pycroscopy Documentation

Release 0.60.2

Suhas Somnath, Chris Smith, Numan Laanait

Jul 31, 2018

CONTENTS

Scientific analysis of nanoscale materials imaging data

Jump to our GitHub project page

CHAPTER

ONE

DOCUMENTATION INDEX

—

1.1 Pycroscopy

Scientific analysis of nanoscale materials imaging data

Note: We are conducting a workshop on the fundamental tools that power pycroscopy (pyUSID) on Aug 13. Please see this page for more information on joining remotely

1.1.1 What?

- pycroscopy is a python package for processing, analyzing, and visualizing multidimensional imaging and spectroscopy data.
- pycroscopy uses the **Universal Spectroscopy and Imaging Data (USID)** model as its foundation, which:
  - facilitates the representation of any spectroscopic or imaging data regardless of its origin, modality, size, or dimensionality.
  - enables the development of instrument- and modality- agnostic data processing and analysis algorithms.
- pycroscopy uses a data-centric model wherein the raw data collected from the instrument, results from analysis and processing routines are all written to USID **hierarchical data format (HDF5)** files for traceability, reproducibility, and provenance.
- pycroscopy uses pyUSID which provides tools to read, write, visualize, and process **USID** stored in HDF5 files. In addition, pycroscopy uses popular packages such as numpy, scipy, scikit-image, scikit-learn, joblib, matplotlib, etc. for most of the computation, analysis and visualization.
- You can choose to perform your analysis outside pycroscopy if you prefer and use pyUSID to standardize the data storage.
- See a high-level overview of pycroscopy in this presentation
- See scientific research enabled by pycroscopy.
- Jump to our GitHub project

With pycroscopy we aim to:

1. significantly lower the barrier to advanced data analysis procedures by simplifying I/O, processing, visualization, etc.
2. serve as a hub for collaboration across scientific domains (microscopists, material scientists, biologists…)

1.1.2 Why?

As we see it, there are a few opportunities in scientific imaging (that surely apply to several other scientific domains):

1. Growing data sizes

- Cannot use desktop computers for analysis
- *Need: High performance computing, storage resources and compatible, scalable file structures*

2. Increasing data complexity

- Sophisticated imaging and spectroscopy modes resulting in 5,6,7... dimensional data
- *Need: Robust software and generalized data formatting*

3. Multiple file formats

- Different formats from each instrument. Proprietary in most cases
- Incompatible for correlation
- *Need: Open, instrument-independent data format*

4. Disjoint communities

- Similar analysis routines written by each community (SPM, STEM, TOF SIMs, XRD...) *independently*!
- *Need: Centralized repository, instrument agnostic analysis routines that bring communities together*

5. Expensive analysis software

- Software supplied with instruments often insufficient / incapable of custom analysis routines
- Commercial software (Eg: Matlab, Origin..) are often prohibitively expensive.
- *Need: Free, powerful, open source, user-friendly software*

6. Closed science

- Analysis software and data not shared
- No guarantees of reproducibility or traceability
- *Need: open source data structures, file formats, centralized code and data repositories*

1.1.3 How?

- pycroscopy uses the Universal Spectroscopy and Imaging Data model that facilitates the storage of data, regardless of dimensionality (conventional 1D spectra and 2D images to 9D hyperspectral datasets and beyond!) or instrument of origin (AFMs, STEMs, Raman spectroscopy etc.).
- This generalized representation of data allows us to write a single and generalized version of analysis and processing functions that can be applied to any kind of data.
- The data are stored in hierarchical data format (HDF5) files which have numerous benefits including flexibility in storing multiple datasets of arbitrary sizes and dimensionality, supercomputer compatibility, storage of important metadata.
- Once the relevant data and metadata are extracted from proprietary raw data files and written into USID HDF5 files via a translation process, the user gains access to the rest of the utilities present in `pycroscopy.*`.
- Scientific workflows are developed and disseminated through jupyter notebooks that are interactive and portable web applications containing text, images, code / scripts, and graphical results. Notebooks containing the complete / parts of workflow from raw data to publishable figures often become supplementary material for journal publications thereby enabling traceability, reproducibility for open science.

1.1.4 Who?

- This project begun largely as an effort by scientists and engineers at the **I**nstitute for **F**unctional **I**maging of **M**aterials (IFIM) to implement a python library that can support the I/O, processing, and analysis of the gargantuan stream of images that their microscopes generate (thanks to the large IFIM users community!).
- It is now being developed and maintained by Suhas Somnath of the **A**dvanced **D**ata & **W**orkflows **G**roup (ADWG) at the **O**ak Ridge National Laboratory **L**eadership **C**omputing **F**acility (OLCF) and Chris R. Smith of IFIM.
- By sharing our methodology and code for analyzing scientific imaging data we hope that it will benefit the wider scientific community. We also hope, quite ardently, that other scientists would follow suit.
- Please visit our credits and acknowledgements page for more information.

1.2 Getting Started

- Follow these instructions to install pycroscopy
- See the many journal publications made possible by pycroscopy and try out the jupyter notebooks that were used for the paper
- Also see examples to see other scientific applications of pycroscopy
- Python novices are highly recommended to go through **Nick Mostovych's** guides and pitfalls for python novices
- We have also compiled a list of handy tutorials on basic / prerequisite topics such as programming in python, data analytics, machine learning, etc.
- See tutorials to get started on using and writing your own pyUSID functions that power pycroscopy
- **We already have many translators that transform data from popular microscope data formats to pycroscopy compatible H**
  - pyUSID also has a tutorial to get you started on importing your other data to pycroscopy.
- Details regarding the definition and guidelines for the Universal Spectroscopy and Imaging Data (USID) model and implementation in HDF5 are also available in pyUSID's documentation.
- Please see our document on the organization of pycroscopy to find out more on what is where and why.
- If you are interested in contributing your code to pycroscopy, please look at our guidelines
- If you need detailed documentation on all our classes, functions, etc., please visit our API
- Have questions? See our FAQ to see if we have already answered them.
- Need help or need to get in touch with us? See our contact information.

1.3 Installation

1.3.1 Preparing for pycroscopy

Pycroscopy requires many commonly used scientific and numeric python packages such as numpy, scipy etc. To simplify the installation process, we recommend the installation of Anaconda which contains most of the prerequisite packages, conda - a package / environment manager, as well as an interactive development environment - Spyder.

1. Recommended - uninstall existing Python distribution(s) if installed. Restart computer afterwards.

2. Install Anaconda 4.2 (Python 3.5) 64-bit - Mac / Windows / Linux

Compatibility

- Pycroscopy was initially developed in python 2 but all current / future development for pycroscopy will be on python 3.5+. Nonetheless, we will do our best to ensure continued compatibility with python 2.
- We currently do not support 32 bit architectures
- We only support text that is UTF-8 compliant due to restrictions posed by HDF5

1.3.2 Installing pycroscopy

Once the appropriate Anaconda distribution has been successfully installed, pycroscopy can be installed easily either via the python package index (pypi / pip) or conda-forge (conda).

pip installation

Open a terminal (mac / linux) or command prompt (windows - be sure to install in a location where you have write access. Don't install as administrator unless you are required to do so.) and type:

```
pip install pycroscopy
```

conda installation

First, open a terminal / command window. Add *conda-forge* to your channels with:

```
conda config --append channels conda-forge
```

Once the *conda-forge* channel has been enabled, *pycroscopy* can be installed with:

```
conda install pycroscopy
```

Installing a legacy version

The latest version of pycroscopy (as of May 2018) has been thoroughly revamped and is likely to have caused some incompatibilities in your existing code. In order to go back to / install the last version of pycroscopy before this major revision, please use the following commands in a terminal:

```
pip install pycroscopy==0.59.8
```

Installing from a specific branch (advanced users ONLY)

Here, we are installing pycroscopy from the latest development branch. Note that we do not recommend installing pycroscopy this way.

Before you can install pycroscopy, you need to install git.

```
conda install git
```

Once git has installed, you can install a specific branch of pycroscopy (`dev` in this case):

```
pip install -U git+https://github.com/pycroscopy/pycroscopy@dev
```

1.3.3 Updating pycroscopy

We recommend periodically updating your anaconda distribution. To fully update run the following commands in a terminal / command window.

```
conda upgrade anaconda
conda update --all
```

If you installed pycroscopy via conda, the last command should update pycroscopy as well.

Updating via pip

If you already have pycroscopy installed and want to update to the latest version, use the following command in a terminal / command window:

```
pip install -U --no-deps pycroscopy
```

If it does not work try reinstalling the package:

```
pip uninstall pycroscopy
pip install pycroscopy
```

Updating via conda

If you installed pycroscopy via *conda*, open a terminal / command window and type:

```
conda update pycroscopy
```

1.3.4 Other software

We recommend HDF View for exploring HDF5 files generated by and used in pycroscopy.

1.4 Tutorials on Basics

Suhas Somnath and Chris Smith

1.4.1 Python, packages, editors, . . .

Our sister project, **PyUSID**, has a list of excellent tutorials on most common topics such as on reading, using / running and writing code. Tutorials the ones relevant to pycroscopy are listed below.

1.4.2 Data Science

The following tutorials provide an overview of different concepts ranging from data wrangling to statistical and machine learning:

- Coursera Machine learning course by Andrew Ng
- Python for Data Science book with Notebooks
- Python DataCamp
- Coursera Deep Learning specialization by Andrew Ng

1.5 Package Organization

1.5.1 Sub-packages

The package structure is simple, with 5 main modules:

1. `io`: Translators to extract data from custom & proprietary microscope formats and write them to HDF5 files.
2. `analysis`: Physics-dependent analysis of information.
3. `processing`: Physics-independent processing of data including machine learning, image processing, signal filtering.
4. `viz`: Plotting functions and interactive jupyter widgets for scientific applications
5. `simulation`: Simulations and modelling here

pycroscopy.io

- `HDFWriter` (previously named `ioHDF5`) - Legacy class that was used for writing HDF5 files. This is now deprecated and will be removed in a future release.
- `VirtualDataset` (previously named `MicroDataset`), `VirtualGroup` (previously named `MicroDataGroup`) - Legacy classes that were virtual representations of HDF5 dataset and HDF5 group objects. Both these objects have been deprecated and will be removed in a future release.
- `write_utils` - Utilities that assist in writing ancillary datasets using `VirtualDataset` objects. The functions in this module are deprecated and will be removed in a future release.
- `translators` - subpackage containing several `Translator` classes that extract data from custom & proprietary microscope formats and write them to HDF5 files.
  - `df_utils` - any utilities that assist in the functioning of the `Translator` classes in `translators`

pycroscopy.processing

- `Cluster` - Facilitates data transformation to perform clustering using sklearn and writing of results to file
- `Decomposition` - Facilitates data transformation to perform decomposition using sklearn and writing of results to file
- `SVD` - Facilitates data transformation to perform SVD using sklearn and writing of results to file. Plus SVD related utilities such as rebuilding data using SVD results
- `SignalFilter` - Class that facilitates FFT-based signal filtering

- `fft` - module with FFT and FFT filtering related functions
- `gmode_utils` - Utilities that support `SignalFilter`
- `tree` - utilities that facilitate the propagation of information (for example endmembers from clustering)
- `proc_utils` - utilities used by Cluster, SVD, Decomposition
- `histogram` - Utilities for generating histograms on spectral data
- `image_processing` - Classes for enabling image-windowing and SVD-based image cleaning
- `contrib` - user contributed code

pycroscopy.analysis

- `Fitter` - An absrtact class that supports science-agnostic functions that facilitate the fitting of data to analytical functions
- `optimize` - A class and set of utilities that facilitate parallel functional fitting using scipy.optimize
- `guess_methods` - A set of functions that provide good initial values for functional fitting
- `fit_methods` - A set of functions that facilitate functional fitting.
- `BESHOFitter` - A class that handles the fitting of piezoresponse spectra to a simple harmonic oscillator model
- `BELoopFitter` - A class that handles fitting of piezoresponse hysteresis loops to an analytical function
- `GIVBayesian` - A class that performs Bayesian inference on G-mode current-voltage data
- `contrib` - A collection of user contribution code

pycroscopy.simulation

- AFM simulation and many more to come.

pycroscopy.core

This engineering-focused sub-package has been moved to a new package - pyUSID. pyUSID supports science-focused packages such as pycroscopy similar to how scipy depends on numpy. The current release of pycroscopy imports pyUSID and makes it available as `pycroscopy.core` so that existing imports do not break. In the next release of pycroscopy, this implicit import will be removed and the modules would have to be imported directly from `pyUSID`. See the what's new under **June 28 2018**.

1.5.2 Branches

- `master` : Stable code based off which the pip installer works. Recommended for most people.
- `dev` : Experimental code with new features that will be made available in `master` periodically after thorough testing. By its very definition, this branch is recommended only for developers.

1.6 Examples & Tutorials

Please visit this page to see the many papers that used pycroscopy for their research

1.6.1 Tutorials on fundamental data tools

See tutorials on how to use pyUSID in order to learn the basics that will help you write code in / for pycroscopy.

1.6.2 Scientific Data Analysis

Note: Click *here* to download the full example code

FFT & Filtering of Atomically Resolved Images

Stephen Jesse and Suhas Somnath

- Institute for Functional Imaging of Materials
- Center for Nanophase Materials Sciences

Oak Ridge National Laboratory, Oak Ridge TN 37831, USA

9/28/2015

Fourier transforms offer a very fast and convenient means to analyze and filter multidimensional data including images. The power of the Fast Fourier Transform (FFT) is due in part to (as the name suggests) its speed and also to the fact that complex operations such as convolution and differentiation/integration are made much simpler when performed in the Fourier domain. For instance, convolution in the real space domain can be performed using multiplication in the Fourier domain, and differentiation/integration in the real space domain can be performed by multiplying/dividing by $i\omega$ in the Fourier domain (where ω is the transformed variable). We will take advantage of these properties and demonstrate simple image filtering and convolution with example.

A few properties/uses of FFT's are worth reviewing.

In this example we will load an image, Fourier transform it, apply a smoothing filter, and transform it back.

```
from __future__ import division, unicode_literals, print_function
import matplotlib.pyplot as plt
import numpy as np
import numpy.fft as npf
import os
import subprocess
import sys

def install(package):
    subprocess.call([sys.executable, "-m", "pip", "install", package])
# for downloading files:
try:
    # This package is not part of anaconda and may need to be installed.
    import wget
except ImportError:
    print('wget not found.  Will install with pip.')
    import pip
    install('wget')
    import wget
try:
    import pyUSID as usid
except ImportError:
    print('pyUSID not found.  Will install with pip.')
```

(continues on next page)

(continued from previous page)

```
    import pip
    install('pyUSID')
    import pyUSID as usid
```

We will be using an image available on our GitHub project page by default. You are encouraged to download this document as a Jupyter Notebook (button at the bottom of the page) and use your own image instead. When using your own image, you can skip this cell and provide the path to your data using the variable - data_file_path

Coming back to our example, lets start by downloading the file from GitHub:

```
data_file_path = 'temp_STEM_STO.txt'
# download the data file from Github:
url = 'https://raw.githubusercontent.com/pycroscopy/pycroscopy/master/data/STEM_STO_2_
↪20.txt'
_ = wget.download(url, data_file_path, bar=None)
```

The image is stored as a tab delimited text file. One can load its contents to memory by using the following command:

```
image_raw = np.loadtxt(data_file_path, dtype='str', delimiter='\t')

# delete the temporarily downloaded file:
os.remove(data_file_path)

# convert the file from a string array to a numpy array of floating point numbers
image_raw = np.array(image_raw)
image_raw = image_raw[0:, 0:-1].astype(np.float)
```

Prior to transforming, it is sometimes convenient to set the image mean to zero, because if the mean value of an image is large, the magnitude of the zero-frequency bin can dominate over the other signals of interest.

```
image_raw = image_raw - np.mean(image_raw)  # subtract out the mean of the image
```

An important aspect of performing Fourier transforms is keeping track of units between transformations and also being aware of conventions used with regard to the locations of the image centers and extents when transformed. Below is code that builds the axes in the space domain of the original image.

```
x_pixels, y_pixels = np.shape(image_raw)  # [pixels]
x_edge_length = 5.0  # [nm]
y_edge_length = 5.0  # [nm]
x_sampling = x_pixels / x_edge_length  # [pixels/nm]
y_sampling = y_pixels / y_edge_length  # [pixels/nm]
x_axis_vec = np.linspace(-x_edge_length / 2, x_edge_length / 2, x_pixels)  # vector␣
↪of locations along x-axis
y_axis_vec = np.linspace(-y_edge_length / 2, y_edge_length / 2, y_pixels)  # vector␣
↪of locations along y-axis
x_mat, y_mat = np.meshgrid(x_axis_vec, y_axis_vec)  # matrices of x-positions and y-
↪positions
```

Similarly, the axes in the Fourier domain are defined below. Note, that since the number of pixels along an axis is even, a convention must be followed as to which side of the halfway point the zero-frequency bin is located. In Matlab, the zero frequency bin is located to the left of the halfway point. Therefore the axis extends from -ω_sampling/2 to one frequency bin less than +ω_sampling/2.

```
u_max = x_sampling / 2
v_max = y_sampling / 2
u_axis_vec = np.linspace(-u_max / 2, u_max / 2, x_pixels)
```

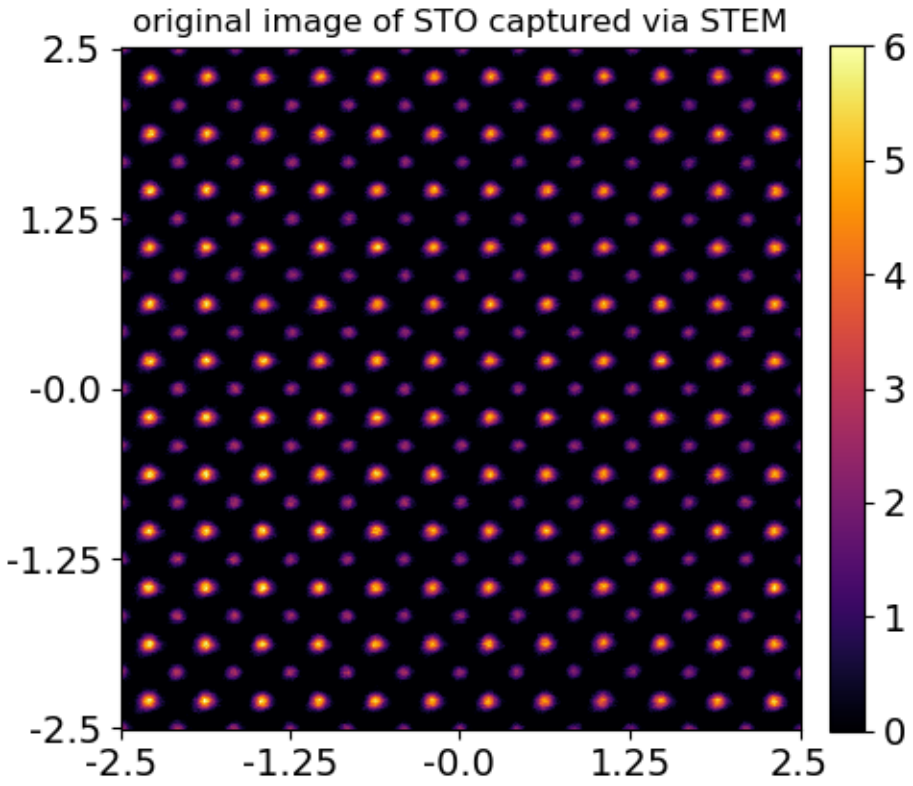
(continues on next page)

(continued from previous page)

```
v_axis_vec = np.linspace(-v_max / 2, v_max / 2, y_pixels)
u_mat, v_mat = np.meshgrid(u_axis_vec, v_axis_vec)  # matrices of u-positions and v-
↪positions
```

A plot of the data is shown below (STEM image of STO).

```
fig, axis = plt.subplots(figsize=(5, 5))
_ = usid.plot_utils.plot_map(axis, image_raw, cmap=plt.cm.inferno, clim=[0, 6],
                             x_vec=x_axis_vec, y_vec=y_axis_vec, num_ticks=5)
axis.set_title('original image of STO captured via STEM')
```

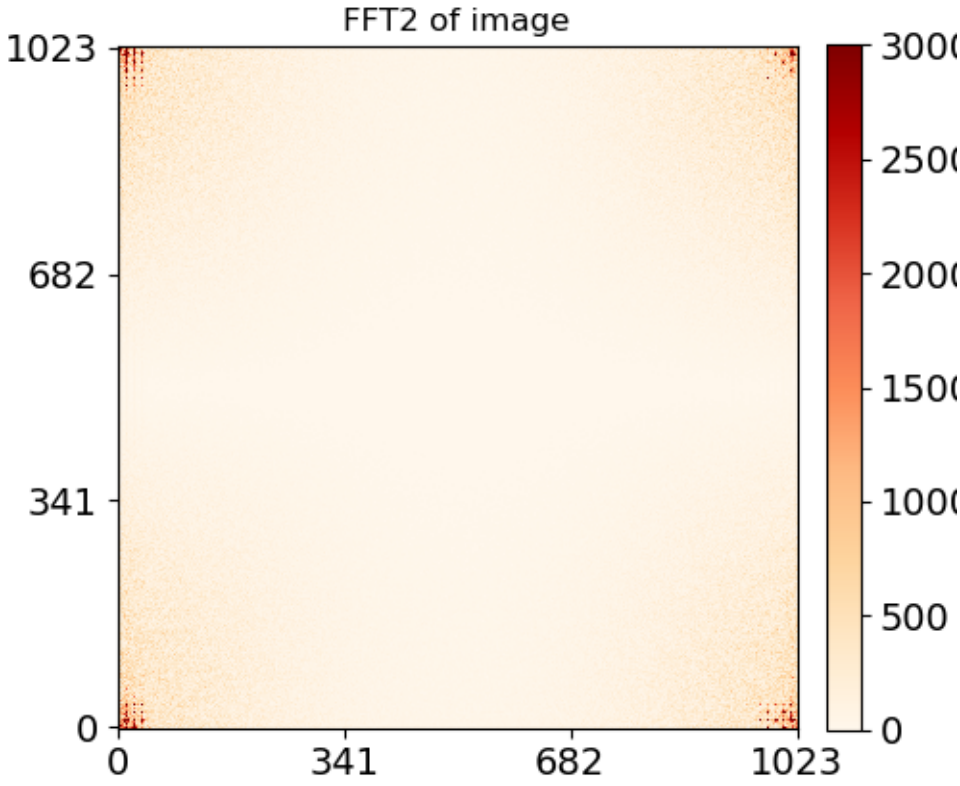


The Fourier transform can be determined with one line of code:

```
fft_image_raw = npf.fft2(image_raw)
```

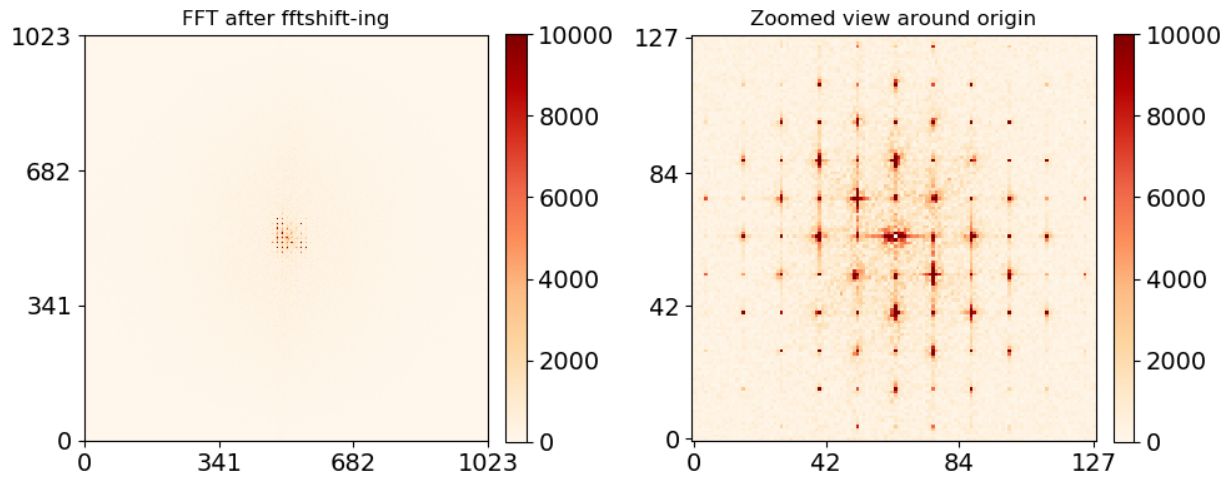
Plotting the magnitude 2D-FFT on a vertical log scales shows something unexpected: there appears to be peaks at the corners and no information at the center. This is because the output for the 'fft2' function flips the frequency axes so that low frequencies are at the ends, and the highest frequency is in the middle.

```
fig, axis = plt.subplots(figsize=(5, 5))
_ = usid.plot_utils.plot_map(axis, np.abs(fft_image_raw), cmap=plt.cm.OrRd, clim=[0,␣
↪3E+3])
axis.set_title('FFT2 of image')
```



To correct this, use the 'fftshift' command. fftshift brings the lowest frequency components of the FFT back to the center of the plot

```
fft_image_raw = npf.fftshift(fft_image_raw)
fft_abs_image_raw = np.abs(fft_image_raw)


def crop_center(image, cent_size=128):
    return image[image.shape[0]//2 - cent_size // 2: image.shape[0]//2 + cent_size //␣
↪2,
                 image.shape[1]//2 - cent_size // 2: image.shape[1]//2 + cent_size //␣
↪2]


# After the fftshift, the FFT looks right
fig, axes = plt.subplots(ncols=2, figsize=(10, 5))
for axis, img, title in zip(axes, [fft_abs_image_raw, crop_center(fft_abs_image_raw)],
↪ ['FFT after fftshift-ing',
                                                                                    ␣
↪  'Zoomed view around origin']):
    _ = usid.plot_utils.plot_map(axis, img, cmap=plt.cm.OrRd, clim=[0, 1E+4])
    axis.set_title(title)
fig.tight_layout()
```

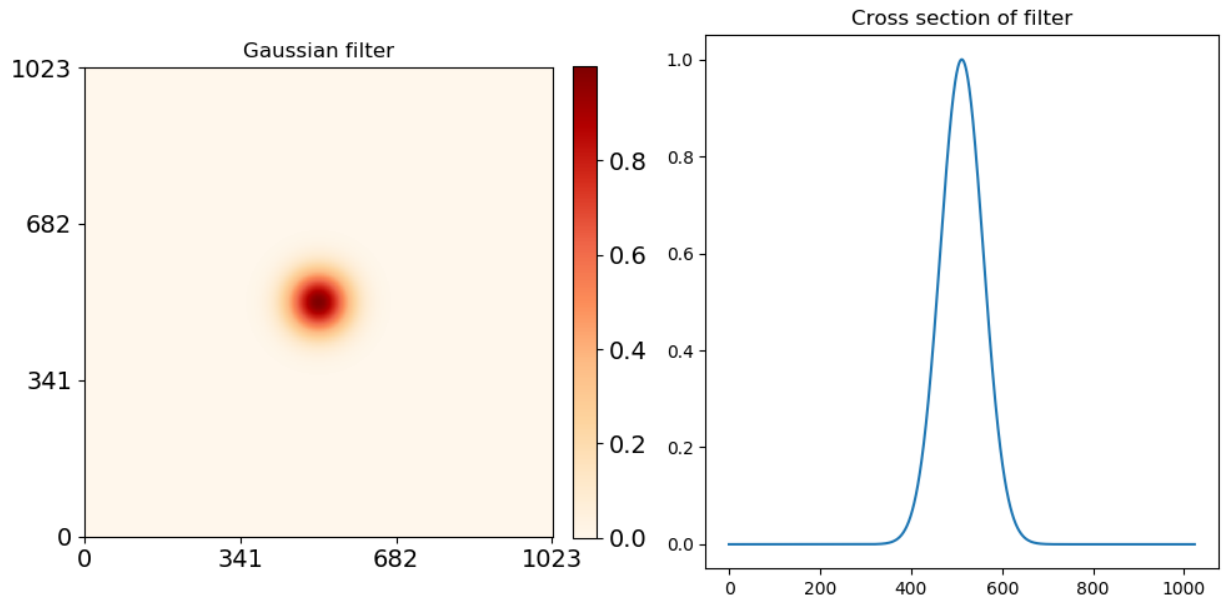


The first filter we want to make is a 2D, radially symmetric, low-pass Gaussian filter. To start with, it is helpful to redefine the Fourier domain in polar coordinates to make building the radially symmetric function easier.

```
r = np.sqrt(u_mat**2+v_mat**2) # convert cartesian coordinates to polar radius
```

An expression for the filter is given below. Note, the width of the filter is defined in terms of the real space dimensions for ease of use.

```
filter_width = .15  # inverse width of gaussian, units same as real space axes
gauss_filter = np.e**(-(r*filter_width)**2)

fig, axes = plt.subplots(ncols=2, figsize=(10, 5))
_ = usid.plot_utils.plot_map(axes[0], gauss_filter, cmap=plt.cm.OrRd)
axes[0].set_title('Gaussian filter')
axes[1].plot(gauss_filter[gauss_filter.shape[0]//2])
axes[1].set_title('Cross section of filter')
fig.tight_layout()
```

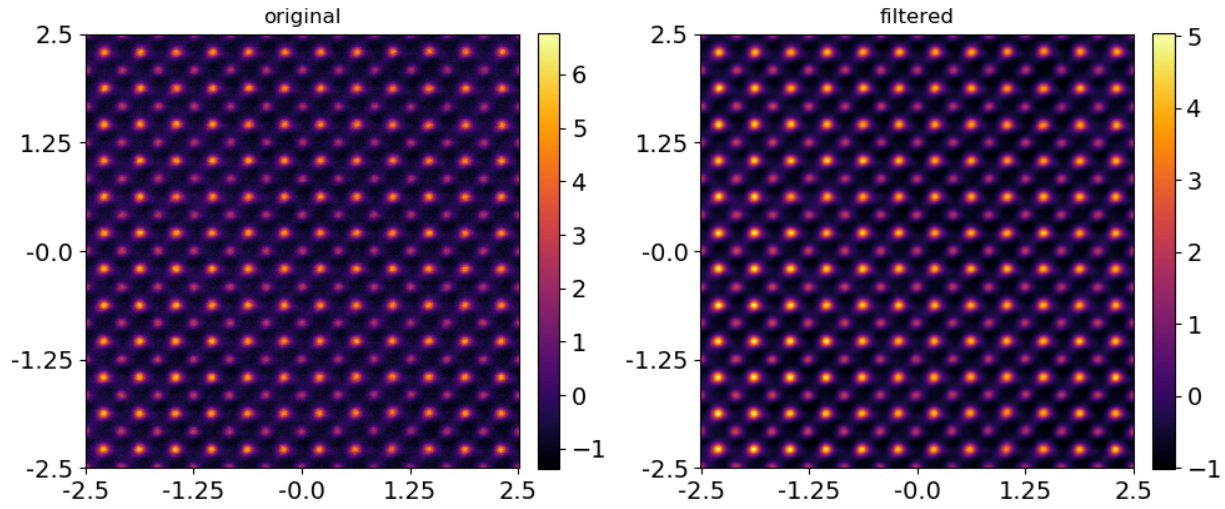


Application of the filter to the data in the Fourier domain is done simply by dot-multiplying the two matrices.

```
F_m1_filtered = gauss_filter * fft_image_raw
```

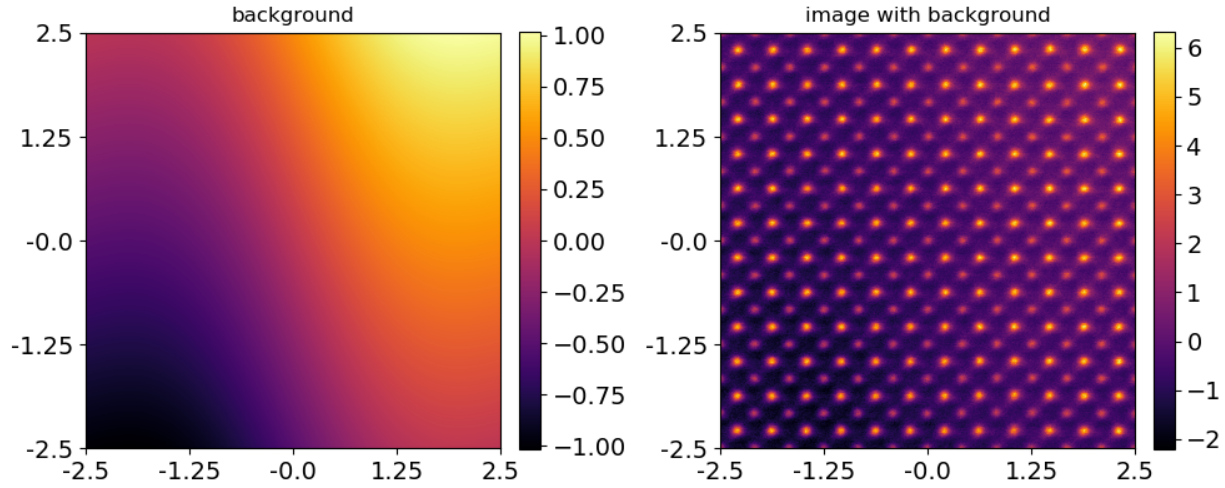
To view the filtered data in the space domain, simply use the inverse fast Fourier transform ('ifft2'). Remember though that python expects low frequency components at the corners, so it is necessary to use the inverse of the 'fftshift' command ('ifftshift') before performing the inverse transform. Also, note that even though theoretically there should be no imaginary components in the inverse transformed data (because we multiplied two matrices together that were both symmetric about 0), some of the numerical manipulations create small round-off errors that result in the inverse transformed data being complex (the imaginary component is ~1X1016 times smaller than the real part). In order to account for this, only the real part of the result is kept.

```
image_filtered = npf.ifft2(npf.ifftshift(F_m1_filtered))
image_filtered = np.real(image_filtered)

fig, axes = plt.subplots(ncols=2, figsize=(10, 5))
for axis, img, title in zip(axes, [image_raw, image_filtered], ['original', 'filtered
↪']):
    _ = usid.plot_utils.plot_map(axis, img, cmap=plt.cm.inferno,
                                 x_vec=x_axis_vec, y_vec=y_axis_vec, num_ticks=5)
    axis.set_title(title)
fig.tight_layout()
```



