HDF5_BLS

Release v1.0.2

Pierre Bouvet

Dec 05, 2025

FILE FORMAT GUIDE

About HDF5_BLS

The *HDF5_BLS* project is a Python package allowing users to store Brillouin Light Scattering relevant data in a single HDF5 file. The package is designed to integrate in existing Python workflows, to be minimally constrained, and to be as easy to use as possible. The package is meant to answer three main specifications:

- **Simplicity**: Make it easy to store and retrieve data from a single file.
- **Universality**: Allow all modalities to be stored in a single file, while unifying the metadata associated to the data.
- **Unification**: Allow and develop unified data processing tools to be used on BLS data.

A project aiming at unifying and standardizing the storage of Brillouin Light Scattering (BLS) data in a HDF5 files, and to develop unified tools to process these data.

This documentation is intended to help users to understand the project and to use it in their own workflows.

Part I

File format guide

CHAPTER

ONE

THE PROJECT IN A NUTSHELL

1.1 A quick overview

The file format is built to reproduce the file structure of a classical directory, with added benefits:

- storage of datasets in a universally readable format
- storage of all data related to a given experiment in a single file that can be shared and that cannot be used to store executable code
- storage of metadata associated to the data together with the data
- storage of any BLS-related measures together with the BLS data (e.g. fluorescence, absorbance, grayscale images, etc.)
- storage of results of data processing
- storage of custom codes to process the data
- storage of algorithms steps when using the developped data processing tools

As an example, Fig. 1.1 shows a concrete example of file structure corresponding to an experiment (left) and the corresponding HDF5 file (right, displayed using Panoply), used to store Brillouin Light Scattering data.

1.2 File format limitations

The file format is built to store datasets "as is" to help the community integrate the project in their existing workflows. Of course, a normalization of the datasets is needed to make them compatible with the rest of the community. This normalization is done in a latter stage, as it is not always necessary and can be time consuming (e.g. when performing the characterization of an instrument that is not destined to be used outside of the laboratory).

Metadata are also not constrained per default. We however recommend users to store their metadata following the nomenclature proposed by the community. This will ease the eventual normalization of the file when needed.

Codes can be stored as text files in the attributes of the file. This allows you to store virtually everything you might ever use or have used to treat or visualize your data. Normalizing these data processes however, will ask the user to use fixed algorithms. We propose two libraries to help the community to develop and use their own algorithms: - The *HDF5_BLS_analyse* module allows to define new algorithms to analyse the data (to extract a Power Spectral Density and a frequency axis from the data) - The *HDF5_BLS_treat* module allows to define new algorithms to treat the data (to fit the data to a model, extract the results of the fit, etc.)

Of course normalizing entire processing pipelines will be time consuming, we therefore recommend users to first get used to the file format and to store their data in it, and then evolve to store their data in a more normalized fashion (or develop custom algorithms to convert the data to a normalized format).

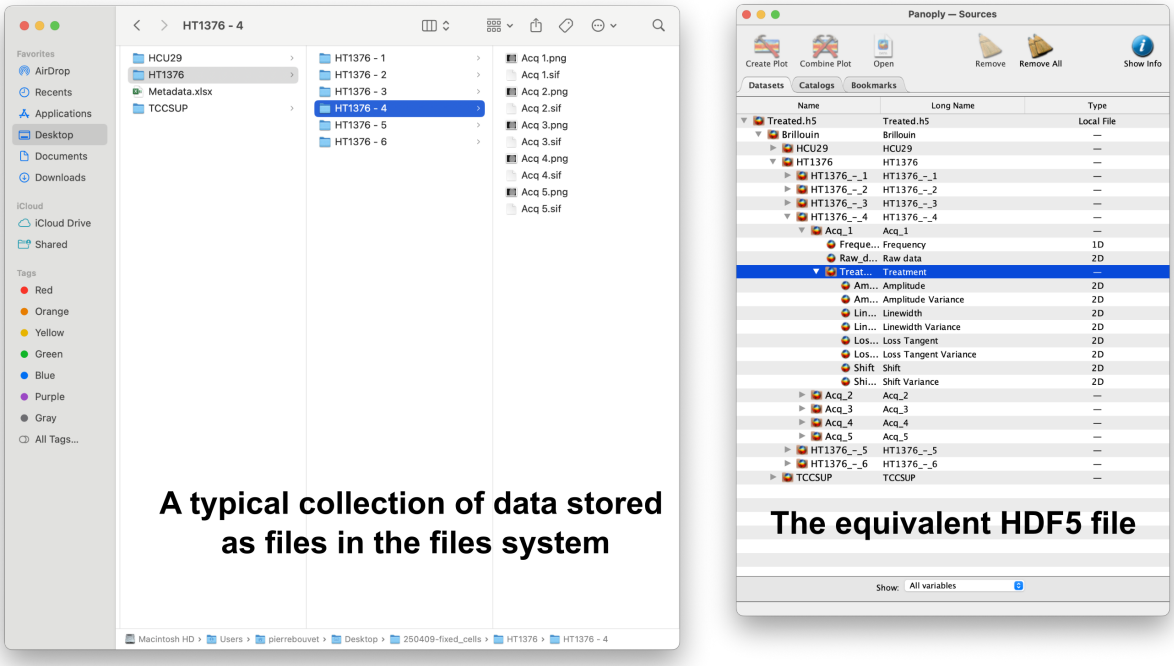


Fig. 1.1: A concrete example of measures stored in a file system (left) and the corresponding HDF5 file structure (right, using Panoply).

1.3 Storing data in the file format

In a general BLS experiment, the user will retrieve a series of values that have been collected. These values are not necessarily of the same physical nature (it can be the number of photons collected, an intensity profile varying with position or time, etc.). These measures are stored in the file format as "raw data".

They however all carry enough information to "see" the Brillouin light scattering (BLS) signal on a spectrum (the Power Spectral Density or PSD). This more valuable information is stored as a "PSD" dataset, and is associated to a "frequency" dataset. The collection of these two datasets constitutes a basis for any measure, and is stored in a dedicated group for the measure.

When these measures correspond to different parameters (a position in a sample, a time point, a given concentration, etc.), we need to store together with these measures, the values of these parameters. This is done with an "abscissa" dataset that is stored together with the raw data or PSD, under the same "measure" group.

From there, we usually retrieve the shift and the linewidth of the peaks. These are stored as "shift" and "linewidth" datasets. Other results can also be stored such as the amplitude of the peaks, the Brillouin loss tangent (BLT), the standard deviation of the shift, the standard deviation of the linewidth, etc. To allow for multiple treatments of the same "PSD" datasets to be stored together with it, we create for each collection of results, a dedicated "treatment" group, where these results are stored.

Additionally, we can store impulse responses to characterize the instrument in a dedicated "Impulse response" group to differentiate it from the normal measures. We can also store the calibration spectrum in a dedicated "Calibration spectrum" group with the same idea. This distinction is aimed at allowing noramlization algorithms to be applied to the data automatically in future meta-studies involving all the community.

To organize this file hierarchically, we also need groups dedicated to storing groups. As they form the basis of a branch of the file tree, we call them "Root" groups.

Overall, the diffrent types of groups and datasets that can be stored in the file format are shown in Fig. 1.2.

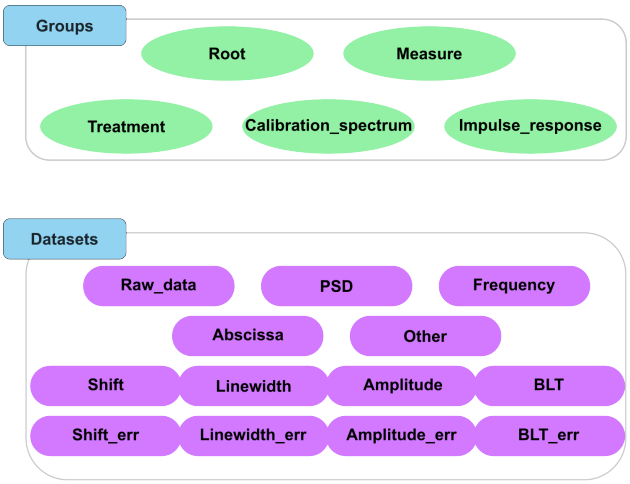


Fig. 1.2: A visual representation of the Brillouin_type attribute for groups and datasets in the HDF5 file.

To make the file human readable, we do not store the types of this groups and datasets in their name, but we create instead a dedicated attribute called "Brillouin_type" that allows to know the type of the element. Note that if this string is not given or recognized, it is set to "Other" by default. This attribute is therefore the first step in normalizing the file format.

Note that this very logical vision of the file format is not restrictive, and if you prefer to not store the raw data because you think it's not useful anymore, you don't have to. Same goes for the results: if you only need the shift array, you can just store this array. Same goes for the hierarchy: if you want to store the abscissa array in a parent group so that it applies to all the children measure groups, you can do so. This is the beauty of the file format: it is flexible and allows you to store whatever you need. From there, the challenge is to allow all these structures to be stored in a unified way, so that they can be used by the community. This is done in a second step, where we propose to define normalization rules.

CHAPTER

TWO

PROJECT DEVELOPMENT

2.1 Project specification sheet

The project is based on the following specifications:

1. Format 1.1. Simplicity 1.1.1. The format should be conceptually simple, unambiguous and easy to create and use.

- 1.1.1.1. Storing a single measure should be easy and unambiguous.
- 1.1.1.2. Storing arguments associated with a measure should be easy and unambiguous. The arguments should have a predefined nomenclature.
- 1.1.1.3. Storing measures performed with changes of different hyper parameters should be easy and unambiguous.
- 1.1.1.4. Storing a PSD and Frequency arrays extracted from a measure should be easy and unambiguous.
- 1.1.1.5. Storing the result of a treatment should be easy and unambiguous.

1.2. Universality 1.2.1. The format should be compatible with existing HDF5 softwares.

- 1.2.1.1. Opening the file with an HDF5 viewer should give us access to a user-friendly hierarchical structure.

1.2.2. The format should allow the storage of complemetary datasets obtained with other modalities (e.g. Raman, NMR, fluorescence, etc.).

- 1.2.2.1. The format should make a clear distinction between datasets related to BLS measurements and other datasets.
- 1.2.2.2. The format should allow user to store datasets related to other modalities with an arbitrary structure in the first description of the format.

1.2.3. The format should be adapted for storing Brillouin spectra obtained with different techniques.

- 1.2.3.1. The format should allow measures obtained with all techniques to be stored. This includes the sotrage of datasets with arbitrary dimensions, additional technique-specific datasets and additional technique-specific attributes.

1.3. Expandability 1.3.1. The format should accomodate future needs

- 1.3.1.1. The format should classify the data preferably by the hyper parameters that were varied for the experiment.

2. Analysis and treatment 2.1. Power Spectral Density 2.1.1. It must be possible to obtain a Power Spectral Density.

- 2.1.1.1. All relevant information and datasets for the obtention of the PSD should be in the same file at the moment of treatment.
- 2.1.1.2. All relevant information and datasets should be unambiguously and simply accessible.

- 2.1.1.3. The PSD and Frequency arrays should be stored unambiguously in the file so that we know which arrays and parameters were used to obtain them.
- 2.1.1.4. The process of obtaining the PSD should be documented in the file in a reusable way.

2.1.2. It must be possible to create custom algorithms to extract the PSD and frequency.

- 2.1.2.1. The function(s) extracting the PSD and frequency should have a fixed nomenclature for their name.
- 2.1.2.2. The function(s) extracting the PSD and frequency should all return the same type of data.
- 2.1.2.3. The function(s) extracting the PSD and frequency should all use the same type of documentation.
- 2.1.2.4. All function(s) extracting the PSD and frequency should be stored in the same module.
- 2.1.2.5. All function(s) extracting the PSD and frequency should also return the parameters they used to obtain the PSD and frequency, with which it is possible to reproduce the PSD and frequency.

2.2. Extraction of peak information 2.2.1. The PSD and Frequency datasets should allow for a unified way of extracting information, independent on the spectrometer used.

- 2.2.1.1. It should be unambiguous to assign a frequency axis to a PSD dataset based on the PSD and Frequency arrays.

2.2.2. The format should allow the user to extract information with different treatment and store all the results in the same file.

- 2.2.2.1. The treatments should have an identifier that can be incremented to allow the user to store different treatments.
- 2.2.2.2. Each new treatment should also store the parameters used to obtain the information in a way that the user can reproduce the information.

3. Graphical User Interface 3.1. Simplicity 3.1.1. Interaction with files and attributes should be as intuitive as possible.

- 3.1.1.1. Adding a file (attribute file or data file) to the format should be possible by dragging and dropping it in the GUI.
- 3.1.1.2. Changing a visible property of a dataset or group should be possible by left clicking on it and editing its value in the GUI.
- 3.1.1.3. A right click on a file or group should open a context menu with the action options to perform on the file or group.
- 3.1.1.4. It should be possible to perform the same action in the GUI by at least 2 redundant ways.
- 3.1.1.5. The GUI should be idiot proof and be able to flag all actions that might damage the file, make a treatment impossible, or induce any type of incompatibility.

2.2 Project structure

The project is structured around two axes:

- **File format**: Design a HDF5-based file format that is minimally constrained and that allows all BLS-derived data to be stored
- **Data processing**: Create data processing tools to treat BLS measures and extract relevant parameters from them - **For unifying the obtention of Power Spectral Density (PSD)**: Create a unified way to obtain the PSD from a measure given a spectrometer, and to store the results of the treatment of the data - **For extracting information from the PSD**: Create a unified way to extract information from the PSD, such as the shift, the linewidth, the amplitude, the BLT, etc.

2.3 Project timeline

The timeline of the project is divided in four phases:

- **Phase 1**: Design a HDF5-based file format that is minimally constrained and that allows all BLS-derived data to be stored
- **Phase 2**: Stress test the file format to make sure it answers all the needs of the community, in particular ensure compatibility with different spectrometers
- **Phase 3**: Create data processing tools to treat BLS measures and extract relevant parameters from them
- **Phase 3**: Define normalization rules to store particular data formats (single spectra, series of spectra evolving with respect to one hyper parameter, spatial images, etc.), and develop dedicated visualization tools to display the data, metadata and results.

2.4 User interfaces

The project consists of 3 independent Python packages to be used in Python scripts or Jupyter notebooks: - **HDF5_BLS**: The package to create and manipulate HDF5 files - **HDF5_BLS_analyse**: The package to perform the conversion of BLS measures to an associated Power Spectral Density (PSD) - **HDF5_BLS_treat**: The package to process the PSD to extract relevant information from it

To allow a use of the project without recurring to scripts, a Graphical User Interface (GUI) called *HDF5_BLS - The GUI* is accessible. The GUI is now capable of:

- **Creating HDF5 files following the structure of HDF5_BLS v1.0:**
  - Structure the file in a hierarchical way
  - Import measure data (drag and drop functionality implemented)
  - Import attributes from a CSV or Excel spreadsheet file (drag and drop functionality implemented)
  - Modify parameters of data both by group and individually from the GUI
- Inspect existing HDF5 files and in particular, ones made with the HDF5_BLS package
- Export sub-HDF5 files from meta files
- Export Python or Matlab code to access individual datasets
- Visualize 2D arrays as images
- Analyse raw spectra obtained with a VIPA spectrometer

CHAPTER

THREE

QUICKSTART

3.1 Spirit of the project

The idea of the package is to provide a simple way to store and retrieve data relevant to Brillouin Light Scattering experiments together with the metadata associated to the data. The file we propose to use is the HDF5 file format (standing for "Hierarchical Data Format version 5"). The idea of this project is to use this file format to reproduce the structure of a filesystem within a single file, storing all files corresponding to a given expoeriment in a single "group". For example, a typical structure of the HDF5 file could be:

```
file.h5
└── Brillouin
    ├── Measure of water
    │   ├── Image of the power spectral density
    │   ├── Channels associated to the power spectral density
    │   ├── Results after data processing
    │   │   ├── Shift
    │   │   ├── Shift variance
    │   │   ├── Linewidth
    │   │   ├── Linewidth variance
    │   │   ├── Amplitude
    │   │   ├── Amplitude variance
    │   │   ├── ...
    ├── Measure of methanol
    │   ├── ...
```

To allow this file format to be used with other modalities (e.g. electrophoresis assays to complement a Brillouin experiment), we propose to use a top-level group corresponding a minima to the modality (e.g. "Brillouin"). We also propose to add to each element of the HDF5 file, a "Brillouin_type" attribute that will allow to know the type of the element. For datasets, these types are:

- Raw_data: the raw data
- PSD: a power spectral density array
- Frequency: a frequency array associated to the power spectral density
- Abscissa_x: an abscissa array for the measures where the name is written after the underscore.
- Shift: the shift array obtained after the treatment
- Shift_err: the array of errors on the shift array obtained after the treatment
- Linewidth: the linewidth array obtained after the treatment
- Linewidth_err: the array of errors on the linewidth array obtained after the treatment

- Amplitude: the amplitude array obtained after the treatment
- Amplitude_err: the array of errors on the amplitude array obtained after the treatment
- BLT: the Loss Tangent array obtained after the treatment
- BLT_err: the array of errors on the Loss Tangent array obtained after the treatment

For groups, these types are:

- Calibration_spectrum: the calibration spectrum
- Impulse_response: the impulse response
- Measure: the measure
- Root: the root group
- Treatment: the treatment

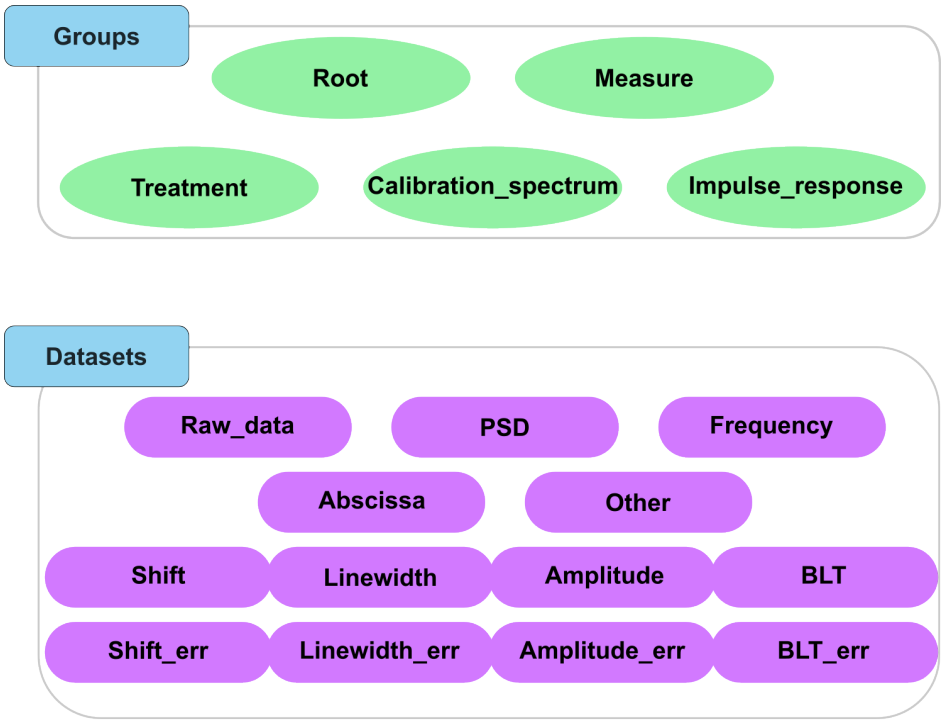


Fig. 3.1: A visual representation of the Brillouin_type attribute for groups and datasets in the HDF5 file.

