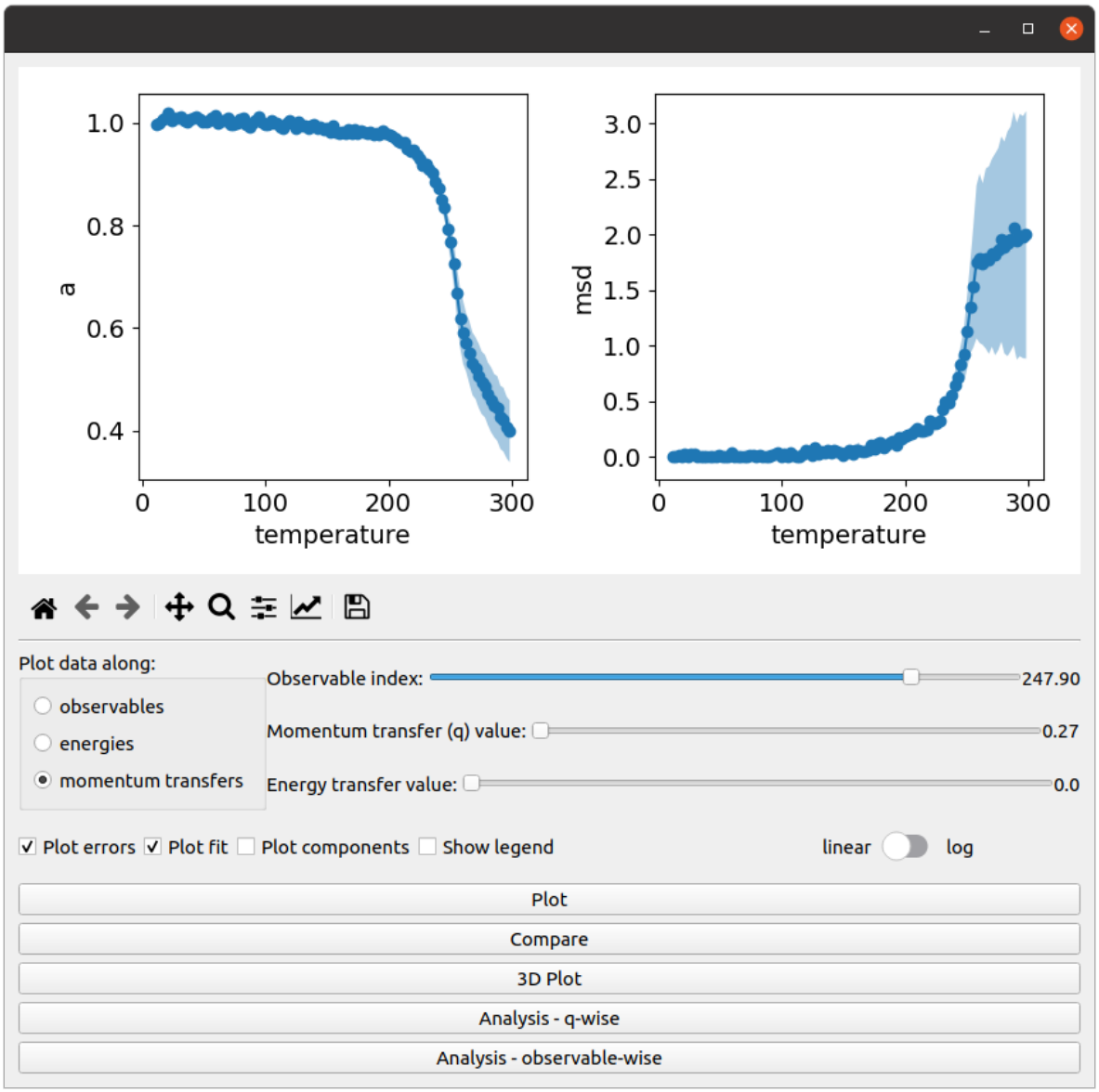

Fig. 7: The optimized parameters can be plotted along different axis (observable, energy, q-values). Here, the uncertainty on the parameters is represented by the blue shaded area around the curve.

- **concentration**, the concentration of the sample.
- **pressure**, the pressure used experimentally.
- **buffer**, a description of the buffer used experimentally.
- **q**, the values for the momentum transfer q.
- **beamline**, the name of the beamline used.
- **observable_name**, the name of the observable variable.

Note: The **errors** metadata is special as it is updated for various operations that are performed on the data array such as indexing or for the use of universal functions. For instance, indexing of the data will be performed on **errors** as well if its shape is the same as for the data. Also, addition, subtraction and other universal functions will lead to automatic error propagation. Some other metadata might change as well, like **q**, but only for the use of methods specific of the *Sample* class and not for methods inherited from numpy.

Examples

A sample can be created using the following:

```
>>> s1 = Sample(
...     np.arange(5),
...     dtype='float32',
...     errors=np.array([0.1, 0.2, 0.12, 0.14, 0.15])
... )
```

```
>>> buffer = Sample(
...     [0., 0.2, 0.4, 0.3, 0.1],
...     dtype='float32',
...     errors=np.array([0.1, 0.2, 0.05, 0.1, 0.2])
... )
```

where *my_data*, *my_errors* and *q_values* are numpy arrays. A buffer subtraction can be performed using:

```
>>> s1 = s1 - buffer
Sample([0. , 0.80000001, 1.60000002, 2.70000005, 3.9000001], dtype=float32)
```

where *buffer1* is another instance of *Sample*. The error propagation is automatically performed and the other attributes are taken from the first operand (here s1). Other operations such as scaling can be performed using:

```
>>> s1 = 0.8 * s1
Sample([0. , 0.80000001, 1.60000002, 2.4000001, 3.20000005], dtype=float32)
```

You can transform another *Sample* instance into a column vector and look how broadcasting and error propagation work:

```
>>> s2 = Sample(
...     np.arange(5, 10),
...     dtype='float32',
...     errors=np.array([0.1, 0.3, 0.05, 0.1, 0.2])
... )
>>> s2 = s2[:, np.newaxis]
>>> res = s1 * s2
>>> res.errors
```

(continues on next page)

(continued from previous page)

```
array([[0.5       , 1.00498756, 0.63245553, 0.76157731, 0.85      ],
       [0.6       , 1.23693169, 0.93722996, 1.23109707, 1.5       ],
       [0.7       , 1.40089257, 0.84593144, 0.99141313, 1.06887792],
       [0.8       , 1.60312195, 0.98061205, 1.15948264, 1.26491106],
       [0.9       , 1.81107703, 1.1516944 , 1.3955644 , 1.56923548]])
```

T

Override the corresponding NumPy function to process axes too.

absorptionCorrection(*ec*, *canType='tube'*, *canScaling=0.9*, *neutron_wavelength=6.27*, *absco_kwargs=None*, *useModel=True*)

Computes absorption Paalman-Pings coefficients

Can be used for sample in a flat or tubular can and apply corrections to data, for each q-value in *data.qVals* attribute.

Parameters

- **ec** (*Sample*) – The data corresponding to the empty can.
- **canType** (*{'tube', 'slab'}*) – Type of can used, either 'tube' or 'slab'. (default, 'tube')
- **canScaling** (*float*) – Scaling factor for empty can contribution term, set it to 0 to use only correction of sample self-attenuation.
- **neutron_wavelength** (*float*) – Incident neutrons wavelength.
- **absco_kwargs** (*dict*) – Geometry arguments for absco library. from Joachim Wuttke[1].

References

bin(*bin_size*, *axis=-1*)

Bin data with the given bin size along specified axis.

Parameters

- **bin_size** (*int*) – The size of the bin (in number of data points).
- **axis** (*int, optional*) – The axis over which the binning is to be performed. (default, -1 for energies)

Returns **out_arr** – A binned instance of *Sample* with the same metadata except for **errors** and the corresponding axis values, which are binned as well.

Return type *Sample*

discardData(*indices*, *axis=0*)

Discard data at given indices along the given axis.

Parameters

- **indices** (*int, list*) – The indices of the data to be discarded.
- **axis** (*int*) – The index of the axis along which the data are discarded.

fit(*model=None*, *cleanData='replace'*, *res=None*, *ec=None*, *bkgd=None*, *volume_fraction_bkgd=0.95*, ***kwargs*)

Fit the dataset using the *model* attribute.

