SFGTools

Release 1.0

jdpicks

Oct 01, 2021

CONTENTS

CHAPTER

ONE

SFGTOOLS : PROCESSING AND ANALYSING SFG SPECTROSCOPY DATA

1.1 Overview

SFGTools is a Python module designed to facilitate easy processing and analysis of Sum Frequency Generation (SFG) spectroscopy datasets.

In its current form, it consists of a module containing functions and methods for working with SFG data in a Python script/interpreter, `sfgtools.py`. Using SFGTools in this way (importing into Python and using in a script) is the most flexible way to use it, and provides the most extended functionality for complex use cases. However, packaged with this module is a graphical frontend made using Qt Designer, `SFGTools_ui.ui`, which is converted to a Python file, `SFGTools_ui.py`. This frontend is interfaced with `sfgtools.py` via the `SFGToolsGUI.py` file, and provides a user-friendly GUI where SFG data can be quickly and easily processed and plotted (as would be useful in a working lab), but without all of the extended functionality of the bare module.

1.1.1 Scope

Currently, `sfgtools` can be used to process SFG spectra recorded in the *.spe* filetype, which is common in high-sensitivity CCD cameras for spectroscopy. Support is provided for *.spe* version 2.x and 3.0 (commonly produced by Andor and Princeton Instruments cameras respectively). The graphical frontend and plotting capability is currently designed around plotting one-dimensional SFG spectra (where the data on the CCD is binned into one row), but the underlying software supports a wider range of data shapes and formats - they just may not plot correctly when using the GUI. Use of the module in a Python script is advised for this purpose.

As far as possible, the module has been written to be easily extended, and is well documented. Thus, other data types, data formats, and processing functions can be easily integrated. See REF LATER.

1.2 Using SFGTools in a Python Script

For ultimate flexibility for future development, the *sfgtools.py* module defines a class `SFGProcessTools()` which contains all relevant classes, methods, and attributes for processing SFG data. To load the module in a script, do something like:

```
import sfgtools as sfgtools
SFGTools = sfgtools.SFGProcessTools()
```

Here we have imported the module and created `SFGTools` as an instance of the `SFGProcessTools()` class.

Now we will load and process some example data to illustrate the module. The example data files and complete script can be found in REF.

First, we define our files to be processed. A publishable SFG spectrum normally consists of a **signal** spectrum that is background subtracted using a **background** spectrum. This subtracted signal spectrum is then normalised by dividing it by a **reference** spectrum, which is itself backgorund subtracted from a **reference background** spectrum.

These four names are used extensively in various full and shortened forms throughout the module to denote these different kinds of data. We can load the spectra as follows:

```
directory = './examples/'
signal = ['signal_example.spe']
background = ['background_example.spe']
reference = ['reference_example.spe']
reference_bg = ['reference_background_example.spe']
```

Note that we have defined a data directory, `directory`, and then four appropriately named Python **lists** containing the four raw *.spe* files to be processed. The files could be used as bare strings, but loading them as a list makes it easier to batch process files later on.

Before we can process these four files, we have to provide some more information to the program about how we want to process our data. In addition to background subtraction and normalisation, SFG spectra are almost always **downconverted**, which means that the spectrum energy axis is lowered in energy such that reflects the vibrational (or electronic, or whatever) response of the molecule. The energy is lowered by the energy of the upconversion beam used in the experiment. Furthermore, the spectra are often **calibrated** by applying a linear offset to the energy axis, to account for any poor calibration of the spectrometer. We have to to tell the program the wavelength of the upconverter, and any calibration shift, and we do this as follows:

```
SFGTools.upconversion_line_num = 808
SFGTools.calibration_offset = 15
```

Here we have defined the class attributes `upconversion_line_num` and `calibration_offset`, in nanometres and wavenumbers respectively. See specific documentation for more information. Finally, we have to tell the program that we do want to downconvert, calibrate, background subtract, and normalise our spectrum. We also want to correct the spectra for exposure time differences (so subtracting them from each other makes sense), and we might want to remove contributions from cosmic rays (but not needed in this example):

```
SFGTools.downconvert_check = True
SFGTools.calibrate_check = True
SFGTools.subtract_check = True
SFGTools.normalise_check = True
SFGTools.exposure_check = True
SFGTools.cosmic_kill_check = False
```

Setting these boolean flags will tell the program what to do (there are many other flags, see further documentation). Now we are ready to read our data files, but need to create a place to put the data. To do this we create an `SFGDataStore` instance:

```
datastore = SFGTools.SFGDataStore()
```

The instance `datastore` of the `SFGDataStore()` class will hold our data, together with a large amount of associated metadata if we desire it. Understanding how this class works is central to effectively using (and developing for) this module. The data and metadata are all stored in class attributes, and core processing functions that are applied to the data (such as all of the processes mentioned above) are implemented as methods of this class. You can also create a list of datastore instances for an arbitary number of files by using the function:

```
datastores = SFGTools.create_data_stores(len(signal))
```

Where the length of your *signal* list defines the number of datastores. Now we can load our data into the datastores as follows:

```
SFGTools.read_files( directory+signal[0], datastore, 'sig')
SFGTools.read_files( directory+background[0], datastore, 'bg')
SFGTools.read_files( directory+reference[0], datastore, 'ref')
SFGTools.read_files( directory+reference_bg[0], datastore, 'refbg')
```

The function `read_files()` takes the path to the datafile as the first argument, then the `SFGDataStore` instance to store the data in, and finally a string that determines whether the loaded file is marked as a signal (*'sig'*), background (*'bg'*), reference (*'ref'*), or reference background (*'refbg'*) file. This ensures that the data is put in the right place for further processing. If you have a list of datastores, or simply want to avoid writing this out, you can also call the function:

```
SFGTools.populate_data_stores(datastores, directory, signal, background, reference,␣
↪reference_bg)
```

