The HDF5_BLS package and its user interface

Pierre Bouvet

July 17, 2025

Contents

Introduction

Welcome to the interactive UI tutorial. This book will guide you through the steps to create and use the UI effectively.

The HDF5_BLS package is a Python library for handling Brillouin light scattering (BLS) data and converting it into a standardized HDF5 format. The library provides functions to open raw data files, store their properties, convert them into a Power Spectral Density (PSD) and analyze the PSD with a standardized treatment protocol. The library is currently compatible with the following file formats:

- ***.dat** files: spectra returned by the GHOST software or obtained using Time Domain measurements
- ***.tif** files: an image format that can be used to export 2D detector images.
- ***.npy** files: an arbitrary numpy array
- ***.sif** files: image files obtained with Andor cameras

The package comes with a graphical user interface (GUI) that allows users to easily open, edit, and save data. This interface is the preferred way to use the package and the subject of this tutorial. The GUI is currently compatible with the following spectrometers:

- Tandem Fabry-Perot (TFP) spectrometers
- Angle-resolved VIPA (ar-VIPA) spectrometers

Preamble: The project in a nutshell

HDF5 file structure

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.

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:

```
file.h5
+-- Brillouin (group)
|   +-- RWPE1 organoids (group)
|   |   +-- Morphogenesis day 1 (group)
|   |   |   +-- Sample 1 (group)
|   |   |   |   +-- Measure (dataset)
|   |   |   +-- Sample 2 (group)
|   |   |   +-- ...
|   |   +-- Morphogenesis day 2 (group)
|   |   +-- ...
|   +-- H6C7 organoids
|   ...
```

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, etc.).

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,
            ...
        },
        "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.

The BLS_type attribute

To improve user-friendliness of the format for the BioBrillouin community, we propose to add a new attribute: *BLS_type* to the groups and datasets of the file. This attribute will allow the user to quickly differentiate the content of the groups and datasets, and will also allow the automation of certain tasks.

For groups, this attribute can have the following values:

- "Calibration_spectrum": the group contains a calibration spectrum.
- "Impulse_response": the group contains an impulse response function.
- "Measure": the group contains a measure (value by default).
- "Root": used to specify that the group is a structure group that contains other groups.
- "Treatment": the group contains treated data derived from a PSD and frequency arrays located in its parent group.

For datasets, this attribute can have the following values:

- "Abscissa_...": An abscissa array for the measures where the name is written after the underscore.
- "Amplitude": The dataset contains the values of the fitted amplitudes.
- "Amplitude_std": The dataset contains the standard deviation of the fitted amplitudes.
- "BLT": The dataset contains the values of the fitted amplitudes.
- "BLT_std": The dataset contains the standard deviation 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_std": The dataset contains the standard deviation of the fitted linewidths.
- "PSD": A power spectral density array
- "Raw_data": The raw data
- "Shift": The dataset contains the values of the fitted frequency shifts.
- "Shift_std": The dataset contains the standard deviation of the fitted frequency shifts.
- "Other": The dataset contains other data that will not be used by the library.

Example of an abscissa dataset: Let's consider a dataset consisting of 10x10 point mappings of cells in a grid of $10\mu m$ by $10\mu m$. We can store the abscissa in two ways:

- We can store the abscissa as two 1D array of 10 elements each, for axis x and y for example. The array corresponding to the x axis will be stored in the dataset with the attribute "Brillouin_type" set to "Abscissa_0_1". The array corresponding to the y axis will be stored in the dataset with the attribute "Brillouin_type" set to "Abscissa_1_2".

- We can store the abscissa as a 2D array of 10x10 elements. This array will be stored in the dataset with the attribute "Brillouin_type" set to "Abscissa_0_2".

A dedicated function is meant to do this step without the user having to worry about the attribute names.

Note: The "Brillouin_type" attribute is not mandatory, but it is highly recommended to use it to differentiate the content of the groups and datasets. If this parameter is not present, the library will assume that group are "Root" group if they contain other groups and a "Measure" dataset if not, and that datasets are "Other" datasets.

A base example

A single measure that has been treated, together with a calibration spectrum and an impulse response (both stored as a couple of Power Spectra Density and Frequency), will be stored in the following structure (the "Brillouin_type" attribute is indicated in slanted text):

file.h5
+ Brillouin (group, *Root*)
| + IRF (group, *Impulse_response*)
| | + PSD (dataset, *PSD*)
| | + Frequency (dataset, *Frequency*)
| + Calibration Water (group, *Calibration_spectrum*)
| | + PSD (dataset, *PSD*)
| | + Frequency (dataset, *Frequency*)
| + Measure (group, *Measure*)
| | + Raw measure (dataset, *Raw_data*)
| | + PSD (dataset, *PSD*)
| | + Frequency (dataset, *Frequency*)
| | + Treatment (dataset, *Treatment*)
| | | + Shift (dataset, *Shift*)
| | | + Linewidth (dataset, *Linewidth*)
| | | + Shift error (dataset, *Shift_std*)
| | | + Linewidth error (dataset, *Linewidth_std*)

Note: The name of the groups and datasets are left to the user, note that the Brillouin_type attribute allows to unambiguously differentiate each group and dataset.

Other examples can be found in appendix

A unified way to go from Measure to Results

This format is designed to unify the way to store data in the BioBrillouin Community. From there, we propose to unify the treatment and analysis of the data. The storage being independent from the analysis, we propose to create an intermediate step where a Power Spectral Density is obtained.

The treatment of any PSD being essentially the same: a fitting of one or several lineshapes that takes into account the response of the instrument to deconvolve its effect, we further propose to unify the analysis after obtaining the PSD. This results in the following pipeline for any BLS experiment

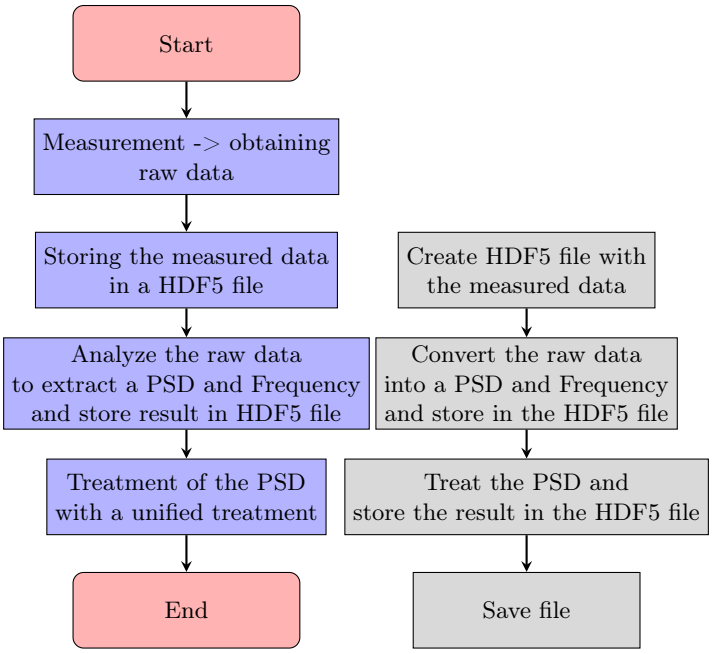


Figure 1: Flowchart of the pipeline

On this figure, the left side represents the steps of the pipeline, while the right side represents the steps that the HDF5_BLS package is designed to perform.

Part I

HDF5_BLS package tutorial

Installation and presentation

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.

0.0.0.0.1 Using pip

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

```
pip install HDF5_BLS
```

0.0.0.0.2 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:

```
pip 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.

0.0.0.0.3 The Wrapper object

The `Wrapper` object is the main class of the package. Its role is to interface the HDF5 file and the code while ensuring a safe access to the data and a conservation of the structure of the file. This class is meant to do these four actions:

- Create a structure that is universal to the BioBrillouin community in a seamless manner.
- Allow the user to acces the data and attributes of the file.
- Allow the user to add data to the file with attributes that are specific to the BioBrillouin community.
- Allow the user to add or modify attributes of the file.

To use this object, the user must first import it from the HDF5_BLS package:

```
from HDF5_BLS.wrapper import Wrapper
```

Chapter 1

The Wrapper module

IN A NUTSHELL:

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

"path/to/file.h5" is the path where you want to store the file or where the file is stored.

To initialize a Wrapper object after having imported the library, you can run the following command:

```
wrp = Wrapper()
```

This will create a new Wrapper object with no attributes or data 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"
```

This file is stored temporarily in the library folder and is deleted 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")
```

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".

Chapter 2

Adding arrays to the HDF5 file using the Wrapper object

IN A NUTSHELL:

- With the add_dictionnary method

```
wrp = Wrapper()
dic = {"Raw_data": {"Name": "Wonderful measure 019",
                    "Data": np.random.random((100, 100))},
       "PSD": {"Name": "PSD extracted blabla",
               "Data": np.random.random((100, 100))},
       "Frequency": {"Name": "The Frequency",
                     "Data": np.random.random((100))},
       "Abscissa_t": {"Name": "Time (s)",
                      "Data": np.random.random((100)),
                      "Unit": "s",
                      "Dim_start": "0",
                      "Dim_end": "1"}}
wrp.add_dictionnary(dic,
                    parent_group = "Brillouin",
                    name = "Data_0",
                    brillouin_type = "Measure",
                    overwrite = False)
```

- With specific methods for each type of data:
  - add_raw_data: To add raw data to a group

```
wrp = Wrapper()
data = np.random.random((10, 10, 512)) # The raw data that you
    want to add
wrp.add_raw_data(data,
                 parent_group = "Brillouin/Data_0",
                 name = "Raw_data")
```

  - add_PSD: To add a PSD to a group

```
wrp = Wrapper()
data = np.random.random((10, 10, 512)) # The PSD that you want to
    add
wrp.add_PSD(data,
            parent_group = "Brillouin/Data_0",
            name = "PSD")
```

– add_frequency: To add a frequency axis to a group

```
wrp = Wrapper()
data = np.random.random((512)) # The frequency axis that you want
    to add
wrp.add_frequency(data,
                  parent_group = "Brillouin/Data_0",
                  name = "Frequency")
```

– add_abscissa: To add an abscissa to a group

```
wrp = Wrapper()
data = np.random.random((10, 10)) # The abscissa that you want to
    add
wrp.add_abscissa(data,
                 parent_group = "Brillouin/Data_0",
                 name = "x and y",
                 unit = "microns",
                 dimension_PSD_start = 0,
                 dimension_PSD_end = 1)
```

– add_treated_data: To add a shift, linewidth and their respective standard deviations to a dedicated "Treatment" group

```
wrp = Wrapper()
shift = np.random.random((10, 10)) # The shift array to add
linewidth = np.random.random((10, 10)) # The linewidth array to
    add
shift_std = np.random.random((10, 10)) # The shift standard
    deviation array to add
linewidth_std = np.random.random((10, 10)) # The linewidth
    standard deviation array to add
wrp.add_treated_data(shift = shift,
                     linewidth = linewidth,
                     shift_std = shift_std,
                     linewidth_std = linewidth_std,
                     parent_group = "Brillouin/Data_0",
                     name_group = "NnMF - 5GHz")
```

2.1 An overview

Adding datasets to the HDF5 file through the HDF5_BLS package is always handled by the Wrapper.add_dictionnary method. This method is complex and not user-friendly. There are therefore other derived methods that are meant to simplify the process of adding data to the HDF5 file, specific to each type of data.

In this tutorial, we present only the derived methods that are specific to each type of data. To get a better understanding on how to use the add_dictionnary method, please refer to the developper guide section.

To add a single dataset to a group, we first need to specify the type of dataset we want to add, following the ones presented in preamble:

- "Abscissa_...": An abscissa array for the measures (time, position, ...)
- "Frequency": A Frequency dataset associated to the PSD dataset
- "PSD": A Power Spectral Density dataset
- "Raw_data": A dataset that is not yet a PSD (for example the measure coming out of a VIPA spectrometer)

- "Shift": A shift array
- "Shift_std": The standard deviation of the shift array
- "Linewidth": A linewidth array
- "Linewidth_std": The standard deviation of the linewidth array

From there, 5 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 standard deviations to a dedicated "Treatment" group

2.2 The logic behind the adding functions

We have decided to separate the addition of a PSD and a Frequency array as in some setups, the Frequency array can be common to a range of experiments, like when a TFP is used.

All these functions have the same approach to the process of adding a dataset to a group:

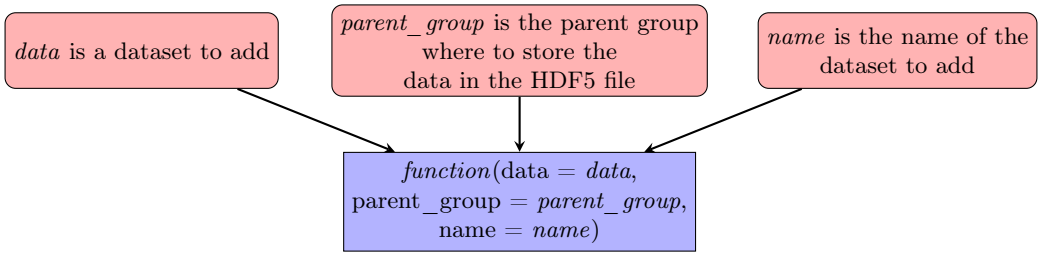


Figure 2.1: Flowchart of the get_attributes method

This approach totally explains the definition of add_raw_data, add_PSD and add_frequency methods. We'll see later how this logic had to be modified to add abscissa and treated data.

2.2.0.0.1 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")
```

2.3 Adding treated data or abscissa

Now for adding abscissa and treated data, we need to improve our approach. For abscissa, we need to be able to specify the dimension of the PSD and treated arrays to which the abscissa corresponds to and the unit of the axis. For treated data, we need to add more than just one dataset to the group.

Let's look at the flowchart describing how to use the add_abscissa method:

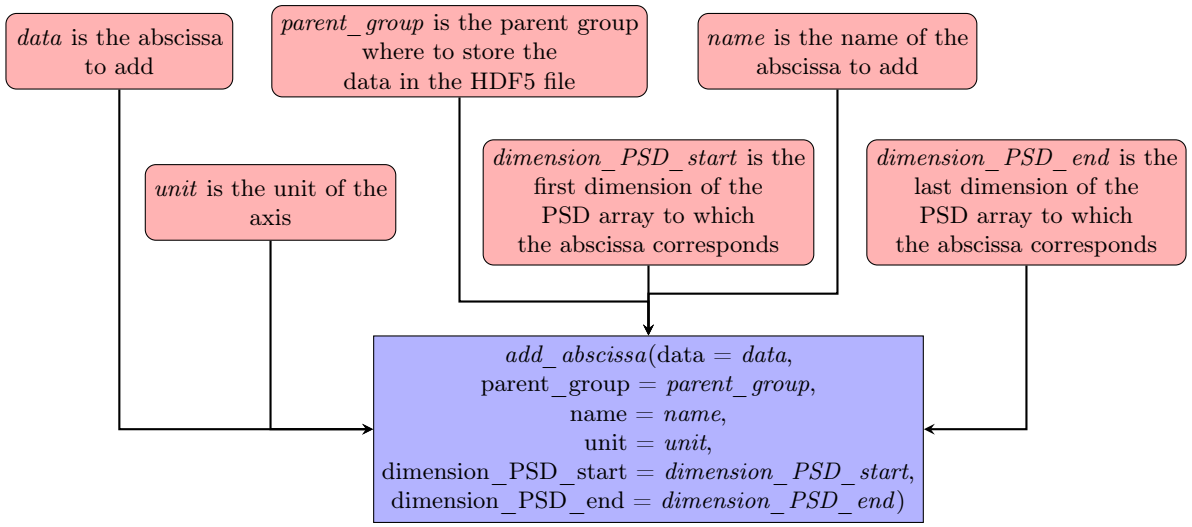


Figure 2.2: Flowchart of the add_abscissa method

2.3.0.0.1 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)
```

Of course if you want to import saved 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)
```

Now, for adding treated data (that is the shift, linewidth and their standard deviations), the flowchart of the add_treated_data method becomes:

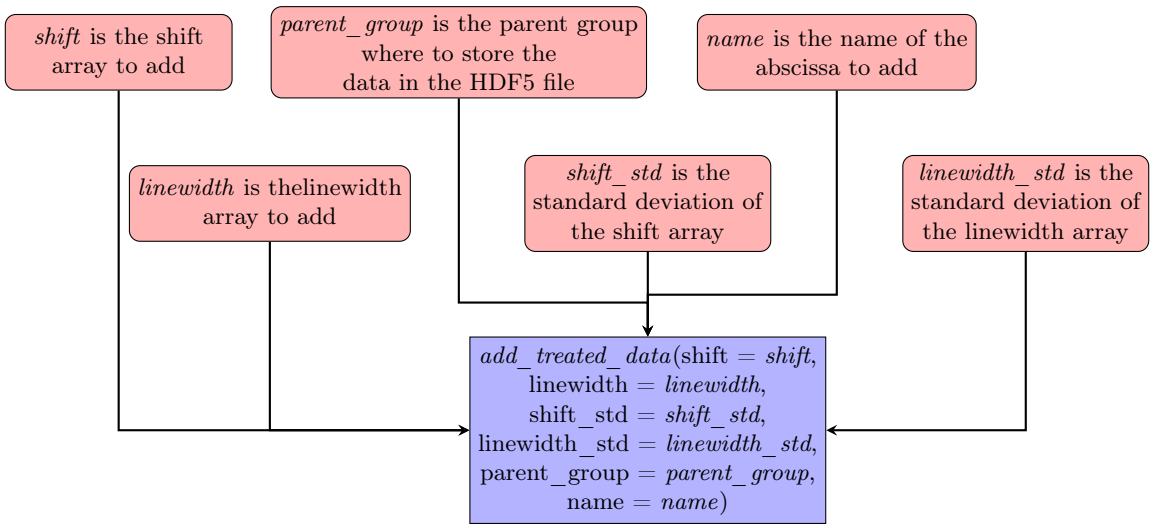


Figure 2.3: Flowchart of the add_abscissa method

2.3.0.0.2 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 standard deviations (shift_std and linewidth_std). 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_std = shift_std,
                     linewidth_std = linewidth_std,
                     parent_group = "Brillouin/Test",
                     name_group = "NnMF - 5GHz")
```

Chapter 3

Importing data to the HDF5 file using the Wrapper object

IN A NUTSHELL:
Importing data from files is done with specific methods for each type of data that is imported:

- Wrapper.import_abscissa: To import an abscissa from a measure file.

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

- Wrapper.import_frequency: To import a frequency array from a measure file.

```
wrp.import_frequency(filepath = "path/to/file.txt", parent_group =
     "Brillouin/Measure", creator = None, parameters = None, name
    = "Frequency", reshape = None, overwrite = False)
```

- Wrapper.import_PSD: To import a Power Spectral Density array from a measure file.

```
wrp.import_PSD(filepath = "path/to/file.txt", parent_group = "
    Brillouin/Measure", creator = None, parameters = None, name =
    "PSD", reshape = None, overwrite = False)
```

- Wrapper.import_raw_data: To import a Raw data array from a measure file.

```
wrp.import_raw_data(filepath = "path/to/file.txt", parent_group =
    "Brillouin/Measure", creator = None, parameters = None, name =
     "PSD", reshape = None, overwrite = False)
```

- Wrapper.import_treated_data: To import the data arrays resulting from a treatment.