[1] http://apps.jcns.fz-juelich.de/doku/sc/absco

Parameters

- **model** (`Model` instance) – The model to be used for fitting. If None, will look for a model instance in 'model' attribute of the class instance. If not None, will override the model attribute of the class instance.
- **cleanData** (*{'replace', 'omit'} or anything else for no, optional*) – If set to 'replace' the locations of null or inf values in data are set to *np.inf* in weights prior to fitting. If set to 'omit' the locations of null or inf values in data are removed from data, weights and x prior to fitting. Else, nothing is done.
- **res** (*bool, optional*) – If True, will use the attribute *resData*, fix the parameters, and convolve it with the data using: `model = ConvolvedModel(self, resModel)`
- **ec** (*bool, optional*) – If True, will use the attribute *ECData*, fix the parameters, model by calling: `ECModel = self.ECData.fixedModel` and generate a new model by calling: `model = self.model + ECModel`
- **bkgd** (*bool, optional*) – If True, will use the attribute *D2OData* to obtain the fixed model by calling: `D2OModel = self.D2OData.fixedModel` and generate a new model by calling: `model = self.model + D2OModel`
- **volume_fraction_bkgd** (*float [0, 1]*) – Volume fraction for the D2O in the sample. (default 0.95)
- **kwargs** (*dict, optional*) – Additional keyword arguments to pass to *Model.fit* method. It can override any parameters obtained from the dataset, which are passed to the fit function ('data', 'errors', 'x',...).

fit_best(***kwargs*)
Return the fitted model.

Parameters **kwargs** (*dict*) – Additional keyword arguments to pass to *ModelResult.eval*.

fit_components(***kwargs*)
Return the fitted components.

Parameters **kwargs** (*dict*) – Additional keyword arguments to pass to *ModelResult.eval_components*.

fit_result
Return the full result of the fit, if available.

getFixedOptParams(*obsIdx*)
Return the fixed optimal parameters

The parameters are return for the given observable value at index *obsIdx* or the first entry if there is only one observable.

get_energy_range(*min*, *max*)
Helper function to select a specific energy range.

The function assumes that time values correspond to the first dimension of the data set.

Parameters

- **min** (*int*) – The minimum value for time.
- **max** (*int*) – The maximum value for time.

Returns **out** – A new instance of the class with the selected energy range.

Return type *Sample*

get_observable_range(*min*, *max*)
Helper function to select a specific observable range.

The function assumes that time values correspond to the first dimension of the data set.

Parameters

- **min** (*int*) – The minimum value for the observable.
- **max** (*int*) – The maximum value for the observable.

Returns **out** – A new instance of the class with the selected observable range.

Return type *Sample*

get_q_range(*min*, *max*)
Helper function to select a specific momentum transfer range.

The function assumes that q values correspond to the last dimension of the data set.

Parameters

- **min** (*int*) – The minimum value for the momentum transfer q range.
- **max** (*int*) – The maximum value for the momentum transfer q range.

Returns **out** – A new instance of the class with the selected q range.

Return type *Sample*

model
Return the model instance.

model_best
Return the model with the fitted parameters.

normalize(*ref=None*)
Normalize the data using sample intensities or reference sample.

The integration to get the normalization factor is performed along the energy axis.

Parameters **ref** (*Sample*) – A reference sample that is used for as resolution function.

params
Return the best values and errors from the fit result.

plot(*fig_ax=None*, *cb_ax=None*, *axis=-1*, *xlabel=None*, *ylabel='$\\rm S(q, \\hbar \\omega)$'*, *label=None*, *yscale='log'*, *plot_errors=True*, *plot_legend=True*, *max_lines=15*, *colormap='jet'*)
Helper function for quick plotting.

Parameters

- **fig_ax** (*matplotlib Axis, optional*) – An instance of Axis from *matplotlib* to be used for plotting. (default, None)
- **cb_ax** (*matplotlib Axis, optional*) – An instance of Axis from *matplotlib* to be used for the colorbar if needed. (default, None, for 1D arrays)
- **axis** (*int*) – The axis corresponding abscissa. (default, -1)
- **xlabel** (*str*) – The label for the x-axis. (default None, will be guessed for **axes** attribute)
- **ylabel** (*str*) – The label for the y-axis. (default '$rm S(q, hbar omega)$')
- **label** (*str*) – The label for curve. (default, the *name* attribute of the sample)
- **yscale** (*str*) – The scale of the y-axis. (default, 'log')

- **plot_errors** (*bool*) – If True, plot the error bars for each data point.
- **plot_legend** (*bool*) – If True, add the legend to the plot.
- **max_lines** (*int*) – For 2D data, maximum number of lines to be plotted.
- **colormap** (*str*) – The colormap to be used for 2D data.

plot_3D(*fig_ax=None*, *axis='observable'*, *index=0*, *xlabel=None*, *ylabel=None*, *zlabel='$\\rm S(q, \\hbar \\omega)$'*, *zscale='log'*, *colormap='winter'*)

Helper function for quick plotting.

Parameters

- **fig_ax** (*matplotlib axis*) – An instance of Axis from *matplotlib* to be used for plotting. (default, None)
- **axis** (*{'observable', 'q', 'energies', 'time', 'temperature'}*) – The axis along which the data are plotted. Valid for 3D arrays, has no effect for 2D arrays. (default, 'observable')
- **index** (*int*) – The index on the axis given for plotting. Valid for 3D arrays. For 2D, the whole dataset is plotted.
- **xlabel** (*str*) – The label for the x-axis. (default None, will be guessed for **axes** attribute)
- **ylabel** (*str*) – The label for the y-axis. (default None, will be guessed for **axes** attribute)
- **zlabel** (*str*) – The label for the z-axis. (default '$rm S(q, hbar omega)$')
- **zscale** (*str*) – The scale of the z-axis. (default, 'linear')
- **new_fig** (*bool*) – If true, create a new figure instead of plotting on the existing one.
- **colormap** (*str*) – The colormap to be used. (default, 'winter')

sliding_average(*win_size*, *axis=0*)

Performs a sliding average of data and errors along given axis.

Parameters

- **win_size** (*int*) –
- **axis** (*int, optional*) – The axis over which the average is to be performed. (default, 0)

Returns **out_arr** – An averaged instance of *Sample* with the same metadata except for **errors** and the corresponding axis values, which are processed as well.

Return type *Sample*

squeeze(*axis=None*)

Override the corresponding NumPy function to process axes too.

swapaxes(*axis1*, *axis2*)

Override the corresponding NumPy function to process axes too.

take(*indices*, *axis=None*)

Override the corresponding NumPy function to process axes too.

transpose(**axes*)

Override the corresponding NumPy function to process axes too.

sample.**ensure_fit**(*func*)

Ensures the class has a fitted model.

sample.**implements**(*np_function*)
Register an __array_function__ implementation for DiagonalArray objects.

3.5.2 dataParsers

mantidNexus

mantidNexus.**processNexus**(*dataFile*, *FWS=False*)
This script is meant to be used with IN16B data pre-processed (reduction, (EC correction) and vanadium centering) with Mantid.

It can handle both QENS and fixed-window scans.

Then the result is stored as a namedtuple containing several members (all being numpy arrays).

- **intensities - 3D array of counts values for each frame** (axis 0), q-value (axis 1) and energy channels (axis 2)
- **errors - 3D array of errors values for each frame** (axis 0), q-value (axis 0) and energy channels (axis 2)
- energies - 1D array of energy offsets used
- **temps - 2D array of temperatures, the first dimension** is of size 1 for QENS, and of the same size as the number of energy offsets for FWS. The second dimensions represents the frames
- **times - same structure as for temps but representing** the time
- name - name that is stored in the 'subtitle' entry
- qVals - 1D array of q-values used
- qIdx - same as selQ but storing the indices
- **observable - data for the observable used for data series** ('time' or 'temperature')
- observable_name - name of the observable used for data series
- norm - boolean, whether data were normalized or not

mantidWorkspace

inxConvert

inxConvert.**convert**(*datafile*, *FWS=None*)
This method takes a single dataFile as argument and returns the corresponding dataSet.

Then the result is stored as a namedtuple containing several members (all being numpy arrays).

- **intensities - 3D array of counts values for each frame** (axis 0), q-value (axis 1) and energy channels (axis 2)
- **errors - 3D array of errors values for each frame** (axis 0), q-value (axis 0) and energy channels (axis 2)
- energies - 1D array of energy offsets used
- **temps - 2D array of temperatures, the first dimension** is of size 1 for QENS, and of the same size as the number of energy offsets for FWS. The second dimensions represents the frames
- **times - same structure as for temps but representing** the time

- name - name that is stored in the 'subtitle' entry
- qVals - 1D array of q-values used
- **selQ - same as qVals, used later to define a q-range** for analysis
- qIdx - same as selQ but storing the indices
- **observable - data for the observable used for data series** ('time' or 'temperature')
- observable_name - name of the observable used for data series
- norm - boolean, whether data were normalized or not

IN16B_nexus

Parser for .nxs files from IN16B

class in16b_nexus.**IN16B_nexus**(*scanList*, *observable='time'*)

This class can handle raw data from IN16B at the ILL in the hdf5 format.

