pyUSID Documentation

Release 0.0.4

Suhas Somnath, Chris R. Smith, Stephen Jesse

Jul 31, 2018

CONTENTS

Framework for storing, visualizing, and processing Universal Spectroscopic and Imaging Data (USID)

Jump to our GitHub project page

CHAPTER

ONE

DOCUMENTATION INDEX

—

1.1 pyUSID

Python framework for storing, visualizing, and processing Universal Spectroscopy and Imaging Data (USID)

1.1.1 What?

- The USID model:
  - 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.
  - is just a definition or a blueprint rather than something tangible and readily usable.
- pyUSID is a python package that currently provides three pieces of functionality:
  1. **io**: Primarily, it enables the storage and access of USID in **hierarchical data format** (HDF5) files (referred to as h5USID files) using python
  2. **viz**: It has handy tools for visualizing USID and general scientific data
  3. **processing**: It provides a framework for formulating scientific problems into computational problems. See pycroscopy - a sister project that uses pyUSID for analysis of microscopy data.
- pyUSID uses a data-centric approach wherein the raw data collected from the instrument, results from analysis and processing routines are all written to the same h5USID file for traceability, reproducibility, and provenance.
- Just as scipy uses numpy underneath, scientific packages like **pycroscopy** use **pyUSID** for all file-handling, general plotting utilities and a data processing framework
- pyUSID uses popular packages such as numpy, h5py, joblib, matplotlib, etc. for most of the storage, computation, and visualization.
- See a high-level overview of pyUSID in this presentation
- Jump to our GitHub project

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. 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*

5. 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 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. Please visit our credits and acknowledgements page for more information.
- 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.

1.2 Getting Started

- Follow these instructions to install pyUSID
- We have compiled a list of handy tutorials on basic / prerequisite topics such as programming in python, hdf5 handling, etc.
- See our examples to get started on using and writing your own pyUSID functions.
  - Please see this pyUSID tutorial for beginners based on the examples on this project.
- Details regarding the definition, implementation, and guidelines for Universal Spectroscopy and Imaging Data (USID) and its implementation in HDF5 (h5USID) are available in this document.
- Please see our document on the organization of pyUSID to find out more on what is where and why.

- If you are interested in contributing your code to pyUSID, please look at our guidelines
- We also have a handy document for converting your matlab code to python.
- If you need detailed documentation on all our classes, functions, etc., please visit our API
- Please get in touch if you would like to use USID and pyUSID for other new or mature scientific packages.
- 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.

1.2.1 Guide for python novices

For the python novices by a python novice - **Nick Mostovych, Brown University**

1. Learn about the philosophy, purpose, etc. of pyUSID.
2. Get an idea of the different resources available by reading the getting started section
3. Watch the video on installing Anaconda from the Tutorials on Basics page
4. Follow instructions on the installation page to install Anaconda.
5. Watch the video tutorial from the `Jupyter Notebooks` section in the Tutorials on Basics page
6. Read the whole Tutorial on Basics page. Do NOT proceed unless you are familiar with basic python programming and usage.
7. Read the document on the USID model and h5USID. This is very important and highlights the advantages of using USID. New users should not jump to the examples until they have a good understanding of the data format.
8. Depending on your needs, go through the recommended sequence of tutorials and examples

1.2.2 Tips and pitfalls

For the python novices by a python novice - **Nick Mostovych, Brown University**

- Documentation and examples on this website are for the latest version of pyUSID. If something does not work as shown on this website, chances are that you may be using an older version of pyUSID. Follow the instructions to update pyUSID to the latest version
- pyUSID has excellent documentation (+ examples too) for all functions. If you are ever confused with the usage of a function or class and you can get help in numerous ways:
  - If you are using jupyter notebooks, just hit the `Shift+Tab` keys after typing the name of your function. See this quick video for a demo. E.g. - type `px.USIDataset(`. Hit `Shift+Tab` twice or four times. You should be able to see the documentation for the class / function to learn how to supply inputs / extract outputs
  - Use the search function and reference the source code in the API section for detailed comments. Most detailed questions are answered there.
- Use the USIDataset everywhere possible to simplify your tasks.
- Many functions in pyUSID have a `verbose` keyword argument that can be set to `True` to get detailed print logs of intermediate steps in the function. This is **very** handy for debugging code

If there are tips or pitfalls you would like to add to this list, please write to us

1.3 Installation

1.3.1 Preparing for pyUSID

PyUSID requires many commonly used scientific and numeric python packages such as numpy, h5py 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

- Development on pyUSID is on python 3.5+. Nonetheless, we will do our best to ensure continued compatibility with python 2. Please raise an issue if you find a bug.
- We 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 pyUSID

Once the appropriate Anaconda distribution has been successfully installed, pyUSID can be installed easily from the python package index (pypi) via pip.

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 pyUSID
```

conda installation

pyUSID is a pure python package and is currently only available via pip

Installing from a specific branch (advanced users ONLY)

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

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

```
conda install git
```

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

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

1.3.3 Updating pyUSID

We recommend periodically updating your conda / anaconda distribution. Please see these instructions to update anaconda.

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

```
pip install -U --no-deps pyUSID
```

If it does not work try reinstalling the package:

```
pip uninstall pyUSID
pip install pyUSID
```

1.3.4 Other software

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

1.4 Tutorials on Basics

Here are a list of excellent tutorials from other websites and sources that describe some of the many important topics on reading, using / running and writing code:

Contents

1.4.1 Python and packages

There are several concepts such as file operations, parallel computing, etc. that are heavily used and applied in pyUSID. Most of these concepts are realized using add-ons or packages in python. Here is a compilation of useful tutorials:

Python

The following tutorials go over the basics of python programming:

- Official Python tutorial
- The Hitchhiker guide to Python
- Introduction to programming in Python 3
- O'Riley has a nice book on Python too.
- A nice guide on intermediate Python

HDF5 and h5py

pyUSID uses the h5py python package to store data in hierarchical data format (HDF5) files. Given that pyUSID is designed to be file-centric, we highly recommend learning more about HDF5 and h5py:

- Basics of HDF5 (especially the last three tutorials)
- Quick start to h5py
- Another tutorial on HDF5 and h5py
- The O-Reilly book where we learnt h5py

1.4.2 Installing software

python

Anaconda is a popular source for python which also comes with a large number of popular scientific python packages that are all correctly compiled and installed in one go. Tutorial for installing Anaconda (Python + all necessary packages)

python packages

Two popular methods for installing packages in python are:

- **pip**:
  - included with basic python and standard on Linux and Mac OS
  - Works great for installing pure python and other simple packages
- **conda**
  - included with Anaconda installation
  - Ideally suited for installing packages that have complex dependencies
- Here's a nice tutorial on installing packages using both pip and conda

Updating packages

Following these instructions, open a terminal or the command prompt (Windows) and type:

```
conda update conda
conda update anaconda
```

Note that you could use the following line instead of or in addition to `conda update anaconda` but it can lead to incompatible package versions

```
conda update --all
```

Note that this does **not** update python itself.

Upgrading python

Follow these instructions to upgrade python using conda to the latest or specific version

1.4.3 Writing code

Text Editors

These software often do not have any advanced features found in IDEs such as syntax highlighting, real-time code-checking etc. but are simple, and most importantly, open files quickly. Here are some excellent text editors for each class of operating system:

- Mac OS - Atom
- Linux - gEdit
- Windows - Notepad++

Integrated Development Environments (IDE)

These applications often come with a built-in text editor, code management capabilities, a python console, a terminal, integration with software repositories, etc. that make them ideal for executing and developing code. We only recommend two IDEs at this point: Spyder for users, PyCharm for developers. Both of these work in Linux, Mac OS, and Windows.

- Spyder is a great IDE that is simple and will be immediately familiar for users of Matlab.
  - Basics of Spyder
  - Python with Spyder - this was written with Python 2.7 in mind, but most concepts will still apply
- Pycharm
  - Official PyCharm Tutorial from Jetbrains

Jupyter Notebooks

These are interactive documents containing live cells with code, equations, visualizations, and narrative text. The interactive nature of the document makes Jupyter notebooks an ideal medium for conveying information and a narrative. These documents are neither text editors nor IDEs and are a separate category.

- Notebook basics

- Video tutorial
- Another video overview.

1.4.4 Software development basics

This section is mainly focused on the other tools that are mainly necessary for those interested in developing their own code and possibly contributing back to pyUSID.

Environments

Environments allow users to set up and segregate software sandboxes. For example, one could set up separate environments in python 2 and 3 to ensure that a certain desired code works in both python 2 and 3. For python users, there are two main and popular modes of creating and managing environments - **virtual environments** and **conda environments**.

- **Virtual environment**
  - Basic python ships with virtual enviroments. Anaconda is not required for this
  - How to use venv
- **Conda environments**
  - Basics of Conda
  - How to manage environments in conda
  - Managing Python Environments with Conda

Version control

Version control is a tool used for managing changes in code over time. It lifts the burden of having to check for changes line-by-line when multiple people are working on the same project. For example, pyUSID uses Git, the most popular version control software (VCS) for tracking changes etc. By default, git typically only comes with a command-line interface. However, there are several software packages that provide a graphical user interface on top of git. One other major benefit of using an IDE over jupyter or a text editor is that (some) IDEs come with excellent integration with VCS like Git. Here are a collection of useful resources to get you started on git:

- Tutorial on the basics of git
- Our favorite git client - GitKraken
- Our favorite IDE with excellent integration with Git: PyCharm
- Our own guide to setting up and using git with PyCharm

1.5 Package Organization

1.5.1 Sub-packages

The package structure is simple, with 3 main modules:

1. `io`: utilities that simplify the storage and accessing of data stored in h5USID files
2. `processing`: utilities and classes that support the piecewise (parallel) processing of arbitrarily large datasets

3. `viz`: plotting utilities and jupyter widgets that simplify common scientific visualization problems

`io`

- `hdf_utils` - Utilities for greatly simplifying reading and writing to h5USID files.
- `write_utils` - Utilities that assist in writing to HDF5 files
- `dtype_utils` - Utilities for data transformation (to and from real-valued, complex-valued, and compound-valued data arrays) and validation operations
- `io_utils` - Utilities for simplifying common computational, communication, and string formatting operations
- `USIDataset` - extends h5py.Dataset. We expect that users will use this class at every opportunity in order to simplify common operations on datasets such as interactive visualization in jupyter notebooks, slicing by the dataset's N-dimensional form and simplified access to supporting information about the dataset.
- `Translator` - An abstract class that provides the blueprint for other Translator classes to extract data and meta-data from other raw-data files and write them into h5USID files
- `ImageTranslator` - Translates data in common image file formats such as .png and .tiff to a h5USID file
- `NumpyTranslator` - A generic translator that simplifies writing of a dataset in memory into a h5USID file

`processing`

- `Process` - Modularizes, formalizes, and simplifies robust data processing
- `parallel_compute()` - Highly simplified one-line call to perform parallel computing on a data array

`viz`

- `plot_utils` - utilities to simplify common scientific tasks such as plotting a set of curves within the same or separate plots, plotting a set of 2D images in a grid, custom color-bars, etc.
- `jupyter_utils` - utilities to enable interactive visualization on generic 4D datasets within jupyter notebooks

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

1.6.1 Guides to pyUSID

- The documents below should be sufficient for you to learn how to use and even write code for pyUSID.
- If you haven't already, please go through the external tutorials on python, scientific data analysis in python, etc.
- All the documents listed below will assume that you understand the **Universal Spectroscopy and Imaging Data (USID)** model. Please read that document **before** beginning

To learn **how to use pyUSID**, go through the following documents in the recommended order:

1. Primer to h5py
2. The USIDataset
3. Data translation and the NumpyTranslator
4. Plot utilities (not necessary to learn pyUSID but we have several functions you are likely to find handy)
5. Utilities to read USID HDF5 files (a.k.a. **h5USID** files)
6. Data type manipulation utilities
7. Parallel computation

To learn **how to write to h5USID files, write data processing classes, or adding functionality to pyUSID**, go through these additional documents in the recommended order:

8. Utilities that assist in writing data
9. Utilities to write h5USID files
10. Formalizing Data Processing
11. Input / Output / Computing utilities
12. Guidelines for contributing code

Please get in touch with us if there are topics that you think need to be clarified / expanded / deserve new tutorials, or if you find any errors.

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

01. Primer to HDF5 and h5py

Suhas Somnath

4/18/2018

This document serves as a quick primer to HDF5 files and the h5py package used for reading and writing to such files

Introduction

We create and consume digital information stored in various file formats on a daily basis such as news presented in HTML files, scientific journal articles in PDF files, tabular data in XLSX spreadsheets and so on. Commercially available scientific instruments generate data in a variety of, typically proprietary, file formats. The proprietary nature of the data impedes scientific research of individual researchers and the collaboration within the scientific community at large. Hence, pycroscopy stores all relevant information including the measurement data, metadata etc. in the most popular file format for scientific data - Hierarchical Data Format (HDF5) files.

HDF5 is a remarkably straightforward file format to understand since it mimics the familiar folders and files paradigm exposed to users by all operating systems such as Windows, Mac OS, Linux, etc. HDF5 files can contain:

- `Datasets` - similar to spreadsheets and text files with tabular data.
- `Groups` - similar to folders in a regular file system
- `Attributes` - small metadata that provide additional information about the Group or Dataset they are attached to.
- other advanced features such as hard links, soft links, object and region references, etc.

h5py is the official software package for reading and writing to HDF5 files in python. Consequently, Pycroscopy relies entirely on h5py for all file related operations. While there are several high-level functions that simplify the reading and writing of Pycroscopy stylized data, it is still crucial that the users of Pycroscopy understand the basics of HDF5 files and are familiar with the basic functions in h5py. There are several tutorials available elsewhere to explain h5py in great detail. This document serves as a quick primer to the basics of interacting with HDF5 files via h5py.

Import all necessary packages

For this primer, we only need some very basic packages, all of which come with the standard Anaconda distribution:

- `os` - to manipulate and remove files
- `numpy` - for basic numerical work
- `h5py` - the package that will be the focus of this primer

```
from __future__ import print_function, division, unicode_literals
import os
import numpy as np
import h5py
```

Creating a HDF5 files using h5py is similar to the process of creating a conventional text file using python. The File class of h5py requires the path for the desired file with a .h5, .hdf5, or similar extension.

```
h5_path = 'hdf5_primer.h5'
h5_file = h5py.File('hdf5_primer.h5')
print(h5_file)
```

Out:

```
<HDF5 file "hdf5_primer.h5" (mode r+)>
```

At this point, a file in the path specified by h5_path has been created and is now open for modification. The returned value - h5_file is necessary to perform other operations on the file including creating groups and datasets.

Groups

create_group()

We can use the `create_group()` function on an existing object such as the open file handle (`h5_file`) to create a group:

```
h5_group_1 = h5_file.create_group('Group_1')
print(h5_group_1)
```

Out:

```
<HDF5 group "/Group_1" (0 members)>
```

The output of the above print statement reveals that a group named `Group_1` was successfully created at location: '/' (which stands for the root of the file). Furthermore, this group contains 0 objects or members. .name ——- One can find the full / absolute path where this object is located from its `name` property:

```
print(h5_group_1.name)
```

Out:

```
/Group_1
```

Groups in Groups

Much like folders in a computer, these groups can themselves contain more groups and datasets.

Let us create a few more groups the same way. Except, let us create these groups within the newly created. To do this, we would need to call the `create_group()` function on the h5_group_1 object and not the h5_file object. Doing the latter would result in groups created under the file at the same level as `Group_1` instead of inside `Group_1`.

```
h5_group_1_1 = h5_group_1.create_group('Group_1_1')
h5_group_1_2 = h5_group_1.create_group('Group_1_2')
```

Now, when we print h5_group, it will reveal that we have two objects - the two groups we just created:

```
print(h5_group_1)
```

Out:

```
<HDF5 group "/Group_1" (2 members)>
```

Lets see what a similar print of one of the newly created groups looks like:

```
print(h5_group_1_1)
```

Out:

```
<HDF5 group "/Group_1/Group_1_1" (0 members)>
```

The above print statement shows that this group named `Group_1_1` exists at a path: `"/Group_1/Group_1_1"`. In other words, this is similar to a folder contained inside another folder.

.parent

The hierarchical nature of HDF5 allows us to access datasets and groups using relationships or paths. For example, every HDF5 object has a parent. In the case of 'Group_1' - its parent is the root or h5_file itself. Similarly, the parent object of 'Group_1_1' is 'Group_1':

```
print('Parent of "Group_1" is {}'.format(h5_group_1.parent))
print('Parent of "Group_1_1" is {}'.format(h5_group_1_1.parent))
```

Out:

```
Parent of "Group_1" is <HDF5 group "/" (1 members)>
Parent of "Group_1_1" is <HDF5 group "/Group_1" (2 members)>
```

In fact the .parent of an object is an HDF5 object (either a HDF5 group or HDF5 File object). So we can check if the parent of the h5_group_1_1 variable is indeed the h5_group_1 variable:

```
print(h5_group_1_1.parent == h5_group_1)
```

Out:

```
True
```

Accessing H5 objects

Imagine a file or a folder on a computer that is several folders deep from where one is (e.g. - /Users/Joe/Documents/Projects/2018/pycroscopy).One could either reach the desired file or folder by opening one folder after another or directly by using a long path string. If you were at root (/), you would need to paste the entire path (absolute path) of the desired file - `/Users/Joe/Documents/Projects/2018/pycroscopy`. Alternatively, if you were in an intermediate directory (e.g. - `/Users/Joe/Documents/`), you would need to paste what is called the relative path (in this case - `Projects/2018/pycroscopy`) to get to the desired file.

In the same way, we can also access HDF5 objects either through `relative paths`, or `absolute paths`. Here are a few ways one could get to the group `Group_1_2`:

```
print(h5_file['/Group_1/Group_1_2'])
print(h5_group_1['Group_1_2'])
print(h5_group_1_1.parent['Group_1_2'])
print(h5_group_1_1.parent.parent['Group_1/Group_1_2'])
```

Out:

```
<HDF5 group "/Group_1/Group_1_2" (0 members)>
<HDF5 group "/Group_1/Group_1_2" (0 members)>
<HDF5 group "/Group_1/Group_1_2" (0 members)>
<HDF5 group "/Group_1/Group_1_2" (0 members)>
```

Now let us look at how one can iterate through the datasets and Groups present within a HDF5 group:

```
for item in h5_group_1:
    print(item)
```

Out:

```
Group_1_1
Group_1_2
```

.items()

Essentially, h5py group objects contain a dictionary of key-value pairs where they key is the name of the object and the value is a reference to the object itself.

What the above for loop does is it iterates only over the keys in this dictionary which are all strings. In order to get the actual dataset object itself, we would need to use the aforementioned addressing techniques to get the actual Group objects.

Let us see how we would then try to find the object for the group named ‘Group_1_2’:

```
for key, value in h5_group_1.items():
    if key == 'Group_1_2':
        print('Found the desired object: {}'.format(value))
```

Out:

```
Found the desired object: <HDF5 group "/Group_1/Group_1_2" (0 members)>
```

Datasets

create_dataset()

We can create a dataset within `Group_1` using a function that is similar to `create_group()`, called `create_dataset()`. Unlike create_group() which just takes the path of the desired group as an input, `create_dataset()` is highly customizable and flexible.

In our experience, there are three modes of creating datasets that are highly relevant for scientific applications:

- dataset with data at time of creation - where the data is already available at the time of creating the dataset
- empty dataset - when one knows the size of data but the entire data is not available
- resizable dataset - when one does not even know how large the data can be. *This case is rare*

Creating Dataset with available data:

Let as assume we want to store a simple greyscale (floating point values) image with 256 x 256 pixels. We would create and store the data as shown below. As the size of the dataset becomes very large, the precision with which the data is stored can significantly affect the size of the dataset and the file. Therefore, we recommend purposefully specifying the data-type (via the `dtype` keyword argument) during creation.

```
h5_simple_dataset = h5_group_1.create_dataset('Simple_Dataset',
                                              data=np.random.rand(256, 256),
                                              dtype=np.float32)
print(h5_simple_dataset)
```

Out:

```
<HDF5 dataset "Simple_Dataset": shape (256, 256), type "<f4">
```

Accessing data

We can access data contained in the dataset just like accessing a numpy array. For example, if we want the value at row `29` and column `167`, we would read it as:

```
print(h5_simple_dataset[29, 167])
```

Out:

```
0.047381084
```

Again, just as before, we can address this dataset in many ways:

```
print(h5_group_1['Simple_Dataset'])
print(h5_file['/Group_1/Simple_Dataset'])
```

Out:

```
<HDF5 dataset "Simple_Dataset": shape (256, 256), type "<f4">
<HDF5 dataset "Simple_Dataset": shape (256, 256), type "<f4">
```

Creating (potentially large) empty datasets:

In certain situations, we know how much space to allocate for the final dataset but we may not have all the data at once. Alternatively, the dataset is so large that we cannot fit the entire data in the computer memory before writing to the HDF5 file. Another possible circumstance is when we have to read N files, each containing a small portion of the data and then write the contents into each slot in the HDF5 dataset.

For example, assume that we have 128 files each having 1D spectra (amplitude + phase or complex value) of length 1024. Here is how one may create the HDF5 dataset to hold the data:

```
h5_empty_dataset = h5_group_1.create_dataset('Empty_Dataset',
                                             shape=(128, 1024),
                                             dtype=np.complex64)
print(h5_empty_dataset)
```

Out:

```
<HDF5 dataset "Empty_Dataset": shape (128, 1024), type "<c8">
```

Note that unlike before, this particular dataset is empty since we only allocated space, so we would be reading zeros when attempting to access data:

```
print(h5_empty_dataset[5, 102])
```

Out:

```
0j
```

populating with data

One could populate each chunk of the dataset just like filling in a numpy array:

```
h5_empty_dataset[0] = np.random.rand(1024) + 1j * np.random.rand(1024)
```

flush()

It is a good idea to ensure that this data is indeed committed to the file using regular flush() operations. There are chances where the data is still in the memory / buffer and not yet in the file if one does not flush():

```
h5_file.flush()
```

Creating resizeable datasets:

This solution is relevant to those situations where we only know how large each unit of data would be but we don't know the number of units. This is especially relevant when acquiring data from an instrument.

For example, if we were acquiring spectra of length 128 on a 1D grid of 256 locations, we may have created an empty 2D dataset of shape (265, 128) using the aforementioned function. The data was being collected ordinarily over the first 13 positions but a change in parameters resulted in spectra of length 175 instead. The data from the 14th positon cannot be stored in the empty array due to a size mismatch. Therefore, we would need to create another empty 256 x 175 dataset to hold the data. If changes in parameters cause 157 changes in spectra length, that would result in the creation of 157 datasets each with a whole lot of wasted space since datasets cannot be shrunk easily.

In such cases, it is easier just to create datasets that can expand one pixel at a time. For this specific example, one may want to create a 2D dataset of shape (1, 128) that could grow up to a maxshape of (256, 128) as shown below:

```
h5_expandable_dset = h5_group_1.create_dataset('Expandable_Dataset',
                                               shape=(1, 128),
                                               maxshape=(256, 128),
                                               dtype=np.float32)
print(h5_expandable_dset)
```

Out:

```
<HDF5 dataset "Expandable_Dataset": shape (1, 128), type "<f4">
```

Space has been allocated for the first pixel, so the data could be written in as:

```
h5_expandable_dset[0] = np.random.rand(128)
```

For the next pixel, we would need to expand the dataset before filling it in:

```
h5_expandable_dset.resize(h5_expandable_dset.shape[0] + 1, axis=0)
print(h5_expandable_dset)
```

Out:

```
<HDF5 dataset "Expandable_Dataset": shape (2, 128), type "<f4">
```

Notice how the dataset has increased in size in the first dimension allowing the second pixel to be stored. The second pixel's data would be stored in the same way as in the first pixel and the cycle of expand and populate-with-data would continue.

It is very important to note that there is a non-trivial storage overhead associated with each resize operation. In other words, a file containing this resizeable dataset that has been resized 255 times will certainly be larger than a similar file where the dataset space was pre-allocated and never expanded. Therefore this mode of creating datasets should used sparingly.

Attributes

- are metadata that can convey information that cannot be efficiently conveyed using Group or Dataset objects.
- are almost exactly like python dictionaries in that they have a key-value pairs.
- can be stored in either Group or Dataset objects.
- are not appropriate for storing large amounts of information. Consider datasets instead
- are best suited for things like experimental parameter such as beam intensity, scan rate, scan width, etc.

Writing

Storing attributes in objects is identical to appending to python dictionaries. Lets store some simple attributes in the group named 'Group_1':

```
h5_simple_dataset.attrs['single_num'] = 36.23
h5_simple_dataset.attrs.update({'list_of_nums': [1, 6.534, -65],
                                'single_string': 'hello'})
```

Reading

We would read the attributes just like we would treat a dictionary in python:

```
for key, val in h5_simple_dataset.attrs.items():
    print('{} : {}'.format(key, val))
```

Out:

```
single_num : 36.23
list_of_nums : [  1.      6.534 -65.    ]
single_string : hello
```

Lets read the attributes one by one and verify that we read what we wrote:

```
print('single_num: {}'.format(h5_simple_dataset.attrs['single_num'] == 36.23))
print('list_of_nums: {}'.format(np.all(h5_simple_dataset.attrs['list_of_nums'] == [1,␣
↪6.534, -65])))
print('single_string: {}'.format(h5_simple_dataset.attrs['single_string'] == 'hello'))
```

Out:

```
single_num: True
list_of_nums: True
single_string: True
```

Caveat

While the low-level attribute writing and reading does appear to work and is simple, it does not work for a list of strings in python 3. Hence the following line will not work and will cause problems.

```
h5_simple_dataset.attrs['list_of_strings'] = ['a', 'bc', 'def']
```

Instead, we recommend writing lists of strings by casting them as numpy arrays:

```
h5_simple_dataset.attrs['list_of_strings'] = np.array(['a', 'bc', 'def'], dtype='S')
```

In the same way, reading attributes that are lists of strings is also not straightforward:

```
print('list_of_strings: {}'.format(h5_simple_dataset.attrs['list_of_strings'] == ['a',
↪ 'bc', 'def']))
```

Out:

```
list_of_strings: False
```

A similar decoding step needs to be taken to extract the actual string values.

To avoid manual encoding and decoding of attributes (different strategies for different versions of python), we recommend:

- writing attributes using: `pycroscopy.hdf_utils.write_simple_attrs()`
- reading attributes using: `pycroscopy.hdf_utils.get_attr() or get_attributes()`

Both these functions work reliably and consistently across all python versions and fix this problem in h5py.

Besides strings and numbers, we tend to store references to datasets as attributes. Here is how one would link the empty dataset to the simple dataset:

```
h5_simple_dataset.attrs['Dataset_Reference'] = h5_empty_dataset.ref
print(h5_simple_dataset.attrs['Dataset_Reference'])
```

Out:

```
<HDF5 object reference>
```

Here is how one would get a handle to the actual dataset from the reference:

```
# Read the attribute how you normally would
h5_ref = h5_simple_dataset.attrs['Dataset_Reference']
# Get the handle to the actual dataset:
h5_dset = h5_file[h5_ref]
# Check if this object is indeed the empty dataset:
print(h5_empty_dataset == h5_dset)
```

Out:

```
True
```

Once we are done reading or manipulating an HDF5 file, we need to close it to avoid and potential damage:

```
h5_file.close()
os.remove(h5_path)
```

As mentioned in the beginning this is not meant to be a comprehensive overview of HDF5 or h5py, but rather just a quick overview of the important functionality we recommend everyone to be familiar with. We encourage you to read more about h5py and HDF5 if you are interested.

Total running time of the script: (0 minutes 0.009 seconds)

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

02. The USIDataset

Suhas Somnath

11/11/2017

This document illustrates how the pyUSID.USIDataset class substantially simplifies accessing information about, slicing, and visualizing N-dimensional Universal Spectroscopy and Imaging Data (USID) Main datasets

USID Main Datasets

According to the **Universal Spectroscopy and Imaging Data (USID)** model, all spatial dimensions are collapsed to a single dimension and, similarly, all spectroscopic dimensions are also collapsed to a single dimension. Thus, the data is stored as a two-dimensional (N x P) matrix with N spatial locations each with P spectroscopic data points.

This general and intuitive format allows imaging data from any instrument, measurement scheme, size, or dimensionality to be represented in the same way. Such an instrument independent data format enables a single set of analysis and processing functions to be reused for multiple image formats or modalities.

`Main datasets` are greater than the sum of their parts. They are more capable and information-packed than conventional datasets since they have (or are linked to) all the necessary information to describe a measured dataset. The additional information contained / linked by `Main datasets` includes:

- the recorded physical quantity
- units of the data
- names of the position and spectroscopic dimensions
- dimensionality of the data in its original N dimensional form etc.

USIDatasets = USID Main Datasets

Regardless, `Main datasets` are just concepts or blueprints and not concrete digital objects in a programming language or a file. `USIDatasets` are **tangible representations of Main datasets**. From an implementation perspective, the USIDataset class extends the `h5py.Dataset object`. In other words, USIDatasets have all the capabilities of standard HDF5 / h5py Dataset objects but are supercharged from a scientific perspective since they:

- are self-describing
- allow quick interactive visualization in Jupyter notebooks
- allow intuitive slicing of the N dimensional dataset
- and much much more.

While it is most certainly possible to access this information and enable these functionalities via the native `h5py` functionality, it can become tedious very quickly. In fact, a lot of the functionality of USIDataset comes from orchestration of multiple functions in `pyUSID.hdf_utils` outlined in other documents. The USIDataset class makes such necessary information and functionality easily accessible.

Since Main datasets are the hubs of information in a USID HDF5 file (**h5USID**), we expect that the majority of the data interrogation will happen via USIDatasets

Recommended pre-requisite reading

- USID data model
- Crash course on HDF5 and h5py
- Utilities for reading and writing h5USID files using pyUSID

Example scientific dataset

Before, we dive into the functionalities of USIDatasets we need to understand the dataset that will be used in this example. For this example, we will be working with a Band Excitation Polarization Switching (BEPS) dataset acquired from advanced atomic force microscopes. In the much simpler Band Excitation (BE) imaging datasets, a single spectra is acquired at each location in a two dimensional grid of spatial locations. Thus, BE imaging datasets have two position dimensions (X, Y) and one spectroscopic dimension (frequency - against which the spectra is recorded). The BEPS dataset used in this example has a spectra for each combination of three other parameters (DC offset, Field, and Cycle). Thus, this dataset has three new spectral dimensions in addition to the spectra itself. Hence, this dataset becomes a 2+4 = 6 dimensional dataset

Load all necessary packages

First, we need to load the necessary packages. Here are a list of packages, besides pyUSID, that will be used in this example:

- `h5py` - to open and close the file
- `wget` - to download the example data file
- `numpy` - for numerical operations on arrays in memory
- `matplotlib` - basic visualization of data

```
from __future__ import print_function, division, unicode_literals
import os
# Warning package in case something goes wrong
from warnings import warn
import subprocess
import sys


def install(package):
    subprocess.call([sys.executable, "-m", "pip", "install", package])
# Package for downloading online files:


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
import h5py
import numpy as np
import matplotlib.pyplot as plt
try:
    import pyUSID as usid
except ImportError:
    warn('pyUSID not found.  Will install with pip.')
    import pip
    install('pyUSID')
    import pyUSID as usid
```

Load the dataset

First, lets download example h5USID file from the pyUSID Github project:

```
url = 'https://raw.githubusercontent.com/pycroscopy/pyUSID/master/data/BEPS_small.h5'
h5_path = 'temp.h5'
_ = wget.download(url, h5_path, bar=None)

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

Out:

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

Next, lets open this HDF5 file in read-only mode. Note that opening the file does not cause the contents to be automatically loaded to memory. Instead, we are presented with objects that refer to specific HDF5 datasets, attributes or groups in the file

```
h5_path = 'temp.h5'
h5_f = h5py.File(h5_path, mode='r')
```

Here, `h5_f` is an active handle to the open file. Lets quickly look at the contents of this HDF5 file using a handy function in `pyUSID.hdf_utils` - `print_tree()`

```
print('Contents of the H5 file:')
usid.hdf_utils.print_tree(h5_f)
```

Out:

```
Contents of the H5 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
    ├ Raw_Data-SHO_Fit_000
      --------------------
      ├ Fit
      ├ Guess
      ├ Spectroscopic_Indices
      ├ Spectroscopic_Values
    ├ Spatially_Averaged_Plot_Group_000
      ---------------------------------
      ├ Bin_Frequencies
      ├ Mean_Spectrogram
      ├ Spectroscopic_Parameter
      ├ Step_Averaged_Response
    ├ Spatially_Averaged_Plot_Group_001
      ---------------------------------
      ├ Bin_Frequencies
      ├ Mean_Spectrogram
      ├ Spectroscopic_Parameter
      ├ Step_Averaged_Response
    ├ Spectroscopic_Indices
    ├ Spectroscopic_Values
    ├ UDVS
    ├ UDVS_Indices
```

For this example, we will only focus on the `Raw_Data` dataset which contains the 6D raw measurement data. First lets access the HDF5 dataset and check if it is a `Main` dataset in the first place:

```
h5_raw = h5_f['/Measurement_000/Channel_000/Raw_Data']
print(h5_raw)
print('h5_raw is a main dataset? {}'.format(usid.hdf_utils.check_if_main(h5_raw)))
```

Out:

```
<HDF5 dataset "Raw_Data": shape (25, 22272), type "<c8">
h5_raw is a main dataset? True
```

It turns out that this is indeed a Main dataset. Therefore, we can turn this in to a USIDataset without any problems.