3.2 Installation

To install the package, you can use pip:

```
pip install HDF5_BLS
```

3.3 Usage

3.3.1 Integration to workflow

Once the package is installed, you can use it in your Python scripts as follows:

```
import HDF5_BLS as bls

# Create a HDF5 file
wrp = bls.Wrapper(filepath = "path/to/file.h5")

################################################################################
# Existing code to extract data from a file
################################################################################
# Storing the data in the HDF5 file (for this example we use a random array)
data = np.random.random((50, 50, 512))
wrp.add_raw_data(data = data, parent_group = "Brillouin/Measure", name = "Raw data")

################################################################################
# Existing code to convert the data to a PSD
################################################################################
# Storing the Power Spectral Density in the HDF5 file together with the associated␣
↪frequency array (for this example we use random arrays)
PSD = np.random.random((50, 50, 512))
frequency = np.arange(512)
wrp.add_PSD(data = PSD, parent_group = "Brillouin/Measure", name = "Power Spectral␣
↪Density")
wrp.add_frequency(data = frequency, parent_group = "Brillouin/Measure", name = "Frequency
↪")

################################################################################
# Existing code to fit the PSD to extract shift and linewidth arrays
################################################################################
# Storing the Power Spectral Density in the HDF5 file together with the associated␣
↪frequency array (for this example we use random arrays)
shift = np.random.random((50, 50))
linewidth = np.random.random((50, 50))
wrp.add_treated_data(parent_group = "Brillouin/Measure", name_group = "Treat_0", shift =␣
↪shift, linewidth = linewidth)

################################################################################
# Other methods to add data to the HDF5 file
################################################################################
# If you want to add an abscissa array, you can use the following command where you␣
↪specify which dimensions the abscissa array applies to
wrp.add_abscissa(data, "Brillouin/Measure", name="Time", unit = "min", dim_start = 0,␣
↪dim_end = 1, overwrite = False)
# If you want to add a particular attribute, you can use the following command
wrp.add_attributes({"an attribute": "its value"}, parent_group="Brillouin/Measure",␣
↪overwrite=True)
# If you want to add any other dataset, you can use the following command
wrp.add_other(data, "Brillouin/Measure", name="Other", overwrite=True)
```

To summarize this example, we first create a HDF5 file. Then we add our data to it, placing it under a group named "Measure". Datasets are added based on their nature, with an associated function. Here we add a PSD, a frequency array, a shift and linewidth array and then an abscissa and another dataset. Here are the liste of functions to use depending on the type of data:

- Raw data (data straight from the spectrometer): add_raw_data

- PSD (a Power Spectral Density array): add_PSD
- Frequency (a frequency array associated to the power spectral density): add_frequency
- Abscissa (an abscissa array for the measures where the name is written after the underscore): add_abscissa
- Shift, Linewidth, or any other type of results: add_treated_data
- Other (the other data): add_other

We can then extract the data from the HDF5 file using the wrapper object. The wrapper object is initialized by running the following command:

```
wrp = Wrapper()
```

3.3.2 Extracting the data from the HDF5 file

Once the data is stored in the HDF5 file, you can extract it as follows:

```
import HDF5_BLS as bls

# Open the file
wrp = bls.Wrapper(filepath = "path/to/file.h5")

# Extract the data
data = wrp["Brillouin/path/in/file/Raw data"]
```

To get the path leading to a dataset, you can either use existing software to browse the file (we recommend Panoply and myHDF5), or you can use the HDF5_BLS package to display the structure of the file:

```
print(wrp)
```

CHAPTER

FOUR

THE FILE FORMAT

4.1 A single measure

This project aims at defining a standard for storing Brillouin Light Scattering measures and associated treatment in a HDF5 file.

HDF5 stands for "Hierarchical Data Format" and is a file format that allows the storage of data in a hierarchical structure. This structure allows to store data in a way that is both human and machine readable. The structure of the file is based on the following base structure, which corresponds to the structure of a file containing a single measure (*Measure*) where no parameters have been stored:

```
file.h5
└── Brillouin (group)
    └── Measure (group)
        └── Measure (dataset)
```

The dimensionality of the dataset is free, there are therefore by design virtually no restrictions on the data that can be stored in this format.

The organization of the file is based on the following principles:

- The file is organized in groups and datasets, which allows to store data in a hierarchical structure.
- Only one dataset corresponding to a measure can be stored per group.
- The groups are used to organize the file and store metadata and parameters related to the measure, and the datasets are used to store the actual data.

4.2 A single measure with physical meaning

From the single measure file, we need to move towards a structure where datasets have a meaning and are not just a collection of numbers. To do so, we propose to always refer to the Power Spectral Density (PSD) as the basis for a measure. In this spirit, we add to our measure group two datasets containing the PSD and the corresponding frequency axis:

```
file.h5
└── Brillouin (group)
    └── Measure (group)
        ├── Measure (dataset)
        ├── PSD (dataset)
        └── Frequency (dataset)
```

To not constrain the names of the datasets, we assign to each of these datasets, a "Brillouin_type" attribute. This attribute is a string that allows to know the type of the element:

```
file.h5
└── Brillouin (group)
    └── Measure (group)
        ├── Measure (dataset, Brillouin_type = "Raw data")
        ├── PSD (dataset, Brillouin_type = "PSD")
        └── Frequency (dataset, Brillouin_type = "Frequency")
```

4.3 Multiple measures stored in the same file

In the case where we want to store multiple measures in the same file, we can multiply the number of groups stored under the "Brillouin" group. For the same reason than before, to not constrain the names of the group, we'll assign them "Brillouin_type" attributes to differentiate the groups that store groups ("Root") from the groups that store measures ("Measure"):

```
file.h5
└── Brillouin (group, Brillouin_type = "Root")
    ├── RWPE1 organoids (group, Brillouin_type = "Root")
    │   ├── Morphogenesis day 1 (group, Brillouin_type = "Root")
    │   │   ├── Sample 1 (group, Brillouin_type = "Measure")
    │   │   │   ├── Measure (dataset, Brillouin_type = "Raw data")
    │   │   │   ├── PSD (dataset, Brillouin_type = "PSD")
    │   │   │   └── Frequency (dataset, Brillouin_type = "Frequency")
    │   │   ├── Sample 2 (group, Brillouin_type = "Measure")
    │   │   │   ├── Measure (dataset, Brillouin_type = "Raw data")
    │   │   │   ├── PSD (dataset, Brillouin_type = "PSD")
    │   │   │   └── Frequency (dataset, Brillouin_type = "Frequency")
    │   │   ...
    │   ├── Morphogenesis day 2 (group, Brillouin_type = "Root")
    │   │   ├── ...
    ├── H6C7 organoids (group, Brillouin_type = "Root")
    ├── ...
```

4.4 Multiple measures stored with their results

The next step is to store the results of the treatment of the measure. The difficulty lies in the fact that multiple treatments can be applied to the same measure. To solve this problem, we propose to store all the results of a treatment in a dedicated "Treatment" group. This group will be named after the treatment and will contain for example the shift, the linewidth, the amplitude, the BLT, the error on the shift, etc.

Following the same logic, each of these datasets will be associated to a "Brillouin_type" attribute to recognize the nature of the dataset:

```
file.h5
└── Brillouin (group, Brillouin_type = "Root")
    └── RWPE1 organoids (group, Brillouin_type = "Root")
        └── Morphogenesis day 1 (group, Brillouin_type = "Root")
            ├── Sample 1 (group, Brillouin_type = "Measure")
            │   ├── Measure (dataset, Brillouin_type = "Raw data")
            │   ├── PSD (dataset, Brillouin_type = "PSD")
            │   ├── Frequency (dataset, Brillouin_type = "Frequency")
            │   └── Treatment (group, Brillouin_type = "Treatment")
```

(continues on next page)

(continued from previous page)

```
            ├── Shift (dataset, Brillouin_type = "Shift")
            ├── Linewidth (dataset, Brillouin_type = "Linewidth")
        └── Sample 2 (group, Brillouin_type = "Measure")
            ├── Measure (dataset, Brillouin_type = "Raw data")
            ├── PSD (dataset, Brillouin_type = "PSD")
            ├── Frequency (dataset, Brillouin_type = "Frequency")
            └── Treatment (group, Brillouin_type = "Treatment")
                ├── Shift (dataset, Brillouin_type = "Shift")
                └── Linewidth (dataset, Brillouin_type = "Linewidth")
```

4.5 Multiple measures stored with their results and algorithms

The file structure is capable of storing not only datasets, attributes related to how the data was collected, in a hierarchical way, but it also allows users to store their scripts. This is particularly useful when the user performs a new kind of data processing, or when he uses the data to generate figures. Any script can be stored in the file as text. Note that in order to differentiate between scripts and "normal" attributes, we recommend using the prefix "**script_**" for the attribute name, so as to clearly distinguish them from the other attributes.

These scripts can be in any programming language, and will in the future be runnable from the library itself (particularly to re-generate the figures). Here is an example of how a file storing a script would look like:

```
file.h5
└── Brillouin (group, Brillouin_type = "Root", script_plot_distributions = "import numpy␣
↪as np\nimport ...")
    ├── Concentration 1mMol/mL (group, Brillouin_type = "Measure")
    │   ├── PSD (dataset, Brillouin_type = "PSD")
    │   ├── Frequency (dataset, Brillouin_type = "Frequency")
    │   └── Treatment (group, Brillouin_type = "Treatment")
    │       ├── Shift (dataset, Brillouin_type = "Shift")
    │       └── Linewidth (dataset, Brillouin_type = "Linewidth")
    ├── Concentration 2mMol/mL (group, Brillouin_type = "Measure")
    │   ├── PSD (dataset, Brillouin_type = "PSD")
    │   ├── Frequency (dataset, Brillouin_type = "Frequency")
    │   └── Treatment (group, Brillouin_type = "Treatment")
    │       ├── Shift (dataset, Brillouin_type = "Shift")
    │       └── Linewidth (dataset, Brillouin_type = "Linewidth")
    ├── Concentration 3mMol/mL (group, Brillouin_type = "Measure")
    │   ├── ...
```

CHAPTER

FIVE

NORMALIZATION RULES FOR THE BIOBRILLOUIN COMMUNITY

In order to be able to share and compare data between members of the BioBrillouin community, we need to agree on a set of normalization rules. We here propose a basis, largely inspired by the OME community.

This section is divided in two parts:

- Normalization rules for attributes
- Normalization rules for datasets

5.1 Normalization rules for attributes

5.1.1 The Brillouin_type Attributes

The Brillouin_type attribute is used to recognize the type of the element. Note that this attribute is therefore the first step in normalizing the file format. It can be any one of the following values:

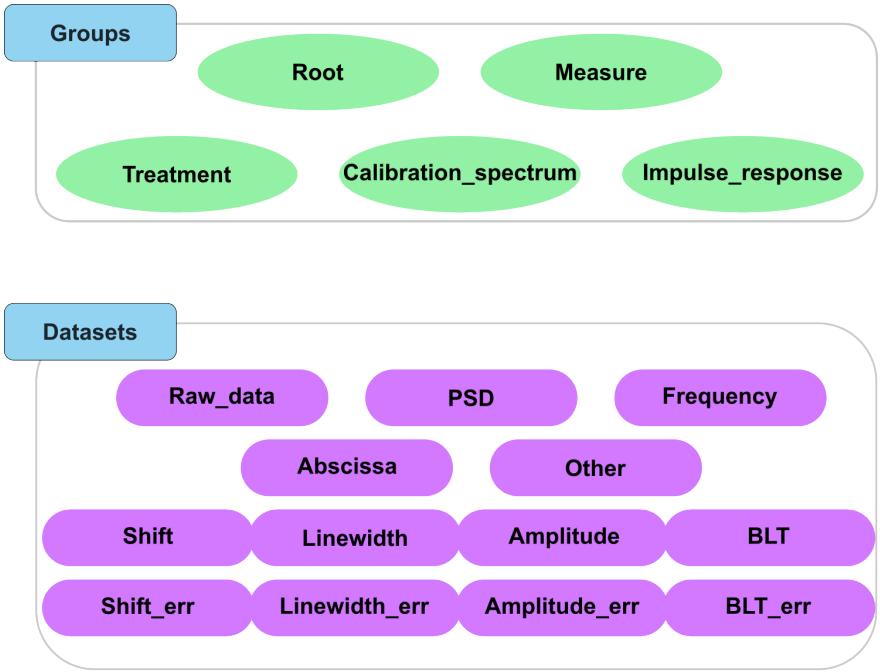


Fig. 5.1: A visual representation of the Brillouin_type attribute for groups and datasets in the HDF5 file.

5.1.2 Hierarchical strorage of attributes

HDF5 file format allows the storage of attributes in the metadata of the groups and datasets. As all the attributes applying to a measure will apply to all the elements of the "measure" group, we propose to store all the attributes concerning an experiment in the attributes of the group:

```
file.h5
└── Brillouin (group)
    ├── Measure (group) -> attributes of the measure
        └── Measure (dataset)
```

Being a hierarchical format, we also propose to store attributes hierarchically: all attributes of parent group apply to childre groups (if not redefined in children groups). Storing attributes in large files can therefore be done the following way:

```
file.h5
└── Brillouin (group) -> attributes shared by Measure 0 and Measure 1
    ├── Measure 0 (group) -> other attributes specific to Measure 0
    │   └── Measure (dataset)
    └── Measure 1 (group) -> other attributes specific to Measure 1
        └── Measure (dataset)
```

This approach allows us to store for example the parameters of the spectrometer used in a series of measures in the topmost group of the file.

Note that the same logic applies to the treatment of the data. The attributes of the treatment will be stored in the attributes of the "Treatment" group.

```
file.h5
└── Brillouin (group)
    └── Measure (group) -> attributes of the measure
        ├── Measure (dataset)
        └── Treatment (group) -> attributes of the treatment
            └── Shift (dataset)
```

5.1.3 Attribute type

In an effort to avoid any incompatibility, we propose to store the values of the attributes as ascii-encoded text. The library will then convert the strings to the appropriate type (e.g. a float for a temperature value).

5.1.4 Organization of the attributes

5.1.4.1 Prefix

We differentiate 5 types of attributes, that we differentiate using the following prefixes:

- SPECTROMETER - Attributes that are specific to the spectrometer used, such as the wavelength of the laser, the type of laser, the type of detector, etc. These attributes are recognized by the capital letter word "SPECTROMETER" in the name of the attribute.
- MEASURE - Attributes that are specific to the sample, such as the date of the measure, the name of the sample, etc. These attributes are recognized by the capital letter word "MEASURE" in the name of the attribute.
- FILEPROP - Attributes that are specific to the original file format, such as the name of the file, the date of the file, the version of the file, the precision used on the storage of the data, etc. These attributes are recognized by the capital letter word "FILEPROP" in the name of the attribute.

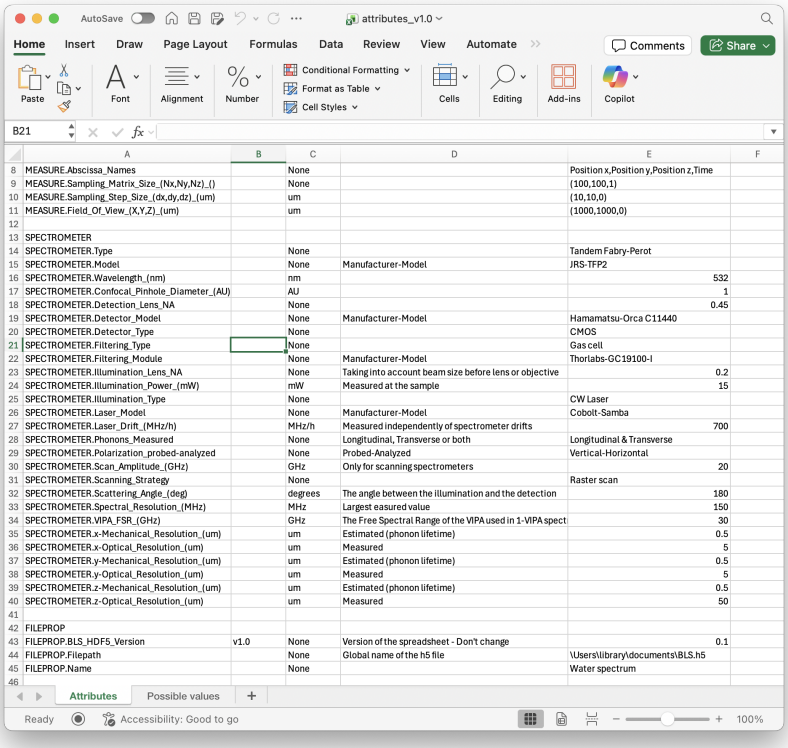
- PROCESS - Attributes that are specific to the storage of algorithms. These attributes are recognized by the capital letter word “PROCESS” in the name of the attribute.
- Attributes that are used inside the HDF5 file, such as the “Brillouin_type” attribute. These attributes are the only ones without a prefix.

5.1.4.2 Units

The name of the attributes contains the unit of the attribute if it has units, in the shape of an underscore followed by the unit in parenthesis. Some parameters will also be given following a given norm, such as the ISO8601 for dates. These norms are not specified in the name of the attribute. Here are some examples of attributes:

- “SPECTROMETER.Detector_Type” is the type of the detector used.
- “MEASURE.Sample” is the name of the sample.
- “MEASURE.Exposure_(s)” is the exposure of the sample given in seconds
- “MEASURE.Date_of_measurement” is the date of the measurement, stored following the ISO8601 norm.
- “FILEPROP.Name” is the name of the file.

To ease the unification of nomenclature and norms of attributes, we propose to use a spreadsheet that contains the list of attributes, their definition, their unit and an example of value. This spreadsheet is available on the project repository and is updated as new attributes are added to the project. Each attribute has a version number that is also stored in the attributes of each data attribute (under FILEPROP.version).