See further documentation for more information. Note that some parameters such as the exposure time and spectrum dimensions are automatically extracted from the *.spe* file, but others can either be added manually, or by using other functions provided in the module. If we wanted to look at our imported data, we can simply look at some of the attributes of datastore - perhaps we want to look at the raw signal data, then type into the Python shell:

```
datastore.signal_raw
```

And you should see a **numpy array** object outputted. All the data are stored as numpy arrays, with a shape of *(frameheight, framewidth)*, where frameheight and framewidth refer to the height and width of the data read from the *.spe* file. In general, this array will then have a shape of *(1, n)* where *n* is the width of the spectrum. The additional dimension is preserved such that reading without binning the CCD chip is possible. Similarly, we can look at the energy axis:

```
datastore.xaxis
```

This is also a numpy array, but is now one dimensional, as the energy axis will never have two dimensions. All attributes of the datastore can be printed to the shell using the `print_attributes()` method. Anyway, let us now proceed with processing our data, which we can do as follows:

```
SFGTools.process_data(datastore, SFGTools.downconvert_check, SFGTools.subtract_check,
                      SFGTools.normalise_check, SFGTools.exposure_check,
                      SFGTools.calibrate_check, SFGTools.cosmic_kill_check )
```

The `process_data` method takes the populated datastore and all our flags as arguments. Under the bonnet, this is calling the appropriate methods of the `SFGDataStore` class (see further documentation). If you have multiple datastores in a list, you can batch process as follows:

```
SFGTools.batch_process(datastores)
```

This doesn't take all the flag arguments and inherits them from the class. The reason for this is that sometimes it is desirable to process the same data with or without certain kinds of processing, and the `process_data()` method makes this easier. The `batch_process()` method is mainly intended for use with the GUI frontend for this module.

Now we have the data processed, we can use our preferred plotting program to look at it, I like matplotlib, so something like:

```
import matplotlib.pyplot as plt
plt.figure()
plt.plot(datastore.xaxis, datastore.signal_normalised[0])
```

Will show us the data. You can also apply whatever processing you want to do to the data (fitting etc..) to the processed data. Alternatively, if we have already imported matplotlib, we can use plot our data using functions in the module. These are again mostly intended for use with the GUI, so have some additional arguments which need not worry us here:

```
SFGTools.custom_region_start = 2800
SFGTools.custom_region_end = 3100
SFGTools.plot_data(datastore, iteration=1, num_files=1, figure=plt.figure() )
```

Here we have defined two new class attributes which plot the interesting range of our data. The *iteration* and *num_files* variables are used to control plotting in the GUI.

There you go! You have processed and plotted some SFG data using the `sfgtools` module! Please explore the rest of the documentation to see what other methods are available (there are many), and do not be shy about hacking apart the code and bending it to your will.

CHAPTER

TWO

USING SFGTOOLS WITH A GUI

Also provided with the **SFGTools** module is a graphical user interface to make doing the most common data processing simple and efficient. The GUI can be run (if you have Python and the right packages installed) from the a UNIX shell or Windows Powershell, after navigating to the directory containing *SFGToolsGUI.py* simply by running the file with your Python interpreter:

```
python SFGToolsGUI.py
```

You should now see a GUI window open on your screen. If you don't, it is probably due to you not having Python or the relevant dependencies installed - a standalone *.exe* version of the GUI is coming very soon. The SFGTools GUI works simply by implementing the functions defined in the *sfgtools.py* module but takes user input of the files and processing flags from the GUI window. Lets take a tour of the GUI panel.

Examining panels clockwise from the top left, we have: * **Data Input** - this panel contains two tabs that give two options for data input. A 'smart' reader (efficient if your filenaming is consistent and as the program expects - see later), and a 'manual' reader (a bit more labour intensive). * **Data Processing** - this panel is where options related to the data processing can be selected, currently it holds checkboxes that relate to all the bool flags that determine how the data is processed. * **File Manager** - this panel contains two tables that will show the loaded data files (top table is for signal data, bottom is for reference data). * **Program Options** - this panel contains options that relate to the running of the GUI. * **Output** - this panel contains settings that determine how and where the program outputs processed data. * **Experimental Parameters** - this panel contains parameters that are specific to the experiment but affect the data processing.

Use of the GUI once you have loaded the data files is pretty straightforward, just click the checkboxes you want and hit "Go!" to process and plot/write the data. The slightly complex part comes in understanding the file input methods. Using the example files in */examples/GUI/* we can illustrate this.

There are two ways that data files can be loaded into the SFGToolsGUI - **smart**, and **manual**. The **smart** method works best if your data conform to a (somewhat flexible) naming format, and is helpful if you need to process large amounts of data at once. The **manual** method is slightly more labour intensive. Recall that the processing in *SFGTools* is centered around defining **signal**, **background**, **reference**, and **reference background** data files, which are then loaded into a datastore and processed.

2.1 Using the Smart File Reader

The first thing to do when using either file reader is to select the **data directory** that you want to read the files from. As of the current version, all the files to be read must be in the same folder (otherwise use the module in a Python script to get more flexibility). This is done either using the *select directory* button on the Data Input panel, or the *Browse...* button on the File Manager panel. Both point to the same slot in the underlying program, and will open a file dialog that allows you navigate and select a folder. Select the */examples/GUI/* folder now.