Parameters

- **scanList** – a string or a list of files to be read and parsed to extract the data. It can be a path to a folder as well.
- **sumScans** – whether the scans should be summed or not
- **alignPeaks** – if True, will try to align peaks of the monitor with the ones from the PSD data.
- **peakFindWindow** – the size (in number of channels) of the window to find and align the peaks of the monitor to the peaks of the data.
- **detGroup** – detector grouping, i.e. the channels that are summed over along the position-sensitive detector tubes. It can be an integer, then the same number is used for all detectors, where the integer defines a region (middle of the detector +/- detGroup). It can be a list of integers, then each integers of the list should corresponds to a detector. Or it can be a string, defining a path to an xml file as used in Mantid. If set to *no*, no detector gouping is performed and the data represents the signal for each pixel on the detectors. In this case, the observable become the momentum transfer q in the vertical direction.
- **normalize** – whether the data should be normalized to the monitor
- **observable** – the observable that might be changing over scans. It can be *time*, *temperature*
- **offset** – If not None, only the data with energy offset that equals the given value will be imported.

process()

Extract data from the provided files and reduce them using the given parameters.

IN16B_QENS

This module is used for importation of raw data from IN16B instrument.

class in16b_qens_scans_reduction.**IN16B_QENS**(*scanList*, *sumScans=True*, *unmirroring=True*, *vanadiumRef=None*, *refPeaks=None*, *detGroup=None*, *normalize=True*, *strip=25*, *observable='time'*, *slidingSum=None*)

This class can handle raw QENS data from IN16B at the ILL in the hdf5 format.

Parameters

- **scanList** – a string or a list of files to be read and parsed to extract the data. It can be a path to a folder as well.
- **sumScans** – whether the scans should be summed or not.
- **unmirroring** – whether the data should be unmirrored or not.
- **vanadiumRef** – if :arg unmirroring: is True, then the peaks positions are identified using the data provided with this argument. If it is None, then the peaks positions are identified using the data in scanList.
- **refPeaks** – if :arg unmirroring: is True, and :arg vanadiumRef: is False, then the given peak positions are used. If it is None, then the peaks positions are identified using the data in scanList.
- **detGroup** – detector grouping, i.e. the channels that are summed over along the position-sensitive detector tubes. It can be an integer, then the same number is used for all detectors, where the integer defines a region (middle of the detector +/- detGroup). It can be a list of integers, then each integers of the list should corresponds to a detector. Or it can be a string, defining a path to an xml file as used in Mantid. If set to *no*, no detector gouping is performed and the data represents the signal for each pixel on the detectors. In this case, the observable become the momentum transfer q in the vertical direction.
- **normalize** – whether the data should be normalized to the monitor
- **strip** – an integer defining the number of points that are ignored at each extremity of the spectrum.
- **observable** – the observable that might be changing over scans. It can be *time* or *temperature*

getReference()

Process files to obtain reference values for elastic signal.

process()

Extract data from the provided files and reduce them using the given parameters.

IN16B_FWS

Classes

class in16b_fws_scans_reduction.**IN16B_FWS**(*scanList*, *sumScans=False*, *alignPeaks=True*, *detGroup=None*, *normalize=True*, *observable='time'*, *offset=None*)

This class can handle raw E/IFWS data from IN16B at the ILL in the hdf5 format.

Parameters

- **scanList** – a string or a list of files to be read and parsed to extract the data. It can be a path to a folder as well.
- **sumScans** – whether the scans should be summed or not

- **alignPeaks** – if True, will try to align peaks of the monitor with the ones from the PSD data.
- **detGroup** – detector grouping, i.e. the channels that are summed over along the position-sensitive detector tubes. It can be an integer, then the same number is used for all detectors, where the integer defines a region (middle of the detector +/- detGroup). It can be a list of integers, then each integers of the list should corresponds to a detector. Or it can be a string, defining a path to an xml file as used in Mantid. If set to *no*, no detector gouping is performed and the data represents the signal for each pixel on the detectors. In this case, the observable become the momentum transfer q in the vertical direction.
- **normalize** – whether the data should be normalized to the monitor
- **observable** – the observable that might be changing over scans. It can be *time*, *temperature*
- **offset** – If not None, only the data with energy offset that equals the given value will be imported.

process()
Extract data from the provided files and reduce them using the given parameters.

IN16B_BATS

This module is used for importation of raw data from IN16B instrument.

class in16b_bats_scans_reduction.**IN16B_BATS**(*scanList*, *sumScans=True*, *detGroup=None*, *normalize=True*, *strip=0*, *observable='time'*, *tElastic=None*, *monitorCutoff=0.8*, *pulseChopper='C34'*, *slidingSum=None*)
This class can handle raw data from IN16B-BATS at the ILL in the hdf5 format.

Parameters

- **scanList** (*string or list*) – A string or a list of files to be read and parsed to extract the data. It can be a path to a folder as well.
- **sumScans** (*bool*) – Whether the scans should be summed or not.
- **detGroup** (*string, int*) – Detector grouping, i.e. the channels that are summed over along the position-sensitive detector tubes. It can be an integer, then the same number is used for all detectors, where the integer defines a region (middle of the detector +/- detGroup). It can be a list of integers, then each integers of the list should corresponds to a detector. Or it can be a string, defining a path to an xml file as used in Mantid. If set to *no*, no detector gouping is performed and the data represents the signal for each pixel on the detectors. In this case, the observable become the momentum transfer q in the vertical direction.
- **normalize** – Whether the data should be normalized to the monitor
- **strip** – An integer defining the number of points that are ignored at each extremity of the spectrum.
- **observable** – The observable that might be changing over scans. It can be *time* or *temperature*.
- **tElastic** (*int, float*) – Time for the elastic peak. Optional, if None, will be guessed from peak fitting.
- **monitorCutoff** – Cutoff with respect to monitor maximum to discard data.

- **pulseChopper** (*{'C12', 'C34'}*) – Chopper pair that is used to define the pulse.

getReference()
Process files to obtain reference values for elastic signal.

process(*center=None*, *peaks=None*, *monPeaks=None*)
Extract data from the provided files and reduce them using the given parameters.

Parameters

- **center** (*int*) – Position of the elastic signal along channels.
- **peaks** (*2D array*) – Reference position of the peaks in dataset. Column vector with integer position for each q value.
- **monPeaks** (*int*) – Reference position of monitor peak signal.

Process functions

Process functions for raw data from position sensitive detectors.

process_functions.**alignGroups**(*data*, *position=None*)
Align the peaks along the z-axis of the detectors.

Parameters

- **data** (*sample.Sample*) – Instance of *sample.Sample*. First axis is assumed to be q-values.
- **position** (*int (optional)*) – Position of the center along the 'channels' axis. (default, None, is determined automatically)

Returns

- **data** (*sample.Sample*) – Instance of *sample.Sample* for which the data maxima were aligned along the z direction.
- **center** (*int*) – The center determined by the algorithm, which can then be used to convert the time-of-flight to energies as it defines the elastic peak.

process_functions.**alignTo**(*data*, *refPos*, *peaks=None*)
Align data peaks to zero energy transfer.

Parameters

- **data** (*sample.Sample*) – Instance of *sample.Sample*.
- **refPos** (*int*) – Reference index on energy/channels axis.
- **peaks** (*np.ndarray (optional)*) – Array of peak positions for each momentum transfer q value. (default, None - will be determined automatically)

process_functions.**alignToZero**(*data*, *peaks=None*)
Align data peaks to the zero of energy transfers.

Parameters

- **data** (*sample.Sample*) – Instance of *sample.Sample*.
- **peaks** (*np.ndarray (optional)*) – Array of peak positions for each momentum transfer q value. (default, None - will be determined automatically)

process_functions.**avgAlongObservable**(*data*)
Average a single dataset along with monitor over the observable.

Parameters

- **data** (*sample.Sample*) – Instance of *sample.Sample*.
- **peaks** (*np.ndarray (optional)*) – Array of peak positions for each momentum transfer q value. (default, None - will be determined automatically)

process_functions.**convertChannelsToEnergy**(*data*, *type*, *refDist=33.388*, *tElastic=None*)

Convert the 'channels' axis to 'energies'

Parameters

- **data** (*sample.Sample*) – Instance of *sample.Sample*.
- **type** (*{'qens', 'fws', 'bats'}*) – Type of dataset that is being processed.
- **refDist** (*float (optional)*) – Reference distance from the pulse chopper used in BATS mode to the sample.
- **tElastic** (*int (optional)*) – Reference value of time-of-flight for the elastic signal.

process_functions.**detGrouping**(*data*, *detGroup=None*)

The function performs a sum along detector tubes using the provided range to be kept.

It makes use of the :arg detGroup: argument.

Parameters

- **data** (*sample.Sample*) – Instance of *sample.Sample*
- **detGroup** (*int, list, file path*) – If the argument is a scalar, it sums over all values that are in the range [center of the tube - detGroup : center of the tube + detGroup].

  If the argument is a list of integers, then each element of the list is assumed to correspond to a range for each corresponding detector in ascending order.

