Standardization of Brillouin data storage and processing

Pierre Bouvet Carlo Bevilacqua Sebastian Hambura

2025

Contents

1 BrimView web app

You can navigate to https://biobrillouin.org/brimview/ and the webapp should load directly from the browser. We currently only support the latest version of Chrome and Egde. Firefox can be used as well by activating JSPI (a page explaining how to do should automatically show on Firefox).

1.1 Loading sample data

Once the page finishes loading (it might take a few moments), you can load sample data using the dedicated widget, as highlighted in figure 1

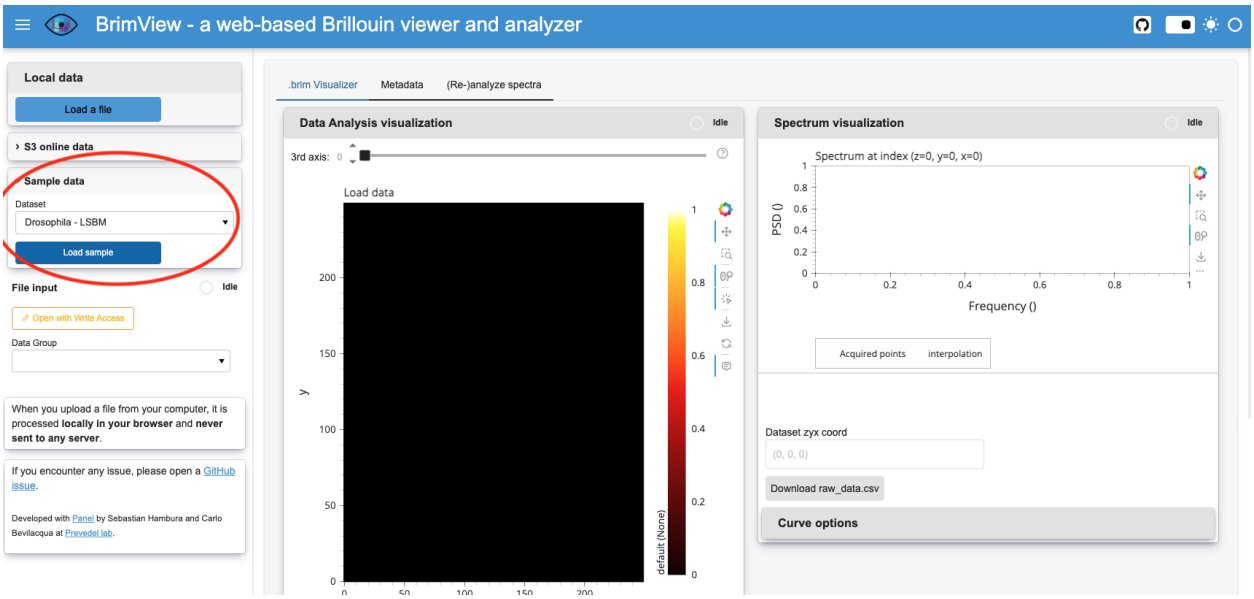


Figure 1: BrimView GUI. The widget to load sample data is encircled in red.

1.2 Exploring the data

Once the image is loaded you can click on any pixel and the corresponding spectrum will show on the right, including a table with the numerical values of the fitted parameters below (figure 2).

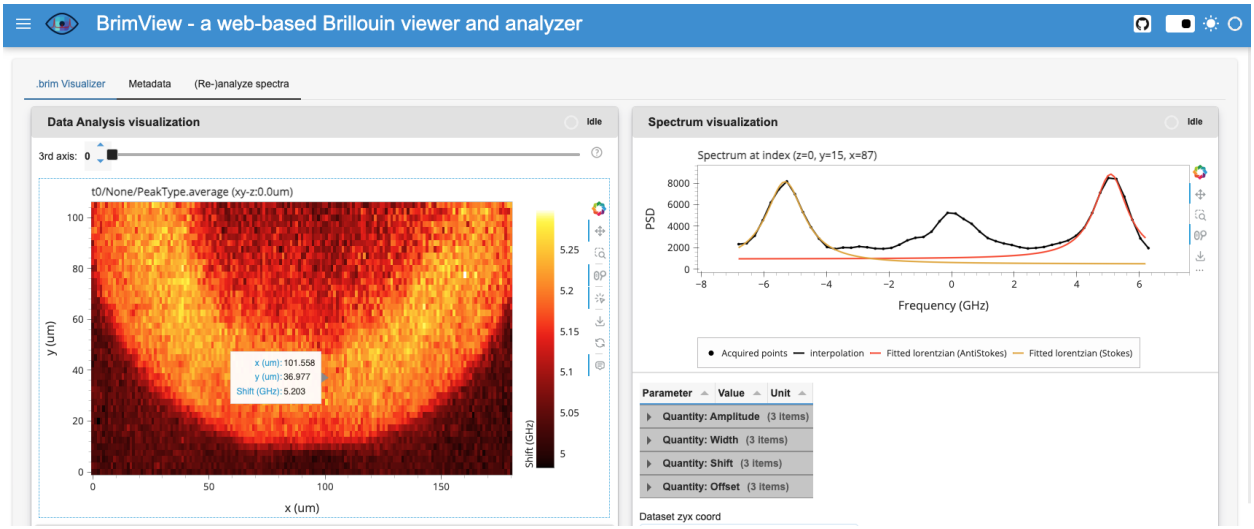


Figure 2: The spectrum and the corresponding fitted parameters can be displayed for each pixel in the image.

You can select which quantity (i.e. shift, linewidth, amplitude, ...), peak (i.e. Stokes, anti-Stokes, average), axis (x, y,z) and color range from the widgets below the image (figure 3).

Figure 3: Different settings for displaying the image can be selected from the widgets below the images.

2 Brimfile

2.1 The brim file format

We defined a new file format - brimfile (= Brillouin imaging) - with the intent of associating spatial maps to their corresponding spectral information and metadata in a well-defined yet general and flexible fashion. More information about the brimfile format can be found here.

2.2 The brimfile Python package

To easily save data to the brim file format or read from it, we developed a Python library called brimfile. The full documentation of the library can be found at https://prevedel-lab.github.io/brimfile/brimfile.html.

2.2.1 Installing brimfile

We recommend installing brimfile in a virtual environment.