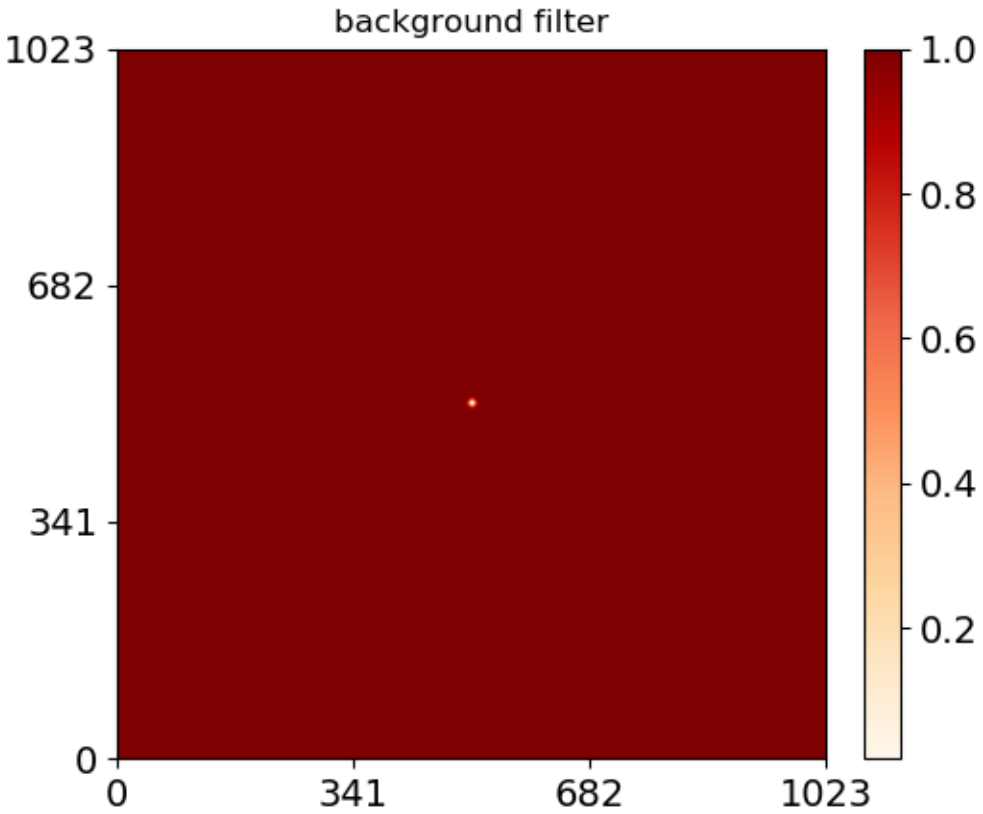
Filtering can also be used to help flatten an image. To demonstrate this, let's artificially add a background to the original image, and later try to remove it.

```
background_distortion = 0.2 * (x_mat + y_mat + np.sin(2 * np.pi * x_mat / x_edge_
↪length))
image_w_background = image_raw + background_distortion

fig, axes = plt.subplots(figsize=(10, 5), ncols=2)
for axis, img, title in zip(axes, [background_distortion, image_w_background], [
↪'background', 'image with background']):
    _ = usid.plot_utils.plot_map(axis, img, cmap=plt.cm.inferno,
                                 x_vec=x_axis_vec, y_vec=y_axis_vec, num_ticks=5)
    axis.set_title(title)
fig.tight_layout()
```



Since large scale background distortions (such as tilting and bowing) are primarily low frequency information. We can make a filter to get rid of the lowest frequency components. Here we again use a radially symmetric 2D Gaussian, however in this case we invert it so that it is zero at low frequencies and 1 at higher frequencies.

```
filter_width = 2  # inverse width of gaussian, units same as real space axes
inverse_gauss_filter = 1-np.e**(-(r*filter_width)**2)

fig, axis = plt.subplots()
_ = usid.plot_utils.plot_map(axis, inverse_gauss_filter, cmap=plt.cm.OrRd)
axis.set_title('background filter')
```



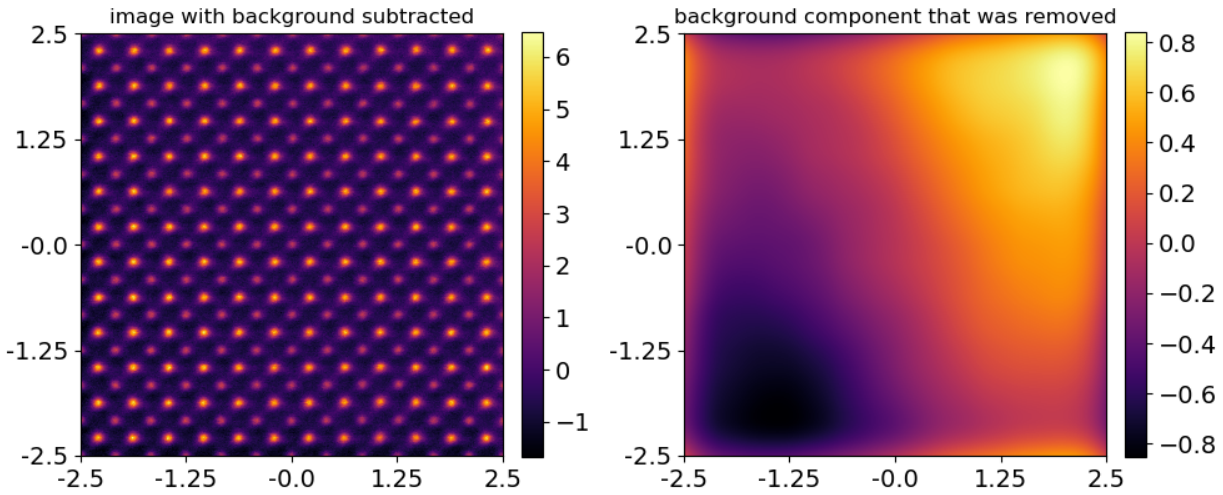
Let's perform the same process of taking the FFT of the image, multiplying with the filter and taking the inverse Fourier transform of the image to get the filtered image.

```
# take the fft of the image
fft_image_w_background = npf.fftshift(npf.fft2(image_w_background))
fft_abs_image_background = np.abs(fft_image_w_background)

# apply the filter
fft_image_corrected = fft_image_w_background * inverse_gauss_filter

# perform the inverse fourier transform on the filtered data
image_corrected = np.real(npf.ifft2(npf.ifftshift(fft_image_corrected)))

# find what was removed from the image by filtering
filtered_background = image_w_background - image_corrected
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(ncols=2, figsize=(10, 5))
for axis, img, title in zip(axes, [image_corrected, filtered_background],
                            ['image with background subtracted',
                             'background component that was removed']):
    _ = usid.plot_utils.plot_map(axis, img, cmap=plt.cm.inferno,
                                 x_vec=x_axis_vec, y_vec=y_axis_vec, num_ticks=5)
    axis.set_title(title)
fig.tight_layout()
```



Total running time of the script: (0 minutes 2.606 seconds)

Note: Click *here* to download the full example code

Image alignment and registration

Stephen Jesse, Alex Belianinov, Suhas Somnath

- Institute for Functional Imaging of Materials
- Center for Nanophase Materials Sciences

Oak Ridge National Laboratory, Oak Ridge TN 37831, USA

7/29/2015

Often scientists find themselves wanting to compare data of various origins on the same sample that has a location of interest where multiple experiments have been carried out. Often, these data sets may not have the same resolution, not have been captured over the exact same region, be distorted in different ways due to drift, and even have different dimensionality. In this example, we will make use of algorithms which attempt to find the best alignment transformation between data sets.

Below, the Normalized Topo Image is a result of STM (scanning tunneling microscope) imaging captured prior to collecting a lower spatial resolution, 3D scanning tunneling spectroscopic (STS) data set that experienced drift as is

evident shown in Z Channel Spectroscopy (where Z is height roughly corresponding to topography). We would like to map the spectroscopic result onto exact locations in the 2D map (high resolution) to correlate topo features to the electronic effects captured by STS. To do this, the image (and the associated full 3D data set) on the right needs to be transformed to look like the image on the left.

Gather our tools and load necessary libraries

```
from __future__ import print_function, division, unicode_literals  # ensure python3␣
↪compatibility
import numpy as np  # fast math
from warnings import warn
import matplotlib.pyplot as plt  # plotting
import h5py  # reading the data file
import os  # file operations
from scipy import interpolate, stats  # various convenience tools
from skimage import transform  # image processing and registration
import subprocess
import sys

def install(package):
    subprocess.call([sys.executable, "-m", "pip", "install", package])
try:
    # This package is not part of anaconda and may need to be installed.
    import wget
except ImportError:
    warn('wget not found.  Will install with pip.')
    import pip
    install('wget')
    import wget
try:
    import pyUSID as usid # used mainly for visualization purposes here
except ImportError:
    warn('pyUSID not found.  Will install with pip.')
    import pip
    install('pyUSID')
    import pyUSID as usid
```

Defining a few handy functions that will be reused multiple times

```
def twin_image_plot(images, titles, cmap=plt.cm.viridis):
    """
    Handy function that plots two images side by side with colorbars

    Parameters
    ----------
    images : list or array-like
        List of two images defined as 2D numpy arrays
    titles : list or array-like
        List of the titles for each image
    cmap : (Optional) matplotlib.pyplot colormap object or string
        Colormap to use for displaying the images

    Returns
    -------
    fig : Figure
        Figure containing the plots
    axes : 1D array_like of axes objects
        Axes of the individual plots within `fig`
```

(continues on next page)

(continued from previous page)

```
    """
    fig, axes = plt.subplots(ncols=2, figsize=(10, 5))
    for axis, img, title in zip(axes.flat, images, titles):
        usid.plot_utils.plot_map(axis, img, cmap=cmap)
        axis.set_title(title)
    fig.tight_layout()
    return fig, axes


def normalize_image(image):
    """
    Normalizes the provided image from 0 to 1

    Parameters
    ----------
    image : np.array object
        Image to be normalized

    Returns
    -------
    image : np.array object
        Image normalized from 0 to 1
    """
    return (image - np.amin(image)) / (np.amax(image) - np.amin(image))
```

Load the data from the hdf5 file

We will be using an data file available on our GitHub project page by default. You are encouraged to download this document as a Jupyter Notebook (button at the bottom of the page) and use your own data instead. When using your own data, you can skip this cell and provide the path to your data using the variable - h5_path

We begin by loading the high resolution STM image, the Z component image of the spectroscopic data set, and the spectroscopic data set itself

```
# Downloading the example file from the pycroscopy Github project
url = 'https://raw.githubusercontent.com/pycroscopy/pycroscopy/master/data/sts_data_
↪image_registration.h5'
h5_path = 'temp.h5'
_ = wget.download(url, h5_path, bar=None)

print('Working on:\n' + h5_path)

with h5py.File(h5_path, mode='r') as h5_f:
    sts_spectral_data = h5_f['sts_spectra'][()]  # STS spectral data set
    high_res_topo = h5_f['stm_topography'][()]  # STM image
    sts_z_contr = h5_f['sts_z_contrast'][()]  # STS Z contrast image
```

Out:

```
Working on:
temp.h5
```

Normalize images

```
high_res_topo = normalize_image(high_res_topo)
sts_z_contr = normalize_image(sts_z_contr)
```

Shapes of datasets

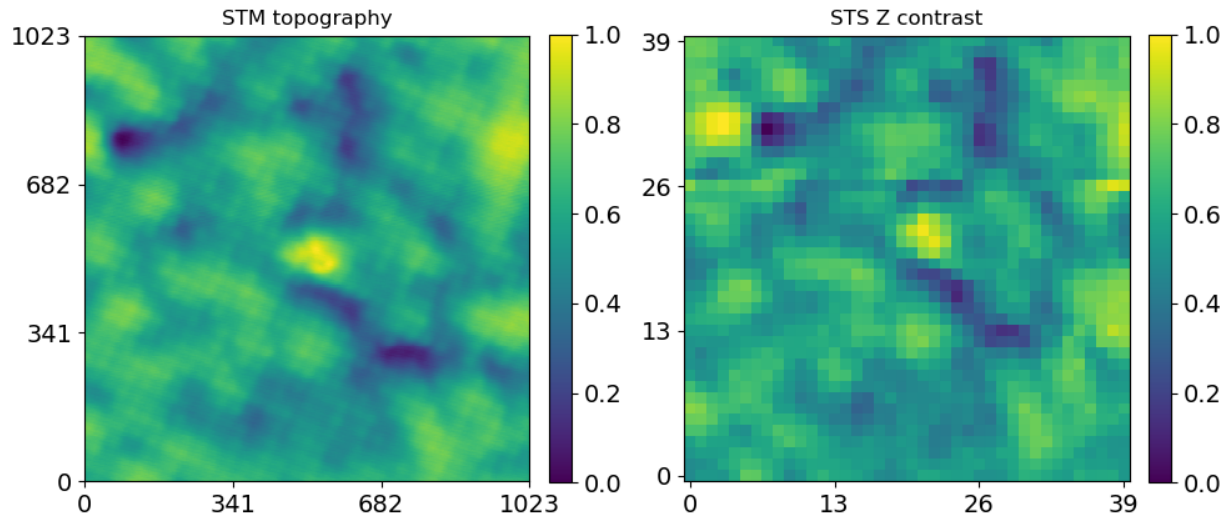
```
print('STS Spectra shape:', sts_spectral_data.shape)
print('STM Topography shape:', high_res_topo.shape)
print('STS Z contrast shape:', sts_z_contr.shape)
```

Out:

```
STS Spectra shape: (40, 40, 256)
STM Topography shape: (1024, 1024)
STS Z contrast shape: (40, 40)
```

visualization

```
fig, axes = twin_image_plot([high_res_topo, sts_z_contr],
                            ['STM topography', 'STS Z contrast'])
```



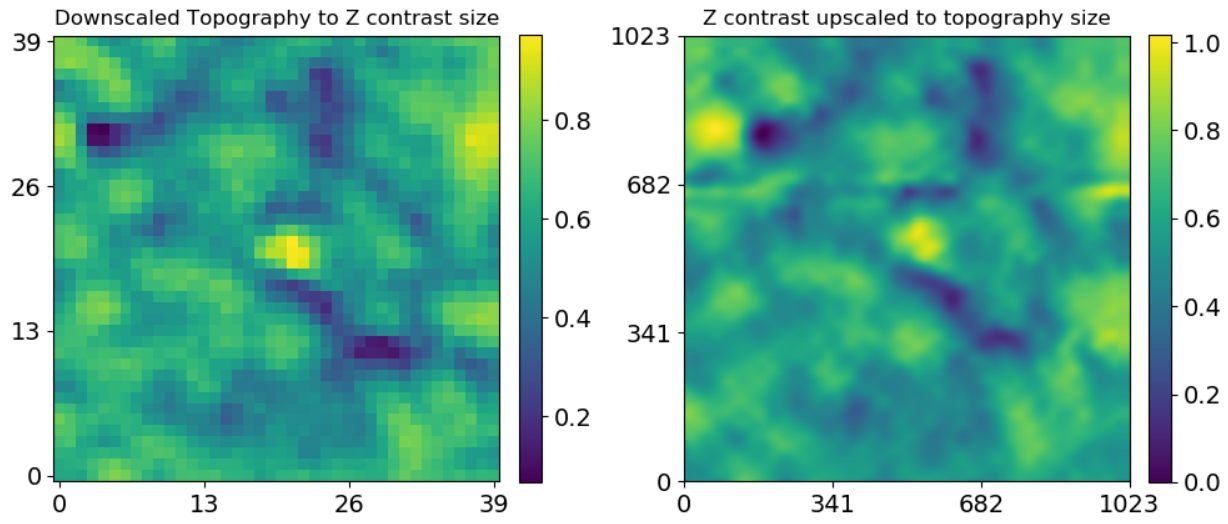
Interpolate image and the Z channel data

Since our goal is to maximize overlap between the two datasets, we should create some additional datasets with matching sizes. The Topo image data is 1024 x 1024 pixels, whereas the STS data is 40 x 40 spatial pixels (plus a 3rd dimension of 256 points)

Let's create two additional images – an interpolated STS image that is now 1024 x 1024 and a reduced Topo image that's 40 x 40

First we create an interpolant model, and then evaluate it at an interval that would give us the desired image size

```
z_shape = sts_z_contr.shape
topo_shape = high_res_topo.shape

z_upscaler = interpolate.RectBivariateSpline(np.arange(z_shape[0]),
                                             np.arange(z_shape[1]),
                                             sts_z_contr)

# we now use the instance of the class to run it on a new set of values, which
# are still 1 to 40, but now take 40/1024 size steps returning 1024x1024 image
z_upscaled = z_upscaler(np.arange(0, z_shape[0], z_shape[0] / topo_shape[0]),
                        np.arange(0, z_shape[1], z_shape[1] / topo_shape[1]))

# repeat the process for downscaling as above, but simply reverse the direction
# feed the fine steps first & then...
topo_downscaler = interpolate.RectBivariateSpline(np.arange(0, z_shape[0], z_shape[0]␣
↪/ topo_shape[0]),
                                                  np.arange(0, z_shape[1], z_shape[1]␣
↪/ topo_shape[1]),
                                                  high_res_topo)

# use the class with only big steps, downscalign to a 40x40
topo_downscaled = topo_downscaler(np.arange(z_shape[0]),
                                  np.arange(z_shape[1]))

# visualization
fig, axes = twin_image_plot([topo_downscaled, z_upscaled],
                            ['Downscaled Topography to Z contrast size',
                             'Z contrast upscaled to topography size'])
```



Preparing for image registration

We now have a choice to make – whether we want to register two 40 x 40 images: Reduced Topo and the original STS, or register the two 1024 x 1024 images: Original Topo and the interpolated STS, or do both.

If you have the time and resources, doing both is a good idea to get a handle on the quality of the registration. Let's try mapping the larger images in this example

Before getting started, it's a good idea to also think about what data you'd like to use a reference and which data should get transformed. Data that is to remain invariant is referred to as the 'fixed' image and the data to get the transformation is referred to as 'moving.' In our case we want to map the STS data onto the image data, so Topo is 'fixed' and STS is 'moving'

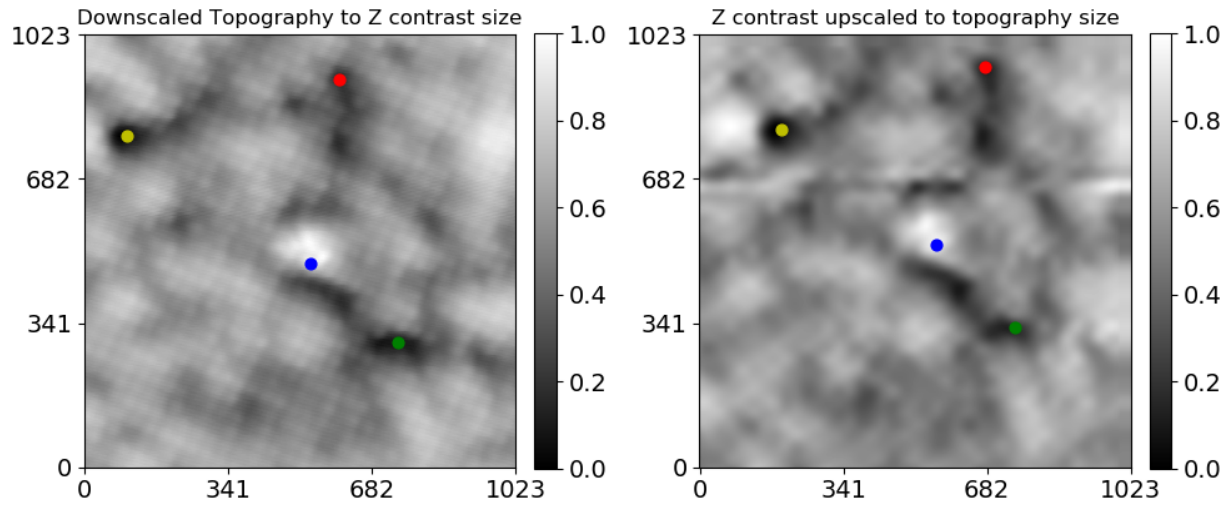
Python does not automatically calculate a transformation matrix for you. This is only possible if your images contain similar features, which is the case here, as this quick registration is intensity based. There are ways to do this in Python using an OpenCV library. Since this library isn't installed natively with Anaconda, I invite you to explore those concepts on your own. You have to specify a transform matrix on your own, which is essentially a coordinate matching problem. This seems harder initially, but for most of the data this is the only way forward because features will not match in different information channels (topography & Raman for example)

We have selected few points quickly here. As the number of points is increased and the coordinates are more carefully selected, the registration quality improves substantially

```
# First normalize the up and downscaled images:
z_upscaled = normalize_image(z_upscaled)
topo_downscaled = normalize_image(topo_downscaled)

# define the topography as the image that is fixed and the upscaled Z contrast image␣
↪as the one that moves
# during the image registration
fixed = high_res_topo
moving = z_upscaled

# Define the points that are common:
src = [(536, 482),
       (100, 785),
       (745, 294),
       (604, 918)]
dst = [(561, 527),
       (193, 800),
       (749, 332),
       (678, 946)]
```

Let's visualize the pointers on the two images

```
# First plot the two images
fig, axes = twin_image_plot([high_res_topo, z_upscaled],
                            ['Downscaled Topography to Z contrast size',
                             'Z contrast upscaled to topography size'],
                            cmap='gray')

# Defining a quick function that plots markers on an image with different colors
def plot_markers(axis, coordinates, colors):
    for clr, point in zip(colors, coordinates):
        axis.scatter(point[0], point[1], color=clr, s=40)

# Now add the markers
pointer_colors = ['b','y', 'g', 'r']
plot_markers(axes[0], src, pointer_colors)
plot_markers(axes[1], dst, pointer_colors)
```



Its also a good idea to look at the overlaid raw data to gauge the difficulty of the transformation prior to starting

```
fig, axis = plt.subplots(figsize=(5, 5))
axis.imshow(fixed, cmap='Reds', alpha=0.8)
axis.imshow(moving, cmap='Blues', alpha=0.8)
axis.set_title('Images overlayed')
```

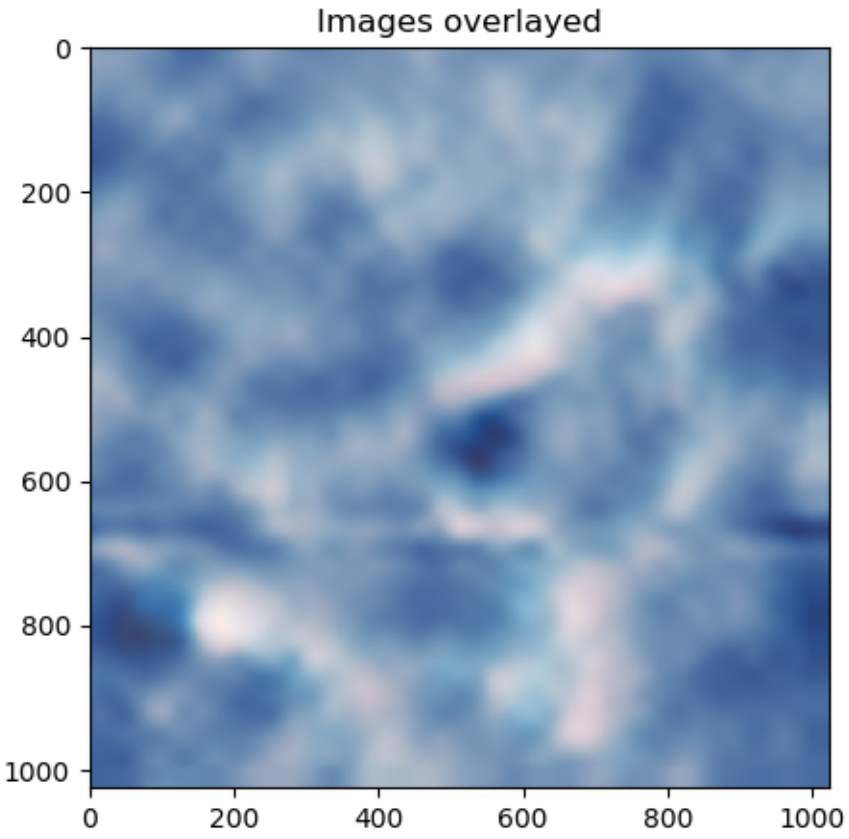


Image Registration

Before starting the registration, we need to provide an optimizer configuration and a metric. OPTIMIZER contains settings used to configure the intensity similarity optimization. METRIC configures the image similarity metric to be used during registration.

The next set of statements populates a transform_cell with different types of registrations available:

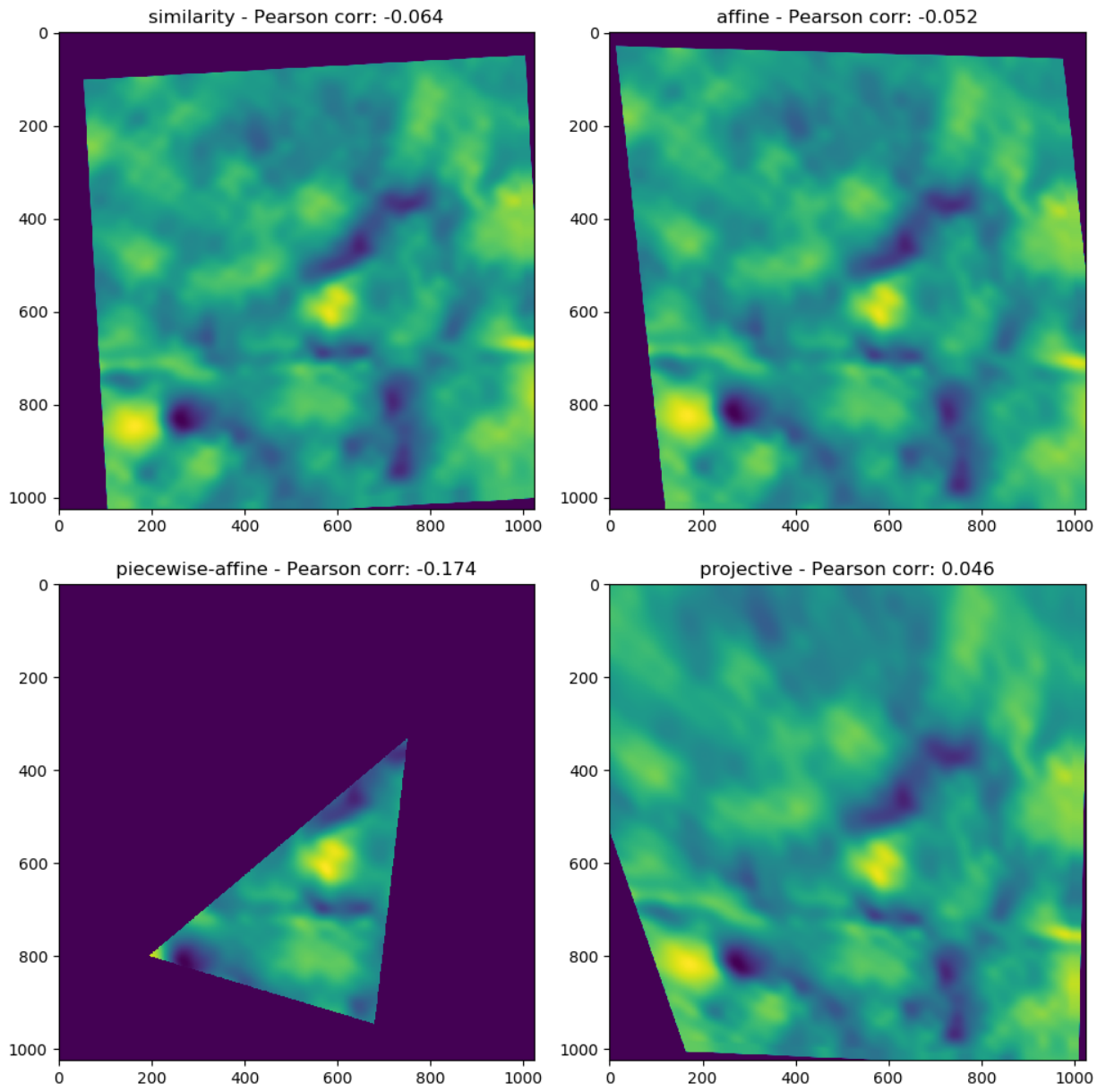
- Translation – translation types of distortion
- Rigid – translation and rotation types of distortion
- Similarity – translation, rotation and scale types of distortion
- Affine – translation, rotation, scale and shear types of distortion

In cases when you aren't sure which transformation is best, and time is permitting. All 4 can be tried.

Lets take a closer look inside the transformation process:

Inside this loop we are registering an image, getting the transform, applying it to the 'moving' data set and inspecting the results. Since we are not sure which transform to pick, we are trying all of them one at a time – hence why this code is inside a 'for' loop We use the Pearson Correlation to look at how well (higher number is better) each of the transforms performed

```
trans_names = ['similarity', 'affine', 'piecewise-affine', 'projective']  # transform␣
↪list

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 10))

for tform_type, axis in zip(trans_names, axes.flat):  # looping through transforms

    # build the model
    tform = transform.estimate_transform(tform_type, np.array(src), np.array(dst))

    # use the model
    raw_corrected_Z = transform.warp(moving, inverse_map=tform.inverse, output_
↪shape=np.shape(moving))

    # one way to do correlations
    corr = stats.pearsonr(np.reshape(fixed, [1024 * 1024, 1]),
                          np.reshape(raw_corrected_Z, [1024 * 1024, 1]))[0][0]

    # visualize the transformation
    axis.set_title(tform_type + ' - Pearson corr: ' + str(np.round(corr, 3)))
    axis.imshow(raw_corrected_Z)

fig.suptitle('Different transforms applied to the images', y=1.03)
fig.tight_layout()
```



delete the h5_file

```
os.remove(h5_path)
```

Total running time of the script: (0 minutes 2.186 seconds)

Note: Click *here* to download the full example code

Spectral Unmixing

Suhas Somnath, Rama K. Vasudevan, Stephen Jesse

- Institute for Functional Imaging of Materials

- Center for Nanophase Materials Sciences

Oak Ridge National Laboratory, Oak Ridge TN 37831, USA

In this notebook we load some spectral data, and perform basic data analysis, including:

- KMeans Clustering
- Non-negative Matrix Factorization
- Principal Component Analysis

Software Prerequisites:

- Standard distribution of **Anaconda** (includes numpy, scipy, matplotlib and sci-kit learn)
- **pycroscopy** : Though pycroscopy is mainly used here for plotting purposes only, it's true capabilities are realized through the ability to seamlessly perform these analyses on any imaging dataset (regardless of origin, size, complexity) and storing the results back into the same dataset among other things

```
# Import packages

# Ensure that this code works on both python 2 and python 3
from __future__ import division, print_function, absolute_import, unicode_literals

# basic numeric computation:
import numpy as np

# The package used for creating and manipulating HDF5 files:
import h5py

# Plotting and visualization:
import matplotlib.pyplot as plt

# for downloading files:
import wget
import os

# multivariate analysis:
from sklearn.cluster import KMeans
from sklearn.decomposition import NMF
import subprocess
import sys

def install(package):
    subprocess.call([sys.executable, "-m", "pip", "install", package])
# Package for downloading online files:
# finally import pycroscopy:
try:
    import pyUSID as usid
except ImportError:
    print('pyUSID not found.  Will install with pip.')
    import pip
    install('pyUSID')
    import pyUSID as usid
try:
```

(continues on next page)

(continued from previous page)

```
    import pycroscopy as px
except ImportError:
    print('pycroscopy not found.  Will install with pip.')
    import pip
    install('pycroscopy')
    import pycroscopy as px
from pycroscopy.viz import cluster_utils
```

The Data

In this example, we will work on a **Band Excitation Piezoresponse Force Microscopy (BE-PFM)** imaging dataset acquired from advanced atomic force microscopes. In this dataset, a spectra was collected for each position in a two dimensional grid of spatial locations. Thus, this is a three dimensional dataset that has been flattened to a two dimensional matrix in accordance with the pycroscopy data format.