Creating a USIDataset

All one needs for creating a USIDataset object is a Main dataset. Here is how we can supercharge h5_raw:

```
pd_raw = usid.USIDataset(h5_raw)
print(pd_raw)
```

Out:

```
<HDF5 dataset "Raw_Data": shape (25, 22272), type "<c8">
located at:
        /Measurement_000/Channel_000/Raw_Data
Data contains:
        Cantilever Vertical Deflection (V)
Data dimensions and original shape:
Position Dimensions:
        X - size: 5
        Y - size: 5
Spectroscopic Dimensions:
        Frequency - size: 87
        DC_Offset - size: 64
        Field - size: 2
        Cycle - size: 2
Data Type:
        complex64
```

Notice how easy it was to create a USIDataset object. Also, note how the USIDataset is much more informative in comparison with the conventional h5py.Dataset object.

USIDataset = Supercharged(h5py.Dataset)

Remember that USIDataset is just an extension of the h5py.Dataset object class. Therefore, both the `h5_raw` and `pd_raw` refer to the same object as the following equality test demonstrates. Except `pd_raw` knows about the `ancillary datasets` and other information which makes it a far more powerful object for you.

```
print(pd_raw == h5_raw)
```

Out:

```
True
```

Easier access to information

Since the USIDataset is aware and has handles to the supporting ancillary datasets, they can be accessed as properties of the object unlike HDF5 datasets. Note that these ancillary datasets can be accessed using functionality in pyUSID.hdf_utils as well. However, the USIDataset option is far easier.

Let us compare accessing the Spectroscopic Indices via the USIDataset and hdf_utils functionality:

```
h5_spec_inds_1 = pd_raw.h5_spec_inds
h5_spec_inds_2 = usid.hdf_utils.get_auxiliary_datasets(h5_raw, 'Spectroscopic_Indices
↪')[0]
print(h5_spec_inds_1 == h5_spec_inds_2)
```

Out:

```
True
```

In the same vein, it is also easy to access **string descriptors** of the ancillary datasets and the Main dataset. The `hdf_utils` alternatives to these operations / properties also exist and are discussed in an alternate document, but will not be discussed here for brevity.:

```
print('Desctiption of physical quantity in the Main dataset:')
print(pd_raw.data_descriptor)
print('Position Dimension names and sizes:')
for name, length in zip(pd_raw.pos_dim_labels, pd_raw.pos_dim_sizes):
    print('{} : {}'.format(name, length))
print('Spectroscopic Dimension names and sizes:')
for name, length in zip(pd_raw.spec_dim_labels, pd_raw.spec_dim_sizes):
    print('{} : {}'.format(name, length))
print('Position Dimensions:')
print(pd_raw.pos_dim_descriptors)
print('Spectroscopic Dimensions:')
print(pd_raw.spec_dim_descriptors)
```

Out:

```
Desctiption of physical quantity in the Main dataset:
Cantilever Vertical Deflection (V)
Position Dimension names and sizes:
X : 5
Y : 5
Spectroscopic Dimension names and sizes:
Frequency : 87
DC_Offset : 64
Field : 2
Cycle : 2
Position Dimensions:
['X (m)', 'Y (m)']
Spectroscopic Dimensions:
['Frequency (Hz)', 'DC_Offset (V)', 'Field ()', 'Cycle ()']
```

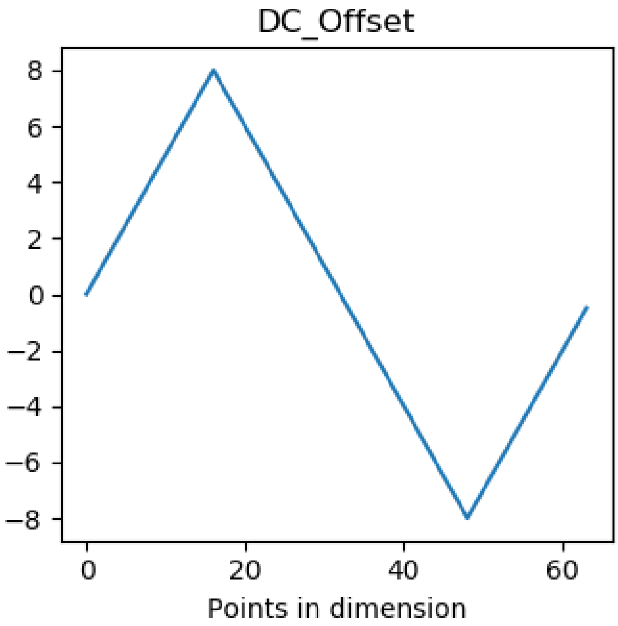
Values for each Dimension

When visualizing the data it is essential to plot the data against appropriate values on the X, Y, Z axes. The USIDataset object makes it very easy to access the values over which a dimension was varied using the `get_pos_values()`

and get_spec_values() functions. This functionality is enabled by the get_unit_values() function in pyUSID.hdf_utils.

For example, let us say we wanted to see how the DC_Offset dimension was varied, we could:

```
dim_name = 'DC_Offset'
dc_vec = pd_raw.get_spec_values(dim_name)
fig, axis = plt.subplots(figsize=(3.5, 3.5))
axis.plot(dc_vec)
axis.set_xlabel('Points in dimension')
axis.set_title(dim_name)
fig.tight_layout()
```



Reshaping to N dimensions

The USID model stores N dimensional datasets in a flattened 2D form of position x spectral values. It can become challenging to retrieve the data in its original N-dimensional form, especially for multidimensional datasets such as the one we are working on. Fortunately, all the information regarding the dimensionality of the dataset are contained in the spectral and position ancillary datasets. PycoDataset makes it remarkably easy to obtain the N dimensional form of a dataset:

```
ndim_form = pd_raw.get_n_dim_form()
print('Shape of the N dimensional form of the dataset:')
print(ndim_form.shape)
print('And these are the dimensions')
print(pd_raw.n_dim_labels)
```

Out:

```
Shape of the N dimensional form of the dataset:
(5, 5, 87, 64, 2, 2)
And these are the dimensions
['X', 'Y', 'Frequency', 'DC_Offset', 'Field', 'Cycle']
```

Slicing

It is often very challenging to grapple with multidimensional datasets such as the one in this example. It may not even be possible to load the entire dataset in its 2D or N dimensional form to memory if the dataset is several (or several hundred) gigabytes large. Slicing the 2D Main dataset can easily become confusing and frustrating. To solve this problem, USIDataset has a `slice()` function that efficiently loads the only the sliced data into memory and reshapes the data to an N dimensional form. Best of all, the slicing arguments can be provided in the actual N dimensional form!

For example, imagine that we cannot load the entire example dataset in its N dimensional form and then slice it. Lets try to get the spatial map for the following conditions without loading the entire dataset in its N dimensional form and then slicing it :

- 14th index of DC Offset
- 1st index of cycle
- 0th index of Field (remember Python is 0 based)
- 43rd index of Frequency

To get this, we would slice as:

```
spat_map_1, success = pd_raw.slice({'Frequency': 43, 'DC_Offset': 14, 'Field': 0,
↪'Cycle': 1})
```

As a verification, lets try to plot the same spatial map by slicing the N dimensional form we got earlier and compare it with what we got above:

```
spat_map_2 = np.squeeze(ndim_form[:, :, 43, 14, 0, 1])
print('2D slicing == ND slicing: {}'.format(np.allclose(spat_map_1, spat_map_2)))
```

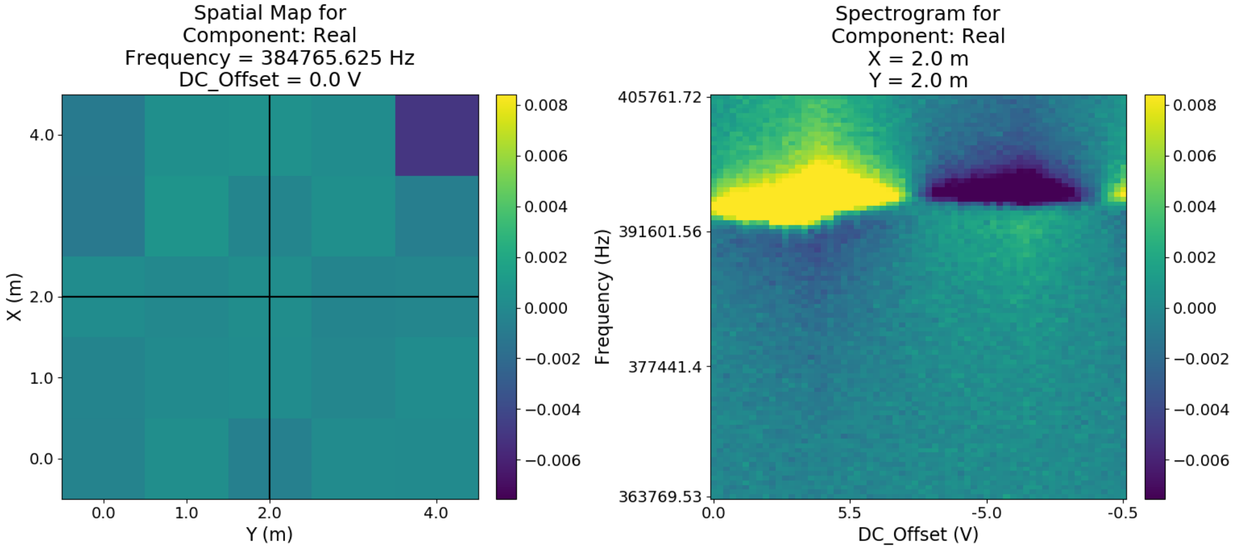
Out:

```
2D slicing == ND slicing: True
```

Interactive Visualization

USIDatasets also enable quick, interactive, and easy visualization of data up to 2 position and 2 spectroscopic dimensions (4D datasets). Since this particular example has 6 dimensions, we would need to slice two dimensions in order to visualize the remaining 4 dimensions. Note that this interactive visualization ONLY works on Jupyter Notebooks. This html file generated by a python script does not allow for interactive visualization and you may only see a set of static plots. We encourage you to click on the Download as Jupyter Notebook button below to try it out yourself.

```
pd_raw.visualize(slice_dict={'Field': 0, 'Cycle': 1});
```



Out:

```
interactive(children=(IntSlider(value=2, description='X', max=4), IntSlider(value=2,␣
↪description='Y', max=4), IntSlider(value=43, description='Frequency', max=86),␣
↪IntSlider(value=31, description='DC_Offset', max=63), Dropdown(description=
↪'component', options=('Real', 'Imaginary', 'Amplitude', 'Phase'), value='Real'),␣
↪Output()), _dom_classes=('widget-interact',))
```

Close and delete the h5_file

```
h5_f.close()
os.remove(h5_path)
```

Total running time of the script: (0 minutes 1.123 seconds)

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

03. Translation and the NumpyTranslator

Suhas Somnath

8/8/2017

This document illustrates an example of extracting data out of proprietary raw data files and writing the information into a **Universal Spectroscopy and Imaging Data (USID)** HDF5 file (referred to as a **h5USID** file) using the `pyUSID.NumpyTranslator`

Introduction

Before any data analysis, we need to access data stored in the raw file(s) generated by the microscope. Often, the data and parameters in these files are **not** straightforward to access. In certain cases, additional / dedicated software packages are necessary to access the data while in many other cases, it is possible to extract the necessary information from built-in **numpy** or similar python packages included with **anaconda**.

The USID model aims to make data access, storage, curation, etc. simply by storing the data along with all relevant parameters in a single file (HDF5 for now).

The process of copying data from the original format to **h5USID** files is called **Translation** and the classes available in pyUSID and children packages such as pycroscopy that perform these operation are called **Translators**

Simply put, so long as one has the metadata and the actual data extracted from the raw data file, the `pyUSID.NumpyTranslator` will correctly write the contents to a h5UID / HDF5 file. Note that the complexity or size of the raw data may necessitate a custom Translator class. However, the rough process of translation is the same regardless of the origin, complexity, or size of the raw data:

- Investigating how to open the proprietary raw data file
- Reading the metadata
- Extracting the data
- Writing to h5USID file

The goal of this document is to demonstrate how one would extract data and parameters from a Scanning Tunnelling Spectroscopy (STS) raw data file obtained from an Omicron Scanning Tunneling Microscope (STM) into a h5USID file.

While there is an AscTranslator available in our sister-package - `pycroscopy` that can translate these files in just a **single** line, we will intentionally assume that no such translator is available. Using a handful of useful functions in pyUSID, we will translate the files from the source **.asc** format to h5USID files in just a few lines.

The same methodology can be used to translate other data formats

Recommended pre-requisite reading

Before proceeding with this example, we recommend reading the previous documents to learn more about:

- USID model

Import all necessary packages

There are a few setup procedures that need to be followed before any code is written. In this step, we simply load a few python packages that will be necessary in the later steps.

```
# Ensure python 3 compatibility:
from __future__ import division, print_function, absolute_import, unicode_literals

# The package for accessing files in directories, etc.:
import os
import zipfile

# Warning package in case something goes wrong
from warnings import warn
import subprocess
import sys


def install(package):
    subprocess.call([sys.executable, "-m", "pip", "install", package])
```

(continues on next page)

(continued from previous page)

```
# Package for downloading online files:
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

# The mathematical computation package:
import numpy as np

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

# Packages for plotting:
import matplotlib.pyplot as plt

# Finally import pyUSID:
try:
    import pyUSID as usid
except ImportError:
    warn('pyUSID not found.  Will install with pip.')
    import pip
    install('pyUSID')
    import pyUSID as usid
```

Step 0. Procure the Raw Data file

```
# Download the compressed data file from Github:
url = 'https://raw.githubusercontent.com/pycroscopy/pyUSID/master/data/STS.zip'
zip_path = 'STS.zip'
if os.path.exists(zip_path):
    os.remove(zip_path)
_ = wget.download(url, zip_path, bar=None)

zip_path = os.path.abspath(zip_path)
# figure out the folder to unzip the zip file to
folder_path, _ = os.path.split(zip_path)
zip_ref = zipfile.ZipFile(zip_path, 'r')
# unzip the file
zip_ref.extractall(folder_path)
zip_ref.close()
# delete the zip file
os.remove(zip_path)

data_file_path = 'STS.asc'
```

Step 1. Exploring the Raw Data File

Inherently, one may not know how to read these `.asc` files. One option is to try and read the file as a text file one line at a time.

It turns out that these .asc files are effectively the standard ASCII text files.

Here is how we tested to see if the asc files could be interpreted as text files. Below, we read just the first 10 lines in the file

```
with open(data_file_path, 'r') as file_handle:
    for lin_ind in range(10):
        print(file_handle.readline())
```

Out:

```
# File Format = ASCII

# Created by SPIP 4.6.5.0 2016-09-22 13:32

# Original file: C:\Users\Administrator\AppData\Roaming\Omicron␣
↪NanoTechnology\MATRIX\default\Results\16-Sep-2016\I(V) TraceUp Tue Sep 20 09.17.08␣
↪2016 [14-1]  STM_Spectroscopy STM

# x-pixels = 100

# y-pixels = 100

# x-length = 29.7595

# y-length = 29.7595

# x-offset = -967.807

# y-offset = -781.441

# z-points = 500
```

Step 2. Loading the data

Now that we know that these files are simple text files, we can manually go through the file to find out which lines are important, at what lines the data starts etc. Manual investigation of such .asc files revealed that these files are always formatted in the same way. Also, they contain parameters in the first 403 lines and then contain data which is arranged as one pixel per row. STS experiments result in 3 dimensional datasets (X, Y, current). In other words, a 1D array of current data (as a function of excitation bias) is sampled at every location on a two dimensional grid of points on the sample. By knowing where the parameters are located and how the data is structured, it is possible to extract the necessary information from these files. Since we know that the data sizes (<200 MB) are much smaller than the physical memory of most computers, we can start by safely loading the contents of the entire file to memory

```
# Extracting the raw data into memory
file_handle = open(data_file_path, 'r')
string_lines = file_handle.readlines()
file_handle.close()
```

Step 3. Read the parameters

The parameters in these files are present in the first few lines of the file

```
# Reading parameters stored in the first few rows of the file
parm_dict = dict()
for line in string_lines[3:17]:
    line = line.replace('# ', '')
    line = line.replace('\n', '')
    temp = line.split('=')
    test = temp[1].strip()
    try:
        test = float(test)
        # convert those values that should be integers:
        if test % 1 == 0:
            test = int(test)
    except ValueError:
        pass
    parm_dict[temp[0].strip()] = test

# Print out the parameters extracted
for key in parm_dict.keys():
    print(key, ':\t', parm_dict[key])
```

Out:

```
x-pixels :       100
y-pixels :       100
x-length :       29.7595
y-length :       29.7595
x-offset :       -967.807
y-offset :       -781.441
z-points :       500
z-section :      491
z-unit :         nV
z-range :        2000000000
z-offset :       1116.49
value-unit :     nA
scanspeed :      59519000000
voidpixels :     0
```

Step 3.a Prepare to read the data

Before we read the data, we need to make an empty array to store all this data. In order to do this, we need to read the dictionary of parameters we made in step 2 and extract necessary quantities

```
num_rows = int(parm_dict['y-pixels'])
num_cols = int(parm_dict['x-pixels'])
num_pos = num_rows * num_cols
spectra_length = int(parm_dict['z-points'])
```

Step 3.b Read the data

Data is present after the first `403` lines of parameters.

```
# num_headers = len(string_lines) - num_pos
num_headers = 403

```

(continues on next page)

(continued from previous page)

```
# Extract the STS data from subsequent lines
raw_data_2d = np.zeros(shape=(num_pos, spectra_length), dtype=np.float32)
for line_ind in range(num_pos):
    this_line = string_lines[num_headers + line_ind]
    string_spectrum = this_line.split('\t')[:-1]  # omitting the new line
    raw_data_2d[line_ind] = np.array(string_spectrum, dtype=np.float32)
```

Step 4.a Preparing some necessary parameters

```
max_v = 1  # This is the one parameter we are not sure about

folder_path, file_name = os.path.split(data_file_path)
file_name = file_name[:-4] + '_'

# Generate the x / voltage / spectroscopic axis:
volt_vec = np.linspace(-1 * max_v, 1 * max_v, spectra_length)

h5_path = os.path.join(folder_path, file_name + '.h5')

sci_data_type = 'STS'
quantity = 'Current'
units = 'nA'
```

Step 4.b. Defining the Dimensions

Position and spectroscopic dimensions need to defined using `Dimension` objects. Remember that the position and spectroscopic dimensions need to be specified in the correct order.

```
pos_dims = [usid.write_utils.Dimension('X', 'a. u.', parm_dict['x-pixels']),
            usid.write_utils.Dimension('Y', 'a. u.', parm_dict['y-pixels'])]
spec_dims = usid.write_utils.Dimension('Bias', 'V', volt_vec)
```

Step 4.c. Calling the NumpyTranslator to create the h5USID file

The NumpyTranslator simplifies the creation of h5USID files. It handles the HDF5 file creation, HDF5 dataset creation and writing, creation of ancillary HDF5 datasets, group creation, writing parameters, linking ancillary datasets to the main dataset etc. With a single call to the NumpyTranslator, we complete the translation process.

```
tran = usid.NumpyTranslator()
h5_path = tran.translate(h5_path, sci_data_type, raw_data_2d,  quantity, units,
                         pos_dims, spec_dims, translator_name='Omicron_ASC_Translator
↪', parm_dict=parm_dict)
```

Notes on translation

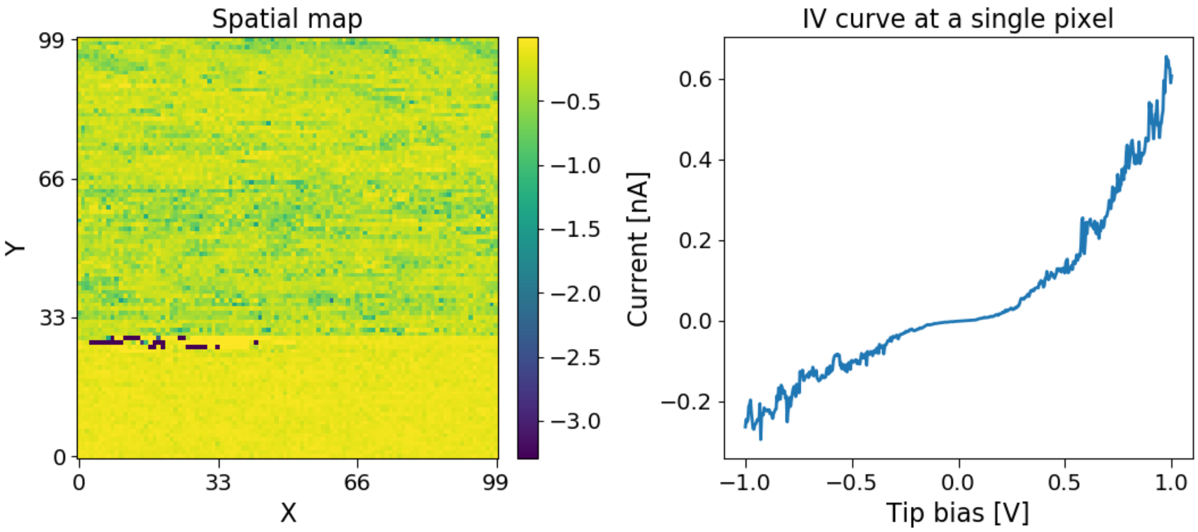
- Steps 1-3 would be performed anyway in order to begin data analysis
- The actual procedure for translation to h5USID is reduced to just 3-4 lines in step 4.

- A modular / formal version of this translator has been implemented as a class in pycroscopy as the AscTranslator. This custom translator packages the same code used above into functions that focus on the individual tasks such as extracting parameters, reading data, and writing to h5USID. The `NumpyTranslator` uses the `pyUSID.hdf_utils.write_main_dataset()` function underneath to write its data. You can learn more about lower- level file-writing functions in another tutorial on writing h5USID files.
- There are many benefits to writing such a formal Translator class instead of standalone scripts like this including:
  - Unlike such a stand-alone script, a Translator class in the package can be used by everyone repeatedly
  - The custom Translator class can ensure consistency when translating multiple files.
  - A single, robust Translator class can handle the finer variations / modes in the data. See the IgorIBWTranslator as an example.
- While this approach is feasible and encouraged for simple and small data, it may be necessary to use lower level calls to write efficient translators. As an example, please see the BEPSndfTranslator
- We have found python packages online to open a few proprietary file formats and have written translators using these packages. If you are having trouble reading the data in your files and cannot find any packages online, consider contacting the manufacturer of the instrument which generated the data in the proprietary format for help.