After activating the new environment, simply run:

```
pip install brimfile
```

If you also need the support for exporting the analyzed data to OME-Tiff files, you can install the optional dependencies with:

```
pip install "brimfile[export-tiff]"
```

2.2.2 Writing to a brimfile

Let's first of all create a function that generate sample data:

```
import numpy as np

def generate_data():
    def lorentzian(x, x0, w):
        return 1/(1+((x-x0)/(w/2))**2)
    Nx, Ny, Nz = (7, 5, 3) # Number of points in x,y,z
    dx, dy, dz = (0.4, 0.5, 2) # Stepsizes (in um)
    n_points = Nx*Ny*Nz  # total number of points

    width_GHz = 0.4
    width_GHz_arr = np.full((Nz, Ny, Nx), width_GHz)
    shift_GHz_arr = np.empty((Nz, Ny, Nx))
    freq_GHz = np.linspace(6, 9, 151)  # 151 frequency points
    PSD = np.empty((Nz, Ny, Nx, len(freq_GHz)))
    for i in range(Nz):
        for j in range(Ny):
            for k in range(Nx):
                index = k + Nx*j + Ny*Nx*i
                #let's increase the shift linearly to have a readout
                shift_GHz = freq_GHz[0] + (freq_GHz[-1]-freq_GHz[0]) * index/
                    n_points
                spectrum = lorentzian(freq_GHz, shift_GHz, width_GHz)
                shift_GHz_arr[i,j,k] = shift_GHz
                PSD[i, j, k,:] = spectrum

    return PSD, freq_GHz, (dz,dy,dx), shift_GHz_arr, width_GHz_arr
```

We can create a brimfile by doing:

```
from brimfile import File, Data, Metadata, StoreType
from datetime import datetime

filename = 'path/to/your/file.brim.zip'

f = File.create(filename, store_type=StoreType.AUTO)
```

We can then add the spectral data:

```
PSD, freq_GHz, (dz,dy,dx), shift_GHz, width_GHz = generate_data()

d0 = f.create_data_group(PSD, freq_GHz, (dz,dy,dx), name='test1')
```

and the metadata:

```
Attr = Metadata.Item
datetime_now = datetime.now().isoformat()
temp = Attr(22.0, 'C')
md = d0.get_metadata()

md.add(Metadata.Type.Experiment, {'Datetime':datetime_now, 'Temperature':temp})
md.add(Metadata.Type.Optics, {'Wavelength':Attr(660, 'nm')})
# Add some metadata to the local data group
temp = Attr(37.0, 'C')
md.add(Metadata.Type.Experiment, {'Temperature':temp}, local=True)
```

We can also add the result of the fit:

```
ar = d0.create_analysis_results_group({'shift':shift_GHz, 'shift_units': 'GHz',
                                       'width': width_GHz, 'width_units':
                                           'Hz'},
                                       {'shift':shift_GHz, 'shift_units':
                                           'GHz',
                                       'width': width_GHz, 'width_units':
                                           'Hz'},
                                       name = 'test1_analysis')
```

We can finally close the file:

```
f.close()
```

2.2.3 Reading a brimfile

Reading from a brimfile follows a very similar logic as writing.

We first open the file:

```
from brimfile import File, Data, Metadata

filename = 'path/to/your/file.brim.zip'

f = File(filename)
```

We can list all the data group in the file and open the first one:

```
#list all the data groups in the file
data_groups = f.list_data_groups(retrieve_custom_name=True)

# get the first data group in the file
d = f.get_data()
```

We can read the metadata:

```
# get the metadata
md = d.get_metadata()
all_metadata = md.all_to_dict()
# the list of metadata is defined here https://github.com/prevedel-lab/Brillouin
    -standard-file/blob/main/docs/brim_file_metadata.md
time = md['Experiment.Datetime']
time.value
time.units
temp = md['Experiment.Temperature']
md_dict = md.to_dict(Metadata.Type.Experiment)
```

We can get the results of the analysis:

```
#get the list of analysis results in the data group
ar_list = d.list_AnalysisResults(retrieve_custom_name=True)
# get the first analysis results in the data group
ar = d.get_analysis_results()
```

and the corresponding image:

```
# get the image of the shift quantity for the average of the Stokes and anti-Stokes peaks
img, px_size = ar.get_image(Data.AnalysisResults.Quantity.Shift, Data.AnalysisResults.PeakType.average)
# get the units of the shift quantity
u = ar.get_units(Data.AnalysisResults.Quantity.Shift)
```

We can save the image as a TIFF:

```
ar_cls = Data.AnalysisResults
ar.save_image_to_OMETiff(ar_cls.Quantity.Shift, ar_cls.PeakType.average, filename='path/to/your/exported_tiff' )
```

We can also get the spectrum corresponding to a specific pixel:

```
coord = (1,3,4)
PSD, frequency, PSD_units, frequency_units = d.get_spectrum_in_image(coord)
```

We can finally close the file:

```
f.close()
```

3 HDF5_BLS package

The HDF5_BLS package is a Python package for creating BrimX files, a file format based on HDF5, designed to support all Brillouin-related data in a human-readable way, and be integrated to all existing workflows of the community. It is compatible with Python 3.10 and above.

3.1 Installation

We recommend setting up a virtual environment to install the package. This can be done using the following command:

- On Mac terminal

```
python -m venv HDF5_BLS_venv
source HDF5_BLS_venv/bin/activate
```

- On Windows terminal

```
python -m venv HDF5_BLS_venv
HDF5_BLS_venv\Scripts\activate
```

On both OS, the package can be installed using pip:

```
pip install HDF5_BLS
```

3.2 An example

Here we propose to create a synthetic Brillouin dataset that we'll store in a BrimX file. We will store 3 different types of synthetic data:

- A single spectrum
- A synthetic time evolution

- A spatial mapping in 2D
- A z-stack of a sphere

3.2.1 Creating the data

Let's define a general DHO function to create the Brillouin spectra.

```
def DHO(nu, nu0, gamma, a, b):
    return b + a * (gamma*nu0)**2/((nu**2-nu0**2)**2+(gamma*nu)**2)
```

Let's now define a frequency axis to create the spectra.

```
nu = np.linspace(-15, 15, 1024)
```

Let's now build synthetic datasets to image different scenarios:

- 0D: Let's say we have a single spectrum, with a Brillouin shift of 5 GHz and linewidth of 1 GHz. Let's give the spectrum an amplitude of 1 and no offset.

```
shift_0D = 5
linewidth_0D = 1
PSD_0D = DHO(nu, shift_0D, linewidth_0D, 1, 0)
```

- 1D: Let's do the same for a 1D dataset now, let's think about a simple time evolution of a sample, where the shift rises quadratically from 5 to 6 GHz over time, and the linewidth from 1 to 2 GHz linearly. Here again, let's keep an amplitude of 1 with no offset.

```
time_1D = np.linspace(0, 10, 100)
shift_1D = 5 + time**2/100
linewidth_1D = np.linspace(1, 2, 100)
for s, l in zip(shift_1D, linewidth_1D):
    PSD_1D = DHO(nu, s, l, 1, 0)
```

- 2D: Let's consider a mapping showing a diagonal shift gradient from 5 to 6 GHz, and a diagonal linewidth gradient from 1 to 2 GHz. We keep an amplitude of 1 with no offset.

```
gradient_x = np.linspace(0, 1, 50)
gradient_y = np.linspace(0, 1, 50)
temp = (np.outer(gradient_y, gradient_x) + np.outer(gradient_y, gradient_x)
    )/2
shift_2D = temp + 5
linewidth_2D = temp + 1
PSD_2D = np.zeros((50, 50, 1024))
for i in range(50):
    for j in range(50):
        PSD_2D[i, j] = DHO(nu, shift_2D[i, j], linewidth_2D[i, j], 1, 0)
```

- 3D: Let's now think about a z-stack. In this example we'll do a z-stack of a sphere embedded in a solution. The solution will have a shift of 5GHz and linewidth 1GHz, while the sphere will have a shift of 6GHz and linewidth 2GHz. Again, we'll keep an amplitude of 1 with no offset. Again, we have an amplitude of 1 and an offset of 0 for all spectra.

```
x = np.linspace(-1, 1, 50)
y = np.linspace(-1, 1, 50)
z = np.linspace(-1, 1, 50)
shift_3D = np.zeros((50, 50, 50))
linewidth_3D = np.zeros((50, 50, 50))
PSD_3D = np.zeros((50, 50, 50, 1024))
for i in range(50):
    for j in range(50):
        for k in range(50):
```

```
            if (x[i]**2 + y[j]**2 + z[k]**2) > 1:
                shift_3D[i, j, k] = 5
                linewidth_3D[i, j, k] = 1
            else:
                shift_3D[i, j, k] = 6
                linewidth_3D[i, j, k] = 2
            PSD_3D[i, j, k] = DHO(nu, shift_3D[i, j, k], linewidth_3D[i, j,
                k], 1, 0)
```

You can of course look at the PSD, shift and linewidth arrays with your favorite visualization tools (e.g. matplotlib).