Fortunately, all statistical analysis, machine learning, spectral unmixing algorithms, etc. only accept data that is formatted in the same manner of [position x spectra] in a two dimensional matrix.

We will be using an data file available on our GitHub project page by default. You are encouraged to download this document as a Jupyter Notebook (button at the bottom of the page) and use your own data instead. When using your own data, you can skip this cell and provide the path to your data using the variable - data_file_path

```
data_file_path = 'temp_um.h5'
# download the data file from Github:
url = 'https://raw.githubusercontent.com/pycroscopy/pycroscopy/master/data/BELine_
↪0004.h5'
data_file_path = wget.download(url, data_file_path, bar=None)

h5_file = h5py.File(data_file_path, mode='r+')

print('Contents of data file:')
print('----------------------')
usid.hdf_utils.print_tree(h5_file)
print('----------------------')

h5_meas_grp = h5_file['Measurement_000']

# Extracting some basic parameters:
num_rows = usid.hdf_utils.get_attr(h5_meas_grp, 'grid_num_rows')
num_cols = usid.hdf_utils.get_attr(h5_meas_grp, 'grid_num_cols')

# Getting a reference to the main dataset:
h5_main = usid.USIDataset(h5_meas_grp['Channel_000/Raw_Data'])
usid.hdf_utils.write_simple_attrs(h5_main, {'quantity': 'Deflection', 'units': 'V'})

# Extracting the X axis - vector of frequencies
h5_spec_vals = usid.hdf_utils.get_auxiliary_datasets(h5_main, 'Spectroscopic_Values
↪')[-1]
freq_vec = np.squeeze(h5_spec_vals.value) * 1E-3

print('Data currently of shape:', h5_main.shape)

x_label = 'Frequency (kHz)'
y_label = 'Amplitude (a.u.)'
```

Out:

```
Contents of data file:
----------------------
/
├ Measurement_000
  ---------------
  ├ Channel_000
    -----------
    ├ Bin_FFT
    ├ Bin_Frequencies
    ├ Bin_Indices
    ├ Bin_Step
    ├ Bin_Wfm_Type
    ├ Excitation_Waveform
    ├ Noise_Floor
    ├ Position_Indices
    ├ Position_Values
    ├ Raw_Data
    ├ Spatially_Averaged_Plot_Group_000
      --------------------------------
      ├ Bin_Frequencies
      ├ Mean_Spectrogram
      ├ Spectroscopic_Parameter
      ├ Step_Averaged_Response
    ├ Spectroscopic_Indices
    ├ Spectroscopic_Values
    ├ UDVS
    ├ UDVS_Indices
----------------------
Data currently of shape: (16384, 119)
```

1. Singular Value Decomposition (SVD)

SVD is an eigenvector decomposition that is defined statistically, and therefore typically produces non-physical eigenvectors. Consequently, the interpretation of eigenvectors and abundance maps from SVD requires care and caution in interpretation. Nonetheless, it is a good method for quickly visualizing the major trends in the dataset since the resultant eigenvectors are sorted in descending order of variance or importance. Furthermore, SVD is also very well suited for data cleaning through the reconstruction of the dataset using only the first N (most significant) components.

SVD results in three matrices:

- V - Eigenvectors sorted by variance in descending order
- U - corresponding abundance maps
- S - Variance or importance of each of these components

Advantage of pycroscopy:

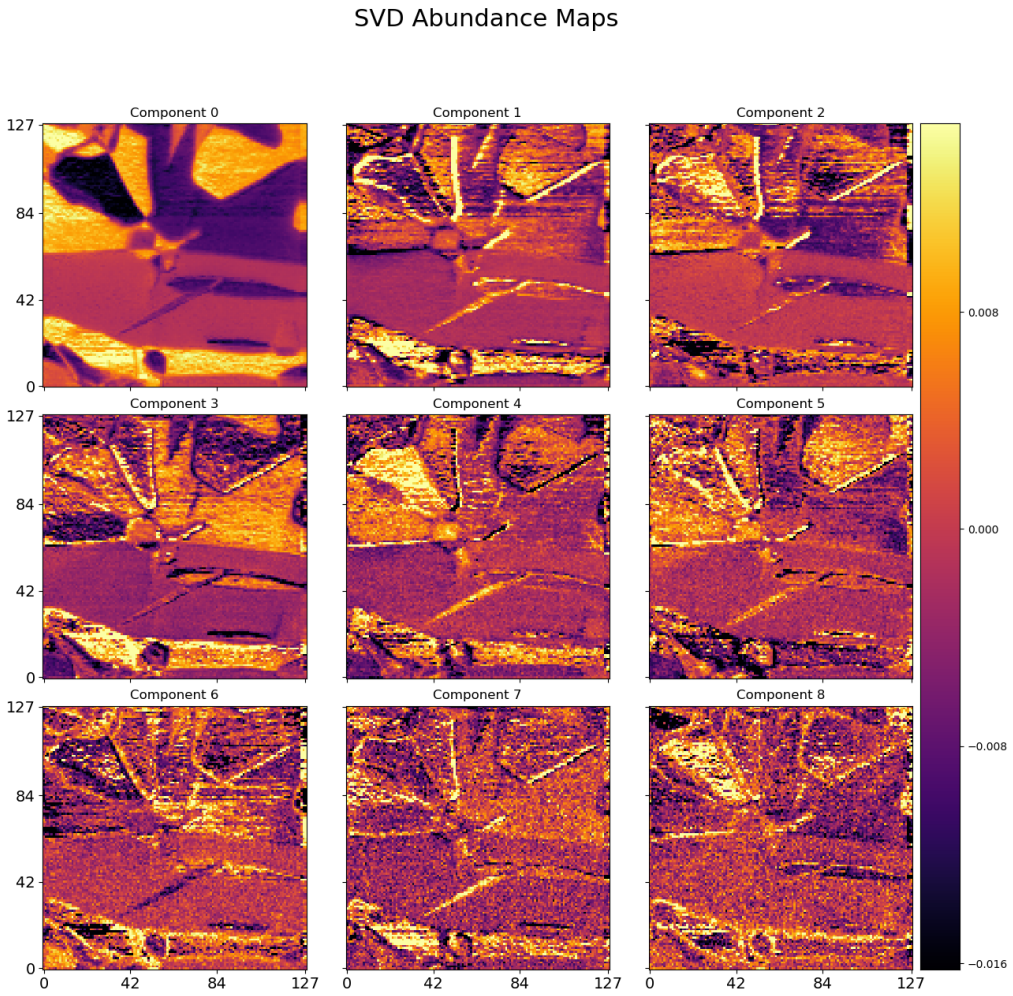
Notice that we are working with a complex valued dataset. Passing the complex values as is to SVD would result in complex valued eigenvectors / endmembers as well as abundance maps. Complex valued abundance maps are not physical. Thus, one would need to restructure the data such that it is real-valued only.

One solution is to stack the real value followed by the magnitude of the imaginary component before passing to SVD. After SVD, the real-valued eigenvectors would need to be treated as the concatenation of the real and imaginary components. So, the eigenvectors would need to be restructured to get back the complex valued eigenvectors.

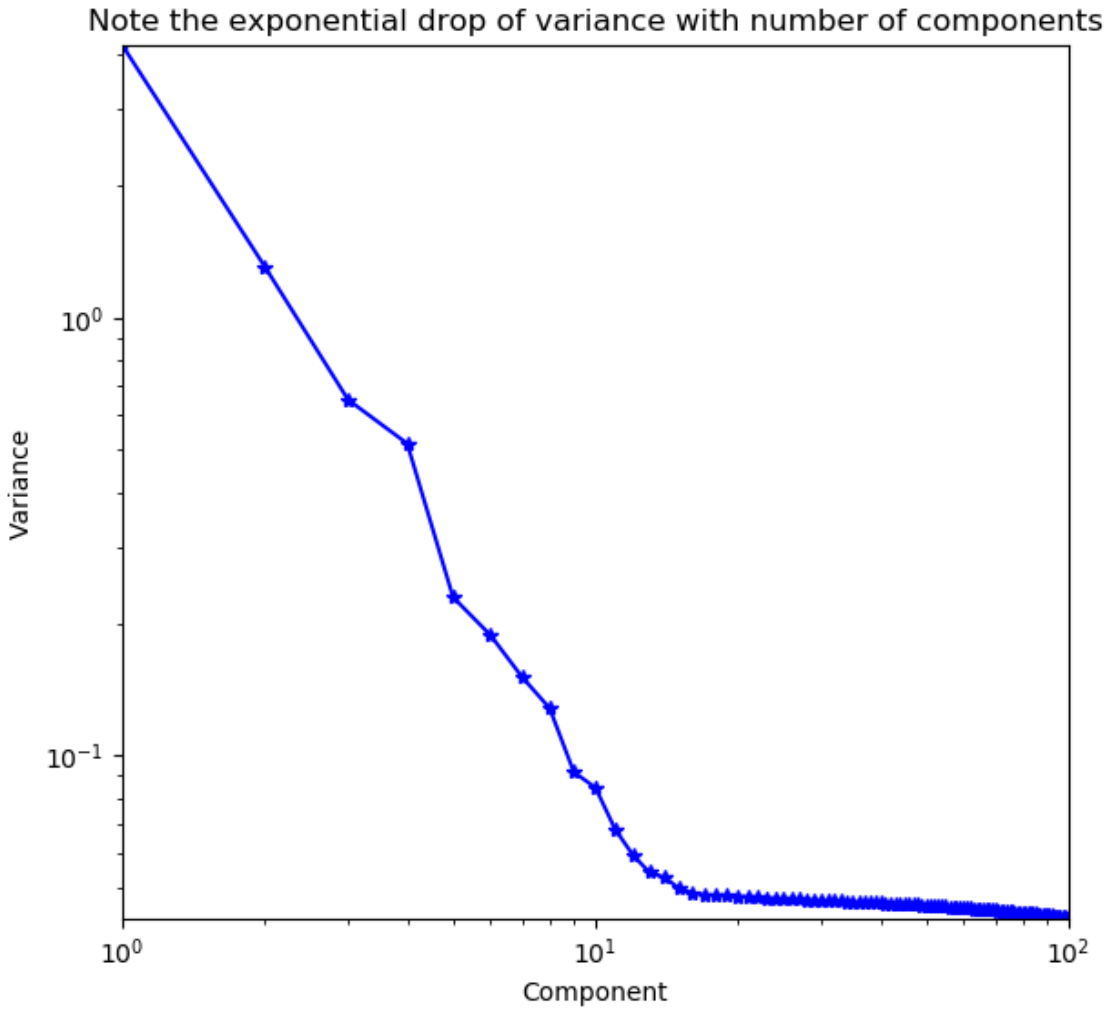
Pycroscopy handles all these data transformations (both for the source dataset and the eigenvectors) automatically. In general, pycroscopy handles compound / complex valued datasets everywhere possible

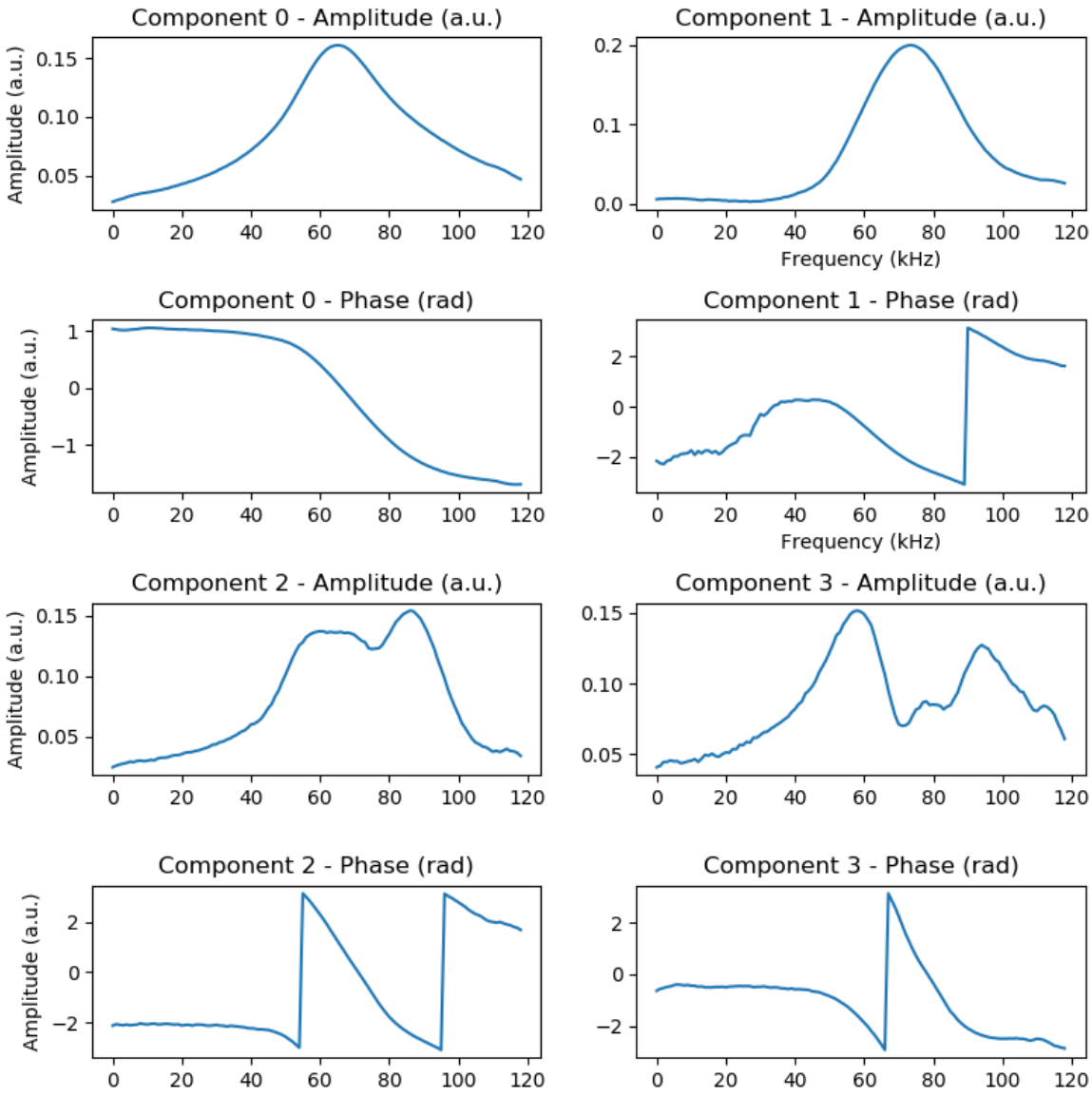
Furthermore, while it is not discussed in this example, pycroscopy also writes back the results from SVD back to the same source h5 file including all relevant links to the source dataset and other ancillary datasets

```
decomposer = px.processing.svd_utils.SVD(h5_main, num_components=100)
h5_svd_group = decomposer.compute()

h5_u = h5_svd_group['U']
h5_v = h5_svd_group['V']
h5_s = h5_svd_group['S']

# Since the two spatial dimensions (x, y) have been collapsed to one, we need to␣
↪reshape the abundance maps:
abun_maps = np.reshape(h5_u[:, :25], (num_rows, num_cols, -1))

usid.plot_utils.plot_map_stack(abun_maps, num_comps=9, title='SVD Abundance Maps',␣
↪reverse_dims=True,
                               color_bar_mode='single', cmap='inferno', title_yoffset=0.
↪95)


# Visualize the variance / statistical importance of each component:
usid.plot_utils.plot_scree(h5_s, title='Note the exponential drop of variance with␣
↪number of components')

# Visualize the eigenvectors:
_ = usid.plot_utils.plot_complex_spectra(h5_v[:9, :], x_label=x_label, y_label=y_
↪label,
                                         title='SVD Eigenvectors', evenly_spaced=False)
```


SVD Abundance Maps
Component 0
Component 1
Component 2
Component 3
Component 4
Component 5
Component 6
Component 7
Component 8
127
84
42
0
0.008
0.000
−0.008
−0.016

•


Note the exponential drop of variance with number of components
Variance
10^0
10^{-1}
10^0
10^1
10^2
Component



Out:

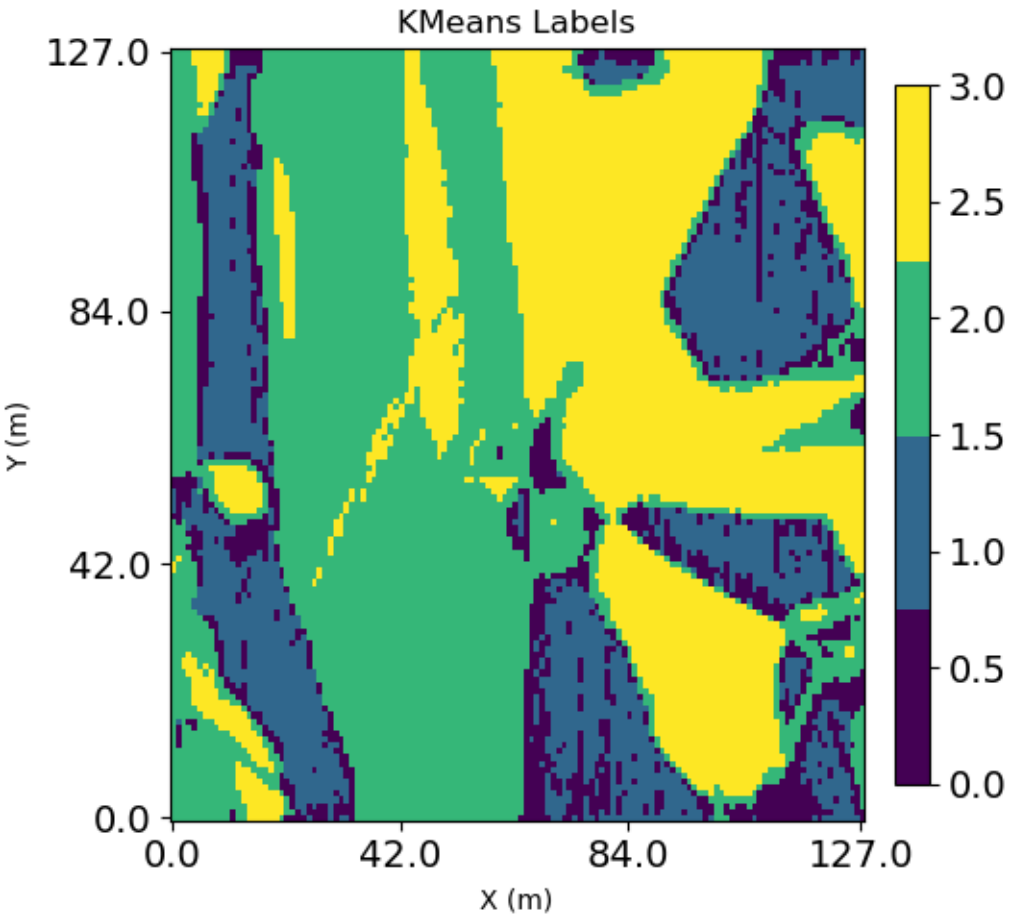
```
Consider calling test() to check results before calling compute() which computes on␣
↪the entire dataset and writes back to the HDF5 file
Took 480.98 msec to compute randomized SVD
```

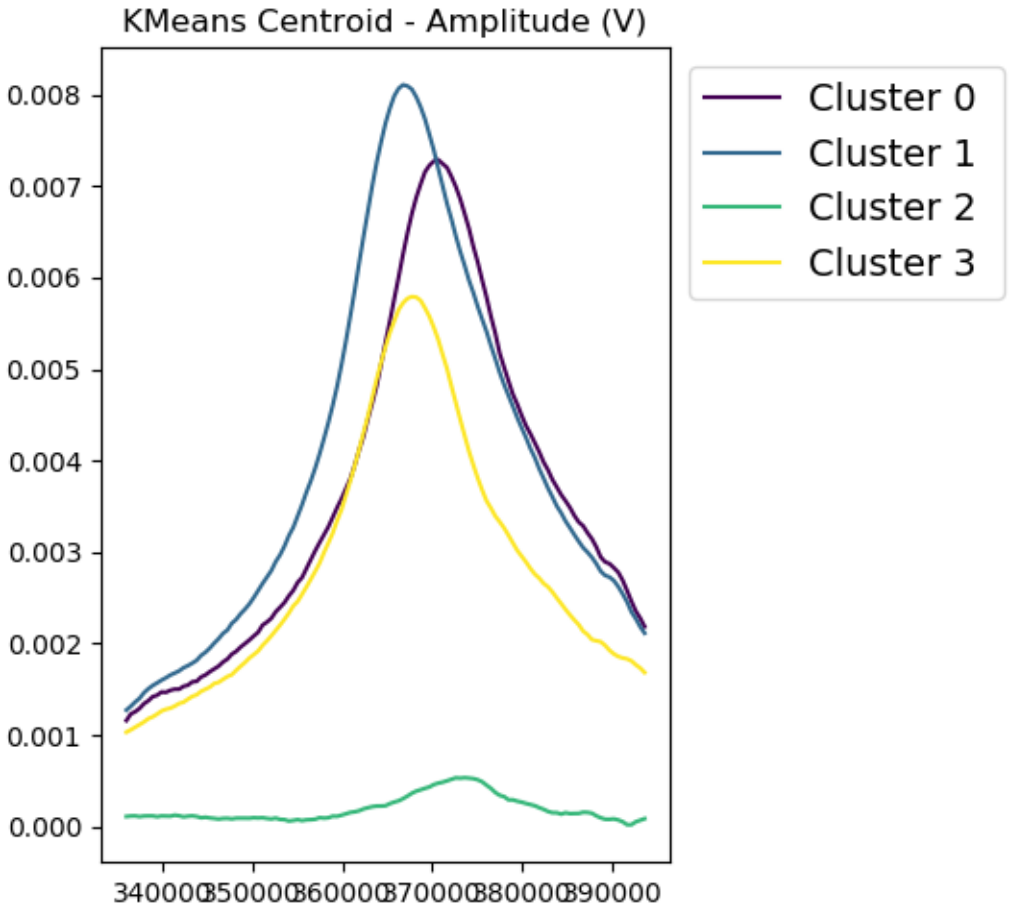
2. KMeans Clustering

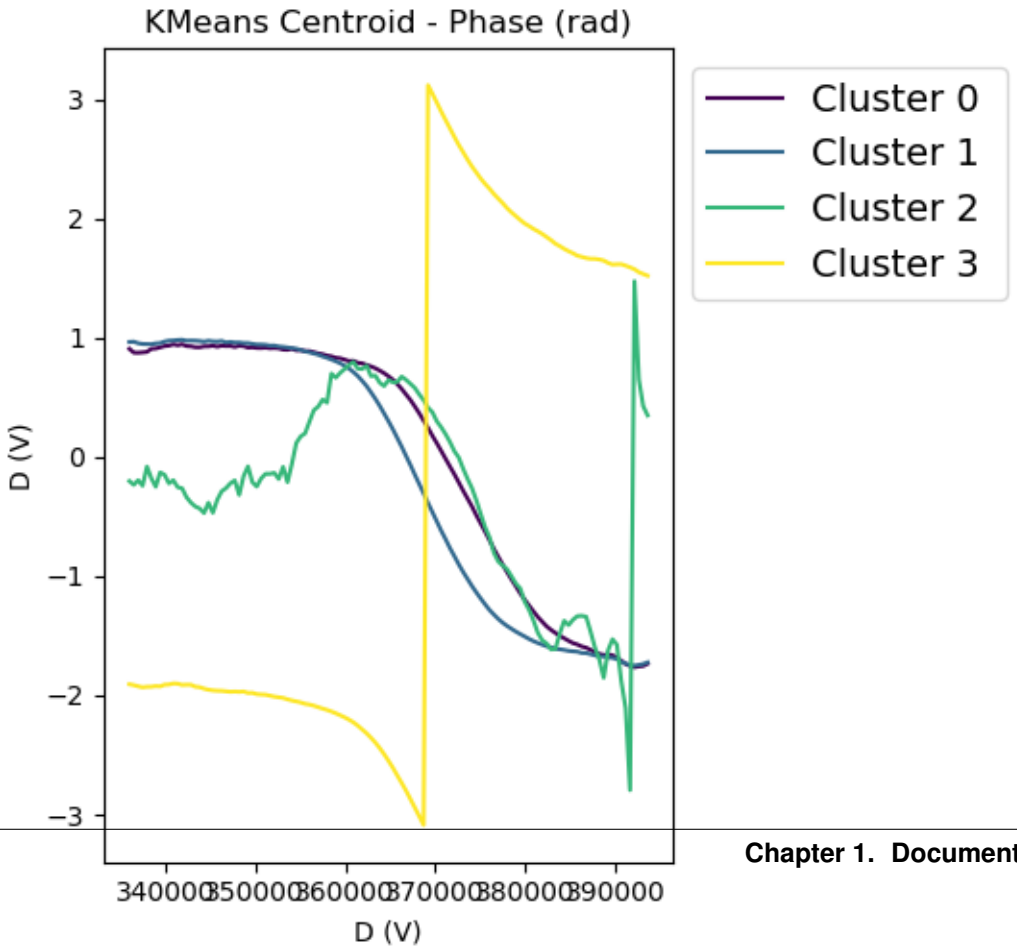
KMeans clustering is a quick and easy method to determine the types of spectral responses present in the data. It is not a decomposition method, but a basic clustering method. The user inputs the number of clusters (sets) to partition the data into. The algorithm proceeds to find the optimal labeling (ie., assignment of each spectra as belonging to the kth set) such that the within-cluster sum of squares is minimized.

Set the number of clusters below

```
num_clusters = 4

estimator = px.processing.Cluster(h5_main, KMeans(n_clusters=num_clusters))
h5_kmeans_grp = estimator.compute(h5_main)
h5_kmeans_labels = h5_kmeans_grp['Labels']
h5_kmeans_mean_resp = h5_kmeans_grp['Mean_Response']

cluster_utils.plot_cluster_h5_group(h5_kmeans_grp)
```




KMeans Centroid - Amplitude (V)
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0.000
340000
350000
360000
370000
380000
390000
Cluster 0
Cluster 1
Cluster 2
Cluster 3

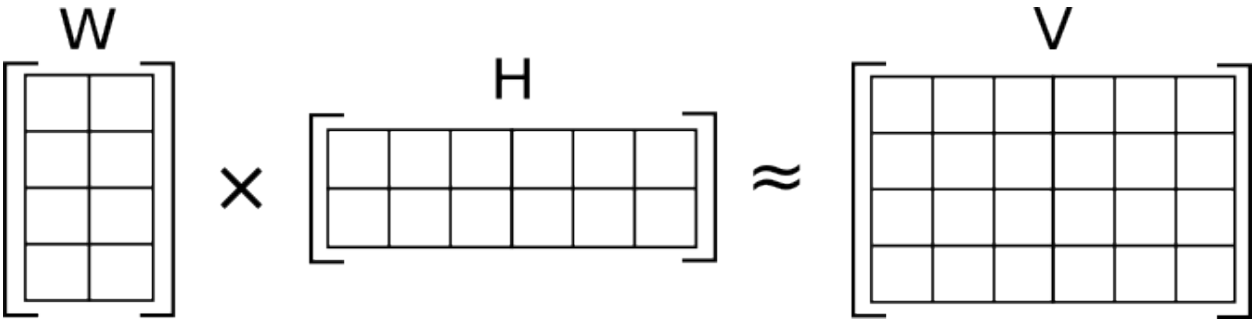

KMeans Centroid - Phase (rad)
3
2
1
0
−1
−2
−3
D (V)
340000
350000
360000
370000
380000
390000
D (V)
Cluster 0
Cluster 1
Cluster 2
Cluster 3

Out:

```
Consider calling test() to check results before calling compute() which computes on␣
↪the entire dataset and writes back to the HDF5 file
Performing clustering on /Measurement_000/Channel_000/Raw_Data.
Took 1.45 sec to compute KMeans
Calculated the Mean Response of each cluster.
Starting computing on 1 cores (requested 1 cores)
Computing serially ...
Took 360.44 msec to calculate mean response per cluster
Writing clustering results to file.
```

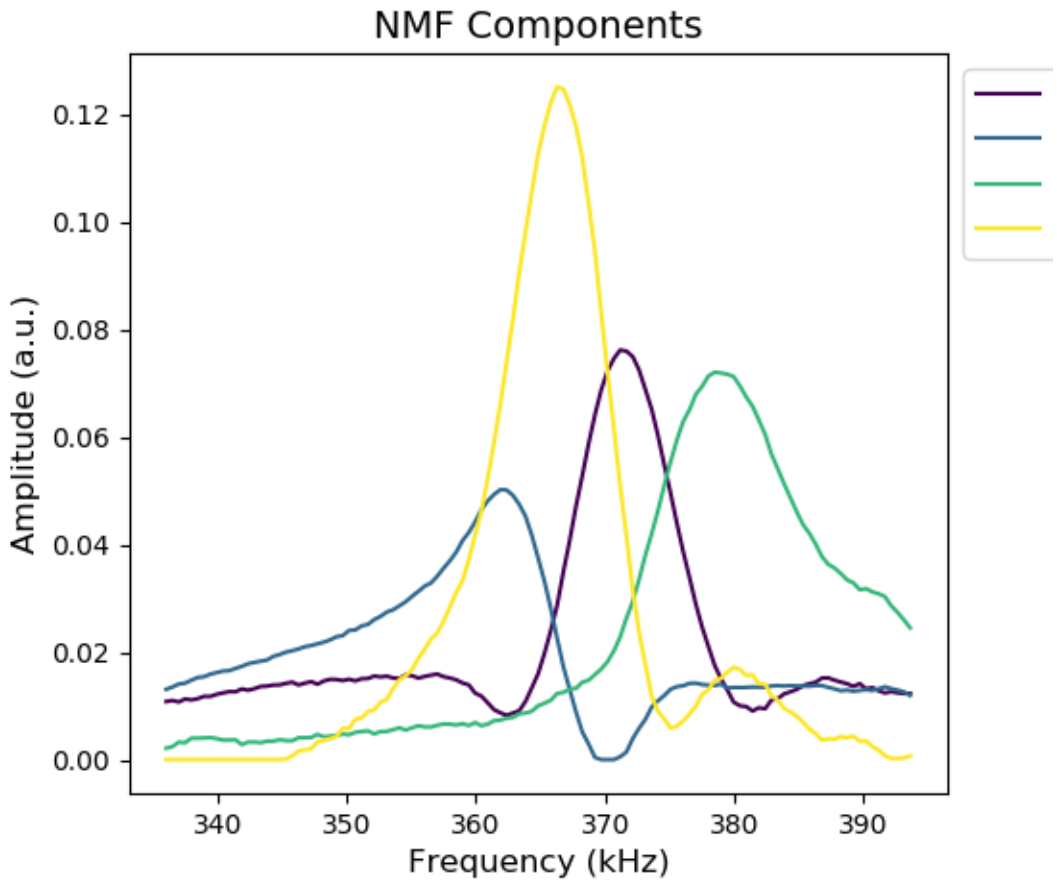
3. Non-negative Matrix Factorization (NMF)

NMF, or non-negative matrix factorization, is a method that is useful towards unmixing of spectral data. It only works on data with positive real values. It operates by approximate determination of factors (matrices) W and H, given a matrix V, as shown below



Unlike SVD and k-Means that can be applied to complex-valued datasets, NMF only works on non-negative datasets. For illustrative purposes, we will only take the amplitude component of the spectral data

```
num_comps = 4

# get the non-negative portion of the dataset
data_mat = np.abs(h5_main)

model = NMF(n_components=num_comps, init='random', random_state=0)
model.fit(data_mat)

fig, axis = plt.subplots(figsize=(5.5, 5))
usid.plot_utils.plot_line_family(axis, freq_vec, model.components_, label_prefix='NMF␣
↪Component #')
axis.set_xlabel(x_label, fontsize=12)
axis.set_ylabel(y_label, fontsize=12)
axis.set_title('NMF Components', fontsize=14)
axis.legend(bbox_to_anchor=[1.0, 1.0], fontsize=12)
```




```
# Close and delete the h5_file
h5_file.close()
os.remove(data_file_path)
```

Total running time of the script: (0 minutes 5.342 seconds)

1.7 Data Translators

- Pycroscopy uses `Translators` to extract data and metadata from files (often measurement data stored in instrument-generated proprietary file formats) and write them into Universal Spectroscopy and Imaging Data (USID) HDF5 files.
- You can write your own `Translator` easily by following this example on our sister project's documentation.
- Below is a list of `Translators` already available in pycroscopy to translate data.
- These translators can be accessed via `pycroscopy.io.translators` or `pycroscopy.translators`
- We tend to add new `Translators` to this list frequently.
  - We understand that this list does not (yet) comprehensively cover all modalities and instrument manufacturers, but we are working towards a comprehensive set of translators.

