The HDF5_BLS file format and its user interface

The International BioBrillouin Society

March 11, 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

The vision of the project is to unify all BLS data into a single HDF5 file. This file will be used to store all the data and metadata of the BLS. The data will be stored in a hierarchical structure, all the data related to BLS being stored the "Data" group. This choice allows results from other techinques to be stored in the same file, only in other, independent groups, to minimize the risk of nomenclature competition between techinques. The "Data" group will contain all the data. These data are arranged in groups, whose identifiers are "Data_i". Each group can only contain one measure, in the form of an array, whose dimension is however not restricted. This measure will be called "Raw_data". The attributes of the measure are stored in the group attributes.

The following structure represents the base structure of the file:

```
file.h5
+-- Data (group)
|   +-- Data_1 (group)
|   |   +-- Raw_data (dataset)
```

Attributes

The attributes will follow a hierarchical structure. The attributes that apply to all "Data_i" will be stored in the "Data" group, while the parameters that apply to a specific "Data_i" will be stored in the "Data_i" group. All the attributes are stored as text. Attributes are then divided in four categories:

- 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.
- Attributes that are specific to the sample, such as the date of the measurement, the name of the sample, etc. These attributes are recognized by the capital letter word "MEASURE" in the name of the attribute.
- 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.
- Attributes that are used inside the HDF5 file, such as the name of the group, the name of the dataset, etc. These attributes are the only ones without a prefix.

The name of the attribues contains the unit of the attribute if it has units, in the shape of an underscore followed by the unit in parenthesis. Some parameters that can be represented by a series of norms will also be defined in a given norm, such as the ISO8601 for the date. These norms are however 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 exposire of the sample given in seconds
- "MEASURE.Date_of_measurement" is the date of the measurement.
- "FILEPROP.Name" is the name of the file.

To unify the name of attributes, a spreadsheet is accesible, containing all the attributes and their units. This spreadsheet will be updated as new attributes are added to the project and defined with a version number that will also be stored in the attribtutes of each data attributes (under FILEPROP.version). This spreadsheet is meant to be exported in a CSV file that can be used to update the attributes of the data.

Meta-files

It can be useful to store in a same file, measures coming from different instruments, taken in different conditions, or that we just want to separate from other groups of measures. In that end, we propose a tree-like structure of the HDF5 file, where each group can contain sub-groups, which can also contain sub-groups etc. In order to unify the way we access these groups, we propose to identify them by a unique identifier of the form "Data_i", where "i" is an integer. Here is an example of the structure of a meta-file:

```
file.h5
+-- Data
|   +-- Data_1
|   |   +-- Data_1
|   |   |   +-- ...
|   |   +-- Data_2
|   |   |   +-- ...
|   |   +-- ...
|   +-- Data_2
|   |   +-- Data_1
|   |   |   +-- Data_1
|   |   |   |   +-- ...
|   |   |   +-- Data_2
|   |   |   |   +-- ...
|   |   |   ...
|   |   ...
|   ...
```

Complete structure definition

The file is intended to be used to store not only raw data but also treated data together with the parameters used for treatment. As such, we propose to complete the structure defined above with the following structure:

```
file.h5
+-- Data (group)
|   +-- Data_0 (group)
|   |   +-- Raw_data (dataset)
|   |   +-- Abscissa_0 (dataset)
|   |   +-- Abscissa_1 (dataset)
|   |   +-- ...
|   |   +-- PSD (dataset)
|   |   +-- Frequency (dataset)
|   |   +-- Treat_0(group)
```

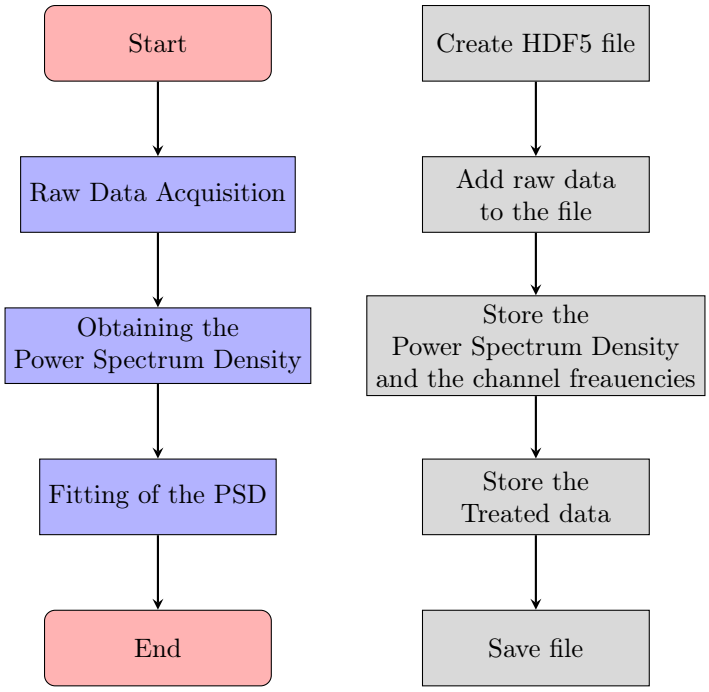
```
|   |   |   +-- Shift (dataset)
|   |   |   +-- Shift_std (dataset)
|   |   |   +-- Linewidth (dataset)
|   |   |   +-- Linewidth_std (dataset)
|   |   +-- Treat_1(group)
|   |   ...
|   +-- Data_1 (group)
|   ...
```