3.2.2 Adding the data to a BrimX file

Here we're going to structure our file to store the four different datasets we've created. Each dataset will be stored in a separate group that we'll name with example names.

First we create the BrimX file at a given path, and define an object "wrp" to interact with it.

```
wrp = wrapper.Wrapper('path/to/file.h5')
```

For this example, we will structure the file as follows:

```
file.h5
|- Brillouin
|  |- A single spectrum
|  |- A time series
|  |- A mapping
|  |- A z-stack
```

In each of these groups, we'll store the corresponding Power spectral density datasets, frequency array and shift and linewidth arrays. To do so, we will use the three following functions:

- "add_frequency": to store the frequency array
- "add_PSD": to store the Power spectral density array
- "add_treated_data": to store the shift and linewidth arrays

Each of these functions work the same way:

1. You indicate the dataset to add.
2. You specify where the dataset will be stored in the file (argument "parent_group").
3. You provide a name for the dataset (argument "name").

Because you might need in the future to add other treatment modalities, we have made it so that all the datasets obtained after treatment are added in a same group. Therefore instead of "name" the "add_treated_data" function takes "name_group" as argument, which is the name of the group where all the treated datasets will be stored. The names of the datasets themselves will be the type of data by default (e.g. "Shift" for the shift array, "Linewidth" for the linewidth array, etc.).

With the provided code, this gives us:

```
# Adding the 0D datasets
wrp.add_frequency(data=nu, parent_group="Brillouin/A single spectrum", name = "
    Frequency")
wrp.add_PSD(data=PSD_0D, parent_group="Brillouin/A single spectrum", name = "PSD
    ")
wrp.add_treated_data(parent_group="Brillouin/A single spectrum", name_group = "
    Treated Data", shift = shift_0D, linewidth = linewidth_0D)

# Adding the 1D datasets
```

```
wrp.add_frequency(data=nu, parent_group="Brillouin/A time series", name = "
    Frequency")
wrp.add_PSD(data=PSD_1D, parent_group="Brillouin/A time series", name = "PSD")
wrp.add_treated_data(parent_group="Brillouin/A time series", name_group = "
    Treated Data", shift = shift_1D, linewidth = linewidth_1D)

# Adding the 2D datasets
wrp.add_frequency(data=nu, parent_group="Brillouin/A mapping", name = "Frequency
    ")
wrp.add_PSD(data=PSD_2D, parent_group="Brillouin/A mapping", name = "PSD")
wrp.add_treated_data(parent_group="Brillouin/A mapping", name_group = "Treated
    Data", shift = shift_2D, linewidth = linewidth_2D)

# Adding the 3D datasets
wrp.add_frequency(data=nu, parent_group="Brillouin/A z-stack", name = "Frequency
    ")
wrp.add_PSD(data=PSD_3D, parent_group="Brillouin/A z-stack", name = "PSD")
wrp.add_treated_data(parent_group="Brillouin/A z-stack", name_group = "Treated
    Data", shift = shift_3D, linewidth = linewidth_3D)
```

You are now the proud owner of a BrimX file containing synthetic Brillouin datasets!
Note: you can add other types of datasets to the BrimX file. The available types are:

- "Frequency" with the function "add_frequency"
- "PSD" with the function "add_PSD"
- "Other" with the function "add_other", this can be anything you want to store (grayscale images, Raman spectra, fluorescence maps...)
- "Raw Data" with the function "add_raw_data", where you can store your data "as is" (what the spectrometer returns).
- "Abscissa" with the function "add_abscissa" to add abscissa for your axis. Please refer to footnote [1]

You can also add other types of datasets after treatment, in that case you will use the "add_treated_data" function and just specify the datasets you want to add from this list:

- A record of shift values -> argument "shift"
- A record of linewidth values -> argument "linewidth"
- A record of the values of amplitude -> argument "amplitude"
- Values for the loss tangent -> argument "BLT"
- A record for the errors on the shift values -> argument "shift_err"
- A record for the errors on the linewidth values -> argument "linewidth_err"
- A record for the errors on the amplitude values -> argument "amplitude_err"
- A record for the errors on the BLT values -> argument "BLT_err"

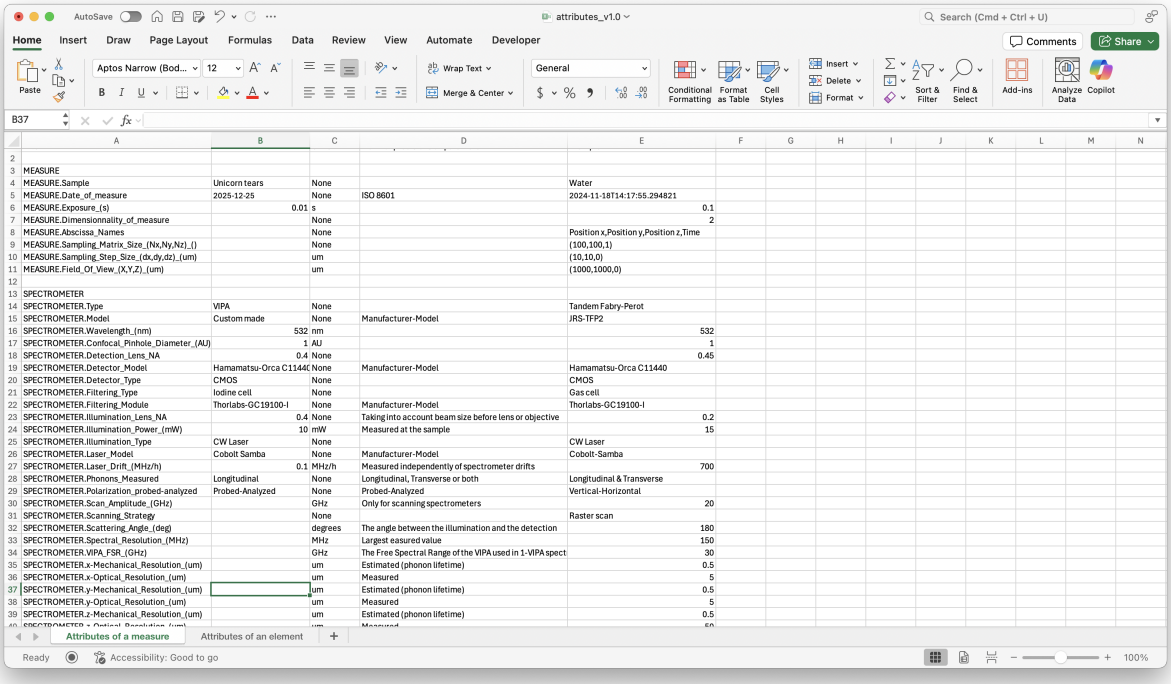
[1] In that case, you need to provide, aside from the dataset itself, the place where to store it and its name, the units of the abscissa (attribute "unit" given as a string, for example "mm" or "K"), the first dimension of the PSD it applies to and the last dimension of the PSD it applies to (attributes "dim_strart" and "dim_end" respectively, given as integers and with the usual convention: starting from 0, starting index included, ending index excluded). For example:

```
    t = np.linspace(0,10,100) # 100 values of time ranging from 0 to 10s
    PSD = np.random.random((100,512)) # 100 PSD associated to the time array
    wrp.add_PSD(data = PSD, parent_group = "Brillouin/A new time series", name =
        "PSD")
    wrp.add_abscissa(data = t, parent_group = "Brillouin/A new time series",
        name = "Time", unit = "s", dim_start = 0, dim_end = 1)
```