Verifying the newly written H5 file:

- We will only perform some simple and quick verification to show that the data has indeed been translated correctly.
- Please see the next notebook in the example series to learn more about reading and accessing data.

```
with h5py.File(h5_path, mode='r') as h5_file:
    # See if a tree has been created within the hdf5 file:
    usid.hdf_utils.print_tree(h5_file)

    h5_main = h5_file['Measurement_000/Channel_000/Raw_Data']
    usid.plot_utils.use_nice_plot_params()
    fig, axes = plt.subplots(ncols=2, figsize=(11, 5))
    spat_map = np.reshape(h5_main[:, 100], (100, 100))
    usid.plot_utils.plot_map(axes[0], spat_map, origin='lower')
    axes[0].set_title('Spatial map')
    axes[0].set_xlabel('X')
    axes[0].set_ylabel('Y')
    axes[1].plot(np.linspace(-1.0, 1.0, h5_main.shape[1]),
                 h5_main[250])
    axes[1].set_title('IV curve at a single pixel')
    axes[1].set_xlabel('Tip bias [V]')
    axes[1].set_ylabel('Current [nA]')

    fig.tight_layout()

# Remove both the original and translated files:
os.remove(h5_path)
os.remove(data_file_path)
```



Out:

```
/
├ Measurement_000
  ---------------
  ├ Channel_000
    -----------
    ├ Position_Indices
    ├ Position_Values
    ├ Raw_Data
    ├ Spectroscopic_Indices
    ├ Spectroscopic_Values
```

Total running time of the script: (0 minutes 6.779 seconds)

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

04. Plotting utilities

Suhas Somnath

8/12/2017

This is a short walkthrough of useful plotting utilities available in pyUSID

Introduction

Some of the functions in `pyUSID.plot_utils` fill gaps in the default matplotlib package, some were developed for scientific applications, and others were developed specifically for handling **Universal Spectroscopy and Imaging Data (USID)** `main` datasets. These functions have been developed to substantially simplify the generation of high quality figures for journal publications.

```
# Ensure python 3 compatibility:
from __future__ import division, print_function, absolute_import, unicode_literals
```

(continues on next page)

(continued from previous page)

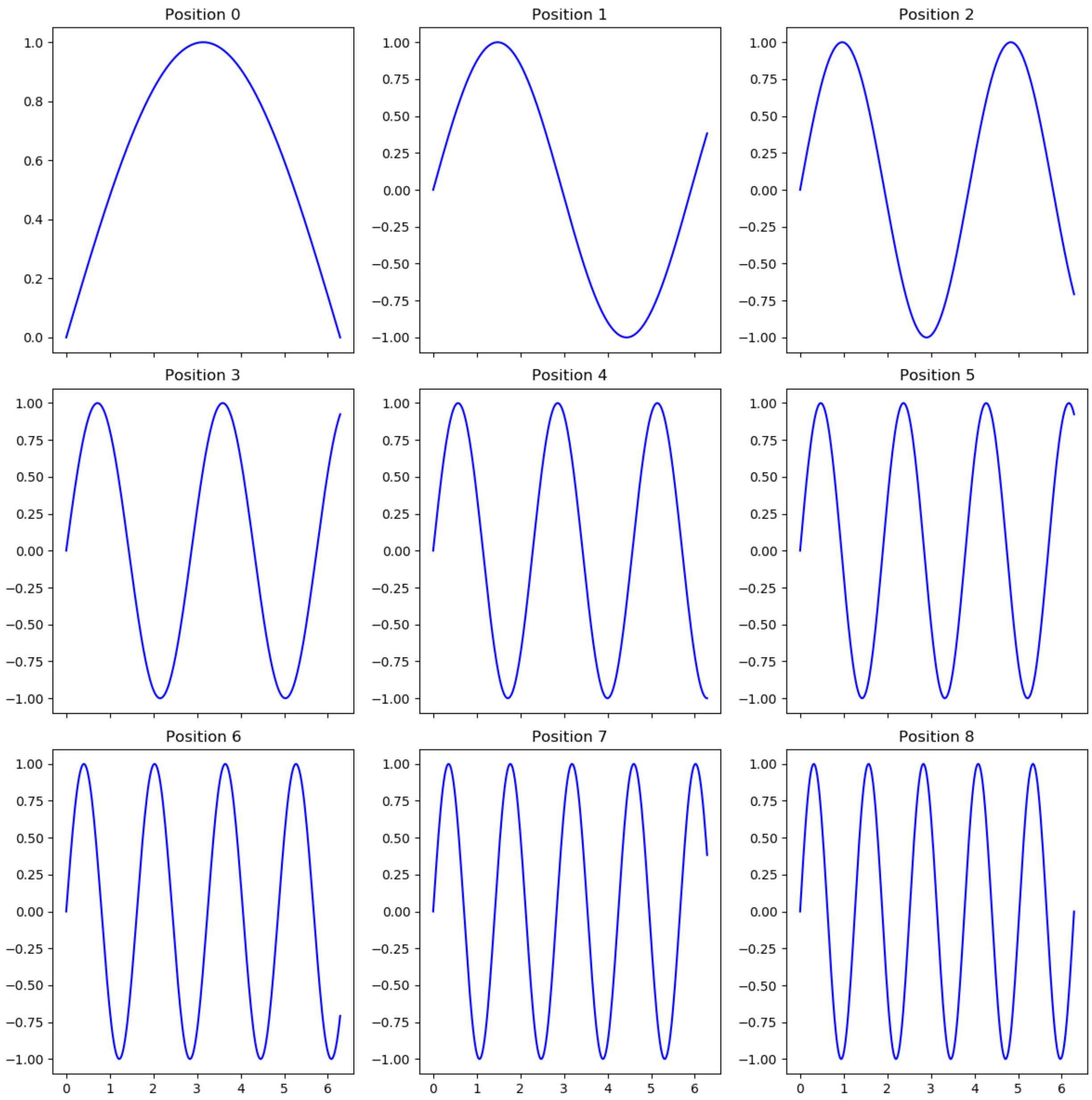
```
import numpy as np
from warnings import warn
import matplotlib.pyplot as plt
import subprocess
import sys


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

1D plot utilities

plot_curves()

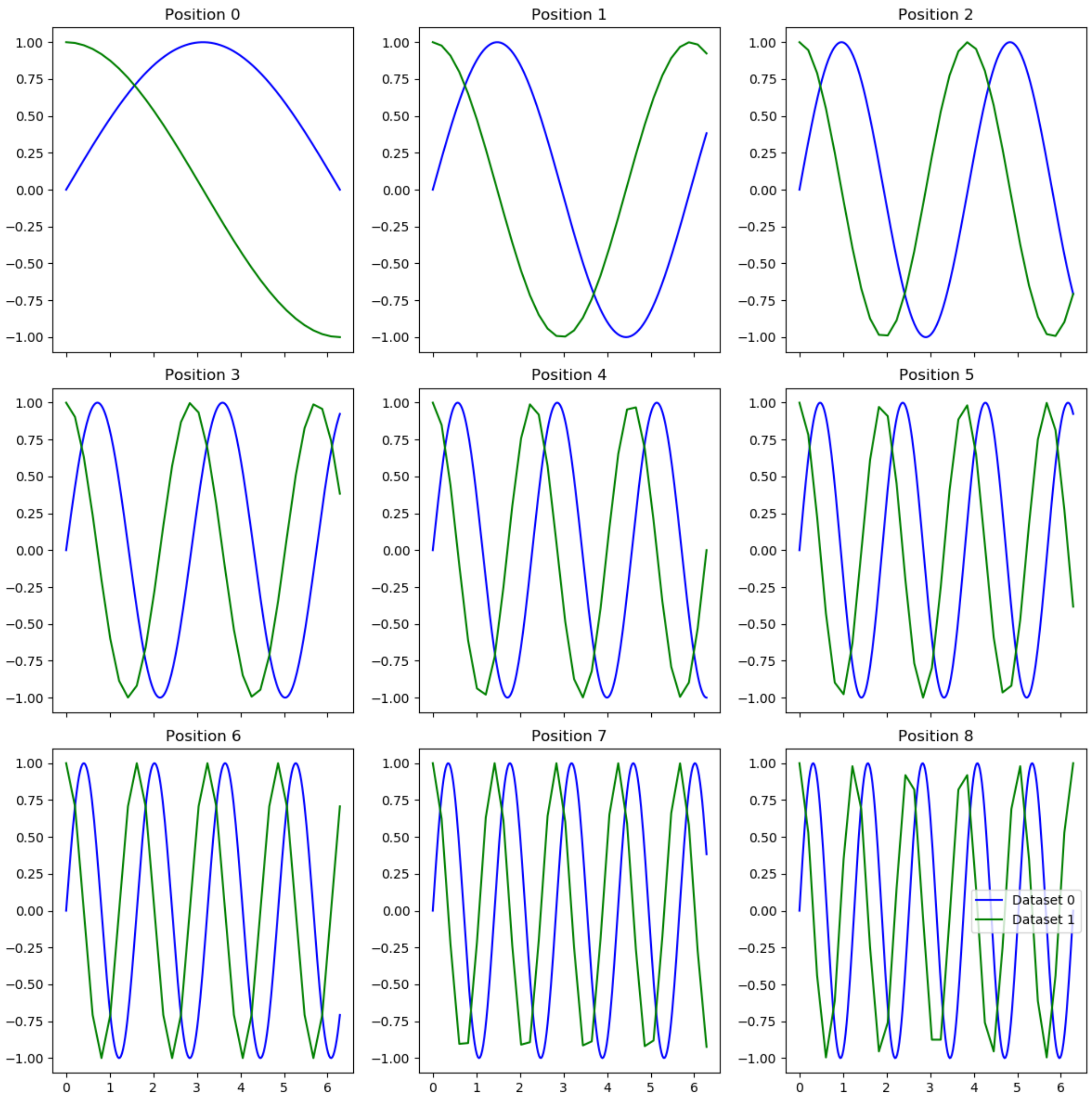
This function is particularly useful when we need to plot a 1D signal acquired at multiple locations. The function is rather flexible and can take on several optional arguments that will be alluded to below In the below example, we are simply simulating sine waveforms for different frequencies (think of these as different locations on a sample)

```
x_vec = np.linspace(0, 2*np.pi, 256)
# The different frequencies:
freqs = np.linspace(0.5, 5, 9)
# Generating the signals at the different "positions"
y_mat = np.array([np.sin(freq * x_vec) for freq in freqs])

usid.plot_utils.plot_curves(x_vec, y_mat)
```



Frequently, we may need to compare signals from two different datasets for the same positions The same plot_curves function can be used for this purpose even if the signal lengths / resolutions are different

```
x_vec_1 = np.linspace(0, 2*np.pi, 256)
x_vec_2 = np.linspace(0, 2*np.pi, 32)
freqs = np.linspace(0.5, 5, 9)
y_mat_1 = np.array([np.sin(freq * x_vec_1) for freq in freqs])
y_mat_2 = np.array([np.cos(freq * x_vec_2) for freq in freqs])

usid.plot_utils.plot_curves([x_vec_1, x_vec_2], [y_mat_1, y_mat_2],
                            title='Sine and Cosine of different resolutions')
```



plot_line_family()

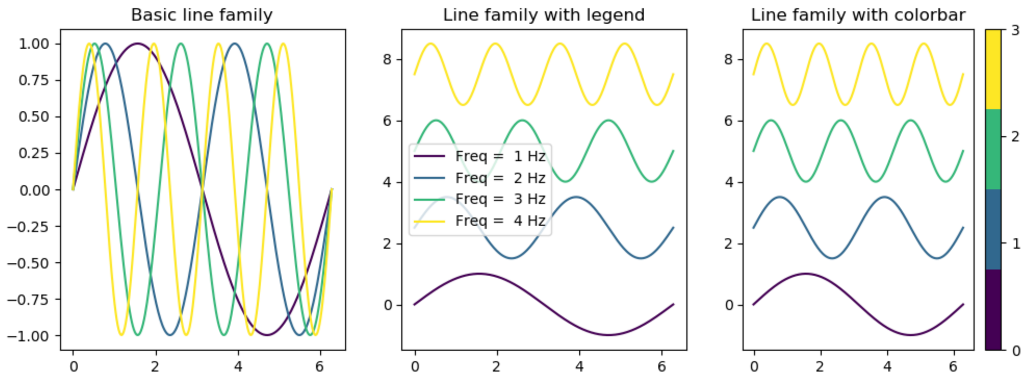
Often there is a need to visualize multiple spectra or signals on the same plot. plot_line_family is a handy function ideally suited for this purpose and it is highly configurable for different styles and purposes A few example applications include visualizing X ray / IR spectra (with y offsets), centroids from clustering algorithms

```
x_vec = np.linspace(0, 2*np.pi, 256)
freqs = range(1, 5)
y_mat = np.array([np.sin(freq * x_vec) for freq in freqs])
freq_strs = [str(_) for _ in freqs]

fig, axes = plt.subplots(ncols=3, figsize=(12, 4))
usid.plot_utils.plot_line_family(axes[0], x_vec, y_mat)
```

(continues on next page)

(continued from previous page)

```
axes[0].set_title('Basic line family')

# Option suitable for visualiing spectra with y offsets:
usid.plot_utils.plot_line_family(axes[1], x_vec, y_mat,
                               line_names=freq_strs, label_prefix='Freq = ', label_
↪suffix='Hz',
                                 y_offset=2.5)
axes[1].legend()
axes[1].set_title('Line family with legend')

# Option highly suited for visualizing the centroids from a clustering algorithm:
usid.plot_utils.plot_line_family(axes[2], x_vec, y_mat,
                               line_names=freq_strs, label_prefix='Freq = ', label_
↪suffix='Hz',
                                 y_offset=2.5, show_cbar=True)
axes[2].set_title('Line family with colorbar')
```



plot_complex_spectra()

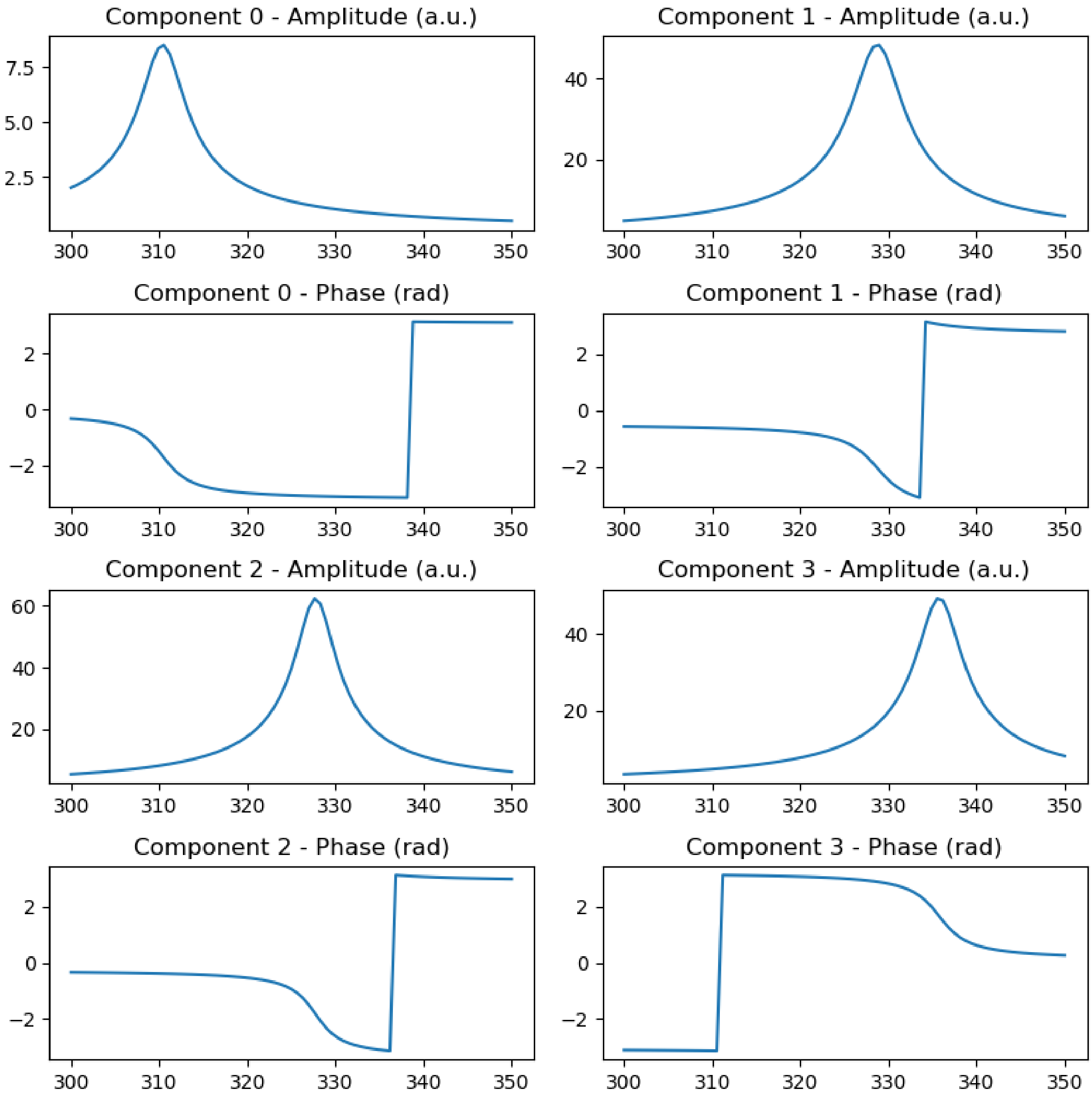
This handy function plots the amplitude and phase components of multiple complex valued spectra Here we simulate the signal coming from a simple harmonic oscillator (SHO).

```
num_spectra = 4
spectra_length = 77
w_vec = np.linspace(300, 350, spectra_length)
amps = np.random.rand(num_spectra)
freqs = np.random.rand(num_spectra)*35 + 310
q_facs = np.random.rand(num_spectra)*25 + 50
phis = np.random.rand(num_spectra)*2*np.pi
spectra = np.zeros((num_spectra, spectra_length), dtype=np.complex)


def sho_resp(parms, w_vec):
    """
    Generates the SHO response over the given frequency band
    Parameters
    -----------
    parms : list or tuple
        SHO parae=(A,w0,Q,phi)
    w_vec : 1D numpy array
```

(continues on next page)

(continued from previous page)

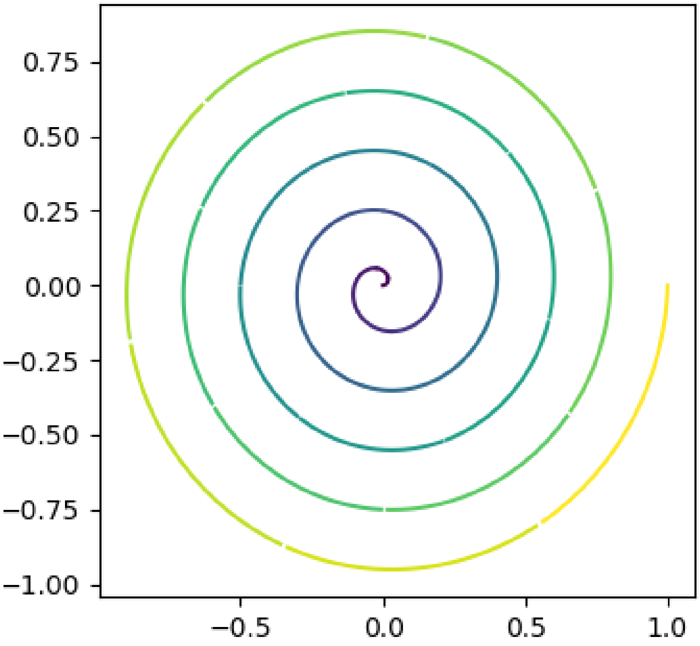
```
        Vector of frequency values
    """
    return parms[0] * np.exp(1j * parms[3]) * parms[1] ** 2 / \
        (w_vec ** 2 - 1j * w_vec * parms[1] / parms[2] - parms[1] ** 2)


for index, amp, freq, qfac, phase in zip(range(num_spectra), amps, freqs, q_facs,␣
↪phis):
    spectra[index] = sho_resp((amp, freq, qfac, phase), w_vec)

fig, axis = usid.plot_utils.plot_complex_spectra(spectra, w_vec, title='Oscillator␣
↪responses')
```



rainbow_plot()

This function is ideally suited for visualizing a signal that varies as a function of time or when the directionality of the signal is important

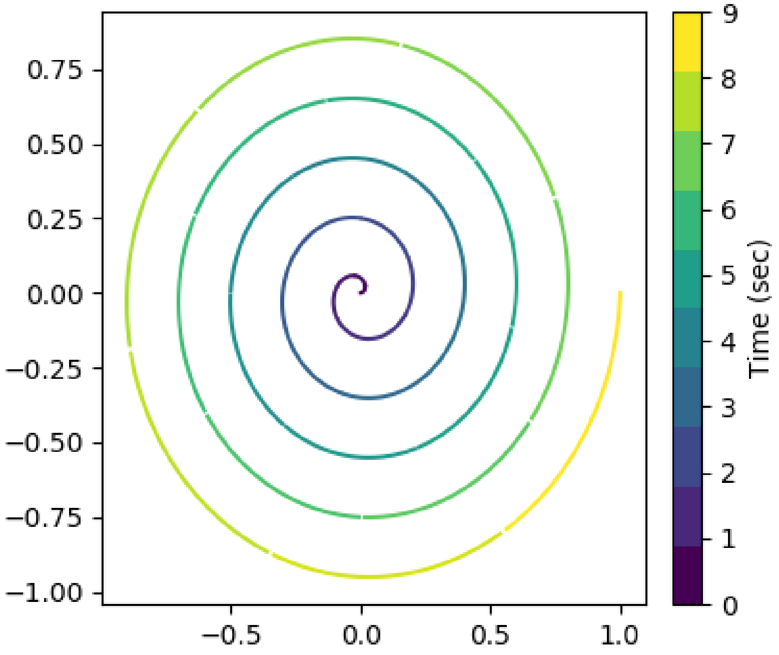
```
num_pts = 1024
t_vec = np.linspace(0, 10*np.pi, num_pts)

fig, axis = plt.subplots(figsize=(4, 4))
usid.plot_utils.rainbow_plot(axis, np.cos(t_vec)*np.linspace(0, 1, num_pts),
                             np.sin(t_vec)*np.linspace(0, 1, num_pts),
                             num_steps=32)
```



cbar_for_line_plot()

Note that from the above plot it may not be clear if the signal is radiating outwards or spiraling inwards. In these cases it helps to add a colorbar. However, colorbars can typically only be added for 2D images. In such cases we can use a handy function: `cbar_for_line_plot()`

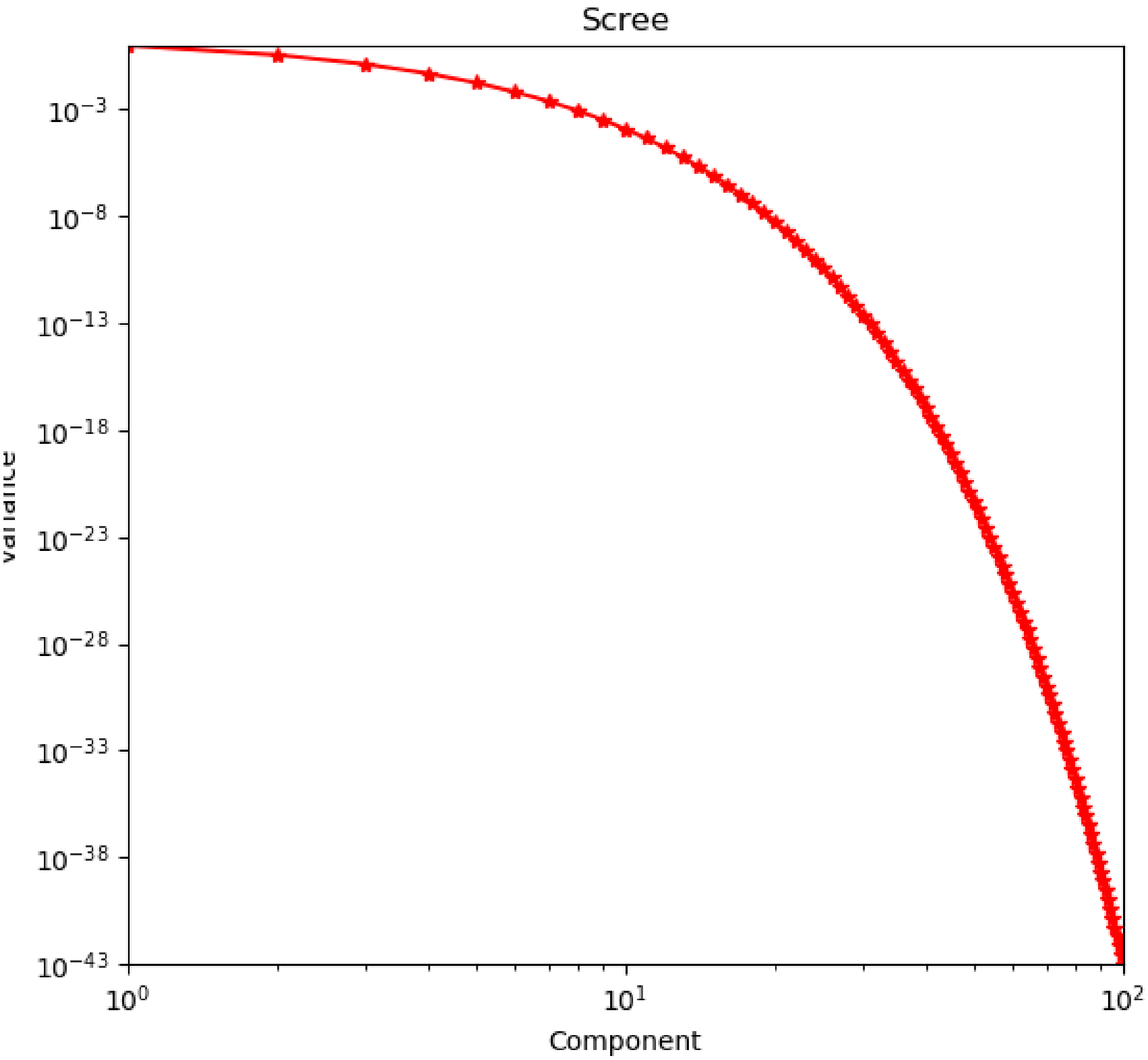
```
num_pts = 1024
t_vec = np.linspace(0, 10*np.pi, num_pts)

fig, axis = plt.subplots(figsize=(4.5, 4))
usid.plot_utils.rainbow_plot(axis, np.cos(t_vec)*np.linspace(0, 1, num_pts),
                             np.sin(t_vec)*np.linspace(0, 1, num_pts),
                             num_steps=32)

cbar = usid.plot_utils.cbar_for_line_plot(axis, 10)
cbar.set_label('Time (sec)')
```



plot_scree()

One of the results of applying Singular Value Decomposition is the variance or statistical significance of the resultant components. This data is best visualized via a log-log plot and plot_scree is available exclusively to visualize this kind of data

```
scree = np.exp(-1 * np.arange(100))
usid.plot_utils.plot_scree(scree, color='r')
```



Colormaps

plot_utils has a handful of colormaps suited for different applications.

cmap_jet_white_center()

This is the standard jet colormap with a white center instead of green. This is a good colormap for images with divergent data (data that diverges slightly both positively and negatively from the mean). One example target is the ronchigrams from scanning transmission electron microscopy (STEM)

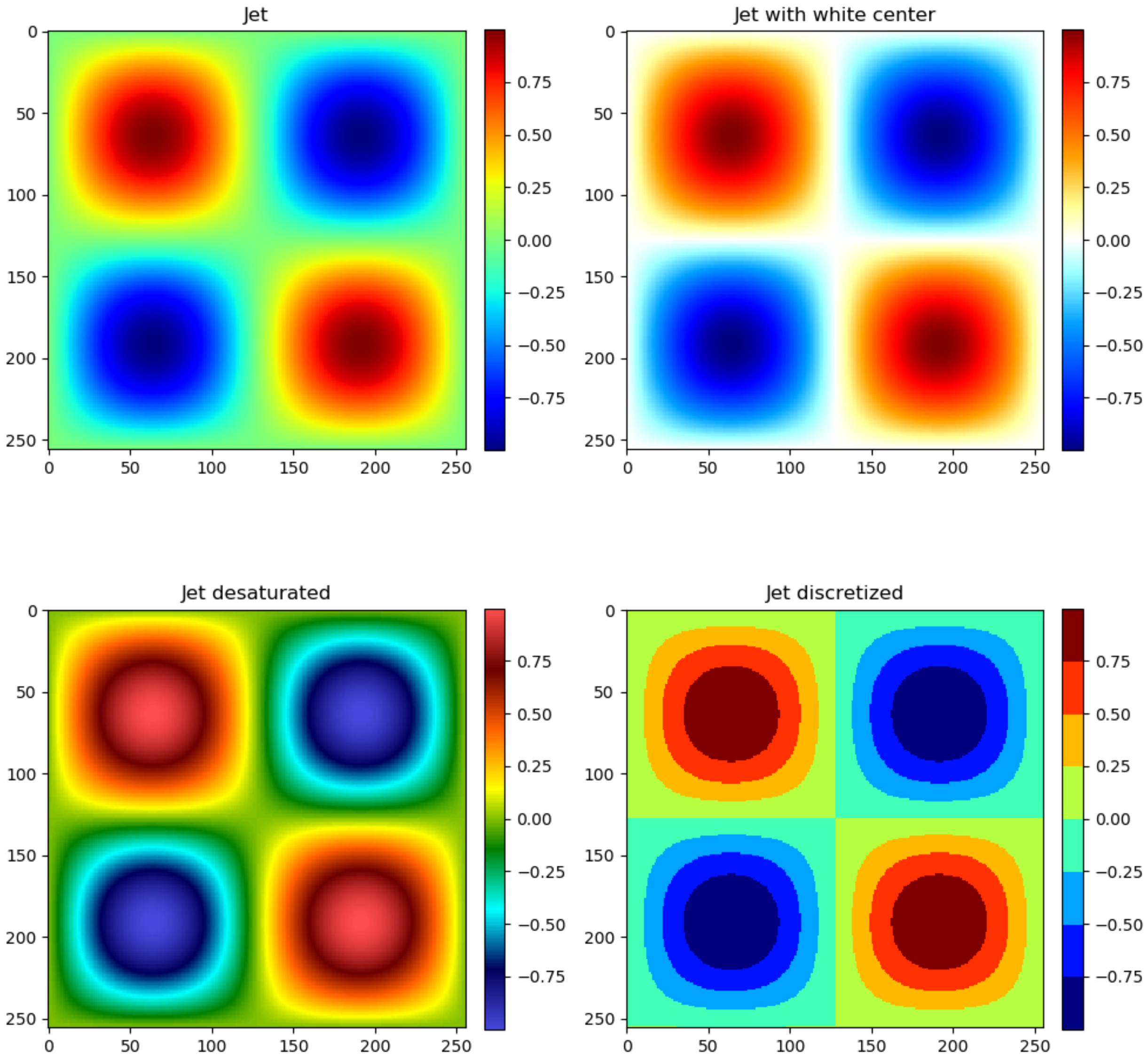
cmap_hot_desaturated()

This is a desaturated version of the standard jet colormap

discrete_cmap()

This function helps create a discretized version of the provided colormap. This is ideally suited when the data only contains a few discrete values. One popular application is the visualization of labels from a clustering algorithm

```
x_vec = np.linspace(0, 2*np.pi, 256)
y_vec = np.sin(x_vec)

test = y_vec * np.atleast_2d(y_vec).T

fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(10, 10))
for axis, title, cmap in zip(axes.flat,
                             ['Jet',
                              'Jet with white center',
                              'Jet desaturated',
                              'Jet discretized'],
                             [plt.cm.jet,
                              usid.plot_utils.cmap_jet_white_center(),
                              usid.plot_utils.cmap_hot_desaturated(),
                              usid.plot_utils.discrete_cmap(8, cmap='jet')]):
    im_handle = axis.imshow(test, cmap=cmap)
    cbar = plt.colorbar(im_handle, ax=axis, orientation='vertical',
                        fraction=0.046, pad=0.04, use_gridspec=True)
    axis.set_title(title)
fig.tight_layout()
```



make_linear_alpha_cmap()

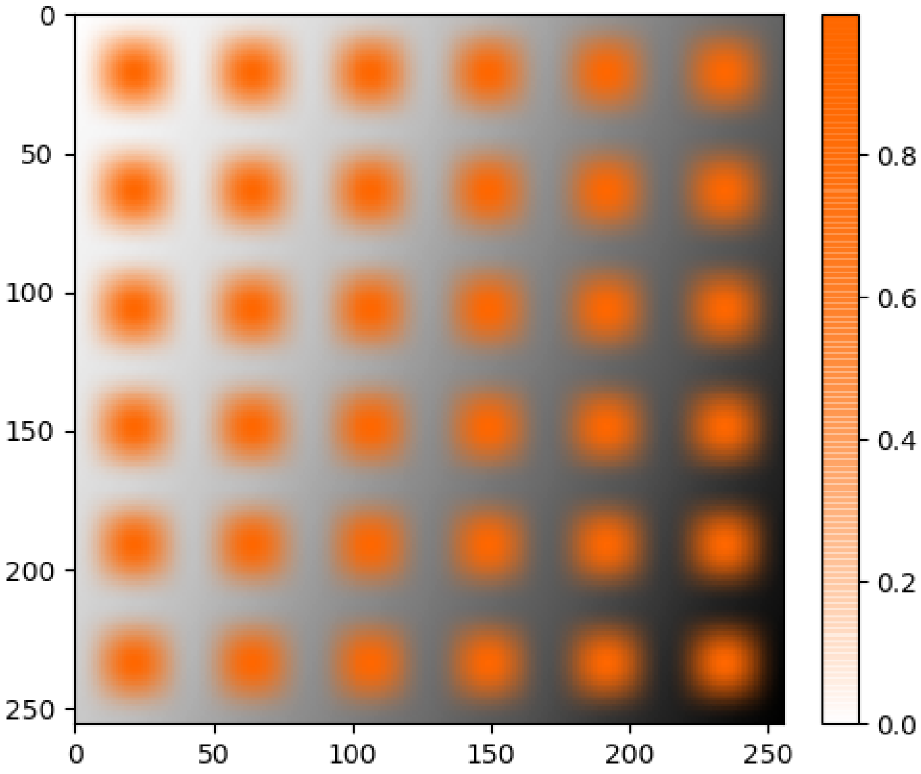
On certain occasions we may want to superimpose one image with another. However, this is not possible by default since colormaps involve solid colors. This function allows one to plot multiple images using a transparent-to-solid colormap. Here we will demonstrate this by plotting blobs representing atomic columns over some background intensity.