	A	B	C	D	E	F
8	MEASURE.Abscissa_Names		None		Position x,Position y,Position z,Time	
9	MEASURE.Sampling_Matrix_Size_(Nx,Ny,Nz)_()		None		(100,100,1)	
10	MEASURE.Sampling_Step_Size_(dx,dy,dz)_(um)		um		(10,10,0)	
11	MEASURE.Field_Of_View_(X,Y,Z)_(um)		um		(1000,1000,0)	
12						
13	SPECTROMETER					
14	SPECTROMETER.Type		None		Tandem Fabry-Perot	
15	SPECTROMETER.Model		None	Manufacturer-Model	JRS-TFP2	
16	SPECTROMETER.Wavelength_(nm)		nm		532	
17	SPECTROMETER.Confocal_Pinhole_Diameter_(AU)		AU		1	
18	SPECTROMETER.Detection_Lens_NA		None		0.45	
19	SPECTROMETER.Detector_Model		None	Manufacturer-Model	Hamamatsu-Orca C11440	
20	SPECTROMETER.Detector_Type		None		CMOS	
21	SPECTROMETER.Filtering_Type		None		Gas cell	
22	SPECTROMETER.Filtering_Module		None	Manufacturer-Model	Thorlabs-GC19100-I	
23	SPECTROMETER.Illumination_Lens_NA		None	Taking into account beam size before lens or objective	0.2	
24	SPECTROMETER.Illumination_Power_(mW)		mW	Measured at the sample	15	
25	SPECTROMETER.Illumination_Type		None		CW Laser	
26	SPECTROMETER.Laser_Model		None	Manufacturer-Model	Cobolt-Samba	
27	SPECTROMETER.Laser_Drift_(MHz/h)		MHz/h	Measured independently of spectrometer drifts	700	
28	SPECTROMETER.Phonons_Measured		None	Longitudinal, Transverse or both	Longitudinal & Transverse	
29	SPECTROMETER.Polarization_probed-analyzed		None	Probed-Analyzed	Vertical-Horizontal	
30	SPECTROMETER.Scan_Amplitude_(GHz)		GHz	Only for scanning spectrometers	20	
31	SPECTROMETER.Scanning_Strategy		None		Raster scan	
32	SPECTROMETER.Scattering_Angle_(deg)		degrees	The angle between the illumination and the detection	180	
33	SPECTROMETER.Spectral_Resolution_(MHz)		MHz	Largest easured value	150	
34	SPECTROMETER.VIPA_FSR_(GHz)		GHz	The Free Spectral Range of the VIPA used in 1-VIPA spect	30	
35	SPECTROMETER.x-Mechanical_Resolution_(um)		um	Estimated (phonon lifetime)	0.5	
36	SPECTROMETER.x-Optical_Resolution_(um)		um	Measured	5	
37	SPECTROMETER.y-Mechanical_Resolution_(um)		um	Estimated (phonon lifetime)	0.5	
38	SPECTROMETER.y-Optical_Resolution_(um)		um	Measured	5	
39	SPECTROMETER.z-Mechanical_Resolution_(um)		um	Estimated (phonon lifetime)	0.5	
40	SPECTROMETER.z-Optical_Resolution_(um)		um	Measured	50	
41						
42	FILEPROP					
43	FILEPROP.BLS_HDF5_Version	v1.0	None	Version of the spreadsheet - Don't change	0.1	
44	FILEPROP.Filepath		None	Global name of the h5 file	\Users\library\documents\BLS.h5	
45	FILEPROP.Name		None		Water spectrum	
46						

Fig. 5.2: A visualization of the spreadsheet containing the list of attributes.

5.1.5 Storing analysis and treatment processes performed with the HDF5_BLS package

Analysis and treatment processes are stored in the "PROCESS" attribute of the treatment groups. This attribute is a JSON file converted to a string, which contains the list of treatment steps performed on the data. This JSON file has the following structure:

```
{
    "name": "The name of the algorithm",
    "version": "v 0.1",
    "author": "Author name and affiliation",
    "description": "The description of the algorithm",
    "functions": [
    {
        "function": "The 1st function name in the class",
        "parameters": {
            "parameter_1": value,
            "parameter_2": value,
            ...
        },
        "description": "The description of the function"
    },
    {
        "function": "The 2nd function name in the class",
        "parameters": {
            "parameter_1": value,
            "parameter_2": value,
            ...
        },
        "description": "The description of the function"
    },
    ...
    ]
}
```

When the treatment is performed using the modules of the HDF5_BLS package, this attribute is automatically updated. Note that custom treatments can also be stored in this attribute by the user.

This attribute can be exported to a standalone JSON file using the library. This attribute also allows the library to re-apply the treatment to the data, and modify steps of the treatment if needed.

5.1.6 Storing custom data processing code and visualization code

To ease the use of the file format, we propose the user to store their own codes directly as attributes of the groups they apply to. For example, if a code performing a statistical analysis has been used on the data, the user can store this code as a text attribute to the topmost group of the file. This is an unconventional way to store the code, but is has the great advantage of allowing the user to recover the code easily and to re-run it or modify it if needed in the future.

Note that a function of the HDF5_BLS.Wrapper class (Wrapper.store_script) is available to perform the saving of the script easily. When called in a script, this function retrieves the caller's filename, converts the python script to a text string, and stores it in the HDF5 file (as is).

5.2 Normalization rules for datasets

5.2.1 General guidelines

1. Datasets should be presented as tensors where the dimensions are attribuated to the different hyperparameters of the experiment (see Fig. 5.3).

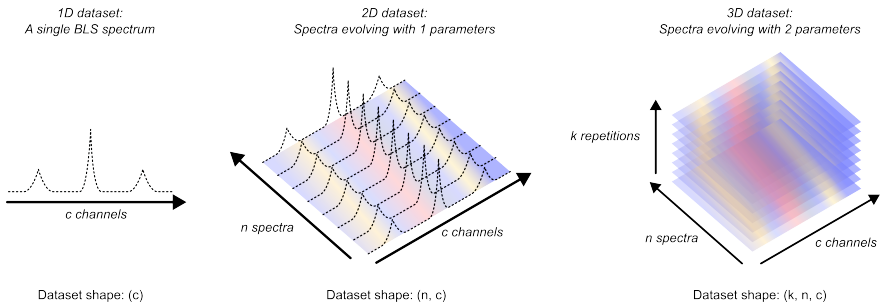


Fig. 5.3: A visual representation of different multidimensional datasets storing BLS spectra

2. The dimensions of the tensor should be ordered following this general convention, from last to first dimension (see Fig. 5.4):
    - Channels (n)
    - x-axis (n-1)
    - y-axis (n-2)
    - z-axis (n-3)
    - Time (n-4)
    - Temperature (n-5)
    - Angle (n-6)
    - Other (n-…)

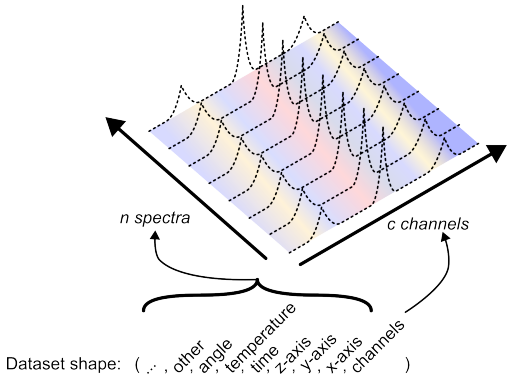


Fig. 5.4: A visual representation of the normalization of the physical meaning of the dimensions of a dataset

3. In cases of non-uniform sampling of one or more of the hyperparameters used in the experiment, the associated dimensions are broadcasted on the highest dimension, and the non-uniform dimension is set to 1.

Note: The arrangement of the hyperparameters differs from the one proposed by the OME community as most spectrometers will return the variation of the data with the frequency axis as the last dimension. Note however that for images, a simple transposition of the axes is enough to get a dataset compliant with the OME community convention.

5.2.2 Clarification on dataset shapes

A dataset containing *k* values varying for a single parameter, should have all their lower dimensions set to 1.

For example:

- a list of values of x-axis coordinates for each step should be of shape (k, 1), where *k* is the number of x coordinates.
- a list of values of y-axis coordinates for each step should be of shape (k, 1, 1), where *k* is the number of y coordinates.
- a list of values of z-axis coordinates for each step should be of shape (k, 1, 1, 1), where *k* is the number of z coordinates.
- a list of values of time steps for each step should be of shape (k, 1, 1, 1, 1), where *k* is the number of x coordinates.
- etc.

5.2.3 Uniform sampling examples

5.2.3.1 Series of measures on a 1D array

In this scenario, we vary one hyper parameter and measure the Brillouin spectra for each value this parameter can take. The hyper parameter is a 1D array of length *k*, and eqch Brillouin spectrum is an array of shape *c*. We consider that all the measures use the same frequency array for their channels.

- **In the case where the hyperparameter being varied is the *x-axis*:**
  - the PSD array is of shape (k, c)
  - the frequency array is of shape (c)
  - the abscissa array is of shape (k, 1)
  - all the result arrays are of shape (k, 1)
- **In the case where the hyperparameter being varied is the *y-axis*:**
  - the PSD array is of shape (k, 1, c)
  - the frequency array is of shape (c)
  - the abscissa array is of shape (k, 1, 1)
  - all the result arrays are of shape (k, 1, 1)
- **In the case where the hyperparameter being varied is the *temperature*:**
  - the PSD array is of shape (k, 1, 1, 1, 1, c)
  - the frequency array is of shape (c)
  - the abscissa array is of shape (k, 1, 1, 1, 1)
  - all the result arrays are of shape (k, 1, 1, 1, 1)

5.2.3.2 Series of measures on a 2D grid

Let's consider the scenario of a study performed on 2 varying hyperparameters.

Case 1: Uniform sampling on each dimension of the grid

The first case is when the grid is uniformly sampled on each dimension. Let's consider *k1* samples on the first dimension and *k2* samples on the second dimension. The number of channels is still *c*.

- **In the case where the hyperparameter being varied are the *x-axis* and *y-axis*:**

  - the PSD array is of shape (k1, k2, c)
  - the frequency array is of shape (c)
  - the x abscissa array is of shape (k2, 1)
  - the time abscissa array is of shape (k1, 1, 1)
  - all the result arrays are of shape (k1, k2, 1)
- **In the case where the hyperparameter being varied are the *x-axis* and *time*:**
  - the PSD array is of shape (k1, 1, 1 k2, c)
  - the frequency array is of shape (c)
  - the x abscissa array is of shape (k2, 1)
  - the time abscissa array is of shape (k1, 1, 1, 1, 1)
  - all the result arrays are of shape (k1, 1, 1 k2, 1)
- **In the case where the hyperparameter being varied are the *z-axis* and *time*:**
  - the PSD array is of shape (k1, k2, 1, 1, c)
  - the frequency array is of shape (c)
  - the z abscissa array is of shape (k2, 1, 1, 1)
  - the time abscissa array is of shape (k1, 1, 1, 1, 1)
  - all the result arrays are of shape (k1, k2, 1, 1, 1)

Case 2: Non-uniform sampling on each dimension of the grid

In this case, the grid is not uniformly sampled on each dimension. This means that the second dimension of the hyperparameter depends on the first. In other terms, if a spectrum was measured at the coordinate (x, y), then it is now measured at the coordinate (x, y(x)). In that case, we broadcast the dimension on the latest dimension of the hyperparameter, and set it's normal dimension to 1. Let's consider a total of k measures performed on the grid. The number of channels is still c.

- **In the case where the hyperparameter being varied are the *x-axis* and *y-axis*:**
  - the PSD array is of shape (1, k, c)
  - the frequency array is of shape (c)
  - the abscissa array is of shape (1, k, 1)
  - all the result arrays are of shape (1, k, 1)
- **In the case where the hyperparameter being varied are the *x-axis* and *time*:**
  - the PSD array is of shape (1, 1, 1, k, c)
  - the frequency array is of shape (c)
  - the x-abscissa array is of shape (1, 1, 1, k, 1)
  - the time abscissa array is of shape (k, 1, 1, 1, 1)
  - all the result arrays are of shape (1, 1, 1, k, 1)
- **In the case where the hyperparameter being varied are the *z-axis* and *time*:**
  - the PSD array is of shape (k1, k2, 1, 1, c)
  - the frequency array is of shape (c)

  - the z abscissa array is of shape (1, k, 1, 1, 1)
  - the time abscissa array is of shape (k, 1, 1, 1, 1)
  - all the result arrays are of shape (1, k, 1, 1, 1)

Case 3: Non-uniform sampling on higher dimensions

A particularly interesting case in screening is when spectra are acquired as function of random values for an arbitrarily large number of hyper parameters. In this very special case, all the hyper parameters will vary with respect to all other hyper parameters. In this very special case, the user will store his data with the same idea:

- the PSD array is of shape (1, ..., 1, k, 1, ..., 1, c)
- the frequency array is of shape (c)
- all the abscissa arrays associated to any of the hyperparameters will be of shape (1, ..., 1, k, 1, ..., 1)
- all the result arrays are of shape (1, ..., 1, k, 1, ..., 1)

Note that these conventions are to allow data to be stored and shared between the community. In this latter case, it is very unlikely that the user will share these measures with the community, therefore enforcing the present convention is left to the user.

Part II

User Guide

CHAPTER

SIX

THE HDF5_BLS PACKAGE

6.1 Installation and presentation

6.1.1 Installation

To install the package, you can either: * Download the source code from the GitHub repository (the hard way) * Use pip to install the package (easy way)

6.1.1.1 With pip

To install the HDF5_BLS library to use in Python scripts or Jupyter notebooks, you can use pip:

```
pip install HDF5_BLS
```

This will install the latest version of the package from the Python Package Index (PyPI).

Note: This allows you to install the HDF5_BLS library but not the HDF5_BLS_analyse and HDF5_BLS_treat libraries that have to be installed separately. This installation also does not allow you to run the GUI. To use the GUI, please download the entire GitHub repository and run the GUI from source.

6.1.1.2 From source

The source code can be downloaded from the GitHub repository. To download the source code, click on the green button "Code" and then click on the "Download ZIP" button. Once the download is complete, unzip the file and open a terminal in the folder where the code is stored. To install the package, run the following command:

```
python setup.py install
```

This will install the package and all its dependencies. To check if the package was installed correctly, run the following command:

```
python -c "import HDF5_BLS"
```

If the package was installed correctly, the command will not return any error.

This option is especially useful if you want to contribute to the package, as you can also clone the repository, and then use pip to install the package in editable mode. This will allow you to make changes or update the package from Git without having to reinstall the package every time.

6.1.2 Presentation

The HDF5_BLS library is a Python package meant to interface Python code with a HDF5 file. The development of this solution was based on three main goals: 1. Simplicity: Make it easy to store and retrieve data from a single file. 2. Universality: Allow all modalities to be stored in a single file, while unifying the metadata associated to the data. 3. Expandability: Allow the format to grow with the needs of the community.

Practically, this means using this solution should be easy, intuitive and allow a seamless integration of the proposed standard to your existing code. Here is a quick code example to show the integration of the package in a simple case (you want to store a signal out of a spectrometer, its corresponding power spectral density, its frequency axis and its shift and linewidth arrays):

```
##############################################################################
# Existing imports
##############################################################################
from HDF5_BLS import wrapper

# Create a new file
wrp = wrapper.Wrapper(filepath = "path/to/the/file.h5")

##############################################################################
# Existing code extracting data from a file
##############################################################################

# Store the data in the file
wrp.add_raw_data(data = data, parent_group = "Brillouin/path/in/the/file", name = "Name␣
↪of the dataset")

##############################################################################
# Existing code extracting a PSD and a frequency vector from the data
##############################################################################

# Store the frequency vector together with the raw data
wrp.add_frequency(data = frequnecy, parent_group = "Brillouin/path/in/the/file", name =
↪"Frequency vector")

# Store the PSD dataset together with the raw data
wrp.add_PSD(data = PSD, parent_group = "Brillouin/path/in/the/file", name = "PSD")

##############################################################################
# Existing code extracting the shift and linewidth of the data
##############################################################################

# Store the PSD dataset together with the raw data
wrp.add_treated_data(shift = shift, linewidth = linewidth, parent_group = "Brillouin/
↪path/in/the/file", name = "PSD")
```

Unification

This package also aims at unifying all the processing steps to extract PSD from raw data and extract Brillouin shift and linewidth from the PSD. These unifying steps are however not necessary to store your data in an HDF5 file. We will describe later the solution we propose to do this.

6.2 The Wrapper object

The "wrapper" module has one main object: *Wrapper*. This object is used to interact with the HDF5 file. It is used to read the data, to write the data and to modify any aspect of the HDF5 file (dataset, groups or attributes). The module also provides different error objects used to recognize errors when using the Wrapper object and raise exceptions.

The Wrapper object is initialized by running the following command:

```
wrp = Wrapper()
```

This will create a new Wrapper object with no attributes or data, and with the following structure:

```
file.h5
 └── Brillouin (group)
```

By default, the attributes of the "Brillouin" group are the following:

```
file.h5
└── Brillouin (group)
    ├── Brillouin_type -> "Root"
    └── HDF5_BLS_version -> "0.1" # The version of the HDF5_BLS package
```

As long as no filepaths are given to the Wrapper object, the file is stored in a temporary folder (the temporary folder of the operating system). Note that this temporary file is deleted either when the Wrapper object is destroyed or when the file is stored elsewhere. It is therefore good practice to specify a non-temporary filepath to the file when creating a new Wrapper object, with the "filepath" parameter:

```
wrp = Wrapper(filepath = "path/to/file.h5")
```

This will create a new Wrapper object with no attributes or data, and with the following structure:

```
path/to/file.h5
└── Brillouin (group)
```

Note that this works both for new files, and for files that already exist, in the latter case, the wrapper object applies to the file located at "path/to/file.h5".

6.3 Adding data to the HDF5 file (from script)

The addition of any type of data or attribute to the HDF5 file has been centralized in the *Wrapper.add_ dictionary* method. This method is safe but complex and not user-friendly. Methods derived from this method are meant to simplify the process of adding data to the HDF5 file, specific to each type of data.

To add a single dataset to a group, we first need to specify the type of dataset we want to add, which are the following (see `Dataset_types`):

- "Abscissa_…": An abscissa array for the measures where the dimensions on which the dataset applies are given after the underscore.
- "Amplitude": The dataset contains the values of the fitted amplitudes.
- "Amplitude_err": The dataset contains the error of the fitted amplitudes.
- "BLT": The dataset contains the values of the fitted amplitudes.
- "BLT_err": The dataset contains the error of the fitted amplitudes.
- "Frequency": A frequency array associated to the power spectral density

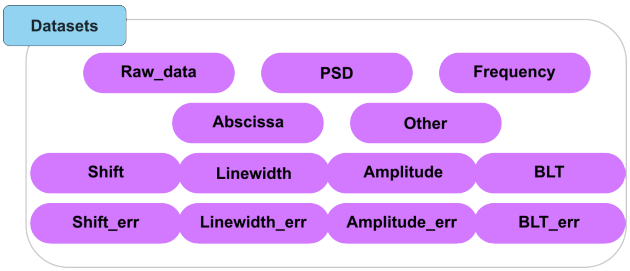


Fig. 6.1: A visual representation of the types of datasets that can be added to the HDF5 file.

- "Linewidth": The dataset contains the values of the fitted linewidths.
- "Linewidth_err": The dataset contains the error of the fitted linewidths.
- "PSD": A power spectral density array
- "Raw_data": The dataset containing the raw data obtained after a BLS experiment.
- "Shift": The dataset contains the values of the fitted frequency shifts.
- "Shift_err": The dataset contains the error of the fitted frequency shifts.
- "Other": The dataset contains other data that will not be used by the library.