  - We only have access to a small subset of all available instruments and data which limits our ability to add more translators.
  - Given that this is a **community-driven effort**, you can help by providing:
    * Example datasets (scans, force curves, spectra, force-maps, spectra acquired on grids of locations, etc.)
    * Links to existing packages that have figured out how to extract this data. (More often than not, there are researchers who have put up their code)
    * Your own code for extracting data. We invite you to come onboard and add your tools to the package.
    * Guidance in correctly extracting the metadata (parameters) and data
    * Your time. We are interested in collaborating with you to develop translators.
- We are also interested in collaborating with instrument manufacturers to integrate pycroscopy into instrumentation or analysis software.
- We are working on writing translators to popular open-source software / formats such as `WSxM`, `Gwyddion`, and `ImageJ`.

1.7.1 Generic File Formats

- PNG, TIFF images - `ImageTranslator`
- Movie (frames of) represented by a stack of images - `MovieTranslator`
- numpy data in memory - `NumpyTranslator`
- Gwyddion - `GwyddionTranslator` - work in progress

1.7.2 Scanning Transmission Electron Microscopy (STEM)

- Nion Company - NData - `NDataTranslator`
- One View - `OneViewTranslator`
- Digital Micrograph DM3 and DM4 files - `PtychographyTranslator`
- TIFF image stack for 4D STEM - `PtychographyTranslator`

1.7.3 Scanning Tunnelling Microscopy (STM)

- Omicron STMs - Scanning Tunnelling Spectroscopy - `AscTranslator`
- Nanonis Controllers - `NanonisTranslator`

1.7.4 Atomic Force Microscopy (AFM)

Common formats

- Asylum Research - Igor IBWs for images and force curves - `IgorIBWTranslator`
- Bruker / Veeco / Digital Instruments - images, force curves, force maps - `BrukerAFMTranslator`
- Nanonis Controllers - `NanonisTranslator`

IFIM specific

- Band Excitation (BE):
  - BE-Line and BEPS (Pre 2013) - `BEodfTranslator`
  - BEPS (Post 2013) - `BEPSndfTranslator`
  - BE Relaxation - `BEodfRelaxationTranslator`
  - Time Resolved Kelvin Probe Force Microscopy (trKPFM) - `TRKPFMTranslator`
  - Synthetic BEPS data generator - `FakeBEPSGenerator`
  - Post 2016 Band Excitation data patcher - `LabViewH5Patcher`
- General Mode (G-mode):
  - G-Mode Line - `GLineTranslator`
  - G-Mode Current-Voltage (G-IV) - `GIVTranslator`
  - G-Mode Frequency Tune - `GTuneTranslator`
  - General Dynamic Mode (GDM) - `GDMTranslator`
  - Speedy First Order Reversal Curve (SPORC) - `SporcTranslator`

1.8 Papers / Conferences

1.8.1 Journal Papers using pycroscopy

Note: Notebooks for the papers below are made available through NBViewer. We have been made aware that the **Download** button may not work for certain browsers like Chrome and Safari. *We are working on simplifying the download of the notebooks.*

Current workarounds:

- Try right clicking on the download icon and selecting `Download Linked File`
- Try a different browser like internet explorer

Electron Microscopy

1. Big Data Analytics for Scanning Transmission Electron Microscopy Ptychography by *S. Jesse* et al., **Scientific Reports** (2015)
   - Jupyter notebook
   - Completed notebook
2. Feature extraction via similarity search: application to atom finding and denoising in electron and scanning probe microscopy imaging by *S. Somnath* et al.; **Advanced Structural and Chemical Imaging** (2018);
   - Jupyter notebook
   - Completed notebook

Scanning Probe Microscopy

1. Rapid mapping of polarization switching through complete information acquisition by *S. Somnath* et al., **Nature Communications** (2016)
    - Jupyter notebook
    - Completed notebook
2. Improving superconductivity in BaFe2As2-based crystals by cobalt clustering and electronic uniformity by *L. Li* et al., **Scientific Reports** (2017)
    - Jupyter notebook
3. Direct Imaging of the Relaxation of Individual Ferroelectric Interfaces in a Tensile-Strained Film by *L. Li* et al.; **Advanced Electronic Materials** (2017)
    - Jupyter notebook
    - Completed notebook
4. Decoding apparent ferroelectricity in perovskite nanofibers by *R. Ganeshkumar* et al., **ACS Applied Materials & Interfaces** (2017).
5. Ultrafast Current Imaging via Bayesian Inference by *S. Somnath* et al., **Nature Communications** (2018)
    - Jupyter notebook
    - Completed notebook
    - Raw, intermediate, and final data
6. Reconstructing phase diagrams from local measurements via Gaussian processes: mapping the temperature-composition space to confidence by *D. K. Pradhan* et al., **NPJ Computational Materials** (2018)

Many more coming soon. . . .

Note: Please get in touch with us if you would like to add papers that used pycroscopy / pyUSID

Note: If you used pycroscopy for your research, we would appreciate it if you could reference pycroscopy.

1.8.2 Workshops on pycroscopy

- Aug 13-15 2018 - Full day workshop on Imaging and Spectral Data Analysis in Python at the 2018 CNMS User Meeting, Oak Ridge National Laboratory.
- Aug 5th 2018 - Tutorial at Microscopy and Microanalysis Conference 2018.
- Nov 27 2017 - Tutorial on Machine Learning for Image and Hyperspectral Data
- Aug 9 2017 @ 8:30 - 10:00 AM - Microscopy and Microanalysis conference; X40 - Tutorial session on Large Scale Data Acquisition and Analysis for Materials Imaging and Spectroscopy by S. Jesse and S. V. Kalinin

1.8.3 pycroscopy at International conferences

- May 16-18 2018 - Poster at ORNL Software Expo
- May 18 2018 - **Invited** talk at ImageXD

- Feb 28 2018 - Webinar on Jupyter for Supporting a Materials Imaging User Facility (and beyond). see this Youtube video
- Nov 29 2017 @ 8-10 PM - Poster at the Materials Research Society Fall 2017 Meeting
- Oct 31 2017 @ 6:30 PM - American Vacuum Society conference; Session: SP-TuP1; poster 1641
- Aug 8 2017 @ 10:45 AM - Microscopy and Microanalysis conference - poster.
- Apr 2017 - Lecture on atom finding
- Dec 2016 - Poster + abstract at the 2017 Spring Materials Research Society (MRS) conference

1.9 Frequently asked questions

Contents

1.9.1 My question was still not answered here

Please see a complementary list of FAQ on our sister package - pyUSID.

Pycroscopy philosophy

1.9.2 What is pycroscopy and how is it different from python?

Python is an (interpreted) programming langauge similar to R, Java, C, C++, Fortran etc. Pycroscopy is an addon module to python that provides it the ability to analyze scientific data (especially imaging data). As an (oversimplified) analogy, think of python as Windows or Mac OS and pycroscopy as Firefox or Chrome or Safari.

1.9.3 Is pycroscopy only for the scientific imaging / microscopy communities?

Not at all. We have ensured that the basic data and file formatting paradigm is general enough that it can be extended to any other scientific domain so long as each experiment involves `N` identical measurements of `S` values.

Note that one of the major strengths of pycroscopy is that it can be **science- and instrument-agnostic**. For example, some of our scientific analysis algorithms such as curve-fitting for spectra, image denoising are written in a general enough manner that they can easily find applications in scientific domains beyond imaging and microscopy.

Our data and file format as well as programming framework can easily be extended to or adopted by other scientific domains such as neutron science, nuclear sciences, etc.

We are eager to hear about the many research domains that find our data format and package useful. Please send us an email at pycroscopy@gmail.com

1.9.4 Who uses pycroscopy?

- The Institute for Functional Imaging of Materials (IFIM) at Oak Ridge National Laboratory uses pycroscopy exclusively for in-house research as well as supporting the numerous users who visit IFIM to use their state-of-art scanning probe microscopy techniques.
- Synchrotron Radiation Research at Lund University
- Nuclear Engineering and Health Physics, Idaho State University
- Prof. David Ginger's group at Department of Chemistry, University of Washington
- Idaho National Laboratory
- Central Michigan University
- Iowa State University
- George Western University
- Brown University
- University of Mons
- and many more groups in universities and national labs.
- Please get in touch with us if you would like your group / university to be added here.

1.9.5 How is pycroscopy different from ImageJ, ImageSXM, WSxM, Gwyddion, or xarray?

- **Data sizes and dimensionality**: ImageJ, FIJI, ImageSXM, WSxM, SpectraFox, and OpenFovea are all excellent software packages for dealing with conventional and popular microscopy data such as 2D images or a handful of (simple) spectra that are at best 1- 100 MB in size. We think that pycroscopy is complementary to these other software and packages. Pycroscopy was built from scratch to handle **arbitrarily large datasets** (gigabytes /

terabytes) of datasets which regularly have a large number of position or spectroscopic dimensions (we have had absolutely no problem dealing with datasets with more than 9 unique dimensions - see Band Excitation Polarization Switching + First Order Reversal Curve probing).

- **Data centric**: In addition, Pycroscopy takes a data centric approach aimed towards open science wherein all the data starting from the raw measurement from the instrument, all the way to the final data that is plotted in the resulting scientific publication, are contained in the same file. The processing steps applied to the data are completely transparent and traceable.
- **Data Processing framework**: Most of the aforementioned packages are a collection of several popularly used algorithms applied to data. Again, most of these are applied to small datasets in memory. Perhaps more importantly, the algorithms are tied to a specific scientific domain / application. In contrast, the universal data format used by pycroscopy allows the development of a single version of an algorithm that can be applied to any data.
- **Scalable**: Furthermore, pycroscopy was developed from the ground up to run on laptops while aiming towards compatibility with supercomputers. Nearly all the aforementioned software are applicable to laptops only. Supercomputer / cloud computing scaling in pycroscopy will arrive in the later part of 2018.
- **Flexibility / Customizable**: Like ImageJ / FIJI, it is far easier to add features to pycroscopy when compared to WSxM or Gwyddion
- **User Interface**: Pycroscopy relies on Jupyter notebooks + interactive widgets instead of graphical interface used in most other alternatives.
- **Other complimentary software**:
  - GXSM is another software package that focuses more on the data acquisition from instruments rather than advanced data analysis.
  - xarray has many similar and more advanced features for handling scientific multidimensional data compared to pycroscopy. However, while pycroscopy is a file-based package, xarray enables the features for data in memory only. We see xarray as a package that is complementary to pycroscopy.
- For simple data operations such as flattening, finding maximum, etc. on 2D images or spectra, ImageJ / Gwyddion / WSxM may be better alternatives to pycroscopy.

Using pycroscopy

1.9.6 Why doesn't pycroscopy use graphical user interfaces?

Traditional graphical interfaces are rather time consuming to develop. Instead, we provide jupyter widgets to interact with data wherever possible. Here are some great examples that use jupyter widgets to simplify interaction with the data:

- Band Excitation jupyter notebook developed by The Institute for Functional Imaging of Materials for supporting its users
- Image cleaning and atom finding notebook

1.9.7 What do I do when something is broken?

Often, others may have encountered the same problem and may have brought up a similar issue. Try searching on google and trying out some suggested solutions. If this does not work, raise an `issue` here and one of us will work with you to resolve the problem.

1.9.8 Do I still need to use standard software for plotting figures for papers?

Not at all. Python has an excellent set of libraries for generating even complicated figures for journal papers. Pycroscopy has several functions that make it easier to quickly generate publication-ready figures. There are several publications that have only used pycroscopy and matplotlib to generate figures for papers. If you are still not convinced, you can always export your data to text / csv files and use conventional softwares like Origin Pro.

1.9.9 How can I reference pycroscopy?

Somnath, Suhas, Chris R. Smith, Nouamane Laanait, and Stephen Jesse. Pycroscopy. Computer software. Vers. 0.60.0. Oak Ridge National Laboratory, 01 June 2016. Web. <https://pycroscopy.github.io/pycroscopy/about.html>.

1.9.10 Can Pycroscopy read data files from instrument X?

Pycroscopy has numerous translators that extract the data and metadata (e.g. - instrument / imaging parameters) from some popular file formats and store the information in HDF5 files. You can find a list of available translators here.

Becoming a part of the effort

1.9.11 I would like to help but I don't know programming

Your contributions are very valuable to the imaging and scientific community at large. You can help even if you DON'T know how to program!

- You can spread the word - tell anyone who you think may benefit from using pycroscopy.
- Tell us what you think of our documentation or share your own.
- Let us know what you would like to see in pycroscopy.
- Put us in touch with others working on similar efforts so that we can join forces.
- Guide us in developing data translators

1.9.12 I would like to help and I am OK at programming

Chances are that you are far better at python than you might think! Interesting tidbit - The (first version of the) first module of pycroscopy was written less than a week after we learnt how to write code in python. We weren't great programmers when we began but we would like to think that we have gotten a lot better since then.

You can contribute in numerous ways including but not limited to:

- Writing `translators` to convert data from proprietary formats to the pycroscopy format
- Writing image processing, signal processing code, functional fitting, etc.

Send us an email at pycroscopy@gmail.com or a message on our slack group.

1.9.13 Can you add my code to pycroscopy?

Please see our guidelines for contributing code

1.10 Contact us

- Join our google group to discuss about pycroscopy, ask questions, report bugs, get help, etc.
- Feel free to get in touch with us at pycroscopy@gmail.com
- If you find any bugs or if you want a feature added to pycroscopy, raise an issue. You will need a (free) Github account to do this.
  - Please submit the errors you see as part of your issue along with basic details regarding your computer, operating system, etc.

1.11 Credits

The core pycroscopy team consists of:

- @ssomnath (Suhas Somnath)
- @CompPhysChris (Chris R. Smith)

Universal Spectroscopy and Imaging Data (USID) model conceived by @stephenjesse (Stephen Jesse)

Substantial contributions from many developers including:

- @nlaanait (Numan Laanait)
- @ianton86 (Anton Ievlev)
- @str-eat (Daniel Streater)
- @ealopez (Enrique Alejandro Lopez-Guerra)
- @carlodri (Carlo Dri)
- @ramav87 (Rama K. Vasudevan)
- @ondrejdyck (Ondrej Dyck)
- @nmosto (Nick Mostovych)
- @rajgiriUW (Raj Giridharagopal)
- @donpatrice (Patrik Marschalik)
- @Liambcollins (Liam Collins)
- Arpitha Nagaraj for our logo
- and many more

1.12 Acknowledgements

- Funding agencies:
  - Oak Ridge National Laboratory - Laboratory Director's Research and Development fund
  - U.S. Department of Energy, Office of Science
- People who have advised us:
  - Stefan Van Der Walt (@stefanv)

  - Brett Naul (@bnaul)
- People who (continue to) advertise us:
  - Sergei V. Kalinin from IFIM
- Besides the packages used in pycroscopy, we would like to thank the developers of the following software packages:
  - Anaconda
  - PyCharm
  - GitKraken

1.13 What's New

1.13.1 Jun 28 2018:

Moved `pycroscopy.core` into separate package - pyUSID pyUSID will be the engineering package that supports science-focused packages such as pycroscopy similar to how scipy depends on numpy. All references to pycroscopy.core within the pycroscopy package are now referencing pyusid instead. The current release of pycroscopy imports pyUSID and makes it available as `pycroscopy.core` so that existing imports in user-code do not break. In the next release of pycroscopy, this implicit import will be removed and the following modules would have to be imported directly from `pyUSID`:

- `hdf_utils`
- `write_utils`
- `dtype_utils`
- `io_utils`
- `PycroDataset` - renamed to `USIDataset`
- `Translator`
- `ImageTranslator`
- `NumpyTranslator`
- `Process`
- `parallel_compute()`
- `plot_utils`
- `jupyter_utils`

Thus, imports and usages of such modules as:

```
import pycroscopy as px
px.plot_utils.plot_map(...)
px.hdf_utils.print_tree(h5_file)
px.PycroDataset(h5_dset)
# Other non-core classes:
px.processing.SignalFilter(h5_main, ...)
```

would need to be changed to:

```
# Now import pyUSID along with pycroscopy
import pyUSID as usid
import pycroscopy as px
# functions and classes still work the same way
# just use usid instead of px for anything that was in core (see list above).
usid.plot_utils.plot_map(...)
usid.hdf_utils.print_tree(h5_file)
# The only difference is the renaming of the PycroDataset to USIDataset:
usid.USIDataset(h5_dset)
# Other classes and functions outside .core are addressed just as before:
px.processing.SignalFilter(h5_main, ...)
```

1.13.2 Jun 19 2018:

- Thanks to @Liambcollins for bug-fixes to `GTuneTranslator`

1.13.3 Jun 18 2018:

- Thanks to @Liambcollins for bug-fixes to `GLineTranslator`

1.13.4 Jun 15 2018:

- Thanks to @ramav87 for bug-fixes in BEPS related translators and notebooks

1.13.5 Jun 14 2018:

- Thanks to @ealopez for adding AFM simulations
- Thanks to @nmosto for guides for python novices

1.13.6 Jun 13 2018:

- Thanks to @str-eat for implementing a PycroDataset to csv exporter

1.13.7 Jun 04 2018:

- Thanks to @donpatrice for fixing a UTF8 isue with the `NanonisTranslator`

1.13.8 Jun 01 2018:

- First skeleton `GwyddionTranslator` being worked on by @str-eat
- Added guidelines for contributing code

1.13.9 May 31 2018:

- All `Translators` now use absolute paths
- Improved examples and documentation

1.13.10 May 30 2018:

- Thanks to @carlodri for donating his utility to read Gwyddion Simple File (gsf) reader
- Added `gwyfile` to the requirements of pycroscopy
- `NumpyTranslator` now accepts extra datasets and keyword arguments that will be passed on to `hdf_utils.write_main_dataset()`

1.13.11 May 26 2018:

- Implemented a general function for reading sections of binary files
- First version of the `BrukerTranslator` for translating Bruker Icon and other AFM files

1.13.12 May 03 2018:

- `plot_utils.plot_map()` now accepts the extent or the tick values
- Fixed bug in `hdf_utils.write_reduced_spec_dsets()` and `analysis.BESHOFitter`
- General improvements to the `analysis.Fitter` class
- Documentation updates

1.13.13 May 02 2018:

- Fixed bug in `svd_rebuild()`

1.13.14 May 01 2018:

- Minor corrections to documentation formatting
- `pycroscopy.hdf_utils.get_auxillary_datasets()` renamed to `pycroscopy.hdf_utils.get_auxiliary_datasets()`
- Example on parallel computing rewritten to focus on `pycroscopy.parallel_compute()`
- Added `setUp()` and `tearDown()` to unit testing classes for `hdf_utils` and `PycroDataset`
- Fixed bug in the sorting capability of `pycroscopy.hdf_utils.reshape_to_n_dims()`
- Added logo to website

1.13.15 Apr 29 2018 2:

- Centralized verification of slice dictionary in `pycroscopy.PycroDataset`
- The `slice_dict` kwarg in `pycroscopy.PycroDataset.slice()` now the first required argument
- Lots of minor formatting changes to examples.
- Removed jupyter notebooks from which the examples were generated.

1.13.16 Apr 29 2018 1:

- Fixed errors in broken examples
- Replaced example BE datasets with ones where the central datasets now have `quantity` and `units` attributes to make them `Main datasets`
- Replaced example STS dataset with a zip file which will download a lot faster for examples. Corrected the example on NumpyTranslator

1.13.17 Apr 28 2018 2:

- Fixed unit tests for python 2. assertWarns() only applied to python 3 now
- Added `from future` import statement to all modules in `pycroscopy.core`

1.13.18 Apr 28 2018 1:

(Massive) merge of (skunkworks) branch unity_dev into master‘‘

- Added unit tests for all (feasible) modules in `pycroscopy.core`
- Added examples for every single function or class in `pycroscopy.core` (**10** cookbooks in total!)
- Added a primer to h5py and HDF5
- Added document with instructions on converting unit tests to examples of functions.
- Added web page with links to external tutorials
- Added web page describing contents of package, organization,
- Added web page with FAQs
- Moved a simplified (non ptychography version of) ImageTranslator to `pycroscopy.core`
- Package reconfigured to use `pytests` instead of `Nose`
- Converted last few `assert` statements into descriptive `Errors`
- Legacy HDF writing classes and functions **deprecated now** and will be removed in a future release:
  - `hdf_writer` and `virtual_data` modules moved out of `pycroscopy.core.io` and back into `pycroscopy.io`.
  - Moved functions in `pycroscopy.write_utils` using above deprecated classes into `pycroscopy.io.write_utils`. These functions are also deprecated
  - `pycroscopy.translators.BEODFTranslator`, `pycroscopy.analysis.BESHOFitter`, and `pycroscopy.BELoopFitter`, `pycroscopy.processing.SignalFilter`, `pycroscopy.translators.GIVTranslator`, `pycroscopy.analysis.GIVBayesian`, `pycroscopy.processing.gmode_utils`, etc. now do **not** use deprecated classes as proof that even the most complex classes can easily be transitioned to using functions in `pycroscopy.core.io.hdf_utils` and `pycroscopy.core.io.write_utils`
  - Unit tests for modules in `pycroscopy.core.io` rewritten to not use deprecated functions or classes.
  - Deprecated classes only being used in translators, two analyses modules and two process modules
  - Removed old examples and tutorials, especially on deprecated classes
- Upgrades to `pycroscopy.Process`:

  - `pycroscopy.Process` now has a new function - `test()` that allows much easier in-place testing of processes before applying to the entire dataset
  - *pycroscopy.processing.Cluster*', `pycroscopy.processing.Decomposition`, `pycroscopy.processing.SVD`, `pycroscopy.processing.SignalFilter`, `pycroscopy.processing.GIVBayesian` all implement the new `test()` functionality - return results as correct N-dimensional datasets in expected datatypes
  - `pycroscopy.processing.Cluster`, `pycroscopy.processing.Decomposition` now use a user-configured sklearn objects as inputs instead of creating an estimator object
  - `SVD`, `Cluster`, `Decomposition` now correctly write results as `Main` datasets
- More robust `pycroscopy.gmode_utils` functions
- Updates to `pycroscopy.plot_utils`:
  - `plot_complex_loop_stack` merged into `plot_complex_spectra()`
  - new function that provides best row / column configuration for (identical) subplots: `get_plot_grid_size()`
  - moved clustering related utilities into `pycroscopy.viz.cluster_utils` <– significantly revised many functions in there
  - `plot_map_stack()` accepts x, y labels. `plot_map()` accepts X and Y vectors instead of sizes for more granular control
  - All compound functions now pass kwargs to underlying functions wherever possible
- Updates to `pycroscopy.write_utils`:
  - `pycroscopy.write_utils.AncillaryDescriptor` and `pycroscopy.jupyter_utils.VizDimension` merged and significantly simplified to `pycroscopy.write_utils.Dimension`
  - Swapped all usages of `AncillaryDescriptor` with `Dimension` in the entire package
  - More robust handling of attributes with numpy strings as values
  - Added new functions to simplify building of matrices for ancillary datasets - `build_ind_val_matrices()`
- Updates to `pycroscopy.hdf_utils`:
  - Functions updated to using the new `Dimension` objects
  - Added a few new functions such as `write_book_keeping_attrs()`, `create_indexed_group()`, `create_results_group()`
  - `write_main_dataset()` can now write empty datasets, use different prefixes for ancillary dataset names, etc.
  - Centralized the writing of book-keeping attributes to `write_book_keeping_attrs()`
  - generalized certain functions such as `copy_attributes`, `write_simple_attributes()` so that they can be applied to any HDF5 object
  - `write_main_dataset()` and `create_empty_dataset()` now validate the `dtype` correctly
  - `print_tree()` now prints cleaner versions of the tree, only `Main datasets` if requested.
  - `write_book_keeping_attrs()` now writes the operating system version and pycroscopy version in addition to the timestamp and machine ID
  - Region references functions such as `copy_region_refs()` now more robust
- bug fixes to BE translation, visualization, plotting

1.13.19 Mar 27 2018:

- Small changes to make pycroscopy available on `Conda forge`. Thanks to @carlodri !
- `pycroscopy.translators.NanonisTranslator` added to translate Nanonis data files

1.13.20 Mar 2 2018:

- Fixed decode error in `pycroscopy.translators.IgorTranslator` relevant for new versions of Asylum Research microscope software versions

1.13.21 Mar 3 2018: (on `unity_dev` and not on `master`)

- `pycroscopy.plot_utils.plot_map` now accepts X and Y vectors
- Lots of small bug fixes
- More checks for more robust code in `pycroscopy.core`
- New handy function - `pycroscopy.hdf_utils.get_region()` - directly returns the referenced data as a numpy array
- Added two new examples on `pycroscopy.io_utils` and `pycroscopy.dtype_utils`

1.13.22 Feb 18 2018: (on `unity_dev` and not on `master`)

Massive restructuring and overhaul of code:

- Renamed `pycroscopy.ioHDF` to `pycroscopy.HDFWriter`
- Renamed `pycroscopy.MicroDataset` and *pycroscopy.MicroDataGroup'* to `pycroscopy.VirtualDataset` and `pycroscopy.VirtualGroup`
- Data type manipulation functions moved out of `pycroscopy.io_utils` into `pycroscopy.dtype_utils`
- Moved core foundational / science agnostic / engineering elements of pycroscopy into a new subpackage - `pycroscopy.core`. Rule for move - nothing in `.core` should import anything out of `.core`. This may be spun off as its own package at a later stage if deemed appropriate. Contents of `pycroscopy.core`:
  - `core.io` - `HDFWriter`, `VirtualData`, `hdf_utils`, `write_utils`, `io_utils`, `dtype_utils`, `Translator`, `NumpyTranslator`
  - `core.processing` - `Process`, `parallel_compute()`
  - `core.viz` - `plot_utils`, `jupyter_utils`
- Started adding a lot of type and value checks for safer and more robust file reading/writing. Expect a lot of descriptive Exceptions that will help in identifying problems easier and sooner.
- Implemented modular and standalone functions in `pycroscopy.hdf_utils` that form a (much simpler and more robust) feature-equivalent alternative to `pycroscopy.HDFWriter` + `pyroscopy.VirtualData`. `pycroscopy.HDFWriter` + `pyroscopy.VirtualData` **will be phased out in the near future**.
  - First implementation of what may be one of the most popular and important functions - `pycroscopy.hdf_utils.write_main_dataset()` -
    - Thoroughly checks and validates all inputs. Only if these pass,
    - Writes the a dataset containing the central data

        * Creates / reuses `ancillary datasets`
        * links `Ancillary datasets` to create a `Main dataset`
        * writes `quantity` and `units` attributes - **now mandatory**
        * Also writes any other attributes
    - Other notable functions include `write_simple_attrs()`, `write_region_references`, `write_ind_val_dsets()`
- `pycroscopy.NumpyTranslator` now simply calls `pycroscopy.hdf_utils.write_main_dataset()`
    - `pycroscopy.Translator.simple_write()` removed. Translators can extend `NumpyTranslator` instead.
- Added first batch of unit tests for modules in `pycroscopy.core`.
- More robust `pycroscopy.parallel_compute()` via type checking
- Added a new class called `pycroscopy.AuxillaryDescriptor` to describe Position and spectroscopic dimensions. All major functions like `write_main_dataset()` and `write_ind_val_dsets()` to use this descriptor

1.13.23 Jan 16 2018: (on `unity_dev` and not on `master`)

- `pycroscopy.processing.Cluster` and `pycroscopy.processing.Decomposition` now extend `pycroscopy.Process`
- More robust HDF functions for checking the existence of prior results groups.
- Fixed important bugs for better python2 compatibility (HDF I/O, plotting, etc.)
- More FFT signal filtering functions
- Several bug fixes to `pycroscopy.viz.plot_utils`
- Simplifications to the `image cleaning` and `GIV` notebooks to use the new capabilities of `pycroscopy.processing.SVD`, `pycroscopy.processing.Cluster`

1.13.24 Dec 7 2017:

- New function (`visualize()`) added to `pycroscopy.PycroDataset` to facilitate interactive visualization of data in for any dataset (< 4 dimensions)
- Significantly more customizable plotting functions in `pycroscopy.plot_utils`
- Improved `pycroscopy.Process` that provides the framework for:
    - checking for prior instances of a process run on the same dataset with the same parameters
    - resuming an aborted process / computation
- Reorganized `doSVD()` into a new Process called `pycroscopy.processing.SVD` to take advantage of above advancements.
    - The same changes will be rolled out to `pycroscopy.processing.Cluster` and `pycroscopy.processing.Decomposition` soon

1.13.25 Nov 17 2017:

- Significant improvements and bug fixes to Bayesian Inference for G-mode IV.

1.13.26 Nov 11 2017:

- New robust class for Bayesian Inference on G-mode IV data - `pycroscopy.processing.GIVBayesian`
- Utilities for reading files from Nanois controllers
- New robust class for FFT Signal Filtering on any data including G-mode - `pycroscopy.processing.SignalFilter`
- FFT filtering rewritten and simplified to use objects

1.13.27 Oct 9 2017:

- Added `pycroscopy.PycroDataset` - a class that simplifies handling, reshaping, and interpretation of `Main` datasets.

1.13.28 Sep 6 2017:

- Added `pycroscopy.Process` - New class that provides a framework for data processing in Pycroscopy.

1.13.29 Sep 5 2017:

- Improved the example on parallel computing

1.13.30 Aug 31 2017:

- New plot function - `single_img_cbar_plot()` (now merged into `plot_map()`) for nicer 2D image plots with color-bars.

1.13.31 Aug 29 2017:

- Improvements to Bayesian Inference on G-mode IV data including resistance compensation.

1.14 API Reference

1.14.1 Package Structure

The package structure is simple, with 5 main modules:

1. *io*: Translating from custom & proprietary microscope formats to HDF5.
2. *processing*: Multivariate Statistics, Machine Learning, and Filtering.
3. *analysis*: Model-dependent analysis of image information.
4. *viz*: Visualization and interactive slicing of high-dimensional data by lightweight Qt viewers.

5. *simulation*: atomic force microscopy simulations, etc.

The Pycroscopy package.

Submodules

core	

pycroscopy.core

pycroscopy:

pycroscopy.analysis	Pycroscopy's analysis submodule
pycroscopy.io	Pycroscopy's I/O module
pycroscopy.processing	Pycroscopy's processing module
pycroscopy.viz	Pycroscopy's visualization module

pycroscopy.analysis

Pycroscopy's analysis submodule

Submodules

be_loop_fitter	Created on Thu Aug 25 11:48:53 2016
be_sho_fitter	Created on 7/17/16 10:08 AM @author: Suhas Somnath, Chris R.
fit_methods	Created on 12/15/16 3:44 PM @author: Numan Laanait – nlaanait@gmail.com
fitter	Created on 7/17/16 10:08 AM @author: Numan Laanait, Suhas Somnath, Chris Smith
giv_bayesian	Created on Thu Nov 02 11:48:53 2017
guess_methods	Created on 10/5/16 3:44 PM @author: Numan Laanait – nlaanait@gmail.com
optimize	Created on 12/15/16 10:08 AM @author: Numan Laanait

pycroscopy.analysis.be_loop_fitter

Created on Thu Aug 25 11:48:53 2016

@author: Suhas Somnath, Chris R. Smith, Rama K. Vasudevan

Classes

BELoopFitter(h5_main[, variables, parallel])	Analysis of Band excitation loops using functional fits
LoopOptimize([data, dtype, guess, dtype, ...])	Subclass of Optimize with BE Loop specific changes

class BELoopFitter(*h5_main*, *variables=None*, *parallel=True*)
Analysis of Band excitation loops using functional fits

Parameters

- **h5_main** (*h5py.Dataset instance*) – The dataset over which the analysis will be performed. This dataset should be linked to the spectroscopic indices and values, and position indices and values datasets.
- **variables** (*list(string), Default ['Frequency']*) – Lists of attributes that h5_main should possess so that it may be analyzed by Model.
- **parallel** (*bool, optional*) – Should the parallel implementation of the fitting be used. Default True.

Returns

Return type None

Notes

Quantitative mapping of switching behavior in piezoresponse force microscopy, Stephen Jesse, Ho Nyung Lee, and Sergei V. Kalinin, Review of Scientific Instruments 77, 073702 (2006); doi: http://dx.doi.org/10.1063/1.2214699

do_fit(*processors=None*, *max_mem=None*, *solver_type='least_squares'*, *solver_options=None*, *obj_func=None*, *get_loop_parameters=True*, *h5_guess=None*)
Fit the loops

Parameters

- **processors** (*uint, optional*) – Number of processors to use for computing. Currently this is a serial operation Default None, output of psutil.cpu_count - 2 is used
- **max_mem** (*uint, optional*) – Memory in MB to use for computation Default None, available memory from psutil.virtual_memory is used
- **solver_type** (*str*) – Which solver from scipy.optimize should be used to fit the loops
- **solver_options** (*dict of str*) – Parameters to be passed to the solver defined by *solver_type*
- **obj_func** (*dict of str*) – Dictionary defining the class and method for the loop residual function as well as the parameters to be passed
- **get_loop_parameters** (*bool, optional*) – Should the physical loop parameters be calculated after the guess is done Default True
- **h5_guess** (*h5py.Dataset*) – Existing guess to use as input to fit. Default None

Returns **results** – List of the results returned by the solver

Return type list

do_guess(*max_mem=None*, *processors=None*, *get_loop_parameters=True*)
Compute the loop projections and the initial guess for the loop parameters.