```
wrp.import_treated_data(self, filepath_shift, filepath_linewidth,
    filepath_shift_std, filepath_linewidth_std, parent_group,
    creator = None, parameters = None, name = None, reshape = None
    , overwrite = False):
```

Note: Importing the data is done using the load_data module. All the data format and extensions handled by the module are described in the load_data section.

3.1 An overview

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 is done using the load_data module.

2. Adding the data and metadata to the HDF5 file. This is done using the Wrapper.add_dictionnary 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).

3.2 The logic behind the adding functions

Following the same approach as described in the previous chapter, we import data in function of their nature, using dedicated 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.

The logic behind the import functions is the following:

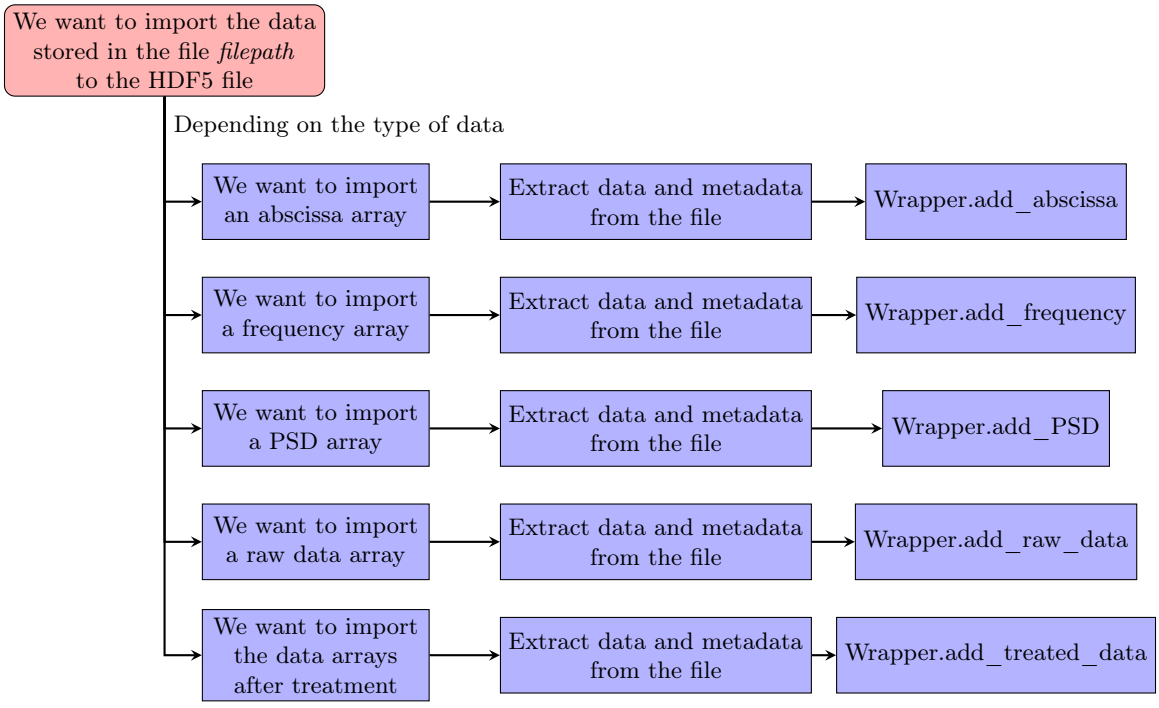


Figure 3.1: Flowchart of the import functions

3.3 The arguments of the import functions

The import functions combine both the arguments needed for the "add_" functions and the arguments needed to extract the data and metadata from the files. These are the following arguments:

- **filepath**: The path to the file containing the data to import. Note that for the "Wrapper.import_treated_data" function, there are 4 different files to import (for the fshift, the linewidth, the shift_std and the linewidth_std arrays).
- **parent_group**: The path to the group or dataset where the data will be added in the form "Brillouin/Measure/...".
- **creator** *(optional, default None)*: Some file extensions are not structured in the same way. Depending on how particular labs store their data, the "creator" argument can be used to specify the structure of the file that has to be loaded. In most cases, this argument is not used and can be left to None. If it however has to be used, a LoadError_creator will be raised.
- **parameters** *(optional, default None)*: The parameters that are to be used to import the data correctly. These parameters are specific to the techniques used to obtain the data. In most cases, this argument is not used and can be left to None. If it however has to be used, a LoadError_parameters will be raised.
- **name** *(optional, default None)*: The name that will be given to the dataset. If None, the name is set to whatever the type of data is (e.g. "Frequency").
- **reshape** *(optional, default None)*: The new shape of the array. If None, the shape is not changed.
- **overwrite** *(optional, default False)*: If True, the attributes of the selected group or dataset are overwritten if they already exist in the file.

Chapter 4

Import or combinine HDF5 files using the Wrapper object

IN A NUTSHELL:
There are two ways to import a HDF5 file into another HDF5 file using the `Wrapper` object:

- **Importing:** In that case, the HDF5 file that is imported is added to the current file in a new group with name the name of the added file. This is done using the add_hdf5 method:

```
wrp = Wrapper()
wrp.add_hdf5(filepath = "path/to/file.h5", parent_group = "
    Brillouin")
```

- **Combining:** We can also combine two HDF5 files into a single one. This means that we create a new HDF5 file where all the groups under the "Brillouin" group of the two files will be stored under the "Brillouin" group of the resulting file. This is done using the __add__ dunder method, or simply with the "+" operator:

```
wrp1 = Wrapper()
wrp2 = Wrapper()
wrp = wrp1 + wrp2
```

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 would therefore be interesting to save it after the addition of the two files with:

```
wrp.save_as_hdf5(filepath = wrp1.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 5

The Treat module

The treat module is the module that allows the user to extract information from a Power Spectrum Density. The module can work with all PSD relying on a 1D frequency vector and where the frequency dependence is the last dimension of the PSD. The module allows a seemless storage of the functions that have been called during a treatment. This allows the user to easily store the treatment but also reapply it to the data if needed. This chapter will present the way the module works then the way it can be used and finally how the module can be extended.

The philosophy of the treat module is to apply functions from the class to treat the data. This choice allows for the class to store any of the algorithm steps that have been applied to the data.

5.1 The treatment module

Opening the "treat" module you can see 4 different classes:

- *Treat_error*
- *Models*
- *Treat_backend*
- *Treat*

Each of these classes has a specific purpose that will be detailed in the following sections.

5.1.1 The *Treat_error* class

The *Treat_error* class is a class that is used to store the errors that can occur during the treatment. It is the default error type when the treatment fails. This class simply raises an exception for now but can be extended to return the type of error, the position of the point that has caused the error in a dataset, or any other relevant information.

5.1.2 The *Models* class

The *Models* class is a class that is used to store the analytical models that can be used to fit the data. These can be both fundamental lineshapes (lorentzian, Gaussian, DHO, Voigt, ...) or custom lineshapes for example if a first order Taylor expansion is used to reduce the effect of the elastic peak on the Brillouin scattered peak.

The class is composed of a series of functions that are used to define the models. These functions are then stored as elements of the *Models.models* dictionnary attribute. The keys of this dictionnary are the names of the models and the values are the functions that define the model. This choice was made to allow the user to easily select a model by simply writing its name.

Note that all models have by default an "IR" parameter. This parameter is meant to be the lineshape of the response of the instrument. If this parameter is given, then the model is numerically convolved with the IR. This allows the user to essentially deconvolve the response of the instrument during the fitting of the data.

5.1.2.1 Signature

For now we have restricted the models to functions that can describe a peak by a central position, a width and a height. The signature of functions where no elastic peak correction is applied is tehrefore the following:

```
def function_name(self, nu, b, a, nu0, gamma, IR = None):
    ...
```

where:

- *self* is the class itself
- *nu* is the frequency array
- *b* is the constant offset of the data
- *a* is the amplitude of the peak
- *nu0* is the center position of the function
- *gamma* is the linewidth of the function
- *IR* is the impulse response of the instrument, by default None

The signature of functions where the elastic peak correction is applied by using a first order Taylor expansion is the following:

```
def function_name(self, nu, ae, be, a, nu0, gamma, IR = None:
    ...
```

where:

- *self* is the class itself
- *nu* is the frequency array
- *ae* is the slope of the first order Taylor expansion of the elastic peak at the position of the peak fitted
- *be* is the constant offset of the data
- *a* is the amplitude of the peak
- *nu0* is the center position of the function
- *gamma* is the linewidth of the function
- *IR* is the impulse response of the instrument, by default None

To maintain a seemless compatibility with all the models, we ask the user to use the following convention:

- The parameter *gamma* is always the full width at half maximum of the peak. Make sure that your model is defined in this way.
- The models are normalized to 1. The parameter *a* is therefore the amplitude of the peak.
- The models have no offset. The parameter *b* (or *be* when elastic correction is applied) is therefore the constant offset of the data.

5.1.2.2 Adding a new model

To add a new model, you need to

- Create a new function that follows the signature of the functions already present in the class
- Add the function to the *Models.models* dictionnary attribute

For example if you want to add a Gaussian model, you would need to create the following function:

```
def gaussian(self, nu, b, a, nu0, gamma, IR = None):
    ...
```

Then you would need to add the function to the *Models.models* dictionnary attribute:

```
Models.models["Gaussian"] = lambda nu, b, a, nu0, gamma, IR=None: self.gaussian(
    nu, b, a, nu0, gamma, IR)
```

We strongly encourage the user to add a docstring to the function. As for the rest of the library, numpy style docstrings are used to document the functions. The docstrings should be written in the following way:

```
def function_name(self, nu, b, a, nu0, gamma, IR = None):
    """
    Brief description of the function

    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
    -------
    function
        The function associated to the given parameters
    """
```

5.1.3 The *Treat_backend* class

The *Treat_backend* class is the base class for the treat class. Its purpose is to provide the background functions to record, open, create and save algorithms, and to store the different steps of the treatment and their effects on the data.

The philosophy of this class is to be a silent spie and handyman to the *Treat* class where the treatment codes are stored. The first job of this class is to act as an intermediary between the calling of the function and their execution so as to store what was called and the arguments that were given to the function. This class also acts as a proxy for the user when a standard algorithm has been selected, by asking for the execution of the standard algorithm. This class has the final job of handling these algorithms, presenting them as JSON files, saving them to external files, displaying them properly, etc. This class is not meant to be directly used by the end user and thus modified by most users.

Therefore we highly recommend that before a change is applied on any of the functions of this class you contact the original developpers of the library to discuss the changes you want to make (if done incorrectly you could break the library).

This class is composed of the following functions:

- **__init__**
  This function initializes the class. This includes both the initialization of the algorithm arguments and the arguments that will store the data, the treatment-related arguments (initial parameters, points of interest, etc.) and the treatment results.

- **__getattribute__**
  This is a redefinition of the native __getattribute__ function of the class. This redefinition is used to store each calling of a function of the class in the _algorithm attribute of the class together with the arguments that were given to the function. This allows the user to easily store the treatment but also reapply it to the data if needed. Note that this function only records calling to functions whose name doesn't start with an underscore. This function also stores the evolutions of the attributes of the class in the _history attribute if the _save_history attribute is set to True. This is particularly usefull if the user wants to go through the steps of the treatment to verify they are correct without having to re-run all the algorithm.

- **_clear_points**
  This function clears the list of points and the list of windows stored in the class.

- **_create_algorithm**
  This function allows the user to specify the name, version, author and description of the algorithm. It also allows the user to create a new algorithm or overwrite an existing one.

- **_move_step**
  This function allows the user to move a step from one position to another in the _algorithm attribute. This function deletes the elements of the _history attribute that are after the moved step (included).

- **_open_algorithm**
  This function allows the user to open an existing JSON algorithm and store it in the _algorithm attribute. This function also creates an empty history.

- **_remove_step**
  This function allows the user to remove a step from the _algorithm attribute. This function also deletes the elements of the _history attribute that are after the removed step (included).

- **_return_string_algorithm**
  This function returns a string representation of the algorithm stored in the _algorithm attribute of the class.

- **_run_algorithm**
  This function allows the user to run the algorithm of the class. It also allows the user to 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).

- **_save_algorithm**
  This function allows the user to save the algorithm to an external 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.

The *Treat_backend* class is meant to be a low-level class that is never used directly by the user except when the user wants to develop his own treatment platform. In most cases, the user will use the *Treat* class instead, a class that inherits from the *Treat_backend* class and that is meant to allow the user to define the functions he needs to perform the treatment and to apply them to the data seemlessly.

5.1.4 The *Treat* class

The *Treat* class is the class that is meant to be used by the user to define the functions he needs to perform the treatment and to apply them to the data seemlessly. This class inherits from the *Treat_backend* class and thus inherits all the functions of the latter.

5.1.4.1 Organization of the class

The *Treat* class is divided in 8 different types of functions:

- *Point and window definition*: functions that are used to define the points and windows of interest on the data. These can then be used to apply correction to the data (like a normalization) or a fitting.
- *Estimation functions*: functions that are used to estimate the parameters of the model.
- *Fitting model definition*: functions that are used to define the model to be used to fit the data.
- *Automatic estimation functions*: functions that are used to automatically estimate the parameters of the model.
- *Fitting functions*: functions that are used to fit the data.
- *Post-treatment functions*: functions that are used to perform post-treatment tasks on the data.
- *Outliers and errors functions*: functions that are used to identify outliers and errors in the data.
- *Algorithm application functions*: functions that are used to apply the algorithm to the data.

5.1.4.2 Return of functions

All the functions of the *Treat* class are silent by default, meaning that the result of the function is not returned and they only modify the attributes of the class. This strategy allows for a reliable tracking of the treatment, its effect on the data and the ability to examine the treatment at any step of the algorithm.

The attributes of the class that are modified by the functions are:

- *_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*: only the PSD array stored in the *PSD_sample* attribute is treated
  - *errors*: only the spectra in the PSD dataset whose position are stored in the *point_error* attribute are treated
- *point_error*: the position of the spectra that lead to an error, in the PSD dataset. This is a list of tuples.
- *point_error_type*: the type of the error (fit error, shift outlier, etc.) corresponding to each point in the *point_error* attribute. This is a list of strings.
- *point_error_value*: the value of the error (NaN if fit error, the shift value if shift outlier, etc.) corresponding to each point in the *point_error* attribute. This is a list of floats.
- *PSD_sample*: the PSD array on which the treatment is performed. This can either be an average of the whole PSD on the frequency axis or any individual PSD of the array.
- *frequency_sample*: the frequency array on which the treatment is performed. This is equivalent to the *frequency* attribute of the class for 1D frequency arrays (the only available option for now).
- *points*: a list of points that are not well fitted
- *windows*: a list of windows that are not well fitted
- *width_estimator*: the estimation of the width of each peak. This is a list of floats with the same length as the *points* attribute.
- *fit_model*: the string corresponding to the model used to fit the data, matching one of the keys of the *Models.models* dictionnary.

- *shift*: the shift array obtained after the treatment
- *linewidth*: the linewidth array obtained after the treatment
- *shift_var*: the array of standard deviation of the shift 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
- *BLT*: the BLT array obtained after the treatment
- *BLT_var*: the array of standard deviation of the BLT array obtained after the treatment

5.1.4.3 Functions signature

Each function defined in the *Treat* class uses key-word only arguments where the type of the argument is specified in the function signature. Defining the type of the arguments allows for an easier integration of the treatment in the GUI. The arguments with type *booleans* for example will be recognized and presented to the user as a checkbox in the GUI.

Using key-word only arguments also ensures that the function can be called without any argument without raising an error. This is particularly usefull to create algorithms without parameters by just calling the functions one after the other. Note that for a safe-by-default definition of the functions, we recommend defining a specific breakpoint at the beginning of the function to check if arguments are given to the function and return nothing if they are not.

To further improve the seamless integration of new functions to the GUI, arguments of the function have a preferred nomenclature. This nomenclature works by using defined prefixes to the arguments to orient the best way to define them in the GUI. The list of prefixes is extendable. In the actual version of the library, these prefixes are:

- *position*: The argument refers to a position on the x axis of the data. The GUI recognizes this parameter and allows the user to click on the graph to select a position.
- *window*: The width of the window around the peak. The GUI recognizes this parameter and creates a text box to enter the value of the window.
- *type_peak*: The type of the peak. Can be either "Stokes", "Anti-Stokes" or "Elastic".
- *center_type*: The type of the peak to center the x axis around. Must be either "Elastic" or "Inelastic".
- *model*: The model to be used to fit the data. The GUI recognizes this parameter and creates a combobox to select the model based on the keys of the *Models.models* dictionnary.

All other arguments will be displayed in the GUI as text boxes.

Note that using the beforementionned nomenclature of the arguments is not mandatory for the function to work.

5.1.4.4 Example of a function

Let's define here a function that applies a Savitsky-Golay filter to the data before fitting. This function is meant to be used before the fitting so we'll define it as a function that is called before the fitting. A Savitsky-Golay filter is a type of filter that is used to smooth the data by fitting a polynomial to the data. The function will be applied on the classe's argument "PSD" takes the following arguments:

- *window*: The length of the window to be used to fit the data.
- *order*: The order of the polynomial to be used to fit the data.

Both of these parameters are integers so we don't need to use a particular prefix for them. We'll give the function initial values to be recognized as the break case. The signature of the function is therefore the following:

```
def savgol_filter(self, window : int = None, order : int = None):
    ...
```

We can now define the break case of the function when called without arguments:

```
def savgol_filter(self, window = None, order = None):
    if window is None or order is None:
        return
    ...
```

Following this step, we can just apply the function to the data. Note that fits are applied to the attribute "PSD_sample", and attribute that is updated when calling "apply_algorithm_on_all". Therefore here, the function will write:

```
def savgol_filter(self, window = None, order = None):
    if window is None or order is None:
        return
    self.PSD_sample = signal.savgol_filter(self.PSD_sample, window_length =
        window, polyorder = order)
```

The last step here is to add the docstring of the function. Here we could for example add the following:

```
def savgol_filter(self, window = None, order = None):
    """
    Applies a Savitsky-Golay filter to the given PSD.

    Parameters
    ----------
    window : int, optional
        The length of the window to be used to fit the data, by default None.
    order : int, optional
        The order of the polynomial to be used to fit the data, by default None.
    """
    if window is None or order is None:
        return
    self.PSD_sample = signal.savgol_filter(self.PSD_sample, window_length =
        window, polyorder = order)
```

5.2 Using the treatment module

The treatment module has the particularity of being able to be used in four different ways:

- To simply use the functions it provides to treat the data
- To define a new treatment algorithm
- To apply an existing treatment algorithm to the data
- To re-use an existing treatment and optimize it

In the following sections, we'll present the choices made for its development and the different ways to use the treatment module.

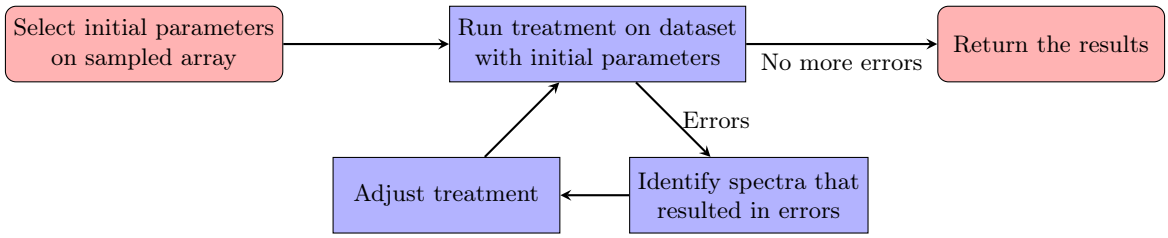
5.2.1 The logic behind the module

The first and main goal of the treatment module is to unify the way users treat their data. This entails an apparent paradox in the way the module has to work. On one hand, the module has to be flexible enough

to allow all the data to be treated in a similar way. On the other hand, the module has to be enforce a repeatable way of treating the data.

To achive these two goals, we ahve decided to think the module as a modulable algorithm maker. This means that the module does not give the user a single function to treat the data. Instead, algorithms combining the standard treatment functions are assembled to create a flexible yet repeatable way of treating the data. The core of the module is therefore to allow the user to either define an algorithm or to apply an existing algorithm to the data.