From there, the following functions are available to add the dataset to the HDF5 file:

- add_raw_data: To add raw data to a group
- add_PSD: To add a PSD to a group
- add_frequency: To add a frequency axis to a group
- add_abscissa: To add an abscissa to a group
- add_treated_data: To add a shift, linewidth and their respective errors to a dedicated "Treatment" group
- add_other: To add a shift, linewidth and their respective errors to a dedicated "Treatment" group

6.3.1 General approach to adding data to the HDF5 file

Adding a dataset to the file always come with three other pieces of information:

- Where to add the dataset in the file
- What to call the added dataset
- What is the type of the dataset we want to add

To add a dataset to the file, we'll therefore call type-specific functions with the data to add, the place where to add it and the name to give the dataset as arguments, following a code of line resembling:

```
wrp.add_raw_data(data = data,
                 parent_group = "Brillouin/Water spectrum",
                 name = "Measure of the year")
```

This approach is the one used for * *add_raw_data* * *add_PSD* * *add_frequency* * *add_other*

Example Let's consider the following example: we have just initialized a wrapper object and want to add a spectrum obtained from our spectrometer. We have already converted this spectrum to a numpy array, and named it *data*. Now we want to add this data in a group called "Water spectrum" in the root group of the HDF5 file and call this raw data "Measure of the year". Then we will write:

```
wrp.add_raw_data(data = data,
                 parent_group = "Brillouin/Water spectrum",
                 name = "Measure of the year")
```

Now let's say that we have analyzed this spectrum and obtained a PSD (stored in the variable "psd") and frequency array (stored in the variable "freq"). We want to add these two arrays in the same group, and call them "PSD" and "Frequency" respectively. We will write:

```
wrp.add_PSD(data = psd,
            parent_group = "Brillouin/Water spectrum",
            name = "PSD")
wrp.add_frequency(freq,
                  parent_group = "Brillouin/Water spectrum",
                  name = "Frequency")
```

6.3.2 Exception 1: Adding treated data

Adding treated data differs slightly from adding individual datasets as we'll usually collect a number of different results to store. Therefore, instead of using different functions to store a shift or linewidth array, we have chosen to use a single function to add all the results of treatment, and create the group dedicated to storing the treatment results. As such, the function will have the following attributes:

- parent_group: The parent group where to store the data in the HDF5 file
- name_group: The name of the group that will contain the treatment results
- amplitude (optional): The amplitude array to add
- amplitude_err (optional): The error of the amplitude array
- blt (optional): The Loss Tangent array to add
- blt_err (optional): The error of the Loss Tangent array
- linewidth (optional): The linewidth array to add
- linewidth_err (optional): The error of the linewidth array
- shift(optional): The shift array to add
- shift_err (optional): The error of the shift array
- treat (optional): An HDF5_BLS_Treat.Treat object to add. This object stores all teh optional parameters above. It also stores the process as a JSON file that is stored as an attribute of the group.

Example

Let's consider the following example: we have treated our data and have obtained a shift array (shift), a linewidth array (linewidth) and their errors (shift_err and linewidth_err). We want to add these arrays in the same group as the PSD, that is the group "Test". The treated data are stored in a separate group nested in the "Test" group by the choices made while building the structure of the file. This is so the name of the treatment group can be chosen freely. Let's say that in this case, we have performed a non-negative matrix factorization (NnMF) on the data, and extracted the shift values closest to 5GHz. We will therefore call this treatment "NnMF - 5GHz". We will write:

```
wrp.add_treated_data(shift = shift,
                     linewidth = linewidth,
                     shift_err = shift_err,
                     linewidth_err = linewidth_err,
```

(continues on next page)

(continued from previous page)

```
                    parent_group = "Brillouin/Test",
                    name_group = "NnMF - 5GHz")
```

6.3.3 Exception 2: Adding an abscissa

Adding abscissa also differs from the general case as we might want to add an abscissa array that is multi-dimensional and be able to know which dimensions of the PSD the abscissa correspomnds to. The *add_abscissa* method therefore has the following attributes:

- parent_group: The parent group where to store the data in the HDF5 file
- name: The name of the abscissa to add
- unit: The unit of the axis
- dim_start: The first dimension of the abscissa array, by default 0
- dim_end: The last dimension of the abscissa array, by default the last number of dimension of the array

Example Let's consider the following example: we have just initialized a wrapper object and want to add an abscissa axis corresponding to our measures that have been stored in the group "Brillouin/Temp". Say that this abscissa axis corresponds to temperature values, from 35 to 40 degrees and that there are 10 points in the axis. We will therefore call this abscissa axis "Temperature". We will write:

```
wrp.add_abscissa(data = np.linspace(35, 40, 10),
                 parent_group = "Brillouin/Temp",
                 name = "Temperature",
                 unit = "C",
                 dim_start = 0,
                 dim_end = 1)
```

If you now want to use custom values for this axis, you can also specify them directly in the function call:

```
wrp.add_abscissa(data = data,
                 parent_group = "Brillouin/Temp",
                 name = "Temperature",
                 unit = "C",
                 dim_start = 0,
                 dim_end = 1)
```

6.4 Importing data from external files

Importing datasets to the HDF5 file from independent data files, through the HDF5_BLS package, is always done following to successive steps:

1. Extracting the data and the metadata that can be extracted from the data files. This can be done using the *load_data* module.
2. Adding the data and metadata to the HDF5 file. This is done using the *Wrapper.add_dictionary* method.

To make the process more user friendly, we have developed a set of derived methods that are specific to each type of data that is to be added (Raw data, PSD, Frequency, Abscissa or treated data).

In this section, we will present these methods. We encourage interested readers to refer to the chapter dedicated to the load_data module for more information on the extraction of the data and the metadata.

6.4.1 General approach for importing data

Much like adding data from a script, we can import data from external files by using type-specific functions. These functions are:

- *Wrapper.import_abscissa*: To import an abscissa array.
- *Wrapper.import_frequency*: To import a frequency array.
- *Wrapper.import_PSD*: To import a PSD array.
- *Wrapper.import_raw_data*: To import raw data.
- *Wrapper.import_treated_data*: To import the data arrays resulting from a treatment.

These function work a bit differently from the ones used to add data, as we might need parameters to extract the data from the file. Therefore, these functions have the following attributes:

- filepath: The filepath to the file to import the data from.
- parent_group: The parent group where to store the data in the HDF5 file.
- name: The name of the dataset to add.
- creator: An identifier of the creator of the file. This is used to differentiate different structures of files using the same format (for example .dat files).
- parameters: A dictionary containing the parameters that are needed to extract the data from the file. This is used to either access the data if it is somehow encoded or to interpret the data if a routine pipeline is used to obtain for example a PSD from a time-domain dataset
- reshape: The new shape of the array, by default None means that the shape is not changed
- overwrite: A parameter to indicate whether the dataset should be overwritten if it already exists, by default False - attributes are not overwritten.
- *all the attributes corresponding to the type of data (abscissa, frequency, PSD, raw data, treated data)*

Example Let's consider the following example: we have just initialized a wrapper object and want to import an abscissa axis corresponding to our measures that have been stored in a .npy file (for example if a Python routine has been used to impose conditions for the measure). In that case, the array can be interpreted without any parameters nor specification.

```
wrp.import_abscissa(filepath = "path/to/file.npy", parent_group = "Brillouin/Measure",␣
↪creator = None, parameters = None, name = "Time", unit = "s", dim_start = 0, dim_end =␣
↪1, reshape = None, overwrite = False)
```

6.5 Adding and merging HDF5 files

Creating a new HDF5 file based on two existing ones can be done one of two ways depending on the desired end result.

- The *__add__* dunder metthod. If we want to combine two HDF5 files into a single one "plainly", for example if we are generating a new HDF5 after each measure, with this structure:

```
20250214_HVEC_03.h5
└──Brillouin (group)
    ├── 20250214_HVEC_02
        └── Measure (dataset)
```

  and we already have a HDF5 file containing the data of the previous experiment:

```
20250214_HVEC.h5
├── Brillouin (group)
    ├── 20250214_HVEC_01
    │   └── Measure (dataset)
    ├── 20250214_HVEC_02
        └── Measure (dataset)
```

We can simply add the first HDF5 file to the second one with:

```
wrp1 = Wrapper(filepath = ".../20250214_HVEC.h5")
wrp2 = Wrapper(filepath = ".../20250214_HVEC_02.h5")
wrp = wrp1 + wrp2
```

This will create a new HDF5 file with the following structure:

```
20250214_HVEC.h5
└── Brillouin (group)
    ├── 20250214_HVEC_01
    │   └── Measure (dataset)
    ├── 20250214_HVEC_02
    │   └── Measure (dataset)
    └── 20250214_HVEC_03
        └── Measure (dataset)
```

WARNING: The new file is a temporary file, it is therefore important to save it after the addition of the two files with:

```
wrp.save_as_hdf5(filepath = wrp.filepath)
```

Note that from there, wrp1 and wrp will be the same as the wrapper does not store any data in memory but just acts as an access facilitator to the file.

- The *add_hdf5* method. If we want to import the HDF5 as a new group, for example if we have this HDF5 file containing the data of a cell study:

```
Neuronal_cell_study.h5
└── Brillouin (group)
    ├── Neuronal (group)
    │   ├── GT 1-7 (group)
    │   │   └── ...
    │   ├── L-fibroblast (group)
    │   │   └── ...
    └── Skeletal (group)
        └── ...
```

And we want to import data done on another neuronal cell line, say "MOV", that have been stored in the following HDF5 file:

```
MOV.h5
└── Brillouin (group)
    └── MOV (group)
        └── ...
```

We can simply add the second HDF5 file to the first one by specifying the path to the second file in the *Wrapper.add_hdf5* method:

```
wrp1 = Wrapper(filepath = ".../Neuronal_cell_study.h5")
wrp1.add_hdf5(filepath = ".../MOV.h5", parent_group = "Brillouin/Neuronal")
```

This will create a new HDF5 file with the following structure:

```
Neuronal_cell_study.h5
└── Brillouin (group)
    ├── Neuronal (group)
    │   ├── GT 1-7 (group)
    │   │   └── ...
    │   ├── L-fibroblast (group)
    │   │   └── ...
    │   ├── MOV (group)
    │   │   └── ...
    └── Skeletal (group)
        └── ...
```

CHAPTER

SEVEN

DATA PROCESSING WITH HDF5_BLS

7.1 Specifications

Data processing is arguably the second most important part of any study, following the acquisition of the data. Currently, no consensus exists on the way data are processed in BLS. Here, we propose a solution to that problem by offering two packages for unifying the extraction of a Power Spectral Density (PSD) from measures and extracting information from the PSD.

These two packages, namely *HDF5_BLS_analyse* and *HDF5_BLS_treat* are based on the same idea: encouraging the use of unified algorithms while allowing other algorithms to be developped and used. To that end, we built these packages with the following specifications in mind:

- **Modularity**: Algorithms are defined as a succession of "standard functions" that serve as basic blocks. Any algorithm utilizing these basic blocks can therefore be applied.
- **Reversible processing**: Algorithms can be stored in objects and re-run entirely or partially (up to a certain function)
- **Algorithm storage**: Algorithms can be stored as standalone files which can be imported, exported and edited.
- **Algorithm readability**: Algorithms are saved with a description of what they do and how they do it. The docstring of the functions used can also be compiled together to obtain a human readable description of the process followed during the algorithm
- **Developper friendliness**: Addition of functions to the processing module should be accessible to most.

With these 5 points in mind, we can describe the process followed in the development of the two processing modules.

7.2 Organization of the modules

The modules are organized in classes, that are either used as "accesories" (for storing analytical definition of models for example) or as "main actors" (classes where the processing functions are defined). These latter classes inherit from "backend" classes which are silent objects used as observers of the execution of the code, and tools for the manipulation of algorithms. The backend classes are:

- **Treat_backend**: The backend class of the *HDF5_BLS_treat* module
- **Analyse_backend**: The backend class of the *HDF5_BLS_analyse* module

Important

These classes are low-level classes developped as the basis for the rest of the modules to be built on. They should be silent and not be used directly by the user.

7.2.1 HDF5_BLS_analyse

For the *HDF5_BLS_analyse* module, a general class performing general operations on the data is defined. This class is called **Analyse_general** and inherits from **Analyse_backend**. This class is used to perform operations on the data that are not specific to a particular type of spectrometer. For example, the function to add a remarkable point to the data. Functions specific to a particular spectrometers are defined in classes that inherit from **Analyse_general**, and are dedicated to the analysis of data obtained with that spectrometer.

Here are a few examples of such classes: - **Analyse_VIPA**: for the analysis of data obtained with VIPA spectrometers. - **Analyse_TFP**: for the analysis of data obtained with tandem Fabry-Pérot interferometers. (*Not yet implemented*) - **Analyse_TD**: for the analysis of data obtained with time-domain spectrometers.(*Not yet implemented*) - **Analyse_SBS**: for the analysis of data obtained with echelle spectrometers(*Not yet implemented*)

Note for developpers

It is possible to develop new spectrometer classes based on spectrometer classes if the new spectrometers are based on a specific type of existing instrument. For example, the Ultrafast Stimulation-Synchronized Brillouin spectrometer (USS-BM spectrometr) being based on a VIPA spectrometer, it would be possible to create a class **Analyse_USS_BM** inheriting from **Analyse_VIPA**. This would allow to reuse the functions already defined for VIPA spectrometers, and to add functions specific to the USS-BM spectrometer.

For now, only the **Analyse_VIPA** class is stable. It inherits from **Analyse_general**, and is defined for the analysis of data obtained with VIPA spectrometers.

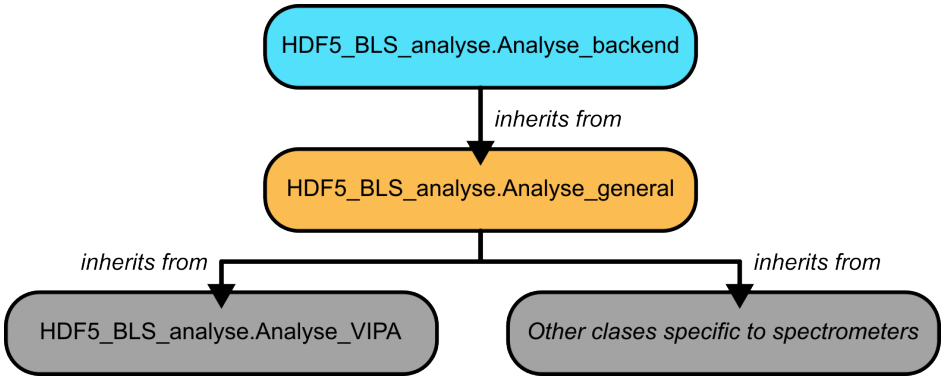


Fig. 7.1: A visual representation of the structure of the *HDF5_BLS_analyse* module.

7.2.2 HDF5_BLS_treat

For the *HDF5_BLS_treat* module, a class storing the models that can be fitted is defined and called **Models**. This class is a standalone class that can be used to define new models or to use these models in custom codes. The class **Treat** is the main class of the module. This class is used to perform the treatment of the data. It inherits from **Treat_backend**. This class allows the user to fit the data to a model and to extract the results of the fit. Finally, the class **TreatmentError** is defined to allow the user to catch errors that might occur during the processing of PSD.

7.3 Usage points common to both modules

The modules share some methods, in particular:

- For opening, saving and editing algorithms
- For running algorithms

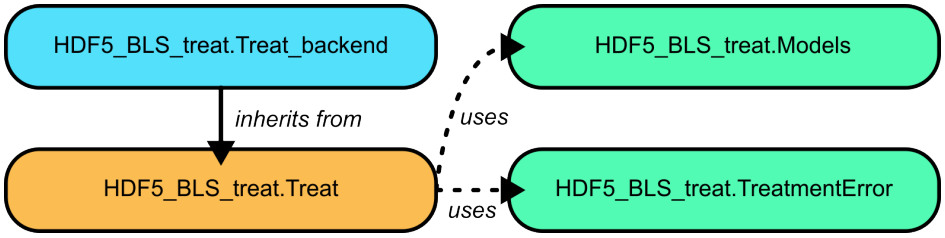


Fig. 7.2: A visual representation of the structure of the *HDF5_BLS_treat* module.

- For adding points of interest

These methods are silent in the sense that they cannot be stored in an algorithm file. Please refer to the API of the modules (see *HDF5_BLS_analyse* and *HDF5_BLS_treat*) for more information.

7.4 Using the HDF5_BLS_analyse module

Using the *HDF5_BLS_analyse* module is done following these steps:

1. Create an object corresponding to the type of spectrometer
2. Define the algorithm to use to perform the analysis (either by creating it or by opening an existing one)
3. Perform the analysis
4. (optional) Save the algorithm to a JSON file

7.4.1 Example with a VIPA spectrometer

The following example shows how to perform the analysis of a VIPA spectrometer. The code is written in Python and assumes that the *HDF5_BLS_analyse* module is installed.

```
from HDF5_BLS_analyse import Analyse_VIPA
import numpy as np

# Initialising the Analyse_VIPA object on the mock spectrum
analyser = Analyse_VIPA(x = np.arange(mock_spectrum.size), y = mock_spectrum)

# Creating a blank algorithm to perform the analysis
analyser.silent_create_algorithm(algorithm_name="VIPA spectrum analyser",
                                 version="v0",
                                 author="Pierre Bouvet",
                                 description="This algorithm allows the user to recover a␣
↪frequency axis basing ourselves on a single Brillouin spectrum obtained with a VIPA␣
↪spectrometer. Considering that only one Brillouin Stokes and anti-Stokes doublet is␣
↪visible on the spectrum, the user can select the peaks he sees, and then perform a␣
↪quadratic interpolation to obtain the frequency axis. This interpolation is obtained␣
↪either by entering a value for the Brillouin shift of the material or by entering the␣
↪value of the Free Spectral Range (FSR) of the spectrometer. The user can finally␣
↪recenter the spectrum either using the average between a Stokes and an anti-Stokes␣
↪peak or by choosing an elastic peak as zero frequency.")

# Adding points corresponding to peaks to the algorithm
analyser.add_point(position_center_window=12, type_pnt="Elastic", window_width=5)
analyser.add_point(position_center_window=37, type_pnt="Anti-Stokes", window_width=5)
```

(continues on next page)

(continued from previous page)

```
analyser.add_point(position_center_window=236, type_pnt="Stokes", window_width=5)
analyser.add_point(position_center_window=259, type_pnt="Elastic", window_width=5)
analyser.add_point(position_center_window=282, type_pnt="Anti-Stokes", window_width=5)
analyser.add_point(position_center_window=466, type_pnt="Stokes", window_width=5)
analyser.add_point(position_center_window=488, type_pnt="Elastic", window_width=5)
analyser.add_point(position_center_window=509, type_pnt="Anti-Stokes", window_width=5)

# Defining the FSR of the VIPA used
analyser.interpolate_elastic_inelastic(FSR = 60)

# Identifying the central peaks and recentering the spectrum to have them be symetric
analyser.add_point(position_center_window=57, type_pnt="Stokes", window_width=1)
analyser.add_point(position_center_window=68.5, type_pnt="Anti-Stokes", window_width=1)
analyser.center_x_axis(center_type = "Inelastic")

# Saving the algorithm to a standalone JSON file
analyser.silent_save_algorithm(filepath = "algorithms/Analysis/VIPA spectrometer/Test.
↪json", save_parameters=True)

# Extracting the PSD and frequency axis from the analyser object
frequency = analyser.x
PSD = analyser.y
```

7.5 Using the HDF5_BLS_treat module

Using the *HDF5_BLS_treat* module is done following these steps:

1. Create an object corresponding to the type of spectrometer
2. Define the algorithm to use to perform the analysis (either by creating it or by opening an existing one)
3. Perform the analysis
4. (optional) Save the algorithm to a JSON file

7.5.1 Example

The following example shows how to perform a simple fit of a Stokes and an Anti-Stokes peak in a PSD.

```
from HDF5_BLS_treat import Treat
import numpy as np

# Initialising the Treat object on the a doublet of frequency and PSD
treat = Treat(frequency = frequency, PSD = data)

# Creating a blank algorithm to perform the analysis
treat.silent_create_algorithm(algorithm_name="VIPA spectrum analyser",
                        version="v0",
                        author="Pierre Bouvet",
                        description="This algorithm allows the user to recover a␣
↪frequency axis basing ourselves on a single Brillouin spectrum obtained with a VIPA␣
↪spectrometer. Considering that only one Brillouin Stokes and anti-Stokes doublet is␣
↪visible on the spectrum, the user can select the peaks he sees, and then perform a␣
```

(continues on next page)

(continued from previous page)

```
↪quadratic interpolation to obtain the frequency axis. This interpolation is obtained␣
↪either by entering a value for the Brillouin shift of the material or by entering the␣
↪value of the Free Spectral Range (FSR) of the spectrometer. The user can finally␣
↪recenter the spectrum either using the average between a Stokes and an anti-Stokes␣
↪peak or by choosing an elastic peak as zero frequency.")

# Adding points corresponding to the central peaks to the algorithm so as to normalize␣
↪the PSD
treat.add_point(position_center_window=-6, type_pnt="Anti-Stokes", window_width=5)
treat.add_point(position_center_window=6, type_pnt="Stokes", window_width=5)
treat.normalize_data(threshold_noise = 0.05)

# Adding the peaks to fit
treat.add_point(position_center_window=-6, type_pnt="Anti-Stokes", window_width=5)
treat.add_point(position_center_window=6, type_pnt="Stokes", window_width=5)

# Defining the model for fitting the peaks
treat.define_model(model="DHO", elastic_correction=False)

# Estimating the linewidth from selected peaks
treat.estimate_width_inelastic_peaks(max_width_guess=5)

# Fitting all the selected inelastic peaks with multiple peaks fitting
treat.single_fit_all_inelastic(guess_offset=True,
                               update_point_position=True,
                               bound_shift=[[-7, -5], [5, 7]],
                               bound_linewidth=[[0, 5], [0, 5]])

# Applying the algorithm to all the spectra (in the case where PSD is a 2D array)
treat.apply_algorithm_on_all()

# Combining the two fitted peaks together here weighing the result on the standard␣
↪deviation of the shift
treat.combine_results_FSR(FSR = 60, keep_max_amplitude = False, amplitude_weight = False,
↪ shift_err_weight= True)

# Extracting the results
shift = treat.shift
linewidth = treat.linewidth
amplitude = treat.amplitude
BLT = treat.BLT
shift_var = treat.shift_var
linewidth_var = treat.linewidth_var
amplitude_var = treat.amplitude_var
BLT_var = treat.BLT_var
```

Part III

Application Programming Interface

CHAPTER

EIGHT

HDF5_BLS

Independent module to access and modify HDF5 files.