```
num_pts = 256

fig, axis = plt.subplots()
axis.hold(True)

# Prepare some backround signal
```

(continues on next page)

(continued from previous page)

```
x_mat, y_mat = np.meshgrid(np.linspace(-0.2*np.pi, 0.1*np.pi, num_pts), np.linspace(0,
↪ 0.25*np.pi, num_pts))
background_distortion = 0.2 * (x_mat + y_mat + np.sin(0.25 * np.pi * x_mat))

# plot this signal in grey
axis.imshow(background_distortion, cmap='Greys')

# prepare the signal of interest (think of this as intensities in a HREM dataset)
x_vec = np.linspace(0, 6*np.pi, num_pts)
y_vec = np.sin(x_vec)**2
atom_intensities = y_vec * np.atleast_2d(y_vec).T

# prepare the transparent-to-solid colormap
solid_color = plt.cm.jet(0.8)
translucent_colormap = usid.plot_utils.make_linear_alpha_cmap('my_map', solid_color,
                                                               1, min_alpha=0, max_
↪alpha=1)

# plot the atom intensities using the custom colormap
im_handle = axis.imshow(atom_intensities, cmap=translucent_colormap)
cbar = plt.colorbar(im_handle, ax=axis, orientation='vertical',
                        fraction=0.046, pad=0.04, use_gridspec=True)
```



get_cmap_object()

This function is useful more for developers writing their own plotting functions that need to manipulate the colormap object. This function makes it easy to ensure that you are working on the colormap object and not the string name of the colormap (both of which are accepted by most matplotlib functions). Here we simply compare the returned values when passing both the colormap object and the string name of the colormap

```
usid.plot_utils.get_cmap_object('jet') == usid.plot_utils.get_cmap_object(plt.cm.jet)
```

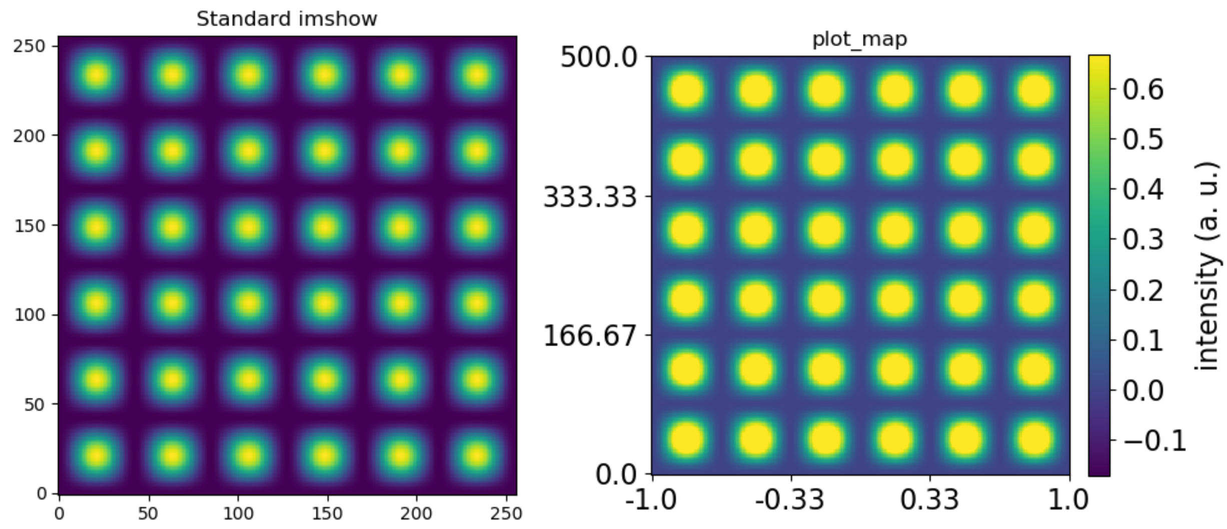
2D plot utilities

plot_map()

This function adds several popularly used features to the basic image plotting function in matplotlib including:

- easy addition of a colorbar
- custom x and y tick values
- clipping the colorbar to N standard deviations of the mean

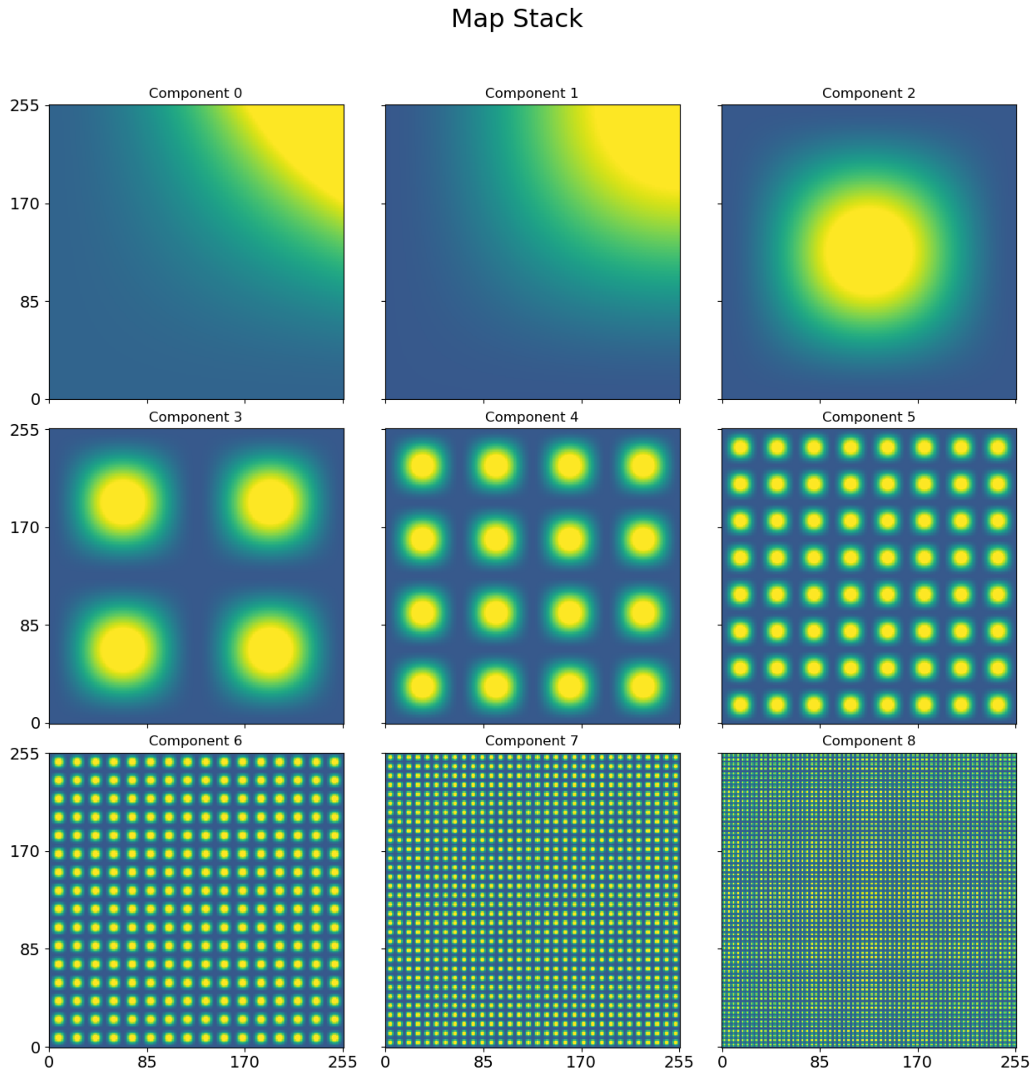
```
x_vec = np.linspace(0, 6*np.pi, 256)
y_vec = np.sin(x_vec)**2

atom_intensities = y_vec * np.atleast_2d(y_vec).T

fig, axes = plt.subplots(ncols=2, figsize=(10, 5))
# Standard imshow plot for reference
axes[0].imshow(atom_intensities, origin='lower')
axes[0].set_title('Standard imshow')
# Now plot_map with some options enabled:
usid.plot_utils.plot_map(axes[1], atom_intensities, stdevs=1.5, num_ticks=4,
                       x_vec=np.linspace(-1, 1, atom_intensities.shape[0]),
                       y_vec=np.linspace(0, 500, atom_intensities.shape[1]),
                       cbar_label='intensity (a. u.)', tick_font_size=16)
axes[1].set_title('plot_map')
fig.tight_layout()
```



plot_map_stack()

One of the most popular operations in scientific research is the visualization of a stack of images. This function is built specifically for that purpose. Here we simply simulate some images using sinusoidal functions for demonstration purposes.

```
def get_sine_2d_image(freq):
    x_vec = np.linspace(0, freq*np.pi, 256)
    y_vec = np.sin(x_vec)**2
    return y_vec * np.atleast_2d(y_vec).T


frequencies = [0.25, 0.5, 1, 2, 4 ,8, 16, 32, 64]
image_stack = [get_sine_2d_image(freq) for freq in frequencies]
image_stack = np.array(image_stack)

fig, axes = usid.plot_utils.plot_map_stack(image_stack, reverse_dims=False, title_
↪yoffset=0.95)
```



plot_complex_spectra()

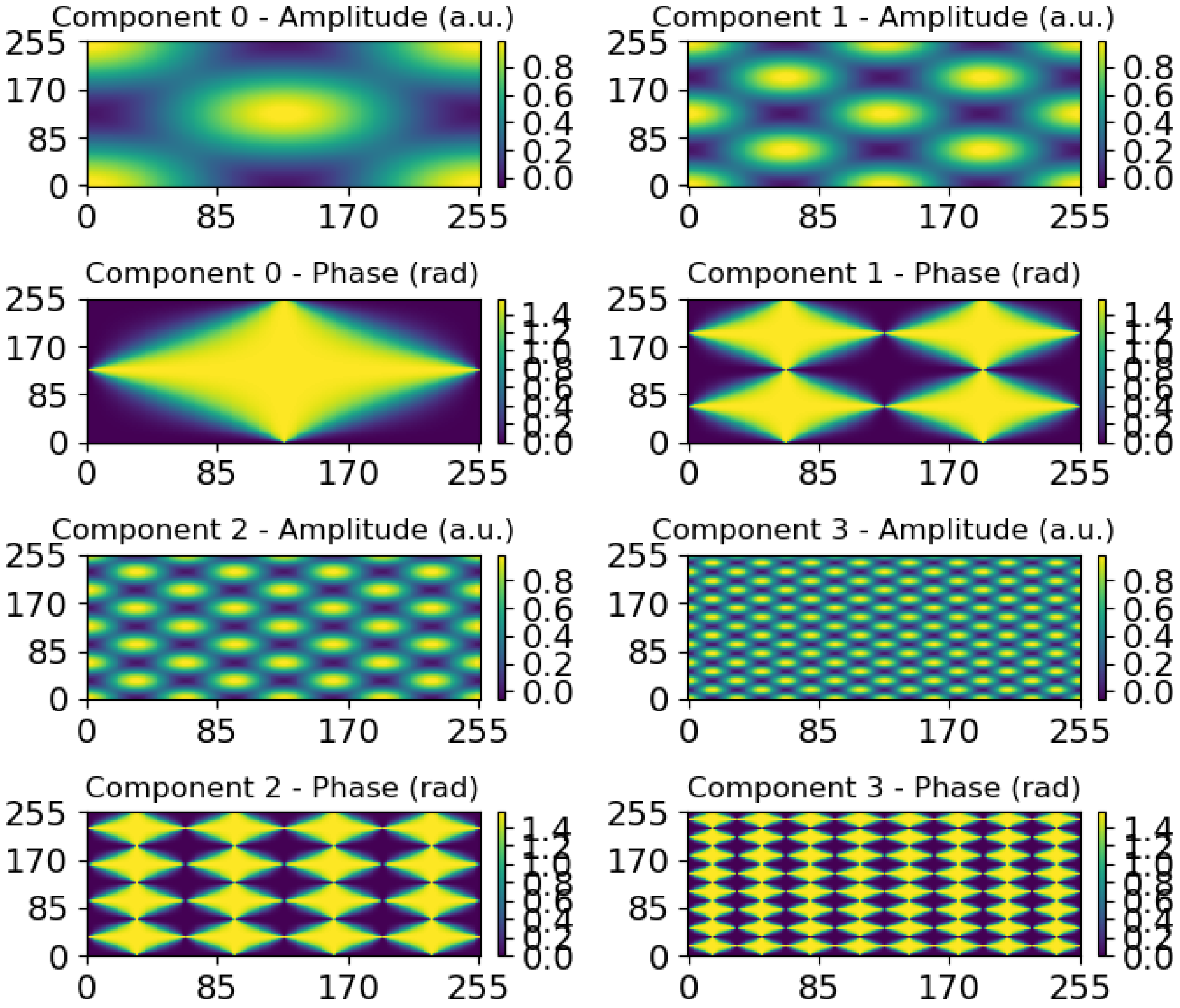
This function plots the amplitude and phase components of a stack of complex valued 2D images. Here we simulate the data using sine and cosine components

```
def get_complex_2d_image(freq):
    # Simple function to generate images
    x_vec = np.linspace(0, freq*np.pi, 256)
    y_vec_1 = np.sin(x_vec)**2
    y_vec_2 = np.cos(x_vec)**2
    return y_vec_2 * np.atleast_2d(y_vec_2).T + 1j*(y_vec_1 * np.atleast_2d(y_vec_1).
↪T)
```

(continues on next page)

(continued from previous page)

```
# The range of frequences over which the images are generated
frequencies = 2 ** np.arange(4)
image_stack = [get_complex_2d_image(freq) for freq in frequencies]

fig, axes = usid.plot_utils.plot_complex_spectra(np.array(image_stack), figsize=(3.5,␣
↪3))
```



General Utilities

set_tick_font_size()

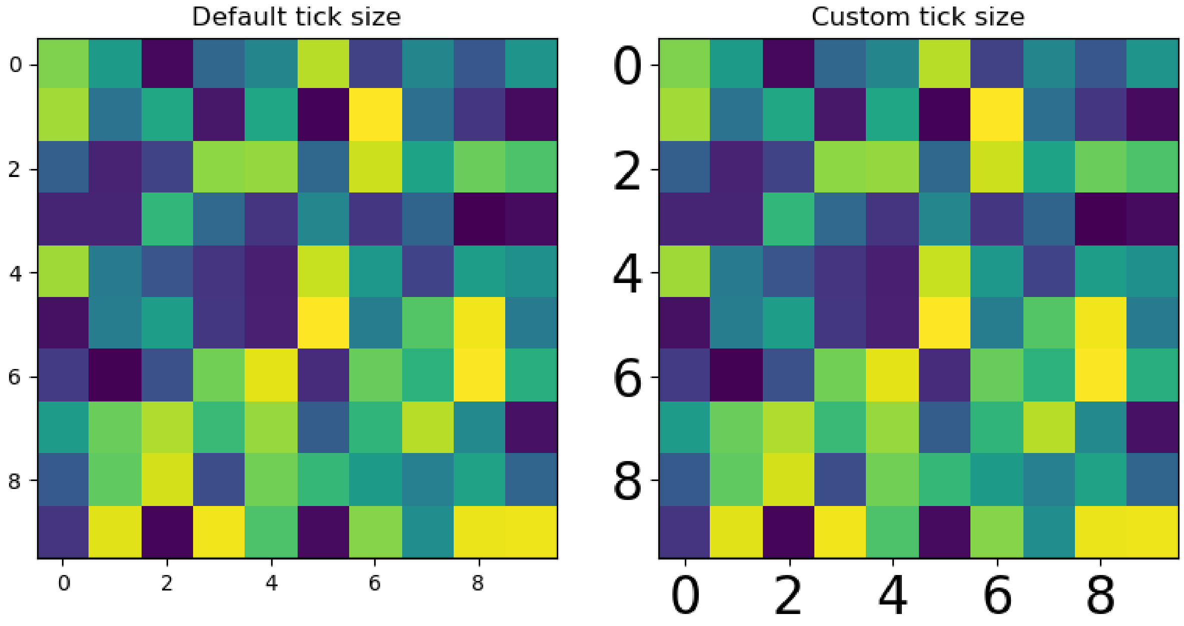
Adjusting the font sizes of the tick marks is often necessary for preparing figures for journal papers. However, adjusting the tick sizes is actually tedious in python and this function makes this easier.

```
test = np.random.rand(10, 10)
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(ncols=2, figsize=(8, 4))
for axis, title in zip(axes, ['Default', 'Custom']):
    axis.imshow(test)
    axis.set_title(title + ' tick size')
# only changing the tick font size on the second plot:
usid.plot_utils.set_tick_font_size(axes[1], 24)
fig.tight_layout()
```



get_plot_grid_size()

This handy function figures out the layout for a 2D grid of sub-plots given a desired number of plots

```
print('Subfigures\tFewer Rows\tFewer Columns')
for num_plots in range(1, 17):
    print('{}\t\t{}\t\t{}'.format(num_plots,
                                  usid.plot_utils.get_plot_grid_size(num_plots, fewer_
↪rows=True),
                                  usid.plot_utils.get_plot_grid_size(num_plots, fewer_
↪rows=False)))
```

Out:

```
Subfigures      Fewer Rows      Fewer Columns
1               (1, 1)          (1, 1)
2               (1, 2)          (2, 1)
3               (1, 3)          (3, 1)
4               (2, 2)          (2, 2)
5               (2, 3)          (3, 2)
6               (2, 3)          (3, 2)
7               (2, 4)          (4, 2)
8               (2, 4)          (4, 2)
9               (3, 3)          (3, 3)
10              (3, 4)          (4, 3)
```

(continues on next page)

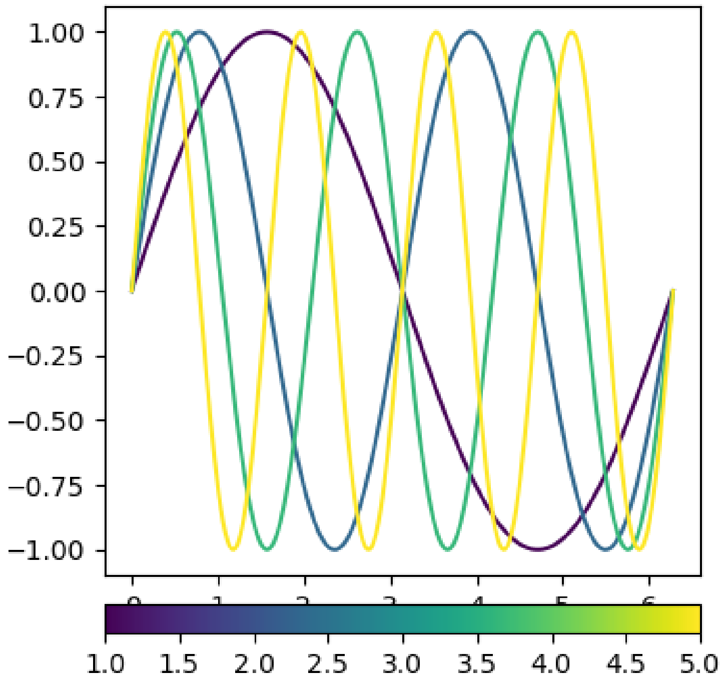
(continued from previous page)

```
11                (3, 4)          (4, 3)
12                (3, 4)          (4, 3)
13                (3, 5)          (5, 3)
14                (3, 5)          (5, 3)
15                (3, 5)          (5, 3)
16                (4, 4)          (4, 4)
```

make_scalar_mappable()

This is a low-level function that is used by `cbar_for_line_plot()` to generate the color bar manually. Here we revisit the example for plot_line_family() but we generate the colorbar by hand using `make_scalar_mappable()`. In this case, we make the colorbar horizontal just as an example.

```
x_vec = np.linspace(0, 2*np.pi, 256)
freqs = range(1, 5)
y_mat = np.array([np.sin(freq * x_vec) for freq in freqs])

fig, axis = plt.subplots(figsize=(4, 4.75))
usid.plot_utils.plot_line_family(axis, x_vec, y_mat)

num_steps = len(freqs)

sm = usid.plot_utils.make_scalar_mappable(1, num_steps+1)

cbar = plt.colorbar(sm, ax=axis, orientation='horizontal',
                    pad=0.04, use_gridspec=True)
```



cmap_from_rgba()

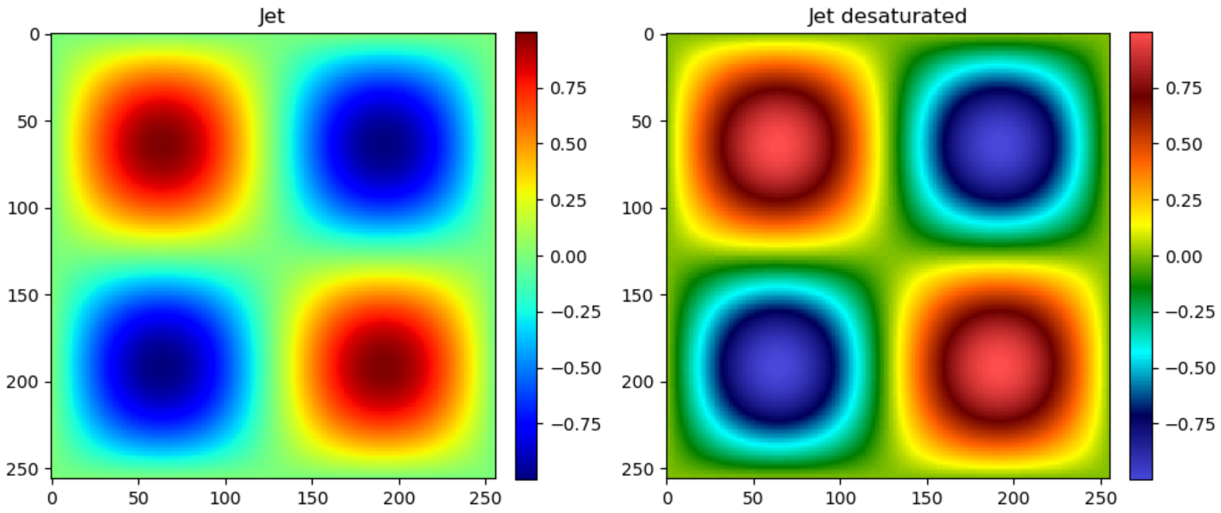
This function is handy for converting a Matlab-style colormap instructions (lists of [reg, green, blue, alpha]) to matplotlib's style:

```
hot_desaturated = [(255.0, (255, 76, 76, 255)),
                   (218.5, (107, 0, 0, 255)),
                   (182.1, (255, 96, 0, 255)),
                   (145.6, (255, 255, 0, 255)),
                   (109.4, (0, 127, 0, 255)),
                   (72.675, (0, 255, 255, 255)),
                   (36.5, (0, 0, 91, 255)),
                   (0, (71, 71, 219, 255))]

new_cmap = usid.plot_utils.cmap_from_rgba('hot_desaturated', hot_desaturated, 255)

x_vec = np.linspace(0, 2*np.pi, 256)
y_vec = np.sin(x_vec)

test = y_vec * np.atleast_2d(y_vec).T

fig, axes = plt.subplots(ncols=2, figsize=(10, 5))
for axis, title, cmap in zip(axes.flat,
                             ['Jet', 'Jet desaturated'],
```

(continues on next page)

(continued from previous page)

```
                              [plt.cm.jet, new_cmap]):
    im_handle = axis.imshow(test, cmap=cmap)
    cbar = plt.colorbar(im_handle, ax=axis, orientation='vertical',
                        fraction=0.046, pad=0.04, use_gridspec=True)
    axis.set_title(title)
fig.tight_layout()
```



use_nice_plot_params()

This function changes the default plotting parameters so that the figures look nicer and are closer to publication- ready figures. Note that all subsequent plots will be generated using the new defaults

reset_plot_params()

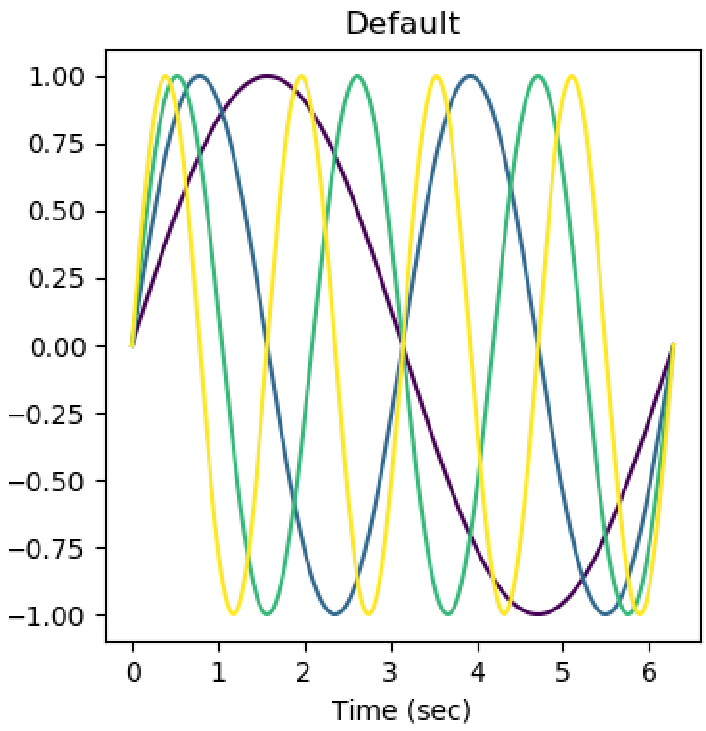
This function resets the plot parameters to matplotlib defaults. The following sequence of default >> nice >> default parameters will illustrate this.

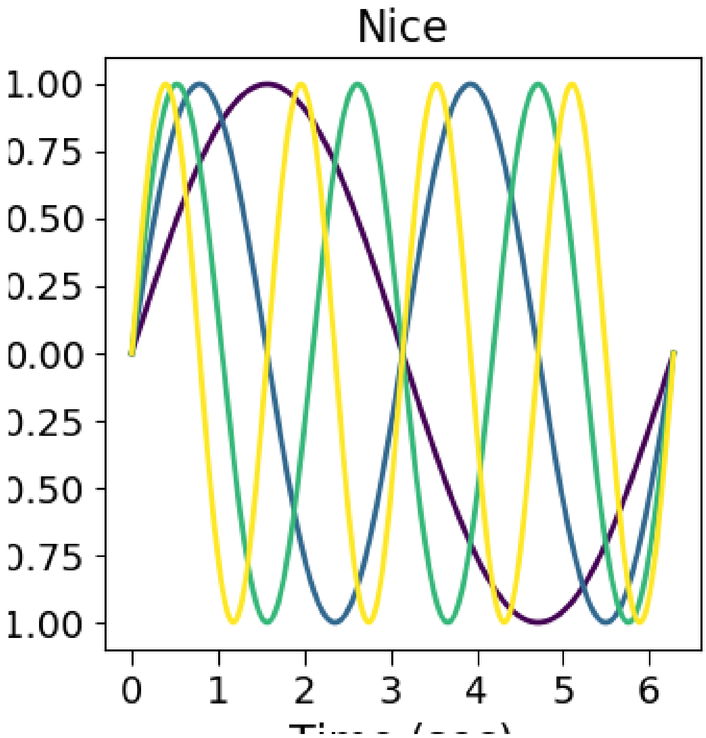
```
x_vec = np.linspace(0, 2*np.pi, 256)
freqs = range(1, 5)
y_mat = np.array([np.sin(freq * x_vec) for freq in freqs])

for nice in [False, True, False]:
    if nice:
        usid.plot_utils.use_nice_plot_params()
    else:
        usid.plot_utils.reset_plot_params()
    fig, axis = plt.subplots(figsize=(4, 4))
    usid.plot_utils.plot_line_family(axis, x_vec, y_mat)
    axis.set_xlabel('Time (sec)')
    axis.set_ylabel('Amplitude (a. u.)')
    if nice:
```

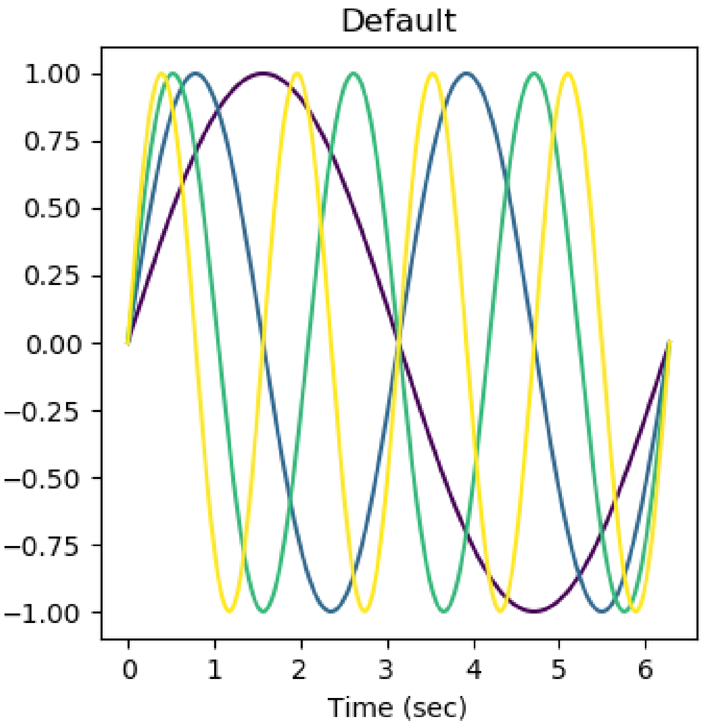
(continues on next page)

(continued from previous page)