3.2.3 Adding metadata to the BrimX file

To add metadata to the BrimX file, we will choose the easy way for this example, and just edit the standard spreadsheet given with the package and downloadable at this link: https://github.com/bio-brillouin/HDF5_BLS/blob/main/spreadsheets/attributes_v1.0.xlsx.

A list of established parameters is given, their name, units and definition is written in the different columns. For this example, we are filling the datasheet as presented on figure 4.



	A	B	C	D	E
2					
3	MEASURE				
4	MEASURE.Sample	Unicorn tears	None		Water
5	MEASURE.Date_of_measure	2025-12-25	None	ISO 8601	2024-11-18T14:17:55.294821
6	MEASURE.Exposure_(s)	0.01	s		0.1
7	MEASURE.Dimensionnality_of_measure		None		2
8	MEASURE.Abscissa_Names		None		Position x,Position y,Position z,Time
9	MEASURE.Sampling_Matrix_Size_(Nx,Ny,Nz)_()		None		(100,100,1)
10	MEASURE.Sampling_Step_Size_(dx,dy,dz)_(um)		um		(10,10,0)
11	MEASURE.Field_Of_View_(X,Y,Z)_(um)		um		(1000,1000,0)
12					
13	SPECTROMETER				
14	SPECTROMETER.Type	VIPA	None		Tandem Fabry-Perot
15	SPECTROMETER.Model	Custom made	None	Manufacturer-Model	JRS-TFP2
16	SPECTROMETER.Wavelength_(nm)	532	nm		532
17	SPECTROMETER.Confocal_Pinhole_Diameter_(AU)	1	AU		1
18	SPECTROMETER.Detection_Lens_NA	0.4	None		0.45
19	SPECTROMETER.Detector_Model	Hamamatsu-Orca C11440	None	Manufacturer-Model	Hamamatsu-Orca C11440
20	SPECTROMETER.Detector_Type	CMOS	None		CMOS
21	SPECTROMETER.Filtering_Type	Iodine cell	None		Gas cell
22	SPECTROMETER.Filtering_Module	Thorlabs-GC19100-I	None	Manufacturer-Model	Thorlabs-GC19100-I
23	SPECTROMETER.Illumination_Lens_NA	0.4	None	Taking into account beam size before lens or objective	0.2
24	SPECTROMETER.Illumination_Power_(mW)	10	mW	Measured at the sample	15
25	SPECTROMETER.Illumination_Type	CW Laser	None		CW Laser
26	SPECTROMETER.Laser_Model	Cobolt Samba	None	Manufacturer-Model	Cobolt-Samba
27	SPECTROMETER.Laser_Drift_(MHz/h)	0.1	MHz/h	Measured independently of spectrometer drifts	700
28	SPECTROMETER.Phonons_Measured	Longitudinal	None	Longitudinal, Transverse or both	Longitudinal & Transverse
29	SPECTROMETER.Polarization_probed-analyzed	Probed-Analyzed	None	Probed-Analyzed	Vertical-Horizontal
30	SPECTROMETER.Scan_Amplitude_(GHz)		GHz	Only for scanning spectrometers	20
31	SPECTROMETER.Scanning_Strategy		None		Raster scan
32	SPECTROMETER.Scattering_Angle_(deg)		degrees	The angle between the illumination and the detection	180
33	SPECTROMETER.Spectral_Resolution_(MHz)		MHz	Largest easured value	150
34	SPECTROMETER.VIPA_FSR_(GHz)		GHz	The Free Spectral Range of the VIPA used in 1-VIPA spect	30
35	SPECTROMETER.x-Mechanical_Resolution_(um)		um	Estimated (phonon lifetime)	0.5
36	SPECTROMETER.x-Optical_Resolution_(um)		um	Measured	5
37	SPECTROMETER.y-Mechanical_Resolution_(um)		um	Estimated (phonon lifetime)	0.5
38	SPECTROMETER.y-Optical_Resolution_(um)		um	Measured	5
39	SPECTROMETER.z-Mechanical_Resolution_(um)		um	Estimated (phonon lifetime)	0.5

Figure 4: Example of the filled datasheet used to add metadata to the BrimX file

You can of course play with this file, add your own parameters, change the values for the ones given, add values for the ones not given, etc. We recommend however not changing the names of the existing parameters so that all the community calls the same parameters the same way.

You can then save the file at your prefered location, and apply it to the BrimX file you just created. To do so, you can use the following code:

```
wrp.import_properties_data(filepath='path/to/attributes.xlsx')
```

This line of code applies the metadata to the whole BrimX file. You can also add metadata only to a specific group by adding the argument "path" to the function:

```
wrp.import_properties_data(filepath='path/to/attributes.xlsx', path = "Brillouin
    /A z-stack")
```

3.3 Template for integrating BrimX files to your workflow

You can now try integrating the use of BrimX files in your workflow. Note that the example we showed above is limited to 4 kinds of datasets (frequency, PSD, shift and linewidth). The format can handle a total of 13 different kinds of datasets that are defined in the documentation of the package: https://github.com/bio-brillouin/HDF5_BLS/blob/main/guides/Tutorial/Tutorial.pdf.

Here is a base template to get you started:

```python
import HDF5_BLS as bls

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

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

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

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

3.4 Extracting data from the BrimX file

You can now interact directly with the BrimX file as any other file storing data. To extract a dataset located at a known path in the file, you can use the following line of code:

```python
dataset = wrp["path/to/dataset/in/the/file"][:]
```

For example, using the example BrimX file we created in this tutorial, you can extract the PSD of the first spectrum using the following code:

```python
psd = wrp["Brillouin/A single spectrum/PSD"][:]
```

To extract the metadata that is applied to an element of the file, you can use the following code:

```python
metadata = wrp.get_attributes("Brillouin/A single spectrum/PSD")
```

This will list all the metadata applied to the dataset located at the given path (in that case the PSD of the single spectrum).

The paths can be obtained by visualizing the file as explained in the next section.

3.5 Visualizing the BrimX file

3.5.1 Online

You can visualize the BrimX file online using the myhdf5 web application: https://myhdf5.hdfgroup.org. This application runs locally in your web browser so you can use it safely even to observe sensitive data.

When you open the application, you can upload your BrimX file and explore its contents either by clicking the "Select HDF5 File" button or by dragging and dropping your file in the window (figure 5). Let's try with the example file we created in this tutorial.

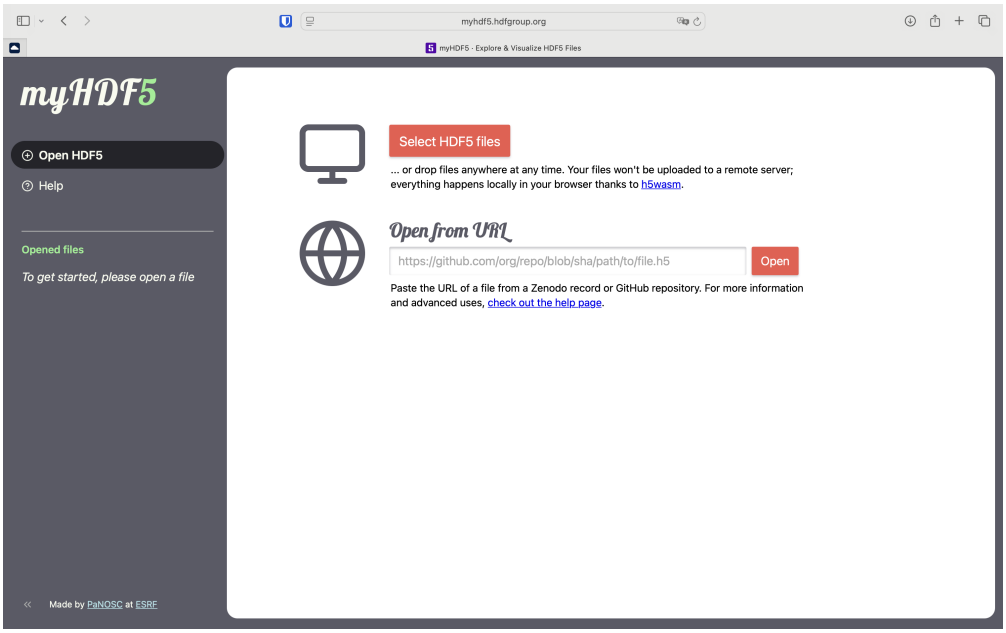


Figure 5: The welcome window of the myhdf5 application

Once your file is loaded, you can explore its contents on the left panel (figure 6).

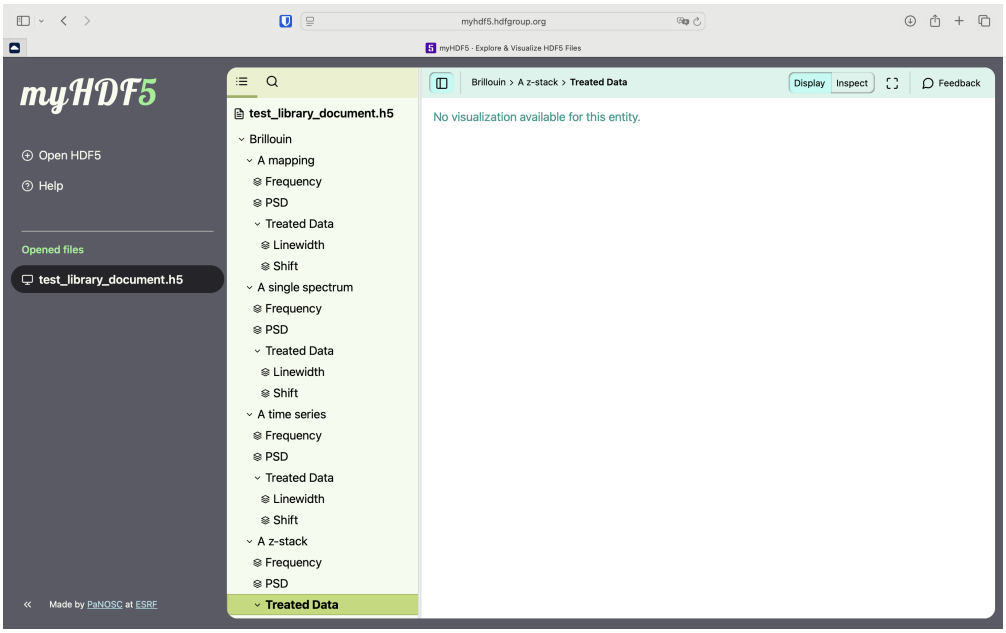


Figure 6: The developed structure view of your HDF5 file in the myhdf5 application

From there you can explore the file, visualize attributes (figure 7) and datasets of any dimension (figure 8).

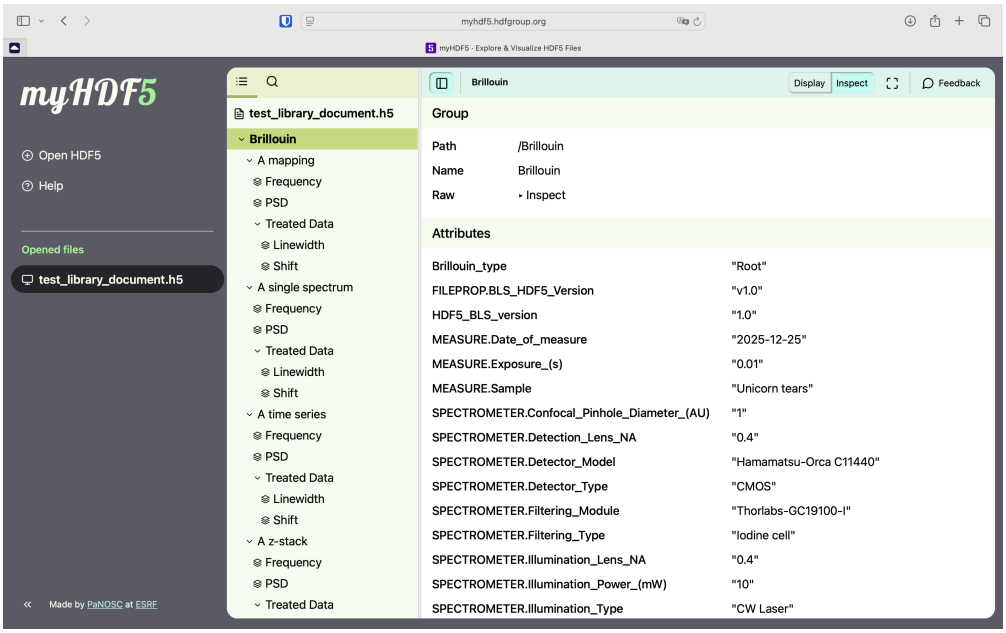


Figure 7: Visualizing the attributes stored in the "Brillouin" group

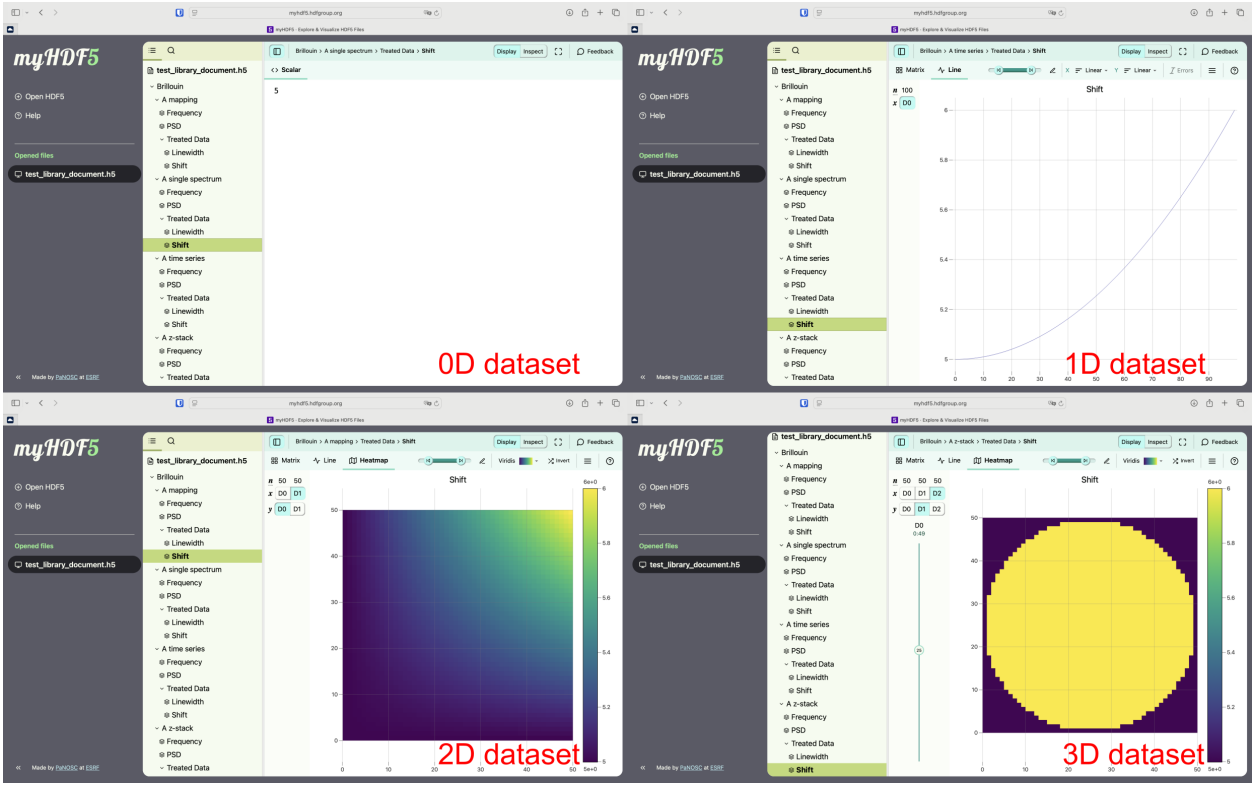


Figure 8: Visualizing the datasets using the MyHDF5 web application

3.5.2 Using Panoply

Panoply is an application developed by NASA to visualize geo-referenced data and more generally HDF5 files (thank you NASA!). It can be downloaded for free at this link: https://www.giss.nasa.gov/tools/panoply/.

Here are some screenshots of the application with the example file we created in this tutorial:

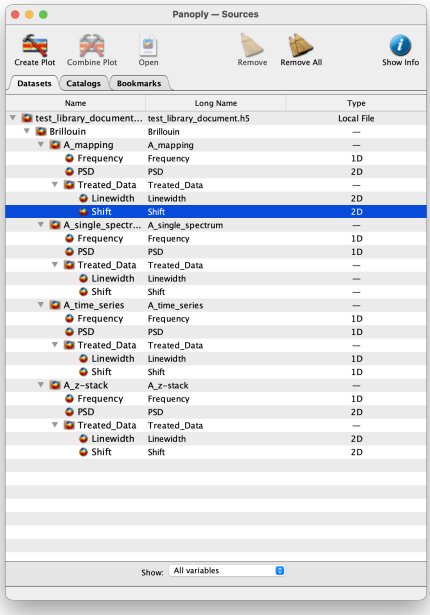


Figure 9: A simple visualization of the HDF5 file structure using the Panoply application

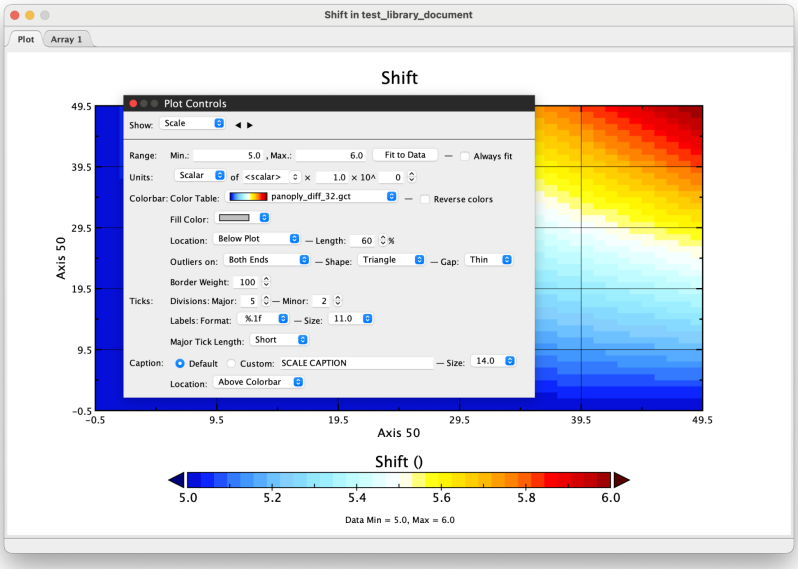


Figure 10: Visualizing datasets using the Panoply application

3.6 The HDF5_BLS_GUI

The HDF5_BLS_GUI is a graphical user interface for the HDF5_BLS package, designed to facilitate the exploration and analysis of HDF5 files. It provides a user-friendly environment for users to create and interact with their data. The GUI is meant to evolve to allow users to treat their data from it, but these developments are still in a beta phase.