Defining algorithms can be a tidious task, especially when thousands of data points have to be treated. Adjusting the parameters of the treatment to ensure optimal treatment is also difficult, particularly when only part of the data are not well fitted. To address these issues, the module has been designed to allow the user to re-fit the data with the same algorithm, only with slightly different initial parameters, on the spectra that resulted in errors. Using the treat module is therefore entering in the logic represented by the following flowchart:



In most cases, a "treatment" will not fundamentally change between a run performed on a "good" spectrum and a run performed on a "bad" spectrum that results in an error. It is therefore interesting to allow the user to re-use the steps that were performed during the initial treatment. To allow this treatment of work on the errors, we can just change the parameters of the function, essentially modifying the initial parameters of the treatment.

We have therefore developped a silent background utility that stores the steps of the treatment when running functions. Before going into the details of how to re-use a treatment, let's therefore first see how we can use the functions.

5.2.2 Using the functions of the treatment module

Let's take a practical example to start this section. Consider that you want to run the following simple algorithm:

- Normalize the data to have a noise mean of zero and the amplitude of the Brillouin peaks located at $-5GHz$ to 1.
- Fit two peaks located at $-5GHz$ and $5Ghz$ on a window of width 5 GHz using a Lorentzian model on all the spectra of the dataset.

Using the treat module to apply this algorithm would look like this:

```
1    treat = treat.Treat(freq, psd)
2    treat.add_point(point_position = -5, window_width = 2)
3    treat.normalize_data()
4    treat.add_point(point_position = -5, window_width = 5, type="Anti-Stokes")
5    treat.add_point(point_position = 5, window_width = 5, type="Stokes")
6    treat.define_model(model = "Lorentzian")
7    treat.estimate_width_inelastic_peaks()
8    treat.multi_fit_all_inelastic()
```

where *freq* and *psd* are the frequency and power spectral density arrays of the data.

The results of the treatment will be stored in the attributes of the class. Let's now look at how we can extract these results.

5.2.3 Getting to the results

After the treatment defined earlier on has been run, we'd like to see the results of the treatment. Let's first understand how the results are obtained. Let's say that we want to treat a dataset corresponding to a 2d array of points. Doing so will require giving the treat module two arrays:

- *frequency*: the frequency array of the data (a 1D array)
- *PSD*: the power spectral density array of the data (a 3D array, with the last dimension corresponding to the frequency axis)

Defining the treatment cannot be done on the whole dataset, we need a sample of the dataset to test the treatment on. The module therefore makes a distinction between the whole dataset and the samples that are used to test the treatment. There are therefore two main categories of attributes in the class:

	Sample attributes	**Dataset attributes**
Frequency	frequency_sample	frequency
PSD	PSD_sample	PSD
Shift	shift_sample	shift
Linewidth	linewidth_sample	linewidth
...	...	...

This is really helpfull to allow the same functions to be used on selected points of the dataset, either to treat each point individually, to verify the treatment on a particular point or also to adjust the treatment on a particular point.

The difficulty is passing from the sample attributes to the dataset attributes. The strategy of the module is to work only on the sample attributes, and use combination functions to obtain the dataset attributes.

5.3 Extending the treatment module

Part II

HDF5_BLS GUI tutorial

Chapter 1

GUI quick start guide

To get started, you need to install the repository. Follow the instructions below:

- Step 1: Make sure you have Python 3.10 or higher installed. You can download Python at this link.
- Step 2: Clone the repository at this link.
- Step 3: Create a virtual environment and install the requirements. To do so, open a terminal, navigate to the repository folder and install the requirements. For windows users, you can open the terminal into the cloned and extracted repository (shift+left click over the folder -> Open in terminal) and use the following command:

```
python -m venv venv
.\venv\Scripts\activate
pip install -r requirements.txt
```

  For Mac users, you can navigate to the repository, make sure you can view the path bar at the bottom of Finder (if not, check View/Show Path Bar in the menu bar), then press control and left click on the folder and select "Open in Terminal". Then, use the following command:

```
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

  For Linux users, you can navigate to the repository, open a terminal in the folder and use the same command as for Mac users.

- Step 4: Run the `HDF5_BLS_GUI/main.py` file with

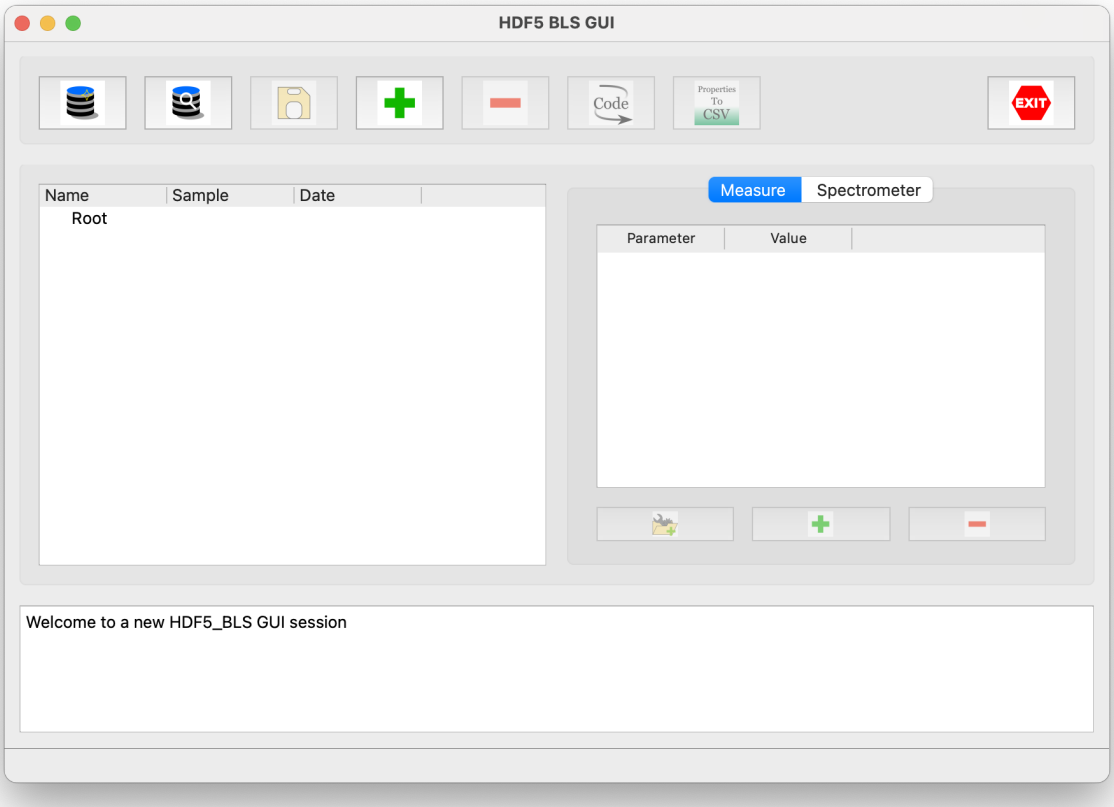
```
python HDF5\_BLS\_GUI/main.py
```

Chapter 2

First Steps

2.1 Creating a new file

After running all these steps, you should see the following window:



You can then drag and drop your data into the left pannel and structure it as you wish.

You can also add properties to your data in the form of a standard CSV file which model can be found in the `spreadsheets` folder of the repository. To add a new property file to your measure, select your measure on the left pannel and drag and drop your property file to the right pannel from a file viewer.

Note that you can add property to a group of data. In that case, the property apply to all its elements.

Part III

Development Guide

Chapter 1

Wrapper

1.1 A little tour of the class

The "Wrapper" class is the main class of the package. Its role is to interface the file and the software while ensuring a user-friendly access to the data and a conservation of the structure of the file. This class is meant to do these four actions:

1. Creates a structure that is universal to the BioBrillouin community in a seamless manner.
2. Allow the user to acces the data and attributes of the file.
3. Allow the user to add data to the file with attributes that are specific to the BioBrillouin community.
4. Allow the user to add or modify attributes of the file.

1.1.0.0.1 Memory management: HDF5 files are notorious for behing heavy. Storing a whole HDF5 file in flash memory is therefore generally a bad idea. As such, the "Wrapper" class works by accessing a file on the disk. By default this file is a temporary file stored in the project's directory. The class is however designed to work on existing HDF5 files stored in permanent locations.

1.1.0.0.2 Private attributes: At initialization:

- **self.filepath**: The path to the HDF5 file. By default, the path to the temporary file is used.
- **self.save**: A boolean that indicates whether the file has been saved or is still in the temporary file.

1.1.0.0.3 Dunders: The "Wrapper" class has the following dunder methods that are described in the following sections:

- Wrapper.__init__ -> Wrapper(): The method that initializes the object
- Wrapper.__getitem__ -> Wrapper[key]: The preferred method to access an element located at a given path, it allows to access an element by placing its path in the brackets of the "Wrapper" object.
- Wrapper.__add__ -> Wrapper + Wrapper: A magic command to merge two wrappers together. It merges the contents of the "Brillouin" group.

1.1.0.0.4 Main Methods: The "Wrapper" class has a number of methods that will be described in the following sections. The construction of the object has been done with a bottleneck strategy, where the critical interactions with the file are done with these methods:

- add_hdf5: A method to populate the file from a HDF5 file.
- add_dictionnary: A method to populate the file from a dictionnary.
- create_group: A method to create a new group inside the HDF5 file.
- delete_element: A method to delete a dataset or group in the HDF5 file.
- get_attributes: A method to extract the attributes of a group or dataset.
- get_structure: A method to extract the structure of the file.
- save_as_hdf5: A method to save the file to a desired location.
- set_attributes_data: A method to set the attributes of a group or dataset.

1.1.0.0.5 Derived Methods: To simplify however the creation and use of the file, these other methods also exist:

- add_abscissa: A method to add an abscissa arrayto the file.
- add_frequency: A method to add a frequency array to the file.
- add_psd: A method to add a PSD to the file.
- add_raw_data: A method to add raw data to the file.
- add_treated_data: A method to add a shift, linewidth and their respective standard deviations to the file.
- import_abscissa: A method to import an abscissa array to the file.
- import_frequency: A method to import a frequency array to the file.
- import_psd: A method to import a PSD array to the file.
- import_raw_data: A method to import raw data to the file.
- import_treated_data: A method to import the data arrays resulting from a treatment.
- import_properties_data: A method to import the attributes of a dataset or group from a spreadsheet.
- update_property: A method to update the attributes of a dataset or group.

1.1.0.0.6 Console specific Methods: Additionnaly, the "Wrapper" class has a few methods specifically to interact with the file from a terminal:

- print_structure: A method to print a tree view of the file in the terminal.
- print_metadata: A method to print all the attributes of a dataset or group at a given path in the terminal.

1.1.0.0.7 Wrapper Errors: The "Wrapper" class has a number of errors that can be raised by its methods:

- **WrapperError_StructureError**: Raised when a problem occurs with the structure of the file.
- **WrapperError_FileNotFound**: Raised when a file stored in the disk is not found.
- **WrapperError_Overwrite**: Raised when a group or dataset already exists in the file.
- **WrapperError_ArgumentType**: Raised when the arguments given to the function are not of the expected type.

1.2 Dunder methods of the Wrapper class

1.2.1 Wrapper.__init__ -> Wrapper()

This magic command allows the user to initialize the wrapper. This dunder method is called with the following syntax:

```
wrp = Wrapper(filepath)
```

1.2.1.0.1 Attributes:

- **filepath**: *optional, default None* The path to the HDF5 file. By default, a temporary file is created in the project's directory.

This method is called automatically when the "Wrapper" object is created. If it has to create a HDF5 file, it automatically creates 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 -> "1.0" # The version of the package
```

If the file already exists, the wrapper object just stores the path to the file and does not overwrite it.

1.2.2 Wrapper.__getitem__ -> Wrapper[key]

This magic command allows the user to access an element located at a given path. This dunder method is called with the following syntax:

```
wrp["Brillouin/Measure/PSD"]
```

Of course, the path can be changed.
If the path leads to a group, the entire group is returned as a closed group. If the path leads to a dataset, the dataset is returned as a numpy array.

1.2.2.0.1 Attributes:

- **path**: The path to the element to access in the form "Brillouin/Measure/PSD".

1.2.2.0.2 Raises:

- **WrapperError_StructureError**: Raises an error if the path does not lead to a valid element in the file.

1.2.3 Wrapper.__add__ -> Wrapper + Wrapper

This magic command allows the user to add two wrappers together. This dunder method is called with the following syntax:

```
new_wrapper = wrp1 + wrp2
```

In this example, wrp1 and wrp2 are two wrappers that are added together. The resulting wrapper is stored in the variable new_wrapper. The addition of the two wrappers is done by adding the elements stored in the "Brillouin" group of both wrappers. The addition is possible only if the two wrappers have the same "HDF5_BLS_version" attribute. If the two wrappers have different "HDF5_BLS_version" attributes, the addition is not possible and an error is raised.

If you prefer to add a second wrapper to a dedicated group of the first wrapper, please use the Wrapper.add_hdf5 method.

The addition of the two wrappers also affects the addition of attributes. Common attributes to all the groups will be set to the root group whereas attributes specific to each file will be set to their respective groups. Access to attributes is guaranteed by the Wrapper.get_attributes method.

1.2.3.0.1 Attributes:

- **wrp2**: A wrapper to add to the current wrapper wrp1

1.2.3.0.2 Raises:

- **WrapperError_FileNotFound**: If one of the two wrappers leads to a temporary file.
- **WrapperError_StructureError**: If the two wrappers don't have the same version.
- **WrapperError_Overwrite**: If the two wrappers share a group of same name.
- **WrapperError**: If an error occured while adding the data.

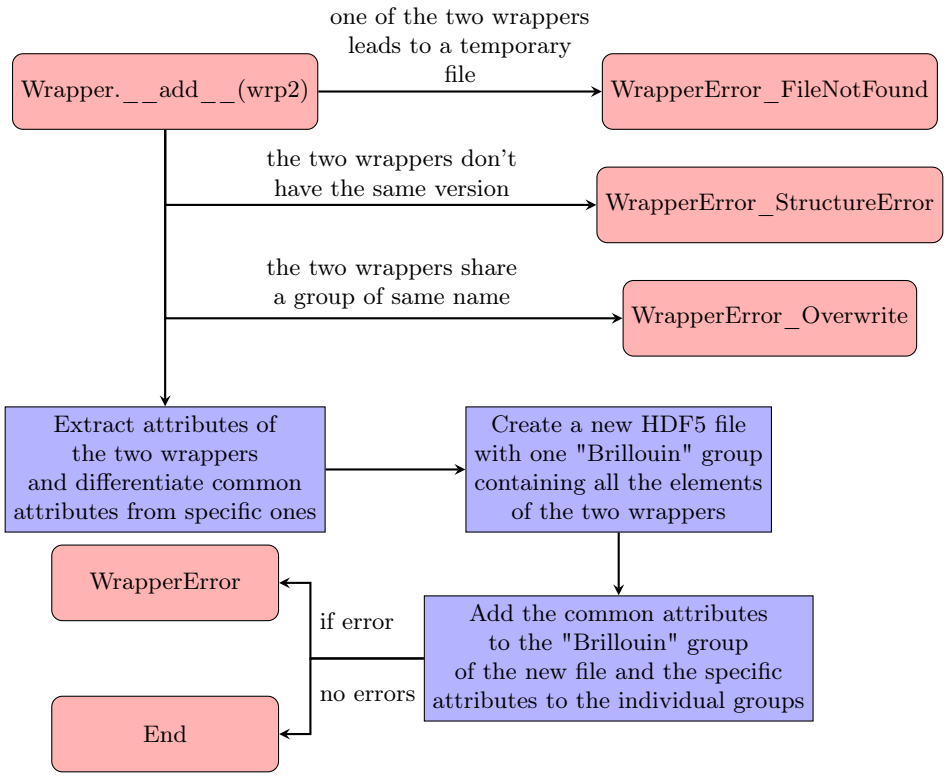


Figure 1.1: Flowchart of the __add__ method

1.3 Principal methods of the Wrapper class

1.3.1 Wrapper.add_hdf5

This method allows the user to add an HDF5 file to the file under a specific group. The group is created if it does not exist. The attributes of the HDF5 file are only added to the created group if they are different from the parent's attribute.

```
1 def add_hdf5(self, filepath, parent_group = None, overwrite = False):
```

1.3.1.0.1 Attributes:

- **filepath**: The path to the HDF5 file to add.
- **parent_group** *(optional, default None)*: 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/Measure/...".
- **overwrite** *(optional, default False)*: If True, the group of same name than the HDF5 file that is added is overwritten. If False, only new attributes are added to the existing ones.

1.3.1.0.2 Returns: Nothing

1.3.1.0.3 Raises:

- **WrapperError_FileNotFound**: Raises an error if the filepath of the HDF5 file does not exist.
- **WrapperError_StructureError**: Raises an error if the parent group does not exist in the HDF5 file.
- **WrapperError_Overwrite**: Raises an error if the file name already exists in the selected group.
- **WrapperError**: Raises an error if the addition of the HDF5 file failed.

1.3.1.0.4 Flowchart: The function's logic is represented in the following flowchart:

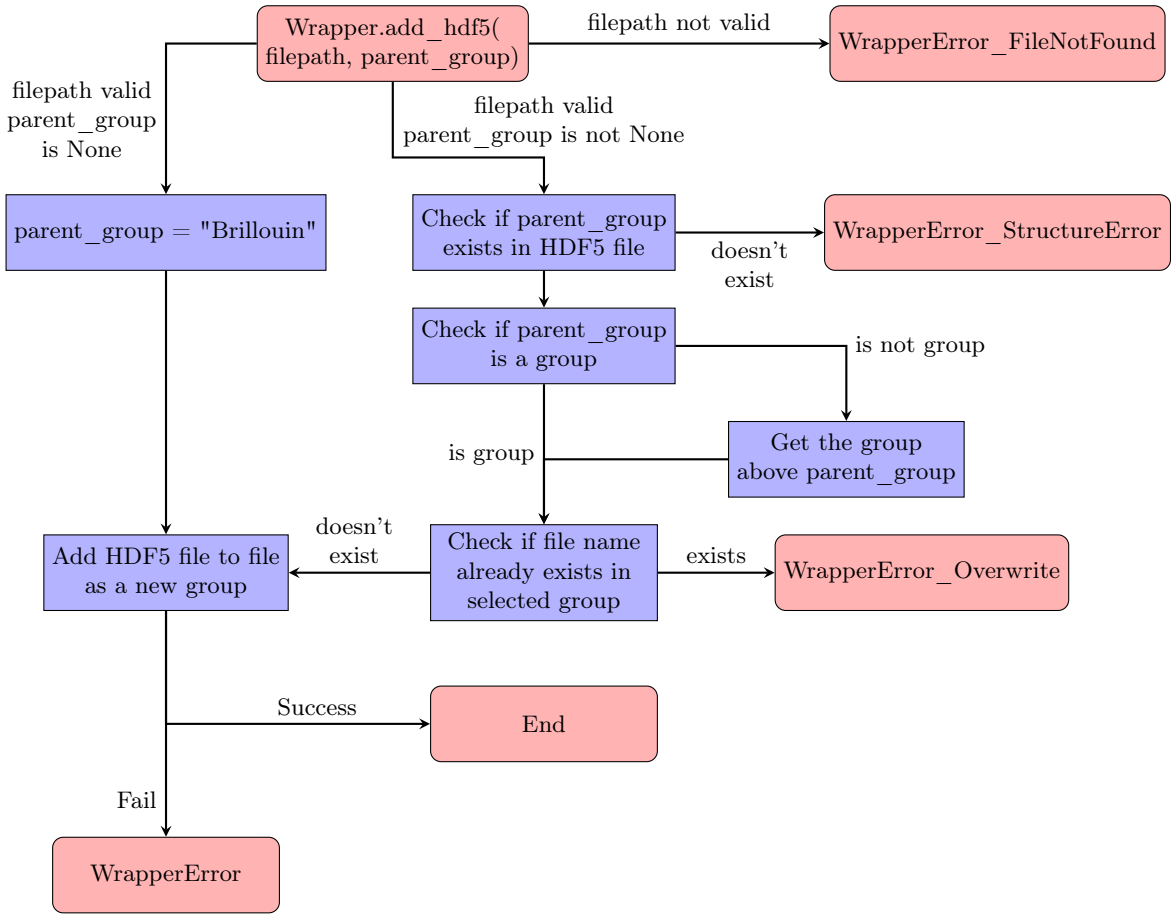


Figure 1.2: Flowchart of the add_hdf5 method

1.3.2 Wrapper.add_dictionnary

This method is the preferred way to add data to the wrapper. It allows the user to create a dictionary with the data and the attributes of the data he wants to add to the file.

```
def add_dictionnary(self, dic, parent_group = None, name_group = None,
    brillouin_type = "Measure", overwrite = False):
```

1.3.2.0.1 Attributes:

- **dic**: The dictionnary to add to the wrapper. This dictionnary has a preferred nomenclature for its keys that will allow the addition of the data with specific attributes to our file format. These keys are identical to the Brillouin_types for dataset defined in preamble. Each element of the dictionnary is another dictionnary with the following keys:
  - "Name": The name of the dataset that will be stored. This can be any name that the user wants and find useful.
  - "Data": The array that will be stored, or the dictionnary if we are adding attributes
  - "Unit": *Specific to "Abscissa_..." keys.* The unit of the abscissa
  - "Dim_start": *Specific to "Abscissa_..." keys.* The first dimension to which the abscissa corresponds.