```
        axis.set_title('Nice')
    else:
        axis.set_title('Default')
```

- 

- 

- 

Total running time of the script: (0 minutes 1.722 seconds)

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

05. Utilities for reading h5USID files

Suhas Somnath

4/18/2018

This document illustrates the many handy functions in pyUSID.hdf_utils that significantly simplify reading data and metadata in Universal Spectroscopy and Imaging Data (USID) HDF5 files (h5USID files)

Introduction

The USID model uses a data-centric approach to data analysis and processing meaning that results from all data analysis and processing are written to the same h5 file that contains the recorded measurements. **Hierarchical Data Format (HDF5)** files allow data, whether it is raw measured data or results of analysis, to be stored in multiple datasets within the same file in a tree-like manner. Certain rules and considerations have been made in pyUSID to ensure consistent and easy access to any data.

The h5py python package provides great functions to create, read, and manage data in HDF5 files. In `pyUSID.hdf_utils`, we have added functions that facilitate scientifically relevant, or USID specific functionality such as checking if a dataset is a Main dataset, reshaping to / from the original N dimensional form of the data, etc. Due to the wide breadth of the functions in `hdf_utils`, the guide for hdf_utils will be split in two parts - one that focuses on functions that facilitate reading and one that facilitate writing of data. The following guide provides examples of how, and more importantly when, to use functions in `pyUSID.hdf_utils` for various scenarios.

Recommended pre-requisite reading

- USID data model
- Crash course on HDF5 and h5py
- Utilities for writing h5USID files using pyUSID

Import all necessary packages

Before we begin demonstrating the numerous functions in `pyUSID.hdf_utils`, we need to import the necessary packages. Here are a list of packages besides pyUSID that will be used in this example:

- `h5py` - to open and close the file
- `wget` - to download the example data file
- `numpy` - for numerical operations on arrays in memory
- `matplotlib` - basic visualization of data

```python
from __future__ import print_function, division, unicode_literals
import os
# Warning package in case something goes wrong
from warnings import warn
import subprocess
import sys


def install(package):
    subprocess.call([sys.executable, "-m", "pip", "install", package])
# Package for downloading online files:

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
import h5py
import numpy as np
import matplotlib.pyplot as plt
# Finally import pyUSID.
try:
    import pyUSID as usid
except ImportError:
    warn('pyUSID not found.  Will install with pip.')
    import pip
    install('pyUSID')
    import pyUSID as usid
```

In order to demonstrate the many functions in hdf_utils, we will be using a h5USID file containing real experimental data along with results from analyses on the measurement data

This scientific dataset

For this example, we will be working with a **Band Excitation Polarization Switching (BEPS)** dataset acquired from advanced atomic force microscopes. In the much simpler **Band Excitation (BE)** imaging datasets, a single spectrum is acquired at each location in a two dimensional grid of spatial locations. Thus, BE imaging datasets have two position dimensions (X, Y) and one spectroscopic dimension (Frequency - against which the spectrum is recorded). The BEPS dataset used in this example has a spectrum for **each combination of** three other parameters (DC offset, Field, and Cycle). Thus, this dataset has three new spectral dimensions in addition to Frequency. Hence, this dataset becomes a 2+4 = **6 dimensional dataset**

Load the dataset

First, let us download this file from the pyUSID Github project:

```
url = 'https://raw.githubusercontent.com/pycroscopy/pyUSID/master/data/BEPS_small.h5'
h5_path = 'temp.h5'
_ = wget.download(url, h5_path, bar=None)

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

Out:

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

Next, lets open this HDF5 file in read-only mode. Note that opening the file does not cause the contents to be automatically loaded to memory. Instead, we are presented with objects that refer to specific HDF5 datasets, attributes or groups in the file

```
h5_path = 'temp.h5'
h5_f = h5py.File(h5_path, mode='r')
```

Here, h5_f is an active handle to the open file

Inspect HDF5 contents

The file contents are stored in a tree structure, just like files on a contemporary computer. The file contains groups (similar to file folders) and datasets (similar to spreadsheets). There are several datasets in the file and these store:

- The actual measurement collected from the experiment
- Spatial location on the sample where each measurement was collected
- Information to support and explain the spectral data collected at each location
- Since the USID model stores results from processing and analyses performed on the data in the same h5USID file, these datasets and groups are present as well
- Any other relevant ancillary information

print_tree()

Soon after opening any file, it is often of interest to list the contents of the file. While one can use the open source software HDFViewer developed by the HDF organization, pyUSID.hdf_utils also has a very handy function - print_tree() to quickly visualize all the datasets and groups within the file within python.

```
print('Contents of the H5 file:')
usid.hdf_utils.print_tree(h5_f)
```

Out:

```
Contents of the H5 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
    ├ Raw_Data-SHO_Fit_000
      --------------------
      ├ Fit
      ├ Guess
      ├ Spectroscopic_Indices
      ├ Spectroscopic_Values
    ├ Spatially_Averaged_Plot_Group_000
      ---------------------------------
      ├ Bin_Frequencies
      ├ Mean_Spectrogram
      ├ Spectroscopic_Parameter
      ├ Step_Averaged_Response
    ├ Spatially_Averaged_Plot_Group_001
      ---------------------------------
      ├ Bin_Frequencies
      ├ Mean_Spectrogram
      ├ Spectroscopic_Parameter
      ├ Step_Averaged_Response
    ├ Spectroscopic_Indices
    ├ Spectroscopic_Values
    ├ UDVS
    ├ UDVS_Indices
```

By default, `print_tree()` presents a clean tree view of the contents of the group. In this mode, only the group names are underlined. Alternatively, it can print the full paths of each dataset and group, with respect to the group / file of interest, by setting the `rel_paths` keyword argument. `print_tree()` could also be used to display the contents of and HDF5 group instead of complete HDF5 file as we have done above. Lets configure it to print the relative paths of all objects within the `Channel_000` group:

```
usid.hdf_utils.print_tree(h5_f['/Measurement_000/Channel_000/'], rel_paths=True)
```

Out:

```
/Measurement_000/Channel_000
Bin_FFT
Bin_Frequencies
```

(continues on next page)

(continued from previous page)

```
Bin_Indices
Bin_Step
Bin_Wfm_Type
Excitation_Waveform
Noise_Floor
Position_Indices
Position_Values
Raw_Data
Raw_Data-SHO_Fit_000
Raw_Data-SHO_Fit_000/Fit
Raw_Data-SHO_Fit_000/Guess
Raw_Data-SHO_Fit_000/Spectroscopic_Indices
Raw_Data-SHO_Fit_000/Spectroscopic_Values
Spatially_Averaged_Plot_Group_000
Spatially_Averaged_Plot_Group_000/Bin_Frequencies
Spatially_Averaged_Plot_Group_000/Mean_Spectrogram
Spatially_Averaged_Plot_Group_000/Spectroscopic_Parameter
Spatially_Averaged_Plot_Group_000/Step_Averaged_Response
Spatially_Averaged_Plot_Group_001
Spatially_Averaged_Plot_Group_001/Bin_Frequencies
Spatially_Averaged_Plot_Group_001/Mean_Spectrogram
Spatially_Averaged_Plot_Group_001/Spectroscopic_Parameter
Spatially_Averaged_Plot_Group_001/Step_Averaged_Response
Spectroscopic_Indices
Spectroscopic_Values
UDVS
UDVS_Indices
```

Finally, `print_tree()` can also be configured to only print USID Main datasets besides Group objects using the `main_dsets_only` option

```
usid.hdf_utils.print_tree(h5_f, main_dsets_only=True)
```

Out:

```
/
├ Measurement_000
  ---------------
  ├ Channel_000
    -----------
    ├ Raw_Data
    ├ Raw_Data-SHO_Fit_000
      --------------------
      ├ Fit
      ├ Guess
    ├ Spatially_Averaged_Plot_Group_000
      ---------------------------------
    ├ Spatially_Averaged_Plot_Group_001
      ---------------------------------
```

Accessing Attributes

HDF5 datasets and groups can also store metadata such as experimental parameters. These metadata can be text, numbers, small lists of numbers or text etc. These metadata can be very important for understanding the datasets and guide the analysis routines.

While one could use the basic h5py functionality to access attributes, one would encounter a lot of problems when attempting to decode attributes whose values were strings or lists of strings due to some issues in h5py. This problem has been demonstrated in our primer to HDF5 and h5py. Instead of using the basic functionality of h5py, we recommend always using the functions in pyUSID that reliably and consistently work for any kind of attribute for any version of python:

get_attributes()

get_attributes() is a very handy function that returns all or a specified set of attributes in an HDF5 object. If no attributes are explicitly requested, all attributes in the object are returned:

```
for key, val in usid.hdf_utils.get_attributes(h5_f).items():
    print('{} : {}'.format(key, val))
```

Out:

```
instrument : cypher_west
user_name : John Doe
data_type : BEPSData
current_position_x : 4
translate_date : 2017_08_22
current_position_y : 4
project_name : Band Excitation
xcams_id : abc
sample_name : PZT
comments : Band Excitation data
project_id : CNMS_2015B_X0000
experiment_unix_time : 1503428472.2374
experiment_date : 26-Feb-2015 14:49:48
data_tool : be_analyzer
grid_size_y : 5
sample_description : Thin Film
translator : ODF
grid_size_x : 5
Pycroscopy version : 0.0.a51
```

get_attributes() is also great for only getting selected attributes. For example, if we only cared about the user and project related attributes, we could manually request for any that we wanted:

```
proj_attrs = usid.hdf_utils.get_attributes(h5_f, ['project_name', 'project_id', 'user_
↪name'])
for key, val in proj_attrs.items():
    print('{} : {}'.format(key, val))
```

Out:

```
project_name : Band Excitation
project_id : CNMS_2015B_X0000
user_name : John Doe
```

get_attr()

If we are sure that we only wanted a specific attribute, we could instead use get_attr() as:

```
print(usid.hdf_utils.get_attr(h5_f, 'user_name'))
```

Out:

```
John Doe
```

check_for_matching_attrs()

Consider the scenario where we are have several HDF5 files or Groups or datasets and we wanted to check each one to see if they have the certain metadata / attributes. `check_for_matching_attrs()` is one very handy function that simplifies the comparision operation.

For example, let us check if this file was authored by `John Doe`:

```
print(usid.hdf_utils.check_for_matching_attrs(h5_f, new_parms={'user_name': 'John Doe
↪'}))
```

Out:

```
True
```

Finding datasets and groups

There are numerous ways to search for and access datasets and groups in H5 files using the basic functionalities of h5py. pyUSID.hdf_utils contains several functions that simplify common searching / lookup operations as part of scientific workflows.

find_dataset()

The `find_dataset()` function will return all datasets that whose names contain the provided string. In this case, we are looking for any datasets containing the string `UDVS` in their names. If you look above, there are two datasets (UDVS and UDVS_Indices) that match this condition:

```
udvs_dsets_2 = usid.hdf_utils.find_dataset(h5_f, 'UDVS')
for item in udvs_dsets_2:
    print(item)
```

Out:

```
<HDF5 dataset "UDVS": shape (256, 7), type "<f4">
<HDF5 dataset "UDVS_Indices": shape (22272,), type "<u8">
```

As you might know by now, h5USID files contain three kinds of datasets:

- `Main` datasets that contain data recorded / computed at multiple spatial locations.
- `Ancillary` datasets that support a main dataset
- Other datasets

For more information, please refer to the documentation on the USID model.

check_if_main()

`check_if_main()` is a very handy function that helps distinguish between `Main` datasets and other objects (`Ancillary` datasets, other datasets, Groups etc.). Lets apply this function to see which of the objects within the `Channel_000` Group are `Main` datasets:

```
h5_chan_group = h5_f['Measurement_000/Channel_000']

# We will prepare two lists - one of objects that are ``main`` and one of objects␣
↪that are not

non_main_objs = []
main_objs = []
for key, val in h5_chan_group.items():
    if usid.hdf_utils.check_if_main(val):
        main_objs.append(key)
    else:
        non_main_objs.append(key)

# Now we simply print the names of the items in each list

print('Main Datasets:')
print('----------------')
for item in main_objs:
    print(item)
print('\nObjects that were not Main datasets:')
print('--------------------------------------')
for item in non_main_objs:
    print(item)
```

Out:

```
Main Datasets:
----------------
Raw_Data

Objects that were not Main datasets:
--------------------------------------
Bin_FFT
Bin_Frequencies
Bin_Indices
Bin_Step
Bin_Wfm_Type
Excitation_Waveform
Noise_Floor
Position_Indices
Position_Values
Raw_Data-SHO_Fit_000
Spatially_Averaged_Plot_Group_000
Spatially_Averaged_Plot_Group_001
Spectroscopic_Indices
Spectroscopic_Values
UDVS
UDVS_Indices
```

The above script allowed us to distinguish Main datasets from all other objects only within the Group named `Channel_000`.

get_all_main()

What if we want to quickly find all Main datasets even within the sub-Groups of Channel_000? To do this, we have a very handy function called - get_all_main():

```
main_dsets = usid.hdf_utils.get_all_main(h5_chan_group)
for dset in main_dsets:
    print(dset)
    print('--------------------------------------------------------------------')
```

Out:

```
<HDF5 dataset "Raw_Data": shape (25, 22272), type "<c8">
located at:
        /Measurement_000/Channel_000/Raw_Data
Data contains:
        Cantilever Vertical Deflection (V)
Data dimensions and original shape:
Position Dimensions:
        X - size: 5
        Y - size: 5
Spectroscopic Dimensions:
        Frequency - size: 87
        DC_Offset - size: 64
        Field - size: 2
        Cycle - size: 2
Data Type:
        complex64
--------------------------------------------------------------------
<HDF5 dataset "Fit": shape (25, 256), type "|V20">
located at:
        /Measurement_000/Channel_000/Raw_Data-SHO_Fit_000/Fit
Data contains:
        SHO parameters (compound)
Data dimensions and original shape:
Position Dimensions:
        X - size: 5
        Y - size: 5
Spectroscopic Dimensions:
        DC_Offset - size: 64
        Field - size: 2
        Cycle - size: 2
Data Fields:
        Amplitude [V], Frequency [Hz], Quality Factor, Phase [rad], R2 Criterion
--------------------------------------------------------------------
<HDF5 dataset "Guess": shape (25, 256), type "|V20">
located at:
        /Measurement_000/Channel_000/Raw_Data-SHO_Fit_000/Guess
Data contains:
        SHO parameters (compound)
Data dimensions and original shape:
Position Dimensions:
        X - size: 5
        Y - size: 5
Spectroscopic Dimensions:
        DC_Offset - size: 64
        Field - size: 2
```

(continues on next page)

(continued from previous page)

```
        Cycle - size: 2
Data Fields:
        Amplitude [V], Frequency [Hz], Quality Factor, Phase [rad], R2 Criterion
--------------------------------------------------------------------
```

The datasets above show that the file contains three main datasets. Two of these datasets are contained in a HDF5 Group called `Raw_Data-SHO_Fit_000` meaning that they are results of an operation called `SHO_Fit` performed on the `Main` dataset - `Raw_Data`. The first of the three main datasets is indeed the `Raw_Data` dataset from which the latter two datasets (`Fit` and `Guess`) were derived.

The USID model allows the same operation, such as `SHO_Fit`, to be performed on the same dataset (`Raw_Data`), multiple times. Each time the operation is performed, a new HDF5 Group is created to hold the new results. Often, we may want to perform a few operations such as:

- Find the (source / main) dataset from which certain results were derived
- Check if a particular operation was performed on a main dataset
- Find all groups corresponding to a particular operation (e.g. - `SHO_Fit`) being applied to a Main dataset

`hdf_utils` has a few handy functions for many of these use cases.

find_results_groups()

First, lets show that `find_results_groups()` finds all Groups containing the results of a `SHO_Fit` operation applied to `Raw_Data`:

```
# First get the dataset corresponding to Raw_Data
h5_raw = h5_chan_group['Raw_Data']

operation = 'SHO_Fit'
print('Instances of operation "{}" applied to dataset named "{}":'.format(operation,␣
↪h5_raw.name))
h5_sho_group_list = usid.hdf_utils.find_results_groups(h5_raw, operation)
print(h5_sho_group_list)
```

Out:

```
Instances of operation "SHO_Fit" applied to dataset named "/Measurement_000/Channel_
↪000/Raw_Data":
[<HDF5 group "/Measurement_000/Channel_000/Raw_Data-SHO_Fit_000" (4 members)>]
```

As expected, the `SHO_Fit` operation was performed on `Raw_Data` dataset only once, which is why `find_results_groups()` returned only one HDF5 Group - `SHO_Fit_000`.

check_for_old()

Often one may want to check if a certain operation was performed on a dataset with the very same parameters to avoid recomputing the results. `hdf_utils.check_for_old()` is a very handy function that compares parameters (a dictionary) for a new / potential operation against the metadata (attributes) stored in each existing results group (HDF5 groups whose name starts with `Raw_Data-SHO_Fit` in this case). Before we demonstrate `check_for_old()`, lets take a look at the attributes stored in the existing results groups:

```
print('Parameters already used for computing SHO_Fit on Raw_Data in the file:')
for key, val in usid.hdf_utils.get_attributes(h5_chan_group['Raw_Data-SHO_Fit_000']).
↪items():
    print('{} : {}'.format(key, val))
```

Out:

```
Parameters already used for computing SHO_Fit on Raw_Data in the file:
SHO_guess_method : pycroscopy BESHO
timestamp : 2017_08_22-15_02_08
machine_id : mac109728.ornl.gov
SHO_fit_method : pycroscopy BESHO
```

Now, let us check for existing results where the `SHO_fit_method` attribute matches an existing value and a new value:

```
print('Checking to see if SHO Fits have been computed on the raw dataset:')
print('\nUsing "pycroscopy BESHO":')
print(usid.hdf_utils.check_for_old(h5_raw, 'SHO_Fit',
                                   new_parms={'SHO_fit_method': 'pycroscopy BESHO'}))
print('\nUsing "alternate technique"')
print(usid.hdf_utils.check_for_old(h5_raw, 'SHO_Fit',
                                   new_parms={'SHO_fit_method': 'alternate technique'}))
```

Out:

```
Checking to see if SHO Fits have been computed on the raw dataset:

Using "pycroscopy BESHO":
[<HDF5 group "/Measurement_000/Channel_000/Raw_Data-SHO_Fit_000" (4 members)>]

Using "alternate technique"
[]
```

Clearly, while find_results_groups() returned any and all groups corresponding to `SHO_Fit` being applied to `Raw_Data`, `check_for_old()` only returned the group(s) where the operation was performed using the same specified parameters (`sho_fit_method` in this case).

Note that `check_for_old()` performs two operations - search for all groups with the matching nomenclature and then compare the attributes. `check_for_matching_attrs()` is the handy function, that enables the latter operation of comparing a giving dictionary of parameters against attributes in a given object.

get_source_dataset()

`hdf_utils.get_source_dataset()` is a very handy function for the inverse scenario where we are interested in finding the source dataset from which the known result was derived:

```
h5_sho_group = h5_sho_group_list[0]
print('Datagroup containing the SHO fits:')
print(h5_sho_group)
print('\nDataset on which the SHO Fit was computed:')
h5_source_dset = usid.hdf_utils.get_source_dataset(h5_sho_group)
print(h5_source_dset)
```

Out:

```
Datagroup containing the SHO fits:
<HDF5 group "/Measurement_000/Channel_000/Raw_Data-SHO_Fit_000" (4 members)>

Dataset on which the SHO Fit was computed:
<HDF5 dataset "Raw_Data": shape (25, 22272), type "<c8">
located at:
        /Measurement_000/Channel_000/Raw_Data
Data contains:
        Cantilever Vertical Deflection (V)
Data dimensions and original shape:
Position Dimensions:
        X - size: 5
        Y - size: 5
Spectroscopic Dimensions:
        Frequency - size: 87
        DC_Offset - size: 64
        Field - size: 2
        Cycle - size: 2
Data Type:
        complex64
```

Since the source dataset is always a `Main` dataset, `get_source_dataset()` results a `USIDataset` object instead of a regular `HDF5 Dataset` object.

Note that `hdf_utils.get_source_dataset()` and `find_results_groups()` rely on the USID rule that results of an operation be stored in a Group named `Source_Dataset_Name-Operation_Name_00x`.

```
# get_auxiliary_datasets()
# -------------------------
# The association of datasets and groups with one another provides a powerful␣
↪mechanism for conveying (richer)
# information. One way to associate objects with each other is to store the reference␣
↪of an object as an attribute of
# another. This is precisely the capability that is leveraged to turn Central␣
↪datasets into USID Main Datasets or
# ``USIDatasets``. USIDatasets need to have four attributes that are references to␣
↪the ``Position`` and ``Spectroscopic``
# ``ancillary`` datasets. Note, that USID does not restrict or preclude the storage␣
↪of other relevant datasets as
# attributes of another dataset.
#
# For example, the ``Raw_Data`` dataset appears to contain several attributes whose␣
↪keys / names match the names of
# datasets we see above and values all appear to be HDF5 object references:

for key, val in usid.hdf_utils.get_attributes(h5_raw).items():
    print('{} : {}'.format(key, val))
```

Out:

```
Excitation_Waveform : <HDF5 object reference>
Position_Indices : <HDF5 object reference>
Position_Values : <HDF5 object reference>
Spectroscopic_Indices : <HDF5 object reference>
UDVS : <HDF5 object reference>
Bin_Step : <HDF5 object reference>
Bin_Indices : <HDF5 object reference>
```

(continues on next page)

(continued from previous page)

```
UDVS_Indices : <HDF5 object reference>
Bin_Frequencies : <HDF5 object reference>
Bin_FFT : <HDF5 object reference>
Bin_Wfm_Type : <HDF5 object reference>
in_field_Plot_Group : <HDF5 region reference>
out_of_field_Plot_Group : <HDF5 region reference>
Noise_Floor : <HDF5 object reference>
Spectroscopic_Values : <HDF5 object reference>
quantity : Cantilever Vertical Deflection
units : V
```

As the name suggests, these HDF5 object references are references or addresses to datasets located elsewhere in the file. Conventionally, one would need to apply this reference to the file handle to get the actual HDF5 Dataset / Group object.

`get_auxiliary_datasets()` simplifies this process by directly retrieving the actual Dataset / Group associated with the attribute. Thus, we would be able to get a reference to the `Bin_Frequencies` Dataset via:

```
h5_obj = usid.hdf_utils.get_auxiliary_datasets(h5_raw, 'Bin_Frequencies')[0]
print(h5_obj)
# Lets prove that this object is the same as the 'Bin_Frequencies' object that can be␣
↪directly addressed:
print(h5_obj == h5_f['/Measurement_000/Channel_000/Bin_Frequencies'])
```

Out:

```
<HDF5 dataset "Bin_Frequencies": shape (87,), type "<f4">
True
```

Accessing Ancillary Datasets

One of the major benefits of h5USID is its ability to handle large multidimensional datasets at ease. `Ancillary` datasets serve as the keys or legends for explaining the dimensionality, reshape-ability, etc. of a dataset. There are several functions in hdf_utils that simplify many common operations on ancillary datasets.

Before we demonstrate the several useful functions in hdf_utils, lets access the position and spectroscopic ancillary datasets using the `get_auxiliary_datasets()` function we used above:

```
dset_list = usid.hdf_utils.get_auxiliary_datasets(h5_raw, ['Position_Indices',
↪'Position_Values',
                                                         'Spectroscopic_Indices',
↪'Spectroscopic_Values'])
h5_pos_inds, h5_pos_vals, h5_spec_inds, h5_spec_vals = dset_list
```

As mentioned above, this is indeed a six dimensional dataset with two position dimensions and four spectroscopic dimensions. The `Field` and `Cycle` dimensions do not have any units since they are dimensionless unlike the other dimensions.

get_dimensionality()

Now lets find out the number of steps in each of those dimensions using another handy function called `get_dimensionality()`:

```
pos_dim_sizes = usid.hdf_utils.get_dimensionality(h5_pos_inds)
spec_dim_sizes = usid.hdf_utils.get_dimensionality(h5_spec_inds)
pos_dim_names = usid.hdf_utils.get_attr(h5_pos_inds, 'labels')
spec_dim_names = usid.hdf_utils.get_attr(h5_spec_inds, 'labels')

print('Size of each Position dimension:')
for name, length in zip(pos_dim_names, pos_dim_sizes):
    print('{} : {}'.format(name, length))
print('\nSize of each Spectroscopic dimension:')
for name, length in zip(spec_dim_names, spec_dim_sizes):
    print('{} : {}'.format(name, length))
```

Out:

```
Size of each Position dimension:
X : 5
Y : 5

Size of each Spectroscopic dimension:
Frequency : 87
DC_Offset : 64
Field : 2
Cycle : 2
```

get_sort_order()

In a few (rare) cases, the spectroscopic / position dimensions are not arranged in descending order of rate of change. In other words, the dimensions in these ancillary matrices are not arranged from fastest-varying to slowest. To account for such discrepancies, `hdf_utils` has a very handy function that goes through each of the columns or rows in the ancillary indices matrices and finds the order in which these dimensions vary.