If it was not already so, the *Data Directory* field in File Manager should now be filled in with the correct directory. Now, we have to provide three strings to the smart data reader so that it can get our data: * **Sample String** - this string must be at the start of your filename, and is what identifies it as the correct type of data file to plot (for example - you may have recorded several different samples on one day, but only want to plot one at a time). * **Reference String** - this string must be at the start of the filename of any reference files, and identifies them as a

reference. * **Bg String** - this string is what identifies a file as a background file, and can be present anywhere in the filename.

Some of these fields may already be filled in depending on the settings when the GUI was last run on your system. Set the sample string to 'sample', the reference to 'reference', and the background to '_bg'. When the fields are filled in and the *Get Data* button is pressed, then the program will look in the specified directory and find the relevant files. Try this now. You should see files that have been loaded in the File Manager panel - four in total.

It is perhaps useful to illustrate how this method would work with a more complex filename. Imagine that we have the following six files:

```
LkA_SSP_3000nm_1.spe
LkA_SSP_3000nm_2.spe
LkA_SSP_3000nm_3.spe
LkA_SSP_3000nm_bg.spe
Au_PPP_1.spe
Au_PPP_bg.spe
```

In our fictional experiment, our sample is 'LkA', and our reference is 'Au' (classic SFG). The smart file reader would identify the top three files as signal files, the fourth file as a background to the signal files, the fifth file as a reference file, and the sixth file as a background to the reference file. The reader is quite capable, but please try and break it so it can be improved!

Now our files are loaded, but we need to **sort** the files before they can be processed. For the example case, this is not strictly necessary, but if we have multiple signal files that share a background or reference, or multiple references, then this is important. Clicking the *Auto Sort* button in File Manager will attempt to match each signal file with it's correct background and reference files (see module documentation for more explanation). Of course, if it doesn't do this correctly (or you have some inconsistent naming), you can manually edit the generated data tables and it will update the data to be processed.

Having got and sorted the data, you can now ensure that the *Plot Data* checkbox is ticked and plot the data.

2.2 Using the Manual File Reader

The manual file reader is simpler to understand but more clunky to use. Changing the tab from *smart* to *manual* you will see that there are now four buttons labelled with the four filetypes we need for our data processing. Again, the first thing to do is to ensure the correct directory is selected.

Once that is done, clicking each of the four buttons will bring up a file dialog where data files can be selected. The files you select in the dialog that arises from the *Signal Files* button will be stored as signal files, and so on. Once you have done this for all four buttons you will see that the tables are populated - then you can either sort out the backgrounds/references manually, or you can use the Auto Sort function again.

Try to use the Manual file reader now with the files in the */examples/GUI/* folder. You should find it gives the same result as the smart reader.

You will note that the reference and signal tables are separate, and that references are created with a number (the **refID**) next to them. Most of the time in SFG, there are many more signal files than reference files, so you may only have two or three reference files loaded that provide the references for tens of different signal files. You can see which reference files are associated with each reference after sorting by comparing the numbers in the final column of each table - it is fairly self explanatory.

Sensible skepticism. Before processing, but after the files are loaded and sorted, you can press the *Check Files* button, and this will print a table of files to the shell. The four columns of this table are:

```
[signal_files, background_files, reference_files, reference_bg_files]
```

Each row represents a different signal file, and the files in the same row are all stored in the same `SFGDataStore` instance, and thus are processed together - i.e. the background file in the same row as the signal file is the background for that signal, and so on. In this way you can check what processing is actually going to happen before

you hit *Go* and potentially generate hundreds of meaningless spectra. Again, note that if the sorting function does not work for you, you can manually edit the table. But let me know if you find bugs.

2.3 Processing Options

Once the files are loaded, I think it is quite self explanatory how the processing checkboxes work. Simply select or deselect the ones you want. Note that if you have a spectrum you want to plot without a background/reference, then just deselect these and the program will not get angry about there not being a loaded background/reference. If it expects there to be a file and one is not provided, you only have yourself to blame for the crash to desktop.

You also need to input the upconversion wavelength in the appropriate box - in **nanometres**. The calibration offset must be provided in **wavenumbers**.

The cosmic ray remover is a function implemented by Steven and I, but has not been thoroughly tested and is not the most robust thing. Try it if needed and see - but future versions will have more functionality in this area. In the same vein, a polynomial calibration will be coming soon.

2.4 Data Output

In terms of data output, the options are to plot the data using matplotlib, write it to a *.txt* file, or both. If the data are plotted, there are some limited options: * **Stack Plots** - will overlay the data from all loaded signal files on a single figure. * **Close Plots** - will close any open plots the next time *Go!* is pressed. * **Region of Interest** - these two textboxes define the start and end of the region to be plotted, in **wavenumbers**. Often you want to ignore all the noise at the edges due to normalisation, and this does that.

If the data are written to a *.txt* file, they are written to the directory specified in the *Write Directory* box, which can be selected using the *Browse* button next to it. The data are written to a file with the same name as the signal file, but with an appended string showing they have been processed. The files written are relatively chunky, as there is a header that records a large amount of the processing options that were used (for future reference), and also many parts of the raw data are written out, not just the final processed output - this is for possible future reference. The first two columns of this file contain the final processed spectral data (xaxis, yaxis).

2.5 Miscellaneous Notes

- The current version of this GUI is only going to give reliable results when processing **one dimensional** spectral data. Use the module in a Python script for more complex cases.
- There are some program options to do with verbosity that can be selected - this just changes what is printed to terminal during processing.
- After each run, the program will store the last used parameters internally and save them on closing, so that when the program is reinitialised on your machine, your previous settings will be reloaded. The *Restore Defaults* button clears this memory, so an empty GUI will be loaded on the next startup.

2.5.1 Dependencies

The *sfgtools* module requires Python to use. It was written using Python 3.8.8. It requires the following modules to run:

- **NumPy** - for all the array handling.
- **PyQt5** - for the GUI.
- **Matplotlib** - for plotting data.
- **pathlib** - for file and path handling.