HDF5_BLS.Wrapper	This object is used to store data and attributes in a unified structure.
HDF5_BLS.load_data	
HDF5_BLS.errors	

8.1 HDF5_BLS.Wrapper

class HDF5_BLS.**Wrapper**(*filepath=None*)

Bases: `object`

This object is used to store data and attributes in a unified structure.

filepath

The path to the HDF5 file

Type
str

need_for_repack

A flag to check wether elements were deleted in the file using the "del" method. If so, a repacking of the file is needed to optimize memory usage.

Type
bool

save

A flag to check wether the file needs to be saved or not. If the file needs to be saved, it means that the user has worked on a temporary file located in the module directory, that will be deleted when the class is closed.

Type
bool

__init__(*filepath=None*)

Initializes the wrapper. If no filepath is given, a temporary HDF5 file is created in the temporary directory of the operating system. A parent "Brillouin" group is then created and the attribute "HDF5_BLS_version" is set to the current version of the library. If a filepath is given but the file does not exist, it is created. A parent "Brillouin" group is then created and the attribute "HDF5_BLS_version" is set to the current version

of the library. If a filepath is given and the file exists, it is opened and a compatibility check is performed. If the file is not compatible with the current version of the library, a series of changes are applied to make it compatible.

Parameters

filepath (*str, optional*) – The filepath of the HDF5 file to load, by default None means that a temporary file is created in the temporary directory of the operating system.

Example

```
>>> wrp = HDF5_BLS() # Creates a temporary HDF5 file in the temporary directory␣
↪of the operating system
>>> wrp = HDF5_BLS("path/to/file.h5") # Creates a HDF5 file at the given path␣
↪or opens an existing one at the given path
```

Methods

__init__([filepath])	Initializes the wrapper.
add_PSD(data[, parent_group, name, overwrite])	Adds a PSD array to the wrapper by creating a new group.
add_abscissa(data, parent_group[, name, ...])	Adds abscissa as a dataset to the "parent_group" group.
add_attributes(attributes[, parent_group, ...])	Adds attributes to the wrapper.
add_dictionary(dic, parent_group[, ...])	Adds a data dictionary to the wrapper.
add_dictionnary(dic[, parent_group, ...])	Adds a data dictionnary to the wrapper.
add_frequency(data[, parent_group, name, ...])	Adds a frequency array to the wrapper by creating a new group.
add_hdf5(filepath[, parent_group, overwrite])	Adds an HDF5 file to the wrapper by specifying in which group the data have to be stored.
add_other(data[, parent_group, name, overwrite])	Adds an "Other" dataset to the file at the given location.
add_raw_data(data, parent_group[, name, ...])	Adds a raw data array to the wrapper by creating a new group.
add_treated_data(parent_group[, name_group, ...])	Adds the arrays resulting from the treatment of the PSD to the wrapper by creating a new group.
change_brillouin_type(path, brillouin_type)	Changes the brillouin type of an element in the HDF5 file.
change_name(path, name)	Changes the name of an element in the HDF5 file.
clear_empty_attributes(path)	Deletes all the attributes that are empty at the given path.
close([delete_temp_file])	Closes the wrapper and deletes the temporary file if it exists
combine_datasets(datasets, parent_group, name)	Combines a list of elements into a unique dataset.
compatibility_changes()	Applies changes from previous versions of the wrapper to newest versions using the compat module.
copy_dataset(path, copy_path)	This function allows to copy a dataset from the file to a different location while keeping the last location.
create_group(name[, parent_group, ...])	Creates a group in the file under the given parent group with the given name and Brillouin type.
delete_element([path, file])	Deletes an element from the file and sets the need_for_repack flag to True.
export_brim(path_to)	Converts a brimX file to a brim file.

continues on next page

Table 8.2 – continued from previous page

export_dataset(path, filepath[, export_type])	Exports the dataset at the given path as a numpy array.
export_group(path, filepath[, overwrite])	Exports the group at the given path as a HDF5 file.
export_image(path, filepath[, simple_image, ...])	Exports the dataset at the given path as an image.
get_attributes([path])	Returns the attributes associated to a given path.
get_children_elements([path, Brillouin_type])	Returns the children elements of a given path.
get_special_groups_hierarchy([path, ...])	Get all the groups with desired brillouin type that are hierarchically above a given path.
get_structure([filepath])	Returns the structure of an HDF5 file (by default the one stored in the object).
get_type([path, return_Brillouin_type])	Returns the type of the element
import_other(filepath[, parent_group, name, ...])	Adds a raw data array to the wrapper from a file.
import_properties_data(filepath[, path, ...])	Imports properties from an excel or CSV file into a dictionary.
import_raw_data(filepath[, parent_group, ...])	Adds a raw data array to the HDF5 file from a file.
move(path, new_path)	Moves an element from one path to another.
move_channel_dimension_to_last(path[, ...])	Moves the channel dimension to the last dimension of the data to comply with the HDF5_BLS convention.
print_metadata([path])	Prints the metadata of a group or dataset in the console taking into account the hierarchy of the file.
print_structure([lvl])	Prints the structure of the file in the console
repack([force_repack])	Repacks the wrapper to minimize its size.
save_as_hdf5([filepath, remove_old_file, ...])	Saves the data and attributes to an HDF5 file.
save_properties_csv(filepath[, path])	Saves the attributes of the data in the HDF5 file to a CSV file.
store_script([path, attribute_name, ...])	Read the full text of a script, store it on this object at the given path and under the given attribute name.
update_property(name, value, path[, ...])	Updates a property of the HDF5 file given a path to the dataset or group, the name of the property and its value.

Attributes

BRILLOUIN_TYPES_DATASETS
BRILLOUIN_TYPES_GROUPS

BRILLOUIN_TYPES_DATASETS = ['Abscissa', 'Amplitude', 'Amplitude_err', 'BLT', 'BLT_err', 'Frequency', 'Linewidth', 'Linewidth_err', 'Other', 'PSD', 'Raw_data', 'Shift', 'Shift_err']

BRILLOUIN_TYPES_GROUPS = ['Calibration_spectrum', 'Impulse_response', 'Measure', 'Root', 'Treatment']

add_PSD(*data*, *parent_group='Brillouin'*, *name=None*, *overwrite=False*)

Adds a PSD array to the wrapper by creating a new group.

Parameters

- **data** (*np.ndarray*) – The PSD array to add to the wrapper.
- **parent_group** (*str, optional*) – The parent group where to store the data of the HDF5 file. The format of this group should be "Brillouin/Measure".

- **name** (*str, optional*) – The name of the frequency dataset we want to add, and as it will be displayed in the file by any HDF5 viewer. By default the name is "PSD".
- **overwrite** (*bool, optional*) – A parameter to indicate whether the dataset should be overwritten if a dataset with same name already exist or not, by default False - not overwritten.

Raises

WrapperError_StructureError – If the parent group does not exist in the HDF5 file.

add_abscissa(*data*, *parent_group*, *name=None*, *unit='AU'*, *dim_start=0*, *dim_end=None*, *overwrite=False*)

Adds abscissa as a dataset to the "parent_group" group.

Parameters

- **data** (*np.ndarray*) – The array corresponding to the abscissa that is to be addedto the wrapper.
- **parent_group** (*str, optional*) – The parent group where to store the data of the HDF5 file, by default the parent group is the top group "Data". The format of this group should be "Brillouin/Measure".
- **name** (*str, optional*) – The name of that is given to the abscissa dataset. If the name is not specified, it is set to "Abscissa_{dim_start}_{dim_end}"
- **unit** (*str, optional*) – The unit of the abscissa array, by default AU for Arbitrary Units
- **dim_start** (*int, optional*) – The first dimension of the abscissa array, by default 0
- **dim_end** (*int, optional*) – The last dimension of the abscissa array, by default the last number of dimension of the array
- **overwrite** (*bool, optional*) – A parameter to indicate whether the group should be overwritten if they already exist or not, by default False - attributes are not overwritten.

Raises

WrapperError_StructureError – If the parent group does not exist in the HDF5 file.

add_attributes(*attributes*, *parent_group='Brillouin'*, *overwrite=False*)

Adds attributes to the wrapper.

Parameters

- **attributes** (*dict*) – The attributes to add to the wrapper. The keys of the dictionary should be the name of the attributes, and the values should be the values of the attributes.
- **path** (*str, optional*) – The parent group where to store the attributes of the HDF5 file. The format of this group should be "Brillouin/Measure". By default parent_group is set to "Brillouin".
- **overwrite** (*bool, optional*) – If True, the attributes will be overwritten if they already exist.

add_dictionary(*dic*, *parent_group*, *create_group=False*, *brillouin_type_parent_group=None*, *overwrite=False*)

Adds a data dictionary to the wrapper. This is the preferred way to add data using the GUI.

Parameters

- **dic** (*dict*) – The data dictionary to add. The accepted keys for this dictionary are either the one given in the self.BRILLOUIN_TYPES_DATASET list, a key starting with 'Abscissa' or 'Attributes'. All the element of the dictionary are also dictionnaries. Except for attributes, each dictionary has at least two keys: "Name" and "Data". If an abscissa is to

be added, then the keys "Dim_start", "Dim_end" and "Units" need to be populated. For attributes, each key is the name of the attribute, and the value is the value of the attribute, which will automatically be converted to string if it is not a string.

- **parent_group** (*str, optional*) – The path in the file where to store the dataset.
- **create_group** (*bool, optional*) – If set to True, the parent group will be created if it does not exist. If False and the group does not exist, an error will be raised. Default is False.
- **brillouin_type_parent_group** (*str, optional*) – The type of the data group where the data are stored. This argument must be given if a new group is being created. If this argument is given and overwrite is set to True, then the brillouin type of the parent group will be overwritten. Otherwise, the original brillouin type of the parent group will be used if the group already exists.
- **overwrite** (*bool, optional*) – If set to True, any element of the file with a name corresponding to a name given in the dictionary will be overwritten. Similarly any existing argument will be overwritten and Brillouin type will be redefined. Default is False

Raises

- ***WrapperError_StructureError*** – Raises an error if the parent group does not exist in the HDF5 file.
- ***WrapperError_Overwrite*** – Raises an error if the group already exists in the parent group.
- ***WrapperError_ArgumentType*** – Raises an error if arguments given to the function do not match the expected type.

Example

```
>>> wrp = HDF5_BLS() # Creates a temporary HDF5 file in the temporary directory␣
↪of the operating system
>>> dic = {"Raw_data": {"Name": "Raw data", "Data": np.random.random((50, 50,␣
↪512))}}
>>> wrp.add_dictionary(dic, parent_group = "Brillouin/Group", create_group =␣
↪True, brillouin_type_parent_group = "Measure") # Adds the dictionary to the
↪"Brillouin/Group" group (which is here created with Brillouin_type "Measure")
>>> dic = {"PSD": {"Name": "Power Spectral Density", "Data": np.random.
↪random((50, 50, 512))}, "Frequency": {"Name": "Frequency", "Data": np.
↪arange(512)}}
>>> wrp.add_dictionary(dic, parent_group = "Brillouin/Group", create_group =␣
↪True, brillouin_type_parent_group = "Measure") # Adds the PSD and Frequency␣
↪arrays to "Brillouin/Group".
```

add_dictionnary(*dic*, *parent_group=None*, *name_group=None*, *brillouin_type='Measure'*, *overwrite=False*)

Adds a data dictionnary to the wrapper. This is the preferred way to add data using the GUI.

Parameters

- **dic** (*dict*) – The data dictionnary. Support for the following keys: - "Raw_data": the raw data - "PSD": a power spectral density array - "Frequency": a frequency array associated to the power spectral density - "**Abscissa_**...": An abscissa array for the measures where the name is written after the underscore. Each of these keys can either be a numpy array or a dictionnary with two keys: "Name" and "Data". The "Name" key is the name that will be given to the dataset, while the "Data" key is the data itself. The "**Abscissa_**..." keys

are forced to link to a dictionnary with five keys: “Name”, “Data”, “Unit”, “Dim_start”, “Dim_end”. If the abscissa applies to dimension 1 for example, the “Dim_start” key should be set to 1, and the “Dim_end” to 2.

- **parent_group** (*str, optional*) – The path to the parent path, by default None
- **name_group** (*str, optional*) – The name of the data group, by default the name is “Data_i”.
- **brillouin_type** (*str, optional*) – The type of the data group, by default the type is “Measure”. Other possible types are “Calibration_spectrum”, “Impulse_response”, … Please refer to the documentation of the Brillouin software for more information.
- **overwrite** (*bool, optional*) – If set to True, any name in the file corresponding to an element to be added will be overwritten. Default is False

Raises

- *WrapperError_StructureError* – Raises an error if the parent group does not exist in the HDF5 file.
- *WrapperError_Overwrite* – Raises an error if the group already exists in the parent group.
- *WrapperError_ArgumentType* – Raises an error if arguments given to the function do not match the expected type.
- **WrapperError_AttributeError** – Raises an error if the keys of the dictionnary do not match the expected keys.

add_frequency(*data*, *parent_group='Brillouin'*, *name=None*, *overwrite=False*)

Adds a frequency array to the wrapper by creating a new group.

Parameters

- **data** (*np.ndarray*) – The frequency array to add to the wrapper.
- **parent_group** (*str, optional*) – The parent group where to store the data of the HDF5 file. The format of this group should be “Brillouin/Measure”.
- **name** (*str, optional*) – The name of the frequency dataset we want to add, and as it will be displayed in the file by any HDF5 viewer. By default the name is “Frequency”.
- **overwrite** (*bool, optional*) – A parameter to indicate whether the dataset should be overwritten if a dataset with same name already exist or not, by default False - not overwritten.

Raises

WrapperError_StructureError – If the parent group does not exist in the HDF5 file.

add_hdf5(*filepath*, *parent_group='Brillouin'*, *overwrite=False*)

Adds an HDF5 file to the wrapper by specifying in which group the data have to be stored. Default is the “Brillouin” group. If the specified group does not exist, it will be created.

Parameters

- **filepath** (*str*) – The filepath of the hdf5 file to add.
- **parent_group** (*str, optional*) – The parent group where to store the data of the HDF5 file, by default the parent group is the top group “Brillouin”. The format of this group should be “Brillouin/Group/…”. If the parent group does not exist, it will be created.
- **overwrite** (*bool, optional*) – A boolean that indicates whether the data should be overwritten if it already exists, by default False

Raises

- *WrapperError_FileNotFound* – Raises an error if the file could not be found.
- *WrapperError_StructureError* – Raises an error if the parent group does not exist in the HDF5 file.
- *WrapperError_Overwrite* – Raises an error if the group already exists in the parent group.
- *WrapperError* – Raises an error if the hdf5 file could not be added to the main HDF5 file.

Example

```
>>> wrp = HDF5_BLS() # Creates a temporary HDF5 file in the temporary directory␣
↪of the operating system
>>> wrp.add_hdf5("path/to/file.h5", "Brillouin/Group") # Adds the HDF5 file at␣
↪the given path to the "Brillouin/Group" group (which is here created)
```

add_other(*data*, *parent_group='Brillouin'*, *name=None*, *overwrite=False*)

Adds an "Other" dataset to the file at the given location. If the location does not exist, it is created. If the name is not specified, it is set to "Data_i" with i chosen to not overwrite any other dataset. If the name is specified and exists in the file at the given location, the dataset is overwritten if overwrite is set to True, else, a WrapperError_Overwrite is raised.

Parameters

- **data** (*np.ndarray*) – The dataset to add
- **parent_group** (*str, optional*) – The path to the group where to add the dataset, by default "Brillouin"
- **name** (*str, optional*) – The name of the dataset, by default "Data_i" with i chosen to not overwrite any other dataset
- **overwrite** (*bool, optional*) – A flag to overwrite any dataset with the same name, by default False

add_raw_data(*data*, *parent_group*, *name=None*, *overwrite=False*)

Adds a raw data array to the wrapper by creating a new group.

Parameters

- **data** (*np.ndarray*) – The raw data array to add to the wrapper.
- **parent_group** (*str, optional*) – The parent group where to store the data of the HDF5 file. The format of this group should be "Brillouin/Measure".
- **name** (*str, optional*) – The name of the frequency dataset we want to add, and as it will be displayed in the file by any HDF5 viewer. By default the name is "Raw data".
- **overwrite** (*bool, optional*) – A parameter to indicate whether the dataset should be overwritten if a dataset with same name already exist or not, by default False - not overwritten.

Raises

WrapperError_StructureError – If the parent group does not exist in the HDF5 file.

add_treated_data(*parent_group*, *name_group=None*, *overwrite=False*, ***kwargs*)

Adds the arrays resulting from the treatment of the PSD to the wrapper by creating a new group.

Parameters

- **parent_group** (*str, optional*) – The parent group where to store the data of the HDF5 file. The format of this group should be "Brillouin/Measure".
- **name_group** (*str, optional*) – The name of the group that will be created to store the treated data. By default the name is "Treat_i" with i the number of the treatment so that the name is unique.
- **overwrite** (*bool, optional*) – A parameter to indicate whether the dataset should be overwritten if a dataset with same name already exist or not, by default False - not overwritten.
- **shift** (*np.ndarray, optional*) – The shift array to add to the wrapper.
- **linewidth** (*np.ndarray, optional*) – The linewidth array to add to the wrapper.
- **amplitude** (*np.ndarray, optional*) – The amplitude array to add to the wrapper.
- **blt** (*np.ndarray, optional*) – The Loss Tangent array to add to the wrapper.
- **shift_err** (*np.ndarray, optional*) – The shift error array to add to the wrapper.
- **linewidth_err** (*np.ndarray, optional*) – The linewidth error array to add to the wrapper.
- **amplitude_err** (*np.ndarray, optional*) – The amplitude error array to add to the wrapper.
- **blt_std** (*np.ndarray, optional*) – The Loss Tangent error array to add to the wrapper.
- **treat** (*HDF5_BLS_Treat.Treat*) – The treatment object to add to the wrapper. If given, all other keyword arguments are ignored.