  If the argument is a mantid-related xml file (a python string), the xml_detector_grouping module is then used to parse the xml file and the provided values are used to define the range.

process_functions.**findPeaks**(*data*, *peakFindingMask=None*)

Find the peak for each momentum transfer in data.

The function always return a single peak for each momentum transfer value. Hence, it should be called twice for mirrored data, once for each wing, before unmirroring.

The data are expected to have the momentum transfer q-values in the first dimension, the channels in the second dimension and, for 3D arrays, the momentum transfer in vertical position qz in the third dimension.

Parameters

- **data** (*sample.Sample*) – Instance of *sample.Sample*
- **peakFindingMask** (*np.ndarray (optional)*) – A mask to exclude some points from peak search. (default None, use a small window centered on the 'channel' axis)

process_functions.**findPeaksFWS**(*data*)

Find peaks in FWS data.

For arrays with more than one dimension, the function assumes that the first axis is the momentum tansfer q values ('q') and the second the recorded channels ('channels').

Parameters **data** (*sample.Sample*) – Instance of *sample.Sample*

process_functions.**mergeDataset**(*dataList*, *observable='time'*)

Produce a single dataset from multiple FWS data.

In the case of different sampling for the energy transfers used in FWS data, the function interpolates the smallest arrays to produce a unique numpy array of FWS data.

Parameters

- **data** (list of *sample.Sample*) – list of instances of *sample.Sample*.
- **observable** (*{'time', 'temperature', 'pressure'} (optional)*) – The name of the observable used for series of data. (default, 'time')

process_functions.**normalizeToMonitor**(*data*, *peaks=None*, *monPeaks=None*, *fws=False*)

Normalize the data by divinding by the monitor.

If *peaks* an *monPeaks* are not None, the data are aligned to monitor peaks for each momenum transfer prior to normalization. For FWS data, only the values at peak positions are used.

Parameters

- **data** (*sample.Sample*) – Instance of *sample.Sample*
- **peaks** (*np.ndarray*) – The position of the peak(s) for each momentum transfer. Requires 'monPeaks' as well.
- **monPeaks** (*np.ndarray*) – The position of the peak(s) in monitor. Requires 'peaks' as well.
- **fws** (*bool*) – Whether data are FWS or not.

process_functions.**sumAlongObservable**(*data*)

Sum a single dataset along with monitor over the observable.

Parameters

- **data** (*sample.Sample*) – Instance of *sample.Sample*.
- **peaks** (*np.ndarray (optional)*) – Array of peak positions for each momentum transfer q value. (default, None - will be determined automatically)

process_functions.**unmirror**(*data*, *refPeaks=None*)

Unmirror data using the elastic peak as a reference.

Parameters

- **data** (*sample.Sample*) – Instance of *sample.Sample*.
- **refPeaks** (*np.ndarray (optional)*) – Reference peak positions for the elastic signal. Should have one entry for each momentum transfer q-values in the first dimension and two entries in the second dimension for the peak in the left and right wing, respectively. (default None, will be determined automatically)

3.5.3 models

Model

This module provides a template class to build models that can be used to fit the data.

class model.**Component**(*name*, *func*, *skip_convolve=False*, ***funcArgs*)

Component class to be used with the *Model* class.

Parameters

- **name** (*str*) – Name for the component.
- **func** (*callable*) – The function to be used for this component.

- **skip_convolve** (*bool*) – If True, no convolution is performed for this model. It can be useful for background or normalization terms.
- **funcArgs** (*dict of str, int, float or arrays*) – Values to be passed to the function arguments. This is a dicitonary of argument names mapped to values. The values can be of different types:
  - **int, float or array**, the values are directly passed to the function.
  - **str**, the values are evaluated first. If any word in the string is present in the *Model.params* dictionary keys, the corresponding parameter value is substituted.

Examples

For a *Model* class that has the following key in its *params* attribute: ('amplitude', 'sigma'), the component for a Lorentzian, the width of which depends on a defined vector q, can be created using:

```
>>> def lorentzian(x, scale, width):
...     return scale / np.pi * width / (x**2 + width**2)
>>> myComp = Component(
...     'lor', lorentzian, scale='scale', width='width * q**2')
```

If the Lorentzian width is constant, use:

```
>>> myComp = Component('lor', lorentzian, scale='scale', width=5)
```

Some math functions can be used as well (below the exponential):

```
>>> myComp = Component('lor', lorentzian, scale='np.exp(-q**2 * msd)')
```

eval(*x*, *params*, ***kwargs*)
Evaluate the components using the given parameters.

Parameters

- **params** (Parameters instance) – Parameters to be passed to the component
- **kwargs** (*dict*) – Additional parameters to be passed to the function. Can override params.

processFuncArgs(*params*, ***kwargs*)
Return the evaluated argument for the function using given parameters and keyword arguments.

class model.**FindParamNames**(*key*, *params*)
Helper class to parse strings to evaluation for function arguments in *Component*.

Parameters **params** (Parameters) – An instance of Parameters from which the parameter names are to be found and substituted by the corresponding values.

visit_Name(*node*)
Name visitor.

class model.**Model**(*params*, *name='Model'*, *convolutions=None*, *on_undef_conv='numeric'*)
Model class to be used within nPDyn.

The model is structured in components that can be added together, each component consisting of a name, a callable function and a dictionary of parameters. The parameters of two different components can have the same name such that they can be shared by several components just like for the switching diffusive state model.

Also, the components are separated in two classes, namely *eisfComponents* and *qisfComponents*, in order to provide the possibility to separately extract the elastic and quasi-elastic parts for analysis and plotting.

Parameters

- **params** (`Parameters` instance) – Parameters to be used with the model
- **name** (*str, optional*) – A name for the model.
- **convolutions** (*dict of dict*) – Dictionary that defines the mapping '(function1, function2)' to 'convolutionFunction(function1, function2)'. Analytic convolutions or user defined operators can be defined this way.
- **on_undef_conv** (*{'raise', 'numeric'}*) – Defines the behavior of the class on missing convolution function in the 'convolutions' attribute. The option 'raise' leads to a *KeyError* and the option 'numeric' to a numerical convolution.

addComponent(*comp*, *op='+'*)

Add a component to the model.

Parameters

- **comp** (*Component*) – An instance of *Component* to be added to the model.
- **op** (*{"+", "-", "*", "/"}, optional*) – Operator to be used to combine the new component with the others. If this is the first component, the operator is ignored. (default "+")

bic

Return the bayesian information criterion (BIC).

components

Return the model components.

copy()

Return a copy of the model.

eval(*x*, *params=None*, *convolve=None*, ***kwargs*)

Perform the assembly of the components and call the provided functions with their parameters to compute the model.

Parameters

- **x** (*np.ndarray*) – Values for the x-axis variable
- **params** (*list, np.array, optional*) – Parameters to be passed to the components. Will override existing parameters in *self.params*.
- **convolve** (*Model*) – Another model to be convolved with this one.
- **kwargs** – Additional keyword arguments to be passed to the components. Can override params too.

Returns

- If *returnComponents* is False – The computed model in an array, the dimensions of which depend on *x* and *params* attributes and the function called.
- *else* – A dictionary with key being the component names and the values are the evaluated components.

eval_components(*x*, *params=None*, *convolve=None*, ***kwargs*)

Alias for *eval* with 'returnComponents' set to True.

Perform the computation of the components with the given x-axis values, parameters and convolutions.

Returns

- *A dictionary with key being the component names and the values*
- *are the evaluated components.*

fit (*x*, *data=None*, *weights=None*, *fit_method='curve_fit'*, *fit_kws=None*, *params=None*, ***kwargs*)
Fit the experimental data using the provided arguments.

Parameters

- **x** (*np.ndarray*) – Values for the indenpendent variable.
- **data** (*np.ndarray*) – Experimental data to be fitted.
- **weights** (*np.ndarray, optional*) – Weights associated with the experimental data (the experimental errors).
- **fit_method** (*str, optional*) – The method to be used for fitting. Currently available methods are (from Scipy): - "curve_fit" - "basinhopping" - "differential_evolution" - "shgo" - "minimize"
- **fit_kws** (*dict, optional*) – Additional keywords to be passed to the fit method.
- **params** (Parameters class instance, optional) – Parameters to be used (default None, will use the parameters associated with the model).
- **kwargs** (*dict, optional*) – Additional keywords arguments to give for the evaluation of the model. Can override parameters too.

Returns

Return type A copy of the fitted model instance.

fitResult
Return the full result of the fit.

on_undef_conv
Return the class behavior on undefined convolution.

optParams
Return the result of the fit.

userkws
Return the keywords used for the fit.

Params

The module contains a Parameter class to be used with the Model class.

class params.**Parameters** (*params=None*, ***kwargs*)
A parameter class that handles names, values and bounds.