- **lxml** - for handling XML in SPE3.0 files.
- **glob** - for smart file reading.
- **inspect, sys, warnings** - boring Python stuff.

Aside from using the standalone *.exe*, another way to use the program is to install Python using **Anaconda** - this *should* handle dependency problems and install the needed packages for you, and you can then run the GUI from a python shell.

SFGTools

SFGToolsGUI module

class SFGToolsGUI.**ItemDelegate**

Bases: PyQt5.QtWidgets.QStyledItemDelegate

createEditor(*self*, *QWidget*, *QStyleOptionViewItem*, *QModelIndex*) → QWidget

editingfinished

editingstarted

class SFGToolsGUI.**MainWindowUIClass**

Bases: PyQt5.QtWidgets.QMainWindow, *SFGTools_ui.Ui_MainWindow*

auto_sortSlot()

auto_sort_checkSlot()

bg_string_boxSlot()

browse_background_filesSlot()

browse_directorySlot()

browse_reference_background_filesSlot()

browse_reference_filesSlot()

browse_signal_filesSlot()

browse_write_directorySlot()

calibrate_checkboxSlot()

calibrate_offsetSlot()

clear_persistent_settingsSlot()

close_plotsSlot()

cosmic_kill_checkboxSlot()

cosmic_thresholdSlot()

cosmic_widthSlot()

custom_region_end_textboxSlot()

custom_region_start_textboxSlot()

data_directorySlot()

detonateSlot()

downconvert_checkboxSlot()

editSlot()

exposure_checkboxSlot()

get_dataSlot()

normalise_checkboxSlot()

plot_data_checkboxSlot()

quit_Slot()

ref_stringSlot()

sample_stringSlot()

setupUi(*mainWindow*)

stack_plots_checkboxSlot()

stupid_verboseSlot()

subtract_checkboxSlot()

testSlot()

upconversion_line_dropdownSlot()

update_gui_tables()

verboseSlot()

write_directory_boxSlot()

write_file_checkboxSlot()

class SFGToolsGUI.**TableModel**(*data*, *header*)

Bases: PyQt5.QtCore.QAbstractTableModel

add_rows(*new_rows*)

columnCount(*self*, *parent: QModelIndex = QModelIndex()*) → int

data(*self*, *QModelIndex*, *role: int = Qt.DisplayRole*) → Any

flags(*self*, *QModelIndex*) → Qt.ItemFlags

headerData(*self*, *int*, *Qt.Orientation*, *role: int = Qt.DisplayRole*) → Any

rowCount(*self*, *parent: QModelIndex = QModelIndex()*) → int

setData(*self*, *QModelIndex*, *Any*, *role: int = Qt.EditRole*) → bool

SFGToolsGUI.**main**()

SFGTools_ui module

class SFGTools_ui.**Ui_MainWindow**

Bases: object

retranslateUi(*MainWindow*)

setupUi(*MainWindow*)

sfgtools module

Tools for processing SFG Data.

Classes: SFGProcess Tools

class sfgtools.**SFGProcessTools**

Bases: object

A class that contains SFG processing tools.

This is implemented as a class so it can be easily invoked as a model for a PyQT GUI, under the MVC framework.

verbose
If true tells program to print verbose output,

Type bool

stupid_verbose
If true tells program to print REALLY verbose output.

Type bool

sum_accumulations
If true tells program to sum multiple frames stored in the same .spe file

Type bool

series_accumulations
If true tells program to store multiple frames from the same .spe file as different arrays (series).

Type bool

downconvert_check
If true then spectra are downconverted.

Type bool

subtract_check
If true then spectra are background subtracted.

Type bool

normalise_check
If true then spectra are normalised by a reference.

Type bool

calibrate_check
If true then spectrum energy is shifted by +calibration_offset.

Type bool

exposure_check
If true then all spectra are divided by their relevant exposure time.

Type bool

cosmic_kill_check
If true then cosmic ray spikes are attempted to be removed automatically.

Type bool

stack_plots_check
If true then plots from multiple loaded files are stacked onto one figure.

Type bool

global_force
If true then attempts to (e.g.) downconvert spectra twice are allowed.

Type bool

write_file_check
If true then a .txt file with the processed data is written.

Type bool

plot_data_check
If true then processed data is plotted using matplotlib.

Type bool

close_plots_check
If true then plots are closed between successive runs.

Type bool

data_directory
Direcotry where all the files to processed are.

Type str

write_directory
Directory where output files will be written to.

Type str

samplestring
What the smart file getter looks for at the start of a filename to identify a file as relevant.

Type str

refstring
What the smart file getter looks for at the start of a filename to identify a file as a reference.

Type str

bg_string
What the smart file getter looks for to identify a file as a background file.

Type str

upconversion_line_num
Wavelength of the upconverter in nanometres.

Type float

calibration_offset
Amount to shift a spectrum energy axis in wavenumbers.

Type float

custom_region_start
Defines leftmost edge of plotted data in wavenumbers.

Type float

custom_region_end
Defines rightmost edge of plotted data in wavenumbers.

Type float

cosmic_threshold
Used by the cosmic_ray_killer method. See description for detail.

Type float

cosmic_max_width
Used by the cosmic_ray_killer method. See description for detail.

Type float

spe_version_loc
Location in bytes that the .spe file version is stored in the .spe file.

Type int

footer_offset_loc_loc
Location in bytes that the location in bytes that the offset to the XML footer is found in an SPE3.0 file.

Type int

data_offset_loc_loc
Location in bytes of the location in bytes to the first data in an SPE2.x file.