  - "Dim_end": *Specific to "Abscissa_..." keys.* The last dimension (excluded) to which the abscissa corresponds

- **parent_group** *(optional, default None)*: 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/Measure/...".

- **name_group** *(optional, default None)*: The name of the group where the elements of the dictionnary are stored. If None, the name is set to "Data_i" where "i" is an integer that ensures that the name is unique. If the name already corresponds to a group, then the elements of the dictionnary are added in this group unless elements with same name already exist in the group or adding an element would result in more than one raw data in this group.

- **brillouin_type** *(optional, default "Measure")*: The type of the group that will store the elements of the dictionnary. The possible values are the ones attributed to groups, defined earlier on in the preamble

Here are typical examples of a dictionnary passed as attribute:

```
dic_raw = {
    "Raw_data": {
        "Name": "Raw water spectrum",
        "Data": np.array([...])}
} # a dictionnary with a raw data array straight from the spectrometer - for
    example if you want to do the whole anylisis process with the library.

dic_treated = {
    "Raw_data": {
        "Name": "Raw water spectrum",
        "Data": np.array([...])}
    "PSD": {
        "Name": "PSD water TFP",
        "Data": np.array([...])},
    "Frequency": {
        "Name": "Frequency",
        "Data": np.array([...])}
    "Shift": {
        "Name": "Shift",
        "Data": np.array([...])}
    "Linewidth": {
        "Name": "Shift",
        "Data": np.array([...])}
    "Abscissa_x": {
        "Name": "x",
        "Data": np.array([...]),
        "Unit": "microns",
        "Dim_start": 0,
        "Dim_end": 1}
    "Abscissa_y": {
        "Name": "y",
        "Data": np.array([...]),
        "Unit": "microns",
        "Dim_start": 1,
        "Dim_end": 2},
    "Attributes": {"MEASURE.Sample": "Water",
                   "SPECTROMETER.Type": "TFP"}
} # a dictionnary of measures analyzed by a custom treatment - for example if
    you have your own data and just want to send them following the library's
    format.
```

1.3.2.0.2 Returns: Nothing

1.3.2.0.3 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 arguments given to the function do not match the expected type.

1.3.2.0.4 Flowchart:

The function's logic is represented in the following flowchart:

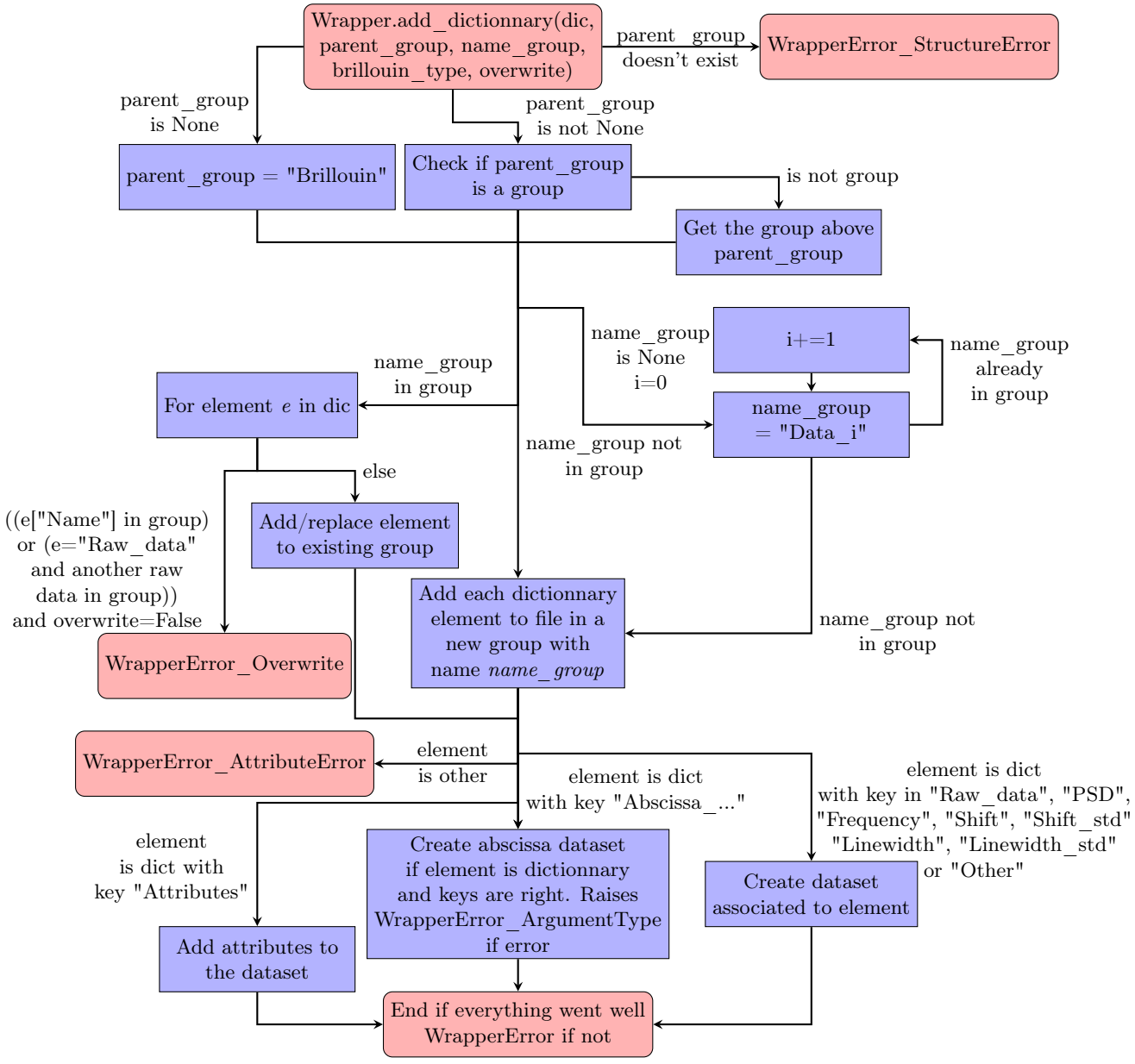


Figure 1.3: Flowchart of the add_dictionnary method

1.3.3 Wrapper.create_group

This method allows the user to create a new group inside the HDF5 file. This is done by verifying that no overwritting occurs and that the parent group exists.

```
1 def create_group(self, name, parent_group = None):
```

1.3.3.0.1 Attributes:

- **name**: The name of the group to create.
- **parent_group** *(optional, default None)*: 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/Measure/...".
- **brillouin_type** *(optional, default "Root")*: The type of the group that has been created. By default "Root". The possible values are listed in preamble
- **overwrite** *(optional, default False)*: If True, if a group of the same name already exists in the selected parent group, all its elements will be deleted.

1.3.3.0.2 Returns: Nothing

1.3.3.0.3 Raises:

- **WrapperError_Overwrite**: Raises an error if the group name already exists at the selected parent path.
- **WrapperError_StructureError**: Raises an error if the parent group does not exist in the HDF5 file.

1.3.3.0.4 Flowchart: The function's logic is represented in the following flowchart:

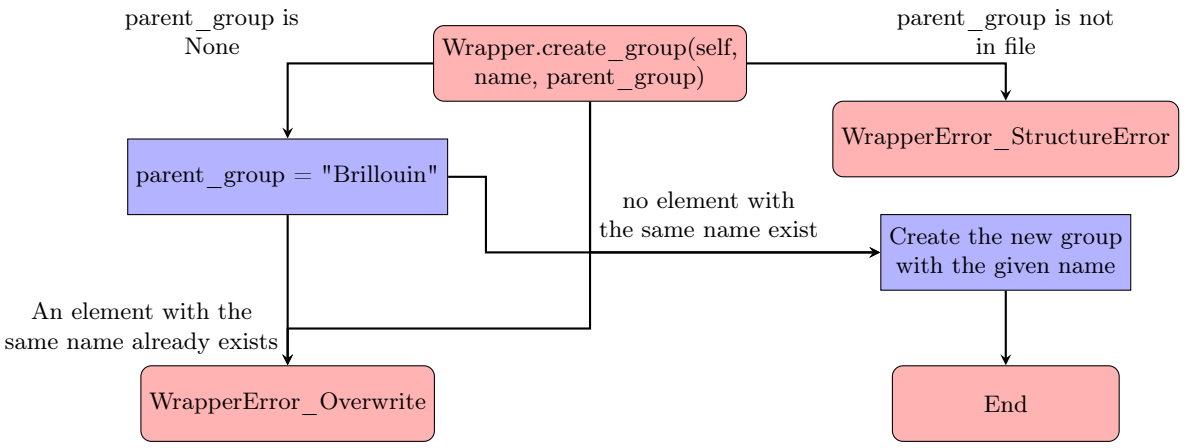


Figure 1.4: Flowchart of the create_group method

1.3.4 Wrapper.delete_element

This method allows the user to delete a dataset or group in the HDF5 file.

```
1 def delete_element(self, path = None):
```

1.3.4.0.1 Attributes:

- **path** *(optional, default None)*: The path to the group or dataset we want to delete in the form "Brillouin/Measure/PSD". If None, the whole file is deleted.

1.3.4.0.2 Raises:

- **WrapperError_StructureError**: If the path does not lead to a valid element in the file.
- **WrapperError**: If there are errors in the deleting process.

1.3.4.0.3 Flowchart:

The function's logic is represented in the following flowchart:

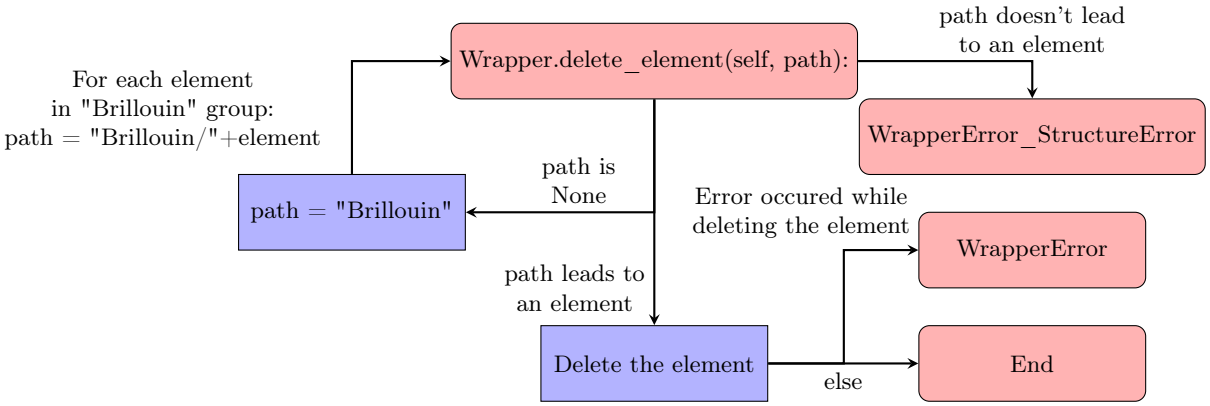


Figure 1.5: Flowchart of the delete_element method

1.3.5 Wrapper.get_attributes

This method allows the user to extract all the attributes of a dataset or group in a hierarchical way. This means that this method opens the file, goes from the root to the selected group, and extracts all the attributes associated to the groups along the way. The function returns a dictionnary, where the keys are the names of the attributes and the values are their values.

```
1 def get_attributes(self, path = None):
```

1.3.5.0.1 Attributes:

- **path** *(optional, default None)*: The path to the group or dataset which attributes we want to extract in the format "Brillouin/Measure/PSD". If None, the attributes of the root group are returned.

1.3.5.0.2 Returns: A dictionnary containing the attributes of the selected group or dataset where the keys are the names of the attributes and the values are their values. The keys of the attributes are listed in the spreadsheet located in the spreadsheets folder of the project.

1.3.5.0.3 Raises:

- **WrapperError_StructureError**: Raises an error if the path does not lead to a valid element in the file.

1.3.5.0.4 Flowchart: The function's logic is represented in the following flowchart:

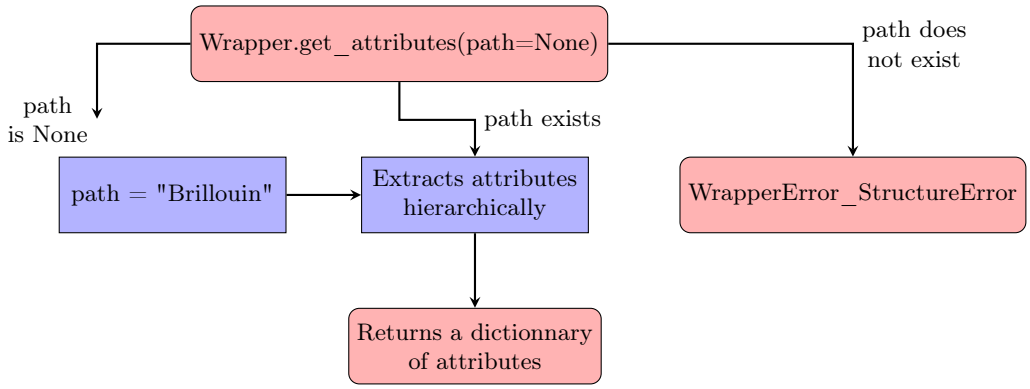


Figure 1.6: Flowchart of the get_attributes method

1.3.6 Wrapper.get_structure

This method allows the user to extract the structure in the form of a dictionnary. This dictionnary is composed of all the elements of the file and their "Brillouin_type" attribute, in the form of a JSON file or dictionnary similar to this one:

```
{'Brillouin': {'Brillouin_type': 'Root',
    'M1': {'Brillouin_type': 'Root',
        'Measure': {'Brillouin_type': 'Measure',
            'Raw_data': {'Brillouin_type': 'Raw_data'}
            }
        }
    }
}
```

```
1 def get_structure(self, filepath = None):
```

1.3.6.0.1 Attributes:

- **filepath** *(optional, default None)*: The filepath to a HDF5 file. If None, the file of the wrapper is used.

1.3.6.0.2 Returns: A dictionnary representing the structure of the file as detailed before

1.3.6.0.3 Flowchart: The function's logic is represented in the following flowchart:

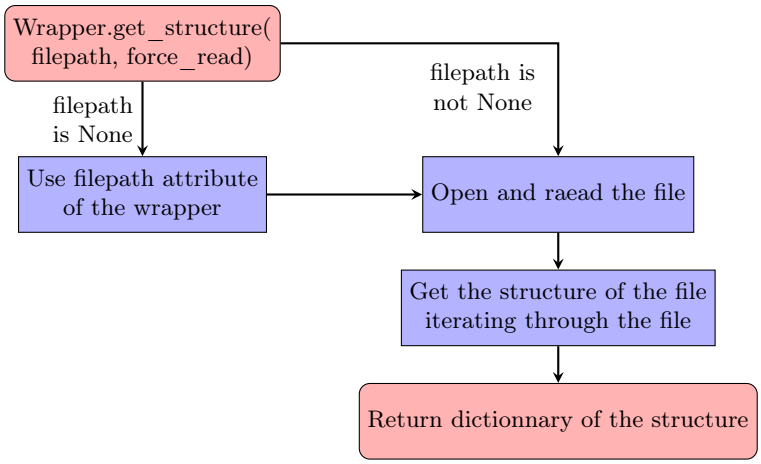


Figure 1.7: Flowchart of the get_structure method

1.3.7 Wrapper.save_as_hdf5

This method allows the user to save the file to a specified location.

```
1 def save_as_hdf5(self, filepath = None, remove_old_file = True):
```

1.3.7.0.1 Attributes:

- **filepath** *(optional, default None)*: The filepath to the HDF5 file to save to.
- **remove_old_file** *(optional, default True)*: A boolean that indicates whether the old file should be removed or not.

1.3.7.0.2 Raises:

- **WrapperError_Overwrite**: If the file already exists.
- **WrapperError**: If an error occured while saving the file.

1.3.7.0.3 Flowchart:

The function's logic is represented in the following flowchart:

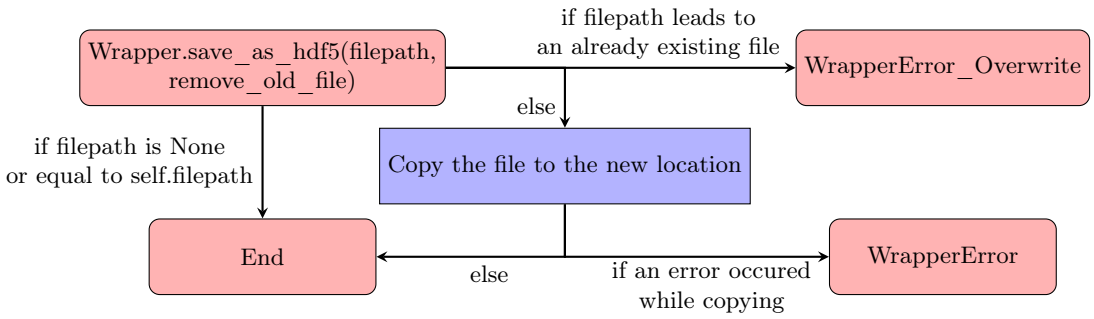


Figure 1.8: Flowchart of the save_as_hdf5 method

1.3.8 Wrapper.set_attributes_data

This method allows the user to update the attributes of the wrapper with the values of a dictionnary. This is the preferred way to update the attributes of a group or dataset of the HDF5 file.

```
def set_attributes_data(self, properties, path = None, overwrite = False):
```

1.3.8.0.1 Attributes:

- **properties**: A dicitonnary of the property(ies) to be updated.
- **path** *(optional, default None)*: The path to the group or dataset whose metadata we want to update. If None, the metadata of the root group are updated.
- **overwrite** *(optional, default False)*: If True, the attributes of the selected group or dataset are overwritten. If False, only new attributes are added to the existing ones.