Raises

WrapperError_StructureError – If the parent group does not exist in the HDF5 file.

change_brillouin_type(*path*, *brillouin_type*)

Changes the brillouin type of an element in the HDF5 file.

Parameters

- **path** (*str*) – The path to the element to change the brillouin type of.
- **brillouin_type** (*str*) – The new brillouin type of the element.

Raises

- *WrapperError_StructureError* – If the path is not a valid path.
- *WrapperError_ArgumentType* – If the type is not valid

change_name(*path*, *name*)

Changes the name of an element in the HDF5 file.

Parameters

- **path** (*str*) – The path to the element to change the name of.
- **name** (*str*) – The new name of the element.

Raises

WrapperError_StructureError – If the path does not lead to an element.

clear_empty_attributes(*path*)

Deletes all the attributes that are empty at the given path.

Parameters
path (*str*) – The path to the element to delete the attributes from.

close(*delete_temp_file=False*)

Closes the wrapper and deletes the temporary file if it exists

Parameters
delete_temp_file (*bool, optional*) – If True, the temporary file is deleted, by default False

combine_datasets(*datasets*, *parent_group*, *name*, *overwrite=False*)

Combines a list of elements into a unique dataset. All the datasets must have the same shape. They are added into a new dataset where the first dimension is the number of datasets, under the group "parent_group". If the dataset already exists and overwrite is set to True, it is overwritten.

Parameters

- **datasets** (*list of str*) – The list of paths in the file to the datasets to combine
- **name** (*str*) – The name of the new dataset
- **overwrite** (*bool, optional*) – If a dataset with the same name already exists, overwrite it, by default False

compatibility_changes()

Applies changes from previous versions of the wrapper to newest versions using the compat module.

copy_dataset(*path*, *copy_path*)

This function allows to copy a dataset from the file to a different location while keeping the last location.

Parameters

- **path** (*str*) – The path to the dataset to copy.
- **copy_path** (*str*) – The path to the group where the dataset is to be copied to.

Return type
None

create_group(*name*, *parent_group=None*, *brillouin_type='Root'*, *overwrite=False*)

Creates a group in the file under the given parent group with the given name and Brillouin type. If overwrite is set to True, if a group with the same name exists in the selected parent group, the previous element is removed.

Parameters

- **name** (*str*) – The name of the group to create
- **parent_group** (*str, optional*) – The parent group where to create the group, by default the parent group is the top group "Data". The format of this group should be "Brillouin/Data"
- **brillouin_type** (*str, optional*) – The type of the group, by default "Root". Can be "Root", "Measure", "Calibration_spectrum", "Impulse_response", "Treatment", "Metadata"
- **overwrite** (*bool, optional*) – If set to True, any name in the file corresponding to an element to be added will be overwritten. Default is False

Raises
WrapperError – If the group already exists

delete_element(*path=None*, *file=None*)

Deletes an element from the file and sets the need_for_repack flag to True.

Parameters

- **path** (*str*) – The path to the element to delete
- **file** (*h5py.File*) – The file to delete the element from. By default this object is created in the function

Raises

WrapperError – Raises an error if the path does not lead to an element.

export_brim(*path_to: str*)

Converts a brimX file to a brim file.

Parameters

path_to (*str*) – The filepath to the exported Brim file.

export_dataset(*path*, *filepath*, *export_type='.npy'*)

Exports the dataset at the given path as a numpy array.

Parameters

- **path** (*str*) – The path to the dataset to export. Warning: only datasets of 2 or less dimensions can be exported to either .csv or .xlsx formats.
- **filepath** (*str*) – The path to the numpy array to export to.
- **export_type** (*str*) – The type of export to perform (currently supported: ".npy", ".csv", ".xlsx).

Return type

None

export_group(*path*, *filepath*, *overwrite=False*)

Exports the group at the given path as a HDF5 file.

Parameters

- **path** (*str*) – The path to the group to export.
- **filepath** (*str*) – The path to the HDF5 file to export to.
- **overwrite** (*bool*) – A boolean to specify if the file we export to needs to be rewritten if it already exists.

Return type

None

export_image(*path*, *filepath*, *simple_image=True*, *image_size=None*, *cmap='viridis'*, *colorbar=False*, *colorbar_label=None*, *axis=False*, *xlabel=None*, *ylabel=None*)

Exports the dataset at the given path as an image.

Parameters

- **path** (*str*) – The path to the dataset to export.
- **filepath** (*str*) – The path to the image to export to.
- **simple_image** (*bool, optional*) – If set to True, the image is exported as a simple image with grayscale colormap. If false, the image is exported with the given colormap and options.

- **image_size** (*tuple, optional*) – The size of the image to export. If None, the size is set to the default figure size.
- **cmap** (*str, optional*) – The colormap to use for the image. Default is 'viridis'. All the available colormaps can be found here: https://matplotlib.org/stable/tutorials/colors/colormaps.html
- **colorbar** (*bool, optional*) – If set to True, a colorbar is added to the image.
- **axis** (*boolean, optional*) – If set to True, the image is displayed with an extent given by the "MEASURE.Field_Of_View_(X,Y,Z)_(um)" attribute. If set to False, the image is displayed without any extent.

Return type
None

get_attributes(*path=None*)

Returns the attributes associated to a given path. The attributes are retireved hierarchically, meaning that the attributes of all the groups above the given path are also retrieved, and their value is only changed if they are redefined at a lower level.

Parameters
path (*str, optional*) – The path to the data, by default None which means the attributes are read from the root of the file (the Brillouin group).

Returns
attr – The attributes of the data

Return type
dict

get_children_elements(*path=None*, *Brillouin_type=None*)

Returns the children elements of a given path. If Brillouin_type is specified, only the children elements with the given Brillouin_type are returned.

Parameters

- **path** (*str, optional*) – The path to the element, by default None which means the root of the file ("Brillouin" group)
- **Brillouin_type** (*str, optional*) – The type of the element, by default None which means all the elements are returned

Returns
The list of children elements

Return type
list

get_special_groups_hierarchy(*path=None*, *brillouin_type=None*)

Get all the groups with desired brillouin type that are hierarchically above a given path.

Parameters

- **path** (*str, optional*) – The path to the group, by default None which means the root group is used.
- **brillouin_type** (*str, optional*) – The type of the group, by default None which means "Root" is used

Returns
The list of all the groups with desired brillouin type that are hierarchically above a given path.

Return type
list

`get_structure`(*filepath=None*)

Returns the structure of an HDF5 file (by default the one stored in the object).

Parameters
filepath (*str, optional*) – The filepath to the HDF5 file, by default None which means the filepath stored in the object is the one observed.

Returns
The structure of the file with the types of each element in the "Brillouin_type" key.

Return type
dict

Raises
WrapperError_StructureError – Raises an error if one of the elements has no

`get_type`(*path=None*, *return_Brillouin_type=False*)

Returns the type of the element

Parameters
path (*str, optional*) – The path to the element, by default None which means the root of the file ("Brillouin" group)

Returns
The type of the element

Return type
str

`import_other`(*filepath*, *parent_group='Brillouin'*, *name=None*, *creator=None*, *parameters=None*, *reshape=None*, *overwrite=False*)

Adds a raw data array to the wrapper from a file.

Parameters

- **filepath** (*str*) – The filepath to the raw data file to import.
- **parent_group** (*str, optional*) – The parent group where to store the data of the HDF5 file. The format of this group should be "Brillouin/Measure".
- **name** (*str, optional*) – The name of the dataset, by default None.
- **creator** (*str, optional*) – The structure of the file that has to be loaded. If None, a LoadError can be raised.
- **parameters** (*dict, optional*) – The parameters that are to be used to import the data correctly. If None, a LoadError can be raised.
- **reshape** (*tuple, optional*) – The new shape of the array, by default None means that the shape is not changed
- **overwrite** (*bool, optional*) – A parameter to indicate whether the dataset should be overwritten if a dataset with same name already exist or not, by default False - not overwritten.

`import_properties_data`(*filepath*, *path=None*, *overwrite=False*, *delete_child_attributes=False*)

Imports properties from an excel or CSV file into a dictionary.

Parameters

- **filepath** (*str*) – The filepath to the csv storing the properties of the measure. This csv is based on the spreadsheet found in the "spreadsheet" folder of the repository.
- **path** (*str*) – The path to the data in the HDF5 file.
- **overwrite** (*bool, optional*) – A boolean that indicates whether the attributes should be overwritten if they already exist, by default False.
- **delete_child_attributes** (*bool, optional*) – If True, all the attributes of the children elements with same name as the ones to be updated are deleted. Default is False.

import_raw_data(*filepath*, *parent_group='Brillouin'*, *name=None*, *creator=None*, *parameters=None*, *reshape=None*, *overwrite=False*)

Adds a raw data array to the HDF5 file from a file.

Parameters

- **filepath** (*str*) – The filepath to the raw data file to import.
- **parent_group** (*str, optional*) – The parent group where to store the data of the HDF5 file. The format of this group should be "Brillouin/Measure".
- **name** (*str, optional*) – The name of the dataset, by default None.
- **creator** (*str, optional*) – The structure of the file that has to be loaded. If None, a LoadError can be raised.
- **parameters** (*dict, optional*) – The parameters that are to be used to import the data correctly. If None, a LoadError can be raised.
- **reshape** (*tuple, optional*) – The new shape of the array, by default None means that the shape is not changed
- **overwrite** (*bool, optional*) – A parameter to indicate whether the dataset should be overwritten if a dataset with same name already exist or not, by default False - not overwritten.

move(*path*, *new_path*)

Moves an element from one path to another. If the new group does not exist, it is created.

Parameters

- **path** (*str*) – The path to the element to move.
- **new_path** (*str*) – The new path to move the element to.

Raises

WrapperError_StructureError – If the path does not lead to an element.

move_channel_dimension_to_last(*path*, *channel_dimension=None*)

Moves the channel dimension to the last dimension of the data to comply with the HDF5_BLS convention.

Parameters

- **path** (*str*) – The path to the dataset to move the channel dimension to the last dimension.
- **channel_dimension** (*int, optional*) – The dimension of the channel. Default is None, which means the channel dimension is the last dimension.

print_metadata(*path=None*)

Prints the metadata of a group or dataset in the console taking into account the hierarchy of the file.

Parameters

lvl (*int, optional*) – The level of indentation, by default 0.

print_structure(*lvl=0*)

Prints the structure of the file in the console

Parameters
lvl (*int, optional*) – The level of indentation, by default 0.

repack(*force_repack=False*)

Repacks the wrapper to minimize its size.

Parameters
force_repack (*bool*) – Flag to force the repacking of the HDF5 file even if not necessary

save_as_hdf5(*filepath=None*, *remove_old_file=True*, *overwrite=False*)

Saves the data and attributes to an HDF5 file. In practice, moves the temporary hdf5 file to a new location and removes the old file if specified.

Parameters
filepath (*str, optional*) – The filepath where to save the hdf5 file. Default is None, which means the file is saved in the same location as the current file.

Raises

- *WrapperError_Overwrite* – If the file already exists.
- *WrapperError* – Raises an error if the file could not be saved

save_properties_csv(*filepath*, *path=None*)

Saves the attributes of the data in the HDF5 file to a CSV file.

Parameters

- **filepath** (*str*) – The filepath to the csv storing the properties of the measure.
- **path** (*str, optional*) – The path to the data in the HDF5 file, by default None leads to the top group "Brillouin"

store_script(*path: str = None*, *attribute_name: str = None*, *script_filepath: str = None*)

Read the full text of a script, store it on this object at the given path and under the given attribute name.

Parameters

- **path** (*str, optional*) – Path inside the HDF5 file to store the script. If None, the root of the file is used.
- **attribute_name** (*str, optional*) – Name of the attribute to store the script under. If None, the script is stored under the "Script" attribute.
- **script_filepath** (*str, optional*) – Path to the script file to store. If None, the caller's filename is inferred.

Raises

- **RuntimeError** – If the caller filename can't be inferred (interactive shells / notebooks) and no path given.
- **FileNotFoundError / UnicodeDecodeError** – If the script file can't be read.

update_property(*name*, *value*, *path*, *apply_to_all=None*)

Updates a property of the HDF5 file given a path to the dataset or group, the name of the property and its value.

Parameters

- **name** (*str*) – The name of the property to update.

- **value** (*str*) – The value of the property to update.
- **path** (*str*) – The path of the property to update. Defaults to None sets the property at the root level.

8.2 HDF5_BLS.load_data

Functions

load_dat_file(filepath[, creator, ...])	Loads DAT files.
load_general(filepath[, creator, ...])	Loads files based on their extensions
load_image_file(filepath[, parameters, ...])	Loads image files using Pillow
load_npy_file(filepath[, brillouin_type])	Loads npy files
load_sif_file(filepath[, parameters, ...])	Loads npy files

HDF5_BLS.load_data.**load_dat_file**(*filepath*, *creator=None*, *parameters=None*, *brillouin_type=None*)

Loads DAT files. The DAT files that can be read are obtained from the following configurations: - GHOST software (fixed brillouin type: PSD) - Time Domain measures (fixed brillouin type: Raw_data)

Parameters

- **filepath** (*str*) – The filepath to the GHOST file
- **creator** (*str, optional*) – The way this dat file has to be loaded. If None, an error is raised. Possible values are: - "GHOST": the file is assumed to be a GHOST file - "TimeDomain": the file is assumed to be a TimeDomain file
- **brillouin_type** (*str, optional*) – The brillouin type of the file (not relevant for .dat files)

Returns

The dictionary with the data and the attributes of the file stored respectively in the keys "Data" and "Attributes". For time domain files, the dictionary also contains the time vector in the key "Abscissa_dt".

Return type

dict

HDF5_BLS.load_data.**load_general**(*filepath*, *creator=None*, *parameters=None*, *brillouin_type=None*)

Loads files based on their extensions

Parameters

- **filepath** (*str*) – The filepath to the file
- **creator** (*str*) – An argument to specify how the data was created, useful when the extension of the file is not enough to determine the type of data.
- **parameters** (*dict*) – A dictionary containing the parameters to be used to interpret the data, for example when multiple files need to be combined to obtain the dataset to add.
- **brillouin_type** (*str*) – The brillouin type of the dataset to load. Please refer to the documentation of the Brillouin software for the possible values.

Returns

The dictionary created with the given filepath and eventually parameters.

Return type

dict

HDF5_BLS.load_data.**load_image_file**(*filepath*, *parameters=None*, *brillouin_type=None*)

Loads image files using Pillow

Parameters

- **filepath** (*str*) – The filepath to the image
- **parameters** (*dict, optional*) – A dictionary with the parameters to load the data, by default None. Please refer to the Note section of this docstring for more information.
- **brillouin_type** (*str, optional*) – The brillouin type of the file.

Returns

The dictionary with the data and the attributes of the file stored respectively in the keys "Data" and "Attributes"

Return type

dict

Note

Possible parameters are: grayscale: bool, optional

If True, the image is converted to grayscale, by default False

HDF5_BLS.load_data.**load_npy_file**(*filepath*, *brillouin_type=None*)

Loads npy files

Parameters

- **filepath** (*str*) – The filepath to the npy file
- **brillouin_type** (*str, optional*) – The brillouin type of the file.

Returns

The dictionary with the data and the attributes of the file stored respectively in the keys "Data" and "Attributes"

Return type

dict

HDF5_BLS.load_data.**load_sif_file**(*filepath*, *parameters=None*, *brillouin_type=None*)

Loads npy files

Parameters

- **filepath** (*str*) – The filepath to the npy file
- **brillouin_type** (*str, optional*) – The brillouin type of the file. Not relevant for sif files

Returns

The dictionary with the data and the attributes of the file stored respectively in the keys "Data" and "Attributes"

Return type

dict

8.3 HDF5_BLS.errors

Exceptions

WrapperError(msg)	Base class for all the exceptions raised by the Wrapper class.
WrapperError_ArgumentType(msg)	Error raised when an argument is of the wrong type.
WrapperError_FileNotFound(msg)	Error raised when a file does not exist.
WrapperError_Overwrite(msg)	Error raised when an element is to be overwritten.
WrapperError_Save(msg)	Error raised when teh file is not saved or cannot be saved.
WrapperError_StructureError(msg)	Error raised when the expected structure of the file is not correct.

exception HDF5_BLS.errors.**WrapperError**(*msg*)

Bases: Exception

Base class for all the exceptions raised by the Wrapper class.

exception HDF5_BLS.errors.**WrapperError_ArgumentType**(*msg*)

Bases: *WrapperError*

Error raised when an argument is of the wrong type.

exception HDF5_BLS.errors.**WrapperError_FileNotFound**(*msg*)

Bases: *WrapperError*

Error raised when a file does not exist.

exception HDF5_BLS.errors.**WrapperError_Overwrite**(*msg*)

Bases: *WrapperError*

Error raised when an element is to be overwritten.

exception HDF5_BLS.errors.**WrapperError_Save**(*msg*)

Bases: *WrapperError*

Error raised when teh file is not saved or cannot be saved.

exception HDF5_BLS.errors.**WrapperError_StructureError**(*msg*)

Bases: *WrapperError*

Error raised when the expected structure of the file is not correct.

CHAPTER

NINE

HDF5_BLS_ANALYSE

Independent module to extract a Power Spectral Density from a raw data.

HDF5_BLS_analyse.Analyse_backend	This class is the base class for all the analyse classes.
HDF5_BLS_analyse.Analyse_general	This class is a class inherited from the Analyse_backend class used to store steps of analysis that are not specific to a particular type of spectrometer and that are not interesting to show in an algorithm.
HDF5_BLS_analyse.Analyse_VIPA	This class is a child class of Analyse_general.

9.1 HDF5_BLS_analyse.Analyse_backend

class HDF5_BLS_analyse.**Analyse_backend**(*y: ndarray*, *x: ndarray = None*)

Bases: `object`

This class is the base class for all the analyse classes. It provides a common interface for all the classes. Its purpose is to provide the basic silent functions to open, create and save algorithms, and to store the different steps of the analysis and their effects on the data. The philosophy of this class is to rely on 4 attributes that will be changed by the different functions of the class: - x: the x-axis of the data - y: the y-axis of the data - points: a list of remarkable points in the data where each point is a 2-list of the form [position, type] - windows: a list of windows in the data where each window is a 2-list of the form [start, end] And to store the different steps of the analysis and their effects on the data: - _algorithm: a dictionary that stores the name of the algorithm, its version, the author, and a description - _history: a list that stores the evolution of the 4 main attributes of the class with the steps of the analysis. The data is defined by 2 1-D arrays: x and y. Additionally, remarkable points and windows are stored in the points and windows attributes. Algorithm steps are stored in 2 attributes: _algorithm and _history. The _algorithm attribute is a dictionary that stores the name of the algorithm, its version, the author, and a description. The _history attribute is a list that stores the steps of the analysis and their effects on the data. The _execute attribute is a boolean that indicates whether the analysis should be executed or not. It is set to True by default. The _auto_run attribute is a boolean that indicates whether the analysis should be executed automatically or not. It is set to False by default. As a general rule, we encourage developers not to modify any of the underscore-prefixed attributes. These attributes are meant to be used internally by the mother class to run, save, and load the analysis and its history. All the functions of the class are functions with a zero-argument call signature that returns None. This means that the parameters of the methods of the children class need to be kew-word arguments, and that if no value for these arguments are given, the default value of the arguments leads the function to do nothing. This specificality ensures the modulability of the class.