Type int

framewidth_loc
Location in bytes of the framewidth of an SPE2.x file.

Type int

frameheight_loc
Location in bytes of the frameheight of an SPE2.x file.

Type int

numframes_loc
Location in bytes of the number of frames stored in an SPE2.x file.

Type int

pixeltype_loc
Location in bytes of the pixel type stored in an SPE2.x file.

Type int

acqtime_loc
Location in bytes of the acquisition time in an SPE2.x file.

Type int

signal_names
Contains the filenames of the signal files to be processed.

Type list

bg_names
Contains the filenames of the background files to be processed.

Type list

ref_names
Contains the filenames of the reference files to be processed.

Type list

ref_bg_names
Contains the filenames of the reference background files to be processed.

Type list

ref_num
Contains the reFID number of each reference file.

Type list

sig_ref_num
Contains the refID number that each signal file needs to be normalised by.

Type list

tabledata
Contains data to be shown in the GUI main data table.

Type list

reftabledata
Contains data to be down in the GUI reference data table.

Type list

current_figure
Figure data is currently being plotted on.

Type pyplot figure

class SFGDataStore

Bases: object

This class is where the SFG data is stored.

An instance of the class is created for every distinct signal file - i.e. a file that is not a background or reference file. The slots define different things that can be associated with the signal file. Each slot has an associated attribute.

acqtime

acqtime_bg

acqtime_ref

acqtime_refbg

applied_calibration

background

background_subtract(*force=False*)

Subtract the background spectrum from the signal spectrum.

Subtracts background attribute from the signal attribute (if it already exists), or raw attribute (if it doesn't). Either way the signal attribute is created.

Parameters

force [bool, optional] Allows subtraction more than once if true. Default False.

background_subtracted

calibrate_spectrum(*calibration_offset*, *force=False*)

Shift the energy axis of a spectrum by +calibration_offset.

Passing "force" and calibrating multiple times will update the applied_calibration attribute of datastore to keep track of what has been done.

Parameters

- **calibration_offset** (*float*) – The amount to shift the energy xaxis, in wavenumbers. Positive numbers shift to higher energy.
- **force** (*bool, optional*) – Allows calibration more than once if true. Default False

calibrated

cosmic_bg

static cosmic_ray_killer(*data*, *threshold*, *max_width*)

Remove cosmic ray contributions from data.

Algorithm from Steven J Roeters. Not thoroughly tested but implemented for future use.

Questionably robust method for doing it that might just require endless tweaking.

Parameters

- **data** (*np array*) – Array containing the data to have cosmic ray contributions removed. Shape (1, width).
- **threshold** (*float*) – Min height above the background signal that a spike has to have to be considered a cosmic ray.
- **max_width** (*int*) – Anything wider that max_width is considered real signal and not a cosmic ray.

Returns

- **data** (*np array*) – Data with cosmic ray contributions removed.
- **rays_removed** (*bool*) – Flag used to keep track of whether or not the data has had cosmic ray contributions removed.

cosmic_ref

cosmic_refbg

cosmic_sig

creationtime

divide_exposure(*force=False*)
Divide all stored spectra by their relevant exposure times.

Call this on the raw data before you do further processing.
Parameters **force** (*bool, optional*) – Allows exposure division more than once if true. Default False

downconvert_spectrum(*upconverter*, *force=False*)
Downconvert the energy axis in the spectrum by upconverter in wavenumbers.

Overwrites xaxis attribute with the downconverted version. The original xaxis attribute is stored as xaxis_raw if needed later.
Parameters
- **upconverter** (*float*) – The energy of the upconversion line to subtract in wavenumbers.
- **force** (*bool, optional*) – Allows downconversion more than once if true. Default False.

downconverted

exp_divided_bg

exp_divided_ref

exp_divided_refbg

exp_divided_sig

static exposure_subroutine(*data*, *time*, *flag*, *string*, *force*)
Divide the given spectrum by its exposure time.

This subroutine is called multiple times by divide_exposure()
Parameters
- **data** (*np.ndarray*) – The data to be divided by exposure time.
- **time** (*float*) – The exposure time to divide by.
- **flag** (*bool*) – True if the spectrum has already been divided, used to prevent unwanted multiple calls.
- **string** (*str*) – Refers to the type of spectrum being subtracted. Possible values: “signal”, “background”, “reference”, “reference background”
- **force** (*bool, optional*) – Allows exposure division more than once if true. Default False

filename_bg

filename_ref

filename_refbg

filename_sig

frameheight

framewidth

group

index

normalise_data(*force=False*)
Divide a signal spectrum by the reference spectrum.

Will divide by the subtracted spectrum if it exists, otherwise from the raw spectrum.
Parameters **force** (*bool, optional*) – Allows normalisation more than once if true. Default False

normalised

numframes

polarisation

ref_background_subtract(*force=False*)
Subtracts background from the reference signal.

Subtracts from ref_raw unless ref_subtracted exists.

Parameters **force** (*bool, optional*) – Allows subtraction more than once if true. Default False

ref_bg

ref_raw

ref_subtracted

refbackground_subtracted

remove_cosmic_rays(*threshold*, *max_width*, *flag*)
Call cosmic_ray_killer on raw data.

Uses flag to decide what kind of raw data to apply the removal to.

Parameters

- **threshold** (*float*) – Min height above the background signal that a spike has to have to be considered a cosmic ray.
- **max_width** (*int*) – Anything wider that max_width is considered real signal and not a cosmic ray.
- **flag** (*str*) – Determines what attribute of SFGDataStore to remove rays from. Possible values are “sig”, “bg”, “ref”, “refbg”, or “all”.

sample

signal_normalised

signal_raw

signal_subtracted

timestamps

upconverter_used

wavelength

xaxis

xaxis_raw

static assign_acqtime_to_storage(*flag*, *datastore*, *acqtime*)
Take the assigned flag and put the acquisition time from the file in the right datastore.