1.3.8.0.2 Raises:

- **WrapperError**: If an error occured while updating the metadata in the HDF5 file

1.3.8.0.3 Flowchart:

The function's logic is represented in the following flowchart:

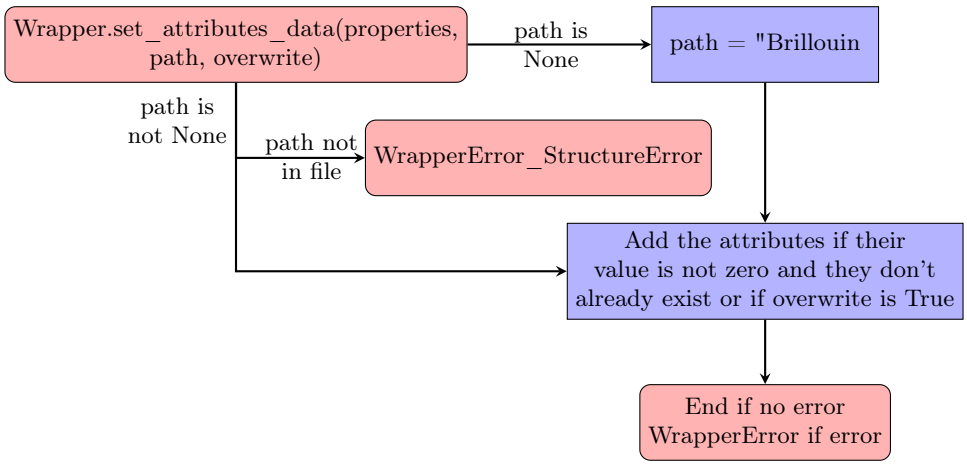


Figure 1.9: Flowchart of the set_attributes_data method

1.4 Derived methods of the Wrapper class

1.4.1 Wrapper.add_abscissa

This method allows the user to add an abscissa to the HDF5 file. It adds the data by calling the Wrapper.add_dictionnary method.

```
def add_abscissa(self, data, parent_group=None, name=None, unit = "1" ,
    dim_start = 0, dim_end = None, overwrite = False):
```

1.4.1.0.1 Attributes:

- **data**: The abscissa array to add to the wrapper.
- **parent_group**: The path to the group or dataset whose the raw data will be added in the form "Brillouin/Measure/...".
- **name** *(optional, default None)*: The name that will be given to the dataset. If None, the name is set to "Raw data".
- **unit** *(optional, default "1")*: The unit of the abscissa (e.g. "microns").
- **dim_start** *(optional, default 0)*: The first dimension of the abscissa.
- **dim_end** *(optional, default None)*: The last dimension of the abscissa. If None, the abscissa is considered to be a vector.
- **overwrite** *(optional, default False)*: If True, the attributes of the selected group or dataset are overwritten if they exist in the file.

1.4.1.0.2 Raises:

- All the errors of the Wrapper.add_dictionnary method.

1.4.2 Wrapper.add_frequency

This method allows the user to add a frequency array to the HDF5 file. It adds the data by calling the Wrapper.add_dictionnary method. By default, the frequency array is stored in "GHz". Note however that this will just affect the presentation of the results and not the process itself.

```
def add_frequency(self, data, parent_group, name = None, overwrite = False):
```

1.4.2.0.1 Attributes:

- **data**: The frequency array to add to the wrapper.
- **parent_group**: The path to the group or dataset where the frequency will be added in the form "Brillouin/Measure/...".
- **name** *(optional, default None)*: The name that will be given to the dataset. If None, the name is set to "Frequency".
- **overwrite** *(optional, default False)*: If True, the attributes of the selected group or dataset are overwritten if they exist in the file.

1.4.2.0.2 Raises:

- **WrapperError_StructureError**: If the parent group does not exist in the HDF5 file.
- All the errors of the Wrapper.add_dictionnary method.

1.4.3 Wrapper.add_PSD

This method allows the user to add a Power Spectral Density array to the HDF5 file. It adds the data by calling the Wrapper.add_dictionnary method.

```
def add_PSD(self, data, parent_group, name = None, overwrite = False):
```

1.4.3.0.1 Attributes:

- **data**: The PSD array to add to the wrapper.
- **parent_group**: The path to the group or dataset where the frequency will be added in the form "Brillouin/Measure/...".
- **name** *(optional, default None)*: The name that will be given to the dataset. If None, the name is set to "PSD".
- **overwrite** *(optional, default False)*: If True, the attributes of the selected group or dataset are overwritten if they exist in the file.

1.4.3.0.2 Raises:

- **WrapperError_StructureError**: If the parent group does not exist in the HDF5 file.
- All the errors of the Wrapper.add_dictionnary method.

1.4.4 Wrapper.add_raw_data

This method allows the user to add raw data to the HDF5 file. It adds the data by calling the Wrapper.add_dictionnary method.

```
1 def add_raw_data(self, data, parent_group, name = None, overwrite = False):
```

1.4.4.0.1 Attributes:

- **data**: The data to add to the wrapper.
- **parent_group**: The path to the group or dataset whose the raw data will be added in the form "Brillouin/Measure/...".
- **name** *(optional, default None)*: The name that will be given to the dataset. If None, the name is set to "Raw data".
- **overwrite** *(optional, default False)*: If True, the attributes of the selected group or dataset are overwritten if they exist in the file.

1.4.4.0.2 Raises:

- **WrapperError_StructureError**: If the parent group does not exist in the HDF5 file.
- All the errors of the Wrapper.add_dictionnary method.

1.4.5 Wrapper.add_treated_data

This method allows the user to add the result of a treatment to the HDF5 file. It adds the data by calling the Wrapper.add_dictionnary method.

```
1 def add_treated_data(self, shift, linewidth, shift_std, linewidth_std,
      parent_group, name_group = None, overwrite = False):
```

1.4.5.0.1 Attributes:

- **shift**: The shift array to add to the wrapper.
- **linewidth**: The linewidth array to add to the wrapper.
- **shift_std**: The standard deviation of the shift array to add to the wrapper.
- **linewidth_std**: The standard deviation of the linewidth array to add to the wrapper.
- **parent_group**: The path to the group or dataset whose the raw data will be added in the form "Brillouin/Measure/...".
- **name_group** *(optional, default None)*: The name that will be given to the group containing all the treated dataset. If None, the name is set to "Treat_i" where "i" is an integer that ensures that the name is unique.
- **overwrite** *(optional, default False)*: If True, the attributes of the selected group or dataset are overwritten if they exist in the file.

1.4.5.0.2 Raises:

- **WrapperError_StructureError**: If the parent group does not exist in the HDF5 file.
- All the errors of the Wrapper.add_dictionnary method.

1.4.6 Wrapper.import_abscissa

This method allows the user to add an abscissa to the HDF5 file from a measure file. It adds the data by calling the Wrapper.add_abscissa method.

```
1 def add_abscissa(self, filepath, parent_group, creator = None, parameters = None
      , name=None, unit = "AU" , dim_start = 0, dim_end = None, reshape = None,
      overwrite = False):
```

1.4.6.0.1 Attributes:

- **filepath**: The path to the file containing the abscissa to import.
- **parent_group**: The path to the group or dataset whose the raw data will be added in the form "Brillouin/Measure/...".
- **creator** *(optional, default None)*: The structure of the file that has to be loaded. If None, a LoadError can be raised.
- **parameters** *(optional, default None)*: The parameters that are to be used to import the data correctly. If None, a LoadError can be raised.
- **name** *(optional, default None)*: The name that will be given to the dataset. If None, the name is set to "Raw data".
- **unit** *(optional, default "1")*: The unit of the abscissa (e.g. "microns").
- **dim_start** *(optional, default 0)*: The first dimension of the abscissa.
- **dim_end** *(optional, default None)*: The last dimension of the abscissa. If None, the abscissa is considered to be a vector.
- **reshape** *(optional, default None)*: The new shape of the array. If None, the shape is not changed.
- **overwrite** *(optional, default False)*: If True, the attributes of the selected group or dataset are overwritten if they exist in the file.

1.4.6.0.2 Raises:

- **WrapperError_FileNotFound**: If the file could not be found.
- All the errors of the Wrapper.add_abscissa method.
- All the LoadError errors of the load_data module.

1.4.7 Wrapper.import_frequency

This method allows the user to import a frequency array to the HDF5 file. It adds the data by calling the Wrapper.add_frequency method. By default, the frequency array is stored in "GHz". Note however that this will just affect the presentation of the results and not the process itself.

```
def import_frequency(self, filepath, parent_group, creator = None, parameters =
    None, name = None, reshape = None, overwrite = False):
```

1.4.7.0.1 Attributes:

- **filepath**: The frequency array to add to the wrapper.
- **parent_group**: The path to the group or dataset where the frequency will be added in the form "Brillouin/Measure/...".
- **creator** *(optional, default None)*: The structure of the file that has to be loaded. If None, a LoadError can be raised.
- **parameters** *(optional, default None)*: The parameters that are to be used to import the data correctly. If None, a LoadError can be raised.
- **name** *(optional, default None)*: The name that will be given to the dataset. If None, the name is set to "Frequency".
- **reshape** *(optional, default None)*: The new shape of the array. If None, the shape is not changed.
- **overwrite** *(optional, default False)*: If True, the attributes of the selected group or dataset are overwritten if they exist in the file.

1.4.7.0.2 Raises:

- **WrapperError_FileNotFound**: If the file could not be found.
- All the errors of the Wrapper.add_frequency method.
- All the LoadError errors of the load_data module.

1.4.8 Wrapper.import_PSD

This method allows the user to import raw data to the HDF5 file. It adds the data by calling the Wrapper.add_raw_data method.

```
def import_raw_data(self, filepath, parent_group, creator = None, parameters =
    None, name = None, reshape = None, overwrite = False):
```

1.4.8.0.1 Attributes:

- **filepath**: The filepath to the raw data to import.
- **parent_group**: The path to the group or dataset where the frequency will be added in the form "Brillouin/Measure/...".
- **creator** *(optional, default None)*: The structure of the file that has to be loaded. If None, a LoadError can be raised.
- **parameters** *(optional, default None)*: The parameters that are to be used to import the data correctly. If None, a LoadError can be raised.
- **name** *(optional, default None)*: The name that will be given to the dataset. If None, the name is set to "Frequency".
- **reshape** *(optional, default None)*: The new shape of the array. If None, the shape is not changed.
- **overwrite** *(optional, default False)*: If True, the attributes of the selected group or dataset are overwritten if they exist in the file.

1.4.8.0.2 Raises:

- **WrapperError_FileNotFound**: If the file could not be found.
- All the errors of the Wrapper.add_frequency method.
- All the LoadError errors of the load_data module.

1.4.9 Wrapper.import_raw_data

This method allows the user to add a Power Spectral Density array to the HDF5 file from a file. It adds the data by calling the Wrapper.add_PSD method.

```
def import_PSD(self, filepath, parent_group, creator = None, parameters = None,
    name = None, reshape = None, overwrite = False):
```

1.4.9.0.1 Attributes:

- **filepath**: The path to the file containing the PSD to import.
- **parent_group**: The path to the group or dataset where the frequency will be added in the form "Brillouin/Measure/...".
- **creator** *(optional, default None)*: The structure of the file that has to be loaded. If None, a LoadError can be raised.
- **parameters** *(optional, default None)*: The parameters that are to be used to import the data correctly. If None, a LoadError can be raised.
- **name** *(optional, default None)*: The name that will be given to the dataset. If None, the name is set to "PSD".
- **reshape** *(optional, default None)*: The new shape of the array. If None, the shape is not changed.
- **overwrite** *(optional, default False)*: If True, the attributes of the selected group or dataset are overwritten if they exist in the file.

1.4.9.0.2 Raises:

- **WrapperError_FileNotFound**: If the file could not be found.
- All the errors of the Wrapper.add_PSD method.
- All the LoadError errors of the load_data module.

1.4.10 Wrapper.import_treated_data

This method allows the user to import the arrays resulting from a treatment to the HDF5 file. It adds the data by calling the Wrapper.add_PSD method.

```
def import_treated_data(self, filepath_shift, filepath_linewidth,
    filepath_shift_std, filepath_linewidth_std, parent_group, creator = None,
    parameters = None, name = None, reshape = None, overwrite = False):
```

1.4.10.0.1 Attributes:

- **filepath_shift**: The path to the file containing the shift array to import.
- **filepath_linewidth**: The path to the file containing the linewidth array to import.
- **filepath_shift_std**: The path to the file containing the shift standard deviation array to import.
- **filepath_linewidth_std**: The path to the file containing the linewidth standard deviation array to import.
- **parent_group**: The path to the group or dataset where the frequency will be added in the form "Brillouin/Measure/...".
- **creator** *(optional, default None)*: The structure of the files that have to be loaded. If None, a LoadError can be raised.
- **parameters** *(optional, default None)*: The parameters that are to be used to import the data correctly. If None, a LoadError can be raised.
- **name** *(optional, default None)*: The name that will be given to the dataset. If None, the name is set to "PSD".
- **reshape** *(optional, default None)*: The new shape of the array. If None, the shape is not changed.
- **overwrite** *(optional, default False)*: If True, the attributes of the selected group or dataset are overwritten if they exist in the file.

1.4.10.0.2 Raises:

- **WrapperError_FileNotFound**: If one of the files could not be found.
- All the errors of the Wrapper.add_treated_data method.
- All the LoadError errors of the load_data module.

1.4.11 Wrapper.import_properties_data

This method allows the user to import data from a CSV, XLS or XLSX file to a dataset or group of the HDF5 file. It reads the data from the file and updates the metadata by calling the Wrapper.set_attributes_data method.

```
def import_properties_data(self, filepath, path = None, overwrite = False):
```

1.4.11.0.1 Attributes:

- **filepath**: The filepath to the file containing the updated properties.
- **path** *(optional, default None)*: The path to the group or dataset whose metadata we want to update in the form: "Brillouin/Measure". If None, the metadata of the root group are updated.
- **overwrite** *(optional, default False)*: If True, the attributes of the selected group or dataset are overwritten. If False, only new attributes are added to the existing ones.

1.4.11.0.2 Raises:

- **WrapperError_FileNotFound**: If the file with the properties is not found or not valid.
- All the errors of the Wrapper.set_attributes_data method.

1.4.11.0.3 Flowchart:

The function's logic is represented in the following flowchart:

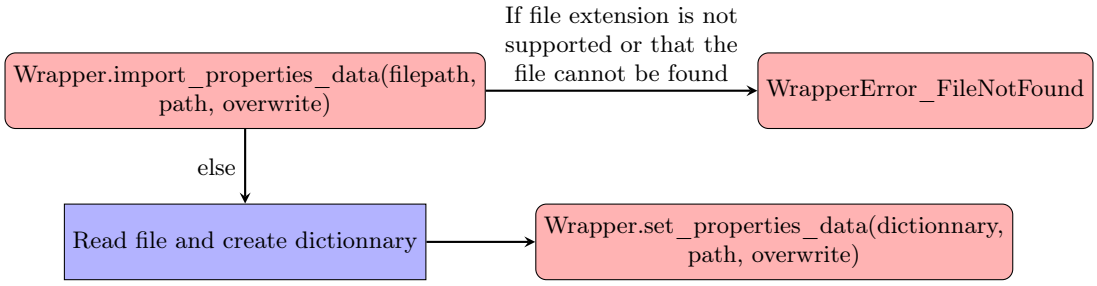


Figure 1.10: Flowchart of the import_properties_data method

1.4.12 Wrapper.update_property

This method allows the user to update one attribute of the wrapper. This method is based on the Wrapper.set_attributes_data method.

```
1 def update_property(self, name, value, path = None):
```

1.4.12.0.1 Attributes:

- **name**: The name of the attribute to update.
- **value**: The new value of the attribute.
- **path** *(optional, default None)*: The path to the group or dataset whose metadata we want to update in the form: "Brillouin/Measure". If None, the metadata of the root group are updated.

1.4.12.0.2 Raises:

- All the errors of the Wrapper.set_attributes_data method.

1.4.12.0.3 Flowchart:

The function's logic is represented in the following flowchart:

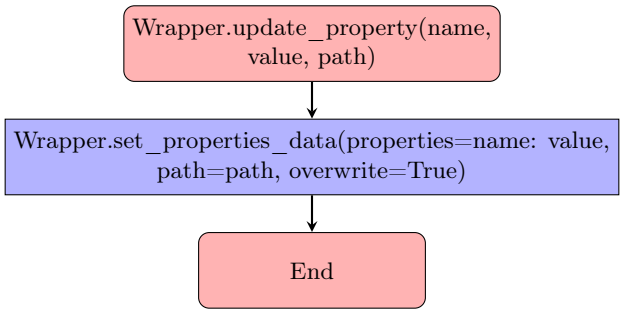


Figure 1.11: Flowchart of the import_properties_data method

1.5 Console-specific methods of the Wrapper class

1.5.1 Wrapper.print_structure

This method allows the user to print in the console the structure of the file.

```
1 def print_structure(self, lvl = 0):
```

1.5.1.0.1 Attributes:

- **lvl** *(optional, default 0)*: This parameter is used by the function to display the tree of the file recursively. It is not meant to be used by the user.

1.5.2 Wrapper.print_metadata

This method allows the user to print in the console all the attributes that apply to a dataset or group.

```
1 def print_metadata(self, path = None, lvl=0):
```

1.5.2.0.1 Attributes:

- **path** *(optional, default None)*: The path to the group or dataset which attributes we want to extract in the format "Brillouin/Measure/PSD". If None, the attributes of the root group are displayed.

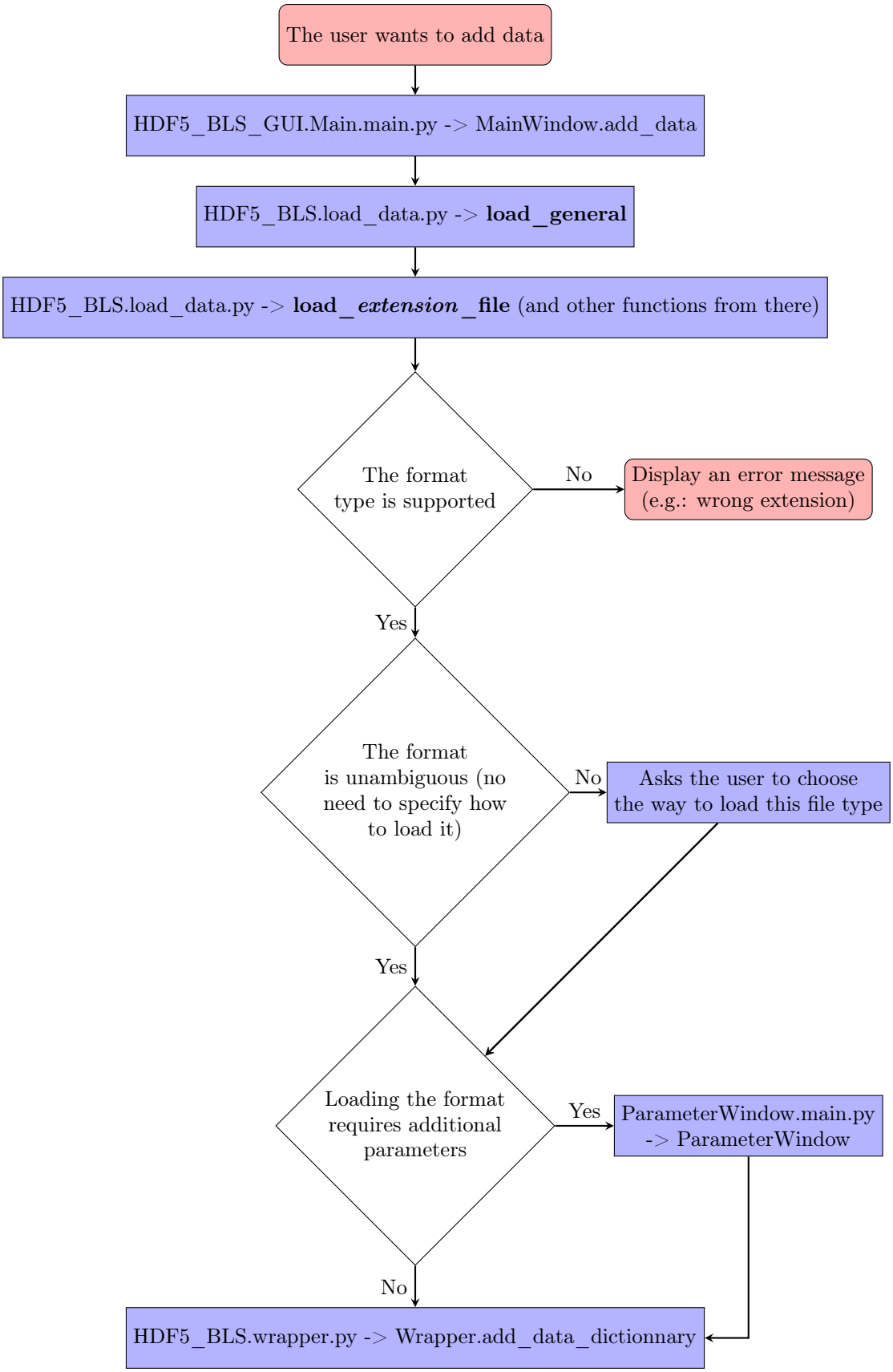
Chapter 2

Load data

Are you using a format that is already supported by the `HDF5_BLS` package?