Parameters

- **processors** (*uint, optional*) – Number of processors to use for computing. Currently this is a serial operation and this attribute is ignored. Default None, output of psutil.cpu_count - 2 is used

- **max_mem** (*uint, optional*) – Memory in MB to use for computation Default None, available memory from psutil.virtual_memory is used
- **get_loop_parameters** (*bool, optional*) – Should the physical loop parameters be calculated after the guess is done Default True

Returns **h5_guess** – h5py dataset containing the guess parameters

Return type h5py.Dataset object

static extract_loop_parameters(*h5_loop_fit*, *nuc_threshold=0.03*)
Method to extract a set of physical loop parameters from a dataset of fit parameters

Parameters

- **h5_loop_fit** (*h5py.Dataset*) – Dataset of loop fit parameters
- **nuc_threshold** (*float*) – Nucleation threshold to use in calculation physical parameters

Returns **h5_loop_parm** – Dataset of physical parameters

Return type h5py.Dataset

static shift_vdc(*vdc_vec*)
Rolls the Vdc vector by a quarter cycle

Parameters **vdc_vec** (*1D numpy array*) – DC offset vector

Returns

- **shift_ind** (*int*) – Number of indices by which the vector was rolled
- **vdc_shifted** (*1D numpy array*) – Vdc vector rolled by a quarter cycle

class LoopOptimize(*data=array([]*, *dtype=float64)*, *guess=array([]*, *dtype=float64)*, *parallel=True*)
Subclass of Optimize with BE Loop specific changes

Parameters **data** –

computeFit(*processors=1*, *solver_type='least_squares'*, *solver_options={}*, *obj_func={'class': 'Fit_Methods'*, *'obj_func': 'SHO'*, *'xvals': array([]*, *dtype=float64)}*)

Parameters

- **processors** (*unsigned int*) – Number of logical cores to use for computing
- **solver_type** (*string*) – Optimization solver to use (minimize,least_sq, etc...). For additional info see scipy.optimize
- **solver_options** (*dict()*) – Default: dict() Dictionary of options passed to solver. For additional info see scipy.optimize
- **obj_func** (*dict()*) – Default is 'SHO'. Can be one of ['wavelet_peaks', 'relative_maximum', 'gaussian_processes']. For updated list, run GuessMethods.methods

Returns **results** – unknown

Return type unknown

computeGuess(*processors=1*, *strategy='wavelet_peaks'*, *options={'peak_step': 20*, *'peak_widths': array([10*, *200])}*, ***kwargs*)
Computes the guess function using numerous cores

Parameters

- **processors** (*unsigned int*) – Number of logical cores to use for computing

- **strategy** (*string*) – Default is 'Wavelet_Peaks'. Can be one of ['wavelet_peaks', 'relative_maximum', 'gaussian_processes']. For updated list, run GuessMethods.methods
- **options** (*dict*) – Default: Options for wavelet_peaks{"peaks_widths": np.array([10,200]), "peak_step":20}. Dictionary of options passed to strategy. For more info see GuessMethods documentation.
- **kwargs** –

  processors: int number of processors to use. Default all processors on the system except for 1.

Returns **results** – unknown

Return type unknown

pycroscopy.analysis.be_sho_fitter

Created on 7/17/16 10:08 AM @author: Suhas Somnath, Chris R. Smith, Numan Laanait

Functions

is_reshapable(h5_main[, step_start_inds])	A BE dataset is said to be reshape-able if the number of bins per steps is constant.
reshape_to_n_steps(raw_mat, num_steps)	Reshapes provided data from (positions * step, bin) to (positions, step * bin).
reshape_to_one_step(raw_mat, num_steps)	Reshapes provided data from (pos, step * bin) to (pos * step, bin).

Classes

BESHOfitter(h5_main[, variables])	Analysis of Band excitation spectra with harmonic oscillator responses.

class BESHOfitter(*h5_main*, *variables=None*, ***kwargs*)

Analysis of Band excitation spectra with harmonic oscillator responses.

Parameters

- **h5_main** (*h5py.Dataset instance*) – The dataset over which the analysis will be performed. This dataset should be linked to the spectroscopic indices and values, and position indices and values datasets.
- **variables** (*list(string), Default ['Frequency']*) – Lists of attributes that h5_main should possess so that it may be analyzed by Model.

do_fit(*max_mem=None*, *processors=None*, *solver_type='least_squares'*, *solver_options={'jac': 'cs'}*, *obj_func={'class': 'Fit_Methods', 'obj_func': 'SHO', 'xvals': array([], dtype=float64)}*, *h5_partial_fit=None*, *h5_guess=None*, *override=False*)

Fits the dataset to the SHO function

Parameters

- **max_mem** (*uint, optional*) – Memory in MB to use for computation Default None, available memory from psutil.virtual_memory is used

- **processors** (*int*) – Number of processors the user requests. The minimum of this and self._maxCpus is used. Default None
- **solver_type** (*string*) – Default is 'Wavelet_Peaks'. Can be one of ['wavelet_peaks', 'relative_maximum', 'gaussian_processes']. For updated list, run GuessMethods.methods
- **solver_options** (*dict*) – Dictionary of options passed to strategy. For more info see GuessMethods documentation. Default {"peaks_widths": np.array([10,200])}.
- **obj_func** (*dict*) – Dictionary defining the class and method containing the function to be fit as well as any additional function parameters.
- **h5_partial_fit** (*h5py.group. optional, default = None*) – Datagroup containing (partially computed) fit results. do_fit will resume computation if provided.
- **h5_guess** (*h5py.group. optional, default = None*) – Datagroup containing guess results. do_fit will use this if provided.
- **override** (*bool, optional. default = False*) – By default, will simply return duplicate results to avoid recomputing or resume computation on a group with partial results. Set to True to force fresh computation.

Returns **h5_results** – Dataset with the fit parameters

Return type h5py.Dataset object

do_guess(*max_mem=None*, *processors=None*, *strategy='complex_gaussian'*, *options={'peak_step': 20*, *'peak_widths': array([10*, *200])}*, *h5_partial_guess=None*, *override=False*, ***kwargs*)

Parameters

- **max_mem** (*uint, optional*) – Memory in MB to use for computation Default None, available memory from psutil.virtual_memory is used
- **processors** (*int*) – Number of processors to use during parallel guess Default None, output of psutil.cpu_count - 2 is used
- **strategy** (*string*) – Default is 'Wavelet_Peaks'. Can be one of ['wavelet_peaks', 'relative_maximum', 'gaussian_processes']. For updated list, run GuessMethods.methods
- **options** (*dict*) – Default Options for wavelet_peaks{"peaks_widths": np.array([10,200]), "peak_step":20}. Dictionary of options passed to strategy. For more info see GuessMethods documentation.
- **h5_partial_guess** (*h5py.group. optional, default = None*) – Datagroup containing (partially computed) guess results. do_guess will resume computation if provided.
- **override** (*bool, optional. default = False*) – By default, will simply return duplicate results to avoid recomputing or resume computation on a group with partial results. Set to True to force fresh computation.

Returns **results** – Dataset with the SHO guess parameters

Return type h5py.Dataset object

is_reshapable(*h5_main*, *step_start_inds=None*)

A BE dataset is said to be reshape-able if the number of bins per steps is constant. Even if the dataset contains multiple excitation waveforms (harmonics), We know that the measurement is always at the resonance peak, so the frequency vector should not change.

Parameters

- **h5_main** (*h5py.Dataset object*) – Reference to the main dataset
- **step_start_inds** (*list or 1D array*) – Indices that correspond to the start of each BE pulse / UDVS step

Returns **reshapable** – Whether or not the number of bins per step are constant in this dataset

Return type Boolean

reshape_to_n_steps(*raw_mat*, *num_steps*)

Reshapes provided data from (positions * step, bin) to (positions, step * bin). Use this to restructure data back to its original form after parallel computing

Parameters

- **raw_mat** (*2D numpy array*) – Data organized as (positions * step, bin)
- **num_steps** (*unsigned int*) – Number of spectroscopic steps per pixel (eg - UDVS steps)

Returns **two_d** – Data rearranged as (positions, step * bin)

Return type 2D numpy array

reshape_to_one_step(*raw_mat*, *num_steps*)

Reshapes provided data from (pos, step * bin) to (pos * step, bin). This is useful when unraveling data for parallel processing.

Parameters

- **raw_mat** (*2D numpy array*) – Data organized as (positions, step * bins)
- **num_steps** (*unsigned int*) – Number of spectroscopic steps per pixel (eg - UDVS steps)

Returns **two_d** – Data rearranged as (positions * step, bin)

Return type 2D numpy array

pycroscopy.analysis.fit_methods

Created on 12/15/16 3:44 PM @author: Numan Laanait – nlaanait@gmail.com

Classes

BE_Fit_Methods()	Contains fit methods that are specific to BE data.
Fit_Methods()	This is a container class for the different objective functions used in BE fitting To implement a new guess generation strategy, add it following exactly how it's done below.
forc_iv_fit_methods()	Any fitting methods specific to FORC_IV should go here.

class BE_Fit_Methods

Contains fit methods that are specific to BE data.

static BE_LOOP(*coef_vec*, *data_vec*, *dc_vec*, **args*)

Parameters

- **coef_vec** (*numpy.ndarray*) –
- **data_vec** (*numpy.ndarray*) –
- **dc_vec** (*numpy.ndarray*) – The DC offset vector
- **args** (*list*) –

Returns **fitness** – The 1-r^2 value for the current set of loop coefficients

Return type float

class Fit_Methods

This is a container class for the different objective functions used in BE fitting To implement a new guess generation strategy, add it following exactly how it's done below.

In essence, the guess methods here need to return a callable function that will take a feature vector as the sole input and return the guess parameters. The guess methods here use the keyword arguments to configure the returned function.

static SHO(*guess*, *data_vec*, *freq_vector*, **args*)

Generates the single Harmonic Oscillator response over the given vector

Parameters

- **guess** (*array-like*) – The set of guess parameters to be tested
- **data_vec** (*numpy.ndarray*) – The data vector to compare the current guess against
- **freq_vector** (*numpy.ndarray*) – The frequencies that correspond to each data point in *data_vec*
- **args** (*list or tuple*) – SHO parameters=(Amp,w0,Q,phi,vector). vector: 1D np.array of frequency values. Amp: amplitude. w0: resonant frequency. Q: Quality Factor. phi: Phase. vector:

Returns **SHO_func**

Return type callable function.

class forc_iv_fit_methods

Any fitting methods specific to FORC_IV should go here.

static SHO(*guess*, *data_vec*, *freq_vector*, **args*)

Generates the single Harmonic Oscillator response over the given vector

Parameters

- **guess** (*array-like*) – The set of guess parameters to be tested
- **data_vec** (*numpy.ndarray*) – The data vector to compare the current guess against
- **freq_vector** (*numpy.ndarray*) – The frequencies that correspond to each data point in *data_vec*
- **args** (*list or tuple*) – SHO parameters=(Amp,w0,Q,phi,vector). vector: 1D np.array of frequency values. Amp: amplitude. w0: resonant frequency. Q: Quality Factor. phi: Phase. vector:

Returns **SHO_func**

Return type callable function.

pycroscopy.analysis.fitter

Created on 7/17/16 10:08 AM @author: Numan Laanait, Suhas Somnath, Chris Smith

Classes

Fitter(h5_main[, variables, parallel, verbose])	Encapsulates the typical routines performed during model-dependent analysis of data.

class Fitter(*h5_main, variables=['Frequency'], parallel=True, verbose=False*)

Encapsulates the typical routines performed during model-dependent analysis of data. This abstract class should be extended to cover different types of imaging modalities.

For now, we assume that the guess dataset has not been generated for this dataset but we will relax this requirement after testing the basic components.

Parameters

- **h5_main** (*h5py.Dataset instance*) – The dataset over which the analysis will be performed. This dataset should be linked to the spectroscopic indices and values, and position indices and values datasets.
- **variables** (*list(string), Default ['Frequency']*) – Lists of attributes that h5_main should possess so that it may be analyzed by Model.
- **parallel** (*bool, optional*) – Should the parallel implementation of the fitting be used. Default True
- **verbose** (*bool, optional. default = False*) – Whether or not to print statements that aid in debugging

do_fit(*processors=None, solver_type='least_squares', solver_options=None, obj_func=None, h5_partial_fit=None, h5_guess=None, override=False*)

Generates the fit for the given dataset and writes back to file

Parameters

- **processors** (*int*) – Number of cpu cores the user wishes to run on. The minimum of this and self._maxCpus is used.
- **solver_type** (*str*) – The name of the solver in scipy.optimize to use for the fit
- **solver_options** (*dict*) – Dictionary of parameters to pass to the solver specified by *solver_type*
- **obj_func** (*dict*) – Dictionary defining the class and method containing the function to be fit as well as any additional function parameters.
- **h5_partial_fit** (*h5py.group. optional, default = None*) – Datagroup containing (partially computed) fit results. do_fit will resume computation if provided.
- **h5_guess** (*h5py.group. optional, default = None*) – Datagroup containing guess results. do_fit will use this if provided.
- **override** (*bool, optional. default = False*) – By default, will simply return duplicate results to avoid recomputing or resume computation on a group with partial results. Set to True to force fresh computation.

Returns **h5_results** – Dataset with the fit parameters

Return type h5py.Dataset object

do_guess (*processors=None*, *strategy=None*, *options={}*, *h5_partial_guess=None*, *override=False*)

Parameters

- **strategy** (*string (optional)*) – Default is 'Wavelet_Peaks'. Can be one of ['wavelet_peaks', 'relative_maximum', 'gaussian_processes']. For updated list, run GuessMethods.methods
- **processors** (*int (optional)*) – Number of cores to use for computing. Default = all available - 2 cores
- **options** (*dict*) – Default, options for wavelet_peaks {"peaks_widths": np.array([10,200]), "peak_step":20}. Dictionary of options passed to strategy. For more info see GuessMethods documentation.
- **h5_partial_guess** (*h5py.group. optional, default = None*) – Datagroup containing (partially computed) guess results. do_guess will resume computation if provided.
- **override** (*bool, optional. default = False*) – By default, will simply return duplicate results to avoid recomputing or resume computation on a group with partial results. Set to True to force fresh computation.

Returns **h5_guess** – Dataset containing guesses that can be passed on to do_fit()

Return type h5py.Dataset

pycroscopy.analysis.giv_bayesian

Created on Thu Nov 02 11:48:53 2017

@author: Suhas Somnath

Classes

GIVBayesian(h5_main, ex_freq, gain[, ...])	Applies Bayesian Inference to General Mode IV (G-IV) data to extract the true current

class GIVBayesian (*h5_main*, *ex_freq*, *gain*, *num_x_steps=250*, *r_extra=110*, ***kwargs*)

Applies Bayesian Inference to General Mode IV (G-IV) data to extract the true current

Parameters

- **h5_main** (*h5py.Dataset object*) – Dataset to process
- **ex_freq** (*float*) – Frequency of the excitation waveform
- **gain** (*uint*) – Gain setting on current amplifier (typically 7-9)
- **num_x_steps** (*uint (Optional, default = 250)*) – Number of steps for the inferred results. Note: this may be end up being slightly different from specified.
- **r_extra** (*float (Optional, default = 110 [Ohms])*) – Extra resistance in the RC circuit that will provide correct current and resistance values
- **kwargs** (*dict*) – Other parameters specific to the Process class and nuanced bayesian_inference parameters

compute (*override=False*, **args*, ***kwargs*)
Creates placeholders for the results, applies the inference to the data, and writes the output to the file. Consider calling test() before this function to make sure that the parameters are appropriate.

Parameters

- **override** (*bool, optional. default = False*) – By default, compute will simply return duplicate results to avoid recomputing or resume computation on a group with partial results. Set to True to force fresh computation.
- **args** (*list*) – Not used
- **kwargs** (*dictionary*) – Not used

Returns **h5_results_grp** – Datagroup containing all the results

Return type h5py.Datagroup object

test (*pix_ind=None*, *show_plots=True*)
Tests the inference on a single pixel (randomly chosen unless manually specified) worth of data.

Parameters

- **pix_ind** (*int, optional. default = random*) – Index of the pixel whose data will be used for inference
- **show_plots** (*bool, optional. default = True*) – Whether or not to show plots

Returns

Return type fig, axes

use_partial_computation (*h5_partial_group=None*)
Extracts the necessary parameters from the provided h5 group to resume computation

Parameters **h5_partial_group** (*h5py.Datagroup object*) – Datagroup containing partially computed results

pycroscopy.analysis.guess_methods

Created on 10/5/16 3:44 PM @author: Numan Laanait – nlaanait@gmail.com

Functions

r_square(data_vec, func, *args, **kwargs)	R-square for estimation of the fitting quality Typical result is in the range (0,1), where 1 is the best fitting

Classes

GuessMethods()	This is a container class for the different strategies used to find guesses by which an optimization routine is initialized.

class GuessMethods
This is a container class for the different strategies used to find guesses by which an optimization routine is

initialized. To implement a new guess generation strategy, add it following exactly how it's done below.

In essence, the guess methods here need to return a callable function that will take a feature vector as the sole input and return the guess parameters. The guess methods here use the keyword arguments to configure the returned function.

static absolute_maximum(*vector*)
Finds maximum in 1d-array :param vector: :type vector: numpy.ndarray

Returns **fastpeak**

Return type callable function

static complex_gaussian(*resp_vec*, **args*, ***kwargs*)
Sets up the needed parameters for the analytic approximation for the Gaussian fit of complex data.

Parameters

- **resp_vec** (*numpy.ndarray*) – Data vector to be fit.
- **args** (*numpy arrays.*) –
- **kwargs** (*Passed to SHOEstimateFit()*) –

Returns **sho_guess**

Return type callable function.

static gaussian_processes(**args*, ***kwargs*)
Not yet implemented

Parameters

- **args** –
- **kwargs** –

static relative_maximum(**args*, ***kwargs*)
Not yet implemented

Parameters

- **args** –
- **kwargs** –

static wavelet_peaks(*vector*, **args*, ***kwargs*)
This is the function that will be mapped by multiprocess. This is a wrapper around the scipy function. It uses a parameter - wavelet_widths that is configured outside this function.

Parameters **vector** (*1D numpy array*) – Feature vector containing peaks

Returns **peak_indices** – List of indices of peaks within the prescribed peak widths

Return type list

r_square(*data_vec*, *func*, **args*, ***kwargs*)
R-square for estimation of the fitting quality Typical result is in the range (0,1), where 1 is the best fitting

Parameters

- **data_vec** (*array_like*) – Measured data points
- **func** (*callable function*) – Should return a numpy.ndarray of the same shape as data_vec
- **args** – Parameters to be pased to func

- **kwargs** – Keyword parameters to be pased to func

Returns **r_squared** – The R^2 value for the current data_vec and parameters

Return type float

pycroscopy.analysis.optimize

Created on 12/15/16 10:08 AM @author: Numan Laanait

Functions

targetFuncFit(args, **kwargs)	Needed to create mappable function for multiprocessing :param args: :param kwargs: :return:
targetFuncGuess(args, **kwargs)	This is just creates mappable function for multiprocessing guess :param args: :param kwargs: :return:

Classes

Optimize([data, dtype, guess, dtype, parallel])	In charge of all optimization and computation and is used within the Model Class.

class Optimize(*data=array([], dtype=float64)*, *guess=array([], dtype=float64)*, *parallel=True*)

In charge of all optimization and computation and is used within the Model Class.

Parameters **data** –

computeFit(*processors=1*, *solver_type='least_squares'*, *solver_options={}*, *obj_func={'class': 'Fit_Methods', 'obj_func': 'SHO', 'xvals': array([], dtype=float64)}*)

Parameters

- **processors** (*unsigned int*) – Number of logical cores to use for computing
- **solver_type** (*string*) – Optimization solver to use (minimize,least_sq, etc...). For additional info see scipy.optimize
- **solver_options** (*dict()*) – Default: dict() Dictionary of options passed to solver. For additional info see scipy.optimize
- **obj_func** (*dict()*) – Default is 'SHO'. Can be one of ['wavelet_peaks', 'relative_maximum', 'gaussian_processes']. For updated list, run GuessMethods.methods

Returns **results** – unknown

Return type unknown

computeGuess(*processors=1*, *strategy='wavelet_peaks'*, *options={'peak_step': 20, 'peak_widths': array([10, 200])}*, ***kwargs*)

Computes the guess function using numerous cores

Parameters

- **processors** (*unsigned int*) – Number of logical cores to use for computing
- **strategy** (*string*) – Default is 'Wavelet_Peaks'. Can be one of ['wavelet_peaks', 'relative_maximum', 'gaussian_processes']. For updated list, run GuessMethods.methods

- **options** (*dict*) – Default: Options for wavelet_peaks{"peaks_widths": np.array([10,200]), "peak_step":20}. Dictionary of options passed to strategy. For more info see GuessMethods documentation.
- **kwargs** –

  processors: int number of processors to use. Default all processors on the system except for 1.

Returns **results** – unknown

Return type unknown

targetFuncFit(*args*, ***kwargs*)
Needed to create mappable function for multiprocessing :param args: :param kwargs: :return:

targetFuncGuess(*args*, ***kwargs*)
This is just creates mappable function for multiprocessing guess :param args: :param kwargs: :return:

class GuessMethods
This is a container class for the different strategies used to find guesses by which an optimization routine is initialized. To implement a new guess generation strategy, add it following exactly how it's done below.

In essence, the guess methods here need to return a callable function that will take a feature vector as the sole input and return the guess parameters. The guess methods here use the keyword arguments to configure the returned function.

static absolute_maximum(*vector*)
Finds maximum in 1d-array :param vector: :type vector: numpy.ndarray

Returns **fastpeak**

Return type callable function

static complex_gaussian(*resp_vec*, **args*, ***kwargs*)
Sets up the needed parameters for the analytic approximation for the Gaussian fit of complex data.

Parameters

- **resp_vec** (*numpy.ndarray*) – Data vector to be fit.
- **args** (*numpy arrays.*) –
- **kwargs** (*Passed to SHOEstimateFit()*) –

Returns **sho_guess**

Return type callable function.

static gaussian_processes(**args*, ***kwargs*)
Not yet implemented

Parameters

- **args** –
- **kwargs** –

static relative_maximum(**args*, ***kwargs*)
Not yet implemented

Parameters

- **args** –
- **kwargs** –

static wavelet_peaks(*vector*, **args*, ***kwargs*)
This is the function that will be mapped by multiprocess. This is a wrapper around the scipy function. It uses a parameter - wavelet_widths that is configured outside this function.

Parameters **vector** (*1D numpy array*) – Feature vector containing peaks

Returns **peak_indices** – List of indices of peaks within the prescribed peak widths

Return type list

class Fitter(*h5_main, variables=['Frequency'], parallel=True, verbose=False*)
Encapsulates the typical routines performed during model-dependent analysis of data. This abstract class should be extended to cover different types of imaging modalities.

For now, we assume that the guess dataset has not been generated for this dataset but we will relax this requirement after testing the basic components.

Parameters

- **h5_main** (*h5py.Dataset instance*) – The dataset over which the analysis will be performed. This dataset should be linked to the spectroscopic indices and values, and position indices and values datasets.
- **variables** (*list(string), Default ['Frequency']*) – Lists of attributes that h5_main should possess so that it may be analyzed by Model.
- **parallel** (*bool, optional*) – Should the parallel implementation of the fitting be used. Default True
- **verbose** (*bool, optional. default = False*) – Whether or not to print statements that aid in debugging

do_fit(*processors=None*, *solver_type='least_squares'*, *solver_options=None*, *obj_func=None*, *h5_partial_fit=None*, *h5_guess=None*, *override=False*)
Generates the fit for the given dataset and writes back to file

Parameters

- **processors** (*int*) – Number of cpu cores the user wishes to run on. The minimum of this and self._maxCpus is used.
- **solver_type** (*str*) – The name of the solver in scipy.optimize to use for the fit
- **solver_options** (*dict*) – Dictionary of parameters to pass to the solver specified by *solver_type*
- **obj_func** (*dict*) – Dictionary defining the class and method containing the function to be fit as well as any additional function parameters.
- **h5_partial_fit** (*h5py.group. optional, default = None*) – Datagroup containing (partially computed) fit results. do_fit will resume computation if provided.
- **h5_guess** (*h5py.group. optional, default = None*) – Datagroup containing guess results. do_fit will use this if provided.
- **override** (*bool, optional. default = False*) – By default, will simply return duplicate results to avoid recomputing or resume computation on a group with partial results. Set to True to force fresh computation.

Returns **h5_results** – Dataset with the fit parameters

Return type h5py.Dataset object

do_guess(*processors=None*, *strategy=None*, *options={}*, *h5_partial_guess=None*, *override=False*)

Parameters

- **strategy** (*string (optional)*) – Default is 'Wavelet_Peaks'. Can be one of ['wavelet_peaks', 'relative_maximum', 'gaussian_processes']. For updated list, run GuessMethods.methods
- **processors** (*int (optional)*) – Number of cores to use for computing. Default = all available - 2 cores
- **options** (*dict*) – Default, options for wavelet_peaks {"peaks_widths": np.array([10,200]), "peak_step":20}. Dictionary of options passed to strategy. For more info see GuessMethods documentation.
- **h5_partial_guess** (*h5py.group. optional, default = None*) – Datagroup containing (partially computed) guess results. do_guess will resume computation if provided.
- **override** (*bool, optional. default = False*) – By default, will simply return duplicate results to avoid recomputing or resume computation on a group with partial results. Set to True to force fresh computation.

Returns **h5_guess** – Dataset containing guesses that can be passed on to do_fit()

Return type h5py.Dataset

class BESHOfitter(*h5_main*, *variables=None*, ***kwargs*)

Analysis of Band excitation spectra with harmonic oscillator responses.

Parameters

- **h5_main** (*h5py.Dataset instance*) – The dataset over which the analysis will be performed. This dataset should be linked to the spectroscopic indices and values, and position indices and values datasets.
- **variables** (*list(string), Default ['Frequency']*) – Lists of attributes that h5_main should possess so that it may be analyzed by Model.

do_fit(*max_mem=None*, *processors=None*, *solver_type='least_squares'*, *solver_options={'jac': 'cs'}*, *obj_func={'class': 'Fit_Methods'*, *'obj_func': 'SHO'*, *'xvals': array([]*, *dtype=float64)}*, *h5_partial_fit=None*, *h5_guess=None*, *override=False*)

Fits the dataset to the SHO function

Parameters

- **max_mem** (*uint, optional*) – Memory in MB to use for computation Default None, available memory from psutil.virtual_memory is used
- **processors** (*int*) – Number of processors the user requests. The minimum of this and self._maxCpus is used. Default None
- **solver_type** (*string*) – Default is 'Wavelet_Peaks'. Can be one of ['wavelet_peaks', 'relative_maximum', 'gaussian_processes']. For updated list, run GuessMethods.methods
- **solver_options** (*dict*) – Dictionary of options passed to strategy. For more info see GuessMethods documentation. Default {"peaks_widths": np.array([10,200])}}.
- **obj_func** (*dict*) – Dictionary defining the class and method containing the function to be fit as well as any additional function parameters.
- **h5_partial_fit** (*h5py.group. optional, default = None*) – Datagroup containing (partially computed) fit results. do_fit will resume computation if provided.

- **h5_guess** (*h5py.group. optional, default = None*) – Datagroup containing guess results. do_fit will use this if provided.
- **override** (*bool, optional. default = False*) – By default, will simply return duplicate results to avoid recomputing or resume computation on a group with partial results. Set to True to force fresh computation.

Returns **h5_results** – Dataset with the fit parameters

Return type h5py.Dataset object

do_guess (*max_mem=None*, *processors=None*, *strategy='complex_gaussian'*, *options={'peak_step': 20*, *'peak_widths': array([10*, *200])}*, *h5_partial_guess=None*, *override=False*, ***kwargs*)

Parameters

- **max_mem** (*uint, optional*) – Memory in MB to use for computation Default None, available memory from psutil.virtual_memory is used
- **processors** (*int*) – Number of processors to use during parallel guess Default None, output of psutil.cpu_count - 2 is used
- **strategy** (*string*) – Default is 'Wavelet_Peaks'. Can be one of ['wavelet_peaks', 'relative_maximum', 'gaussian_processes']. For updated list, run GuessMethods.methods
- **options** (*dict*) – Default Options for wavelet_peaks{"peaks_widths": np.array([10,200]), "peak_step":20}. Dictionary of options passed to strategy. For more info see GuessMethods documentation.
- **h5_partial_guess** (*h5py.group. optional, default = None*) – Datagroup containing (partially computed) guess results. do_guess will resume computation if provided.
- **override** (*bool, optional. default = False*) – By default, will simply return duplicate results to avoid recomputing or resume computation on a group with partial results. Set to True to force fresh computation.

Returns **results** – Dataset with the SHO guess parameters

Return type h5py.Dataset object

class BELoopFitter (*h5_main*, *variables=None*, *parallel=True*)

Analysis of Band excitation loops using functional fits

Parameters

- **h5_main** (*h5py.Dataset instance*) – The dataset over which the analysis will be performed. This dataset should be linked to the spectroscopic indices and values, and position indices and values datasets.
- **variables** (*list(string), Default ['Frequency']*) – Lists of attributes that h5_main should possess so that it may be analyzed by Model.
- **parallel** (*bool, optional*) – Should the parallel implementation of the fitting be used. Default True.

Returns

Return type None

Notes

Quantitative mapping of switching behavior in piezoresponse force microscopy, Stephen Jesse, Ho Nyung Lee, and Sergei V. Kalinin, Review of Scientific Instruments 77, 073702 (2006); doi: http://dx.doi.org/10.1063/1.2214699

do_fit (*processors=None*, *max_mem=None*, *solver_type='least_squares'*, *solver_options=None*, *obj_func=None*, *get_loop_parameters=True*, *h5_guess=None*)

Fit the loops

Parameters

- **processors** (*uint, optional*) – Number of processors to use for computing. Currently this is a serial operation Default None, output of psutil.cpu_count - 2 is used
- **max_mem** (*uint, optional*) – Memory in MB to use for computation Default None, available memory from psutil.virtual_memory is used
- **solver_type** (*str*) – Which solver from scipy.optimize should be used to fit the loops
- **solver_options** (*dict of str*) – Parameters to be passed to the solver defined by *solver_type*
- **obj_func** (*dict of str*) – Dictionary defining the class and method for the loop residual function as well as the parameters to be passed
- **get_loop_parameters** (*bool, optional*) – Should the physical loop parameters be calculated after the guess is done Default True
- **h5_guess** (*h5py.Dataset*) – Existing guess to use as input to fit. Default None

Returns **results** – List of the results returned by the solver

Return type list

do_guess (*max_mem=None*, *processors=None*, *get_loop_parameters=True*)

Compute the loop projections and the initial guess for the loop parameters.

Parameters

- **processors** (*uint, optional*) – Number of processors to use for computing. Currently this is a serial operation and this attribute is ignored. Default None, output of psutil.cpu_count - 2 is used
- **max_mem** (*uint, optional*) – Memory in MB to use for computation Default None, available memory from psutil.virtual_memory is used
- **get_loop_parameters** (*bool, optional*) – Should the physical loop parameters be calculated after the guess is done Default True

Returns **h5_guess** – h5py dataset containing the guess parameters

Return type h5py.Dataset object

static extract_loop_parameters (*h5_loop_fit*, *nuc_threshold=0.03*)

Method to extract a set of physical loop parameters from a dataset of fit parameters

Parameters

- **h5_loop_fit** (*h5py.Dataset*) – Dataset of loop fit parameters
- **nuc_threshold** (*float*) – Nucleation threshold to use in calculation physical parameters

Returns **h5_loop_parm** – Dataset of physical parameters

Return type h5py.Dataset

static shift_vdc(*vdc_vec*)
Rolls the Vdc vector by a quarter cycle

Parameters **vdc_vec** (*1D numpy array*) – DC offset vector

Returns

- **shift_ind** (*int*) – Number of indices by which the vector was rolled
- **vdc_shifted** (*1D numpy array*) – Vdc vector rolled by a quarter cycle

class Optimize(*data=array([], dtype=float64), guess=array([], dtype=float64), parallel=True*)
In charge of all optimization and computation and is used within the Model Class.