Below we illustrate an example of sorting the names of the spectroscopic dimensions from fastest to slowest in the BEPS data file:

```
spec_sort_order = usid.hdf_utils.get_sort_order(h5_spec_inds)
print('Rate of change of spectroscopic dimensions: {}'.format(spec_sort_order))
print('\nSpectroscopic dimensions arranged as is:')
print(spec_dim_names)
sorted_spec_labels = np.array(spec_dim_names)[np.array(spec_sort_order)]
print('\nSpectroscopic dimensions arranged from fastest to slowest')
print(sorted_spec_labels)
```

Out:

```
Rate of change of spectroscopic dimensions: [0 2 1 3]

Spectroscopic dimensions arranged as is:
['Frequency' 'DC_Offset' 'Field' 'Cycle']

Spectroscopic dimensions arranged from fastest to slowest
['Frequency' 'Field' 'DC_Offset' 'Cycle']
```

get_unit_values()

When visualizing the data it is essential to plot the data against appropriate values on the X, Y, or Z axes. Recall that by definition that the values over which each dimension is varied, are repeated and tiled over the entire position or spectroscopic dimension of the dataset. Thus, if we had just the bias waveform repeated over two cycles, spectroscopic values would contain the bias waveform tiled twice and the cycle numbers repeated as many times as the number of points in the bias waveform. Therefore, extracting the bias waveform or the cycle numbers from the ancillary datasets is not trivial. This problem is especially challenging for multidimensional datasets such as the one under consideration. Fortunately, `hdf_utils` has a very handy function for this as well:

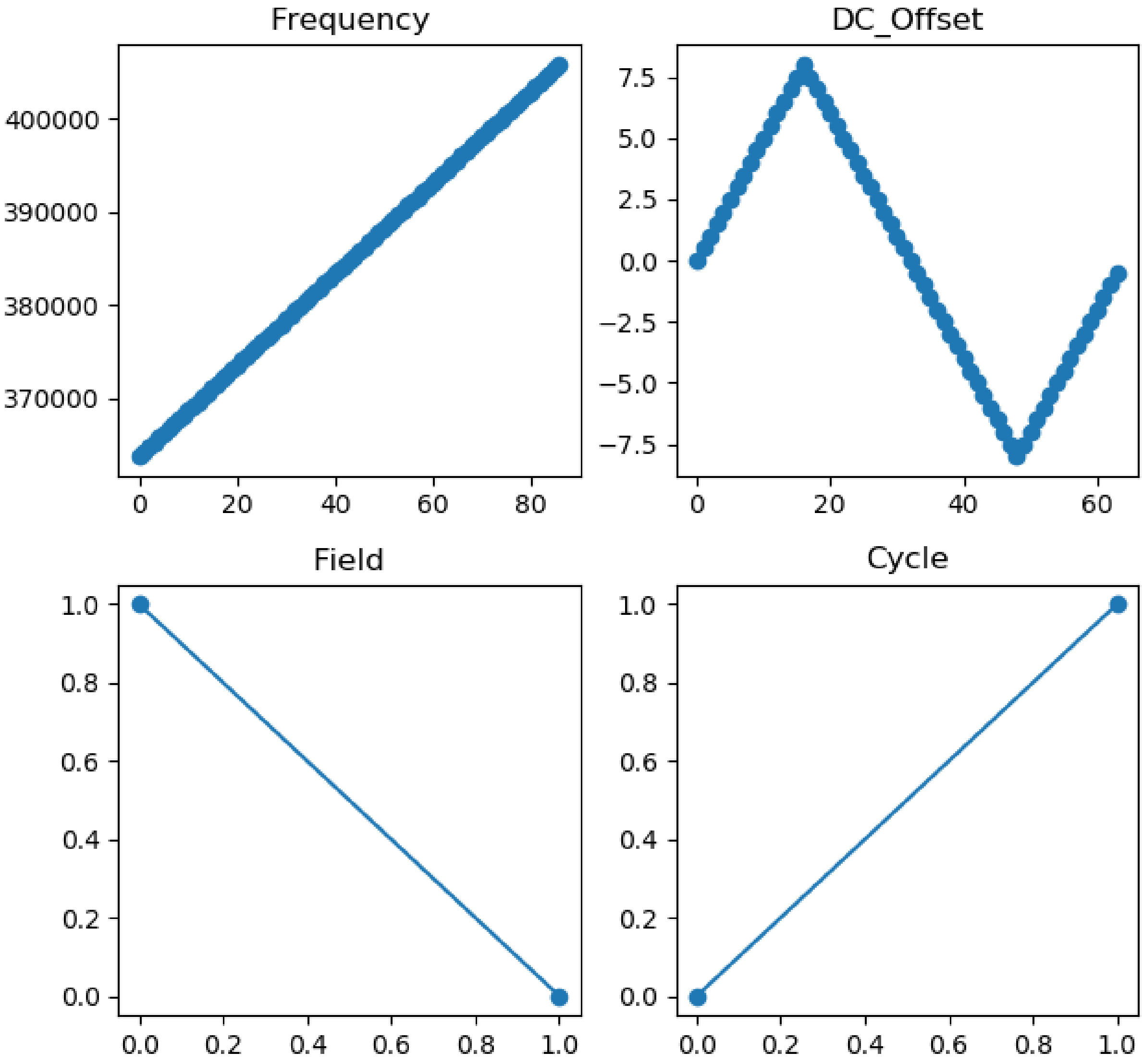
```
pos_unit_values = usid.hdf_utils.get_unit_values(h5_pos_inds, h5_pos_vals)
print('Position unit values:')
for key, val in pos_unit_values.items():
    print('{} : {}'.format(key, val))
spec_unit_values = usid.hdf_utils.get_unit_values(h5_spec_inds, h5_spec_vals)
```

Out:

```
Position unit values:
X : [0. 1. 2. 3. 4.]
Y : [0. 1. 2. 3. 4.]
```

Since the spectroscopic dimensions are quite complicated, lets visualize the results from `get_unit_values()`:

```
fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(6.5, 6))
for axis, name in zip(axes.flat, spec_dim_names):
    axis.set_title(name)
    axis.plot(spec_unit_values[name], 'o-')

fig.suptitle('Spectroscopic Dimensions', fontsize=16, y=1.05)
fig.tight_layout()
```



Reshaping Data

reshape_to_n_dims()

The USID model stores N dimensional datasets in a flattened 2D form of position x spectral values. It can become challenging to retrieve the data in its original N-dimensional form, especially for multidimensional datasets such as the one we are working on. Fortunately, all the information regarding the dimensionality of the dataset are contained in the spectral and position ancillary datasets. `reshape_to_n_dims()` is a very useful function that can help retrieve the N-dimensional form of the data using a simple function call:

```
ndim_form, success, labels = usid.hdf_utils.reshape_to_n_dims(h5_raw, get_labels=True)
if success:
    print('Succeeded in reshaping flattened 2D dataset to N dimensions')
    print('Shape of the data in its original 2D form')
    print(h5_raw.shape)
```

(continues on next page)

(continued from previous page)

```
    print('Shape of the N dimensional form of the dataset:')
    print(ndim_form.shape)
    print('And these are the dimensions')
    print(labels)
else:
    print('Failed in reshaping the dataset')
```

Out:

```
Succeeded in reshaping flattened 2D dataset to N dimensions
Shape of the data in its original 2D form
(25, 22272)
Shape of the N dimensional form of the dataset:
(5, 5, 87, 64, 2, 2)
And these are the dimensions
['X' 'Y' 'Frequency' 'DC_Offset' 'Field' 'Cycle']
```

reshape_from_n_dims()

The inverse problem of reshaping an N dimensional dataset back to a 2D dataset (let's say for the purposes of multivariate analysis or storing into h5USID files) is also easily solved using another handy function - `reshape_from_n_dims()`:

```
two_dim_form, success = usid.hdf_utils.reshape_from_n_dims(ndim_form, h5_pos=h5_pos_
↪inds, h5_spec=h5_spec_inds)
if success:
    print('Shape of flattened two dimensional form')
    print(two_dim_form.shape)
else:
    print('Failed in flattening the N dimensional dataset')
```

Out:

```
Shape of flattened two dimensional form
(25, 22272)
```

Close and delete the h5_file

```
h5_f.close()
os.remove(h5_path)
```

Total running time of the script: (0 minutes 0.897 seconds)

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

06. Utilities for handling data types and transformations

Suhas Somnath

4/18/2018

Introduction

The general nature of the **Universal Spectroscopy and Imaging Data (USID)** model facilitates the representation of any kind of measurement data. This includes:

1. Conventional data represented using floating point numbers such as `1.2345`
2. Integer data (with or without sign) such as `137`
3. Complex-valued data such as `1.23 + 4.5i`
4. Multi-valued or compound valued data cells such as (`'Frequency': 301.2`, `'Amplitude': 1.553E-3`, `'Phase': 2.14`) where a single value or measurement is represented by multiple elements, each with their own names, and data types

While HDF5 datasets are capable of storing all of these kinds of data, many conventional data analysis techniques such as decomposition, clustering, etc. are either unable to handle complicated data types such as complex-valued datasets and compound valued datasets, or the results from these techniques do not produce physically meaningful results. For example, most singular value decomposition algorithms are capable of processing complex-valued datasets. However, while the eigenvectors can have complex values, the resultant complex-valued abundance maps are meaningless. These algorithms would not even work if the original data was compound valued!

To avoid such problems, we need functions that transform the data to and from the necessary type (integer, real-value etc.)

The `pyUSID.dtype_utils` module facilitates comparisons, validations, and most importantly, transformations of one data-type to another. We will be going over the many useful functions in this module and explaining how, when and why one would use them.

Recommended pre-requisite reading

- USID data model
- Crash course on HDF5 and h5py

Import all necessary packages

Before we begin demonstrating the numerous functions in `pyUSID.dtype_utils`, we need to import the necessary packages. Here are a list of packages besides pyUSID that will be used in this example:

- `h5py` - to manipulate HDF5 files
- `numpy` - for numerical operations on arrays in memory

```python
from __future__ import print_function, division, unicode_literals
import os
import subprocess
import sys
def install(package):
    subprocess.call([sys.executable, "-m", "pip", "install", package])

import h5py
import numpy as np
# Finally import pyUSID.
try:
    import pyUSID as usid
except ImportError:
```

(continues on next page)

(continued from previous page)

```
    # Warning package in case something goes wrong
    from warnings import warn
    warn('pyUSID not found.  Will install with pip.')
    import pip
    install('pyUSID')
    import pyUSID as usid
```

Utilities for validating data types

pyUSID.dtype_utils contains some handy functions that make it easy to write robust and safe code by simplifying common data type checking and validation.

contains_integers()

The `contains_integers()` function checks to make sure that each item in a list is indeed an integer. Additionally, it can be configured to ensure that all the values are above a minimum value. This is particularly useful when building indices matrices based on the size of dimensions - specified as a list of integers for example.

```
item = [1, 2, -3, 4]
print('{} : contains integers? : {}'.format(item, usid.dtype_utils.contains_
↪integers(item)))
item = [1, 4.5, 2.2, -1]
print('{} : contains integers? : {}'.format(item, usid.dtype_utils.contains_
↪integers(item)))

item = [1, 5, 8, 3]
min_val = 2
print('{} : contains integers >= {} ? : {}'.format(item, min_val,
                                                 usid.dtype_utils.contains_
↪integers(item, min_val=min_val)))
```

Out:

```
[1, 2, -3, 4] : contains integers? : True
[1, 4.5, 2.2, -1] : contains integers? : False
[1, 5, 8, 3] : contains integers >= 2 ? : False
```

validate_dtype()

The `validate_dtype()` function ensure that a provided object is indeed a valid h5py or numpy data type. When writing a main dataset along with all ancillary datasets, pyUSID meticulously ensures that all inputs are valid before writing data to the file. This comes in very handy when we want to follow the 'measure twice, cut once' ethos.

```
for item in [np.float16, np.complex64, np.uint8, np.int16]:
    print('Is {} a valid dtype? : {}'.format(item, usid.dtype_utils.validate_
↪dtype(item)))


# This function is especially useful on compound or structured data types:

struct_dtype = np.dtype({'names': ['r', 'g', 'b'],
```

(continues on next page)

(continued from previous page)

```
                         'formats': [np.float32, np.uint16, np.float64]})
print('Is {} a valid dtype? : {}'.format(struct_dtype, usid.dtype_utils.validate_
↪dtype(struct_dtype)))
```

Out:

```
Is <class 'numpy.float16'> a valid dtype? : True
Is <class 'numpy.complex64'> a valid dtype? : True
Is <class 'numpy.uint8'> a valid dtype? : True
Is <class 'numpy.int16'> a valid dtype? : True
Is [('r', '<f4'), ('g', '<u2'), ('b', '<f8')] a valid dtype? : True
```

get_compound_sub_dtypes()

One common hassle when dealing with compound / structured array dtypes is that it can be a little challenging to quickly get the individual datatypes of each field in such a data type. The `get_compound_sub_dtypes()` makes this a lot easier:

```
sub_dtypes = usid.dtype_utils.get_compound_sub_dtypes(struct_dtype)
for key, val in sub_dtypes.items():
    print('{} : {}'.format(key, val))
```

Out:

```
r : float32
g : uint16
b : float64
```

is_complex_dtype()

Quite often, we need to treat complex datasets different from compound datasets which themselves need to be treated different from real valued datasets. `is_complex_dtype()` makes it easier to check if a numpy or HDF5 dataset has a complex data type:

```
for dtype in [np.float32, np.float16, np.uint8, np.int16, struct_dtype, bool]:
    print('Is {} a complex dtype?: {}'.format(dtype, (usid.dtype_utils.is_complex_
↪dtype(dtype))))

for dtype in [np.complex, np.complex64, np.complex128, np.complex256]:
    print('Is {} a complex dtype?: {}'.format(dtype, (usid.dtype_utils.is_complex_
↪dtype(dtype))))
```

Out:

```
Is <class 'numpy.float32'> a complex dtype?: False
Is <class 'numpy.float16'> a complex dtype?: False
Is <class 'numpy.uint8'> a complex dtype?: False
Is <class 'numpy.int16'> a complex dtype?: False
Is [('r', '<f4'), ('g', '<u2'), ('b', '<f8')] a complex dtype?: False
Is <class 'bool'> a complex dtype?: False
Is <class 'complex'> a complex dtype?: True
Is <class 'numpy.complex64'> a complex dtype?: True
```

(continues on next page)

(continued from previous page)

```
Is <class 'numpy.complex128'> a complex dtype?: True
Is <class 'numpy.complex256'> a complex dtype?: False
```

Data transformation

Perhaps the biggest benefit of `dtype_utils` is the ability to flatten complex, compound datasets to real-valued datasets and vice versa. As mentioned in the introduction, this is particularly important when attempting to use machine learning algorithms on complex or compound-valued datasets. In order to enable such pipelines, we need functions to transform:

- complex / compound valued datasets to real-valued datasets
- real-valued datasets back to complex / compound valued datasets

flatten_complex_to_real()

As the name suggests, this function stacks the imaginary values of a N-dimensional numpy / HDF5 dataset below its real-values. Thus, applying this function to a complex valued dataset of size `(a, b, c)` would result in a real-valued dataset of shape `(a, b, 2 * c)`:

```
length = 3
complex_array = np.random.randint(-5, high=5, size=length) + 1j * np.random.randint(-
↪5, high=5, size=length)
stacked_real_array = usid.dtype_utils.flatten_complex_to_real(complex_array)
print('Complex value: {} has shape: {}'.format(complex_array, complex_array.shape))
print('Stacked real value: {} has shape: '
      '{}'.format(stacked_real_array, stacked_real_array.shape))
```

Out:

```
Complex value: [-5.-1.j  1.+0.j -1.-2.j] has shape: (3,)
Stacked real value: [-5.  1. -1. -1.  0. -2.] has shape: (6,)
```

flatten_compound_to_real()

This function flattens a compound-valued dataset of shape `(a, b, c)` into a real-valued dataset of shape `(a, b, k * c)` where `k` is the number of fields within the structured array / compound dtype. Here we will demonstrate this on a 1D array of 5 elements each containing 'r', 'g', 'b' fields:

```
num_elems = 5
structured_array = np.zeros(shape=num_elems, dtype=struct_dtype)
structured_array['r'] = np.random.random(size=num_elems) * 1024
structured_array['g'] = np.random.randint(0, high=1024, size=num_elems)
structured_array['b'] = np.random.random(size=num_elems) * 1024
real_array = usid.dtype_utils.flatten_compound_to_real(structured_array)

print('Structured array is of shape {} and have values:'.format(structured_array.
↪shape))
print(structured_array)
print('\nThis array converted to regular scalar matrix has shape: {} and values:'.
↪format(real_array.shape))
print(real_array)
```

Out:

```
Structured array is of shape (5,) and have values:
[(818.8225 , 308, 141.49083787) (117.05978, 936, 836.12060097)
 (623.2409 , 219, 286.13915957) (357.5592 , 871, 918.0494692 )
 (272.34195, 758, 220.45322256)]

This array converted to regular scalar matrix has shape: (15,) and values:
[818.82250977 117.05977631 623.24090576 357.5592041  272.34194946
 308.         936.         219.         871.         758.
 141.49083787 836.12060097 286.13915957 918.0494692  220.45322256]
```

flatten_to_real()

This function checks the data type of the provided dataset and then uses either of the above functions to (if necessary) flatten the dataset into a real-valued matrix. By checking the data type of the dataset, it obviates the need to explicitly call the aforementioned functions (that still do the work). Here is an example of the function being applied to the compound valued numpy array again:

```
real_array = usid.dtype_utils.flatten_to_real(structured_array)
print('Structured array is of shape {} and have values:'.format(structured_array.
↪shape))
print(structured_array)
print('\nThis array converted to regular scalar matrix has shape: {} and values:'.
↪format(real_array.shape))
print(real_array)
```

Out:

```
Structured array is of shape (5,) and have values:
[(818.8225 , 308, 141.49083787) (117.05978, 936, 836.12060097)
 (623.2409 , 219, 286.13915957) (357.5592 , 871, 918.0494692 )
 (272.34195, 758, 220.45322256)]

This array converted to regular scalar matrix has shape: (15,) and values:
[818.82250977 117.05977631 623.24090576 357.5592041  272.34194946
 308.         936.         219.         871.         758.
 141.49083787 836.12060097 286.13915957 918.0494692  220.45322256]
```

The next three functions perform the inverse operation of taking real-valued matrices or datasets and converting them to complex or compound-valued datasets.

stack_real_to_complex()

As the name suggests, this function collapses a N dimensional real-valued array of size `(a, b, 2 * c)` to a complex-valued array of shape `(a, b, c)`. It assumes that the first c values in real-valued dataset are the real components and the following c values are the imaginary components of the complex value. This will become clearer with an example:

```
real_val = np.hstack([5 * np.random.rand(6),
                      7 * np.random.rand(6)])
print('Real valued dataset of shape {}:'.format(real_val.shape))
print(real_val)
```

(continues on next page)

(continued from previous page)

```
comp_val = usid.dtype_utils.stack_real_to_complex(real_val)

print('\nComplex-valued array of shape: {}'.format(comp_val.shape))
print(comp_val)
```

Out:

```
Real valued dataset of shape (12,):
[0.23024321 0.30219327 1.36709669 3.02744143 3.2431705  4.68102575
 6.38175739 1.56057158 6.70709248 4.46336275 5.02398342 0.63507291]

Complex-valued array of shape: (6,)
[0.23024321+6.38175739j 0.30219327+1.56057158j 1.36709669+6.70709248j
 3.02744143+4.46336275j 3.2431705 +5.02398342j 4.68102575+0.63507291j]
```

stack_real_to_compound()

Similar to the above function, this function shrinks the last axis of a real valued dataset to create the desired compound valued dataset. Here we will demonstrate it on the same 3-field `(r,g,b)` compound datatype:

```
num_elems = 5
real_val = np.concatenate((np.random.random(size=num_elems) * 1024,
                           np.random.randint(0, high=1024, size=num_elems),
                           np.random.random(size=num_elems) * 1024))
print('Real valued dataset of shape {}:'.format(real_val.shape))
print(real_val)

comp_val = usid.dtype_utils.stack_real_to_compound(real_val, struct_dtype)

print('\nStructured array of shape: {}'.format(comp_val.shape))
print(comp_val)
```

Out:

```
Real valued dataset of shape (15,):
[981.36521359 720.1854334  988.12682696 259.07465697 419.67539727
 115.         509.         477.         809.         472.
 188.44645763 330.20431081 448.51941018 503.20624415 867.90883715]

Structured array of shape: (5,)
[(981.36523, 115, 188.44645763) (720.1854 , 509, 330.20431081)
 (988.12683, 477, 448.51941018) (259.07465, 809, 503.20624415)
 (419.67538, 472, 867.90883715)]
```

stack_real_to_target_dtype()

This function performs the inverse of `flatten_to_real()` - stacks the provided real-valued dataset into a complex or compound valued dataset using the two above functions. Note that unlike `flatten_to_real()`, the target data type must be supplied to the function for this to work:

```
print('Real valued dataset of shape {}:'.format(real_val.shape))
print(real_val)

```

(continues on next page)

(continued from previous page)

```
comp_val = usid.dtype_utils.stack_real_to_target_dtype(real_val, struct_dtype)

print('\nStructured array of shape: {}'.format(comp_val.shape))
print(comp_val)
```

Out:

```
Real valued dataset of shape (15,):
[981.36521359 720.1854334  988.12682696 259.07465697 419.67539727
 115.         509.         477.         809.         472.
 188.44645763 330.20431081 448.51941018 503.20624415 867.90883715]

Structured array of shape: (5,)
[(981.36523, 115, 188.44645763) (720.1854 , 509, 330.20431081)
 (988.12683, 477, 448.51941018) (259.07465, 809, 503.20624415)
 (419.67538, 472, 867.90883715)]
```

check_dtype()

`check_dtype()` is a master function that figures out the data type, necessary function to transform a HDF5 dataset to a real-valued array, expected data shape, etc. Before we demonstrate this function, we need to quickly create an example HDF5 dataset.

```
file_path = 'dtype_utils_example.h5'
if os.path.exists(file_path):
    os.remove(file_path)
with h5py.File(file_path) as h5_f:
    num_elems = (5, 7)
    structured_array = np.zeros(shape=num_elems, dtype=struct_dtype)
    structured_array['r'] = 450 * np.random.random(size=num_elems)
    structured_array['g'] = np.random.randint(0, high=1024, size=num_elems)
    structured_array['b'] = 3178 * np.random.random(size=num_elems)
    _ = h5_f.create_dataset('compound', data=structured_array)
    _ = h5_f.create_dataset('real', data=450 * np.random.random(size=num_elems),␣
↪dtype=np.float16)
    _ = h5_f.create_dataset('complex', data=np.random.random(size=num_elems) + 1j *␣
↪np.random.random(size=num_elems),
                            dtype=np.complex64)
    h5_f.flush()
```

Now, lets test the the function on compound-, complex-, and real-valued HDF5 datasets:

```
def check_dataset(h5_dset):
    print('\tDataset being tested: {}'.format(h5_dset))
    func, is_complex, is_compound, n_features, type_mult = usid.dtype_utils.check_
↪dtype(h5_dset)
    print('\tFunction to transform to real: %s' % func)
    print('\tis_complex? %s' % is_complex)
    print('\tis_compound? %s' % is_compound)
    print('\tShape of dataset in its current form: {}'.format(h5_dset.shape))
    print('\tAfter flattening to real, shape is expected to be: ({}, {})'.format(h5_
↪dset.shape[0], n_features))
    print('\tByte-size of a single element in its current form: {}'.format(type_mult))
```

(continues on next page)

(continued from previous page)

```
with h5py.File(file_path, mode='r') as h5_f:
    print('Checking a compound-valued dataset:')
    check_dataset(h5_f['compound'])
    print('')
    print('Checking a complex-valued dataset:')
    check_dataset(h5_f['complex'])
    print('')
    print('Checking a real-valued dataset:')
    check_dataset(h5_f['real'])
os.remove(file_path)
```

Out:

```
Checking a compound-valued dataset:
        Dataset being tested: <HDF5 dataset "compound": shape (5, 7), type "|V14">
        Function to transform to real: <function flatten_compound_to_real at␣
↪0x7f03ca849d90>
        is_complex? False
        is_compound? True
        Shape of dataset in its current form: (5, 7)
        After flattening to real, shape is expected to be: (5, 21)
        Byte-size of a single element in its current form: 12

Checking a complex-valued dataset:
        Dataset being tested: <HDF5 dataset "complex": shape (5, 7), type "<c8">
        Function to transform to real: <function flatten_complex_to_real at␣
↪0x7f03ca849d08>
        is_complex? True
        is_compound? False
        Shape of dataset in its current form: (5, 7)
        After flattening to real, shape is expected to be: (5, 14)
        Byte-size of a single element in its current form: 8

Checking a real-valued dataset:
        Dataset being tested: <HDF5 dataset "real": shape (5, 7), type "<f2">
        Function to transform to real: <class 'numpy.float32'>
        is_complex? False
        is_compound? False
        Shape of dataset in its current form: (5, 7)
        After flattening to real, shape is expected to be: (5, 7)
        Byte-size of a single element in its current form: 4
```

Total running time of the script: (0 minutes 0.012 seconds)

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

07. Speed up computations with parallel_compute()

Suhas Somnath, Chris R. Smith

9/8/2017

This document will demonstrate how ‘‘pyUSID.parallel_compute()‘‘ can significantly speed up data processing by using all available CPU cores in a computer

Introduction

Quite often, we need to perform the same operation on every single component in our data. One of the most popular examples is functional fitting applied to spectra collected at each location on a grid. While, the operation itself may not take very long, computing this operation thousands of times, once per location, using a single CPU core can take a long time to complete. Most personal computers today come with at least two cores, and in many cases, each of these cores is represented via two logical cores, thereby summing to a total of at least four cores. Thus, it is prudent to make use of these unused cores whenever possible. Fortunately, there are a few python packages that facilitate the efficient use of all CPU cores with minimal modifications to the existing code.

`pyUSID.parallel_compute()` is a very handy function that simplifies parallel computation significantly to a `single function call` and will be discussed in this document.

Example scientific problem

For this example, we will be working with 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 **Universal Spectroscopy and Imaging Data (USID)** model.

Each spectra in this dataset is expected to have a single peak. The goal is to find the positions of the peaks in each spectra. Clearly, the operation of finding the peak in one spectra is independent of the same operation on another spectra. Thus, we could in theory divide the dataset in to N parts and use N CPU cores to compute the results much faster than it would take a single core to compute the results. There is an important caveat to this statement and it will be discussed at the end of this document.

`Here, we will learn how to fit the thousands of spectra using all available cores on a computer.` Note, that this is applicable only for a single CPU. Please refer to another advanced example for multi-CPU computing.

```python
# Ensure python 3 compatibility:
from __future__ import division, print_function, absolute_import, unicode_literals

# The package for accessing files in directories, etc.:
import os

# Warning package in case something goes wrong
from warnings import warn
import subprocess
import sys

def install(package):
    subprocess.call([sys.executable, "-m", "pip", "install", package])
# Package for downloading online files:
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

# The mathematical computation package:
import numpy as np
```

(continues on next page)

(continued from previous page)

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

# Packages for plotting:
import matplotlib.pyplot as plt

# Parallel computation library:
try:
    import joblib
except ImportError:
    warn('joblib not found.  Will install with pip.')
    import pip
    install('joblib')
    import joblib

# Timing
import time

# A handy python utility that allows us to preconfigure parts of a function
from functools import partial

# Finally import pyUSID:
try:
    import pyUSID as usid
except ImportError:
    warn('pyUSID not found.  Will install with pip.')
    import pip
    install('pyUSID')
    import pyUSID as usid

# import the scientific function:
import sys
sys.path.append('./supporting_docs/')
from peak_finding import find_all_peaks
```

Load the dataset

In order to demonstrate parallel computing, we will be using a real experimental dataset that is available on the pyUSID GitHub project. First, lets download this file from Github:

```
h5_path = 'temp.h5'
url = 'https://raw.githubusercontent.com/pycroscopy/pyUSID/master/data/BELine_0004.h5'
if os.path.exists(h5_path):
    os.remove(h5_path)
_ = wget.download(url, h5_path, bar=None)
```

Now, lets open this HDF5 file. The focus of this example is not on the data storage or arrangement but rather on demonstrating parallel computation so lets dive straight into the main dataset that requires fitting of the spectra:

```
# Open the file in read-only mode
h5_file = h5py.File(h5_path, mode='r')
# Get handle to the the raw data
h5_meas_grp = h5_file['Measurement_000']
```

(continues on next page)

(continued from previous page)

```
# Accessing the dataset of interest:
h5_main = usid.USIDataset(h5_meas_grp['Channel_000/Raw_Data'])
print('\nThe main dataset:\n------------------------------------')
print(h5_main)

num_rows, num_cols = h5_main.pos_dim_sizes
cores_vec = list()
times_vec = list()
```

Out:

```
The main dataset:
------------------------------------
<HDF5 dataset "Raw_Data": shape (16384, 119), type "<c8">
located at:
        /Measurement_000/Channel_000/Raw_Data
Data contains:
        Cantilever Vertical Deflection (V)
Data dimensions and original shape:
Position Dimensions:
        X - size: 128
        Y - size: 128
Spectroscopic Dimensions:
        Frequency - size: 119
Data Type:
        complex64
```

The operation

The scipy package has a very handy function called *find_peaks_cwt()* that facilitates the search for one or more peaks in a spectrum. We will be using a function called *find_all_peaks()* that uses *find_peaks_cwt()*. For the purposes of this example, we do not be concerned with how this function works. All we need to know is that this function takes 3 inputs:

- `vector` - a 1D array containing the spectra at a single location
- `width_bounds` - something like [20, 50] that instructs the function to look for peaks that are 20-50 data-points wide. The function will look for a peak with width of 20, then again for a peak of width - 21 and so on.
- `num_steps` - The number of steps within the possible widths [20, 50], that the search must be performed

The function has one output:

- `peak_indices` - an array of the positions at which peaks were found.

```
def find_all_peaks(vector, width_bounds, num_steps=20, **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
    width_bounds : tuple / list / iterable
```

(continues on next page)

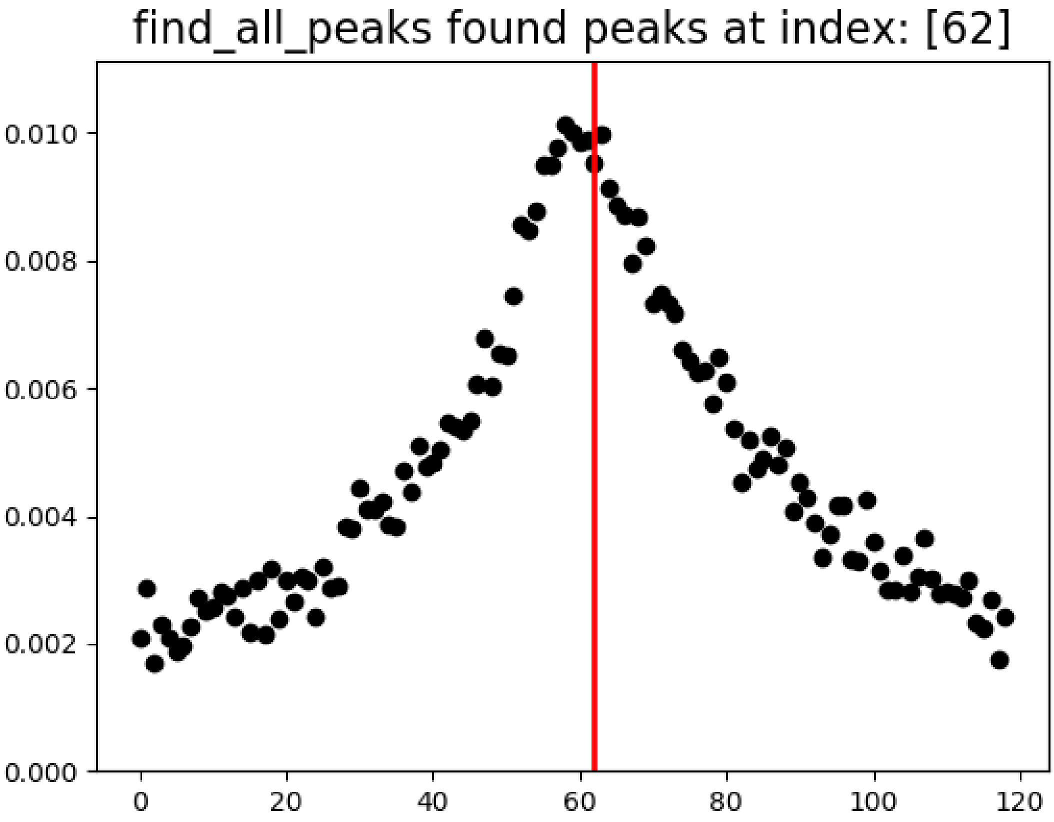
(continued from previous page)

```
        Min and max for the size of the window
    num_steps : uint, (optional). Default = 20
        Number of different peak widths to search

    Returns
    --------
    peak_indices : list
        List of indices of peaks within the prescribed peak widths
    """
    # The below numpy array is used to configure the returned function wpeaks
    wavelet_widths = np.linspace(width_bounds[0], width_bounds[1], num_steps)

    peak_indices = find_peaks_cwt(np.abs(vector), wavelet_widths, **kwargs)

    return peak_indices
```

Testing the function

Let’s see what the operation on an example spectra returns.