Parameters

- **x** (*np.ndarray*) – The x-axis of the data.
- **y** (*np.ndarray*) – The y-axis of the data.

- **points** (*list of 2-list*) – A list of remarkable points in the data where each point is a 2-list of the form [position, type].
- **windows** (*list of 2-list*) – A list of windows in the data where each window is a 2-list of the form [start, end].
- **_algorithm** (*dict*) – The algorithm used to analyse the data.
- **_history** (*list*) – The history of the analysis.

__init__(*y: ndarray*, *x: ndarray = None*)

Initializes the class with the most basic attributes: the ordinates and abscissa of the data.

Parameters

- **y** (*array*) – The array of ordinates of the data
- **x** (*array, optional*) – The array of abscissa of the data. If None, the abscissa is just the index of the points in ordinates.

Methods

__init__(y[, x])	Initializes the class with the most basic attributes: the ordinates and abscissa of the data.
set_x_y(x, y)	Sets the x and y values of the data
silent_create_algorithm([algorithm_name, ...])	Creates a new JSON algorithm with the given name, version, author and description.
silent_move_step(step, new_step)	Moves a step from one position to another in the _algorithm attribute.
silent_open_algorithm([filepath, algorithm_str])	Opens an existing JSON algorithm and stores it in the _algorithm attribute.
silent_remove_step([step])	Removes the step from the _history attribute of the class.
silent_run_algorithm([step])	Runs the algorithm stored in the _algorithm attribute of the class up to the given step (included).
silent_save_algorithm([filepath, ...])	Saves the algorithm to a JSON file with or without the parameters used.

set_x_y(*x*, *y*)

Sets the x and y values of the data

Parameters

- **x** (*array-like*) – The x values of the data
- **y** (*array-like*) – The y values of the data

silent_create_algorithm(*algorithm_name: str = 'Unnamed Algorithm'*, *version: str = '0.1'*, *author: str = 'Unknown'*, *description: str = ''*)

Creates a new JSON algorithm with the given name, version, author and description. This algorithm is stored in the _algorithm attribute. This function also creates an empty history. for the software.

Parameters

- **algorithm_name** (*str, optional*) – The name of the algorithm, by default “Unnamed Algorithm”
- **version** (*str, optional*) – The version of the algorithm, by default “0.1”

- **author** (*str, optional*) – The author of the algorithm, by default "Unknown"
- **description** (*str, optional*) – The description of the algorithm, by default ""

silent_move_step(*step: int*, *new_step: int*)

Moves a step from one position to another in the _algorithm attribute. Deletes the elements of the _history attribute that are after the moved step (included)

Parameters

- **step** (*int*) – The position of the function to move in the _algorithm attribute.
- **new_step** (*int*) – The new position to move the function to.

silent_open_algorithm(*filepath: str = None*, *algorithm_str: str = ''*)

Opens an existing JSON algorithm and stores it in the _algorithm attribute. This function also creates an empty history.

Parameters

filepath (*str, optional*) – The filepath to the JSON algorithm, by default None

silent_remove_step(*step: int = None*)

Removes the step from the _history attribute of the class. If no step is given, removes the last step.

Parameters

step (*int, optional*) – The number of the function up to which the algorithm has to be run. Default is None, means that the last step is removed.

silent_run_algorithm(*step: int = None*)

Runs the algorithm stored in the _algorithm attribute of the class up to the given step (included). If no step is given, the algorithm is run up to the last step.

Parameters

step (*int, optional*) – The number of the function up to which the algorithm has to be run (included), by default None means that all the steps of the algorithm are run.

silent_save_algorithm(*filepath: str = 'algorithm.json'*, *save_parameters: bool = False*)

Saves the algorithm to a JSON file with or without the parameters used. If the parameters are not saved, their value is set to a default value proper to their type.

Parameters

- **filepath** (*str, optional*) – The filepath to save the algorithm to. Default is "algorithm.json".
- **save_parameters** (*bool, optional*) – Whether to save the parameters of the functions. Default is False.

9.2 HDF5_BLS_analyse.Analyse_general

class HDF5_BLS_analyse.**Analyse_general**(*y*, *x=None*)

Bases: *Analyse_backend*

This class is a class inherited from the Analyse_backend class used to store steps of analysis that are not specific to a particular type of spectrometer and that are not interesting to show in an algorithm. For example, the function to add a remarkable point to the data

__init__(*y*, *x=None*)

Initializes the class with the most basic attributes: the ordinates and abscissa of the data.

Parameters

- **y** (*array*) – The array of ordinates of the data
- **x** (*array, optional*) – The array of abscissa of the data. If None, the abscissa is just the index of the points in ordinates.

Methods

__init__(y[, x])	Initializes the class with the most basic attributes: the ordinates and abscissa of the data.
set_x_y(x, y)	Sets the x and y values of the data
silent_clear_points()	Clears the list of points and the list of windows.
silent_create_algorithm([algorithm_name, ...])	Creates a new JSON algorithm with the given name, version, author and description.
silent_move_step(step, new_step)	Moves a step from one position to another in the _algorithm attribute.
silent_open_algorithm([filepath, algorithm_str])	Opens an existing JSON algorithm and stores it in the _algorithm attribute.
silent_remove_step([step])	Removes the step from the _history attribute of the class.
silent_return_string_algorithm()	Returns a string representation of the algorithm stored in the _algorithm attribute of the class.
silent_run_algorithm([step])	Runs the algorithm stored in the _algorithm attribute of the class up to the given step (included).
silent_save_algorithm([filepath, ...])	Saves the algorithm to a JSON file with or without the parameters used.

silent_clear_points()

Clears the list of points and the list of windows.

silent_return_string_algorithm()

Returns a string representation of the algorithm stored in the _algorithm attribute of the class.

Returns
The string representation of the algorithm.

Return type
str

9.3 HDF5_BLS_analyse.Analyse_VIPA

class HDF5_BLS_analyse.**Analyse_VIPA**(*x*, *y*)

Bases: *Analyse_general*

This class is a child class of Analyse_general. It inherits all the methods of the parent class and adds the functions specific to VIPA spectrometers.

__init__(*x*, *y*)

Initializes the class with the most basic attributes: the ordinates and abscissa of the data.

Parameters

- **y** (*array*) – The array of ordinates of the data
- **x** (*array, optional*) – The array of abscissa of the data. If None, the abscissa is just the index of the points in ordinates.

Methods

__init__(x, y)	Initializes the class with the most basic attributes: the ordinates and abscissa of the data.
add_point([position_center_window, ...])	Adds a single point to the list of points together with a window to the list of windows with its type.
center_x_axis([center_type])	Centers the x axis using the first points stored in the class.
interpolate_between_one_order([FSR])	Creates a frequency axis by using the signal between two elastic peaks included.
interpolate_elastic([FSR])	Uses positions of the elastic peaks on the different orders, to obtain a frequency axis by interpolating the position of the peaks with a quadratic polynomial.
interpolate_elastic_inelastic([shift, FSR])	Uses the elastic peaks, and the positions of the Brillouin peaks on the different orders to obtain a frequency axis by interpolating the position of the peaks with a quadratic polynomial.
interpolate_on_one_order([shift])	Uses positions of one elastic peak and associated Brillouin doublet to obtain a frequency axis by interpolating the position of the peak with a quadratic polynomial.
set_x_y(x, y)	Sets the x and y values of the data
silent_clear_points()	Clears the list of points and the list of windows.
silent_create_algorithm([algorithm_name, ...])	Creates a new JSON algorithm with the given name, version, author and description.
silent_move_step(step, new_step)	Moves a step from one position to another in the _algorithm attribute.
silent_open_algorithm([filepath, algorithm_str])	Opens an existing JSON algorithm and stores it in the _algorithm attribute.
silent_remove_step([step])	Removes the step from the _history attribute of the class.
silent_return_string_algorithm()	Returns a string representation of the algorithm stored in the _algorithm attribute of the class.
silent_run_algorithm([step])	Runs the algorithm stored in the _algorithm attribute of the class up to the given step (included).
silent_save_algorithm([filepath, ...])	Saves the algorithm to a JSON file with or without the parameters used.

add_point(*position_center_window: float = 0*, *window_width: float = 0*, *type_pnt: str = 'Elastic'*)

Adds a single point to the list of points together with a window to the list of windows with its type. Each point is an intensity extremum obtained by fitting a quadratic polynomial to the windowed data. The point is given as a value on the x axis (not a position). The "position_center_window" parameter is the center of the window surrounding the peak. The "window_width" parameter is the width of the window surrounding the peak (full width). The "type_pnt" parameter is the type of the peak. It can be either "Stokes", "Anti-Stokes" or "Elastic".

Parameters

- **position_center_window** (*float*) – A value on the self.x axis corresponding to the center of a window surrounding a peak
- **window** (*float*) – A value on the self.x axis corresponding to the width of a window surrounding a peak

- **type_pnt** (*str*) – The nature of the peak. Must be one of the following: "Stokes", "Anti-Stokes" or "Elastic"

center_x_axis(*center_type: str = None*)

Centers the x axis using the first points stored in the class. The parameter "center_type" is used to determine wether to center the axis using the first elastic peak (center_type = "Elastic") or the average of two Stokes and Anti-Stokes peaks (center_type = "Inelastic").

Parameters

center_type (*str*) – The type of the peak to center the x axis around. Must be either "Elastic" or "Inelastic".

interpolate_between_one_order(*FSR: float = None*)

Creates a frequency axis by using the signal between two elastic peaks included. By imposing that the distance in frequency between two neighboring elastic peaks is one FSR, and that the shift of both stokes and anti-stokes peaks to their respective elastic peak is the same, we can obtain a frequency axis. The user has to enter a value for the FSR to calibrate the frequency axis.

Parameters

FSR (*float*) – The free spectral range of the VIPA spectrometer (in GHz).

interpolate_elastic(*FSR: float = None*)

Uses positions of the elastic peaks on the different orders, to obtain a frequency axis by interpolating the position of the peaks with a quadratic polynomial. The user has to enter a value for the FSR to calibrate the frequency axis.

Parameters

FSR (*float*) – The free spectral range of the VIPA spectrometer (in GHz).

interpolate_elastic_inelastic(*shift: float = None*, *FSR: float = None*)

Uses the elastic peaks, and the positions of the Brillouin peaks on the different orders to obtain a frequency axis by interpolating the position of the peaks with a quadratic polynomial. The user can either enter a value for the shift or the FSR, or both. The shift value is used to calibrate the frequency axis using known values of shifts when using a calibration sample to obtain the frequency axis. The FSR value is used to calibrate the frequency axis using a known values of FSR for the VIPA.

Parameters

- **shift** (*float*) – The shift between the elastic and inelastic peaks (in GHz).
- **FSR** (*float*) – The free spectral range of the VIPA spectrometer (in GHz).

interpolate_on_one_order(*shift: float = 1*)

Uses positions of one elastic peak and associated Brillouin doublet to obtain a frequency axis by interpolating the position of the peak with a quadratic polynomial. The user has to enter a value for the Brillouin shift to calibrate the frequency axis. By default, this value is set to 1 so that the measured frequencies are normalized to the calibration sample.

Parameters

shift (*float*) – The frequency shift between the elastic and inelastic peaks (in GHz). By default 1.

CHAPTER

TEN

HDF5_BLS_TREAT

Independent module to fit the power spectral density to a model.

HDF5_BLS_treat.Treat_backend	This class is the base class for all the treat classes.
HDF5_BLS_treat.Models	This class repertoriates all the models that can be used for the fit.
HDF5_BLS_treat.TreatmentError	
HDF5_BLS_treat.Treat	This class is a class inherited from the Treat_backend class used to define functions to treat the data.

10.1 HDF5_BLS_treat.Treat_backend

class HDF5_BLS_treat.**Treat_backend**(*frequency: ndarray*, *PSD: ndarray*, *frequency_sample_dimension=None*)

Bases: `object`

This class is the base class for all the treat classes. Its purpose is to provide the basic silent functions to open, create and save algorithms, and to store the different steps of the treatment and their effects on the data.

Notes

The philosophy of this class is to rely on 2 fixed attributes:

- frequency: the array of frequency axis corresponding to the PSD
- PSD: the array of power spectral density to be treated

And to update the following attributes: - shift: the shift array obtained after the treatment - shift_var: the array of standard deviation of the shift array obtained after the treatment - linewidth: the linewidth array obtained after the treatment - linewidth_var: the array of standard deviation of the linewidth array obtained after the treatment - amplitude: the amplitude array obtained after the treatment - amplitude_var: the array of standard deviation of the amplitude array obtained after the treatment

The treatment itself is stored in the _algorithm attribute and each change to a classe's attribute is stored in the _history attribute:

- _algorithm: a dictionary that stores the name of the algorithm, its version, the author, and a description
- _record_algorithm: a boolean that indicates whether the algorithm should be recorded or not while running the functions of the class

When a treatment fails or is identified as not well fitted, the class stores the information in the following attributes:

- point_errors: the list of points that are not well fitted

Additionally, the class uses sub-attributes to test the treatment on particular spectra. These sub-attributes are:

- frequency_sample: A 1-D sampled frequency array
- PSD_sample: A 1-D sampled PSD array
- offset_sample: A list of offset values obtained on the sampled PSD array (size of lists corresponds to number of peaks analysed)
- shift_sample: A list of shift values obtained on the sampled PSD array(size of lists corresponds to number of peaks analysed)
- shift_err_sample: A list of the standard deviation on the shift values obtained on the sampled PSD array(size of lists corresponds to number of peaks analysed)
- linewidth_sample: A list of linewidth values obtained on the sampled PSD array(size of lists corresponds to number of peaks analysed)
- linewidth_err_sample: A list of the standard deviation on the linewidth values obtained on the sampled PSD array(size of lists corresponds to number of peaks analysed)
- amplitude_sample: A list of amplitude values obtained on the sampled PSD array (size of lists corresponds to number of peaks analysed)
- amplitude_err_sample: A list of the standard deviation on the amplitude values obtained on the sampled PSD array(size of lists corresponds to number of peaks analysed)

For treating these samples, the class offers an argument to store the steps of the treatment. This is stored in the _history attribute. - _history: a list that stores the evolution of the attributes

To switch the class between the treatment of all the PSD array, sampled PSD arrays or error points, we use the attribute:

- **_treat_selection: a string that directs the treatment towards the whole PSD array, sampled PSD arrays or error points with the following options:**
  - all: the whole PSD array is treated
  - sampled: the sampled PSD arrays are treated
  - errors: the error points are treated

Note that the _history attribute is implemented only if _treat_selection is set to "sampled".

__init__(*frequency: ndarray*, *PSD: ndarray*, *frequency_sample_dimension=None*)

Initializes the class by storing the PSD and frequency arrays. Also initializes the sample sub-arrays using the frequency_sample_dimension parameter.

Parameters

- **frequency** (*np.ndarray*) – An array corresponding to the frequency axis of the PSD
- **PSD** (*np.ndarray*) – An array corresponding to the power spectral density to treat
- **frequency_sample_dimension** (*tuple, optional*) – The tuple leading to the 1-D array to use as frequency axis. By default, the first dimension of the frequency is used.

Methods

__init__(frequency, PSD[, ...])	Initializes the class by storing the PSD and frequency arrays.
silent_clear_points()	Clears the list of points and the list of windows.

continues on next page

Table 10.2 – continued from previous page

silent_create_algorithm([algorithm_name, ...])	Creates a new JSON algorithm with the given name, version, author and description.
silent_move_step(step, new_step)	Moves a step from one position to another in the _algorithm attribute.
silent_open_algorithm([filepath])	Opens an existing JSON algorithm and stores it in the _algorithm attribute.
silent_remove_step([step])	Removes the step from the _history attribute of the class.
silent_return_string_algorithm()	Returns a string representation of the algorithm stored in the _algorithm attribute of the class.
silent_run_algorithm([step, algorithm])	Runs an algorithm.
silent_save_algorithm([filepath, ...])	Saves the algorithm to a JSON file with or without the parameters used.

silent_clear_points()

Clears the list of points and the list of windows.

silent_create_algorithm(*algorithm_name: str = 'Unnamed Algorithm'*, *version: str = '0.1'*, *author: str = 'Unknown'*, *description: str = ''*, *new_algorithm=False*)

Creates a new JSON algorithm with the given name, version, author and description. This algorithm is stored in the _algorithm attribute. This function also creates an empty history. for the software.

Parameters

- **algorithm_name** (*str, optional*) – The name of the algorithm, by default "Unnamed Algorithm"
- **version** (*str, optional*) – The version of the algorithm, by default "0.1"
- **author** (*str, optional*) – The author of the algorithm, by default "Unknown"
- **description** (*str, optional*) – The description of the algorithm, by default ""

silent_move_step(*step: int*, *new_step: int*)

Moves a step from one position to another in the _algorithm attribute. Deletes the elements of the _history attribute that are after the moved step (included)

Parameters

- **step** (*int*) – The position of the function to move in the _algorithm attribute.
- **new_step** (*int*) – The new position to move the function to.

silent_open_algorithm(*filepath: str = None*)

Opens an existing JSON algorithm and stores it in the _algorithm attribute. This function also creates an empty history.

Parameters

filepath (*str, optional*) – The filepath to the JSON algorithm, by default None

silent_remove_step(*step: int = None*)

Removes the step from the _history attribute of the class. If no step is given, removes the last step.

Parameters

step (*int, optional*) – The number of the function up to which the algorithm has to be run. Default is None, means that the last step is removed.

silent_return_string_algorithm()

Returns a string representation of the algorithm stored in the _algorithm attribute of the class.

Returns
The string representation of the algorithm.

Return type
str

silent_run_algorithm(*step: int = None*, *algorithm: dict = None*)

Runs an algorithm. By default, the algorithm run is the one stored in the _algorithm attribute of the class. This function can also run other algorithms if specified. The function runs the algorithm up to the given step (included). If no step is given, the algorithm is run up to the last step (included).

Parameters

- **step** (*int, optional*) – The number of the function up to which the algorithm has to be run (included), by default None means that all the steps of the algorithm are run.
- **algorithm** (*dict, optional*) – The algorithm to be run. If None, the algorithm stored in the _algorithm attribute is used. Default is None.

silent_save_algorithm(*filepath: str = 'algorithm.json'*, *save_parameters: bool = False*)

Saves the algorithm to a JSON file with or without the parameters used. If the parameters are not saved, their value is set to a default value proper to their type.

Parameters

- **filepath** (*str, optional*) – The filepath to save the algorithm to. Default is "algorithm.json".
- **save_parameters** (*bool, optional*) – Whether to save the parameters of the functions. Default is False.

10.2 HDF5_BLS_treat.Models

class HDF5_BLS_treat.**Models**

Bases: object

This class repertoriates all the models that can be used for the fit.