- Yes but it doesn't work with my data.
- Yes but I would like to improve the code.
- No, I would like to add support for my data.

In any of these situations, the pipeline to load the data is fixed. Here we schematize this pipeline as a call graph:


The user wants to add data
HDF5_BLS_GUI.Main.main.py -> MainWindow.add_data
HDF5_BLS.load_data.py -> load_general
HDF5_BLS.load_data.py -> load_extension_file (and other functions from there)
The format type is supported
No
Display an error message (e.g.: wrong extension)
Yes
The format is unambiguous (no need to specify how to load it)
No
Asks the user to choose the way to load this file type
Yes
Loading the format requires additional parameters
Yes
ParameterWindow.main.py -> ParameterWindow
No
HDF5_BLS.wrapper.py -> Wrapper.add_data_dictionnary

In order to allow other formats to be used or other techniques to be applied to load files from the same format, the two functions that will be modified are:

- **load_general**: This function acts as a switch to launch different functions depending on the extension of the file and the way the file is loaded.
- **load_*extension*_file**: This function is specific to a file extension and is called by `load_general` to load the file. It itself act as a switch to lauch different functions from the **HDF5_BLS.load_formats.load_*extension*.py**.

2.1 Adding a user-specific function to an already supported format

> You are in the situation where you are using a format that is already supported by the `HDF5_BLS` package (for example ".dat") but that doesn't work with your data.

Here are the steps to follow:

1. Locate the python file that handles your data format in the `load_formats` folder of the `HDF5_BLS` package. The name of the file should correspond to the name of the format you are using (for example "load_dat.py" if you are using ".dat" files).
2. Add the function that will load your data to the file. The function should have the following signature:

```
def load_dat_Wien(filepath, parameters = None):
```

   In the case where you don't need to load the data with parameters, the function should have the following signature:

```
def load_dat_Wien(filepath):
```

3. Write the code that will load your data. Your function should retunr a dictionnary with at least two keys: "Data" and "Attributes". The "Data" key should contain the data you are loading and the "Attributes" key should contain the attributes of the file. You can also add abscissa to your data if you want to, in that case, add the key "Abscissa_*name*" where *name* is the name you want to give to the abscissa (for example "Abscissa_Time").
4. Go to the `load_data.py` file in the `HDF5_BLS` package and locate the function dedicated to the format you are using (for example "load_dat_file" if you are using ".dat" files)
5. Make sure that you are importing the function you just created:

```
from HDF5_BLS.load_formats.load_dat import load_dat_Wien
```

6. Then, define an identifier for your function (for example "Wien") and either create or add your identifier to the if-else statement. Don't forget to add your identifier to the "creator_list" list in the "else" statement:

```
if creator == "GHOST": return load_dat_GHOST(filepath)
...
elif creator == "Wien": return load_dat_Wien(filepath)
else:
creator_list = ["GHOST", "TimeDomain", "Wien"]
raise LoadError_creator(f"Unsupported creator {creator}, accepted values
    are: {', '.join(creator_list)}", creator_list)
```

7. Add a test to the function in the "tests/load_data_test.py" file with a test file placed in the "tests/test_data" folder. This test is important as they are run automatically when the package is pushed to GitHub (ie: it makes my life easier i.

8. You can now use your data format with the `HDF5_BLS` package, and in particular, the GUI. You are invited to push your code to GitHub and create a pull request to the main repository :)

2.2 Improving an already supported function

> You are in the situation where you want to improve a load function of the `HDF5_BLS` package (for example ".dat").

Here are the steps to follow:

1. Locate the python file that handles your data format in the `load_formats` folder of the `HDF5_BLS` package. The name of the file should correspond to the name of the format you are using (for example "load_dat.py" if you are using ".dat" files).
2. Locate the function that loads your data. The function should have a name similar to (might not have parameters):

```
def load_dat_Wien(filepath, parameters = None):
```

3. Update the code. One good measure is to duplicate the function and comment one of the two versions. Then, write your code and run the tests. If the tests fail, you can always go back to the previous version. Note that if the test fails, the code cannot be pushed to GitHub.
4. If everything is sound, you can now use your new function with the `HDF5_BLS` package. You are invited to push your code to GitHub and create a pull request to the main repository :)
5. Note: If you want to improve the loading of the data to the hdf5 file (chunking for example), please contact the maintainer directly.

2.3 Adding a function to a new format

> You are in the situation where you are using a new format that is not supported by the `HDF5_BLS` package.

Here are the steps to follow:

1. Navigate to the `load_formats` folder of the `HDF5_BLS` package.
2. Create a new python file with the name of the format you are using (for example "load_unicorn.py" if you are using ".unicorn" files).
3. Add the function that will load your data to the file. The function should have the following signature:

```
def load_unicorn_Wien(filepath, parameters = None):
```

   In the case where you don't need to load the data with parameters, the function should have the following signature:

```
def load_dat_Wien(filepath):
```

4. Write the code that will load your data. Your function should retunr a dictionnary with at least two keys: "Data" and "Attributes". The "Data" key should contain the data you are loading and the "Attributes" key should contain the attributes of the file. You can also add abscissa to your data if you want to, in that case, add the key "Abscissa_*name*" where *name* is the name you want to give to the abscissa (for example "Abscissa_Time").

5. Go to the `load_data.py` file in the `HDF5_BLS` package and create the function dedicated to the format you are using (for example "load_unicorn_file" if you are using ".unicorn" files)

6. Make sure that you are importing the function you just created:

```
from HDF5_BLS.load_formats.load_unicorn import load_unicorn_Wien
```

7. Add a test to the function in the "tests/load_data_test.py" file with a test file placed in the "tests/test_data" folder. This test is important as they are run automatically when the package is pushed to GitHub (ie: it makes my life easier i.

8. You can now use your data format with the `HDF5_BLS` package, and in particular, the GUI. You are invited to push your code to GitHub and create a pull request to the main repository :)

Chapter 3

Analyze data

The analysis module is built following a Object-Oriented approach. It is composed of classes called *Analysis_type* where "type" refer to the type of technique being used to acquire the data. These classes are the main classes of the module. They rely on 5 core attributes:

- *_algorithm*: A JSON file describing a particular sequence of functions to be applied for the analysis
- x: The abscissa of the data on which we perform the analysis
- y: The data points that are to be analyzed
- *points*: A dictionnary of points on which to base the analysis
- *windows*: A dictionnary of windows on which to base the analysis

This chapter will present successively:

- The different attributes of the class, their structure, the conditions that they must fulfill, and their intended uses (see section 3.1).
- The modular approach of the analysis module and the way functions are to be implemented.
- The creation of an algorithm for reusable analysis.
- The versioning of the analysis module and of its functions.
- The testing strategy.

3.1 The attributes of the classes of the *analysis* moduel

All the classes of the analysis module are built following a Object-Oriented approach and rely on 5 core attributes:

- *_algorithm*: A JSON file describing a particular sequence of functions to be applied for the analysis
- x: The abscissa of the data on which we perform the analysis
- y: The data points that are to be analyzed
- *points*: A dictionnary of points on which to base the analysis
- *windows*: A dictionnary of windows on which to base the analysis

Additionaly, a sixth attribute, *_history* is used to store the evolution of each attribute during the analysis. This is however a "silent" attribute that the developper should not consider as it is automatically updated as will be explained in the following sections.

3.1.1 The _algorithm attribute

The *_algorithm* attribute is a JSON file describing a particular sequence of functions to be applied for the analysis. It is built with two ideas in mind:

- Store the sequence of functions that are applied for the analysis, with all their parameters, so that it is possible to reproduce the analysis or improve it by changing the parameters of the functions.
- Store a human-readable description of the sequence of functions that are applied for the analysis, so that it is possible to understand what is done in the analysis even for someone who is not familiar with the code.

The *_algorithm* attribute is a JSON file that is built with the following structure:

- *name*: The name of the algorithm used for the analysis
- *version*: The version of the algorithm
- *author*: The author of the algorithm (laboratory or person)
- *description*: A human-readable short presentation of the algorithm
- *functions*: A list of functions to be applied for the analysis following this structure:
  - *function*: The name of the function to be applied. This name does not include the name of the class as this name depends on the name of "SPECTROMETER.Type" attribute of the data to be treated.
  - *version*: The version of the function
  - *parameters*: A dictionnary of parameters to be passed to the function
  - *description*: A human-readable presentation of the function

As an example, the following JSON file describes a simple analysis that consists in taking the mean of the data points:

```
{
    "name": "Mean",
    "version": "1.0",
    "author": "Pierre Bouvet - Medical University of Vienna",
    "description": "Mean of the data points",
    "functions": [
        {
            "function": "mean",
            "version": "1.0",
            "parameters": {
                points: None
                }
            "description": "Mean of the data points"
        }
    ]
}
```

This description of the algorithm is "blank" meaning that the parameters are not defined. This is useful when the algorithm is stored as a sequence of functions that have to be adapted. However, if the analysis concerns a specific data set, then we can pass the attribute names as parameters, as well as any other parameters that are used for the analysis. For example, if we want to perform a polynomial fit with a polynomial of order 2 on the data points windowed around the first anti-stokes peak as defined in the "window" attribute, we can write the following JSON file:

```
{
    "name": "Polynomial fit",
    "version": "1.0",
    "author": "Pierre Bouvet - Medical University of Vienna",
    "description": "Polynomial fit of the data points",
    "functions": [
        {
            "function": "polyfit",
            "version": "1.0",
            "parameters": {
                "x": "self.x",
                "y": "self.y",
                "window": "self.windows['anti_stokes_1']",
                "degree": 2
                }
            "description": "Polynomial fit of the data points performed
               on x=self.x and y=self.y, around the window defined by
               self.windows['anti_stokes_1'], with a polynomial of order
                2"
        }
    ]
}
```

Storing algorithms composed of a sequence of function is done by adding a function to the list of functions, following the same structure as the one described above:

```
    {
        "name": "Polynomial fit",
        "version": "1.0",
        "author": "Pierre Bouvet - Medical University of Vienna",
        "description": "Polynomial fit of the data points",
        "functions": [
            {
                "function": "function_1",
                "version": "1.0",
                "parameters": {
                    "x": "self.x",
                    "degree": 2
                    }
                "description": "The first function of the analysis
                   performed on x=self.x with a degree of 2"
            },
            {
                "function": "function_2",
                "version": "1.0",
                "parameters": {
                    "y": "self.y",
                    "resampling": True
                    }
                "description": "The second function of the analysis
                   performed on y=self.y with resampling=True"
            }
        ]
    }
```

Note that these examples are meant to present the principle at the core of the treatment but do not represent the actual implementation of the functions. The actual implementation of the functions is detailed in the following sections. Full examples will also be given at the end of this chapter.

3.1.2 The x and y attributes

The x and y attributes are respectively the abscissa and the ordinate of the data on which the analysis is performed. They are both numpy arrays of the same length. The x attribute is the abscissa of the data points, while the y attribute is the data points themselves. These arrays are 1D as they are plotted on a 1D graph.

These arrays are subject to evolve during the analysis, for example if a normaliation occurs. To store their evolution and maximize the speed of the algorithm, whenever these arrays are changed, the *history* attribute is updated.

3.1.3 Points

The *points* attributes is a dictionnary of points referenced in the *self.x* arrays. The keys of the dictionnary have a nomenclature constrained by the type of points. This nomenclature is of the following form:

prefix_number

where *prefix* is a string describing the type of point and *number* is an integer describing the position of the point in the list of points of this type. The *prefix* can be one of the following:

- *anti-stokes*: A point corresponding to an anti-stokes peak
- *stokes*: A point corresponding to a stokes peak
- *elastic*: A point corresponding to an elastic peak
- *peak*: A point corresponding to an other peak

Example:

```
self.points ={
    "anti-stokes\_1" = 5.5542
    "stokes\_1" = -5.4363
}
```

Note that all the position are stored as float. Note also that the keys of the *points* attribute are excatly the same as the keys of the *windows* attribute.

3.1.4 Windows

The *windows* attributes is a dictionnary of tuple stored as values in the *self.x* array. The keys of the dictionnary have a nomenclature constrained by the type of points. This nomenclature is of the following form:

prefix_number

where *prefix* is a string describing the type of point and *number* is an integer describing the position of the point in the list of points of this type. The *prefix* can be one of the following:

- *anti-stokes*: A point corresponding to an anti-stokes peak
- *stokes*: A point corresponding to a stokes peak
- *elastic*: A point corresponding to an elastic peak
- *peak*: A point corresponding to an other peak

Example:

```
self.windows ={
    "anti-stokes\_1" = (3.0, 8.0)
    "stokes\_1" = (-8.0, -3.0)
}
```

Note that all the position are stored as float. Note also that the keys of the *points* attribute are excatly the same as the keys of the *points* attribute.

3.1.5 The _history attribute

The *_history* attribute resembles the *algorithm* attribute in the sense that it is also a list of dictionnaries where each element corresponds to a step of the analysis. However, the main difference is that the *_history* attribute is not a JSON file but a "silent" attribute that is automatically updated whenever a function is called. This means that the *_history* attribute is not meant to be used by the user but is rather used internally by the module to store the evolution of the attributes during the analysis and speed up the analysis whenever the user decides to change a parameter during the analysis.

The elements of the *_history* attribute are dictionnaries that contain 5 keys, 4 of which are optional to minimize the size of the memory used:

- *function*: The algorithm used for the analysis
- x (optional): The abscissa of the data on which we perform the analysis (only if the attribute has been changed at this step)
- y (optional): The data points that are to be analyzed (only if the attribute has been changed at this step)
- *points* (optional): A dictionnary of points on which to base the analysis (only if the attribute has been changed at this step)
- *windows* (optional): A dictionnary of windows on which to base the analysis (only if the attribute has been changed at this step)

Example:

```
self._history = [
    {
        "function": "__init__",
        "x": np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]),
        "y": np.array([0.0, 1.0, 4.0, 2.0, 3.0, 3.0, 5.0, 3.0, 2.0, 1.0]),
        "points": {},
        "windows": {}
    },
    {
        "function": "normalize",
        "y": np.array([0.0, 0.2, 0.8, 0.4, 0.6, 0.6, 1.0, 0.6, 0.4, 0.2])
    }
    {
        "function": "get_point_window",
        "points": {"anti-stokes\_1": 5.5542},
        "windows": {"anti-stokes\_1": (3.0, 8.0)},
    }
]
```

3.2 The classes of the *analysis* module and the implementation of algorithms

All the classes of the analysis module are built following a Object-Oriented approach and rely on 5 core attributes:

- *_algorithm*: A JSON file describing a particular sequence of functions to be applied for the analysis
- *x*: The abscissa of the data on which we perform the analysis
- *y*: The data points that are to be analyzed
- *points*: A dictionnary of points on which to base the analysis
- *windows*: A dictionnary of windows on which to base the analysis

Additionaly, a sixth attribute, *_history* is used to store the evolution of each attribute during the analysis. This is however a "silent" attribute that the developper should not consider as it is automatically updated as will be explained in the following sections.

3.2.1 The _algorithm attribute

The *_algorithm* attribute is a JSON file describing a particular sequence of functions to be applied for the analysis. It is built with two ideas in mind:

- Store the sequence of functions that are applied for the analysis, with all their parameters, so that it is possible to reproduce the analysis or improve it by changing the parameters of the functions.
- Store a human-readable description of the sequence of functions that are applied for the analysis, so that it is possible to understand what is done in the analysis even for someone who is not familiar with the code.

The *_algorithm* attribute is a JSON file that is built with the following structure:

- *name*: The name of the algorithm used for the analysis
- *version*: The version of the algorithm
- *author*: The author of the algorithm (laboratory or person)
- *description*: A human-readable short presentation of the algorithm
- *functions*: A list of functions to be applied for the analysis following this structure:
  - *function*: The name of the function to be applied. This name does not include the name of the class as this name depends on the name of "SPECTROMETER.Type" attribute of the data to be treated.
  - *version*: The version of the function
  - *parameters*: A dictionnary of parameters to be passed to the function
  - *description*: A human-readable presentation of the function

As an example, the following JSON file describes a simple analysis that consists in taking the mean of the data points:

```
{
    "name": "Mean",
    "version": "1.0",
    "author": "Pierre Bouvet - Medical University of Vienna",
    "description": "Mean of the data points",
    "functions": [
        {
            "function": "mean",
            "version": "1.0",
            "parameters": {
                points: None
```

```
                }
            "description": "Mean of the data points"
        }
    ]
}
```

This description of the algorithm is "blank" meaning that the parameters are not defined. This is useful when the algorithm is stored as a sequence of functions that have to be adapted. However, if the analysis concerns a specific data set, then we can pass the attribute names as parameters, as well as any other parameters that are used for the analysis. For example, if we want to perform a polynomial fit with a polynomial of order 2 on the data points windowed around the first anti-stokes peak as defined in the "window" attribute, we can write the following JSON file:

```
{
    "name": "Polynomial fit",
    "version": "1.0",
    "author": "Pierre Bouvet - Medical University of Vienna",
    "description": "Polynomial fit of the data points",
    "functions": [
        {
            "function": "polyfit",
            "version": "1.0",
            "parameters": {
                "x": "self.x",
                "y": "self.y",
                "window": "self.windows['anti_stokes_1']",
                "degree": 2
                }
            "description": "Polynomial fit of the data points performed
               on x=self.x and y=self.y, around the window defined by
               self.windows['anti_stokes_1'], with a polynomial of order
                2"
        }
    ]
}
```

Storing algorithms composed of a sequence of function is done by adding a function to the list of functions, following the same structure as the one described above:

```
    {
        "name": "Polynomial fit",
        "version": "1.0",
        "author": "Pierre Bouvet - Medical University of Vienna",
        "description": "Polynomial fit of the data points",
        "functions": [
            {
                "function": "function_1",
                "version": "1.0",
                "parameters": {
                    "x": "self.x",
                    "degree": 2
                    }
                "description": "The first function of the analysis
                   performed on x=self.x with a degree of 2"
            },
            {
```

```
                "function": "function_2",
                "version": "1.0",
                "parameters": {
                    "y": "self.y",
                    "resampling": True
                    }
                "description": "The second function of the analysis
                   performed on y=self.y with resampling=True"
            }
        ]
    }
```

Note that these examples are meant to present the principle at the core of the treatment but do not represent the actual implementation of the functions. The actual implementation of the functions is detailed in the following sections. Full examples will also be given at the end of this chapter.

3.2.2 The x and y attributes

The x and y attributes are respectively the abscissa and the ordinate of the data on which the analysis is performed. They are both numpy arrays of the same length. The x attribute is the abscissa of the data points, while the y attribute is the data points themselves. These arrays are 1D as they are plotted on a 1D graph.

These arrays are subject to evolve during the analysis, for example if a normaliation occurs. To store their evolution and maximize the speed of the algorithm, whenever these arrays are changed, the *history* attribute is updated.