Parameters

- **params** (*dict of dict*) – A dictionary of parameter names, each being associated with a namedtuple containing the 'value', the 'bounds', the 'fixed', and the 'error' attributes.
- **kwargs** (*keywords*) – Additional keywords argument to set parameter names, values (and possibly bounds and fixed attributes). Can override params too.

listToParams (*pList*, *errList=None*)
Use the given list to convert a list of parameters to a dictionary similar to the current one.

loadParams (*fileName*)
Load parameters from a file in JSON format.

paramList
Accessor for parameter list.

set (*name*, ***kwargs*)
Set a parameter entry with given attributes in 'kwargs'.

Parameters

- **name** (*str*) – Parameter name to be updated.
- **kwargs** (*dict of float, tuple or namedtuple*) – Parameters to be updates with the associated attributes. The call should be of the form:

```
>>> params.set('amplitude', value=1.2, fixed=True)
>>> params.set('width', value=2.3, bounds=(0., np.inf))
```

update (***kwargs*)
Update the parameters.

writeParams (*fileName*)
Write parameters to given file in JSON format.

params.**pTuple** (*value=1*, *bounds=(-inf*, *inf)*, *fixed=False*, *error=0.0*)
Helper function to create a namedtuple with default values.

Presets

This module provides several preset functions that can be used to create model components and fit your data.

presets.**calibratedD2O** (*x*, *q*, *volFraction*, *temp*, *amplitude=1.0*)
Lineshape for D2O where the Lorentzian width was obtained from a measurement on IN6 at the ILL.

Parameters

- **q** (*np.array or list*) – Array of momentum transfer q values
- **volFraction** (*float in [0, 1]*) – Volume fraction of the D2O in the sample.
- **temp** (*float*) – Sample temperature used for the experiment.
- **amplitude** (*float*) – Amplitude of the D2O signal. The parameter to be fitted.

presets.**conv_delta** (*x*, *comp1*, *comp2*, *params1*, *params2*, ***kwargs*)
Convolution between a Lorentzian and a Gaussian

Parameters

- **x** (*np.ndarray*) – x-axis values
- **comp1** (Component) – First component to be used for the convolution.
- **comp2** (Component) – Second component to be used for the convolution.
- **params1** (Parameters) – Parameters for *comp1*.
- **params2** (Parameters) – Parameters for *comp2*.
- **kwargs** (*dict*) – Additional keyword arguments to pass to the method processFuncArgs() for *comp1* and *comp2*.

presets.**conv_gaussian_gaussian** (*x*, *comp1*, *comp2*, *params1*, *params2*, ***kwargs*)
Convolution between two Gaussians

Parameters

- **x** (*np.ndarray*) – x-axis values
- **comp1** (Component) – First component to be used for the convolution.
- **comp2** (Component) – Second component to be used for the convolution.
- **params1** (Parameters) – Parameters for *comp1*.
- **params2** (Parameters) – Parameters for *comp2*.
- **kwargs** (*dict*) – Additional keyword arguments to pass to the method processFuncArgs() for *comp1* and *comp2*.

presets.**conv_linear**(*x*, *comp1*, *comp2*, *params1*, *params2*, ***kwargs*)

Convolution with a linear model.

The linear model is assumed to be used for a background and is thus not convolved. The function returns simply the linear model. If *comp2* is also a linear model, the two models are simply added.

Parameters

- **x** (*np.ndarray*) – x-axis values
- **comp1** (Component) – First component to be used for the convolution.
- **comp2** (Component) – Second component to be used for the convolution.
- **params1** (Parameters) – Parameters for *comp1*.
- **params2** (Parameters) – Parameters for *comp2*.
- **kwargs** (*dict*) – Additional keyword arguments to pass to the method processFuncArgs() for *comp1* and *comp2*.

presets.**conv_lorentzian_gaussian**(*x*, *comp1*, *comp2*, *params1*, *params2*, ***kwargs*)

Convolution between a Lorentzian and a Gaussian

Parameters

- **x** (*np.ndarray*) – x-axis values
- **comp1** (Component) – First component to be used for the convolution.
- **comp2** (Component) – Second component to be used for the convolution.
- **params1** (Parameters) – Parameters for *comp1*.
- **params2** (Parameters) – Parameters for *comp2*.
- **kwargs** (*dict*) – Additional keyword arguments to pass to the method processFuncArgs() for *comp1* and *comp2*.

presets.**conv_lorentzian_lorentzian**(*x*, *comp1*, *comp2*, *params1*, *params2*, ***kwargs*)

Convolution between two Lorentzians

Parameters

- **x** (*np.ndarray*) – x-axis values
- **comp1** (Component) – First component to be used for the convolution.
- **comp2** (Component) – Second component to be used for the convolution.
- **params1** (Parameters) – Parameters for *comp1*.
- **params2** (Parameters) – Parameters for *comp2*.
- **kwargs** (*dict*) – Additional keyword arguments to pass to the method processFuncArgs() for *comp1* and *comp2*.

presets.**conv_lorentzian_rotations**(*x*, *comp1*, *comp2*, *params1*, *params2*, ***kwargs*)
Convolution between a Lorentzian and rotationLorentzians

Parameters

- **x** (*np.ndarray*) – x-axis values
- **comp1** (Component) – First component to be used for the convolution.
- **comp2** (Component) – Second component to be used for the convolution.
- **params1** (Parameters) – Parameters for *comp1*.
- **params2** (Parameters) – Parameters for *comp2*.
- **kwargs** (*dict*) – Additional keyword arguments to pass to the method processFuncArgs() for *comp1* and *comp2*.

presets.**conv_rotations_gaussian**(*x*, *comp1*, *comp2*, *params1*, *params2*, ***kwargs*)
Convolution between a Lorentzian and a Gaussian

Parameters

- **x** (*np.ndarray*) – x-axis values
- **comp1** (Component) – First component to be used for the convolution.
- **comp2** (Component) – Second component to be used for the convolution.
- **params1** (Parameters) – Parameters for *comp1*.
- **params2** (Parameters) – Parameters for *comp2*.
- **kwargs** (*dict*) – Additional keyword arguments to pass to the method processFuncArgs() for *comp1* and *comp2*.

presets.**delta**(*x*, *scale=1*, *center=0*)
A Dirac delta centered on *center*

Parameters

- **x** (*np.ndarray*) – x-axis values, can be an array of any shape
- **scale** (*int, float, np.ndarray*) – scale factor for the normalized function
- **center** (*int, float, np.ndarray*) – position of the Dirac Delta in energy

presets.**gaussian**(*x*, *scale=1*, *width=1*, *center=0*)
A normalized Gaussian function

Parameters

- **x** (*np.ndarray*) – x-axis values, can be an array of any shape
- **scale** (*int, float, np.ndarray*) – scale factor for the normalized function
- **width** (*int, np.ndarray*) – width of the lineshape
- **center** (*int, float, np.ndarray*) – center from the zero-centered lineshape

presets.**generalizedLorentzian**(*x*, *scale=1*, *alpha=1*, *tau=1*, *center=0*)
A generalized Lorentzian function.

This is the Fourier transform of the Mittag-Leffler function. See[1].

[1] https://doi.org/10.1063/1.5121703

References

presets.**kww**(*x*, *scale=1*, *beta=2*, *tau=1*, *center=0*)
The Fourier transform of the stretched exponential function.

Parameters

- **x** (*np.ndarray*) – Values for the x-axis, can be an array of any shape
- **scale** (*int, float, np.ndarray*) – Scale factor for the normalized function
- **beta** (*int, float*) – Value for power of the exponential
- **tau** (*int, float, np.ndarray*) – Characteristic relaxation time.
- **center** (*int, float, np.ndarray*) – Center from the zero-centered lineshape

presets.**linear**(*x*, *a=0.0*, *b=1.0*)
A linear model of the form $ax + b$

presets.**lorentzian**(*x*, *scale=1*, *width=1*, *center=0*)
A normalized Lorentzian function.

Parameters

- **x** (*np.ndarray*) – x-axis values, can be an array of any shape
- **scale** (*int, float, np.ndarray*) – scale factor for the normalized function
- **width** (*int, np.ndarray*) – width of the lineshape
- **center** (*int, float, np.ndarray*) – center from the zero-centered lineshape

presets.**rotations**(*x*, *q*, *scale=1*, *width=1*, *center=0*)
A sum of normalized Lorentzian functions for rotations.

Parameters

- **x** (*np.ndarray*) – x-axis values, can be an array of any shape
- **q** (*np.ndarray*) – Values for the momentum transfers q
- **scale** (*int, float, np.ndarray*) – scale factor for the normalized function
- **width** (*int, np.ndarray*) – width of the lineshape
- **center** (*int, float, np.ndarray*) – center from the zero-centered lineshape

presets.**voigt**(*x*, *scale=1*, *sigma=1*, *gamma=1*, *center=0*)
A normalized Voigt profile.