Parameters

- **flag** (*str*) – Defines the storage attribute of datastore to put the exposure time into. Possible values “sig”, “bg”, “ref”, “refbg”.
- **datastore** (*SFGDataStore object*) – The datastore instance to put the acquisition time into.
- **acqtime** (*float*) – Acqusition time in seconds.

static assign_data_to_storage(*flag*, *datastore*, *data*)
Take the assigned flag and put the data read from the file in the right datastore attribute.

Parameters

- **flag** (*str*) – Defines the storage attribute of datastore to put the data into. Possible values “sig”, “bg”, “ref”, “refbg”.
- **datastore** (*SFGDataStore object*) – The datastore instance to put the data into.

- **data** (*np array*) – Numpy array of the data to store.

static assign_filename_to_storage(*flag*, *datastore*, *filename*)
Take the assigned flag and put the filename into the right datastore attribute.

Parameters

- **flag** (*str*) – Defines the storage attribute of datastore to put the filename into. Possible values "sig", "bg", "ref", "refbg".
- **datastore** (*SFGDataStore object*) – The datastore instance to put the data into.
- **filename** (*str*) – Filename to store in datastore.

batch_process(*datastores*)
Process all data in the instances contained in datastores.

Assumes you have a list of populated SFGDataStore objects to process (one per file).

Here the bool checks are all called as class attributes via self rather than passed explicitly, which makes life slightly less cumbersome when invoking it in the GUI.

Parameters **datastores** (*list*) – Contains SFGDataStore objects, one per file to be processed.

static cm_to_nm(*data*)
Convert data from wavenumber to nanometre.

static cm_to_thz(*data*)
Convert data from wavenumber to terahertz.

create_data_stores(*num_files*)
Create a list of SFGDataStore classes of length num_files.

Parameters **num_files** (*int*) – Number of files to process (and SFGDataStore instances to create).

Returns **datastores** – Contains empty SFGDataStore instances to be populated.

Return type list

static create_matched_ref_list(*sig_ref_id*, *ref_filenames*, *ref_bg_filenames*, *ref_id*)
Create lists of the reference files matched to signal files for processing.

Reads the list sig_ref_id and creates new lists of the same length as the list of signal files containing the right reference/reference background files for processing.

Parameters

- **sig_ref_id** (*list*) – Contains the refIDs that each signal file needs to be processed with.
- **ref_filenames** (*list*) – Contains the reference filenames.
- **ref_bg_filenames** (*list*) – Contains the reference background filenames.
- **ref_id** (*list*) – Contains the refIDs that each reference file corresponds to.

Returns

- **ref_matched** (*list*) – Contains the filenames of the reference files that each signal file needs to be normalised to.
- **refbg_matched** (*list*) – Contains the filenames of the reference background files that each signal file needs.
- **ref_num** (*list*) – The number of each reference file used (for display in GUI).

get_closest_file(*files*, *target*)
Compare the time of last modification of each element of files, and find the closest one to target.

Parameters

- **files** (*list*) – Contains files that are being tested for closeness-in-time to target.
- **target** (*str*) – File that is used as a target to find the closest file to.

Returns **closest_file** – The element of files that was created at the closest time to target.

Return type str

static get_file_creationtime(*file*)
Return the time of last modification of the file input.

get_filenames_smart()
Globs all .spe files in the current data directory and sorts them into lists.

Used mostly in the GUI so parameters are all class attributes modified by the GUI. This method looks in a directory for all the .spe files that are present, and then splits them up into signal, background, reference, and reference background, in line with the (user supplied) strings that determine if a file is a sample, reference, or background. For example, using attributes:

self.samplestring = sampledata self.refstring = referencedata self.bg_string = _bg

Would sort the following files as follows:

sampledata.spe -> signal sampledata_bg.spe -> background referencedata.spe -> reference reference-data_bg.spe -> reference background

It is important that the files START with the supplied sample and ref strings. The bg string can be anywhere in the filename.

Returns

- **signal_names** (*list*) – Contains names of signal files.
- **bg_names** (*list*) – Contains names of background files.
- **ref_names** (*list*) – Contains names of reference files.
- **ref_bg_names** (*list*) – Contains names of reference background files.
- **ref_id** (*list*) – Contains indexes for each unique reference file (later associated with a corresponding signal file).

get_pixel_type(*pixeltype*)
Convert the pixel type given by SPE 2.x and 3.0 files to numpy datatype.

The pixeltype read from the SPE file is either a string (spe 3.0) or an integer (spe 2.x) that needs decoding. This method converts either to a numpy datatype.

Parameters **pixeltype** (*int or str*) – The pixeltype obtained from reading the SPE file. str for SPE 3.0, int for SPE 2.x.

Returns

- **pixeltype_np** (*numpy datatype*) – Numpy datatype corresponding to the pixel datatype.
- **pixelsize** (*int*) – Size of the pixel in bytes.

static get_window(*data*, *n_base=10*, *n_dev=2*)
UNUSED. Get the indices of the reference array where spectral intensity is non-zero.

It's a bit of a clunkfest and not very reliable.

NOT IMPLEMENTED IN V1.0

match_files(*sig*, *bg*, *ref*, *refbg*, *refid*, *directory*)
Match each signal file with its corresponding reference and background files.

Calls previously defined methods to populate the lists needed to run data processing.

To labour the point, the idea is that for processing you need up to four lists of the same length containing the file names to be processed. The background/reference/reference background file that correspond to element n of the signal list are in element n of their respective lists.