To use the GUI, you need to install the package. Follow the instructions below:

- Clone the repository: `git clone https://github.com/yourusername/HDF5_BLS.git`
- Navigate to the directory: `cd HDF5_BLS`
- Install the packages for the GUI: `pip install -r requirements_GUI.txt`
- Run the GUI: `python HDF5_BLS_GUI/main.py`

You should see the HDF5_BLS_GUI window appear on your screen (figure 11).

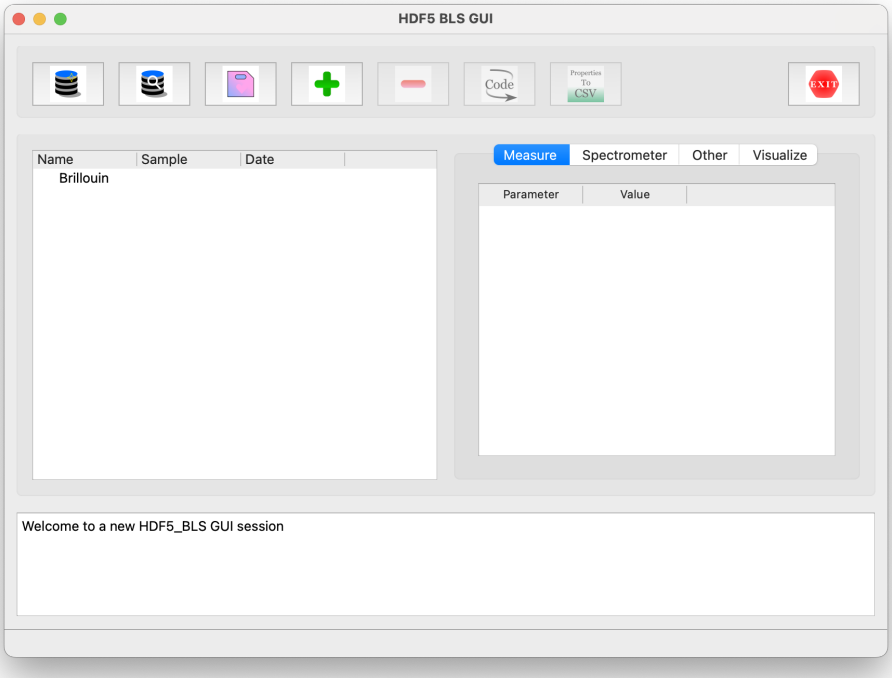


Figure 11: The welcome window of the HDF5_BLS_GUI

You can then drag and drop your data into the left panel if its addition is already supported by the GUI. If not, you'll be forced to add it from script.

You can then structure your file by creating groups, renaming them, dragging files from groups to groups, and generally use the GUI as you would a normal file explorer.

When adding a data, the GUI automatically creates a new group in the HDF5 file. You can change the name of the group by double clicking on it. You can also drag excel files with properties to the right panel to apply metadata to groups.

You can of course drag and drop the BrimX file we have created in this tutorial to the left panel to see its structure (figure 12).

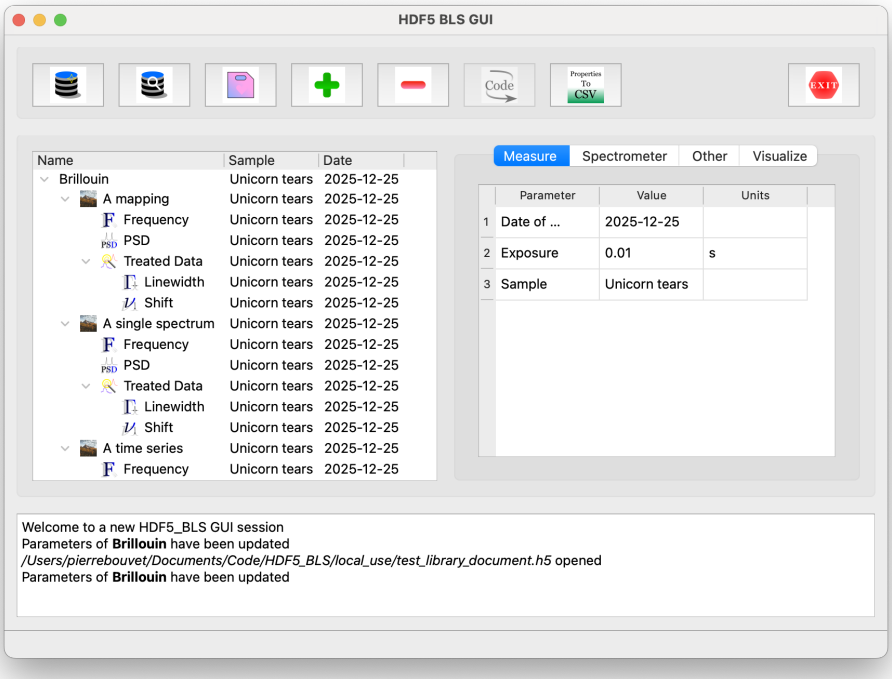


Figure 12: The structure of the example BrimX file in the HDF5_BLS_GUI

This GUI is still in development, so don't hesitate to report bugs or suggest features on the GitHub repository: https://github.com/bio-brillouin/HDF5_BLS/issues.

3.7 Ending remarks

In this tutorial, we have covered the basics of using the HDF5_BLS_GUI for exploring and managing HDF5 files. We encourage you to experiment with the GUI and provide feedback to help us improve it. You can directly contact me at pierre.bouvet@meduniwien.ac.at !

4 HDF5_BLS_treat package

The HDF5_BLS_treat package is a Python library designed to standardize the treatment and analysis of Brillouin data stored in HDF5 format. It provides a set of tools and functions to process Brillouin data and comes with a built-in recording tool to store the steps of the analysis for easy replication and storage.

4.1 Installation

We recommend setting up a virtual environment to install the package. This can be done using the following command:

- On Mac terminal