Parameters

- **x** (*np.ndarray*) – Values for the x-axis, can be an array of any shape
- **scale** (*int, float, np.ndarray*) – Scale factor for the normalized function
- **sigma** (*int, float, np.ndarray*) – Line width of the Gaussian component.
- **gamma** (*int, float, np.ndarray*) – Line width of the Lorentzian component.
- **center** (*int, float, np.ndarray*) – Center from the zero-centered lineshape

Builtins

This module provides several built-in models for incoherent neutron scattering data fitting.

These functions generate a `Model` class instance.

nPDyn.models.builtins.**modelCalibratedD2O**(*q*, *name='D_2O'*, *volFraction=1*, *temp=300*, ***kwargs*)

A model for D2O background containing a single Lorentzian.

Parameters

- **q** (*np.ndarray*) – Array of values for momentum transfer q.
- **name** (*str*) – Name for the model
- **kwargs** (*dict*) – Additional arguments to pass to Parameters. Can override default parameter attributes.

nPDyn.models.builtins.**modelD2OBackground**(*q*, *name='D_2O'*, ***kwargs*)

A model for D2O background containing a single Lorentzian.

Parameters

- **q** (*np.ndarray*) – Array of values for momentum transfer q.
- **name** (*str*) – Name for the model
- **kwargs** (*dict*) – Additional arguments to pass to Parameters. Can override default parameter attributes.

nPDyn.models.builtins.**modelGaussBkgd**(*q*, *name='GaussBkgd'*, ***kwargs*)

A model containing a Gaussian with a background term.

Parameters

- **q** (*np.ndarray*) – Array of values for momentum transfer q.
- **name** (*str*) – Name for the model
- **kwargs** (*dict*) – Additional arguments to pass to Parameters. Can override default parameter attributes.

nPDyn.models.builtins.**modelGeneralizedLorentzian**(*q*, *name='GeneralizedLorentzian'*, *qWise=True*, ***kwargs*)

A model containing a delta and a generalized lorentzian.

This model has been described elsewhere[1].

Parameters

- **q** (*np.ndarray*) – Array of values for momentum transfer q.
- **name** (*str*) – Name for the model
- **kwargs** (*dict*) – Additional arguments to pass to Parameters. Can override default parameter attributes.

References

nPDyn.models.builtins.**modelLorentzianSum**(*q*, *name='LorentzianSum'*, *nLor=2*, *qWise=True*, ***kwargs*)

A model containing a delta and a sum of Lorentzians.

[1] https://doi.org/10.1063/1.5121703

Parameters

- **q** (*np.ndarray*) – Array of values for momentum transfer q.
- **name** (*str*) – Name for the model
- **nLor** (*2*) – Number of Lorentzian to be used.
- **qWise** (*bool*) – If True, no q dependence is imposed on the parameters and the each spectrum is fitted independently.
- **kwargs** (*dict*) – Additional arguments to pass to Parameters. Can override default parameter attributes.

nPDyn.models.builtins.**modelPVoigt**(*q*, *name='PVoigt'*, ***kwargs*)

A model containing a pseudo-Voigt profile.

Parameters

- **q** (*np.ndarray*) – Array of values for momentum transfer q.
- **name** (*str*) – Name for the model
- **kwargs** (*dict*) – Additional arguments to pass to Parameters. Can override default parameter attributes.

nPDyn.models.builtins.**modelPVoigtBkgd**(*q*, *name='PVoigtBkgd'*, ***kwargs*)

A model containing a pseudo-Voigt profile with a background term.

Parameters

- **q** (*np.ndarray*) – Array of values for momentum transfer q.
- **name** (*str*) – Name for the model
- **kwargs** (*dict*) – Additional arguments to pass to Parameters. Can override default parameter attributes.

nPDyn.models.builtins.**modelProteinJumpDiff**(*q*, *name='proteinJumpDiff'*, *qWise=False*, ***kwargs*)

A model for protein in liquid state.

The model contains a Lorentzian of Fickian-type diffusion accounting for center-of-mass motions, a Lorentzian of width that obeys the jump diffusion model[2] accounting for internal dynamics.

Parameters

- **q** (*np.ndarray*) – Array of values for momentum transfer q.
- **name** (*str*) – Name for the model
- **qWise** (*bool*) – If True, no q dependence is imposed on the parameters and the each spectrum is fitted independently.
- **kwargs** (*dict*) – Additional arguments to pass to Parameters. Can override default parameter attributes.

References

nPDyn.models.builtins.**modelTwoStatesSwitchDiff**(*q*, *name='TwoStatesSwitch'*, ***kwargs*)

A model for protein in liquid state.

This model implements the two states switching diffusion model for nPDyn[3].

[2] https://doi.org/10.1103/PhysRev.119.863
[3] https://doi.org/10.1103/PhysRev.119.863

Parameters

- **q** (*np.ndarray*) – Array of values for momentum transfer q.
- **name** (*str*) – Name for the model
- **kwargs** (*dict*) – Additional arguments to pass to Parameters. Can override default parameter attributes.

References

nPDyn.models.builtins.**modelWater**(*q*, *name='waterDynamics'*, ***kwargs*)

A model containing a delta, a Lorentzian for translational motions, a Lorentzian for rotational motions, and a background term.

Parameters

- **q** (*np.ndarray*) – Array of values for momentum transfer q.
- **name** (*str*) – Name for the model
- **kwargs** (*dict*) – Additional arguments to pass to Parameters. Can override default parameter attributes.

3.5.4 lmfit

ConvolvedModel

Can be used to perform analytic convolutions between models.

class convolvedModel.**ConvolvedModel**(*left*, *right*, *on_undefined_conv='numeric'*, *convMap=None*, ***kws*)

Combine two models (*left* and *right*) with the provided analytic convolution function(s).

Parameters

- **left** (Model or CompositeModel) – Left-hand model.
- **right** (Model or CompositeModel) – Right-hand model.
- **on_undefined_conv** (*{'numeric', 'raise'}, optional*) – Determine the behavior when a pair of model has no analytic convolution associated with it:
  - 'numeric' results in a numerical convolution
  - 'raise' raises a KeyError

  (default 'numeric')
- **convMap** (*mapping, optional*) – Dictionary of dictionaries to map the convolution function to a pair of model. A default convMap is already present in the class but can be overridden by this argument.
- ****kws** (*optional*) – Additional keywords are passed to *Model* when creating this new model.

Notes

The two models must use the same independent variables. Only the parameters from left and right are used and exposed. The parameters of the convolution function are not exposed outside the class. They are only used internally and determined inside the convolution function by the combination of the parameters and keywords provided for left and right.

The *eval_components()* returns the convoluted components from *left* by default. This behavior can be changed by using *returnComponents="right"* in the keyword arguments passed to the method.

Examples

First create two models to be convolved (here two Lorentzians):

```
>>> l1 = lmfit.Model.LorentzianModel()
>>> l2 = lmfit.Model.LorentzianModel()
```

Define the convolution function using:

```
>>> def myConv(left, right, params, **kwargs):
...     lp = left.make_funcargs(params, **kwargs)
...     rp = right.make_funcargs(params, **kwargs)
...     amplitude = lp['amplitude'] * rp['amplitude']
...     sigma = lp['sigma'] + rp['sigma']
...     center = lp['center'] + rp['center']
...     out = sigma / (np.pi * ((lp['x'] - center)**2 + sigma**2))
...     return out
```

Eventually perform the convolution:

```
>>> convModel = ConvolvedModel(l1, l2)
```

Assign the convolution function *myConv* to the pair of 'lorentzian' using:

```
>>> convModel.convMap = {'lorentzian': {'lorentzian': myConv}}
```

components
Return components for composite model.

eval(*params=None*, ***kwargs*)
Evaluate model function for convolved model.

eval_components(***kwargs*)
Return OrderedDict of name, results for each component.

on_undefined_conv
Return the parameter 'on_undefined_conv'

param_names
Return parameter names for composite model.

Presets

This module provides several function builders that can be used to fit your data.

These functions generate a *Model* class instance from the **lmfit** package[1].

[1] https://lmfit.github.io/lmfit-py/

References

lmfit_presets.**build_2D_model**(*q*, *funcName*, *funcBody*, *defVals=None*, *bounds=None*, *vary=None*, *expr=None*, *paramGlobals=None*, *prefix=''*, *var='x'*)

Builds a 2D *lmfit.Model*.