```
row_ind, col_ind = 103, 19
pixel_ind = col_ind + row_ind * num_cols
spectra = h5_main[pixel_ind]

peak_inds = find_all_peaks(spectra, [20, 60], num_steps=30)

fig, axis = plt.subplots()
axis.scatter(np.arange(len(spectra)), np.abs(spectra), c='black')
axis.axvline(peak_inds[0], color='r', linewidth=2)
axis.set_ylim([0, 1.1 * np.max(np.abs(spectra))]);
axis.set_title('find_all_peaks found peaks at index: {}'.format(peak_inds),␣
↪fontsize=16)
```



Before we apply the function to the entire dataset, lets load the dataset to memory so that file-loading time is not a factor when comparing the times for serial and parallel computing times:

```
raw_data = h5_main[()]
```

Serial computing

A single call to the function does not take substantial time. However, performing the same operation on each of the `16,384` pixels sequentially can take substantial time. The simplest way to find all peak positions is to simply loop over each position in the dataset:

```
serial_results = list()

t_0 = time.time()
for vector in raw_data:
    serial_results.append(find_all_peaks(vector, [20, 60], num_steps=30))
times_vec.append(time.time()-t_0)
cores_vec.append(1)
print('Serial computation took', np.round(times_vec[-1], 2), ' seconds')
```

Out:

```
Serial computation took 58.97  seconds
```

pyUSID.parallel_compute()

There are several libraries that can utilize multiple CPU cores to perform the same computation in parallel. Popular examples are `Multiprocessing`, `Mutiprocess`, `Dask`, `Joblib` etc. Each of these has their own strengths and weaknesses. Some of them have painful caveats such as the inability to perform the parallel computation within a jupyter notebook. In order to lower the barrier to parallel computation, we have developed a very handy function called `pyUSID.parallel_compute()` that simplifies the process to a single function call.

It is a lot **more straightforward** to provide the arguments and keyword arguments of the function that needs to be applied to the entire dataset. Furthermore, this function intelligently assigns the number of CPU cores for the parallel computation based on the size of the dataset and the computational complexity of the unit computation. For instance, it scales down the number of cores for small datasets if each computation is short. It also ensures that 1-2 cores fewer than all available cores are used by default so that the user can continue using their computer for other purposes while the computation runs.

Lets apply this `parallel_compute` to this problem:

```
cpu_cores = 2
args = [[20, 60]]
kwargs = {'num_steps': 30}

t_0 = time.time()

# Execute the parallel computation
parallel_results = usid.parallel_compute(raw_data, find_all_peaks, cores=cpu_cores,␣
↪func_args=args, func_kwargs=kwargs)

cores_vec.append(cpu_cores)
times_vec.append(time.time()-t_0)
print('Parallel computation with {} cores took {} seconds'.format(cpu_cores, np.
↪round(times_vec[-1], 2)))
```

Out:

```
Starting computing on 2 cores (requested 2 cores)
Finished parallel computation
Parallel computation with 2 cores took 35.68 seconds
```

Compare the results

By comparing the run-times for the two approaches, we see that the parallel computation is substantially faster than the serial computation. Note that the numbers will differ between computers. Also, the computation was performed on a relatively small dataset for illustrative purposes. The benefits of using such parallel computation will be far more apparent for much larger datasets.

Let’s compare the results from both the serial and parallel methods to ensure they give the same results:

```
print('Result from serial computation: {}'.format(serial_results[pixel_ind]))
print('Result from parallel computation: {}'.format(parallel_results[pixel_ind]))
```

Out:

```
Result from serial computation: [62]
Result from parallel computation: [62]
```

Simplifying the function

Note that the `width_bounds` and `num_steps` arguments will not be changed from one pixel to another. It would be great if we didn't have to keep track of these constant arguments. We can use a very handy python tool called `partial()` to do just this. Below, all we are doing is creating a new function that always passes our preferred values for `width_bounds` and `num_steps` arguments to find_all_peaks. While it may seem like this is unimportant, it is very convenient when setting up the parallel computing:

```
find_peaks = partial(find_all_peaks, num_steps=30, width_bounds=[20, 60])
```

Notice that even though `width_bounds` is an argument, it needs to be specified as though it were a keyword argument like `num_steps`. Let's try calling our simplified function, `find_peaks()` to make sure that it results in the same peak index for the aforementioned chosen spectra:

```
print('find_peaks found peaks at index: {}'.format(find_peaks(h5_main[pixel_ind])))
```

Out:

```
find_peaks found peaks at index: [62]
```

More cores!

Lets use `find_peaks()` instead of `find_all_peaks` on the entire dataset but increase the number of cores to 3. Note that we do not need to specify `func_kwargs` anymore. Also note that this is a very simple function and the benefits of `partial()` will be greater for more complex problems.

```
cpu_cores = 3

t_0 = time.time()

# Execute the parallel computation
parallel_results = usid.parallel_compute(raw_data, find_peaks, cores=cpu_cores)

cores_vec.append(cpu_cores)
times_vec.append(time.time()-t_0)
print('Parallel computation with {} cores took {} seconds'.format(cpu_cores, np.
↪round(times_vec[-1], 2)))
```

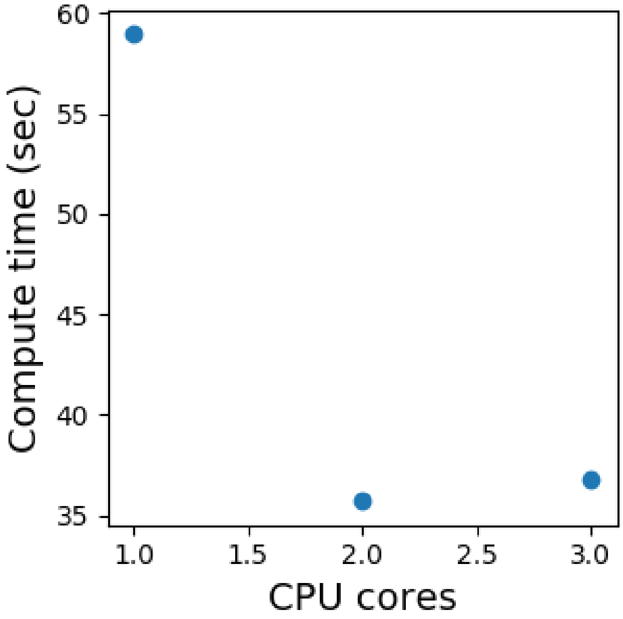
Out:

```
Starting computing on 3 cores (requested 3 cores)
Finished parallel computation
Parallel computation with 3 cores took 36.81 seconds
```

Scalability

Now lets see how the computational time relates to the number of cores. Depending on your computer (and what was running on your computer along with this computation), you are likely to see diminishing benefits of additional cores beyond 2 cores for this specific problem in the plot below. This is because the dataset is relatively small and each peak-finding operation is relatively quick. The overhead of adding additional cores quickly outweighs the speedup in distributing the work among multiple CPU cores.

```
fig, axis = plt.subplots(figsize=(3.5, 3.5))
axis.scatter(cores_vec, times_vec)
axis.set_xlabel('CPU cores', fontsize=14)
axis.set_ylabel('Compute time (sec)', fontsize=14)
fig.tight_layout()
```



Best practices for parallel computing

While it may seem tempting to do everything in parallel, it is important to be aware of some of the trade-offs and best-practices for parallel computing (multiple CPU cores) when compared to traditional serial computing (single CPU core):

- There is noticeable time overhead involved with setting up each parallel computing job. For very simple or small computations, this overhead may outweigh the speed-up gained with using multiple cores.
- Parallelizing computations that read and write to files at each iteration may be actually be noticeably *slower* than serial computation since each core will compete with all other cores for rights to read and write to the file(s) and these input/output operations are by far the slowest components of the computation. Instead, it makes sense to read large amounts of data from the necessary files once, perform the computation, and then write to the files once after all the computation is complete. In fact, this is what we automatically do in the `Fitter` and `Process` classes in pycroscopy

Formalizing data processing and pyUSID.Process

Data processing / analysis typically involves a few basic operations:

1. Reading data from file
2. Parallel computation
3. Writing results to disk

The Process class in pyUSID has modularized these operations for simpler and faster development of standardized, easy-to-debug code. In the case of this example, one would only need to write the find_all_peaks() function along with the appropriate data reading and data writing functions. Other common operations can be inherited from pyUSID.Process.

Please see another example on how to write a Process class for pyUSID based on this example

Lets not forget to close and delete the temporarily downloaded file:

```
h5_file.close()
os.remove(h5_path)
```

Total running time of the script: (2 minutes 13.189 seconds)

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

08. Utilities that assist in writing USID data

Suhas Somnath

4/18/2018

This document illustrates certain helper functions that simplify writing data to Universal Spectroscopy and Imaging Data (USID) HDF5 files or h5USID files

Introduction

The USID model takes a truly unique approach towards towards storing scientific observational data. Several helper functions are necessary to simplify the actual file writing process. `pyUSID.write_utils` is the home for those utilities that assist in writing data but do not directly interact with HDF5 files (as in `pyUSID.hdf_utils`). `pyUSID.write_utils` consist mainly of two broad categories of functions: * functions that assist in building ancillary datasets * miscellaneous functions that assist in formatting data

Note that most of these are low-level functions that are used by popular high level functions in pyUSID.hdf_utils to simplify the writing of datasets.

Recommended pre-requisite reading

- USID model

What to read after this

- Crash course on HDF5 and h5py
- Utilities for reading and writing h5USID files

Import necessary packages

We only need a handful of packages besides pyUSID to illustrate the functions in `pyUSID.write_utils`:

- `numpy` - for numerical operations on arrays in memory

- `matplotlib` - basic visualization of data

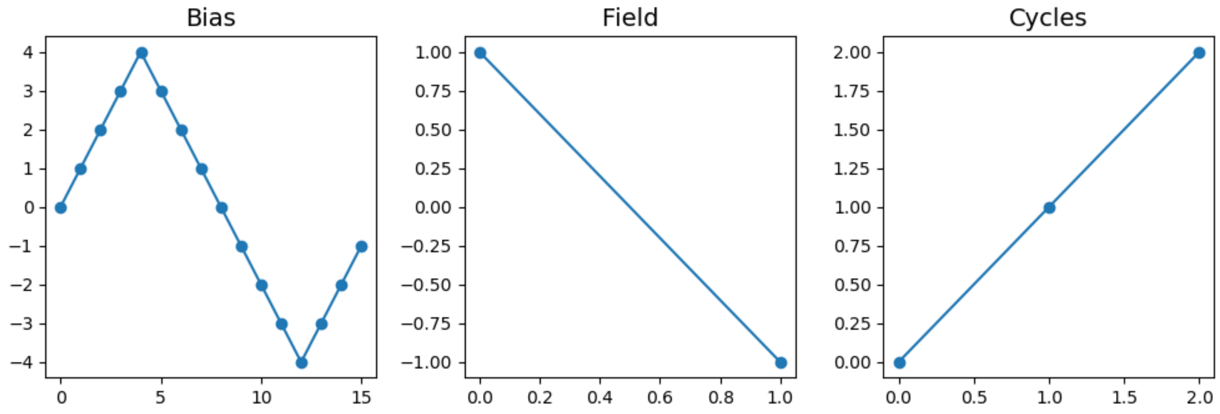
```
from __future__ import print_function, division, unicode_literals
# Warning package in case something goes wrong
from warnings import warn
import numpy as np
import matplotlib.pyplot as plt
import subprocess
import sys


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

Building Ancillary datasets

The USID model uses pairs of `ancillary matrices` to support the instrument-agnostic and compact representation of multidimensional datasets. While the creation of `ancillary datasets` is straightforward when the number of position and spectroscopic dimensions are relatively small (0-2), one needs to be careful when building these `ancillary datasets` for datasets with large number of position / spectroscopic dimensions (> 2). The main challenge involves the careful tiling and repetition of unit vectors for each dimension with respect to the sizes of all other dimensions. Fortunately, `pyUSID.write_utils` has many handy functions that solve this problem.

In order to demonstrate the functions, lets say that we are working on an example `Main dataset` that has three spectroscopic dimensions (`Bias`, `Field`, `Cycle`). The Bias dimension varies as a bi-polar triangular waveform. This waveform is repeated for two Fields over 3 Cycles meaning that the `Field` and `Cycle` dimensions are far simpler than the `Bias` dimension in that they have linearly increasing / decreasing values.

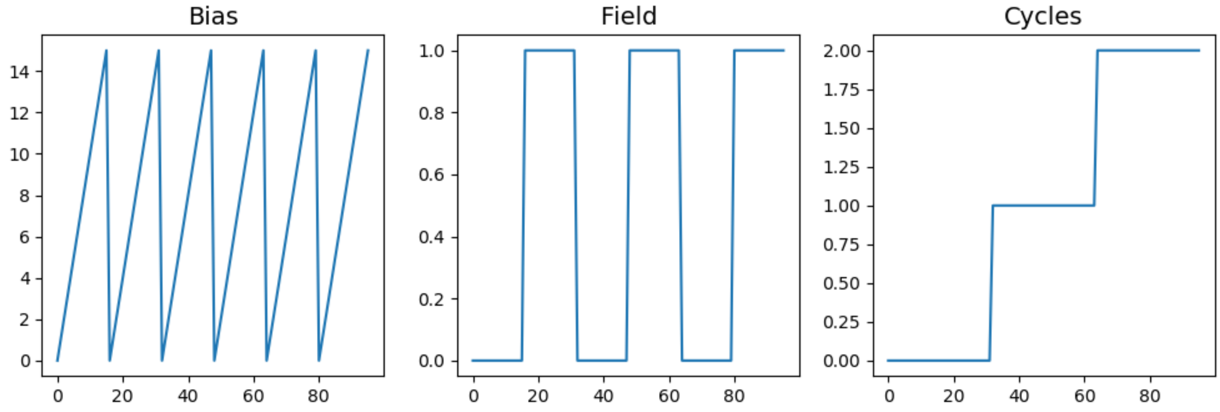
```
max_v = 4
half_pts = 8
bi_triang = np.roll(np.hstack((np.linspace(-max_v, max_v, half_pts, endpoint=False),
                               np.linspace(max_v, -max_v, half_pts, endpoint=False))),
↪ -half_pts // 2)
cycles = [0, 1, 2]
fields = [1, -1]

dim_names = ['Bias', 'Field', 'Cycles']

fig, axes = plt.subplots(ncols=3, figsize=(10, 3.5))
for axis, name, vec in zip(axes.flat, dim_names, [bi_triang, fields, cycles]):
    axis.plot(vec, 'o-')
    axis.set_title(name, fontsize=14)
fig.suptitle('Unit values for each dimension', fontsize=16, y=1.05)
fig.tight_layout()
```



make_indices_matrix()

One half of the work for generating the ancillary datasets is generating the `indices matrix`. The `make_indices_matrix()` function solves this problem. All one needs to do is supply a list with the lengths of each dimension. The result is a 2D matrix of shape: (3 dimensions, points in `Bias``[``16`] * points in `Field``[``2`] * points in `Cycle``[``3`]) = `(3, 96)`.

```
inds = usid.write_utils.make_indices_matrix([len(bi_triang), len(fields),␣
↪len(cycles)], is_position=False)
print('Generated indices of shape: {}'.format(inds.shape))

# The plots below show a visual representation of the indices for each dimension:
fig, axes = plt.subplots(ncols=3, figsize=(10, 3.5))
for axis, name, vec in zip(axes.flat, dim_names, inds):
    axis.plot(vec)
    axis.set_title(name, fontsize=14)
fig.suptitle('Indices for each dimension', fontsize=16, y=1.05)
fig.tight_layout()
```



Out:

```
Generated indices of shape: (3, 96)
```

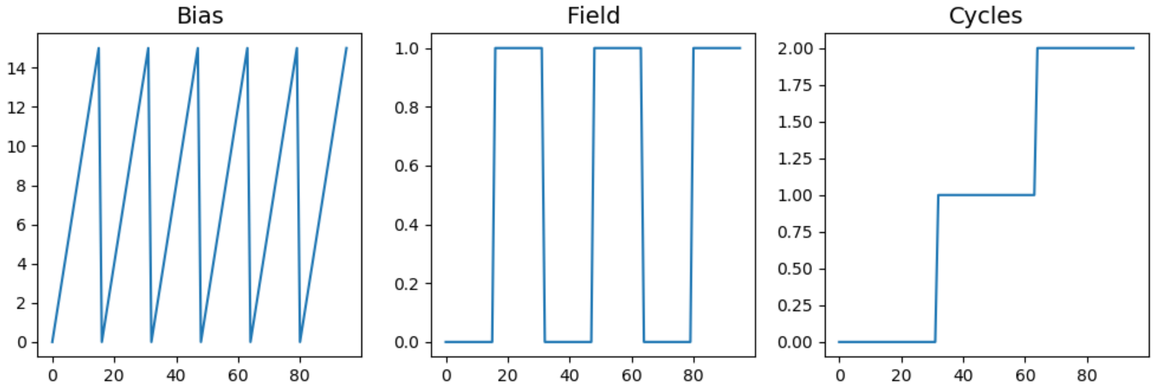
build_ind_val_matrices()

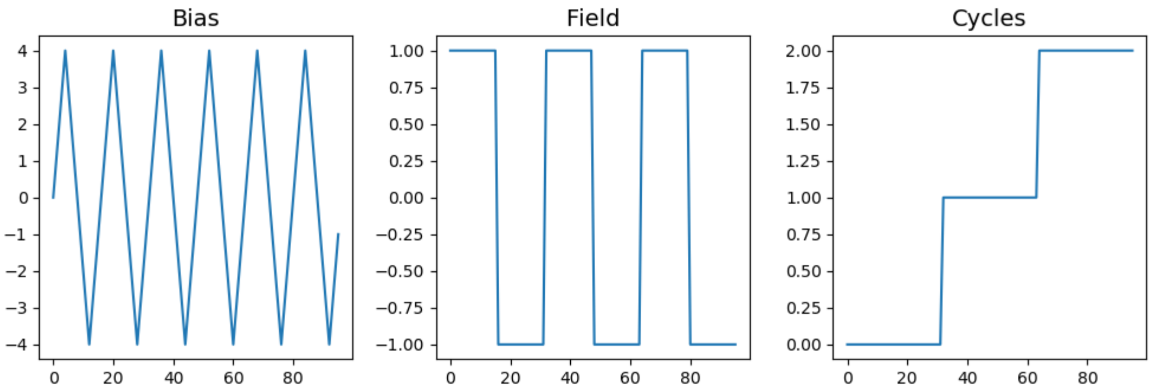
make_indices_matrix() is a very handy function but it only solves one of the problems - the indices matrix. We also need the matrix with the values tiled and repeated in the same manner. Perhaps one of the most useful functions is build_ind_val_matrices() which uses just the values over which each dimension is varied to automatically generate the indices and values matrices that form the ancillary datasets.

In order to generate the indices and values matrices, we would just need to provide the list of values over which these dimensions are varied to build_ind_val_matrices(). The results are two matrices - one for the indices and the other for the values, of the same shape (3, 96).

As mentioned in our document about the data structuring, the Bias would be in the first row, followed by Field, finally followed by Cycle. The plots below illustrate what the indices and values look like for each dimension. For example, notice how the bipolar triangular bias vector has been repeated 2 (Field) * 3 (Cycle) times. Also note how the indices vector is a saw-tooth waveform that also repeats in the same manner. The repeated + tiled indices and values vectors for Cycle and Field look the same / very similar since they were simple linearly increasing values to start with.

```python
inds, vals = usid.write_utils.build_ind_val_matrices([bi_triang, fields, cycles], is_
↪spectral=True)
print('Indices and values of shape: {}'.format(inds.shape))

fig, axes = plt.subplots(ncols=3, figsize=(10, 3.5))
for axis, name, vec in zip(axes.flat, dim_names, inds):
    axis.plot(vec)
    axis.set_title(name, fontsize=14)
fig.suptitle('Indices for each dimension', fontsize=16, y=1.05)
fig.tight_layout()

fig, axes = plt.subplots(ncols=3, figsize=(10, 3.5))
for axis, name, vec in zip(axes.flat, dim_names, vals):
    axis.plot(vec)
    axis.set_title(name, fontsize=14)
fig.suptitle('Values for each dimension', fontsize=16, y=1.05)
fig.tight_layout()
```

- 



-

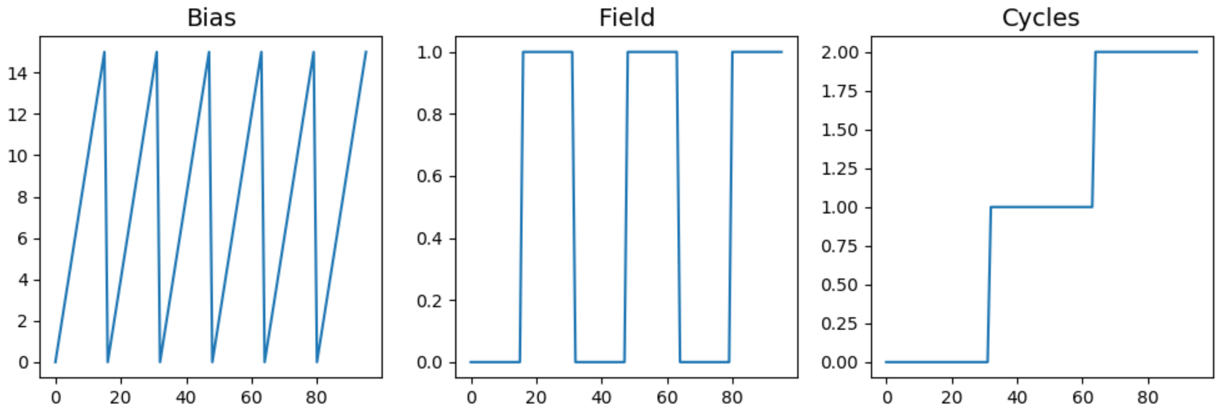
Out:

```
Indices and values of shape: (3, 96)
```

create_spec_inds_from_vals()

When writing analysis functions or classes wherein one or more (typically spectroscopic) dimensions are dropped as a consequence of dimensionality reduction, new ancillary spectroscopic datasets need to be generated when writing the reduced data back to the file. `create_spec_inds_from_vals()` is a handy function when we need to generate the indices matrix that corresponds to a values matrix. For this example, lets assume that we only have the values matrix but need to generate the indices matrix from this:

```
inds = usid.write_utils.create_spec_inds_from_vals(vals)

fig, axes = plt.subplots(ncols=3, figsize=(10, 3.5))
for axis, name, vec in zip(axes.flat, dim_names, inds):
    axis.plot(vec)
    axis.set_title(name, fontsize=14)
fig.suptitle('Indices for each dimension', fontsize=16, y=1.05)
fig.tight_layout()
```



get_aux_dset_slicing()

Region references are a handy feature in HDF5 that allow users to refer to a certain section of data within a dataset by name rather than the indices that define the region of interest.

In USID, we use region references to define the row or column in the ancillary dataset that corresponds to each dimension by its name. In other words, if we only wanted the Field dimension we could directly get the data corresponding to this dimension without having to remember or figure out the index in which this dimension exists.

Let's take the example of the Field dimension which occurs at index 1 in the (3, 96) shaped spectroscopic index / values matrix. We could extract the Field dimension from the 2D matrix by slicing each dimension using slice objects . We need the second row so the first dimension would be sliced as slice(start=1, stop=2). We need all the colummns in the second dimension so we would slice as slice(start=None, stop=None) meaning that we need all the columns. Doing this by hand for each dimension is clearly tedious.

get_aux_dset_slicing() helps generate the instructions for region references for each dimension in an ancillary dataset. The instructions for each region reference in h5py are defined by tuples of slice objects. Lets see the region-reference instructions that this function provides for our aforementioned example dataset with the three spectroscopic dimensions.

```
print('Region references slicing instructions for Spectroscopic dimensions:')
ret_val = usid.write_utils.get_aux_dset_slicing(dim_names, is_spectroscopic=True)
for key, val in ret_val.items():
    print('{} : {}'.format(key, val))

print('\nRegion references slicing instructions for Position dimensions:')
ret_val = usid.write_utils.get_aux_dset_slicing(['X', 'Y'], is_spectroscopic=False)
for key, val in ret_val.items():
    print('{} : {}'.format(key, val))
```

Out:

```
Region references slicing instructions for Spectroscopic dimensions:
Bias : (slice(0, 1, None), slice(None, None, None))
Field : (slice(1, 2, None), slice(None, None, None))
Cycles : (slice(2, 3, None), slice(None, None, None))

Region references slicing instructions for Position dimensions:
X : (slice(None, None, None), slice(0, 1, None))
Y : (slice(None, None, None), slice(1, 2, None))
```

Dimension

In USID, position and spectroscopic dimensions are defined using some basic information that will be incorporated in Dimension objects that contain three vital pieces of information:

- name of the dimension
- units for the dimension
- **values:**
    - These can be the actual values over which the dimension was varied
    - or number of steps in case of linearly varying dimensions such as 'Cycle' below

These objects will be heavily used for creating Main or ancillary datasets in pyUSID.hdf_utils and even to set up interactive jupyter Visualizers in pyUSID.USIDataset.

Note that the Dimension objects in the lists for Position and Spectroscopic must be arranged from fastest varying to slowest varying to mimic how the data is actually arranged. For example, in this example, there are multiple bias points per field and multiple fields per cycle. Thus, the bias changes faster than the field and the field changes faster than the cycle. Therefore, the Bias must precede Field which will precede Cycle. Let's assume that we were describing the spectroscopic dimensions for this example dataset to some other pyUSID function , we would describe the spectroscopic dimensions as:

```
spec_dims = [usid.write_utils.Dimension('Bias', 'V', bi_triang),
             usid.write_utils.Dimension('Fields', '', fields),
             # for the sake of example, since we know that cycles is linearly␣
↪increasing from 0 with a step size of 1,
             # we can specify such a simply dimension via just the length of that␣
↪dimension:
             usid.write_utils.Dimension('Cycle', '', len(cycles))]
```

The application of the Dimension objects will be a lot more apparent in the document about the writing functions in pyUSID.hdf_utils.

Misc writing utilities

calc_chunks()

The h5py package automatically (virtually) breaks up HDF5 datasets into contiguous chunks to speed up reading and writing of datasets. In certain situations the default mode of chunking may not result in the highest performance. In such cases, it helps in chunking the dataset manually. The calc_chunks() function helps in calculating appropriate chunk sizes for the dataset using some apriori knowledge about the way the data would be accessed, the size of each element in the dataset, maximum size for a single chunk, etc. The examples below illustrate a few ways on how to use this function:

```
dimensions = (16384, 16384 * 4)
dtype_bytesize = 4
ret_val = usid.write_utils.calc_chunks(dimensions, dtype_bytesize)
print(ret_val)

dimensions = (16384, 16384 * 4)
dtype_bytesize = 4
unit_chunks = (3, 7)
max_mem = 50000
ret_val = usid.write_utils.calc_chunks(dimensions, dtype_bytesize, unit_chunks=unit_
↪chunks, max_chunk_mem=max_mem)
print(ret_val)
```

Out:

```
(26, 100)
(57, 224)
```

clean_string_att()

As mentioned in our HDF5 primer, the h5py package used for reading and manipulating HDF5 files has issues which necessitate the encoding of attributes whose values are lists of strings. The clean_string_att() encodes lists of strings correctly so that they can directly be written to HDF5 without causing any errors. All other kinds of simple attributes - single strings, numbers, lists of numbers are unmodified by this function.

```
expected = ['a', 'bc', 'def']
returned = usid.write_utils.clean_string_att(expected)
print('List of strings value: {} encoded to: {}'.format(expected, returned))

expected = [1, 2, 3.456]
returned = usid.write_utils.clean_string_att(expected)
print('List of numbers value: {} returned as is: {}'.format(expected, returned))
```

Out:

```
List of strings value: ['a', 'bc', 'def'] encoded to: [b'a' b'bc' b'def']
List of numbers value: [1, 2, 3.456] returned as is: [1, 2, 3.456]
```

Total running time of the script: (0 minutes 0.374 seconds)

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

09. Utilities for writing h5USID files

Suhas Somnath

4/18/2018

This document illustrates the many handy functions in pyUSID.hdf_utils that significantly simplify writing data and information into Universal Spectroscopy and Imaging Data (USID) HDF5 files (h5USID files)

Introduction

The USID model uses a data-centric approach to data analysis and processing meaning that results from all data analysis and processing are written to the same h5 file that contains the recorded measurements. The Hierarchical Data Format (HDF5) allows data, whether it is raw measured data or results of analysis, to be stored in multiple datasets within the same file in a tree-like manner. Certain rules and considerations have been made in pyUSID to ensure consistent and easy access to any data.

The h5py python package provides great functions to create, read, and manage data in HDF5 files. In `pyUSID.hdf_utils`, we have added functions that facilitate scientifically relevant, or pyUSID specific functionality such as easy creation of USID Main datasets, creation of automatically indexed groups to hold results of an analysis, etc. Due to the wide breadth of the functions in `hdf_utils`, the guide for hdf_utils will be split in two parts - one that focuses on functions that facilitate reading and one that facilitate writing of data. The following guide provides examples of how, and more importantly when, to use functions in pyUSID.hdf_utils for various scenarios starting from recording data from instruments to storing analysis data.

Recommended pre-requisite reading

- USID data model
- Crash course on HDF5 and h5py
- Utilities for reading h5USID files using pyUSID

Import all necessary packages

Before we begin demonstrating the numerous functions in pyUSID.hdf_utils, we need to import the necessary packages. Here are a list of packages besides pyUSID that will be used in this example:

- h5py - to open and close the file
- numpy - for numerical operations on arrays in memory
- matplotlib - basic visualization of data

```
from __future__ import print_function, division, unicode_literals
import subprocess
import sys
def install(package):
    subprocess.call([sys.executable, "-m", "pip", "install", package])

import os
# Warning package in case something goes wrong
from warnings import warn
import h5py
import numpy as np
import matplotlib.pyplot as plt

# Finally import pyUSID.
try:
    import pyUSID as usid
except ImportError:
    warn('pyUSID not found.  Will install with pip.')
    import pip
    install('pyUSID')
    import pyUSID as usid
```

Create a HDF5 file

We will be using the h5py functionality to do basic operations on HDF5 files

```
file_path = 'test.h5'
h5_file = h5py.File(file_path)
```

HDF_Utils works with (and uses) h5py

pyUSID and hdf_utils do not preclude the creation of groups and datasets using the h5py package. However, the many functions in hdf_utils are present to make it easier to handle the reading and writing of multidimensional scientific data formatted according to the USID model.