```
python -m venv HDF5_BLS_venv
source HDF5_BLS_venv/bin/activate
```

- On Windows terminal

```
python -m venv HDF5_BLS_venv
HDF5_BLS_venv\Scripts\activate
```

On both OS, the package can be installed using pip:

```
pip install HDF5_BLS_treat
```

4.2 An example

4.2.1 Creating a synthetic spectrum

Let's first create a synthetic spectrum, with a shift of 5GHz and a linewidth of 1GHz, an amplitude of 10 and an offset of 1:

```
nu = np.linspace(-15, 15, 1024)
PSD = DHO(nu, 5, 1, 10, 1)
```

4.2.2 Initialization

The treatment will be performed on these two arrays. Let's thus initialize our treatment object:

```
from HDF5_BLS_treat.treat import Treat
treat = Treat(frequency=nu, PSD=PSD)
```

4.2.3 Normalization of the spectra

A good practice is to normalize the data before applying the treatment. This can be done using the `normalize` function, after having defined one of multiple peaks to be considered as the norm (ie: the average of their amplitude is set to 1). To do so, we start by identifying the peaks to normalize. Here let's take only the Stokes peak. We select the peak by providing an approximation of its position (attribute "position_center_window") and a window size where to search for it (attribute "window_width"). We can also indicate the nature of this peak (attribute "type_pnt" that can be set to "Stokes", "Anti-Stokes" or "Other"), although in this case, this doesn't matter much since we are averaging amplitudes regardless of the nature of the peak.

```
treat.add_point(position_center_window = 5, type_pnt = "Other", window_width =
    3)
```

Now we can normalize the data. This function identifies the noise level in the spectrum, averages it, substracts it from the data, and then divides the data by the average of the amplitude of all peaks that have been given with the "add_point" function. To identify the noise level, we use the "threshold_noise" attribute. This attribute defines the highest possible value of the noise level, given as a ratio of the difference of the maximum and minimum of the data. By default, this value is set to 1%. Let's set it here to 5%.

```
treat.normalize(threshold_noise = 0.05)
```

4.2.4 Identification of the points to fit

Here we can start the fitting procedure. We first need to identify the peaks we want to fit. As before, we use the "add_point" function to do so. In this case, we will fit the Stokes peak and the Anti-Stokes peak. Here it is important to indicate the nature of the peaks (attribute "type_pnt" that can be set to "Stokes", "Anti-Stokes" or "Other") as we'll need to know that parameter to obtain a unique value of shift and linewidth to describe the spectrum.