The nomenclature is defined as follows:

- **"Data_i"** is the identifier of the group containing the i-th measure.
- **"Raw_data"** is the identifier of the dataset containing the raw data of the measure stored in the "Data_i" group.
- **"Abscissa_i"** is the identifier of the dataset containing the i-th abscissa array of "Raw_data" and "PSD". The dimension of this dataset is not forced to one.
- **"PSD"** is the identifier of the dataset of the Power Spectrum Density of the measure, associated with the "Frequency" dataset. "Raw_data" and "PSD" are arrays of same shape. We impose the last dimension(s) of "PSD" to be the same as the dimension(s) of "Frequency".
- **"Frequency"** is the identifier of the dataset containing the frequency axis of the "PSD".
- **"Treat_i"** is the identifier of the group containing the treated data of the i-th measure.
- **"Shift"** is the identifier of the dataset containing the values of the fitted frequency shifts.
- **"Shift_std"** is the identifier of the dataset containing the standard deviation of the fitted frequency shifts.
- **"Linewidth"** is the identifier of the dataset containing the values of the fitted linewidths.
- **"Linewidth_std"** is the identifier of the dataset containing the standard deviation of the fitted linewidths.

Pipeline

The goal of this software is to go from a storaged data file to a usable data set with a reproducible, stable and unified treatment protocol. To schematize this treatment protocol, we propose the following diagram:


Start
Raw Data Acquisition
Obtaining the
Power Spectrum Density
Fitting of the PSD
End
Create HDF5 file
Add raw data
to the file
Store the
Power Spectrum Density
and the channel freauencies
Store the
Treated data
Save file

Chapter 1

Getting Started

1.1 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.

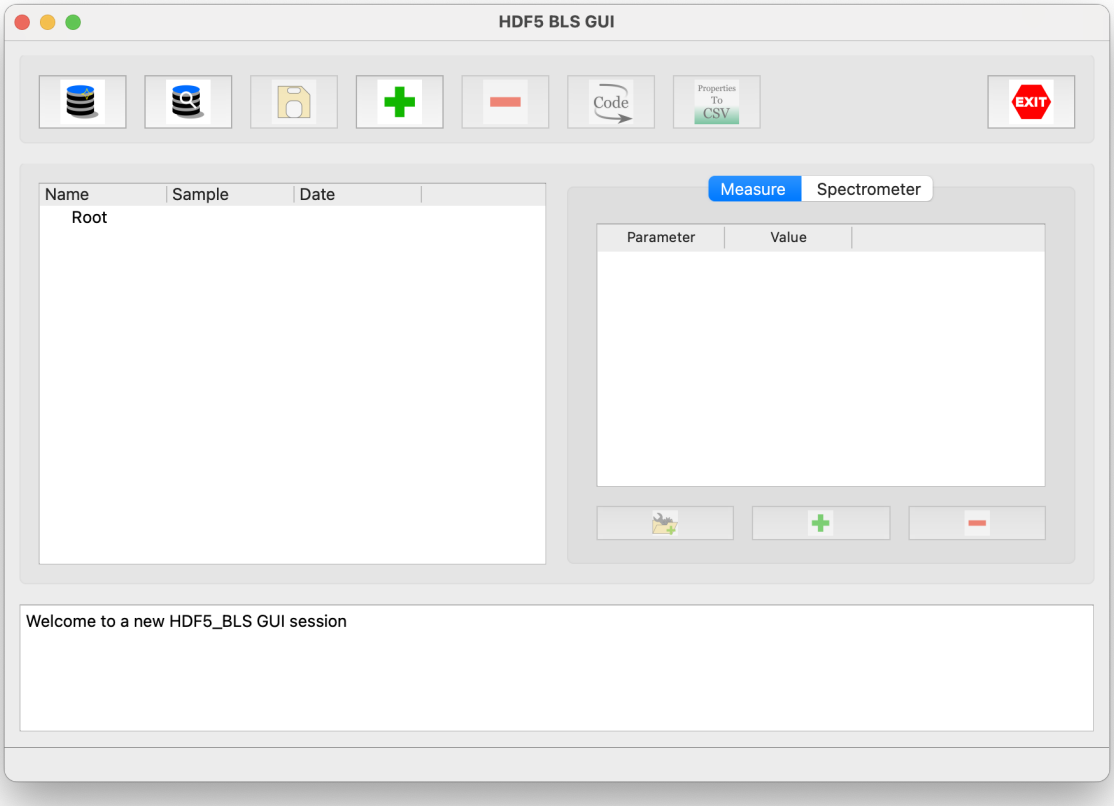
- Step 4: Run the `HDF5_BLS_GUI/main.py` file with

