HDF5_BLS

Release v1.0.1

Pierre Bouvet

Sep 20, 2025

CONTENTS

About HDF5_BLS

The *HDF5_BLS* project is a Python package allowing users to easily store Brillouin Light Scattering relevant data in a single HDF5 file. The package is designed to integrate in existing Python workflows and to be as easy to use as possible. The package is a solution for unifying the data storage of Brillouin Light Scattering experiments, with three main goals:

- **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.
- **Expandability**: Allow the format to grow with the needs of the community.

CHAPTER

ONE

CONTENTS:

1.1 Quick Start

1.1.1 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

1.1.2 Installation

To install the package, you can use pip:

```
pip install HDF5_BLS
```

1.1.3 Usage

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", 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", name = "Power Spectral Density")
wrp.add_frequency(data = frequency, parent_group = "Brillouin", name = "Frequency")

################################################################################
# Existing code to fit the PSD to extract shift and linewidth arrays
################################################################################
```

(continues on next page)

(continued from previous page)

```
# 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", name_group = "Treat_0", shift = shift,␣
↪linewidth = linewidth)
```

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)
```

1.2 The file format

1.2.1 Basic file structure

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.

1.2.2 Meta-files

The Hierarchical Data Format (HDF5) finds its interest in storage for our community, of different measures. These result in "meta-files" where data corresponding to different experiments can be found. The organization of such a file will follow a structure similar to this one:

Although no rules are imposed on the way to organize the file, we propose to associate a hierarchical level to an hyperparameter that has been varied for the experiment. In the given example:

- The first hierarchical level is associated to the cell line that is observed
- The second hierarchical level is associated to the day of the experiment
- The third hierarchical level is associated to the sample that was measured

Note that the format does not impose any restriction on the names of the groups nor the measures. This choice allows you to create user-friendly files that can be opened with any software that can read HDF5 files (e.g. HDFView, HDFCompass, Fiji, H5Web, Panoply, etc.).

1.2.3 Attributes

Storing the attributes of the data in its metadata

HDF5 file format allows the storage of attributes in the metadata of the groups and datasets. We therefore propose to store all the attributes concerning an experiment in the metadata of its parent 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)
```

Note that the access to the whole list of attributes applying to a group or dataset will be possible with the HDF5_BLS package (see *Wrapper.get_attributes*).

Types of attributes

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. float, int, etc.).

Organization of the attributes

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.

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.

Unification and Versioning of attributes

To unify the name of attributes between laboratories, 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).

This spreadsheet will also be the preferred way to define attributes for the measures and the HDF5_BLS package allows to read and import the attributes directly from this spreadsheet (see *Wrapper.import_properties_data*).

Storing analysis and treatment processes

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,
            ...
        },
```

(continues on next page)

(continued from previous page)

```
        "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.

1.3 The HDF5_BLS package

1.3.1 Installation and presentation

Installation

To install the package, you can either download the source code from the GitHub repository or use pip to install the package. We recommend using pip for users who do not intend on working on the package as it is the easiest way to install the package.

Using pip

To install the package using pip, run the following command:

```
pip install HDF5_BLS
```

Downloading the source code

The source code can be downloaded from the 'GitHub repository <https://github.com/bio-brillouin/HDF5_BLS>'__. 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.

Presentation

The HDF5_BLS library is a Python package meant to interface Python code with a HDF5 file.

The goal of this package is to allow the user to semalessly integrate the proposed standard to their existing code. A detailed description of the package will be given in the later sections of this tutorial. Here is however a quick code example to show the integration of the package in a simple case:

```
##############################################################################
# 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")
```

This package also aims at unifying both the way to extract PSD from raw data and extract Brillouin shift and linewidth from the PSD. We will describe later how to do this, we encourage interested readers to already try and add the above code to their code and see how it works.

Module structure

The HDF5_BLS package is built around the following different modules:

- *wrapper*: This module is used to interact with HDF5 files. It is used to read the data, to write the data and to modify any aspect of the HDF5 file (dataset, groups or attributes).
- *analyze*: This module is used to convert raw data taken from a spectrometer into a physically meaningful Power Spectral Density (PSD) array. This process is done to be reliable

- *treat*: This module is used to extract information from the PSD array, such as the frequency shift and line width of the spectral lines.
- *load_data*: This module is used to import data from any formats of interest. This module is an interface between physical files stored on the PC and the wrapper module. It has been designed to be easily extended to any format of data.

1.3.2 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".

1.3.3 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:

- "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
- "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

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")
```

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

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

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)
```

1.3.4 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.

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)
```

1.3.5 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
```

(continues on next page)

(continued from previous page)

```
    │   └── 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

TWO

API

2.1 HDF5_BLS.wrapper

Functions

is_tempfile(filepath)	

Classes

Wrapper([filepath])	This object is used to store data and attributes in a unified structure.

class HDF5_BLS.wrapper.**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

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=None*, *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.
- **parent_group** (*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.
- **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.
- **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")
```

(continues on next page)

(continued from previous page)

```
>>> 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=None*, *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_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.

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.

Return type

None

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*)

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

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

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=None*, *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=None*, *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"

set_attributes_data(*attributes*, *path=None*, *overwrite=False*)

Sets the attributes of the data in the HDF5 file. If the path leads to a dataset, the attributes are added to the group containing the dataset. If overwrite is False, the attributes are not overwritten if they already exist.

Parameters

- **attributes** (*dict*) – The attributes to be added to the HDF5 file. in the form {"MEASURE.Sample": "Water", ... }
- **path** (*str, optional*) – The path to the dataset or group in the HDF5 file, by default None leads to the top group "Brillouin"
- **overwrite** (*bool, optional*) – A parameter to indicate whether the attributes should be overwritten if they already exist or not, by default False - attributes are not overwritten.

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.

HDF5_BLS.wrapper.**is_tempfile**(*filepath*)

2.2 HDF5_BLS.analyze

Classes

Analyze(y[, x])	This class is the base class for all the analyze classes.
Analyze_VIPA(x, y)	This class is a child class of Analyze_general.
Analyze_general(y[, x])	This class is a class inherited from the Analyze 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.

class HDF5_BLS.analyze.**Analyze**(*y: ndarray*, *x: ndarray = None*)

Bases: `object`

This class is the base class for all the analyze 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 analyze the data.
- **_history** (*list*) – The history of the analysis.

class HDF5_BLS.analyze.**Analyze_VIPA**(*x*, *y*)

Bases: *Analyze_general*

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

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).

class HDF5_BLS.analyze.**Analyze_general**(*y*, *x=None*)

Bases: *Analyze*

This class is a class inherited from the Analyze 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

2.3 HDF5_BLS.treat

Classes

Models()	This class repertoriates all the models that can be used for the fit.
Treat(frequency, PSD)	This class is a class inherited from the Treat_backend class used to define functions to treat the data.
Treat_backend(frequency, PSD[, ...])	This class is the base class for all the treat classes.

Exceptions

TreatmentError(message)	

2.4 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

PYTHON MODULE INDEX

h

INDEX

I

L

M

N

P

R

S

U

W