3.2.3 Points

The *points* attributes is a dictionnary of points referenced in the *self.x* arrays. The keys of the dictionnary have a nomenclature constrained by the type of points. This nomenclature is of the following form:

prefix_number

where *prefix* is a string describing the type of point and *number* is an integer describing the position of the point in the list of points of this type. The *prefix* can be one of the following:

- *anti-stokes*: A point corresponding to an anti-stokes peak
- *stokes*: A point corresponding to a stokes peak
- *elastic*: A point corresponding to an elastic peak
- *peak*: A point corresponding to an other peak

Example:

```
self.points ={
    "anti-stokes\_1" = 5.5542
    "stokes\_1" = -5.4363
}
```

Note that all the position are stored as float. Note also that the keys of the *points* attribute are excatly the same as the keys of the *windows* attribute.

3.2.4 Windows

The *windows* attributes is a dictionnary of tuple stored as values in the *self.x* array. The keys of the dictionnary have a nomenclature constrained by the type of points. This nomenclature is of the following form:

prefix_number

where *prefix* is a string describing the type of point and *number* is an integer describing the position of the point in the list of points of this type. The *prefix* can be one of the following:

- *anti-stokes*: A point corresponding to an anti-stokes peak
- *stokes*: A point corresponding to a stokes peak
- *elastic*: A point corresponding to an elastic peak
- *peak*: A point corresponding to an other peak

Example:

```
self.windows ={
    "anti-stokes\_1" = (3.0, 8.0)
    "stokes\_1" = (-8.0, -3.0)
}
```

Note that all the position are stored as float. Note also that the keys of the *points* attribute are excatly the same as the keys of the *points* attribute.

3.2.5 The _history attribute

The *_history* attribute resembles the *algorithm* attribute in the sense that it is also a list of dictionnaries where each element corresponds to a step of the analysis. However, the main difference is that the *_history* attribute is not a JSON file but a "silent" attribute that is automatically updated whenever a function is called. This means that the *_history* attribute is not meant to be used by the user but is rather used internally by the module to store the evolution of the attributes during the analysis and speed up the analysis whenever the user decides to change a parameter during the analysis.

The elements of the *_history* attribute are dictionnaries that contain 5 keys, 4 of which are optional to minimize the size of the memory used:

- *function*: The algorithm used for the analysis
- *x* (optional): The abscissa of the data on which we perform the analysis (only if the attribute has been changed at this step)
- *y* (optional): The data points that are to be analyzed (only if the attribute has been changed at this step)
- *points* (optional): A dictionnary of points on which to base the analysis (only if the attribute has been changed at this step)
- *windows* (optional): A dictionnary of windows on which to base the analysis (only if the attribute has been changed at this step)

Example:

```
self._history = [
    {
        "function": "__init__",
        "x": np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]),
        "y": np.array([0.0, 1.0, 4.0, 2.0, 3.0, 3.0, 5.0, 3.0, 2.0, 1.0]),
        "points": {},
```

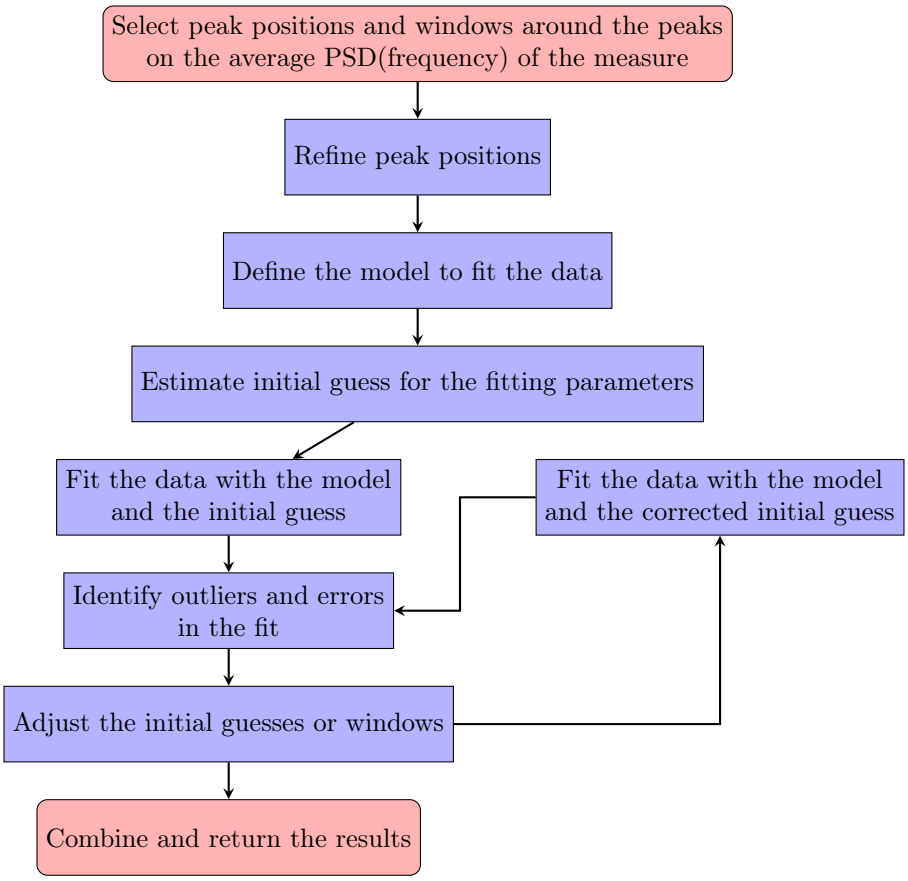
```
        "windows": {}
    },
    {
        "function": "normalize",
        "y": np.array([0.0, 0.2, 0.8, 0.4, 0.6, 0.6, 1.0, 0.6, 0.4, 0.2])
    }
    {
        "function": "get_point_window",
        "points": {"anti-stokes\_1": 5.5542},
        "windows": {"anti-stokes\_1": (3.0, 8.0)},
    }
]
```

Chapter 4

Treat data

The treatment of the data consist in extracting the relevant information from an already obtained Power Spectral Density and a frequency vector. This information is then used to perform the analysis. The treatment module works similarly to the analysis module: a series of functions are defined and an algorithm is created with these functions and then applied to the data. Additionally, the treatment module also contains functions to identify outliers and errors.

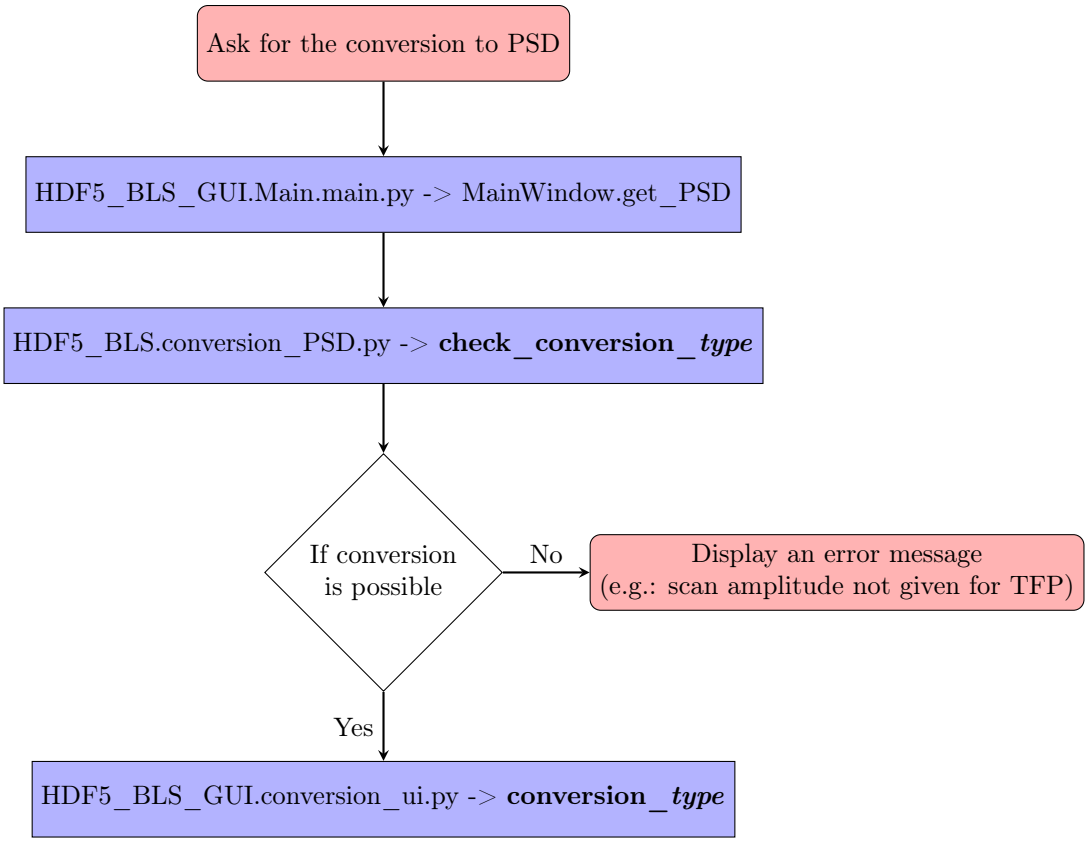
The general treatment approach on which the module is based is the following:



The goal of the treatment module is then to allow the user to define and apply an algorithm to the data. The treatment module is composed of a series of functions that can be used to define the algorithm. The user can also add his own functions to the module and then use them in the algorithm. The treatment module is designed to be flexible and adaptable to the user's needs.

4.1 Treat data to obtain a Power Spectral Density and a frequency vector

This is a call graph of the functions that are used when asking for a conversion to a PSD. The functions that we will develop have names in bold font. The functions have a fixed nomenclature that we will detail later on, based on the type of spectrometer being used. For this call graph, we will call our spectrometer: *type*:



To add a routine to convert raw data into a PSD, we'll therefore need to implement the two following functions:

4.1.0.0.1 HDF5_BLS.conversion_PSD.check_conversion_*type*

- **Parameters**:
  - wrp: the wrapper associated to the main h5 file. You should always use the parent file as some properties might be inherited.
  - path: the path to the data we want to treat in the form "Data/Data_0/..."
- **Returns**:
  - A boolean: True if the conversion is possible, False otherwise.
    Depending on the type of spectrometer, the conditions for conversion might be different. In some cases, like for Time Domain experiments, the conversion to PSD will require the user to enter parameters by hand, in other cases, like for TFP, the conversion can only be performed if the scan amplitude is given. The goal of this function is to block any further processing if the conversion is not possible, if the conversion is possible with arguments that are meant to be entered afterwards, it should return True.

Here is an example of the function `check_conversion_ar_BLS_VIPA`:

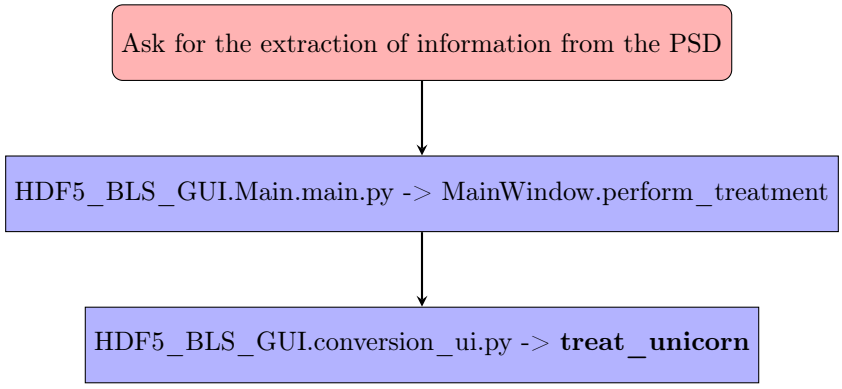
```
def check_conversion_ar_BLS_VIPA(wrapper, path):
    attributes = wrapper.get_attributes_path(path)
    if "TREAT.Center" in attributes.keys():
        return True
    else:
        return False
```

4.1.0.0.2 HDF5_BLS_GUI.conversion_ui.conversion_*type* :

- **Parameters**:
  - parent: the parent GUI window. This is usefull when you want to open other windows to ask the user for some parameters.
  - wrp: the wrapper associated to the main h5 file. You should always use the parent file as some properties might be inherited.
  - path: the path to the data we want to treat in the form "Data/Data_0/..."
- **Returns**:
  - A dictionnary with the following keys:
    - "PSD": the PSD dataset
    - "Frequency": the frequency dataset
    - "Process": a text decribing the process of converting the raw data into a PSD. This text is meant to inform the user of the process of converting the data into a PSD.

4.2 Treat data to extract information from a Power Spectral Density

This is a call graph of the functions that are used when asking treat a PSD and frequency. The functions that we will develop have names in bold font. The functions have a fixed nomenclature that we will detail later on, based on the type of spectrometer being used. For this call graph, we will call our spectrometer: *unicorn*:



To add a routine to extract information from our PSD, we'll therefore need to implement the following functions:

- *HDF5_BLS_GUI.conversion_ui.treat_unicorn*: This function will be used to allow the user to perform a treatment on its data. This function will open a GUI window in the form of a dialog to allow the user to select the parameters of the treatment. Note that this function will not return anything.

For GUI compatibility, the function must be called *treat_ type*, where "type" is the type of the spectrometer being used, and placed in the "HDF5_BLS_GUI/treat_ui.py" file. For example if your spectrometer type is "Unicorn", add the following function:

```
1 def treat_unicorn
```

This function must have the following parameters:

- parent: the parent GUI window
- wrp: the wrapper associated to the main h5 file
- path: the path to the data we want to treat in the form "Data/Data/..."

```
1 def treat_unicorn(parent, wrp, path):
```

From there, there are no guidelines to define your function. The GUI comes however with a few windows that can be inherited from to make the development of custom treatment processes easier. You can find an example of how it was done for the "TFP" treatment in the appendix.

4.3 Adding a new treatment function

To do

Contact

For questions or suggestions, please contact the maintainer at:

pierre.bouvet@meduniwien.ac.at.

Appendix

Appendix A

Examples of file structures

A.1 A single measure with no treatment

In this first example, we want to store a single measure of a water sample.

The following structure represents the base structure of the file:

```
file.h5
+-- Data (group) -> Name = "Measure"
|   +-- Data_0 (group) -> Name = "Water"
|   |   +-- Raw_data (dataset)
```

Note that we have here added arrows and an example of the value of the "Name" attributes.

A.2 A series of measures with no treatment

In this second example, we want to store a series of measures taken on three different samples: Water, Ethanol and Glycerol.

The following structure represents the base structure of the file:

```
file.h5
+-- Data (group) -> Name = "Measure"
|   +-- Data_0 (group) -> Name = "Water"
|   |   +-- Raw_data (dataset)
|   +-- Data_1 (group) -> Name = "Ethanol"
|   |   +-- Raw_data (dataset)
|   +-- Data_2 (group) -> Name = "Glycerol"
|   |   +-- Raw_data (dataset)
```

Note that we have here added arrows and an example of the value of the "Name" attributes.

A.3 A series of series of measures with no treatment but with a calibration spectrum and an impulse response measure

In this third example, we want to store a series of two measures taken on two different samples: Water and Ethanol. We also want to store a calibration curve and an impulse response curve.

The following structure represents the base structure of the file:

```
file.h5
+-- Data (group) -> Name = "Measure"
|   +-- Data_0 (group) -> Name = "Impulse_Response"
```

```
|   |   +-- Raw_data (dataset)
|   +-- Data_1 (group) -> Name = "Calibration"
|   |   +-- Raw_data (dataset)
|   +-- Data_2 (group) -> Name = "Water"
|   |   +-- Data_0 (group) -> Name = "Water_01"
|   |   |   +-- Raw_data (dataset)
|   |   +-- Data_1 (group) -> Name = "Water_02"
|   |   |   +-- Raw_data (dataset)
|   +-- Data_3 (group) -> Name = "Ethanol"
|   |   +-- Data_0 (group) -> Name = "Ethanol_01"
|   |   |   +-- Raw_data (dataset)
|   |   +-- Data_1 (group) -> Name = "Ethanol_02"
|   |   |   +-- Raw_data (dataset)
```

Note that we have here added arrows and an example of the value of the "Name" attributes.

A.4 A single measure converted to a Power Spectrum Density

In this fourth example, we want to store a single measure of a water sample. This measure has been converted into a Power Spectrum Density.

The following structure represents the base structure of the file:

```
file.h5
+-- Data (group) -> Name = "Measure"
|   +-- Data_0 (group) -> Name = "Water"
|   |   +-- Raw_data (dataset)
|   |   +-- PSD (dataset)
|   |   +-- Frequency (dataset)
```

Note that we have here added arrows and an example of the value of the "Name" attributes. In this case, all the steps of the conversion to PSD are stored in the "Data_0" group. The nomenclature of the attribute(s) used to store the parameters of the treatment is not specified.

A.5 Multiple measures converted to a Power Spectrum Density with a time-independent spectrometer

In this fifth example, we are in the situation where a time-independent spectrometer has been used to acquire multiple measures. In this case, the hierrarchy of the file can be used to reduce the number of datasets, by considering that all the PSD share the same frequency axis.

The following structure represents the base structure of the file:

```
file.h5
+-- Data (group) -> Name = "Measure"
|   +-- Frequency (dataset)
|   +-- Data_0 (group) -> Name = "Sample_1"
|   |   +-- Raw_data (dataset)
|   |   +-- PSD (dataset)
|   +-- Data_1 (group) -> Name = "Sample_2"
|   |   +-- Raw_data (dataset)
|   |   +-- PSD (dataset)
```

Note that we have here added arrows and an example of the value of the "Name" attributes.

A.6 A single measure with a treatment

In this sixth example, we want to store a single measure of a water sample that has been treated.

The following structure represents the base structure of the file:

```
file.h5
+-- Data (group) -> Name = "Measure"
|   +-- Data_0 (group) -> Name = "Water"
|   |   +-- Raw_data (dataset)
|   |   +-- PSD (dataset)
|   |   +-- Frequency (dataset)
|   |   +-- Treat_0 (group) -> Name = "Treat_5GHz"
|   |   |   +-- Shift (dataset)
|   |   |   +-- Shift_std (dataset)
|   |   |   +-- Linewidth (dataset)
|   |   |   +-- Linewidth_std (dataset)
```

Note that we have here added arrows and an example of the value of the "Name" attributes. In this case, all the steps of the treatment are stored in the "Treat_0" group. The nomenclature of the attribute(s) used to store the parameters of the treatment is not specified.

A.7 A single measure with two distinct treatments

In this seventh example, we will store a single measure where two different treatments have been performed (for example a measure at an interface between two materials).

The following structure represents the base structure of the file:

```
file.h5
+-- Data (group) -> Name = "Measure"
|   +-- Data_0 (group) -> Name = "Water"
|   |   +-- Raw_data (dataset)
|   |   +-- PSD (dataset)
|   |   +-- Frequency (dataset)
|   |   +-- Treat_0 (group) -> Name = "Treat_5GHz"
|   |   |   +-- Shift (dataset)
|   |   |   +-- Shift_std (dataset)
|   |   |   +-- Linewidth (dataset)
|   |   |   +-- Linewidth_std (dataset)
|   |   +-- Treat_1 (group) -> Name = "Treat_10GHz"
|   |   |   +-- Shift (dataset)
|   |   |   +-- Shift_std (dataset)
|   |   |   +-- Linewidth (dataset)
|   |   |   +-- Linewidth_std (dataset)
```

Note that we have here added arrows and an example of the value of the "Name" attributes. In this case, all the steps of the treatment around 5GHz are stored in the "Treat_0" group and the ones around 10GHz in the "Treat_1" group. The nomenclature of the attribute(s) used to store the parameters of the treatment is not specified.

A.8 A single mapping stored as a single measure

In this eighth example, we want to store a mapping of a sample. This mapping has been obtained with a spectrometer that returns an array of points for all the points mapped. To clarify this example, we will indicate the dimension of each dataset here between brackets.