Parameters

- **sig** (*list*) – Signal files to be matched to.
- **bg** (*list*) – Laoded background files, unmatched.
- **ref** (*list*) – Loaded reference files, umatched.
- **refbg** (*list*) – Loadaed reference background files, unmatched.
- **refid** (*list*) – List of the refIDs that correspond to each reference file.
- **directory** (*str*) – Directory where all the data files are located.

Returns

- **sig** (*list*) – Signal files to be matched to.
- **bg_matched** (*list*) – Background files matched to signal files.
- **ref_matched** (*list*) – Reference files matched to signal files.
- **refbg_matched** (*list*) – Reference background files matched to signal files.
- **sig_ref_id** (*list*) – RefIDs of the reference file each signal file needs.
- *ref_num* – RefIDs of the reference files only.

match_files_with_background(*filenames*, *bg_filenames*, *directory*)

Match each signal file with the appropriate background file.

First checks if there are any files that have the same name when bg_string is removed. If there are then these are defined as the background. If not, then the nearest background file to the signal file (in terms of creation time) is chosen.

Could be updated to include some way of strictly matching polarisations, rather than implicitly as here.

Parameters

- **filenames** (*list*) – Contains the signal filenames that need to be matched with a background.
- **bg_filenames** (*list*) – Contains the background filenames to match with.
- **directory** (*str*) – The directory containing the background and signal filenames.

match_files_with_reference(*sig_filenames*, *ref_filenames*, *ref_id*, *directory*)

Match each signal file with the appropriate reference file.

As there are generally going to be many more signal files than references, each reference is given an ID which is then matched to the signal file. An array containing the refIDs for each signal file is returned.

Finds the closest reference file to the signal file used in terms of creation time.

Parameters

- **sig_filenames** (*list*) – Contains the signal filenames that need to be matched with a reference.
- **ref_filenames** (*list*) – Contains the reference filenames to match with.
- **ref_id** (*list*) – Contains the refID of each reference file
- **directory** (*str*) – The directory containing the background and signal filenames.

Returns **sig_ref_id** – Contains the refID of the reference file that each signal file needs to be normalised to.

Return type list

static match_polarisations_bg(*pol*, *bg_names*)

UNUSED. Identify bg files with different polarisations to signal files.

e.g. If signal file is in SSP polarisation, then a background file could either be SSP or SSS.

NOT IMPLEMENTED IN VERSION 1.0

static nm_to_cm(*data*)
Convert data from nanometre to wavenumber.

static nm_to_thz(*data*)
Convert data from nanometre to terahertz.

open_spe(*fname*, *datastore*, *flag*)
Open an .spe file and send it to the correct reader method.

Different spectroscopy cameras read off different versions of the .spe file that is currently used a lot (why people don't just save the spectrum they want as a .txt file, given that this is how 99% of spectrscopy is done, is a good question). Andor cameras generally read .spe version 2.5, and Princeton Instruments cameras (who maintain the .spe filetype) generally read version 3.0. The difference is largely in flexibility and metadata. See the SPE3.0 file format declaration for info (online in places).

Parameters

- **fname** (*str*) – Name of file to be opened.
- **datastore** (*SFGDataStore object*) – Datastore to put the data read from the file into.
- **flag** (*str*) – Determines whether the data is stored as signal, background, reference, or reference background. Possible values "sig", "bg", "ref", "refbg".

parse_filename(*directory*, *file*, *datastore*)
Parse the name of file and populate datastore with relevant info.

Populates various attributes of datastore with helpful info stored in the filename, such as polarisation, wavelength, creation time, and index/group. Not required in version 1.0 but could be useful later.

Parameters

- **directory** (*str*) – The directory that the files are stored in.
- **file** (*str*) – The name of the file to be parsed.
- **datastore** (*SFGDataStore object*) – The datastore to populate with the information gained from parsing.

plot_data(*datastore*, *iteration*, *num_files*, *figure*)
Plot processed SFG data from datastore to figure.

Plots data contained in datastore, defaults to plotting normalised and subtracted data if present, otherwise subtracted, otherwise plots raw. If custom_region parameters are set in the GUI, then it will constrain the x axis to that region and rescale y, otherwise full range is plotted.

Class attributes are used to determine other parameters that are mostly set by the GUI or plotting script.

Parameters

- **datastore** (*SFGDataStore object*) – Contains data to be plotted.
- **iteration** (*int*) – Current index of file being processed (for batch processing).
- **num_files** (*int*) – Total number of files to process.
- **figure** (*matplotlib.pyplot.figure object*) – Figure to plot data on (if stacking).

Returns **figure** – Figure that data was plotted on in the previous iteration.

Return type matplotlib.pyplot.figure object

populate_data_stores(*datastores*, *directory*, *signal_names*, *bg_names*, *ref_names*, *ref_bg_names*)
Populate the SFGDataStore classes in datastores by reading files.

Reads the data from the .spe file given in the supplied lists and populates the SFGDataStore classes accordingly. Uses bool flags from the class attributes to determine what to read.

The lists are assumed to be ordered, so that the datastore in elemnent [n] of datastores is populated using signal data from signal_names[n], background from bg_names[n], and so on. This can be sorted manually in the command line, but the match_files method does it automatically for the GUI.

Parameters

- **datastores** (*list*) – Contains empty SFGDataStore class instances to be populated.
- **directory** (*str*) – Directory that the data files are stored in.
- **signal_names** (*list*) – Contains filenames of the signal data files to be processed.
- **bg_names** (*list*) – Contains filenames of the background data files of the signal data files to be processed.
- **ref_names** (*list*) – Contains filenames of the reference data files to be processed.
- **ref_bg_names** (*list*) – Contains filenames of the background data files of the reference data files to be processed.

static print_attributes(*datastore*)

Print the attributes of datastore attractively.

process_data(*datastore*, *downconvert_check*, *subtract_check*, *normalise_check*, *exposure_check*, *calibrate_check*, *cosmic_kill_check*, *force=False*)

Process data stored in datastore according to provided check flags.