__init__()

Methods

DHO(nu, b, a, nu0, gamma[, IR])	Model of a simple lorentzian lineshape
DHO_elastic(nu, ae, be, a, nu0, gamma[, IR])	Model of a simple lorentzian lineshape
__init__()	

continues on next page

Table 10.3 – continued from previous page

gaussian(nu, b, a, nu0, gamma[, IR])	Model of a simple gaussian lineshape :param nu: The frequency array :type nu: array :param b: The constant offset of the data :type b: float :param a: The amplitude of the peak :type a: float :param nu0: The center position of the function :type nu0: float :param gamma: The linewidth of the function :type gamma: float :param IR: The impulse response of the instrument, by default None :type IR: array, optional
lorentzian(nu, b, a, nu0, gamma[, IR])	Model of a simple lorentzian lineshape
lorentzian_elastic(nu, ae, be, a, nu0, gamma)	Model of a simple lorentzian lineshape

Attributes

models

DHO(*nu*, *b*, *a*, *nu0*, *gamma*, *IR=None*)

Model of a simple lorentzian lineshape

Parameters

- **nu** (*array*) – The frequency array
- **b** (*float*) – The constant offset of the data
- **a** (*float*) – The amplitude of the peak
- **nu0** (*float*) – The center position of the function
- **gamma** (*float*) – The linewidth of the function
- **IR** (*array, optional*) – The impulse response of the instrument, by default None

Returns

The function associated to the given parameters

Return type

function

DHO_elastic(*nu*, *ae*, *be*, *a*, *nu0*, *gamma*, *IR=None*)

Model of a simple lorentzian lineshape

Parameters

- **nu** (*array*) – The frequency array
- **ae** (*float*) – The slope of the first order Taylor expansion of the elastic peak at the position of the peak fitted
- **be** (*float*) – The constant offset of the data
- **a** (*float*) – The amplitude of the peak
- **nu0** (*float*) – The center position of the function
- **gamma** (*float*) – The linewidth of the function
- **IR** (*array, optional*) – The impulse response of the instrument, by default None

Returns
The function associated to the given parameters

Return type
function

`gaussian`(*nu*, *b*, *a*, *nu0*, *gamma*, *IR=None*)

Model of a simple gaussian lineshape :param nu: The frequency array :type nu: array :param b: The constant offset of the data :type b: float :param a: The amplitude of the peak :type a: float :param nu0: The center position of the function :type nu0: float :param gamma: The linewidth of the function :type gamma: float :param IR: The impulse response of the instrument, by default None :type IR: array, optional

Returns
The function associated to the given parameters

Return type
function

`lorentzian`(*nu*, *b*, *a*, *nu0*, *gamma*, *IR=None*)

Model of a simple lorentzian lineshape

Parameters

- **nu** (*array*) – The frequency array
- **b** (*float*) – The constant offset of the data
- **a** (*float*) – The amplitude of the peak
- **nu0** (*float*) – The center position of the function
- **gamma** (*float*) – The linewidth of the function
- **IR** (*array, optional*) – The impulse response of the instrument, by default None

Returns
The function associated to the given parameters

Return type
function

`lorentzian_elastic`(*nu*, *ae*, *be*, *a*, *nu0*, *gamma*, *IR=None*)

Model of a simple lorentzian lineshape

Parameters

- **nu** (*array*) – The frequency array
- **ae** (*float*) – The slope of the first order Taylor expansion of the elastic peak at the position of the peak fitted
- **be** (*float*) – The constant offset of the data
- **a** (*float*) – The amplitude of the peak
- **nu0** (*float*) – The center position of the function
- **gamma** (*float*) – The linewidth of the function
- **IR** (*array, optional*) – The impulse response of the instrument, by default None

Returns
The function associated to the given parameters

Return type
function

```
models = {}
```

10.3 HDF5_BLS_treat.TreatmentError

exception HDF5_BLS_treat.**TreatmentError**(*message*)

Bases: Exception

10.4 HDF5_BLS_treat.Treat

class HDF5_BLS_treat.**Treat**(*frequency*, *PSD*)

Bases: *Treat_backend*

This class is a class inherited from the Treat_backend class used to define functions to treat the data. Each function is meant to perform the minimum of operation so as to give the user a total control over the treatment.

__init__(*frequency*, *PSD*)

Initializes the class by storing the PSD and frequency arrays. Also initializes the sample sub-arrays using the frequency_sample_dimension parameter.

Parameters

- **frequency** (*np.ndarray*) – An array corresponding to the frequency axis of the PSD
- **PSD** (*np.ndarray*) – An array corresponding to the power spectral density to treat
- **frequency_sample_dimension** (*tuple, optional*) – The tuple leading to the 1-D array to use as frequency axis. By default, the first dimension of the frequency is used.

Methods

__init__(frequency, PSD)	Initializes the class by storing the PSD and frequency arrays.
add_point([position_center_window, ...])	Adds a single point to the list of points together with a window to the list of windows with its type.
add_window([position_center_window, ...])	Adds a single window to the list of windows together with the central point (with type "Window") to the list of windows.
adjust_treatment_on_errors([position, ...])	Reapplies the treatment on the point error located at the position "position" with the new parameters "new_parameters".
apply_algorithm_on_all()	Takes all the steps of the algorithm up to the moment this function is called and applies the steps to each individual spectrum in the dataset.
blind_deconvolution([default_width])	Subtracts a constant width to all the linewidth array and recomputes the BLT array.
combine_results_FSR([FSR, ...])	Combines the results of the algorithm to have a value for frequency shift based on a known Free Spectral Range (FSR) value.
define_model([model, elastic_correction])	Defines the model to be used to fit the data.
estimate_width_inelastic_peaks([max_width_	Estimates the full width at half maximum of the inelastic peaks stored in self.points.

continues on next page

Table 10.5 – continued from previous page

fit_all_inelastic_of_curve([default_width, ...])	Fits a lineshape to each of the inelastic peaks using the points stored as Stokes or Anti-Stokes peaks in the points attribute.
mark_errors_BLT([min_BLT, max_BLT])	Marks the points that present a value of BLT above or below given thresholds.
mark_errors_linewidth([min_linewidth, ...])	Marks the points that present a value of linewidth above or below given thresholds.
mark_errors_shift([min_shift, max_shift])	Marks the points that present a value of shift above or below given thresholds.
mark_errors_std_BLT([max_error_BLT_variance])	Marks the points that present a variance of the BLT greater than a certain threshold.
mark_errors_std_linewidth([...])	Marks the points that present a variance of the linewidth greater than a certain threshold.
mark_errors_std_shift([max_error_shift_varianc	Marks the points that present a variance of the shift greater than a certain threshold.
mark_point_error(position)	Forces a point located at the position "position" to be considered as an error.
multi_fit_all_inelastic([default_width, ...])	Fits all inelastically scattered peak as a single curve.
normalize_data([threshold_noise])	Normalizes the data by identifying the regions corresponding to noise, substracting the average of these regions from the data, and dividing by the average of the amplitude of all peaks that are not of type elastic.
silent_clear_points()	Clears the list of points and the list of windows.
silent_create_algorithm([algorithm_name, ...])	Creates a new JSON algorithm with the given name, version, author and description.
silent_move_step(step, new_step)	Moves a step from one position to another in the _algorithm attribute.
silent_open_algorithm([filepath])	Opens an existing JSON algorithm and stores it in the _algorithm attribute.
silent_remove_step([step])	Removes the step from the _history attribute of the class.
silent_return_string_algorithm()	Returns a string representation of the algorithm stored in the _algorithm attribute of the class.
silent_run_algorithm([step, algorithm])	Runs an algorithm.
silent_save_algorithm([filepath, ...])	Saves the algorithm to a JSON file with or without the parameters used.
single_fit_all_inelastic([default_width, ...])	Fits each inelastically scattered peak individually.

add_point(*position_center_window: float = None*, *window_width: float = None*, *type_pnt: str = 'Other'*)

Adds a single point to the list of points together with a window to the list of windows with its type. Each point is an intensity extremum obtained by fitting a quadratic polynomial to the windowed data. The point is given as a value on the frequency axis (not a position on this axis). The "position_center_window" parameter is the center of the window surrounding the peak. The "window_width" parameter is the width of the window surrounding the peak (full width). The "type_pnt" parameter is the type of the peak. It can be either "Stokes", "Anti-Stokes", "Elastic" or "Other".

Parameters

- **position_center_window** (*float*) – A value on the self.x axis corresponding to the center of a window surrounding a peak
- **window_width** (*float or 2-tuple of float*) – A value on the self.x axis corresponding to the width of a window surrounding a peak. If a tuple is given, the first element is the lower bound of the window and the second element is the upper bound given

in absolute value from center point.

- **type_pnt** (*str*) – The nature of the peak. Must be one of the following: "Other","Stokes", "Anti-Stokes" or "Elastic"

add_window(*position_center_window: float = None*, *window_width: float = None*)

Adds a single window to the list of windows together with the central point (with type "Window") to the list of windows. The positions are given as values on the frequency axis (not a position). The "position_center_window" parameter is the center of the window surrounding the peak. The "window_width" parameter is the width of the window surrounding the peak (full width).

Parameters

- **position_center_window** (*float*) – A value on the self.x axis corresponding to the center of a window surrounding a peak
- **window** (*float*) – A value on the self.x axis corresponding to the width of a window surrounding a peak

adjust_treatment_on_errors(*position=None*, *new_parameters=None*)

Reapplies the treatment on the point error located at the position "position" with the new parameters "new_parameters".

Parameters

- **position** (*list, optional*) – The position of the point error to be adjusted. Default is None.
- **new_parameters** (*list of dictionnaries, optional*) – The list of new parameters to be applied to re-run the treatment on the errors. Each element is either None (if we don't change the parameters) or a dictionnary of the parameters to be passed to the function. Default is None, means that all the parameters used earlier are used.

apply_algorithm_on_all()

Takes all the steps of the algorithm up to the moment this function is called and applies the steps to each individual spectrum in the dataset. This function updates the global attributes of the class concerning the shift, the linewidth and the amplitude together with their variance, taking into account error propagation. If a spectrum could not be fitted, its value is set to 0 in the global attributes. All the points where the spectra could not be fitted are marked with the "fit_error_marker" parameter in the global attributes (shift, linewidth, amplitude, shift_var, linewidth_var, amplitude_var) and their coordinates are stored in the "point_error" list. The "point_error_type" attribute is also updated with the type of error returned by the fit function (see scipy.optimize.curve_fit documentation). The function returns the number of spectra that could not be fitted.

blind_deconvolution(*default_width: float = None*)

Subtracts a constant width to all the linewidth array and recomputes the BLT array. If default_width is not specified, the width is estimated by fitting a Lorentzian to the most proeminent elastic peak. If no elastic peak are specified, and default_width is not specified, no deconvolution is performed.

Parameters

default_width (*float, optional*) – The value of the width to subtract to the linewidth array, by default None. If None, the width is estimated by fitting a Lorentzian to the most proeminent elastic peak.

combine_results_FSR(*FSR: float = 15*, *keep_max_amplitude: bool = False*, *amplitude_weight: bool = False*, *shift_err_weight: bool = False*, *position: list = None*)

Combines the results of the algorithm to have a value for frequency shift based on a known Free Spectral Range (FSR) value. The end shift value is obtained by "moving" the peak by a FSR value until the peak is within the [-FSR/2, FSR/2] range. Then the absolute value of the shift is taken as the end shift value.

The combination of the result is done by taking the average of all the values by default. Alternatively, the user can choose to keep the maximum of the amplitude of the peak by setting the "keep_max_amplitude" parameter to True. The user can also choose to weight the shift and linewidth by the amplitude of the peak by setting the "amplitude_weight" parameter to True. Note that in the latter case, the precise knowledge of the frequency axis is a must as averaging slightly uncentered peaks will lead to a wrong result.

Parameters

- **FSR** (*float, optional*) – The Free Spectral Range of the spectrometer, by default 15Ghz
- **keep_max_amplitude** (*bool, optional*) – If True, the maximum of the peak amplitude is stored in the amplitude array. If False, an average of all the amplitudes is stored. Default is False.
- **amplitude_weight** (*bool, optional*) – If True, the amplitude of the spectra is used to weight the shift and linewidth. If set to false, a simple average is performed. Default is False.
- **shift_err_weight** (*bool, optional*) – If True, the inverse of the standard deviation of the shift is used to weight the shift and linewidth. If set to false, a simple average is performed. Default is False.
- **position** (*list, optional*) – The position of the spectrum to be updated in case we combine the sampled results. This is used to update the values of a spectrum that has been re-fitted.

define_model(*model: str = 'Lorentzian'*, *elastic_correction: bool = False*)

Defines the model to be used to fit the data.

Parameters

- **model** (*str, optional*) – The model to be used. The models should match the names of the attribute "models" of the class Models, by default "Lorentzian"
- **elastic_correction** (*bool, optional*) – Whether to correct for the presence of an elastic peak by setting adding a linear function to the model, by default False

estimate_width_inelastic_peaks(*max_width_guess: float = 2*)

Estimates the full width at half maximum of the inelastic peaks stored in self.points. Note that the data is supposed to have a zero offset. The estimation is done by finding the samples of the data closes to half of the peak height.

Parameters

max_width_guess (*float, optional*) – The higher limit to the estimation of the full width. Default is 2 GHz.

fit_all_inelastic_of_curve(*default_width: float = 1*, *guess_offset: bool = False*, *update_point_position: bool = True*, *bound_shift: list = None*, *bound_linewidth: list = None*)

Fits a lineshape to each of the inelastic peaks using the points stored as Stokes or Anti-Stokes peaks in the points attribute. The linewidth can be estimated beforehand using the function estimate_width_inelastic_peaks. If not estimated, a fixed width is used (default_width). The offset can also be guessed or not (guess_offset). In the case the offset is guessed, the minimum of the data on the selected window is used as an initial guess. The position of the peak can also be updated based on the fitted shift if update_point_position is set to True. If set to False, the positions are not updated.

Parameters

- **default_width** (*float, optional*) – If the width has not been estimated, the default width to use, by default 1 GHz

- **guess_offset** (*bool, optional*) – If True, the offset is guessed based on the minimum of the data on the selected window. If false, the data is supposed to be normalized and the offset guess is set to 0, by default False
- **update_point_position** (*bool, optional*) – If True, the position of the peak is updated based on the fitted shift. If False, the position of the peak is not updated, by default True
- **bound_shift** (*list, optional*) – The lower and upper bounds of the shift, by default None means no restrictions are applied
- **bound_linewidth** (*list, optional*) – The lower and upper bounds of the linewidth, by default None means no restrictions are applied

mark_errors_BLT(*min_BLT: float = 0*, *max_BLT: float = 10*)

Marks the points that present a value of BLT above or below given thresholds.

Parameters

- **min_shift** (*float, optional*) – The threshold below which the shift is marked as an error , by default 0GHz
- **max_shift** (*float, optional*) – The threshold above which the shift is marked as an error , by default 10GHz

mark_errors_linewidth(*min_linewidth: float = 0*, *max_linewidth: float = 10*)

Marks the points that present a value of linewidth above or below given thresholds.

Parameters

- **min_linewidth** (*float, optional*) – The threshold below which the linewidth is marked as an error , by default 0GHz
- **max_linewidth** (*float, optional*) – The threshold above which the linewidth is marked as an error , by default 10GHz

mark_errors_shift(*min_shift: float = 0*, *max_shift: float = 10*)

Marks the points that present a value of shift above or below given thresholds.

Parameters

- **min_shift** (*float, optional*) – The threshold below which the shift is marked as an error , by default 0GHz
- **max_shift** (*float, optional*) – The threshold above which the shift is marked as an error , by default 10GHz

mark_errors_std_BLT(*max_error_BLT_variance: float = 0.005*)

Marks the points that present a variance of the BLT greater than a certain threshold.

Parameters

max_error_shift_err (*float, optional*) – The threshold above which the shift is marked as an error , by default 5MHz

mark_errors_std_linewidth(*max_error_linewidth_variance: float = 0.005*)

Marks the points that present a variance of the linewidth greater than a certain threshold.

Parameters

max_error_linewidth_variance (*float, optional*) – The threshold above which the linewidth is marked as an error , by default 5MHz

mark_errors_std_shift(*max_error_shift_variance: float = 0.005*)

Marks the points that present a variance of the shift greater than a certain threshold.

Parameters
max_error_shift_err (*float, optional*) – The threshold above which the shift is marked as an error , by default 5MHz

mark_point_error(*position: list*)

Forces a point located at the position "position" to be considered as an error.

Parameters
position (*list*) – The position of the point error to be marked.

Return type
None

multi_fit_all_inelastic(*default_width: float = 1*, *guess_offset: bool = False*, *update_point_position: bool = True*, *bound_shift: list = None*, *bound_linewidth: list = None*)

Fits all inelastically scattered peak as a single curve. The linewidth of the individual peaks can be estimated beforehand using the function estimate_width_inelastic_peaks. If not estimated, a fixed width is used (default_width). The offset can also be guessed or not (guess_offset). In the case the offset is guessed, the minimum of the data on the window defined as the combination of all the peaks windoes is used as an initial guess. When applying the fit to data acquired successively, it might be interesting to update the initial position of the peak by selecting the last fitted position. This can be done by setting update_point_position to True.

Parameters

- **default_width** (*float, optional*) – If the width has not been estimated, the default width to use, by default 1 GHz
- **guess_offset** (*bool, optional*) – If True, the offset is guessed based on the minimum of the data on the selected window. If false, the data is supposed to be normalized and the offset guess is set to 0, by default False
- **update_point_position** (*bool, optional*) – If True, the position of the peak is updated based on the fitted shift. If False, the position of the peak is not updated, by default True
- **bound_shift** (*list, optional*) – The lower and upper bounds of the shift, by default None means no restrictions are applied
- **bound_linewidth** (*list, optional*) – The lower and upper bounds of the linewidth, by default None means no restrictions are applied

normalize_data(*threshold_noise: float = 0.01*)

Normalizes the data by identifying the regions corresponding to noise, substracting the average of these regions from the data, and dividing by the average of the amplitude of all peaks that are not of type elastic.

Parameters
threshold_noise (*float, optional*) – The threshold for identifying noise regions. This value is the highest possible value for noise relative to the average intensity of the selected peaks, on the PSD when the minimal value of the PSD is brought to 0. Default is 1%

Return type
None

single_fit_all_inelastic(*default_width: float = 1*, *guess_offset: bool = False*, *update_point_position: bool = True*, *bound_shift: list = None*, *bound_linewidth: list = None*)

Fits each inelastically scattered peak individually. The linewidth can be estimated beforehand using the

function estimate_width_inelastic_peaks. If not estimated, a fixed width is used (default_width). The offset can also be guessed or not (guess_offset). In the case the offset is guessed, the minimum of the data on the selected window is used as an initial guess. When applying the fit to data acquired successively, it might be interesting to update the initial position of the peak by selecting the last fitted position. This can be done by setting update_point_position to True.

Parameters

- **default_width** (*float, optional*) – If the width has not been estimated, the default width to use, by default 1 GHz
- **guess_offset** (*bool, optional*) – If True, the offset is guessed based on the minimum of the data on the selected window. If false, the data is supposed to be normalized and the offset guess is set to 0, by default False
- **update_point_position** (*bool, optional*) – If True, the position of the peak is updated based on the fitted shift. If False, the position of the peak is not updated, by default True
- **bound_shift** (*list, optional*) – The lower and upper bounds of the shift, by default None means no restrictions are applied
- **bound_linewidth** (*list, optional*) – The lower and upper bounds of the linewidth, by default None means no restrictions are applied

PYTHON MODULE INDEX

h