The following structure represents the base structure of the file:

```
file.h5
+-- Data (group) -> Name = "Measure"
|   +-- Data_0 (group) -> Name = "Sample"
|   |   +-- Raw_data (dataset) [X, Y, M]
|   |   +-- PSD (dataset) [X, Y, N]
|   |   +-- Frequency (dataset) [N]
|   |   +-- Abscissa_0 (dataset) [X] -> Name = "x (mm)"
|   |   +-- Abscissa_1 (dataset) [Y] -> Name = "y (mm)"
|   |   +-- Treat_1 (group) -> Name = "Treat"
|   |   |   +-- Shift (dataset) [X, Y]
|   |   |   +-- Shift_std (dataset) [X, Y]
|   |   |   +-- Linewidth (dataset) [X, Y]
|   |   |   +-- Linewidth_std (dataset) [X, Y]
```

Note that we have here added arrows and an example of the value of the "Name" attributes.

A.9 A series of mapping over the same field of view stored as a single measure

In this ninth example, we are in the situation where multiple mappings of same dimension have been obtained with a spectrometer that returns an array of points for all the points mapped. In this case, the hierrarchy of the file can be used to reduce the number of datasets, by considering that all the PSD share the same frequency axis and the same field of view.

The following structure represents the base structure of the file:

```
file.h5
+-- Data (group) -> Name = "Measure"
|   +-- Abscissa_0 (dataset) [X] -> Name = "x (mm)"
|   +-- Abscissa_1 (dataset) [Y] -> Name = "y (mm)"
|   +-- Frequency (dataset) [N]
|   +-- Data_0 (group) -> Name = "Day_1"
|   |   +-- Raw_data (dataset) [X, Y, M]
|   |   +-- PSD (dataset) [X, Y, N]
|   |   +-- Treat_1 (group) -> Name = "Treat"
|   |   |   +-- Shift (dataset) [X, Y]
|   |   |   +-- Shift_std (dataset) [X, Y]
|   |   |   +-- Linewidth (dataset) [X, Y]
|   |   |   +-- Linewidth_std (dataset) [X, Y]
|   +-- Data_1 (group) -> Name = "Day_2"
|   |   +-- Raw_data (dataset) [X, Y, M]
|   |   +-- PSD (dataset) [X, Y, N]
|   |   +-- Treat_1 (group) -> Name = "Treat"
|   |   |   +-- Shift (dataset) [X, Y]
|   |   |   +-- Shift_std (dataset) [X, Y]
|   |   |   +-- Linewidth (dataset) [X, Y]
|   |   |   +-- Linewidth_std (dataset) [X, Y]
```

Note that we have here added arrows and an example of the value of the "Name" attributes.

A.10 A series of mapping over the same field of view stored as multiple measures

In this tenth example, we are in the situation where multiple mappings of same dimension have been obtained with a spectrometer that can't return an array of points for all the points mapped, but returns them one by

one. Because it would be impractical to create groups for each point, we encourage users to compile their data into a single dataset, and refer to example 9.

A.11 A series of mapping obtained with different spectrometers and with different field of view

In this eleventh example, we are in the situation where multiple mappings of different dimensions have been obtained with different spectrometers that all return an array of points for all the points mapped. In this case, the hierrarchy of the file cannot be used to reduce the number of datasets, and each group will need its own abscissa and frequency datasets.

The following structure represents the base structure of the file:

```
file.h5
+-- Data (group) -> Name = "Measure"
|   +-- Data_0 (group) -> Name = "VIPA"
|   |   +-- Raw_data (dataset) [X, Y, M]
|   |   +-- PSD (dataset) [X, Y, N]
|   |   +-- Abscissa_0 (dataset) [X] -> Name = "x (mm)"
|   |   +-- Abscissa_1 (dataset) [Y] -> Name = "y (mm)"
|   |   +-- Frequency (dataset) [N]
|   |   +-- Treat_1 (group) -> Name = "Treat"
|   |   |   +-- Shift (dataset) [X, Y]
|   |   |   +-- Shift_std (dataset) [X, Y]
|   |   |   +-- Linewidth (dataset) [X, Y]
|   |   |   +-- Linewidth_std (dataset) [X, Y]
|   +-- Data_1 (group) -> Name = "TFP"
|   |   +-- Raw_data (dataset) [X, Y, M]
|   |   +-- PSD (dataset) [X, Y, N]
|   |   +-- Abscissa_0 (dataset) [X] -> Name = "x (mm)"
|   |   +-- Abscissa_1 (dataset) [Y] -> Name = "y (mm)"
|   |   +-- Frequency (dataset) [N]
|   |   +-- Treat_1 (group) -> Name = "Treat"
|   |   |   +-- Shift (dataset) [X, Y]
|   |   |   +-- Shift_std (dataset) [X, Y]
|   |   |   +-- Linewidth (dataset) [X, Y]
|   |   |   +-- Linewidth_std (dataset) [X, Y]
```

Note that we have here added arrows and an example of the value of the "Name" attributes.

Appendix B

Examples of treatment pipelines

B.1 Treatment of a TFP spectrometer

We here present the code that was used to treat the data obtained from a TFP spectrometer. This code is meant to be used as an example of how to write a treatment function for the GUI compatibility.

1. We first extract all the PSDs and frequency arrays that are child of the element that has been selected. To do that, we need to go through all the higher layers of our wrapper until our data is found. This is done using the following code:

```
def get_paths_childs(wrp, path = "", frequency = None):
child, freq = [], []
if "Frequency" in wrp.data.keys():
frequency = path+"/Frequency"
for e in wrp.data.keys():
if isinstance(wrp.data[e], wrapper.Wrapper):
    ce, fe = get_paths_childs(wrp.data[e], path+"/"+e, frequency=frequency)
    child += ce
    freq += fe
else:
    if e == "Power Spectral Density":
        freq.append(frequency)
        child.append(path+"/"+e)
return child, freq

# Get the selected data wrapper and frequency array
wrp_temp = wrp
path_loc = path.split("/")[1:]
if "Frequency" in wrp.data.keys(): frequency = wrp.data["Frequency"]
else: frequency = None
for e in path_loc:
if "Frequency" in wrp_temp.data[e].data.keys():
frequency = wrp_temp.data[e].data["Frequency"]
if isinstance(wrp_temp.data[e], wrapper.Wrapper):
wrp_temp = wrp_temp.data[e]

childs, frequency = get_paths_childs(wrp_temp, path)
```

2. From there we have a choice to make: either we treat each PSD individually or all at once, from some globally defined parameters. We therefore need to ask the user if he wants to treat all of them with the same parameters or each one individually. This is done using the following code:

```
# Display a dialog box to ask the user if he wants to treat all of them
    with the same parameters or each one individually
```

```
msgBox = qtw.QMessageBox()
msgBox.setText(f"There are {len(childs)} PSD in the selected data. Do you want to treat all of them at once?")
msgBox.setStandardButtons(qtw.QMessageBox.Yes | qtw.QMessageBox.No | qtw.QMessageBox.Cancel)
msgBox.setDefaultButton(qtw.QMessageBox.Yes)
ret = msgBox.exec()
if ret == qtw.QMessageBox.Yes:
# Treat all PSD at once
elif ret == qtw.QMessageBox.No:
# Treat each PSD individually
```

3. In both cases, we will want to open a window to enter the parameters of the treatment. In the first case, where all the spectra are treated at once, we open the window with all the spectra as parameters. In the second case, where each spectrum is treated individually, we will have a "for" loop to open the window for each spectrum. This is done using the following code:

```
from ParameterCurve.main import TFP_treat

if ret == qtw.QMessageBox.Yes:
dialog = TFP_treat(parent = parent, wrp_base = wrp, path_base = path, path_curves = childs, path_frequency = frequency)
if dialog.exec_() == qtw.QDialog.Accepted:
# Store all the treated values
elif ret == qtw.QMessageBox.No:
for c,f in zip(childs, frequency):
dialog = TFP_treat(parent = parent, wrp_base = wrp, path_base = path, path_curves = childs, path_frequency = frequency)
if dialog.exec_() == qtw.QDialog.Accepted:
    # Store the treated values
```

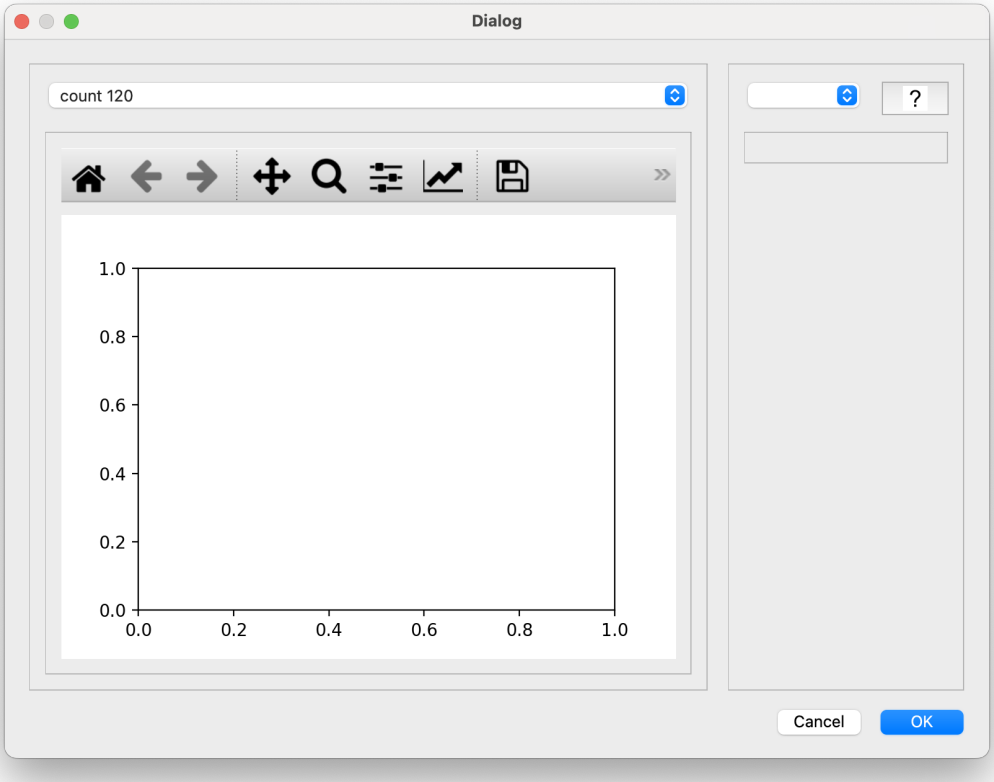
Note that here we are importing another GUI window from the `ParameterCurve` package. The definition of this GUI window is therefore the next step. Let's now look into this `TFP_treat` class.

4. Opening the "HDF5_BLS_GUI/ParameterCurve/main.py" file, we define the `TFP_treat` class as a daughter of the `ParameterCurve` class, which is a GUI window with 4 distinct elements:

   - A combobox to select the curves to plot at the top left of the window.
   - A combobox to select the function to apply at the top right of the window.
   - A graph frame to display the curves at the bottom left of the window.
   - A frame to display the parameters of the treatment at the bottom right of the window, together with buttons to apply the treatment and to close the window.

```
class TFP_treat(ParameterCurve):
def __init__(self, parent=None, wrp_base = None, path_base = None, path_curves = None, path_frequency = None):
super().__init__(parent, wrp_base.get_child(path_base))
```

This initializes the `ParameterCurve` class with the wrapper corresponding to all the curves we are going to treat. Giving the class the path of the selected curves displays them by default in the combobox. Here is an image of a raw ParameterCurve window after this simple initialization:



5. We can now add functionailities to the GUI. First, we display the curve that we select in the combobobx. To do so, we will call the `handle_data` function when the combobox is changed. This function will extract the data from the wrapper corresponding to the selected curve and plot it on the graph.

```
def __init__(self, parent=None, wrp_base = None, path_base = None,
    path_curves = None, path_frequency = None, frequency = None):
super().__init__(parent, wrp_base.get_child(path_base))

if frequency is None:
self.path_curves = path_curves
self.path_frequency = path_frequency
self.path_frequency_unique = None
else:
self.path_curves = None
self.path_frequency = None
self.path_frequency_unique = frequency

# Initializes the graph
self.cb_curves.currentIndexChanged.connect(self.handle_data)
```

Note that we have also stored the paths to the frequencies associated to the curves in respectively the `path_frequency` and `path_curves` attributes. In the case where only one array is shown, then the path to the frequency array is stored in the `path_frequency_unique` attribute.

6. The `handle_data` function extracts the path associated to a value in the combobox and gets both the Power Spectral Density and the frequency array from the wrapper corresponding to the selected curve. It then plots the data on the graph.

```
def handle_data(self):
```

```
"""
Plots the curve that is currently selected in the combobox. This function
    also defines self.data and updates the parameters.
"""
# Extract the raw data from the wrapper corresponding to the selected curve
     in the combobox
wrp = self.wrapper

if len(self.combobox_curve_codes) > 1:
path = self.combobox_curve_codes[self.combobox_curve_names.index(self.
    cb_curves.currentText())]
path = path[5:]

if type(path) == list:
    for e in path:
        wrp = wrp.data[e]
else:
    wrp = wrp.data[path]

self.data = wrp.data["Power Spectral Density"]
if self.path_frequency is None:
    self.frequency = wrp.get_child(self.path_frequency_unique)[:]
else:
    self.frequency = wrp.get_child(self.path_frequency[self.path_curves.
        index(path+"/Power Spectral Density")])[:]

# Plot the data
self.graph_canvas.axes.cla()

self.graph_canvas.axes.plot(self.frequency, self.data)
self.graph_canvas.axes.set_xlabel("Frequency Shift (GHz)")
self.graph_canvas.axes.set_ylabel("Intensity (AU)")
self.graph_canvas.draw()
self.update_parameters()
```

Note that the last line of this function is calling the function `update_parameters`. This function will update the list of parameters needed to run the treatment.

7. We can define the `update_parameters` function. This function will most likely be common to most treatments. Its goal is to inspect the "treat" module from the `HDF5_BLS` package and extract the list of functions and parameters that are needed automatically. Then it displays the list of functions in the dedicated combobox and the list of parameters in the dedicated frame. If the nomenclature of the parameters and the definition of the treatment function follow a fixed nomenclature, this function automatically links the graph with the relevant parameters so that the graph becomes interactive. Further information about how to develop new treatment functions can be found in section Adding a new treatment function. Out of curiosity, here is the detail of the code of this function:

```
def update_parameters(self):
def initialize_parameters(self, module):
functions = [func for func in getmembers(module, isfunction)]
function_names = [func[0] for func in functions]
functions = [func[1] for func in functions]

self.cb_functions.clear()
self.cb_functions.addItems(function_names)
self.cb_functions.setCurrentIndex(0)
self.cb_functions.currentIndexChanged.connect(lambda: self.
    show_parameters_function(functions, function_names))

return functions, function_names

def setup_button_help_function(self, functions, function_names):
def show_help_function():
```

```
    docstring = functions[function_names.index(self.function_name)].__doc__
        or ""
    msgBox = HelpFunction(self, self.function_name, docstring)
    msgBox.exec_()

self.b_helpFunction.clicked.connect(show_help_function)

def onclick_x0(event = None):
if event.inaxes:
    x = float(event.xdata) * 1e6//1
    x = x/1e6
    self.parameters["center_frequency"]["line_edit"].setText(str(x))

def onclick_linewidth(event = None):
if event.inaxes:
    self.temp_linewidth = float(event.xdata)
    self.graph_canvas.mpl_connect('motion_notify_event', on_drag)

def on_drag(event):
if event.inaxes and event.button == 1:
    x1 = float(event.xdata)
linewidth = abs(x1 - self.temp_linewidth) * 1e6//1
linewidth = linewidth/1e6
self.parameters["linewidth"]["line_edit"].setText(str(linewidth))

# Define the module to be used
import HDF5_BLS.treat as module

# Extracts the functions and the function names from the module
self.functions, self.function_names = initialize_parameters(self, module)

# Sets the combobox with the functions
self.show_parameters_function(self.functions, self.function_names)

# Adds the models in the dedicated combobox.
Models = module.Models()
self.parameters["c_model"]["combobox"].addItems(Models.models.keys())

# Connects the QLineEdit widget to the onclick_x0 function
self.parameters["center_frequency"]["line_edit"].mousePressEvent = lambda
    event: self.graph_canvas.mpl_connect('button_press_event', onclick_x0)

# Connects the QLineEdit widget to the onclick_linewidth function
self.parameters["linewidth"]["line_edit"].mousePressEvent = lambda event:
    self.graph_canvas.mpl_connect('button_press_event', onclick_linewidth)

# Sets the help button to display the function's docstring
setup_button_help_function(self, self.functions, self.function_names)
```

Note that the last line of this function is calling the function `button_help_function`. This function is meant to display the docstring of the function in a dedicated window when the "Help" button is pressed on the interface.

8. The next step is to allow the user to apply the selected function with the parameters defined in the dedicated frame. To do so, we will setup a "Treat" button in the "setup_apply_button" function.

```
def setup_button_apply(self):
"""
Creates the layout for the buttons to apply the function.
"""
layout = qtw.QGridLayout(self.frame_confirmParam)

button_treat = qtw.QPushButton()
button_treat.setText("Treat")
```

```
button_treat.clicked.connect(self.apply_function)

layout.addWidget(button_treat, 0, 0, 1, 1)
```

Note that the button is connected to the `apply_function` function. This function returns the entered parameters of the treatement so that it can be performed.

9. The `apply_function` function is meant to read the parameters of the treatment and return an object that will allow the treatment on either one or multiple arrays. This function is developped as a switch between the different treatment functions that were defined in the dedicated combobox. Therefore its structure is the following:

```
def apply_function(self):
"""
Creates the layout for the buttons to apply the function.
"""
func = self.functions[self.function_names.index(self.function_name)]

if self.function_name == "unicorn":
# Extract the parameters proper to the "unicorn" treatment
elif self.function_name == "elf":
# Extract the parameters proper to the "elf" treatment
```

As a more concrete example, here is the code for the `fit_model_v0` treatment function:

```
def apply_function(self):
"""
Extracts the parameters from the GUI and pplies the treatment to the data.
"""
func = self.functions[self.function_names.index(self.function_name)]

if self.function_name == "fit_model_v0":
# Extract the parameters of the function
dic = {}
try:
    dic["center_frequency"] = float(self.parameters["center_frequency"]["
        line_edit"].text())
    dic["linewidth"] = float(self.parameters["linewidth"]["line_edit"].text
        ())
    dic["normalize"] = not bool(self.parameters["normalize"]["checkbox"].
        text())
    dic["c_model"] = str(self.parameters["c_model"]["combobox"].currentText
        ())
    dic["fit_S_and_AS"] = not bool(self.parameters["fit_S_and_AS"]["
        checkbox"].checkState())
    dic["window_peak_find"] = float(self.parameters["window_peak_find"]["
        line_edit"].text())
    dic["window_peak_fit"] = float(self.parameters["window_peak_fit"]["
        line_edit"].text())
    dic["correct_elastic"] = not bool(self.parameters["correct_elastic"]["
        checkbox"].checkState())
    IR_wndw = self.parameters["IR_wndw"]["line_edit"].text()
    if IR_wndw == "None":
        dic["IR_wndw"] = None
    else:
        dic["IR_wndw"] = IR_wndw.replace("(","").replace(")","").replace("
            ","")
        dic["IR_wndw"] = tuple(map(float, dic["IR_wndw"].split(",")))

    self.parameter_return["Parameters"] = dic
    self.parameter_return["Function"] = func

```

```
29      qtw.QMessageBox.information(self, "Treatment parameters stored", "The
            parameters for the treatment have been stored. You can now close the
             window to apply the treatment.")
30
31  except:
32      qtw.QMessageBox.warning(self, "Error while retrieving parameters", "An
            error happened while retrieving the parameters")
```

Note that the parameters are stored in the "parameter_return" dictionnary. This dictionnary is meant to be returned to the `treat_ui` module, which will then apply the treatment to the data.

Appendix C

Specification sheet of the project

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.*