Parameters **data** –

computeFit(*processors=1, solver_type='least_squares', solver_options={}, obj_func={'class': 'Fit_Methods', 'obj_func': 'SHO', 'xvals': array([], dtype=float64)}*)

Parameters

- **processors** (*unsigned int*) – Number of logical cores to use for computing
- **solver_type** (*string*) – Optimization solver to use (minimize,least_sq, etc...). For additional info see scipy.optimize
- **solver_options** (*dict()*) – Default: dict() Dictionary of options passed to solver. For additional info see scipy.optimize
- **obj_func** (*dict()*) – Default is 'SHO'. Can be one of ['wavelet_peaks', 'relative_maximum', 'gaussian_processes']. For updated list, run GuessMethods.methods

Returns **results** – unknown

Return type unknown

computeGuess(*processors=1, strategy='wavelet_peaks', options={'peak_step': 20, 'peak_widths': array([10, 200])}, **kwargs*)
Computes the guess function using numerous cores

Parameters

- **processors** (*unsigned int*) – Number of logical cores to use for computing
- **strategy** (*string*) – Default is 'Wavelet_Peaks'. Can be one of ['wavelet_peaks', 'relative_maximum', 'gaussian_processes']. For updated list, run GuessMethods.methods
- **options** (*dict*) – Default: Options for wavelet_peaks{"peaks_widths": np.array([10,200]), "peak_step":20}. Dictionary of options passed to strategy. For more info see GuessMethods documentation.
- **kwargs** –

  processors: int number of processors to use. Default all processors on the system except for 1.

Returns **results** – unknown

Return type unknown

class GIVBayesian(*h5_main, ex_freq, gain, num_x_steps=250, r_extra=110, **kwargs*)
Applies Bayesian Inference to General Mode IV (G-IV) data to extract the true current

Parameters

- **h5_main** (*h5py.Dataset object*) – Dataset to process

- **ex_freq** (*float*) – Frequency of the excitation waveform
- **gain** (*uint*) – Gain setting on current amplifier (typically 7-9)
- **num_x_steps** (*uint (Optional, default = 250)*) – Number of steps for the inferred results. Note: this may be end up being slightly different from specified.
- **r_extra** (*float (Optional, default = 110 [Ohms])*) – Extra resistance in the RC circuit that will provide correct current and resistance values
- **kwargs** (*dict*) – Other parameters specific to the Process class and nuanced bayesian_inference parameters

compute(*override=False*, **args*, ***kwargs*)

Creates placeholders for the results, applies the inference to the data, and writes the output to the file. Consider calling test() before this function to make sure that the parameters are appropriate.

Parameters

- **override** (*bool, optional. default = False*) – By default, compute will simply return duplicate results to avoid recomputing or resume computation on a group with partial results. Set to True to force fresh computation.
- **args** (*list*) – Not used
- **kwargs** (*dictionary*) – Not used

Returns **h5_results_grp** – Datagroup containing all the results

Return type h5py.Datagroup object

test(*pix_ind=None*, *show_plots=True*)

Tests the inference on a single pixel (randomly chosen unless manually specified) worth of data.

Parameters

- **pix_ind** (*int, optional. default = random*) – Index of the pixel whose data will be used for inference
- **show_plots** (*bool, optional. default = True*) – Whether or not to show plots

Returns

Return type fig, axes

use_partial_computation(*h5_partial_group=None*)

Extracts the necessary parameters from the provided h5 group to resume computation

Parameters **h5_partial_group** (*h5py.Datagroup object*) – Datagroup containing partially computed results

pycroscopy.io

Pycroscopy's I/O module

Submodules

translators	
write_utils	Created on Tue Nov 3 21:14:25 2015

pycroscopy.io.translators

class BEodfTranslator(**args*, ***kwargs*)

Translates either the Band Excitation (BE) scan or Band Excitation Polarization Switching (BEPS) data format from the old data format(s) to .h5

translate(*file_path*, *show_plots=True*, *save_plots=True*, *do_histogram=False*, *verbose=False*)

Translates .dat data file(s) to a single .h5 file

Parameters

- **file_path** (*String / Unicode*) – Absolute file path for one of the data files. It is assumed that this file is of the OLD data format.
- **show_plots** (*(optional) Boolean*) – Whether or not to show intermediate plots
- **save_plots** (*(optional) Boolean*) – Whether or not to save plots to disk
- **do_histogram** (*(optional) Boolean*) – Whether or not to construct histograms to visualize data quality. Note - this takes a fair amount of time
- **verbose** (*(optional) Boolean*) – Whether or not to print statements

Returns **h5_path** – Absolute path of the resultant .h5 file

Return type String / Unicode

class BEPSndfTranslator(**args*, ***kwargs*)

Translates Band Excitation Polarization Switching (BEPS) datasets from .dat files to .h5

translate(*data_filepath*, *show_plots=True*, *save_plots=True*, *do_histogram=False*, *debug=False*)

The main function that translates the provided file into a .h5 file

Parameters

- **data_filepath** (*String / unicode*) – Absolute path of the data file (.dat)
- **show_plots** (*Boolean (Optional. Default is True)*) – Whether or not to show plots
- **save_plots** (*Boolean (Optional. Default is True)*) – Whether or not to save the generated plots
- **do_histogram** (*Boolean (Optional. Default is False)*) – Whether or not to generate and save 2D histograms of the raw data
- **debug** (*Boolean (Optional. default is false)*) – Whether or not to print log statements

Returns **h5_path** – Absolute path of the generated .h5 file

Return type String / unicode

class BEodfRelaxationTranslator(*max_mem_mb=1024*)

Translates old Relaxation data into the new H5 format. This is for the files generated from the old BEPSDAQ program utilizing two cards simultaneously. At present, this version of the translator only works for Out of field measurements It will not work for in-field. This should be fixed at a later date.

translate(*file_path*, *show_plots=True*, *save_plots=True*, *do_histogram=False*)

Basic method that translates .dat data file(s) to a single .h5 file

Inputs: file_path – Absolute file path for one of the data files. It is assumed that this file is of the OLD data format.

Outputs: Nothing

class GIVTranslator(**args*, ***kwargs*)
Translates G-mode Fast IV datasets from .mat files to .h5

translate(*parm_path*)
The main function that translates the provided file into a .h5 file

Parameters **parm_path** (*string / unicode*) – Absolute file path of the parameters .mat file.

Returns **h5_path** – Absolute path of the translated h5 file

Return type string / unicode

class GLineTranslator(**args*, ***kwargs*)
Translated G-mode line (bigtimedata.dat) files from actual BE line experiments to HDF5

translate(*file_path*)
The main function that translates the provided file into a .h5 file

Parameters **file_path** (*String / unicode*) – Absolute path of any file in the directory

Returns **h5_path** – Absolute path of the h5 file

Return type String / unicode

class GTuneTranslator(**args*, ***kwargs*)
Translates G-mode Tune (bigtimedata.dat) files from actual BE line experiments to HDF5

translate(*file_path*)
The main function that translates the provided file into a .h5 file

Parameters **file_path** (*String / unicode*) – Absolute path of any file in the directory

Returns **h5_path** – Absolute path of the h5 file

Return type String / unicode

class GDMTranslator(*max_mem_mb=1024*, **args*, ***kwargs*)
Translates G-mode w^2 datasets from .mat files to .h5

Parameters **max_mem_mb** (*unsigned integer (Optional. Default = 1024)*) – Maximum system memory (in megabytes) that the translator can use

Returns

Return type Translator object

translate(*parm_path*)
Basic method that translates .mat data files to a single .h5 file

Parameters **parm_path** (*string / unicode*) – Absolute file path of the parameters .mat file.

Returns **h5_path** – Absolute path of the translated h5 file

Return type string / unicode

class PtychographyTranslator(**args*, ***kwargs*)
Translate Pytchography data from a set of images to an HDF5 file

translate(*h5_path*, *image_path*, *bin_factor=None*, *bin_func=<function mean>*, *start_image=0*, *scan_size_x=None*, *scan_size_y=None*, *image_type='.tif'*)
Basic method that adds Ptychography data to existing hdf5 thisfile You must have already done the basic translation with BEodfTranslator

Parameters

- **h5_path** (*str*) – Absolute path to where the HDF5 file should be located
- **image_path** (*str*) – Absolute path to folder holding the image files
- **bin_factor** (*array_like of uint, optional*) – Downsampling factor for each dimension. Default is None.
- **bin_func** (*callable, optional*) – Function which will be called to calculate the return value of each block. Function must implement an axis parameter, i.e. numpy.mean. Ignored if bin_factor is None. Default is numpy.mean.
- **start_image** (*int, optional*) – Integer denoting which image in the file path should be considered the starting point. Default is 0, start with the first image on the list.
- **scan_size_x** (*int, optional*) – Number of Ronchigrams in the x direction. Default is None, value will be determined from the number of images and *scan_size_y* if it is given.
- **scan_size_y** (*int, optional*) – Number of Ronchigrams in the y direction. Default is None, value will be determined from the number of images and *scan_size_x* if it is given.
- **image_type** (*str*) – File extension of images to be read. Default '.tif'

Returns **h5_main** – HDF5 Dataset object that contains the flattened images

Return type h5py.Dataset

class SporcTranslator(*max_mem_mb=1024*, **args*, ***kwargs*)

Translates G-mode SPORC datasets from .mat files to .h5

Parameters **max_mem_mb** (*unsigned integer (Optional. Default = 1024)*) – Maximum system memory (in megabytes) that the translator can use

Returns

Return type Translator object

translate(*parm_path*)

Basic method that translates .mat data files to a single .h5 file

Parameters **parm_path** (*string / unicode*) – Absolute file path of the parameters .mat file.

Returns **h5_path** – Absolute path of the translated h5 file

Return type string / unicode

class MovieTranslator(**args*, ***kwargs*)

Translate Pytchography data from a set of images to an HDF5 file

static downSampRoncVec(*ronch_vec*, *binning_factor*)

Downsample the image by taking the mean over nearby values

Parameters

- **ronch_vec** (*ndarray*) – Image data
- **binning_factor** (*int*) – factor to reduce the size of the image by

Returns **ronc_mat3_mean** – Flattened downsampled image

Return type ndarray

translate(*h5_path*, *image_path*, *bin_factor=None*, *bin_func=<function mean>*, *start_image=0*, *image_type='.tif'*)
Basic method that adds Movie data to existing hdf5 file

Parameters

- **h5_path** (*str*) – Absolute path to where the HDF5 file should be located
- **image_path** (*str*) – Absolute path to folder holding the image files
- **bin_factor** (*array_like of uint, optional*) – Downsampling factor for each dimension. Default is None.
- **bin_func** (*callable, optional*) – Function which will be called to calculate the return value of each block. Function must implement an axis parameter, i.e. numpy.mean. Ignored if bin_factor is None. Default is numpy.mean.
- **start_image** (*int, optional*) – Integer denoting which image in the file path should be considered the starting point. Default is 0, start with the first image on the list.
- **image_type** (*str, optional*) – File extension of images to load. Used to filter out other files in the same directory. Default .tif

Returns **h5_main** – HDF5 Dataset object that contains the flattened images

Return type h5py.Dataset

class IgorIBWTranslator(*max_mem_mb=1024*, **args*, ***kwargs*)
Translates Igor Binary Wave (.ibw) files containing images or force curves to .h5

Parameters **max_mem_mb** (*unsigned integer (Optional. Default = 1024)*) – Maximum system memory (in megabytes) that the translator can use

Returns

Return type Translator object

translate(*file_path*, *verbose=False*, *parm_encoding='utf-8'*)
Translates the provided file to .h5

Parameters

- **file_path** (*String / unicode*) – Absolute path of the .ibw file
- **verbose** (*Boolean (Optional)*) – Whether or not to show print statements for debugging
- **parm_encoding** (*str, optional*) – Codec to be used to decode the bytestrings into Python strings if needed. Default 'utf-8'

Returns **h5_path** – Absolute path of the .h5 file

Return type String / unicode

class OneViewTranslator(**args*, ***kwargs*)
Translate Pytchography data from a set of images to an HDF5 file

crop_ronc(*ronc*)
Crop the input Ronchigram by the specified ammount using the specified method.

Parameters **ronc** (*numpy.array*) – Input image to be cropped.

Returns **cropped_ronc** – Cropped image

Return type numpy.array

static downSampRoncVec(*ronch_vec*, *binning_factor*)
Downsample the image by taking the mean over nearby values

Parameters

- **ronch_vec** (*ndarray*) – Image data
- **binning_factor** (*int*) – factor to reduce the size of the image by

Returns **ronc_mat3_mean** – Flattened downsampled image

Return type ndarray

translate(*h5_path*, *image_path*, *bin_factor=None*, *bin_func=<function mean>*, *start_image=0*, *scan_size_x=None*, *scan_size_y=None*, *crop_ammount=None*, *crop_method='percent'*)
Basic method that adds Ptychography data to existing hdf5 thisfile You must have already done the basic translation with BEodfTranslator

Parameters

- **h5_path** (*str*) – Absolute path to where the HDF5 file should be located
- **image_path** (*str*) – Absolute path to folder holding the image files
- **bin_factor** (*array_like of uint, optional*) – Downsampling factor for each dimension. Default is None.
- **bin_func** (*callable, optional*) – Function which will be called to calculate the return value of each block. Function must implement an axis parameter, i.e. numpy.mean. Ignored if bin_factor is None. Default is numpy.mean.
- **start_image** (*int, optional*) – Integer denoting which image in the file path should be considered the starting point. Default is 0, start with the first image on the list.
- **scan_size_x** (*int, optional*) – Number of Ronchigrams in the x direction. Default is None, value will be determined from the number of images and *scan_size_y* if it is given.
- **scan_size_y** (*int, optional*) – Number of Ronchigrams in the y direction. Default is None, value will be determined from the number of images and *scan_size_x* if it is given.
- **crop_ammount** (*uint or list of uints, optional*) – How much should be cropped from the original image. Can be a single unsigned integer or a list of unsigned integers. A single integer will crop the same ammount from all edges. A list of two integers will crop the x-dimension by the first integer and the y-dimension by the second integer. A list of 4 integers will crop the image as [left, right, top, bottom].
- **crop_method** (*str, optional*) – Which cropping method should be used. How much of the image is removed is determined by the value of *crop_ammount*. 'percent' - A percentage of the image is removed. 'absolute' - The specific number of pixel is removed.

Returns **h5_main** – HDF5 Dataset object that contains the flattened images

Return type h5py.Dataset

class NDataTranslator(**args*, ***kwargs*)
Translate Pytchography data from a set of images to an HDF5 file

crop_ronc(*ronc*)
Crop the input Ronchigram by the specified ammount using the specified method.

Parameters **ronc** (*numpy.array*) – Input image to be cropped.

Returns **cropped_ronc** – Cropped image

Return type numpy.array

static downSampRoncVec(*ronch_vec*, *binning_factor*)
Downsample the image by taking the mean over nearby values

Parameters

- **ronch_vec** (*ndarray*) – Image data
- **binning_factor** (*int*) – factor to reduce the size of the image by

Returns **ronc_mat3_mean** – Flattened downsampled image

Return type ndarray

translate(*h5_path*, *image_path*, *bin_factor=None*, *bin_func=<function mean>*, *start_image=0*, *scan_size_x=None*, *scan_size_y=None*, *crop_ammount=None*, *crop_method='percent'*)
Basic method that adds Ptychography data to existing hdf5 thisfile You must have already done the basic translation with BEodfTranslator

Parameters

- **h5_path** (*str*) – Absolute path to where the HDF5 file should be located
- **image_path** (*str*) – Absolute path to folder holding the image files or the path to a specific file
- **bin_factor** (*array_like of uint, optional*) – Downsampling factor for each dimension. Default is None.
- **bin_func** (*callable, optional*) – Function which will be called to calculate the return value of each block. Function must implement an axis parameter, i.e. numpy.mean. Ignored if bin_factor is None. Default is numpy.mean.
- **start_image** (*int, optional*) – Integer denoting which image in the file path should be considered the starting point. Default is 0, start with the first image on the list.
- **scan_size_x** (*int, optional*) – Number of Ronchigrams in the x direction. Default is None, value will be determined from the number of images and *scan_size_y* if it is given.
- **scan_size_y** (*int, optional*) – Number of Ronchigrams in the y direction. Default is None, value will be determined from the number of images and *scan_size_x* if it is given.
- **crop_ammount** (*uint or list of uints, optional*) – How much should be cropped from the original image. Can be a single unsigned integer or a list of unsigned integers. A single integer will crop the same ammount from all edges. A list of two integers will crop the x-dimension by the first integer and the y-dimension by the second integer. A list of 4 integers will crop the image as [left, right, top, bottom].
- **crop_method** (*str, optional*) – Which cropping method should be used. How much of the image is removed is determined by the value of *crop_ammount*. 'percent' - A percentage of the image is removed. 'absolute' - The specific number of pixel is removed.

Returns **h5_main** – HDF5 Dataset object that contains the flattened images

Return type h5py.Dataset

class FakeBEPSGenerator(**args*, ***kwargs*)

calc_loop_coef_mat(*image_list*)
Build the loop coefficient matrix

Parameters **image_list** (*list of numpy.ndarray*) – Images that will be used to generate the coefficients

Returns **coef_mat** – Array of loop coefficients

Return type numpy.ndarray

translate(*h5_path*, *n_steps=32*, *n_bins=37*, *start_freq=300000.0*, *end_freq=350000.0*, *data_type='BEPSData'*, *mode='DC modulation mode'*, *field_mode='in and out-of-field'*, *n_cycles=1*, *FORC_cycles=1*, *FORC_repeats=1*, *loop_a=1*, *loop_b=4*, *cycle_frac='full'*, *image_folder='/home/challtdow/workspace/pycroscopy/pycroscopy/io/translators/df_utils/beps_data_gen_images'*, *bin_factor=None*, *bin_func=<function mean>*, *image_type='.tif'*, *simple_coefs=False*)

Parameters

- **h5_path** (*str*) – Desired path to write the new HDF5 file
- **n_steps** (*uint, optional*) – Number of voltage steps Default - 32
- **n_bins** (*uint, optional*) – Number of frequency bins Default - 37
- **start_freq** (*float, optional*) – Starting frequency in Hz Default - 300E+3
- **end_freq** (*float, optional*) – Final freqency in Hz Default - 350E+3
- **data_type** (*str, optional*) – Type of data to generate Options - 'BEPSData', 'BELineData' Default - 'BEPSData'
- **mode** (*str, optional*) – Modulation mode to use when generating the data. Options - 'DC modulation mode', 'AC modulation mode' Default - 'DC modulation mode'
- **field_mode** (*str, optional*) – Field mode Options - 'in-field', 'out-of-field', 'in and out-of-field' Default - 'in and out-of-field'
- **n_cycles** (*uint, optional*) – Number of cycles Default - 1
- **FORC_cycles** (*uint, optional*) – Number of FORC cycles Default - 1
- **FORC_repeats** (*uint, optional*) – Number of FORC repeats Default - 1
- **loop_a** (*float, optional*) – Loop coefficient a Default - 1
- **loop_b** (*float, optional*) – Loop coefficient b
- **cycle_frac** (*str*) – Cycle fraction parameter. Default - 'full'
- **image_folder** (*str*) – Path to the images that will be used to generate the loop coefficients. There must be 11 images named '1.tif', '2.tif', ..., '11.tif' Default - pycroscopy.io.translators.df_utils.beps_gen_utils.beps_image_folder
- **bin_factor** (*array_like of uint, optional*) – Downsampling factor for each dimension. Default is None.
- **bin_func** (*callable, optional*) – Function which will be called to calculate the return value of each block. Function must implement an axis parameter, i.e. numpy.mean. Ignored if bin_factor is None. Default is numpy.mean.
- **image_type** (*str*) – File extension of images to be read. Default '.tif'
- **simple_coefs** (*bool*) – Should a simpler coefficient generation be used. Ensures loops, but all loops are identical. Default False

class LabViewH5Patcher
Patches the hdf5 files from the LabView V3 data aquisition software to meet the standards of the Pycroscopy data format.

translate(*h5_path*, *force_patch=False*, ***kwargs*)
Add the needed references and attributes to the h5 file that are not created by the LabView data aquisition program.

Parameters

- **h5_path** (*str*) – path to the h5 file
- **force_patch** (*bool, optional*) – Should the check to see if the file has already been patched be ignored. Default False.

Returns **h5_file** – patched hdf5 file

Return type h5py.File

class TRKPFMTranslator(**args*, ***kwargs*)
Translates trKPFM datasets from .mat and .dat files to .h5

translate(*parm_path*)
The main function that translates the provided file into a .h5 file

Parameters **parm_path** (*string / unicode*) – Absolute file path of the parameters .mat file.

Returns **h5_path** – Absolute path of the translated h5 file

Return type string / unicode

class BrukerAFMTranslator(*max_mem_mb=1024*, **args*, ***kwargs*)

Parameters **max_mem_mb** (*unsigned integer (Optional. Default = 1024)*) – Maximum system memory (in megabytes) that the translator can use

Returns

Return type Translator object

translate(*file_path*, **args*, ***kwargs*)
Translates a given Bruker / Veeco / Nanoscope AFM derived file to HDF5. Currently handles scans, force curves, and force-distance maps

Note that this translator was written with a single example file for each modality and may be buggy.

Parameters **file_path** (*str / unicode*) – path to data file

Returns **h5_path** – path to translated HDF5 file

Return type str / unicode

class ImageTranslator(**args*, ***kwargs*)
Translates data from a set of image files to an HDF5 file. This version has been extended to support dm3 and dm4 files

translate(*image_path*, *h5_path=None*, *bin_factor=None*, *bin_func=<function mean>*, *normalize=False*, ***image_args*)
Basic method that adds Ptychography data to existing hdf5 thisfile You must have already done the basic translation with BEodfTranslator

Parameters

- **image_path** (*str*) – Absolute path to folder holding the image files

- **h5_path** (*str, optional*) – Absolute path to where the HDF5 file should be located. Default is None
- **bin_factor** (*array_like of uint, optional*) – Downsampling factor for each dimension. Default is None.
- **bin_func** (*callable, optional*) – Function which will be called to calculate the return value of each block. Function must implement an axis parameter, i.e. numpy.mean. Ignored if bin_factor is None. Default is numpy.mean.
- **normalize** (*boolean, optional*) – Should the raw image be normalized when read in Default False
- **image_args** (*dict*) – Arguments to be passed to read_image. Arguments depend on the type of image.

Returns **h5_main** – HDF5 Dataset object that contains the flattened images

Return type h5py.Dataset

pycroscopy.io.write_utils

Created on Tue Nov 3 21:14:25 2015

@author: Chris Smith

Functions

build_ind_val_dsets(dimensions[, ...])	Creates VirtualDatasets for the position or spectroscopic indices and values of the data.
build_reduced_spec_dsets(h5_parent_group, ...)	Creates new Spectroscopic Indices and Values datasets from the input datasets and keeps the dimensions specified in keep_dim

build_ind_val_dsets(*dimensions*, *is_spectral=True*, *verbose=False*, *base_name=None*)

Creates VirtualDatasets for the position or spectroscopic indices and values of the data. Remember that the contents of the dataset can be changed if need be after the creation of the datasets. For example if one of the spectroscopic dimensions (e.g. - Bias) was sinusoidal and not linear, The specific dimension in the Spectroscopic_Values dataset can be manually overwritten.

Parameters

- **dimensions** (*Dimension or array-like of Dimension objects*) – Sequence of Dimension objects that provides all necessary instructions for constructing the indices and values datasets
- **is_spectral** (*bool, optional. default = True*) – Spectroscopic (True) or Position (False)
- **verbose** (*Boolean, optional*) – Whether or not to print statements for debugging purposes
- **base_name** (*str / unicode, optional*) – Prefix for the datasets. Default: 'Position_' when is_spectral is False, 'Spectroscopic_' otherwise

Returns

- **ds_inds** (*VirtualDataset*) – Reduced Spectroscopic indices dataset

- **ds_vals** (*VirtualDataset*) – Reduces Spectroscopic values dataset

Notes

steps, *initial_values*, *labels*, and 'units' must be the same length as *dimensions* when they are specified.

Dimensions should be in the order from fastest varying to slowest.

build_reduced_spec_dsets(*h5_parent_group*, *h5_spec_inds*, *h5_spec_vals*, *keep_dim*, *step_starts*, *basename='Spectroscopic'*)

Creates new Spectroscopic Indices and Values datasets from the input datasets and keeps the dimensions specified in keep_dim

Parameters

- **h5_parent_group** (*h5py.Group or h5py.File*) – Group under which the indices and values datasets will be created
- **h5_spec_inds** (*HDF5 Dataset*) – Spectroscopic indices dataset
- **h5_spec_vals** (*HDF5 Dataset*) – Spectroscopic values dataset
- **keep_dim** (*Numpy Array, Boolean*) – Array designating which rows of the input spectroscopic datasets to keep
- **step_starts** (*Numpy Array, Unsigned Integers*) – Array specifying the start of each step in the reduced datasets
- **basename** (*str / unicode*) – String to which '_Indices' and '_Values' will be appended to get the names of the new datasets

Returns

- **ds_inds** (*VirtualDataset*) – Reduced Spectroscopic indices dataset
- **ds_vals** (*VirtualDataset*) – Reduces Spectroscopic values dataset

class BEodfTranslator(**args*, ***kwargs*)

Translates either the Band Excitation (BE) scan or Band Excitation Polarization Switching (BEPS) data format from the old data format(s) to .h5

translate(*file_path*, *show_plots=True*, *save_plots=True*, *do_histogram=False*, *verbose=False*)

Translates .dat data file(s) to a single .h5 file

Parameters

- **file_path** (*String / Unicode*) – Absolute file path for one of the data files. It is assumed that this file is of the OLD data format.
- **show_plots** (*(optional) Boolean*) – Whether or not to show intermediate plots
- **save_plots** (*(optional) Boolean*) – Whether or not to save plots to disk
- **do_histogram** (*(optional) Boolean*) – Whether or not to construct histograms to visualize data quality. Note - this takes a fair amount of time
- **verbose** (*(optional) Boolean*) – Whether or not to print statements

Returns **h5_path** – Absolute path of the resultant .h5 file

Return type String / Unicode

class BEPSndfTranslator(**args*, ***kwargs*)

Translates Band Excitation Polarization Switching (BEPS) datasets from .dat files to .h5

translate(*data_filepath*, *show_plots=True*, *save_plots=True*, *do_histogram=False*, *debug=False*)
The main function that translates the provided file into a .h5 file

Parameters

- **data_filepath** (*String / unicode*) – Absolute path of the data file (.dat)
- **show_plots** (*Boolean (Optional. Default is True)*) – Whether or not to show plots
- **save_plots** (*Boolean (Optional. Default is True)*) – Whether or not to save the generated plots
- **do_histogram** (*Boolean (Optional. Default is False)*) – Whether or not to generate and save 2D histograms of the raw data
- **debug** (*Boolean (Optional. default is false)*) – Whether or not to print log statements

Returns **h5_path** – Absolute path of the generated .h5 file

Return type String / unicode

class BEodfRelaxationTranslator(*max_mem_mb=1024*)
Translates old Relaxation data into the new H5 format. This is for the files generated from the old BEPSDAQ program utilizing two cards simultaneously. At present, this version of the translator only works for Out of field measurements It will not work for in-field. This should be fixed at a later date.

translate(*file_path*, *show_plots=True*, *save_plots=True*, *do_histogram=False*)
Basic method that translates .dat data file(s) to a single .h5 file

Inputs: file_path – Absolute file path for one of the data files. It is assumed that this file is of the OLD data format.

Outputs: Nothing

class GIVTranslator(**args*, ***kwargs*)
Translates G-mode Fast IV datasets from .mat files to .h5

translate(*parm_path*)
The main function that translates the provided file into a .h5 file

Parameters **parm_path** (*string / unicode*) – Absolute file path of the parameters .mat file.

Returns **h5_path** – Absolute path of the translated h5 file

Return type string / unicode

class GLineTranslator(**args*, ***kwargs*)
Translated G-mode line (bigtimedata.dat) files from actual BE line experiments to HDF5

translate(*file_path*)
The main function that translates the provided file into a .h5 file

Parameters **file_path** (*String / unicode*) – Absolute path of any file in the directory

Returns **h5_path** – Absolute path of the h5 file

Return type String / unicode

class GTuneTranslator(**args*, ***kwargs*)
Translates G-mode Tune (bigtimedata.dat) files from actual BE line experiments to HDF5

translate(*file_path*)
The main function that translates the provided file into a .h5 file

Parameters **file_path** (*String / unicode*) – Absolute path of any file in the directory

Returns **h5_path** – Absolute path of the h5 file

Return type String / unicode

class GDMTranslator(*max_mem_mb=1024*, **args*, ***kwargs*)

Translates G-mode w^2 datasets from .mat files to .h5

Parameters **max_mem_mb** (*unsigned integer (Optional. Default = 1024)*) – Maximum system memory (in megabytes) that the translator can use

Returns

Return type Translator object

translate(*parm_path*)

Basic method that translates .mat data files to a single .h5 file

Parameters **parm_path** (*string / unicode*) – Absolute file path of the parameters .mat file.

Returns **h5_path** – Absolute path of the translated h5 file

Return type string / unicode

class PtychographyTranslator(**args*, ***kwargs*)

Translate Pytchography data from a set of images to an HDF5 file

translate(*h5_path*, *image_path*, *bin_factor=None*, *bin_func=<function mean>*, *start_image=0*, *scan_size_x=None*, *scan_size_y=None*, *image_type='.tif'*)

Basic method that adds Ptychography data to existing hdf5 thisfile You must have already done the basic translation with BEodfTranslator

Parameters

- **h5_path** (*str*) – Absolute path to where the HDF5 file should be located
- **image_path** (*str*) – Absolute path to folder holding the image files
- **bin_factor** (*array_like of uint, optional*) – Downsampling factor for each dimension. Default is None.
- **bin_func** (*callable, optional*) – Function which will be called to calculate the return value of each block. Function must implement an axis parameter, i.e. numpy.mean. Ignored if bin_factor is None. Default is numpy.mean.
- **start_image** (*int, optional*) – Integer denoting which image in the file path should be considered the starting point. Default is 0, start with the first image on the list.
- **scan_size_x** (*int, optional*) – Number of Ronchigrams in the x direction. Default is None, value will be determined from the number of images and *scan_size_y* if it is given.
- **scan_size_y** (*int, optional*) – Number of Ronchigrams in the y direction. Default is None, value will be determined from the number of images and *scan_size_x* if it is given.
- **image_type** (*str*) – File extension of images to be read. Default '.tif'

Returns **h5_main** – HDF5 Dataset object that contains the flattened images

Return type h5py.Dataset

class SporcTranslator(*max_mem_mb=1024*, **args*, ***kwargs*)
Translates G-mode SPORC datasets from .mat files to .h5

Parameters **max_mem_mb** (*unsigned integer (Optional. Default = 1024)*) – Maximum system memory (in megabytes) that the translator can use

Returns

Return type Translator object

translate(*parm_path*)
Basic method that translates .mat data files to a single .h5 file

Parameters **parm_path** (*string / unicode*) – Absolute file path of the parameters .mat file.

Returns **h5_path** – Absolute path of the translated h5 file

Return type string / unicode

class MovieTranslator(**args*, ***kwargs*)
Translate Pytchography data from a set of images to an HDF5 file

static downSampRoncVec(*ronch_vec*, *binning_factor*)
Downsample the image by taking the mean over nearby values

Parameters

- **ronch_vec** (*ndarray*) – Image data
- **binning_factor** (*int*) – factor to reduce the size of the image by

Returns **ronc_mat3_mean** – Flattened downsampled image

Return type ndarray

translate(*h5_path*, *image_path*, *bin_factor=None*, *bin_func=<function mean>*, *start_image=0*, *image_type='.tif'*)
Basic method that adds Movie data to existing hdf5 file

Parameters

- **h5_path** (*str*) – Absolute path to where the HDF5 file should be located
- **image_path** (*str*) – Absolute path to folder holding the image files
- **bin_factor** (*array_like of uint, optional*) – Downsampling factor for each dimension. Default is None.
- **bin_func** (*callable, optional*) – Function which will be called to calculate the return value of each block. Function must implement an axis parameter, i.e. numpy.mean. Ignored if bin_factor is None. Default is numpy.mean.
- **start_image** (*int, optional*) – Integer denoting which image in the file path should be considered the starting point. Default is 0, start with the first image on the list.
- **image_type** (*str, optional*) – File extension of images to load. Used to filter out other files in the same directory. Default .tif

Returns **h5_main** – HDF5 Dataset object that contains the flattened images

Return type h5py.Dataset

class IgorIBWTranslator(*max_mem_mb=1024*, **args*, ***kwargs*)
Translates Igor Binary Wave (.ibw) files containing images or force curves to .h5

Parameters **max_mem_mb** (*unsigned integer (Optional. Default = 1024)*) – Maximum system memory (in megabytes) that the translator can use

Returns

Return type Translator object

translate(*file_path*, *verbose=False*, *parm_encoding='utf-8'*)
Translates the provided file to .h5

Parameters

- **file_path** (*String / unicode*) – Absolute path of the .ibw file
- **verbose** (*Boolean (Optional)*) – Whether or not to show print statements for debugging
- **parm_encoding** (*str, optional*) – Codec to be used to decode the bytestrings into Python strings if needed. Default 'utf-8'

Returns **h5_path** – Absolute path of the .h5 file

Return type String / unicode

class OneViewTranslator(**args*, ***kwargs*)
Translate Pytchography data from a set of images to an HDF5 file

crop_ronc(*ronc*)
Crop the input Ronchigram by the specified ammount using the specified method.