```
treat.add_point(position_center_window = -5, type_pnt = "Anti-Stokes",
    window_width = 3)
treat.add_point(position_center_window = 5, type_pnt = "Stokes", window_width =
    3)
```

4.2.5 Definition of the lineshape to use for the fit

Different models are given to fit the data. You can for example decide to fit your peaks with a Lorentzian or a DHO lineshape. To tell the treatment which lineshape to use, you can use the "define_model" function with the name of the model given in the "name" argument. You can also indicate if a line should also be fitted. This is a simple yet effective way to reduce the effect of a possible badly attenuated elastic peak. This is done by setting the "elastic_correction" argument to True. Here let's use a DHO lineshape with no elastic correction:

```
treat.define_model(name = "DHO", elastic_correction = False)
```

4.2.6 Initial parameters

A crucial step of any fitting procedure is the definition of the initial parameters. We have normalized the spectra so we have an initial guess for the amplitude of 1, and for the offset of 0. We have also defined the position of our peaks, so this value is also initialized. We haven't however defined the initial guess for the linewidth. The easiest way to estimate the width is to use the "estimate_width_inelastic_peaks" function. Given the amplitude of the peak, this function takes the points of the curve that are approximately at the half height and deduces a value for the linewidth. To prevent aberrant values, we can provide a maximum value for the width (attribute "max_width_guess"). Here let's set it to 2GHz:

```
treat.estimate_width_inelastic_peaks(max_width_guess = 2)
```

4.2.7 Fitting the data

Everything is setup now to fit the spectra. You can perform this fit one of two ways: either you fit each peak individually or fit all the peaks at once. To choose between the two methods, you can use the "single_fit_all_inelastic" (fits every peak invidually) or the "multi_fit_all_inelastic" (fits all the peaks at once) function. Both function have the same parameters:

- "guess_offset": a boolean that indicates whether the offset should be guessed or not. It might be interesting to force the offset to 0 if you know there's no offset in your data after normalization.
- "update_point_position": a boolean that indicates whether the position of the points should be updated during the fit. This is interesting if you're looping the fitting over a large number of spectra, as the position of the peak might change from one spectrum to another due to linear effects (temperature for example).
- "bound_shift": a tuple that defines the lower and upper bounds for the shift parameter.
- "bound_linewidth": a tuple that defines the lower and upper bounds for the linewidth parameter.

To define the bounds for the shift and linewidth parameters, you can simply give a list (or tuple) of the lower and upper bounds for each fitted peak. For example, in this example, it would be weird to get values of shift lower than -5.5GHz and higher than -4.5GHz for the anti-Stokes peak, lower than 4.5GHz and higher than 5.5GHz for the Stokes peak, and linewidths lower than 0.9Ghz and higher than 1.1GHz. We can thus define the bounds and indicate the fitting as follows:

```
bound_shift = [(-5.5, -4.5), (4.5, 5.5)]
bound_linewidth = [(0.9, 1.1), (0.9, 1.1)]
treat.multi_fit_all_inelastic(guess_offset=True, update_point_position=True,
    bound_shift= bound_shift, bound_linewidth=bound_linewidth)
```

Note that at this step, an error on the value of the shift and linewidth is also obtained by estimating the Hessian matrix from the extrapolated Jacobian matrix.

4.2.8 Applying the treatment to a multidimensional PSD dataset

Up to here, we were working with a single spectrum. However, you'll often have a PSD dataset that stores multiple spectra. To apply the treatment to a multidimensional PSD dataset, you can use the "apply_algorithm_on_all" function. This function takes no arguments, it simply runs all the steps sequentially on each spectrum of the dataset, considering that the last dimension is the one of the channels.

```
treat.apply_algorithm_on_all()
```

4.2.9 Combining fits

In most cases, you'll want to combine the results of fits performed on a series of peaks. Under the hypothesis of symmetry, you can do that using the "combine_results_FSR" function. This function considers the type of the peaks, and given a FSR (Free Spectral Range), for the spectrometer, it combines the results of the fits of the peaks. If order N is centered on the 0 of the frequency axis, then it will automatically subtract to the shift of the peaks of order N+1 and N-1 the corresponding FSR offset to get a shift value contained in [-FSR/2, FSR/2]. The function then extracts a single value for the shift and linewidth of all the peaks. It can do so by either considering only the peak of highest amplitude (attribute "keep_max_amplitude" set to True), by averaging the values of all the peaks weighing them by their amplitude (attribute "amplitude_weight" set to True), by averaging the values of all the peaks weighing them by the error on the shift (attribute "shift_err_weight" set to True), or by simply averaging the values of all the peaks (all attributes set to False). Here let's consider a spectrometer with a FSR of 15GHz, and let's combine the results by weighing them by their error on the shift:

```
treat.combine_results_FSR(FSR = 15, keep_max_amplitude = False, amplitude_weight
    = False, shift_err_weight= True)
```

4.2.10 Getting the results

Your fit is over! You can now get the results of the treatment. These are stored in the attributes of the "treat" object. The main attributes are:

- "shift" -> where the shift is stored
- "linewidth" -> where the linewidth is stored
- "amplitude" -> where the amplitude is stored
- "offset" -> where the offset is stored
- "shift_var" -> where the variance on the fitted shift is stored
- "linewidth_var" -> where the variance on the fitted linewidth is stored
- "amplitude_var" -> where the variance on the fitted amplitude is stored

```
shift = treat.shift
linewidth = treat.linewidth
amplitude = treat.amplitude
```

4.3 A code to try

If you don't want to read the last section, you can directly try the following code:

```

from HDF5_BLS_treat.treat import Treat, Models
import sys

nu=np.linspace(-15, 15, 1024)
gradient_x=np.linspace(0, 1, 50)
gradient_y=np.linspace(0, 1, 50)
temp=(np.outer(gradient_y, gradient_x) + np.outer(gradient_y, gradient_x))/2
shift_2D=temp + 5
linewidth_2D=temp + 1
PSD_2D=np.zeros((50, 50, 1024))
for i in range(50):
    for j in range(50):
        PSD_2D[i, j]=DHO(nu, shift_2D[i, j], linewidth_2D[i, j], 1, 0)

treat=Treat(frequency=nu, PSD=PSD_2D)

treat.add_point(position_center_window=5.5, type_pnt="Stokes", window_width=1)
treat.normalize_data(threshold_noise=0.05)

treat.add_point(position_center_window=-5, type_pnt="Anti-Stokes", window_width
    =3)
treat.add_point(position_center_window=5, type_pnt="Stokes", window_width=3)

treat.define_model(model="DHO", elastic_correction=False)

treat.estimate_width_inelastic_peaks(max_width_guess=1)

bound_shift=[(-6.5, -4.5), (4.5, 6.5)]
bound_linewidth=[(0.9, 2.1), (0.9, 2.1)]
treat.multi_fit_all_inelastic(guess_offset=True, update_point_position=True,
    bound_shift= bound_shift, bound_linewidth=bound_linewidth)

treat.apply_algorithm_on_all()
treat.combine_results_FSR(FSR=15, keep_max_amplitude=False, amplitude_weight=
    False, shift_err_weight= True)

plt.figure()
plt.subplot(131)
plt.imshow(treat.shift)
plt.title('Brillouin Shift (GHz)')
plt.colorbar()

plt.subplot(132)
plt.imshow(treat.linewidth)
plt.title('Line Width (GHz)')
plt.colorbar()

plt.subplot(133)
plt.imshow(treat.amplitude)
plt.title('Amplitude (a.u.)')
plt.colorbar()

plt.tight_layout()
plt.show()
```

Thank you for your help! Happy Brillouin data handling!