Datastore contains SFG data to be processed, and methods of the datastore class are called to process the data. Bool flags are used to determine what processing to do. Note that the order in which the methods are called is important.

Parameters

- **datastore** (*SFGDataStore object*) – Contains the data to be processed.
- **downconvert_check** (*bool*) – If true then downconvert the data.
- **subtract_check** (*bool*) – If true then background subtract the data.
- **normalise_check** (*bool*) – If true then normalise the data to reference.
- **exposure_check** (*bool*) – If true then divide all spectra by exposure time.
- **calibrate_check** (*bool*) – If true then shift data energy axis by provided offset.
- **cosmic_kill_check** (*bool*) – If true then remove cosmic rays using cosmic_ray_killer().
- **force** (*bool, optional.*) – If true then allow data to be (e.g.) downconverted more than once (default False).

process_spe2x(*binaryfile*, *datastore*, *flag*)

Process data from an SPE 2.x file and put it in the relevant datastore.

Reads the SPE file at the places defined in the binary header. The output data is in the shape (frame-height, framewidth), so is normally (1, n) for a standard spectrum. Extra dimensions are added if the full chip is read out, or if there is more than one frame in the SPE file.

ftp://ftp.piacton.com/Public/Manuals/Princeton%20Instruments/SPE%203.0%20File%20Format%20Specification%20Issue%206%20(4411-0140).pdf Most of the options are class attributes and set externally by the GUI and not passed directly.

Parameters

- **binaryfile** (*binary file object*) – The opened .spe file to be processed.
- **datastore** (*SFGDataStore object*) – Where the data is stored to.
- **flag** (*str*) – Determines where in datastore the data is saved. Possible values “sig”, “bg”, “ref”, “refbg”.

process_spe3x(*binaryfile*, *datastore*, *flag*)
Process data from an SPE 3.0 file and put it in the relevant datastore.

Reads the SPE file at the places defined in the XML footer. The output data is in the shape (frameheight, framewidth), so is normally (1, n) for a standard spectrum. Extra dimensions are added if the full chip is read out, or if there is more than one frame in the SPE file.

Most of the options are class attributes and set externally by the GUI and not passed directly.

ftp://ftp.piacton.com/Public/Manuals/Princeton%20Instruments/SPE%203.0%20File%20Format%20Specification%20Issue%206%20(4411-0140).pdf

Parameters

- **binaryfile** (*binary file object*) – The opened .spe file to be processed.
- **datastore** (*SFGDataStore object*) – Where the data is stored to.
- **flag** (*str*) – Determines where in datastore the data is saved. Possible values "sig", "bg", "ref", "refbg".

pull_trigger()
Start the processing sequence.

Used mainly in the GUI. When all files are read in and sorted properly, this will create datastores, populate datastores, and process all the data using batch_process() according to the flags supplied. Parameters are all class attributes.

The lists containing data files all need to be properly matched and sorted for this to make sense.

static read_at(*file*, *pos*, *size*, *ntype*)
Read a binary file at a specific position in bytes.

Parameters

- **file** (*file object*) – Opened binary file that you want to read from.
- **pos** (*int*) – Location (in bytes) to start reading from.
- **size** (*int*) – Number of items after pos to read. -1 reads all items.
- **ntype** (*data type*) – Data type the binary is encoded in.

Returns **data** – The data read from the binary file.

Return type ntype

read_files(*fname*, *datastore*, *flag*)
Read fname and put the data in the right place in datastore using flag.

Currently this implementation looks pointless, but it is made so that eventually filetypes other than .spe can be read in the same program.

Parameters

- **fname** (*str*) – File to be read in.
- **datastore** (*SFGDataStore object*) – Datastore to put the data from the file into.
- **flag** (*str*) – Tells the .spe reader where to put this data in datastore. Options are "sig", "bg", "ref", "refbg".

static remove_duplicate_refs(*reftabledata*)
Remove duplicate references from the reftabledata.

Used to stop the GUI showing as many ref files as signal files after matching and sorting.

static slice_data(*datastore*, *data_in*, *data_out*)
Slice a data array into a consistent shape.

For most cases will slice to an array of dimensions (1, framewidth), but if the binning isnt to one pixel then this will become (frameheight, framewidth). Keeping that first dimension for future consistency.

Parameters

- **datastore** (*SFGDataStore object*) – Where the framewidth and frameheight are stored for the indexing.
- **data_in** (*np array*) – 1D array of data read from the data file.
- **data_out** (*np array*) – (frameheight, framewidth) shaped empty (or zero) array.

Returns **data_out** – (frameheight, framewidth) shaped array containing data.

Return type np array

static thz_to_cm(*data*)
Convert data from terahertz to wavenumber.

static thz_to_nm(*data*)
Convert data from terahertz to nanometre.

update_datatable()
Update data to be shown in the main datatable with current filenames.

update_reftable(*remove_duplicates=False*)
Update data to be shown in the reference datatable with current filenames.

write_data_to_file(*datastore*, *directory*)
Write data in datastore to a text file in directory.

Data is written into a text file as columns with a 12 line header explaining what the data is and how it has been processed. Nine columns are written with different types of data, but column 0 and 1 are the ones that contain the useful data in most cases.

Parameters

- **datastore** (*SFGDataStore object*) – Where the data to be written is stored.
- **directory** (*str*) – Where the resulting .txt file is to be saved.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S