Parameters **ronc** (*numpy.array*) – Input image to be cropped.

Returns **cropped_ronc** – Cropped image

Return type numpy.array

static downSampRoncVec(*ronch_vec*, *binning_factor*)
Downsample the image by taking the mean over nearby values

Parameters

- **ronch_vec** (*ndarray*) – Image data
- **binning_factor** (*int*) – factor to reduce the size of the image by

Returns **ronc_mat3_mean** – Flattened downsampled image

Return type ndarray

translate(*h5_path*, *image_path*, *bin_factor=None*, *bin_func=<function mean>*, *start_image=0*, *scan_size_x=None*, *scan_size_y=None*, *crop_ammount=None*, *crop_method='percent'*)
Basic method that adds Ptychography data to existing hdf5 thisfile You must have already done the basic translation with BEodfTranslator

Parameters

- **h5_path** (*str*) – Absolute path to where the HDF5 file should be located
- **image_path** (*str*) – Absolute path to folder holding the image files
- **bin_factor** (*array_like of uint, optional*) – Downsampling factor for each dimension. Default is None.
- **bin_func** (*callable, optional*) – Function which will be called to calculate the return value of each block. Function must implement an axis parameter, i.e. numpy.mean. Ignored if bin_factor is None. Default is numpy.mean.

- **start_image** (*int, optional*) – Integer denoting which image in the file path should be considered the starting point. Default is 0, start with the first image on the list.
- **scan_size_x** (*int, optional*) – Number of Ronchigrams in the x direction. Default is None, value will be determined from the number of images and *scan_size_y* if it is given.
- **scan_size_y** (*int, optional*) – Number of Ronchigrams in the y direction. Default is None, value will be determined from the number of images and *scan_size_x* if it is given.
- **crop_ammount** (*uint or list of uints, optional*) – How much should be cropped from the original image. Can be a single unsigned integer or a list of unsigned integers. A single integer will crop the same ammount from all edges. A list of two integers will crop the x-dimension by the first integer and the y-dimension by the second integer. A list of 4 integers will crop the image as [left, right, top, bottom].
- **crop_method** (*str, optional*) – Which cropping method should be used. How much of the image is removed is determined by the value of *crop_ammount*. 'percent' - A percentage of the image is removed. 'absolute' - The specific number of pixel is removed.

Returns **h5_main** – HDF5 Dataset object that contains the flattened images

Return type h5py.Dataset

class NDataTranslator(**args*, ***kwargs*)

Translate Pytchography data from a set of images to an HDF5 file

crop_ronc(*ronc*)

Crop the input Ronchigram by the specified ammount using the specified method.

Parameters **ronc** (*numpy.array*) – Input image to be cropped.

Returns **cropped_ronc** – Cropped image

Return type numpy.array

static downSampRoncVec(*ronch_vec*, *binning_factor*)

Downsample the image by taking the mean over nearby values

Parameters

- **ronch_vec** (*ndarray*) – Image data
- **binning_factor** (*int*) – factor to reduce the size of the image by

Returns **ronc_mat3_mean** – Flattened downsampled image

Return type ndarray

translate(*h5_path*, *image_path*, *bin_factor=None*, *bin_func=<function mean>*, *start_image=0*, *scan_size_x=None*, *scan_size_y=None*, *crop_ammount=None*, *crop_method='percent'*)

Basic method that adds Ptychography data to existing hdf5 thisfile You must have already done the basic translation with BEodfTranslator

Parameters

- **h5_path** (*str*) – Absolute path to where the HDF5 file should be located
- **image_path** (*str*) – Absolute path to folder holding the image files or the path to a specific file
- **bin_factor** (*array_like of uint, optional*) – Downsampling factor for each dimension. Default is None.

- **bin_func** (*callable, optional*) – Function which will be called to calculate the return value of each block. Function must implement an axis parameter, i.e. numpy.mean. Ignored if bin_factor is None. Default is numpy.mean.
- **start_image** (*int, optional*) – Integer denoting which image in the file path should be considered the starting point. Default is 0, start with the first image on the list.
- **scan_size_x** (*int, optional*) – Number of Ronchigrams in the x direction. Default is None, value will be determined from the number of images and *scan_size_y* if it is given.
- **scan_size_y** (*int, optional*) – Number of Ronchigrams in the y direction. Default is None, value will be determined from the number of images and *scan_size_x* if it is given.
- **crop_ammount** (*uint or list of uints, optional*) – How much should be cropped from the original image. Can be a single unsigned integer or a list of unsigned integers. A single integer will crop the same ammount from all edges. A list of two integers will crop the x-dimension by the first integer and the y-dimension by the second integer. A list of 4 integers will crop the image as [left, right, top, bottom].
- **crop_method** (*str, optional*) – Which cropping method should be used. How much of the image is removed is determined by the value of *crop_ammount*. 'percent' - A percentage of the image is removed. 'absolute' - The specific number of pixel is removed.

Returns **h5_main** – HDF5 Dataset object that contains the flattened images

Return type h5py.Dataset

class FakeBEPSGenerator(**args*, ***kwargs*)

calc_loop_coef_mat(*image_list*)

Build the loop coefficient matrix

Parameters **image_list** (*list of numpy.ndarray*) – Images that will be used to generate the coefficients

Returns **coef_mat** – Array of loop coefficients

Return type numpy.ndarray

translate(*h5_path*, *n_steps=32*, *n_bins=37*, *start_freq=300000.0*, *end_freq=350000.0*, *data_type='BEPSData'*, *mode='DC modulation mode'*, *field_mode='in and out-of-field'*, *n_cycles=1*, *FORC_cycles=1*, *FORC_repeats=1*, *loop_a=1*, *loop_b=4*, *cycle_frac='full'*, *image_folder='/home/challtdow/workspace/pycroscopy/pycroscopy/io/translators/df_utils/beps_data_gen_images'*, *bin_factor=None*, *bin_func=<function mean>*, *image_type='.tif'*, *simple_coefs=False*)

Parameters

- **h5_path** (*str*) – Desired path to write the new HDF5 file
- **n_steps** (*uint, optional*) – Number of voltage steps Default - 32
- **n_bins** (*uint, optional*) – Number of frequency bins Default - 37
- **start_freq** (*float, optional*) – Starting frequency in Hz Default - 300E+3
- **end_freq** (*float, optional*) – Final freqency in Hz Default - 350E+3
- **data_type** (*str, optional*) – Type of data to generate Options - 'BEPSData', 'BELineData' Default - 'BEPSData'

- **mode** (*str, optional*) – Modulation mode to use when generating the data. Options - 'DC modulation mode', 'AC modulation mode' Default - 'DC modulation mode'
- **field_mode** (*str, optional*) – Field mode Options - 'in-field', 'out-of-field', 'in and out-of-field' Default - 'in and out-of-field'
- **n_cycles** (*uint, optional*) – Number of cycles Default - 1
- **FORC_cycles** (*uint, optional*) – Number of FORC cycles Default - 1
- **FORC_repeats** (*uint, optional*) – Number of FORC repeats Default - 1
- **loop_a** (*float, optional*) – Loop coefficient a Default - 1
- **loop_b** (*float, optional*) – Loop coefficient b
- **cycle_frac** (*str*) – Cycle fraction parameter. Default - 'full'
- **image_folder** (*str*) – Path to the images that will be used to generate the loop coefficients. There must be 11 images named '1.tif', '2.tif', ..., '11.tif' Default - pycroscopy.io.translators.df_utils.beps_gen_utils.beps_image_folder
- **bin_factor** (*array_like of uint, optional*) – Downsampling factor for each dimension. Default is None.
- **bin_func** (*callable, optional*) – Function which will be called to calculate the return value of each block. Function must implement an axis parameter, i.e. numpy.mean. Ignored if bin_factor is None. Default is numpy.mean.
- **image_type** (*str*) – File extension of images to be read. Default '.tif'
- **simple_coefs** (*bool*) – Should a simpler coefficient generation be used. Ensures loops, but all loops are identical. Default False

class LabViewH5Patcher

Patches the hdf5 files from the LabView V3 data aquisition software to meet the standards of the Pycroscopy data format.

translate(*h5_path*, *force_patch=False*, ***kwargs*)

Add the needed references and attributes to the h5 file that are not created by the LabView data aquisition program.

Parameters

- **h5_path** (*str*) – path to the h5 file
- **force_patch** (*bool, optional*) – Should the check to see if the file has already been patched be ignored. Default False.

Returns **h5_file** – patched hdf5 file

Return type h5py.File

class TRKPFMTranslator(**args*, ***kwargs*)

Translates trKPFM datasets from .mat and .dat files to .h5

translate(*parm_path*)

The main function that translates the provided file into a .h5 file

Parameters **parm_path** (*string / unicode*) – Absolute file path of the parameters .mat file.

Returns **h5_path** – Absolute path of the translated h5 file

Return type string / unicode

class BrukerAFMTranslator(*max_mem_mb=1024*, **args*, ***kwargs*)

Parameters **max_mem_mb** (*unsigned integer (Optional. Default = 1024)*) – Maximum system memory (in megabytes) that the translator can use

Returns

Return type Translator object

translate(*file_path*, **args*, ***kwargs*)

Translates a given Bruker / Veeco / Nanoscope AFM derived file to HDF5. Currently handles scans, force curves, and force-distance maps

Note that this translator was written with a single example file for each modality and may be buggy.

Parameters **file_path** (*str / unicode*) – path to data file

Returns **h5_path** – path to translated HDF5 file

Return type str / unicode

class ImageTranslator(**args*, ***kwargs*)

Translates data from a set of image files to an HDF5 file. This version has been extended to support dm3 and dm4 files

translate(*image_path*, *h5_path=None*, *bin_factor=None*, *bin_func=<function mean>*, *normalize=False*, ***image_args*)

Basic method that adds Ptychography data to existing hdf5 thisfile You must have already done the basic translation with BEodfTranslator

Parameters

- **image_path** (*str*) – Absolute path to folder holding the image files
- **h5_path** (*str, optional*) – Absolute path to where the HDF5 file should be located. Default is None
- **bin_factor** (*array_like of uint, optional*) – Downsampling factor for each dimension. Default is None.
- **bin_func** (*callable, optional*) – Function which will be called to calculate the return value of each block. Function must implement an axis parameter, i.e. numpy.mean. Ignored if bin_factor is None. Default is numpy.mean.
- **normalize** (*boolean, optional*) – Should the raw image be normalized when read in Default False
- **image_args** (*dict*) – Arguments to be passed to read_image. Arguments depend on the type of image.

Returns **h5_main** – HDF5 Dataset object that contains the flattened images

Return type h5py.Dataset

pycroscopy.processing

Pycroscopy's processing module

Submodules

cluster	Created on Tue Jan 05 07:55:56 2016
contrib	
decomposition	Created on Tue Jan 05 07:55:56 2016
fft	Created on Tue Oct 20 17:42:41 2015
gmode_utils	Created on Thu May 05 13:29:12 2016
image_processing	Created on Jun 16, 2016
proc_utils	Created on Tue Jan 05 07:55:56 2016
signal_filter	Created on Tue Nov 07 11:48:53 2017
svd_utils	Created on Mon Mar 28 09:45:08 2016

pycroscopy.processing.cluster

Created on Tue Jan 05 07:55:56 2016

@author: Suhas Somnath, Chris Smith

Functions

reorder_clusters(labels, mean_response[, ...])	Reorders clusters by the distances between the clusters

Classes

Cluster(h5_main, estimator[, num_comps])	Pycroscopy wrapper around the sklearn.cluster classes.

class Cluster(*h5_main*, *estimator*, *num_comps=None*, ***kwargs*)

Pycroscopy wrapper around the sklearn.cluster classes.

Constructs the Cluster object :param h5_main: Main dataset with ancillary spectroscopic, position indices and values datasets :type h5_main: HDF5 dataset object :param estimator: configured clustering algorithm to be applied to the data :type estimator: sklearn.cluster estimator :param num_comps: Number of features / spectroscopic indices to be used to cluster the data. Default = all :type num_comps: (optional) unsigned int :param args and kwargs: :type args and kwargs: arguments to be passed to the estimator

compute(*rearrange_clusters=True*, *override=False*)

Clusters the hdf5 dataset and calculates mean response for each cluster (by calling test() if it has not already been called), and writes the labels and mean response back to the h5 file.

Consider calling test_on_subset() to check results before writing to file. Results are deleted from memory upon writing to the HDF5 file

Parameters

- **rearrange_clusters** (*bool, optional. Default = True*) – Whether or not the clusters should be re-ordered by relative distances between the mean response
- **override** (*bool, optional. default = False*) – Set to true to recompute results if prior results are available. Else, returns existing results

Returns **h5_group** – Reference to the group that contains the clustering results

Return type HDF5 Group reference

delete_results()

Deletes results from memory.

test (*rearrange_clusters=True*, *override=False*)
Clusters the hdf5 dataset and calculates mean response for each cluster. This function does NOT write results to the hdf5 file. Call compute() to write to the file. Handles complex, compound datasets such that the mean response vector for each cluster matrix is of the same data-type as the input matrix.

Parameters

- **rearrange_clusters** (*bool, optional. Default = True*) – Whether or not the clusters should be re-ordered by relative distances between the mean response
- **override** (*bool, optional. default = False*) – Set to true to recompute results if prior results are available. Else, returns existing results

Returns

- **labels** (*1D unsigned int array*) – Array of cluster labels as obtained from the fit
- **mean_response** (*2D numpy array*) – Array of the mean response for each cluster arranged as [cluster number, response]

use_partial_computation (*h5_partial_group=None*)
Extracts the necessary parameters from the provided h5 group to resume computation

Parameters **h5_partial_group** (*h5py.Datagroup object*) – Datagroup containing partially computed results

reorder_clusters (*labels*, *mean_response*, *transform_function=None*)
Reorders clusters by the distances between the clusters

Parameters

- **labels** (*1D unsigned int numpy array*) – Labels for the clusters
- **mean_response** (*2D numpy array*) – Mean response of each cluster arranged as [cluster , features]
- **transform_function** (*callable, optional*) – Function that will convert the mean_response into real values

Returns

- **new_labels** (*1D unsigned int numpy array*) – Labels for the clusters arranged by distances
- **new_mean_response** (*2D numpy array*) – Mean response of each cluster arranged as [cluster , features]

pycroscopy.processing.contrib

pycroscopy.processing.decomposition

Created on Tue Jan 05 07:55:56 2016

@author: Suhas Somnath, Chris Smith

Classes

Decomposition(h5_main, estimator)	Pycroscopy wrapper around the sklearn.decomposition classes

class Decomposition(*h5_main*, *estimator*)

Pycroscopy wrapper around the sklearn.decomposition classes

Uses the provided (preconfigured) Decomposition object to decompose the provided dataset

Parameters

- **h5_main** (*HDF5 dataset object*) – Main dataset with ancillary spectroscopic, position indices and values datasets
- **estimator** (*sklearn.cluster estimator object*) – configured decomposition object to apply to the data

compute(*override=False*)

Decomposes the hdf5 dataset to calculate the components and projection (by calling test() if it hasn't already been called), and writes the results back to the hdf5 file

Parameters **override** (*bool, optional. default = False*) – Set to true to recompute results if prior results are available. Else, returns existing results

Returns **h5_group** – Reference to the group that contains the decomposition results

Return type HDF5 Group reference

delete_results()

Deletes results from memory.

test(*override=False*)

Decomposes the hdf5 dataset to calculate the components and projection. This function does NOT write results to the hdf5 file. Call compute() to write to the file. Handles complex, compound datasets such that the components are of the same data-type as the input matrix.

Parameters **override** (*bool, optional. default = False*) – Set to true to recompute results if prior results are available. Else, returns existing results

Returns

- **components** (*numpy array*) – Components
- **projections** (*numpy array*) – Projections

use_partial_computation(*h5_partial_group=None*)

Extracts the necessary parameters from the provided h5 group to resume computation

Parameters **h5_partial_group** (*h5py.Datagroup object*) – Datagroup containing partially computed results

pycroscopy.processing.fft

Created on Tue Oct 20 17:42:41 2015

@author: Suhas Somnath

Functions

are_compatible_filters(frequency_filters)	
build_composite_freq_filter(frequency_filters)	

Continued on next page

Table 21 – continued from previous page

build_radius_matrix(image_shape)	Builds a matrix where the value of a given pixel is its L2 distance from the origin, which is located at the center of the provided image rather one of the corners of the image.
down_sample(fft_vec, freq_ratio)	Downsamples the provided data vector
fft_to_real(image)	Provides the real-space equivalent of the provided image in Fourier space
get_2d_gauss_lpf(radius_mat, filter_width)	Builds a 2D, radially symmetric, low-pass Gaussian filter based on the provided radius matrix.
get_fft_stack(image_stack)	Gets the 2D FFT for a single or stack of images by applying a blackman window
get_noise_floor(fft_data, tolerance)	Calculate the noise floor from the FFT data.

Classes

FrequencyFilter(signal_length, samp_rate, ...)	
HarmonicPassFilter(signal_length, samp_rate, ...)	Builds a filter that only keeps N harmonics
LowPassFilter(signal_length, samp_rate, f_cutoff)	Builds a low pass filter
NoiseBandFilter(signal_length, samp_rate, ...)	Builds a filter that removes specified noise frequencies

build_radius_matrix(*image_shape*)

Builds a matrix where the value of a given pixel is its L2 distance from the origin, which is located at the center of the provided image rather one of the corners of the image. The result from this function is required by get_2d_gauss_lpf

Parameters **image_shape** (*list or tuple*) – Number of rows and columns in the image

Returns **radius_mat** – Radius matrix

Return type 2d numpy float array

down_sample(*fft_vec*, *freq_ratio*)

Downsamples the provided data vector

Parameters

- **fft_vec** (*1D complex numpy array*) – Waveform that is already FFT shifted
- **freq_ratio** (*float*) – new sampling rate / old sampling rate (less than 1)

Returns **fft_vec** – downsampled waveform

Return type 1D numpy array

fft_to_real(*image*)

Provides the real-space equivalent of the provided image in Fourier space

Parameters **image** (*2D numpy float array*) – FFT of image that has been fft shifted.

Returns **image** – Image in real space

Return type 2D numpy float array

get_2d_gauss_lpf(*radius_mat*, *filter_width*)

Builds a 2D, radially symmetric, low-pass Gaussian filter based on the provided radius matrix. The corresponding high pass filter can be built simply by subtracting the resulting low-pass filter from 1.

Multiply the output of this function with the (shifted) fft of an image to apply the filter.

Parameters

- **radius_mat** (*2d numpy float array*) – A [NxM] matrix of the same size as the image that this filter will be applied to
- **filter_width** (*float*) – Size of the filter

Returns **gauss_filt** – matrix with a single gaussian peak at the center of the matrix.

Return type 2D numpy float array

get_fft_stack (*image_stack*)

Gets the 2D FFT for a single or stack of images by applying a blackman window

Parameters **image_stack** (*2D or 3D real numpy array*) – Either a 2D matrix [x, y] or a stack of 2D images arranged as [z or spectral, x, y]

Returns **fft_stack** – 2 or 3 dimensional matrix arranged as [z or spectral, x, y]

Return type 2D or 3D real numpy array

get_noise_floor (*fft_data*, *tolerance*)

Calculate the noise floor from the FFT data. Algorithm originally written by Mahmut Okatan Baris

Parameters

- **fft_data** (*1D or 2D complex numpy array*) – Signal in frequency space (ie - after FFT shifting) arranged as (channel or repetition, signal)
- **tolerance** (*unsigned float*) – Tolerance to noise. A smaller value gets rid of more noise.

Returns **noise_floor** – One value per channel / repetition

Return type 1D array-like

pycroscopy.processing.gmode_utils

Created on Thu May 05 13:29:12 2016

@author: Suhas Somnath

Functions

decompress_response(f_condensed_mat, ...)	Returns the time domain representation of waveform(s) that are compressed in the frequency space
reshape_from_lines_to_pixels(h5_main, ...[, ...])	Breaks up the provided raw G-mode dataset into lines and pixels (from just lines)
test_filter(resp_wfm[, frequency_filters, ...])	Filters the provided response with the provided filters.

decompress_response (*f_condensed_mat*, *num_pts*, *hot_inds*)

Returns the time domain representation of waveform(s) that are compressed in the frequency space

Parameters

- **f_condensed_mat** (*1D or 2D complex numpy arrays*) – Frequency domain signals arranged as [position, frequency]. Only the positive frequncy bins must be in the compressed dataset. The dataset is assumed to have been FFT shifted (such that 0 Hz is at

the center).

- **num_pts** (*unsigned int*) – Number of points in the time domain signal
- **hot_inds** (*1D unsigned int numpy array*) – Indices of the frequency bins in the compressed data. This index array will be necessary to reverse map the condensed FFT into its original form

Returns **time_resp** – Time domain response arranged as [position, time]

Return type 2D numpy array

Notes

Memory is given higher priority here, so this function loops over the position instead of doing the inverse FFT on the complete data.

reshape_from_lines_to_pixels(*h5_main*, *pts_per_cycle*, *scan_step_x_m=None*)
Breaks up the provided raw G-mode dataset into lines and pixels (from just lines)

Parameters

- **h5_main** (*h5py.Dataset object*) – Reference to the main dataset that contains the raw data that is only broken up by lines
- **pts_per_cycle** (*unsigned int*) – Number of points in a single pixel
- **scan_step_x_m** (*float*) – Step in meters for pixels

Returns **h5_resh** – Reference to the main dataset that contains the reshaped data

Return type h5py.Dataset object

test_filter(*resp_wfm*, *frequency_filters=None*, *noise_threshold=None*, *excit_wfm=None*, *show_plots=True*, *plot_title=None*, *verbose=False*)
Filters the provided response with the provided filters.

Parameters

- **resp_wfm** (*array-like, 1D*) – Raw response waveform in the time domain
- **frequency_filters** (*(Optional) FrequencyFilter object or list of*) – Frequency filters to apply to signal
- **noise_threshold** (*(Optional) float*) – Noise threshold to apply to signal
- **excit_wfm** (*(Optional) 1D array-like*) – Excitation waveform in the time domain. This waveform is necessary for plotting loops. If the length of resp_wfm matches excit_wfm, a single plot will be returned with the raw and filtered signals plotted against the excit_wfm. Else, resp_wfm and the filtered (filt_data) signal will be broken into chunks matching the length of excit_wfm and a figure with multiple plots (one for each chunk) with the raw and filtered signal chunks plotted against excit_wfm will be returned for fig_loops
- **show_plots** (*(Optional) Boolean*) – Whether or not to plot FFTs before and after filtering
- **plot_title** (*str / unicode (Optional)*) – Title for the raw vs filtered plots if requested. For example - 'Row 15'
- **verbose** (*(Optional) Boolean*) – Prints extra debugging information if True. Default False

Returns

- **filt_data** (*1D numpy float array*) – Filtered signal in the time domain
- **fig_fft** (*matplotlib.pyplot.figure object*) – handle to the plotted figure if requested, else None
- **fig_loops** (*matplotlib.pyplot.figure object*) – handle to figure with the filtered signal and raw signal plotted against the excitation waveform

pycroscopy.processing.image_processing

Created on Jun 16, 2016

@author: Chris Smith – csmith55@utk.edu

Functions

radially_average_correlation(data_mat, num_r_bin)	Calculates the radially average correlation functions for a given 2D image

Classes

ImageWindow(h5_main[, max_RAM_mb, cores, reset])	This class will handle the reading of a raw image file, creating windows from it, and writing those windows to an HDF5 file.

class ImageWindow(*h5_main*, *max_RAM_mb=1024*, *cores=None*, *reset=True*, ***image_args*)

This class will handle the reading of a raw image file, creating windows from it, and writing those windows to an HDF5 file.

Setup the image windowing

Parameters

- **h5_main** (*h5py.Dataset*) – HDF5 Dataset containing the data to be windowed
- **max_RAM_mb** (*int, optional*) – integer maximum amount of ram, in Mb, to use in windowing Default 1024
- **cores** (*int, optional*) – integer number of [logical] CPU cores to use in windowing Defualt None, use number of available cores minus 2
- **reset** (*Boolean, optional*) – should all data in the hdf5 file be deleted

static abs_fft_func(*image*)

Take the 2d FFT of each window in *windows* and return in the proper form.

Parameters **image** (*numpy.ndarray*) – Windowed image to take the FFT of

Returns **windows** – Array of the Magnitude of the FFT of each window for the input *image*

Return type numpy.ndarray

build_clean_image(*h5_win=None*)

Reconstructs the cleaned image from the windowed dataset

Parameters **h5_win** (*HDF5 dataset , optional*) – The windowed image to be reconstructed.

Returns **h5_clean** – The cleaned image

Return type HDF5 dataset

clean_and_build_batch(*h5_win=None*, *components=None*)
Rebuild the Image from the SVD results on the windows Optionally, only use components less than n_comp.

Parameters

- **h5_win** (*hdf5 Dataset, optional*) – dataset containing the windowed image which SVD was performed on
- **components** (*{int, iterable of int, slice} optional*) – Defines which components to keep Default - None, all components kept

  Input Types integer : Components less than the input will be kept length 2 iterable of integers : Integers define start and stop of component slice to retain other iterable of integers or slice : Selection of component indices to retain

Returns **clean_wins** – the cleaned windows

Return type HDF5 Dataset

clean_and_build_separate_components(*h5_win=None*, *components=None*)
Rebuild the Image from the SVD results on the windows Optionally, only use components less than n_comp.

Parameters

- **h5_win** (*hdf5 Dataset, optional*) – dataset containing the windowed image which SVD was performed on
- **components** – Defines which components to keep Default - None, all components kept

Returns **clean_wins** – the cleaned windows

Return type HDF5 Dataset

do_windowing(*win_x=None*, *win_y=None*, *win_step_x=1*, *win_step_y=1*, *win_fft=None*, **args*, ***kwargs*)
Extract the windows from the normalized image and write them to the file

Parameters

- **win_x** (*int, optional*) – size of the window, in pixels, in the horizontal direction Default None, a guess will be made based on the FFT of the image
- **win_y** (*int, optional*) – size of the window, in pixels, in the vertical direction Default None, a guess will be made based on the FFT of the image
- **win_step_x** (*int, optional*) – step size, in pixels, to take between windows in the horizontal direction Default 1
- **win_step_y** (*int, optional*) – step size, in pixels, to take between windows in the vertical direction Default 1
- **win_fft** (*str, optional*) – What kind of fft should be stored with the windows. Options are None - Only the window 'abs' - Only the magnitude of the fft 'data+abs' - The window and magnitude of the fft 'data+complex' - The window and the complex fft Default None

Returns **h5_wins** – Dataset containing the flattened windows

Return type HDF5 Dataset

plot_clean_image(*h5_clean=None*, *image_path=None*, *image_type='png'*, *save_plots=True*, *show_plots=False*, *cmap='gray'*)
Plot the cleaned image stored in the HDF5 dataset h5_clean

Parameters

- **h5_clean** (*HDF5 dataset, optional*) – cleaned image to be plotted
- **image_path** (*str, optional*) – path to save cleaned image file Default None, '_clean' will be appened to the name of the input image
- **image_type** (*str, optional*) – image format to save the cleaned image as Default 'png', all formats recognized by matplotlib.pyplot.imsave are allowed
- **save_plots** (*Boolean, pptional*) – If true, the image will be saved to image_path with the extention specified by image_type Default True
- **show_plots** (*Boolean, optional*) – If true, the image will be displayed on the screen Default False
- **cmap** (*str, optional*) – matplotlib colormap string designation

Returns **clean_image** – object holding the plot of the cleaned image

Return type Axis_Image

static win_abs_fft_func(*image*)
Take the 2d FFT of each window in *windows* and return in the proper form.

Parameters **image** (*numpy.ndarray*) – Windowed image to take the FFT of

Returns **windows** – Array of windows and the Magnitude of the FFT of each window for the input *image*

Return type numpy.ndarray

static win_comp_fft_func(*image*)
Take the 2d FFT of each window in *windows* and return in the proper form.

Parameters **image** (*numpy.ndarray*) – Windowed image to take the FFT of

Returns **windows** – Array of windows and the FFT of each window for the input *image*

Return type numpy.ndarray

static win_data_func(*image*)
Returns the input image in the *windata32* format

Parameters **image** (*numpy.ndarray*) – Windowed image to take the FFT of

Returns **windows** – Array the image in the windata32 format

Return type numpy.ndarray

window_size_extract(*num_peaks=2*, *save_plots=True*, *show_plots=False*)
Take the normalized image and extract from it an optimal window size

Parameters

- **num_peaks** (*int, optional*) – number of peaks to use during least squares fit Default 2
- **save_plots** (*Boolean, optional*) – If True then a plot showing the quality of the fit will be generated and saved to disk. Ignored if do_fit is false. Default True
- **show_plots** (*Boolean, optional*) – If True then a plot showing the quality of the fit will be generated and shown on screen. Ignored if do_fit is false. Default False

Returns

- **window_size** (*int*) – Optimal window size in pixels
- **psf_width** (*int*) – Estimate atom spacing in pixels

radially_average_correlation (*data_mat*, *num_r_bin*)
Calculates the radially average correlation functions for a given 2D image

Parameters

- **data_mat** (*2D real numpy array*) – Image to analyze
- **num_r_bin** (*unsigned int*) – Number of spatial bins to analyze

Returns

- **a_mat** (*2D real numpy array*) – Noise spectrum of the image
- **a_rad_avg_vec** (*1D real numpy array*) – Average value of the correlation as a function of feature size
- **a_rad_max_vec** (*1D real numpy array*) – Maximum value of the correlation as a function of feature size
- **a_rad_min_vec** (*1D real numpy array*) – Minimum value of the correlation as a function of feature size
- **a_rad_std_vec** (*1D real numpy array*) – Standard deviation of the correlation as a function of feature size

pycroscopy.processing.proc_utils

Created on Tue Jan 05 07:55:56 2016

@author: Chris Smith, Suhas Somnath

Functions

get_component_slice(components[, ...])	Check the components object to determine how to use it to slice the dataset
to_ranges(iterable)	Converts a sequence of iterables to range tuples

get_component_slice (*components*, *total_components=None*)
Check the components object to determine how to use it to slice the dataset

Parameters

- **components** (*{int, array-like of ints, slice, or None}*) – Input Options integer: Components less than the input will be kept length 2 iterable of integers: Integers define start and stop of component slice to retain other iterable of integers or slice: Selection of component indices to retain None: All components will be used
- **total_components** (*uint, optional. Default = None*) – Total number of spectral components in the dataset

Returns

- **comp_slice** (*slice or numpy.ndarray of uints*) – Slice or array specifying which components should be kept

- **num_comps** (*uint*) – Number of selected components

to_ranges (*iterable*)
Converts a sequence of iterables to range tuples

From https://stackoverflow.com/questions/4628333/converting-a-list-of-integers-into-range-in-python

Credits: @juanchopanza and @luca

Parameters **iterable** (*collections.Iterable object*) – iterable object like a list

Returns **iterable** – Cast to list or similar to use

Return type generator object

pycroscopy.processing.signal_filter

Created on Tue Nov 07 11:48:53 2017

@author: Suhas Somnath

Classes

SignalFilter(h5_main[, frequency_filters, ...])	Filters the entire h5 dataset with the given filtering parameters.

class SignalFilter (*h5_main*, *frequency_filters=None*, *noise_threshold=None*, *write_filtered=True*, *write_condensed=False*, *num_pix=1*, *phase_rad=0*, ***kwargs*)
Filters the entire h5 dataset with the given filtering parameters.

Parameters

- **h5_main** (*h5py.Dataset object*) – Dataset to process
- **frequency_filters** (*(Optional) single or list of pycroscopy.fft.FrequencyFilter objects*) – Frequency (vertical) filters to apply to signal
- **noise_threshold** (*(Optional) float. Default – None*) – Noise tolerance to apply to data. Value must be within (0, 1)
- **write_filtered** (*(Optional) bool. Default – True*) – Whether or not to write the filtered data to file
- **write_condensed** (*Optional) bool. Default – False*) – Whether or not to write the condensed data in frequency space to file. Use this for datasets that are very large but sparse in frequency space.
- **num_pix** (*(Optional) uint. Default – 1*) – Number of pixels to use for filtering. More pixels means a lower noise floor and the ability to pick up weaker signals. Use only if absolutely necessary. This value must be a divisor of the number of pixels in the dataset
- **phase_rad** (*(Optional) float*) – Degrees by which the output is rotated with respect to the input to compensate for phase lag. This feature has NOT yet been implemented.
- **kwargs** (*(Optional) dictionary*) – Please see Process class for additional inputs

compute (*override=False*, **args*, ***kwargs*)
Creates placeholders for the results, applies the unit computation to chunks of the dataset

Parameters

- **override** (*bool, optional. default = False*) – By default, compute will simply return duplicate results to avoid recomputing or resume computation on a group with partial results. Set to True to force fresh computation.
- **args** (*list*) – arguments to the mapped function in the correct order
- **kwargs** (*dictionary*) – keyword arguments to the mapped function

Returns **h5_results_grp** – Datagroup containing all the results

Return type h5py.Datagroup object

test (*pix_ind=None*, *excit_wfm=None*, ***kwargs*)

Tests the signal filter on a single pixel (randomly chosen unless manually specified) worth of data.