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

1.2 First Steps

1.2.1 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.

Chapter 2

Development Guide

Why are you here?

- I want to import my data to the HDF5 file format
- I want to extract information from my data

2.1 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.

2.1.1 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:

```
1 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:

```
1 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 :)

2.1.2 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.

2.1.3 Adding a user-specific function to an already supported 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 :)

2.2 Treat data

What is the format of your data?

- I just have raw data coming from the spectrometer
- I have a Spectral Power Density together with a frequnecy vector
- I want to define a new treatment function

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

To do

2.2.2 Treat data to extract information from a Power Spectral Density

Here are the steps to follow for the GUI compatibility:

1. If "type" is the type of your spectrometer, add the function "treat_type" 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
```

2. Add the following parameters to your function:

   - 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):
```

3. Define your function. You can find an example of how it was done for the "TFP" treatment in the appendix.

2.2.3 Adding a new treatment function

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.

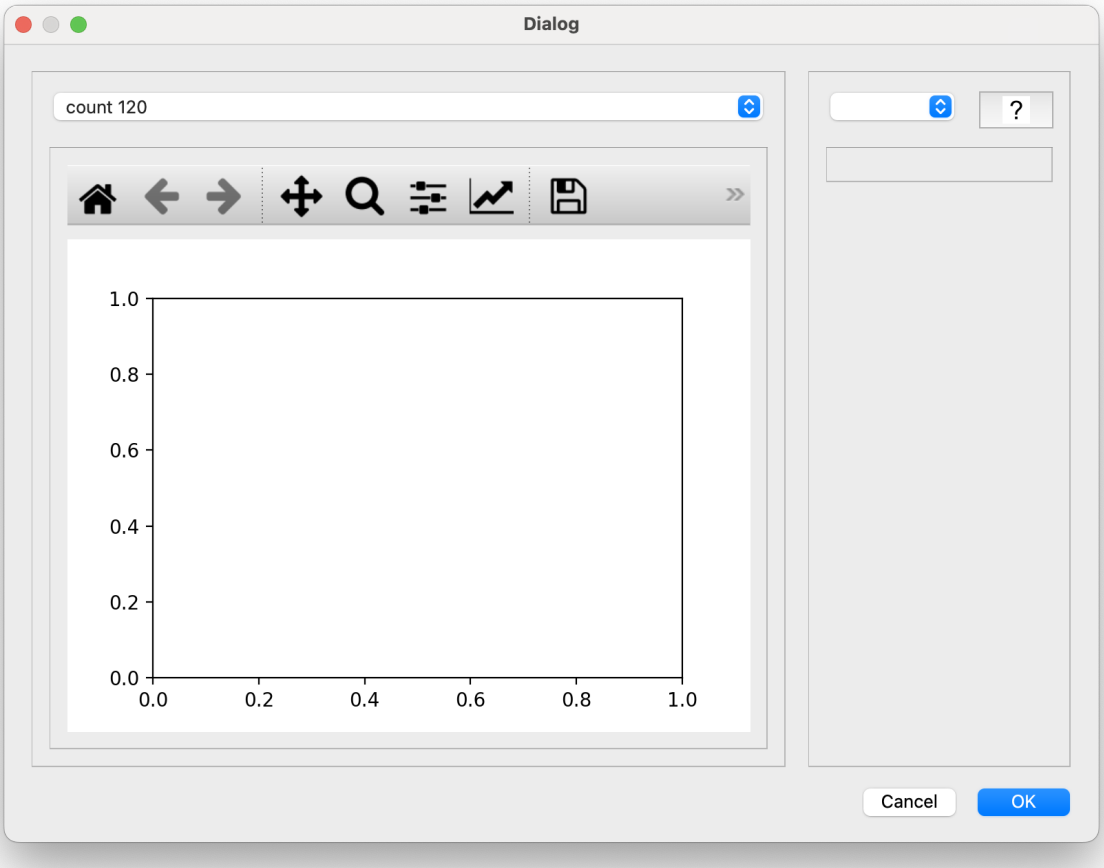
4. Opening the "HDF5_BLS_GUI/ParameterCurve/main.py" file, we define the `TFP_treat` class as a daughter of the `ParameterCurve` class, which is a GUI window with 4 distinct elements:

   - 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))
```

```
11
12          return functions, function_names
13
14      def setup_button_help_function(self, functions, function_names):
15          def show_help_function():
16              docstring = functions[function_names.index(self.function_name)
                    ].__doc__ or ""
17              msgBox = HelpFunction(self, self.function_name, docstring)
18              msgBox.exec_()
19
20          self.b_helpFunction.clicked.connect(show_help_function)
21
22      def onclick_x0(event = None):
23          if event.inaxes:
24              x = float(event.xdata) * 1e6//1
25              x = x/1e6
26              self.parameters["center_frequency"]["line_edit"].setText(str(x)
                    )
27
28      def onclick_linewidth(event = None):
29          if event.inaxes:
30              self.temp_linewidth = float(event.xdata)
31              self.graph_canvas.mpl_connect('motion_notify_event', on_drag)
32
33      def on_drag(event):
34          if event.inaxes and event.button == 1:
35              x1 = float(event.xdata)
36          linewidth = abs(x1 - self.temp_linewidth) * 1e6//1
37          linewidth = linewidth/1e6
38          self.parameters["linewidth"]["line_edit"].setText(str(linewidth))
39
40      # Define the module to be used
41      import HDF5_BLS.treat as module
42
43      # Extracts the functions and the function names from the module
44      self.functions, self.function_names = initialize_parameters(self,
            module)
45
46      # Sets the combobox with the functions
47      self.show_parameters_function(self.functions, self.function_names)
48
49      # Adds the models in the dedicated combobox.
50      Models = module.Models()
51      self.parameters["c_model"]["combobox"].addItems(Models.models.keys())
52
53      # Connects the QLineEdit widget to the onclick_x0 function
54      self.parameters["center_frequency"]["line_edit"].mousePressEvent =
            lambda event: self.graph_canvas.mpl_connect('button_press_event',
            onclick_x0)
55
56      # Connects the QLineEdit widget to the onclick_linewidth function
57      self.parameters["linewidth"]["line_edit"].mousePressEvent = lambda
            event: self.graph_canvas.mpl_connect('button_press_event',
            onclick_linewidth)
58
59      # Sets the help button to display the function's docstring
60      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.

```python
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:

```python
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:

```python
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
```

```
22              else:
23                  dic["IR_wndw"] = IR_wndw.replace("(","").replace(")","").
                        replace(" ","")
24                  dic["IR_wndw"] = tuple(map(float, dic["IR_wndw"].split(",")
                        ))
25
26              self.parameter_return["Parameters"] = dic
27              self.parameter_return["Function"] = func
28
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	Simplicity
1.1	Conceptually simple
1.1.1 1.1.2 1.1.3	**Single measures:** Clearly define how to build a file with a single measure. **Attributes:** Clearly define how to store parameters associated with a single measure. **Hyper parameters:** Clearly define how to store measures that depend on multiple parameters.
1.2	Practically simple
1.2.1	The user should have a GUI that builds the file for him unambiguously.
2	Universality
2.1	Techniques
2.1.1	The format should be able to store any type of spectra from different techniques.
2.2	Dimensionality
2.2.1	The format should be able to store spectra without any dimensionality limitation.
3	Unify Treatment
3.1	Obtention of a Custom PSD
3.1.1 3.1.1	The format should allow any steps leading to the obtention of a doublet {PSD, Frequency} for spectra obtained with any spectrometer. The format should allow the user to easily define a new treatment algorithm for the obtention of a doublet {PSD, Frequency} that is compatible with the rest of the project.
3.2	Unified Treatment
3.2.1	Once a {PSD, Frequency} doublet is obtained, the format should allow to treat unambiguously the data from the GUI with the same treatment algorithm.
3.3	Multiple Treatments
3.3.1	The format should allow the user to apply different treatments on the same set of PSD and Frequency and store the results.
3.4	Process
3.4.1	The format should allow the user to store the process used for the treatment.
4	Expandability for Future Applications
4.1 4.2	The file format should allow the storage of measures depending on an arbitrary number of hyper parameters. The format should unambiguously classify measures by the hyper parameter(s) that were varied for the experiment.
5	Compatibility with Other Techniques
5.1 5.2	The file format should allow the storage of all relevant measures during an experiment (fluorescence, Raman, etc.). All Brillouin data should be stored in a single group to allow other complimentary techniques used in an experiment to be stored in complimentary groups.