Parameters

- **q** (*np.array or list*) – momentum transfer q values of scattering.
- **funcName** (*str*) – name of the function to be built.
- **funcBody** (*str*) – formatted string for the function to be used (in 1D). For a gaussian the string *"{a} * np.exp(-(x - {cen})**2 / {width}*{q}**2)"* will lead to a model with parameters of root names 'a', 'cen' and 'width'. If these parameters are not in argument *paramGlobals*, the parameter names will be 'a_1', 'a_2', 'a_3', ..., 'a_n', where n is the length of the array *q*.
- **defVals** (*dict, optional*) – dictionary of default values for the parameters of the form {'a': 1., 'cen': 0.05, 'width': 2}. If None, set to 1.0 for all parameters.
- **bounds** (*dict, optional*) – dictionary of bounds for the parameters of the form {'a': (0., np.inf}, 'cen': (-10, 10)}. If None, set to (-np.inf, np.inf) for all parameters.
- **vary** (*dict, optional*) – dictionary of parameter hint 'vary' for the parameters of the form {'a': False, 'cen': True}. If None, set to True for all parameters.
- **expr** (*dict, optional*) – dictionary of parameter hint 'expr' for the parameters of the form {'a': 'width / sqrt(2)'}. If None, set to None for all parameters.
- **paramGlobals** (*list, optional*) – defines which parameters should be considered as 'global', that is, a parameter that is fixed for all momentum transfer q values. If set to ['width'], then the resulting model will have parameters of the form ('a_1', ..., 'a_n', 'cen_1', ..., 'cen_n', 'width'), where n is the length of the parameter *q*.
- **prefix** (*str, optional*) – prefix to be given to the model name
- **var** (*str, optional*) – name of the primary independent variable (default 'x')

lmfit_presets.**calibratedD2O**(*q*, *temp=300*, ***kwargs*)

Lineshape for D2O where the Lorentzian width was obtained from a measurement on IN6 at the ILL.

Parameters

- **q** (*np.array or list*) – Array of momentum transfer q values
- **temp** (*float*) – Sample temperature used for the experiment.
- **kwargs** (*dict, optional*) – Additional keywords to pass to *build_2D_model()*.

Notes

The parameter root names are:

- amplitude

lmfit_presets.**delta**(*q*, ***kwargs*)

Normalized Dirac delta.

where the shape of the output array depends on the shape of the independent variable *q*.

Parameters

- **q** (*np.array or list*) – array of momentum transfer q values
- **kwargs** (*dict, optional*) – additional keywords to pass to *build_2D_model()*.

Notes

The parameter root names are:

- amplitude
- center

lmfit_presets.**gaussian**(*q*, *qwise=True*, ***kwargs*)
Normalized Gaussian lineshape.

$$\mathrm{G(x, q; a, c, \sigma)} = \frac{\mathrm{a}}{\sqrt{(\pi\sigma}}\mathrm{e}^{-(\mathrm{x-c})^2/\sigma}$$

where the shape of the output array depends on the shape of the independent variable q and σ can have an explicit dependence on q as $\sigma q * *2$.

Parameters

- **q** (*np.array or list*) – array of momentum transfer q values
- **qwise** (*bool, optional*) – whether the width (sigma) has explicit dependence on q (default False)
- **kwargs** (*dict, optional*) – additional keywords to pass to *build_2D_model()*.

Notes

The parameter root names are:

- amplitude
- center
- sigma

lmfit_presets.**getDelta**(*x*, *amplitude*, *center*)
Helper function for the Dirac delta model.

lmfit_presets.**hline**(*q*, ***kwargs*)
A horizontal line.

lmfit_presets.**jump_diff**(*q*, *qwise=False*, ***kwargs*)
Normalized Lorentzian with jump-diffusion model.

The shape of the output array depends on the shape of the independent variable q.

Parameters

- **q** (*np.array or list*) – array of momentum transfer q values
- **qwise** (*bool, optional*) – whether the width (sigma) has explicit dependence on q (default False)
- **kwargs** (*dict, optional*) – additional keywords to pass to *build_2D_model()*.

Notes

The parameter root names are:

- amplitude
- center
- sigma
- tau

References

For more information see: http://doi.org/10.1103/PhysRev.119.863

lmfit_presets.**kww**(*q*, ***kwargs*)
Fourier transform of the Kohlrausch-William-Watts (KWW) function.

The shape of the output array depends on the shape of the independent variable q.

Parameters

- **q** (*np.array or list*) – array of momentum transfer q values
- **kwargs** (*dict, optional*) – additional keywords to pass to *build_2D_model()*.

Notes

The parameter root names are:

- amplitude
- tau
- beta

References

For more information, see: https://en.wikipedia.org/wiki/Stretched_exponential_function

lmfit_presets.**linear**(*q*, ***kwargs*)
Linear model that can be used for background.

The model reads: $a * x + b$

Notes

Two parameters:

- a
- b

lmfit_presets.**lorentzian**(*q*, *qwise=False*, ***kwargs*)
Normalized Lorentzian lineshape.

$$\mathcal{L}(\mathrm{x}, \mathrm{q}; \mathrm{a}, \mathrm{c}, \sigma) = \frac{\mathrm{a}}{\pi} \frac{\sigma}{(\mathrm{x} - \mathrm{c})^2 + \sigma^2}$$

where the shape of the output array depends on the shape of the independent variable q and σ can have an explicit dependence on q as $\sigma q**2$.

Parameters

- **q** (*np.array or list*) – array of momentum transfer q values
- **qwise** (*bool, optional*) – whether the width (sigma) has explicit dependence on q (default False)
- **kwargs** (*dict, optional*) – additional keywords to pass to *build_2D_model()*.

Notes

The parameter root names are:

- amplitude
- center
- sigma

lmfit_presets.**protein_liquid**(*q*, *qWise=False*, ***kwargs*)

Model for protein in solution and jump diffusion for internal dynamics.

Parameters

- **q** (*np.array or list*) – Array of momentum transfer q-values to be used.
- **qWise** (*bool*) – Whether the Lorentzian width are independent for each momentum transfer q or not (explicit q-dependence of the form 'width * q**2').
- **kwargs** (*dict*) – Additional keyword arguments to pass to *build_2D_model()*

Notes

The parameter root names are:

- beta
- amplitude
- center
- sigma_g
- sigma_i
- tau

lmfit_presets.**pseudo_voigt**(*q*, ***kwargs*)

Pseudo-Voigt profile.

The shape of the output array depends on the shape of the independent variable q.

Parameters

- **q** (*np.array or list*) – array of momentum transfer q values
- **kwargs** (*dict, optional*) – additional keywords to pass to *build_2D_model()*.

Notes

The parameter root names are:

- amplitude
- fraction
- center
- sigma

lmfit_presets.**rotations**(*q*, *qwise=False*, ***kwargs*)

Normalized Lorentzian accounting for rotational motions in liquids.

$$S_r(q,\omega) = A_r J_0^2(qd)\delta(\omega) + \sum_{l=1}(2l+1)J_l^2(qd)\frac{1}{\pi}\frac{l(l+1)\sigma}{(\omega - center)^2 + (l(l+1)\sigma)^2}$$

The shape of the output array depends on the shape of the independent variable q.

Parameters

- **q** (*np.array or list*) – array of momentum transfer q values
- **qwise** (*bool, optional*) – whether the width (sigma) has explicit dependence on q (default False)
- **kwargs** (*dict, optional*) – additional keywords to pass to *build_2D_model()*.

Notes

The parameter root names are:

- amplitude
- center
- sigma
- bondDist

References

For more information see: http://doi.org/10.1139/p66-108

lmfit_presets.**two_diff_state**(*q*, *qwise=False*, ***kwargs*)

Two state switching diffusion model.

The shape of the output array depends on the shape of the independent variable q.

Parameters

- **q** (*np.array or list*) – array of momentum transfer q values
- **kwargs** (*dict, optional*) – additional keywords to pass to *build_2D_model()*.

Notes

The parameter root names are:

- amplitude
- center
- gamma1
- gamma2
- tau1
- tau2

References

For more information, see: http://doi.org/10.1063/1.4950889 or http://doi.org/10.1039/C4CP04944F

lmfit_presets.**voigt**(*q*, ***kwargs*)
Voigt profile.

The shape of the output array depends on the shape of the independent variable *q*.

Parameters

- **q** (*np.array or list*) – array of momentum transfer q values
- **kwargs** (*dict, optional*) – additional keywords to pass to *build_2D_model()*.

Notes

The parameter root names are:

- amplitude
- center
- sigma
- gamma

Convolutions

Basic analytical convolutions between preset functions.

convolutions.**conv_delta**(*left*, *right*, *params*, ***kwargs*)
Convolution with a Dirac delta.

convolutions.**conv_gaussian_gaussian**(*left*, *right*, *params*, ***kwargs*)
Convolution between two Gaussians.

$$a_1 G_{\sigma_1, center_1} \otimes a_2 G_{\sigma_2, center_2} = a_1 a_2 . G_{\sigma_1 + \sigma 2, center_1 + center_2}$$

convolutions.**conv_gaussian_jumpdiff**(*left*, *right*, *params*, ***kwargs*)
Convolution of a Gaussian and a jump-diffusion Lorentzian.

Results in a Voigt profile as defined in lineshapes.

convolutions.**conv_gaussian_lorentzian**(*left*, *right*, *params*, ***kwargs*)
Convolution of a Gaussian and a Lorentzian.