Parameters

- **pix_ind** (*int, optional. default = random*) – Index of the pixel whose data will be used for inference
- **excit_wfm** (*array-like, optional. default = None*) – Waveform against which the raw and filtered signals will be plotted. This waveform can be a fraction of the length of a single pixel's data. For example, in the case of G-mode, where a single scan line is yet to be broken down into pixels, the excitation waveform for a single pixel can br provided to automatically break the raw and filtered responses also into chunks of the same size.

Returns

Return type fig, axes

use_partial_computation (*h5_partial_group=None*)

Extracts the necessary parameters from the provided h5 group to resume computation

Parameters **h5_partial_group** (*h5py.Datagroup object*) – Datagroup containing partially computed results

write_condensed = None

Remember that the default number of pixels corresponds to only the raw data that can be held in memory In the case of signal filtering, the datasets that will occupy space are: 1. Raw, 2. filtered (real + freq space copies), 3. Condensed (substantially lesser space) The actual scaling of memory depends on options:

pycroscopy.processing.svd_utils

Created on Mon Mar 28 09:45:08 2016

@author: Suhas Somnath, Chris Smith

Functions

rebuild_svd(h5_main[, components, cores, ...])	Rebuild the Image from the SVD results on the windows Optionally, only use components less than n_comp.
simplified_kpca(kpca, source_data)	Performs kernel PCA on the provided dataset and returns the familiar eigenvector, eigenvalue, and scree matrices.

Classes

SVD(h5_main[, num_components])	

class SVD(*h5_main*, *num_components=None*)

compute(*override=False*)
Computes SVD (by calling test_on_subset() if it has not already been called) and writes results to file. Consider calling test() to check results before writing to file. Results are deleted from memory upon writing to the HDF5 file

Parameters **override** (*bool, optional. default = False*) – Set to true to recompute results if prior results are available. Else, returns existing results

Returns **h5_results_grp** – Datagroup containing all the results

Return type h5py.Datagroup object

delete_results()
Deletes results from memory.

process_name = None
Calculate the size of the main data in memory and compare to max_mem We use the minimum of the actual dtype's itemsize and float32 since we don't want to read it in yet and do the proper type conversions.

test(*override=False*)
Applies randomised VD to the dataset. This function does NOT write results to the hdf5 file. Call compute() to write to the file. Handles complex, compound datasets such that the V matrix is of the same data-type as the input matrix.

Parameters **override** (*bool, optional. default = False*) – Set to true to recompute results if prior results are available. Else, returns existing results

Returns

- **U** (*numpy.ndarray*) – Abundance matrix
- **S** (*numpy.ndarray*) – variance vector
- **V** (*numpy.ndarray*) – eigenvector matrix

use_partial_computation(*h5_partial_group=None*)
Extracts the necessary parameters from the provided h5 group to resume computation

Parameters **h5_partial_group** (*h5py.Datagroup object*) – Datagroup containing partially computed results

rebuild_svd(*h5_main*, *components=None*, *cores=None*, *max_RAM_mb=1024*)
Rebuild the Image from the SVD results on the windows Optionally, only use components less than n_comp.

Parameters

- **h5_main** (*hdf5 Dataset*) – dataset which SVD was performed on
- **components** (*{int, iterable of int, slice} optional*) – Defines which components to keep Default - None, all components kept

  Input Types integer : Components less than the input will be kept length 2 iterable of integers : Integers define start and stop of component slice to retain other iterable of integers or slice : Selection of component indices to retain

- **cores** (*int, optional*) – How many cores should be used to rebuild Default - None, all but 2 cores will be used, min 1
- **max_RAM_mb** (*int, optional*) – Maximum ammount of memory to use when rebuilding, in Mb. Default - 1024Mb

Returns **rebuilt_data** – the rebuilt dataset

Return type HDF5 Dataset

simplified_kpca(*kpca*, *source_data*)

Performs kernel PCA on the provided dataset and returns the familiar eigenvector, eigenvalue, and scree matrices.

Note that the positions in the eigenvalues may need to be transposed

Parameters

- **kpca** (*KernelPCA object*) – configured Kernel PCA object ready to perform analysis
- **source_data** (*2D numpy array*) – Data arranged as [iteration, features] example - [position, time]

Returns

- **eigenvalues** (*2D numpy array*) – Eigenvalues in the original space arranged as [component,iteration]
- **scree** (*1D numpy array*) – S component
- **eigenvector** (*2D numpy array*) – Eigenvectors in the original space arranged as [component,features]

class Cluster(*h5_main*, *estimator*, *num_comps=None*, ***kwargs*)

Pycroscopy wrapper around the sklearn.cluster classes.

Constructs the Cluster object :param h5_main: Main dataset with ancillary spectroscopic, position indices and values datasets :type h5_main: HDF5 dataset object :param estimator: configured clustering algorithm to be applied to the data :type estimator: sklearn.cluster estimator :param num_comps: Number of features / spectroscopic indices to be used to cluster the data. Default = all :type num_comps: (optional) unsigned int :param args and kwargs: :type args and kwargs: arguments to be passed to the estimator

compute(*rearrange_clusters=True*, *override=False*)

Clusters the hdf5 dataset and calculates mean response for each cluster (by calling test() if it has not already been called), and writes the labels and mean response back to the h5 file.

Consider calling test_on_subset() to check results before writing to file. Results are deleted from memory upon writing to the HDF5 file

Parameters

- **rearrange_clusters** (*bool, optional. Default = True*) – Whether or not the clusters should be re-ordered by relative distances between the mean response
- **override** (*bool, optional. default = False*) – Set to true to recompute results if prior results are available. Else, returns existing results

Returns **h5_group** – Reference to the group that contains the clustering results

Return type HDF5 Group reference

delete_results()

Deletes results from memory.

test(*rearrange_clusters=True*, *override=False*)
Clusters the hdf5 dataset and calculates mean response for each cluster. This function does NOT write results to the hdf5 file. Call compute() to write to the file. Handles complex, compound datasets such that the mean response vector for each cluster matrix is of the same data-type as the input matrix.

Parameters

- **rearrange_clusters** (*bool, optional. Default = True*) – Whether or not the clusters should be re-ordered by relative distances between the mean response
- **override** (*bool, optional. default = False*) – Set to true to recompute results if prior results are available. Else, returns existing results

Returns

- **labels** (*1D unsigned int array*) – Array of cluster labels as obtained from the fit
- **mean_response** (*2D numpy array*) – Array of the mean response for each cluster arranged as [cluster number, response]

use_partial_computation(*h5_partial_group=None*)
Extracts the necessary parameters from the provided h5 group to resume computation

Parameters **h5_partial_group** (*h5py.Datagroup object*) – Datagroup containing partially computed results

class Decomposition(*h5_main*, *estimator*)
Pycroscopy wrapper around the sklearn.decomposition classes

Uses the provided (preconfigured) Decomposition object to decompose the provided dataset

Parameters

- **h5_main** (*HDF5 dataset object*) – Main dataset with ancillary spectroscopic, position indices and values datasets
- **estimator** (*sklearn.cluster estimator object*) – configured decomposition object to apply to the data

compute(*override=False*)
Decomposes the hdf5 dataset to calculate the components and projection (by calling test() if it hasn't already been called), and writes the results back to the hdf5 file

Parameters **override** (*bool, optional. default = False*) – Set to true to recompute results if prior results are available. Else, returns existing results

Returns **h5_group** – Reference to the group that contains the decomposition results

Return type HDF5 Group reference

delete_results()
Deletes results from memory.

test(*override=False*)
Decomposes the hdf5 dataset to calculate the components and projection. This function does NOT write results to the hdf5 file. Call compute() to write to the file. Handles complex, compound datasets such that the components are of the same data-type as the input matrix.

Parameters **override** (*bool, optional. default = False*) – Set to true to recompute results if prior results are available. Else, returns existing results

Returns

- **components** (*numpy array*) – Components

- **projections** (*numpy array*) – Projections

use_partial_computation(*h5_partial_group=None*)
Extracts the necessary parameters from the provided h5 group to resume computation

Parameters **h5_partial_group** (*h5py.Datagroup object*) – Datagroup containing partially computed results

class ImageWindow(*h5_main*, *max_RAM_mb=1024*, *cores=None*, *reset=True*, ***image_args*)
This class will handle the reading of a raw image file, creating windows from it, and writing those windows to an HDF5 file.

Setup the image windowing

Parameters

- **h5_main** (*h5py.Dataset*) – HDF5 Dataset containing the data to be windowed
- **max_RAM_mb** (*int, optional*) – integer maximum amount of ram, in Mb, to use in windowing Default 1024
- **cores** (*int, optional*) – integer number of [logical] CPU cores to use in windowing Defualt None, use number of available cores minus 2
- **reset** (*Boolean, optional*) – should all data in the hdf5 file be deleted

static abs_fft_func(*image*)
Take the 2d FFT of each window in *windows* and return in the proper form.

Parameters **image** (*numpy.ndarray*) – Windowed image to take the FFT of

Returns **windows** – Array of the Magnitude of the FFT of each window for the input *image*

Return type numpy.ndarray

build_clean_image(*h5_win=None*)
Reconstructs the cleaned image from the windowed dataset

Parameters **h5_win** (*HDF5 dataset , optional*) – The windowed image to be reconstructed.

Returns **h5_clean** – The cleaned image

Return type HDF5 dataset

clean_and_build_batch(*h5_win=None*, *components=None*)
Rebuild the Image from the SVD results on the windows Optionally, only use components less than n_comp.

Parameters

- **h5_win** (*hdf5 Dataset, optional*) – dataset containing the windowed image which SVD was performed on
- **components** (*{int, iterable of int, slice} optional*) – Defines which components to keep Default - None, all components kept

  Input Types integer : Components less than the input will be kept length 2 iterable of integers : Integers define start and stop of component slice to retain other iterable of integers or slice : Selection of component indices to retain

Returns **clean_wins** – the cleaned windows

Return type HDF5 Dataset

clean_and_build_separate_components(*h5_win=None*, *components=None*)
Rebuild the Image from the SVD results on the windows Optionally, only use components less than n_comp.

Parameters

- **h5_win** (*hdf5 Dataset, optional*) – dataset containing the windowed image which SVD was performed on
- **components** – Defines which components to keep Default - None, all components kept

Returns **clean_wins** – the cleaned windows

Return type HDF5 Dataset

do_windowing(*win_x=None*, *win_y=None*, *win_step_x=1*, *win_step_y=1*, *win_fft=None*, **args*, ***kwargs*)
Extract the windows from the normalized image and write them to the file

Parameters

- **win_x** (*int, optional*) – size of the window, in pixels, in the horizontal direction Default None, a guess will be made based on the FFT of the image
- **win_y** (*int, optional*) – size of the window, in pixels, in the vertical direction Default None, a guess will be made based on the FFT of the image
- **win_step_x** (*int, optional*) – step size, in pixels, to take between windows in the horizontal direction Default 1
- **win_step_y** (*int, optional*) – step size, in pixels, to take between windows in the vertical direction Default 1
- **win_fft** (*str, optional*) – What kind of fft should be stored with the windows. Options are None - Only the window 'abs' - Only the magnitude of the fft 'data+abs' - The window and magnitude of the fft 'data+complex' - The window and the complex fft Default None

Returns **h5_wins** – Dataset containing the flattened windows

Return type HDF5 Dataset

plot_clean_image(*h5_clean=None*, *image_path=None*, *image_type='png'*, *save_plots=True*, *show_plots=False*, *cmap='gray'*)
Plot the cleaned image stored in the HDF5 dataset h5_clean

Parameters

- **h5_clean** (*HDF5 dataset, optional*) – cleaned image to be plotted
- **image_path** (*str, optional*) – path to save cleaned image file Default None, '_clean' will be appened to the name of the input image
- **image_type** (*str, optional*) – image format to save the cleaned image as Default 'png', all formats recognized by matplotlib.pyplot.imsave are allowed
- **save_plots** (*Boolean, pptional*) – If true, the image will be saved to image_path with the extention specified by image_type Default True
- **show_plots** (*Boolean, optional*) – If true, the image will be displayed on the screen Default False
- **cmap** (*str, optional*) – matplotlib colormap string designation

Returns **clean_image** – object holding the plot of the cleaned image

Return type Axis_Image

static win_abs_fft_func(*image*)
Take the 2d FFT of each window in *windows* and return in the proper form.

Parameters **image** (*numpy.ndarray*) – Windowed image to take the FFT of

Returns **windows** – Array of windows and the Magnitude of the FFT of each window for the input *image*

Return type numpy.ndarray

static win_comp_fft_func(*image*)
Take the 2d FFT of each window in *windows* and return in the proper form.

Parameters **image** (*numpy.ndarray*) – Windowed image to take the FFT of

Returns **windows** – Array of windows and the FFT of each window for the input *image*

Return type numpy.ndarray

static win_data_func(*image*)
Returns the input image in the *windata32* format

Parameters **image** (*numpy.ndarray*) – Windowed image to take the FFT of

Returns **windows** – Array the image in the windata32 format

Return type numpy.ndarray

window_size_extract(*num_peaks=2*, *save_plots=True*, *show_plots=False*)
Take the normalized image and extract from it an optimal window size

Parameters

- **num_peaks** (*int, optional*) – number of peaks to use during least squares fit Default 2
- **save_plots** (*Boolean, optional*) – If True then a plot showing the quality of the fit will be generated and saved to disk. Ignored if do_fit is false. Default True
- **show_plots** (*Boolean, optional*) – If True then a plot showing the quality of the fit will be generated and shown on screen. Ignored if do_fit is false. Default False

Returns

- **window_size** (*int*) – Optimal window size in pixels
- **psf_width** (*int*) – Estimate atom spacing in pixels

class SVD(*h5_main*, *num_components=None*)

compute(*override=False*)
Computes SVD (by calling test_on_subset() if it has not already been called) and writes results to file. Consider calling test() to check results before writing to file. Results are deleted from memory upon writing to the HDF5 file

Parameters **override** (*bool, optional. default = False*) – Set to true to recompute results if prior results are available. Else, returns existing results

Returns **h5_results_grp** – Datagroup containing all the results

Return type h5py.Datagroup object

delete_results()
Deletes results from memory.

process_name = None
Calculate the size of the main data in memory and compare to max_mem We use the minimum of the actual dtype's itemsize and float32 since we don't want to read it in yet and do the proper type conversions.

test(*override=False*)
Applies randomised VD to the dataset. This function does NOT write results to the hdf5 file. Call compute() to write to the file. Handles complex, compound datasets such that the V matrix is of the same data-type as the input matrix.

Parameters **override** (*bool, optional. default = False*) – Set to true to recompute results if prior results are available. Else, returns existing results

Returns

- **U** (*numpy.ndarray*) – Abundance matrix
- **S** (*numpy.ndarray*) – variance vector
- **V** (*numpy.ndarray*) – eigenvector matrix

use_partial_computation(*h5_partial_group=None*)
Extracts the necessary parameters from the provided h5 group to resume computation

Parameters **h5_partial_group** (*h5py.Datagroup object*) – Datagroup containing partially computed results

rebuild_svd(*h5_main*, *components=None*, *cores=None*, *max_RAM_mb=1024*)
Rebuild the Image from the SVD results on the windows Optionally, only use components less than n_comp.

Parameters

- **h5_main** (*hdf5 Dataset*) – dataset which SVD was performed on
- **components** (*{int, iterable of int, slice} optional*) – Defines which components to keep Default - None, all components kept

  Input Types integer : Components less than the input will be kept length 2 iterable of integers : Integers define start and stop of component slice to retain other iterable of integers or slice : Selection of component indices to retain
- **cores** (*int, optional*) – How many cores should be used to rebuild Default - None, all but 2 cores will be used, min 1
- **max_RAM_mb** (*int, optional*) – Maximum ammount of memory to use when rebuilding, in Mb. Default - 1024Mb

Returns **rebuilt_data** – the rebuilt dataset

Return type HDF5 Dataset

class SignalFilter(*h5_main*, *frequency_filters=None*, *noise_threshold=None*, *write_filtered=True*, *write_condensed=False*, *num_pix=1*, *phase_rad=0*, ***kwargs*)
Filters the entire h5 dataset with the given filtering parameters.

Parameters

- **h5_main** (*h5py.Dataset object*) – Dataset to process
- **frequency_filters** (*(Optional) single or list of pycroscopy.fft.FrequencyFilter objects*) – Frequency (vertical) filters to apply to signal
- **noise_threshold** (*(Optional) float. Default – None*) – Noise tolerance to apply to data. Value must be within (0, 1)
- **write_filtered** (*(Optional) bool. Default – True*) – Whether or not to write the filtered data to file

- **write_condensed** (*Optional) bool. Default – False*) – Whether or not to write the condensed data in frequency space to file. Use this for datasets that are very large but sparse in frequency space.
- **num_pix** (*(Optional) uint. Default – 1*) – Number of pixels to use for filtering. More pixels means a lower noise floor and the ability to pick up weaker signals. Use only if absolutely necessary. This value must be a divisor of the number of pixels in the dataset
- **phase_rad** (*(Optional) float*) – Degrees by which the output is rotated with respect to the input to compensate for phase lag. This feature has NOT yet been implemented.
- **kwargs** (*(Optional) dictionary*) – Please see Process class for additional inputs

compute (*override=False*, **args*, ***kwargs*)
Creates placeholders for the results, applies the unit computation to chunks of the dataset

Parameters

- **override** (*bool, optional. default = False*) – By default, compute will simply return duplicate results to avoid recomputing or resume computation on a group with partial results. Set to True to force fresh computation.
- **args** (*list*) – arguments to the mapped function in the correct order
- **kwargs** (*dictionary*) – keyword arguments to the mapped function

Returns **h5_results_grp** – Datagroup containing all the results

Return type h5py.Datagroup object

test (*pix_ind=None*, *excit_wfm=None*, ***kwargs*)
Tests the signal filter on a single pixel (randomly chosen unless manually specified) worth of data.

Parameters

- **pix_ind** (*int, optional. default = random*) – Index of the pixel whose data will be used for inference
- **excit_wfm** (*array-like, optional. default = None*) – Waveform against which the raw and filtered signals will be plotted. This waveform can be a fraction of the length of a single pixel's data. For example, in the case of G-mode, where a single scan line is yet to be broken down into pixels, the excitation waveform for a single pixel can br provided to automatically break the raw and filtered responses also into chunks of the same size.

Returns

Return type fig, axes

use_partial_computation (*h5_partial_group=None*)
Extracts the necessary parameters from the provided h5 group to resume computation

Parameters **h5_partial_group** (*h5py.Datagroup object*) – Datagroup containing partially computed results

write_condensed = None
Remember that the default number of pixels corresponds to only the raw data that can be held in memory In the case of signal filtering, the datasets that will occupy space are: 1. Raw, 2. filtered (real + freq space copies), 3. Condensed (substantially lesser space) The actual scaling of memory depends on options:

class ClusterTree (*linkage_pairing*, *labels*, *distances=None*, *centroids=None*)
Creates a tree representation from the provided linkage pairing. Useful for clustering

Parameters

- **linkage_pairing** (*2D unsigned int numpy array or list*) – Linkage pairing that describes a tree structure. The matrix should result in a single tree apex.
- **labels** (*1D unsigned int numpy array or list*) – Labels assigned to each of the positions in the main dataset. Eg. Labels from clustering
- **distances** (*(Optional) 1D numpy float array or list*) – Distances between clusters
- **centroids** (*(Optional) 2D numpy array*) – Mean responses for each of the clusters. These will be propagated up

centroids = None
this list maintains pointers to the nodes pertaining to that cluster id for quick look-ups By default this lookup table just contains the number indices of these clusters. They will be replaced with node objects as and when the objects are created

labels = None
the labels for the leaf nodes need to be calculated manually from the provided labels Populate the lowest level nodes / leaves first:

pycroscopy.viz

Pycroscopy's visualization module

Submodules

be_viz_utils	Created on Thu Jan 05 13:29:12 2017
cluster_utils	
image_cleaning_utils	

pycroscopy.viz.be_viz_utils

Created on Thu Jan 05 13:29:12 2017

@author: Chris R. Smith, Suhas Somnath

Functions

jupyter_visualize_be_spectrograms(pc_main[, ...])	Jupyer notebook ONLY function.
jupyter_visualize_beps_loops(...[, ...])	Interactive plotting of the BE Loops
jupyter_visualize_beps_sho(pc_sho_dset, ...)	Jupyer notebook ONLY function.
jupyter_visualize_loop_sho_raw_comparison(...)	**param h5_loop_parameters**
jupyter_visualize_parameter_maps(...[, cmap])	Interactive plot of the spatial maps of the loop parameters for all cycles.

Continued on next page

Table 31 – continued from previous page

plot_1d_spectrum(data_vec, freq, title, **kwargs)	Plots the Step averaged BE response
plot_2d_spectrogram(mean_spectrogram, freq)	Plots the position averaged spectrogram
plot_histograms(p_hist, p_hbins, title[, ...])	Plots the position averaged spectrogram
plot_loop_guess_fit(vdc, ds_proj_loops, ...)	Plots the loop guess, fit, source projected loops for a single cycle
plot_loop_sho_raw_comparison(h5_loop_parameters)	**param h5_loop_parameters** Dataset containing the loop parameters
visualize_sho_results(h5_main[, save_plots, ...])	Plots some loops, amplitude, phase maps for BE-Line and BEPS datasets.

jupyter_visualize_be_spectrograms(*pc_main*, *cmap=None*)

Jupyer notebook ONLY function. Sets up a simple visualzier for visualizing raw BE data. Sliders for position indices can be used to visualize BE spectrograms (frequency, UDVS step). In the case of 2 spatial dimensions, a spatial map will be provided as well

Parameters

- **pc_main** (*USIDataset*) – Raw Band Excitation dataset
- **cmap** (*String, or matplotlib.colors.LinearSegmentedColormap object (Optional)*) – Requested color map

jupyter_visualize_beps_loops(*h5_projected_loops*, *h5_loop_guess*, *h5_loop_fit*, *step_chan='DC_Offset'*, *cmap=None*)

Interactive plotting of the BE Loops

Parameters

- **h5_projected_loops** (*h5py.Dataset*) – Dataset holding the loop projections
- **h5_loop_guess** (*h5py.Dataset*) – Dataset holding the loop guesses
- **h5_loop_fit** (*h5py.Dataset*) – Dataset holding the loop fits
- **step_chan** (*str, optional*) – The name of the Spectroscopic dimension to plot versus. Needs testing. Default 'DC_Offset'
- **cmap** (*String, or matplotlib.colors.LinearSegmentedColormap object (Optional)*) – Requested color map

Returns

Return type None

jupyter_visualize_beps_sho(*pc_sho_dset*, *step_chan*, *resp_func=None*, *resp_label='Response'*, *cmap=None*)

Jupyer notebook ONLY function. Sets up an interactive visualizer for viewing SHO fitted BEPS data. Currently, this is limited to DC and AC spectroscopy datasets.

Parameters

- **pc_sho_dset** (*USIDataset*) – dataset to be plotted
- **step_chan** (*string / unicode*) – Name of the channel that forms the primary spectroscopic axis (eg - DC offset)
- **resp_func** (*function (optional)*) – Function to apply to the spectroscopic data. Currently, DC spectroscopy uses A*cos(phi) and AC spectroscopy uses A
- **resp_label** (*string / unicode (optional)*) – Label for the response (y) axis.

- **cmap** (*String, or matplotlib.colors.LinearSegmentedColormap object (Optional)*) – Requested color map

jupyter_visualize_loop_sho_raw_comparison(*h5_loop_parameters*, *cmap=None*)

Parameters

- **h5_loop_parameters** –
- **cmap** (*str or matplotlib.colors.Colormap*) –

jupyter_visualize_parameter_maps(*h5_loop_parameters*, *cmap=None*, ***kwargs*)

Interactive plot of the spatial maps of the loop parameters for all cycles.

Parameters

- **h5_loop_parameters** (*h5py.Dataset*) – The dataset containing the loop parameters to be visualized
- **cmap** (*str or matplotlib.colors.Colormap*) –

Returns

Return type None

plot_1d_spectrum(*data_vec*, *freq*, *title*, ***kwargs*)

Plots the Step averaged BE response

Parameters

- **data_vec** (*1D numpy array*) – Response of one BE pulse
- **freq** (*1D numpy array*) – BE frequency that serves as the X axis of the plot
- **title** (*String*) – Plot group name

Returns

- **fig** (*Matplotlib.pyplot figure*) – Figure handle
- **axes** (*Matplotlib.pyplot axis*) – Axis handle

plot_2d_spectrogram(*mean_spectrogram*, *freq*, *title=None*, ***kwargs*)

Plots the position averaged spectrogram

Parameters

- **mean_spectrogram** (*2D numpy complex array*) – Means spectrogram arranged as [frequency, UDVS step]
- **freq** (*1D numpy float array*) – BE frequency that serves as the X axes of the plot
- **title** (*str, optional*) – Plot group name

Returns

- **fig** (*Matplotlib.pyplot figure*) – Figure handle
- **axes** (*Matplotlib.pyplot axes*) – Axis handle

plot_histograms(*p_hist*, *p_hbins*, *title*, *figure_path=None*)

Plots the position averaged spectrogram

Parameters

- **p_hist** (*2D numpy array*) – histogram data arranged as [physical quantity, frequency bin]
- **p_hbins** (*1D numpy array*) – BE frequency that serves as the X axis of the plot

- **title** (*String*) – Plot group name
- **figure_path** (*String / Unicode*) – Absolute path of the file to write the figure to

Returns **fig** – Figure handle

Return type Matplotlib.pyplot figure

plot_loop_guess_fit(*vdc*, *ds_proj_loops*, *ds_guess*, *ds_fit*, *title=''*)

Plots the loop guess, fit, source projected loops for a single cycle

Parameters

- **– 1D float numpy array** (*vdc*) – DC offset vector (unshifted)
- **– 2D numpy array** (*ds_proj_loops*) – Projected loops arranged as [position, vdc]
- **– 1D compound numpy array** (*ds_fit*) – Loop guesses arranged as [position]
- **– 1D compound numpy array** – Loop fits arranged as [position]
- **– (Optional) String / unicode** (*title*) – Title for the figure

Returns

- *fig - matplotlib.pyplot.figure object* – Figure handle
- *axes - 2D array of matplotlib.pyplot.axis handles* – handles to axes in the 2d figure

plot_loop_sho_raw_comparison(*h5_loop_parameters, selected_loop_parm=None, selected_loop_cycle=0, selected_loop_pos=[0, 0], selected_step=0, tick_font_size=14, cmap='viridis', step_chan='DC_Offset'*)

Parameters

- **h5_loop_parameters** (*h5py.Dataset*) – Dataset containing the loop parameters
- **selected_loop_parm** (*str*) – The initial loop parameter to be plotted
- **selected_loop_cycle** (*int*) – The initial loop cycle to be plotted
- **selected_loop_pos** (*array-like of two ints*) – The initial position to be plotted
- **selected_step** (*int*) – The initial bias step to be plotted
- **tick_font_size** (*int*) – Font size for the axes tick labels
- **cmap** (*str or matplotlib.colors.Colormap*) – Colormap to be used in plotting the parameter map
- **step_chan** (*str*) – Name of spectral dimension loops were fit over

Returns

Return type None

visualize_sho_results(*h5_main*, *save_plots=True*, *show_plots=True*, *cmap=None*)

Plots some loops, amplitude, phase maps for BE-Line and BEPS datasets.

Note: The file MUST contain SHO fit gusses at the very least

Parameters

- **h5_main** (*HDF5 Dataset*) – dataset to be plotted
- **save_plots** (*(Optional) Boolean*) – Whether or not to save plots to files in the same directory as the h5 file

- **show_plots** (*(Optional) Boolean*) – Whether or not to display the plots on the screen
- **cmap** (*String, or matplotlib.colors.LinearSegmentedColormap object (Optional)*) – Requested color map

Returns

Return type None

pycroscopy.viz.cluster_utils

Functions

plot_cluster_centroids(centroids, x_vec[, ...])	**param centroids** 2D array. Centroids of clusters
plot_cluster_dendrogram(label_mat, e_vals, ...)	Creates and plots the dendrograms for the given label_mat and eigenvalues
plot_cluster_h5_group(h5_group[, ...])	Plots the cluster labels and mean response for each cluster
plot_cluster_labels(labels_mat[, ...])	Plots the cluster labels

plot_cluster_centroids(*centroids*, *x_vec*, *legend_mode=1*, *x_label=None*, *y_label=None*, *title=None*, *axis=None*, *overlayed=True*, *amp_units=None*, ***kwargs*)

Parameters

- **centroids** (*numpy.ndarray*) – 2D array. Centroids of clusters
- **x_vec** (*numpy.ndarray*) – 1D array. Vector against which the curves are plotted
- **legend_mode** (*int, optional. default = 1*) –

  Appearance of legend: 0 - inside the plot 1 - outside the plot on the right else - colorbar instead of legend
- **x_label** (*str, optional, default = None*) – Label for x axis
- **y_label** (*str, optional, default = None*) – Label for y axis
- **title** (*str, optional, default = None*) – Title for the plot
- **axis** (*matplotlib.axes.Axes object, optional.*) – Axis to plot this image onto. Will create a new figure by default or will use this axis object to plot into
- **overlayed** (*bool, optional*) – If True - all curves will be plotted overlayed on a single plot. Else, curves will be plotted separately
- **amp_units** (*str, optional*) – Units for amplitude
- **kwargs** (*dict*) – will be passed on to plot_line_family(), plot_complex_spectra, plot_curves

Returns

Return type fig, axes

plot_cluster_dendrogram(*label_mat*, *e_vals*, *num_comp*, *num_cluster*, *mode='Full'*, *last=None*, *sort_type='distance'*, *sort_mode=True*)

Creates and plots the dendrograms for the given label_mat and eigenvalues

Parameters

- **label_mat** (*2D real numpy array*) – structured as [rows, cols], from KMeans clustering
- **e_vals** (*3D real numpy array of eigenvalues*) – structured as [component, rows, cols]
- **num_comp** (*int*) – Number of components used to make eigenvalues
- **num_cluster** (*int*) – Number of cluster used to make the label_mat
- **mode** (*str, optional*) – How should the dendrograms be created. "Full" – use all clusters when creating the dendrograms "Truncated" – stop showing clusters after 'last'
- **last** (*int, optional - should be provided when using "Truncated"*) – How many merged clusters should be shown when using "Truncated" mode
- **sort_type** (*{'count', 'distance'}, optional*) – What type of sorting should be used when plotting the dendrograms. Options are: count - Uses the count_sort from scipy.cluster.hierachy.dendrogram distance - Uses the distance_sort from scipy.cluster.hierachy.dendrogram
- **sort_mode** (*{False, True, 'ascending', 'descending'}, optional*) – For the chosen sort_type, which mode should be used. False - Does no sorting 'ascending' or True - The child with the minimum of the chosen sort parameter is plotted first 'descending' - The child with the maximum of the chosen sort parameter is plotted first

Returns **fig** – Figure containing the dendrogram

Return type matplotlib.pyplot Figure object

plot_cluster_h5_group(*h5_group*, *labels_kwargs=None*, *centroids_kwargs=None*)

Plots the cluster labels and mean response for each cluster

Parameters

- **h5_group** (*h5py.Datagroup object*) – H5 group containing the labels and mean response
- **labels_kwargs** (*dict, optional*) – keyword arguments for the labels plot. NOT enabled yet.
- **centroids_kwargs** (*dict, optional*) – keyword arguments for the centroids plot. NOT enabled yet.

Returns

- **fig_labels** (*figure handle*) – Figure containing the labels
- **fig_centroids** (*figure handle*) – Figure containing the centroids

plot_cluster_labels(*labels_mat*, *num_clusters=None*, *x_label=None*, *y_label=None*, *title=None*, *axis=None*, ***kwargs*)

Plots the cluster labels

Parameters

- **labels_mat** (*numpy.ndarray*) – 1D or 2D unsigned integer array containing the labels of the clusters

- **num_clusters** (*int, optional*) – Number of clusters
- **x_label** (*str, optional*) – Label for x axis
- **y_label** (*str, optional*) – Label for y axis
- **title** (*str, optional*) – Title for the plot
- **axis** (*matplotlib.axes.Axes object, optional.*) – Axis to plot this image onto. Will create a new figure by default or will use this axis object to plot into
- **kwargs** (*dict*) – will be passed on to plot() or plot_map()

Returns

- **fig** (*matplotlib.pyplot.Figure object*) – figure object
- **axis** (*matplotlib.Axes object*) – axis object

pycroscopy.viz.image_cleaning_utils

Functions

plot_image_cleaning_results(raw_image, ...)	**param raw_image**

plot_image_cleaning_results(*raw_image*, *clean_image*, *stdevs=2*, *heading='Image Cleaning Results'*, *fig_mult=(4, 4)*, *fig_args={}*, ***kwargs*)

Parameters

- **raw_image** –
- **clean_image** –
- **stdevs** –
- **fig_mult** –
- **fig_args** –
- **heading** –

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

E

F

G

I

J

L

M

N

O

P

R

S

T

U

V

W