Results in a Voigt profile as defined in lineshapes.

convolutions.**conv_gaussian_pvoigt**(*left*, *right*, *params*, ***kwargs*)
Convolution between a Gaussian and a pseudo-Voigt profile.

$$a_L G_{\sigma_G, center_G} \otimes a_V . \sqrt{\mathcal{V}}_{\sigma, center, fraction} = a_G a_V \left[fraction \mathcal{V}_{\sigma_G, \sigma, center+center_G} + (1 - fraction) G_{\sigma_g+\sigma_G, center+center_G} \right]$$

where $\sqrt{\mathcal{V}}$ is the pseudo-Voigt, $\mathcal{V}$ is a Voigt profile, and $\sigma_g = \frac{\sigma}{\sqrt{(2log(2))}}$.

convolutions.**conv_gaussian_rotations**(*left*, *right*, *params*, ***kwargs*)
Convolution of a Gaussian and a liquid rotations model.

convolutions.**conv_jumpdiff_pvoigt**(*left*, *right*, *params*, ***kwargs*)
Convolution between the jump diffusion model and a pseudo-Voigt profile.

convolutions.**conv_linear**(*left*, *right*, *params*, ***kwargs*)
Convolution with a linear model.

Simply returns the linear model itself as it is assumed to serve as a background term by default.

convolutions.**conv_lorentzian_lorentzian**(*left*, *right*, *params*, ***kwargs*)
Convolution between two Lorentzians.

$$a_1 \mathcal{L}_{\sigma_1, center_1} \otimes a_2 \mathcal{L}_{sigma_2, center_2} = a_1 a_2 . \mathcal{L}_{\sigma_1+\sigma_2, center_1+center_2}$$

convolutions.**conv_lorentzian_pvoigt**(*left*, *right*, *params*, ***kwargs*)
Convolution between a Lorentzian and a pseudo-Voigt profile.

$$a_L \mathcal{L}_{\sigma_L, center_L} \otimes a_V . p\mathcal{V}_{\sigma, center, fraction} = a_L a_V \left[(1 - fraction) \mathcal{V}_{\sigma_g, \sigma, center+center_L} + fraction \mathcal{L}_{\sigma+\sigma_L, center+center_L} \right]$$

where $p\mathcal{V}$ is the pseudo-Voigt, $\mathcal{V}$ is a Voigt profile, $\sigma_g = \frac{\sigma}{\sqrt{(2log(2))}}$ and $\mathcal{L}$ is a Lorentzian.

convolutions.**conv_rotations_pvoigt**(*left*, *right*, *params*, ***kwargs*)
Convolution between the rotation model and a pseudo-Voigt profile.

convolutions.**getGlobals**(*params*)
Helper function to get the global parameters.

Builtins

This module provides several built-in models for incoherent neutron scattering data fitting.

These functions generate a *Model* class instance from the **lmfit** package[1].

References

class lmfit_builtins.**ModelDeltaLorentzians**(*q*, *nLor=2*, ***kwargs*)
A Dirac delta with a given number of Lorentzians.

Parameters

[1] https://lmfit.github.io/lmfit-py/

- **q** (*np.array or list*) – Array of momentum transfer q-values to be used.
- **nLor** (*int, optional*) – Number of Lorentzians to be included in the model.
- **kwargs** (*dict*) – Additional keyword arguments to pass to `build_2D_model()`

class lmfit_builtins.**ModelGaussBkgd**(*q*, ***kwargs*)
A Gaussian with a background term.

Can be useful for empty can signal.

Parameters

- **q** (*np.array or list*) – Array of momentum transfer q-values to be used.
- **kwargs** (*dict*) – Additional keyword arguments to pass to `build_2D_model()`

class lmfit_builtins.**ModelPVoigtBkgd**(*q*, ***kwargs*)
A pseudo-voigt profile with a background term.

Parameters

- **q** (*np.array or list*) – Array of momentum transfer q-values to be used.
- **kwargs** (*dict*) – Additional keyword arguments to pass to `build_2D_model()`

lmfit_builtins.**guess_from_qens**(*pars*, *pGlobals*, *data*, *x*, *q*, *prefix=None*)
Estimate starting values from 2D peak data and create Parameters.

Notes

The dataset should be of shape (number of q-values, energies), that is, the function should be called for each value of 'observable'.

lmfit_builtins.**update_param_vals**(*pars*, *prefix*, ***kwargs*)
Update parameter values with keyword arguments.

3.5.5 plot

plot

Plotting window for Sample class instances.

class plot.**Plot**(*dataset*)

analysisObsPlot()
Plot the fitted parameters.

analysisQPlot()
Plot the fitted parameters.

compare()
Plot the experimental data on one subplot, with or without fit

get_eRange(*idx=0*)
Return the energy values used in the dataset(s).

This assumes the q-values are the same for all datasets.

get_obsRange(*idx=0*)
Return the observables used in the dataset(s).

This assumes the observables are the same for all datasets.

get_qRange(*idx=0*)
Return the q-values used in the dataset(s).

This assumes the q-values are the same for all datasets.

initChecks()
This methods is used to perform some checks before finishing class initialization.

obsIdx
Return a list of index of the closest observable value to the slider value for each dataset.

plot()
Plot the experimental data, with or without fit

plot3D()
3D plot of the whole dataset.

updateLabels()
Update the labels on the right of the sliders.

updatePlot()
Redraw the current plot based on the selected parameters.

plot.**plot**(**samples*)
This methods plot the sample data in a PyQt5 widget allowing the user to show different types of plots.

The resolution function and other parameters are automatically obtained from the current dataset class instance.

Parameters **samples** (nPDyn.Sample) – Samples to be plotted.

subPlotsFormat

subPlotsFormat.**subplotsFormat**(*caller*, *sharex=False*, *sharey=False*, *projection=None*, *params=False*, *FWS=False*)
This method is used to try to determine the best number of rows and columns for plotting. Depending on the size of the fileIdxList, the plot will have a maximum of subplots per row, typically around 4-5 and the required number of rows. :arg sharex: matplotlib's parameter for x-axis sharing :arg sharey: matplotlib's parameter for y-axis sharing :arg projection: projection type for subplots (None, '3d',...)

(optional, default None)

Parameters

- **params** – if True, use size of paramsNames instead of fileIdxList
- **FWS** – if True, use numbers of energy offsets in fixed-window scans instead

Returns axis list from figure.subplots method of matplotlib

subPlotsFormat.**subplotsFormatWithColorBar**(*caller*, *sharex=False*, *sharey=False*, *projection=None*, *params=False*)
This method is used to try to determine the best number of rows and columns for plotting. Depending on the size of the fileIdxList, the plot will have a maximum of subplots per row, typically around 4-5 and the required number of rows. Axes are added to plot colorbars as well, so that the number of columns will be twice the number required initially by the data. :arg sharex: matplotlib's parameter for x-axis sharing :arg sharey: matplotlib's parameter for y-axis sharing :arg projection: projection type for subplots (None, '3d',...)

(optional, default None)

Parameters `params` – if True, use size of paramsNames instead of fileIdxList

Returns axis list from figure.subplots method of matplotlib

3.6 License

GNU GENERAL PUBLIC LICENSE Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for

software and other kinds of works.

The licenses for most software and other practical works are designed

to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not

price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you

these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether

gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps:

(1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains

that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run

modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents.

States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and

modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of

works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this

License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work

in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based

on the Program.

To "propagate" a work means to do anything with it that, without

permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other

parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices"

to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work

for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official

standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other

than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific

operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all

the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users

can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that

same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of

copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not

convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under

the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological

measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid

circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you

receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with

section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey,

and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to

produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

> a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
>
> b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
>
> c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
>
> d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent

works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms

of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

> a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
>
> b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
>
> c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded

from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any

tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods,

procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or

specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a

requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided,

in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this

License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately

under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option

remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you

add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

> a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
>
> b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
>
> c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
>
> d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
>
> e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
>
> f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further

restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you

must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the

form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly

provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your

license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is

reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the

licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or

run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically

receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an

organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the

rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this

License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims

owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free

patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express

agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license,

and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or

arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within

the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting

any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or

otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have

permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of

the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the

Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future

versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different

permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY

APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING

WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided

above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest

possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest

to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

Analysis routines for neutron backscattering data Copyright (C) 2019 Kevin Pounot

> This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

> If the program does terminal interaction, make it output a short

notice like this when it starts in an interactive mode:

> nPDyn Copyright (C) 2019 Kevin Pounot This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’. This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an “about box”.

> You should also get your employer (if you work as a programmer) or school,

if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

> The GNU General Public License does not permit incorporating your program

into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

3.7 Help

A google group is available for any question, discussion on features or comment.

In case of bugs or obvious change to be done in the code use GitHub Issues.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

P

R

S

T

U

V

W