We can always use the h5py functionality to **create a HDF5 group** as shown below:

```
h5_some_group = h5_file.create_group('Some_Group')
print(h5_some_group)
```

Out:

```
<HDF5 group "/Some_Group" (0 members)>
```

In the same way, we can also continue to **create HDF5 datasets** using h5py:

```
h5_some_dataset = h5_some_group.create_dataset('Some_Dataset', np.arange(5))
print(h5_some_dataset)
```

Out:

```
<HDF5 dataset "Some_Dataset": shape (0, 1, 2, 3, 4), type "<f4">
```

Create Groups

create_indexed_group()

In order to accommodate the iterative nature of data recording (multiple sequential and related measurements) and analysis (same analysis performed with different parameters) we add an index as a suffix to HDF5 Group names.

Let us first create a HDF5 group to store some data recorded from an instrument. The below function will automatically create a group with an index as a suffix and write certain book-keeping attributes to the group. We will see how this and similar functions handle situations when similarly named groups already exist.

```
h5_meas_group = usid.hdf_utils.create_indexed_group(h5_file, 'Measurement')
print(h5_meas_group)
```

Out:

```
<HDF5 group "/Measurement_000" (0 members)>
```

Since there were no other groups whose name started with `Measurement`, the function assigned the lowest index - `000` as a suffix to the requested group name. Note that the `-` character is not allowed in the names of the groups since it will be used as the separator character in other functions. This will be made clear when discussing the `create_results_group()` function later.

`create_indexed_group()` calls another handy function called `assign_group_index(` to get the suffix before creating a HDF5 group. Should we want to create another new indexed group called `Measurement`, `assign_group_index()` will notice that a group named `Measurement_000` already exists and will assign the next index (`001`) to the new group - see below. Note that `assign_group_index()` does not create the group; it only assigns a non-conflicting string name for the group.

```
print(usid.hdf_utils.assign_group_index(h5_file, 'Measurement'))
```

Out:

```
Measurement_001
```

Now lets look at datasets and groups in the created file:

```
print('Contents within the file so far:')
usid.hdf_utils.print_tree(h5_file)
```

Out:

```
Contents within the file so far:
/
├ Measurement_000
  ---------------
├ Some_Group
```

(continues on next page)

(continued from previous page)

```
  ----------
  ├ Some_Dataset
```

Clearly, we have the `Measurement_000` Group at the same level as a group named `Some_Group`. The group `Some_Group` contains a dataset named `Some_Dataset` under it.

Both, `Measurement_000` and `Some_Group` have an underline below their name to indicate that they are groups unlike the `Some_Dataset` Dataset

Writing attributes

HDF5 datasets and groups can also store metadata such as experimental parameters. These metadata can be text, numbers, small lists of numbers or text etc. These metadata can be very important for understanding the datasets and guide the analysis routines.

While one could use the basic h5py functionality to write and access attributes, one would encounter a lot of problems when attempting to encode or decode attributes whose values were strings or lists of strings due to some issues in h5py. This problem has been demonstrated in our *primer to HDF5 <./plot_h5py.html>*. Instead of using the basic functionality of `h5py`, we recommend always using the functions in pyUSID that **work reliably and consistently** for any kind of attribute for any version of python:

Here's a look at the (self-explanatory), default attributes that will be written to the indexed group for traceability and posterity. Note that we are using pyUSID's `get_attributes()` function instead of the base h5py capability

```
print('Attributes contained within {}'.format(h5_meas_group))
for key, val in usid.hdf_utils.get_attributes(h5_meas_group).items():
    print('\t%s : %s' % (key, val))
```

Out:

```
Attributes contained within <HDF5 group "/Measurement_000" (0 members)>
        machine_id : challtdow-Aspire-F5-573G
        timestamp : 2018_07_31-13_16_15
        pyUSID_version : 0.0.4
        platform : Linux-4.15.0-29-generic-x86_64-with-debian-stretch-sid
```

Note that these book-keeping attributes written by `create_indexed_group()` are not written when using h5py's `create_group()` function to create a regular group.

```
print('Attributes contained in the basic group created using h5py: {}'.format(h5_some_
↪group))
print(usid.hdf_utils.get_attributes(h5_some_group))
```

Out:

```
Attributes contained in the basic group created using h5py: <HDF5 group "/Some_Group"␣
↪(1 members)>
{}
```

write_book_keeping_attrs()

However, you can always manually add these basic attributes after creating the group using the `write_book_keeping_attrs()`. Note that we can add these basic attributes to Datasets as well as Groups using this function.

```
usid.hdf_utils.write_book_keeping_attrs(h5_some_group)
print('Attributes contained in the basic group after calling write_book_keeping_
↪attrs():')
for key, val in usid.hdf_utils.get_attributes(h5_some_group).items():
    print('\t%s : %s' % (key, val))
```

Out:

```
Attributes contained in the basic group after calling write_book_keeping_attrs():
        machine_id : challtdow-Aspire-F5-573G
        timestamp : 2018_07_31-13_16_15
        pyUSID_version : 0.0.4
        platform : Linux-4.15.0-29-generic-x86_64-with-debian-stretch-sid
```

write_simple_attrs()

Due to the problems in h5py, we use the `write_simple_attrs()` function to add / modify additional attributes to the group:

```
usid.hdf_utils.write_simple_attrs(h5_meas_group, {'Instrument': 'Atomic Force␣
↪Microscope',
                                               'User': 'Joe Smith',
                                               'Room Temperature [C]': 23})
```

copy_attributes()

`hdf_utils.copy_attributes()` is another handy function that simplifies the process of copying attributes from one HDF5 object to another like a Dataset or Group or the file itself. To illustrate, let us copy the attributes from `h5_meas_group` to `h5_some_dataset`:

```
print('Attributes in {} before copying attributes:'.format(h5_some_dataset))
for key, val in usid.hdf_utils.get_attributes(h5_some_dataset).items():
    print('\t%s : %s' % (key, val))
print('\n------------- COPYING ATTRIBUTES ----------------------------\n')
usid.hdf_utils.copy_attributes(h5_meas_group, h5_some_dataset)
print('Attributes in {}:'.format(h5_some_dataset))
for key, val in usid.hdf_utils.get_attributes(h5_some_dataset).items():
    print('\t%s : %s' % (key, val))
```

Out:

```
Attributes in <HDF5 dataset "Some_Dataset": shape (0, 1, 2, 3, 4), type "<f4"> before␣
↪copying attributes:

------------- COPYING ATTRIBUTES ----------------------------

Attributes in <HDF5 dataset "Some_Dataset": shape (0, 1, 2, 3, 4), type "<f4">:
        machine_id : challtdow-Aspire-F5-573G
        timestamp : 2018_07_31-13_16_15
        pyUSID_version : 0.0.4
        platform : Linux-4.15.0-29-generic-x86_64-with-debian-stretch-sid
        Instrument : Atomic Force Microscope
        User : Joe Smith
        Room Temperature [C] : 23
```

Writing Main datasets

Set up a toy problem

Let's set up a toy four-dimensional dataset that has:

- **two position dimensions:**
  - columns - X
  - rows - Y
- **and two spectroscopic dimensions:**
  - (sinusoidal) probing bias waveform
  - cycles over which this bias waveform is repeated

For simplicity, we will keep the size of each dimension small.

```
num_rows = 3
num_cols = 5
num_cycles = 2
bias_pts = 7
```

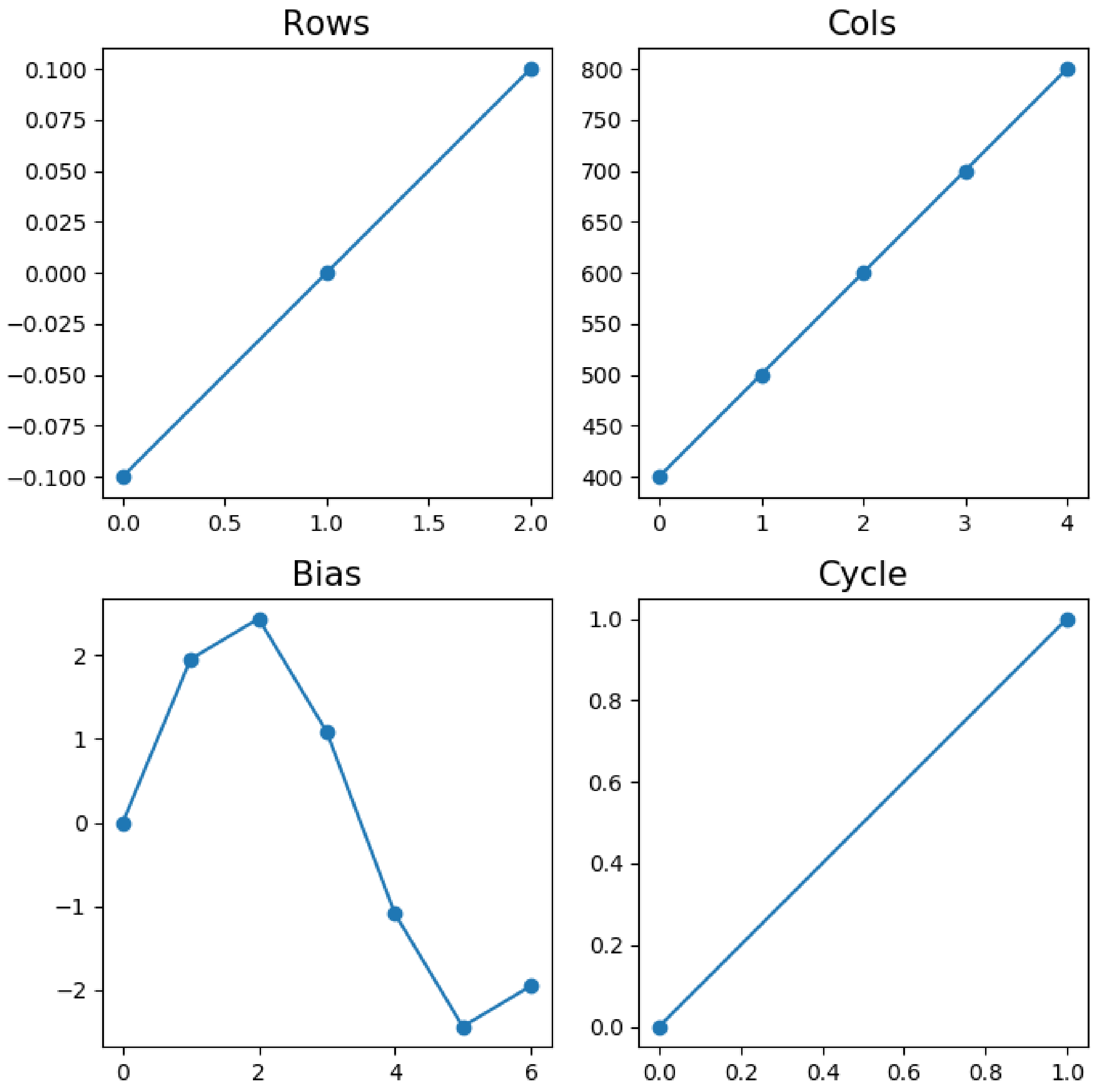
Specify position and spectroscopic dimensions

Next, let us determine how each of the position and spectroscopic dimensions are varied

```
rows_vals = np.arange(-0.1, 0.15, 0.1)
cols_vals = np.arange(400, 900, 100)
bias_vals = 2.5 * np.sin(np.linspace(0, 2*np.pi, bias_pts, endpoint=False))
cycle_vals = np.arange(num_cycles)
```

For better understanding of this dataset, let us take a look at the different values these dimensions can take

```
fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(7, 7))
for axis, vals, dim_name in zip(axes.flat, [rows_vals, cols_vals, bias_vals, cycle_
↪vals],
                                ['Rows', 'Cols', 'Bias', 'Cycle']):
    axis.set_title(dim_name, fontsize=15)
    axis.plot(vals, 'o-')
fig.tight_layout()
```



In the USID model, position and spectroscopic dimensions are defined using some basic information that will be incorporated in **Dimension** objects that contain three vial pieces of information:

- Name of the dimension
- units for the dimension
- **values:**
    - These can be the actual values over which the dimension was varied
    - or number of steps in case of linearly varying dimensions such as `Cycle` below

Note that the Dimension objects in the lists for Positions and Spectroscopic must be arranged from fastest varying to slowest varying to mimic how the data is actually arranged. For example, in this example, there are multiple bias points per cycle and multiple columns per row of data. Thus, the `Bias` changes faster than the `Cycle` and the columns change faster than the rows. Therefore, the `Cols` must come before the `Rows` and `Bias` must precede the `Cycle` dimension:

```
pos_dims = [usid.write_utils.Dimension('Cols', 'nm', cols_vals),
            usid.write_utils.Dimension('Rows', 'um', rows_vals)]
spec_dims = [usid.write_utils.Dimension('Bias', 'V', bias_vals),
             usid.write_utils.Dimension('Cycle', '', num_cycles)]
```

write_main_dataset()

Often, data is is recorded (from instruments) or generated (as a result of some analysis) in chunks (for example - one position at a time). Therefore, it makes sense to first create an empty dataset and then fill in the data as it is generated / recorded.

We will only create an empty dataset first by specifying how large the dataset should be and of what data type (specified using the `dtype` keyword argument). Later, we will go over examples where the whole data is available when creating the HDF5 dataset. The `write_main_dataset()` is **one of the most important and popularly used functions** in `hdf_utils` since it handles:

- thorough validation of all inputs
- the creation of the central dataset
- the creation of the ancillary datasets (if necessary)
- linking the ancillary datasets such that the central dataset becomes a `Main` dataset
- writing attributes

By default h5py does not appear to compress datasets and datasets (especially `Main` datasets) can balloon in size if they are not compressed. Therefore, it is recommended that the compression keyword argument is passed as well. `gzip` is the compression algorithm that is always available with h5py and it does a great job, so we will use this.

We could use the `write_simple_attrs()` function to write attributes to `Raw_Data` at a later stage but we can always pass these attributes to be written at the time of dataset creation if they are already known

```
h5_raw = usid.hdf_utils.write_main_dataset(h5_meas_group,  # parent HDF5 group
                                           (num_rows * num_cols, bias_pts * num_
↪cycles),  # shape of Main dataset
                                           'Raw_Data',  # Name of main dataset
                                           'Current',  # Physical quantity contained␣
↪in Main dataset
                                           'nA',  # Units for the physical quantity
                                           pos_dims,  # Position dimensions
                                           spec_dims,  # Spectroscopic dimensions
                                           dtype=np.float32,  # data type / precision
                                           compression='gzip',
                                           main_dset_attrs={'IO_rate': 4E+6,
↪'Amplifier_Gain': 9})
print(h5_raw)
```

Out:

```
<HDF5 dataset "Raw_Data": shape (15, 14), type "<f4">
located at:
        /Measurement_000/Raw_Data
Data contains:
        Current (nA)
Data dimensions and original shape:
```

(continues on next page)

(continued from previous page)

```
Position Dimensions:
        Cols - size: 5
        Rows - size: 3
Spectroscopic Dimensions:
        Bias - size: 7
        Cycle - size: 2
Data Type:
        float32
```

Let us take a look at the contents of the file again using the `print_tree()` function. What we see is that five new datasets have been created:

- `Raw_Data` was created to contain the 4D measurement we are interested in storing.
- `Spectroscopic_Indices` and Spectroscopic_Values‘‘ contain the information about the spectroscopic dimensions
- `Position_Indices` and `Position_Values` contain the position related information

The underline below `Measurement_000` indicates that this is a HDF5 Group

```
usid.hdf_utils.print_tree(h5_file)
```

Out:

```
/
├ Measurement_000
  ---------------
  ├ Position_Indices
  ├ Position_Values
  ├ Raw_Data
  ├ Spectroscopic_Indices
  ├ Spectroscopic_Values
├ Some_Group
  ----------
  ├ Some_Dataset
```

As mentioned in our *document about the USID model </../../../data_format.html>*, the four supporting datasets (`Indices` and `Values` datasets for `Position` and `Spectroscopic`) help provide meaning to each element in `Raw_Data` such as dimensionality, etc.

Only `Raw_Data` is a `USID Main dataset` while all other datasets are just supporting datasets. We can verify whether a dataset is a Main dataset or not using the `check_if_main()` function:

```
for dset in [h5_raw, h5_raw.h5_spec_inds, h5_raw.h5_pos_vals]:
    print('Is {} is a Main dataset?: {}'.format(dset.name, usid.hdf_utils.check_if_
↪main(dset)))
```

Out:

```
Is /Measurement_000/Raw_Data is a Main dataset?: True
Is /Measurement_000/Spectroscopic_Indices is a Main dataset?: False
Is /Measurement_000/Position_Values is a Main dataset?: False
```

Populating the Dataset:

Note that h5_main still does not contain the values we are interested in filling it in with:

```
print(h5_raw[5])
```

Out:

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Let us simulate a situation where we are recording the data a pixel at a time and writing it to the h5_main dataset:

```
source_main_data = np.random.rand(num_rows * num_cols, bias_pts * num_cycles)

for pixel_ind, pixel_data in enumerate(source_main_data):
    h5_raw[pixel_ind] = pixel_data

# Make sure to ``flush`` the file (write anything in the buffer into the file)
h5_file.flush()
```

Note that we were only simulating a (realistic) situation where all the data was not present at once to write into `Raw_Data` dataset. Let us check the contents at a particular position in the dataset now:

```
print(h5_raw[5])
```

Out:

```
[0.25428426 0.05224128 0.5453549  0.60507995 0.27090117 0.35579985
 0.5196121  0.7235666  0.8526963  0.56558836 0.4830766  0.252037
 0.52279985 0.37431818]
```

Exploring attributes in Main datasets:

Some of the main requirements for promoting a regular dataset to a Main dataset are some mandatory attributes attached to the dataset:

- quantity - What the stored data contains - for example: current, temperature, voltage, strain etc.
- units - the units for the quantity, such as Amperes, meters, etc.
- links to each of the four ancillary datasets

Again, we can use the `get_attributes()` function to see if and how these attributes are stored:

```
for key, val in usid.hdf_utils.get_attributes(h5_raw).items():
    print('{} : {}'.format(key, val))
```

Out:

```
quantity : Current
units : nA
IO_rate : 4000000.0
Amplifier_Gain : 9
machine_id : challtdow-Aspire-F5-573G
timestamp : 2018_07_31-13_16_15
pyUSID_version : 0.0.4
platform : Linux-4.15.0-29-generic-x86_64-with-debian-stretch-sid
Position_Indices : <HDF5 object reference>
Position_Values : <HDF5 object reference>
Spectroscopic_Indices : <HDF5 object reference>
Spectroscopic_Values : <HDF5 object reference>
```

While it is straightforward to read simple attributes like `quantity` or `units`, the values for `Position_Values` or `Spectroscopic_Indices` attributes seem cryptic. These are just references or links to other datasets.

```
print(usid.hdf_utils.get_attr(h5_raw, 'Position_Indices'))
```

Out:

```
<HDF5 object reference>
```

Object references as attributes

We can get access to linked datasets using `get_auxiliary_datasets()`:

```
print(usid.hdf_utils.get_auxiliary_datasets(h5_raw, 'Position_Indices'))
```

Out:

```
[<HDF5 dataset "Position_Indices": shape (15, 2), type "<u4">]
```

Given that `h5_raw` is a `Main` dataset, and‘‘ Position_Indices‘‘ is one of the four essential components of a `Main` dataset, the `USIdataset` object makes it far easier to access the `ancillary datasets` without needing to call a function as above. The USIDataset class has been discussed in greater detail in a separate document.

What do we do if we need to store some other supporting information regarding some measurement? If such supporting datasets do not need to be `USID Main datasets`, we could simply use the basic functionality of `h5py` to create the dataset

```
h5_other = h5_meas_group.create_dataset('Other', np.random.rand(5))
```

h5USID files tend to have a fair number of datasets in them and the most important ones are `Main datasets` and users tend to “walk” or “hop” through the file by stepping only on the `Main datasets`. Thus, we often want to link supporting datasets to the relevant `Main datasets`. This way, such supporting datasets can be accessed via an attribute of the `Main dataset` instead of having to manually specify the path of the supporting dataset.

link_h5_objects_as_attrs()

`link_h5_objects_as_attrs()` makes it easy to link a dataset or group to any other dataset or group. In this example we will link the `Other` dataset to the `Raw_Data` dataset:

```
usid.hdf_utils.link_h5_objects_as_attrs(h5_raw, h5_other)

for key, val in usid.hdf_utils.get_attributes(h5_raw).items():
    print('{} : {}'.format(key, val))
```

Out:

```
quantity : Current
units : nA
IO_rate : 4000000.0
Amplifier_Gain : 9
machine_id : challtdow-Aspire-F5-573G
timestamp : 2018_07_31-13_16_15
pyUSID_version : 0.0.4
platform : Linux-4.15.0-29-generic-x86_64-with-debian-stretch-sid
```

(continues on next page)

(continued from previous page)

```
Position_Indices : <HDF5 object reference>
Position_Values : <HDF5 object reference>
Spectroscopic_Indices : <HDF5 object reference>
Spectroscopic_Values : <HDF5 object reference>
Other : <HDF5 object reference>
```

In the same way, we can even link a group to the Other dataset:

```python
usid.hdf_utils.link_h5_objects_as_attrs(h5_other, h5_some_group)

for key, val in usid.hdf_utils.get_attributes(h5_other).items():
    print('{} : {}'.format(key, val))
```

Out:

```
Some_Group : <HDF5 object reference>
```

What we see above is that 'Other' is now an attribute of the 'Raw_Data' dataset.

One common scenario in scientific workflows is the storage of multiple Main Datasets within the same group. The first Main dataset can be stored along with its four ancillary datasets without any problems. However, if the second Main dataset also requires the storage of Position and Spectroscopic datasets, these datasets would need to be named differently to avoid conflicts with existing datasets (associated with the first Main dataset). Moreover , these ancillary datasets would need to be linked to the second Main dataset with the standard Position_.. and Spectroscopic_.. names for the attributes.

link_h5_obj_as_alias()

link_h5_obj_as_alias() is handy in this scenario since it allows a dataset or group to be linked with a name different from its actual name. For example, we can link the Raw_Data dataset to the Other dataset with an alias:

```python
usid.hdf_utils.link_h5_obj_as_alias(h5_other, h5_raw, 'Mysterious_Dataset')

for key, val in usid.hdf_utils.get_attributes(h5_other).items():
    print('{} : {}'.format(key, val))
```

Out:

```
Some_Group : <HDF5 object reference>
Mysterious_Dataset : <HDF5 object reference>
```

The dataset named Other has a new attribute named Mysterious_Dataset. Let us show that this dataset is none other than Raw_Data:

```python
h5_myst_dset = usid.hdf_utils.get_auxiliary_datasets(h5_other, 'Mysterious_Dataset
↪')[0]
print(h5_myst_dset == h5_raw)
```

Out:

```
True
```

Processing on Datasets

Lets assume that we are normalizing the data in some way and we need to write the results back to the file. As far as the data shapes and dimensionality are concerned, let us assume that the data still remains a 4D dataset.

create_results_group()

Let us first start off with creation of a HDF5 Group that will contain the results. If you recall, groups that contain the results of some processing / analysis on a source dataset are named as `Source_Dataset_name-Process_Name_00x` where the index of the group. The `create_results_group()` function makes it very easy to create a group with such nomenclature and indexing:

```
h5_results_group_1 = usid.hdf_utils.create_results_group(h5_raw, 'Normalization')
print(h5_results_group_1)
```

Out:

```
<HDF5 group "/Measurement_000/Raw_Data-Normalization_000" (0 members)>
```

Let us make up some (random) data which is the result of some Normalization on the `Raw_Data`:

```
norm_data = np.random.rand(num_rows * num_cols, bias_pts * num_cycles)
```

Writing the main dataset

In this scenario we will demonstrate how one might write a `Main dataset` when having the complete processed (in this case some normalization) data is available before even creating the dataset.

One more important point to remember here is that the normalized data is of the same shape and dimensionality as `Raw_Data`. Therefore, we need not unnecessarily create ancillary datasets - we can simply refer to the ones that support `Raw_Data`. During the creation of `Raw_Data`, we passed the `pos_dims` and `spec_dims` parameters for the creation of new `Ancillary datasets`. In this case, we will show how we can ask `write_main_dataset()` to reuse existing ancillary datasets:

```
h5_norm = usid.hdf_utils.write_main_dataset(h5_results_group_1,  # parent group
                                            norm_data,  # data to be written
                                            'Normalized_Data',  # Name of the main␣
↪dataset
                                            'Current',  # quantity
                                            'nA',  # units
                                            None,  # position dimensions
                                            None,  # spectroscopic dimensions
                                            h5_pos_inds=h5_raw.h5_pos_inds,
                                            h5_pos_vals=h5_raw.h5_pos_vals,
                                            h5_spec_inds=h5_raw.h5_spec_inds,
                                            h5_spec_vals=h5_raw.h5_spec_vals,
                                            compression='gzip')
print(h5_norm)
```

Out:

```
<HDF5 dataset "Normalized_Data": shape (15, 14), type "<f8">
located at:
        /Measurement_000/Raw_Data-Normalization_000/Normalized_Data
Data contains:
        Current (nA)
Data dimensions and original shape:
Position Dimensions:
        Cols - size: 5
        Rows - size: 3
Spectroscopic Dimensions:
        Bias - size: 7
        Cycle - size: 2
Data Type:
        float64
```

When we look at the contents of hte file again, what we see below is that the newly created group Raw_Data-Normalization_000 only contains the Normalized_Data dataset and none of the supporting ancillary datasets since it is sharing the same ones created for Raw_Data

```
usid.hdf_utils.print_tree(h5_file)
```

Out:

```
/
├ Measurement_000
  ---------------
  ├ Other
  ├ Position_Indices
  ├ Position_Values
  ├ Raw_Data
  ├ Raw_Data-Normalization_000
    --------------------------
    ├ Normalized_Data
  ├ Spectroscopic_Indices
  ├ Spectroscopic_Values
├ Some_Group
  ----------
  ├ Some_Dataset
```

Shared ancillary datasets

Let us verify that Raw_Data and Normalized_Data share the same ancillary datasets:

```
for anc_name in ['Position_Indices', 'Position_Values', 'Spectroscopic_Indices',
↪'Spectroscopic_Values']:
    # get the handle to the ancillary dataset linked to 'Raw_Data'
    raw_anc = usid.hdf_utils.get_auxiliary_datasets(h5_raw, anc_name)[0]
    # get the handle to the ancillary dataset linked to 'Normalized_Data'
    norm_anc = usid.hdf_utils.get_auxiliary_datasets(h5_norm, anc_name)[0]
    # Show that these are indeed the same dataset
    print('Sharing {}: {}'.format(anc_name, raw_anc == norm_anc))
```

Out:

```
Sharing Position_Indices: True
Sharing Position_Values: True
Sharing Spectroscopic_Indices: True
Sharing Spectroscopic_Values: True
```

Unlike last time with Raw_Data, we wrote the data to the file when creating Normalized_Data, so let us check to make sure that we did in fact write data to disk:

```
print(h5_norm[5])
```

Out:

```
[0.19123188 0.06292482 0.90990684 0.24585696 0.78756491 0.79926715
 0.87184476 0.36021846 0.66203173 0.0718334  0.81087041 0.33533243
 0.68340938 0.60059189]
```

Duplicating Datasets

create_empty_dataset()

Let us say that we are interested in writing out another dataset that is again of the same shape and dimensionality as Raw_Data or Normalized_Data. There is another way to create an empty dataset identical to an existing dataset, and then fill it in. This approach is an alternative to the approach used for Normalized_Data:

```
h5_offsets = usid.hdf_utils.create_empty_dataset(h5_norm, np.float32, 'Offsets')
print(h5_offsets)
```

Out:

```
<HDF5 dataset "Offsets": shape (15, 14), type "<f4">
located at:
        /Measurement_000/Raw_Data-Normalization_000/Offsets
Data contains:
        Current (nA)
Data dimensions and original shape:
Position Dimensions:
        Cols - size: 5
        Rows - size: 3
Spectroscopic Dimensions:
        Bias - size: 7
        Cycle - size: 2
Data Type:
        float32
```

In this very specific scenario, we duplicated practically all aspects of Normalized_Data, including its links to the ancillary datasets. Thus, this h5_offsets automatically also becomes a Main dataset.

However, it is empty and needs to be populated

```
print(h5_offsets[6])
```

Out:

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Since this is an example, we will populate the dataset using same data prepare for norm_data

```
h5_offsets[()] = norm_data
print(h5_offsets[6])
```

Out:

```
[0.22788014 0.8602242  0.9181152  0.27735272 0.65445894 0.535692
 0.06081086 0.5786002  0.3128312  0.80562574 0.6513641  0.6491842
 0.30136934 0.32722104]
```

Creating Ancillary datasets

Often, certain processing of data involves the removal of one or more dimensions (typically `Spectroscopic`). This necessitates careful generation of `indices` and `values` datasets. In our example, we will remove the spectroscopic dimension - `Bias` and leave the position dimensions as is. While we could simply regenerate the spectroscopic indices from scratch knowing that the only remaining spectroscopic dimension is `Cycle`, this is not feasible when writing robust code where we have minimal control or knowledge about the other dimensions. This is especially true when there are 3 or more spectroscopic dimensions and we do not know relationships between the spectroscopic dimensions or the rates of change in these spectroscopic dimensions. Fortunately, `hdf_utils.write_reduced_spec_dsets()` substantially simplifies this problem as shown below.

First, we still need to create the results HDF5 group to hold the results:

```
h5_analysis_group = usid.hdf_utils.create_results_group(h5_norm, 'Fitting')
```

Let us take a look at the contents of the HDF5 file again. Clearly, we do not have any new datasets underneath `Normalized_Data-Fitting_000`

```
usid.hdf_utils.print_tree(h5_file)
```

Out:

```
/
├ Measurement_000
  ---------------
  ├ Other
  ├ Position_Indices
  ├ Position_Values
  ├ Raw_Data
  ├ Raw_Data-Normalization_000
    --------------------------
    ├ Normalized_Data
    ├ Normalized_Data-Fitting_000
      ---------------------------
    ├ Offsets
  ├ Spectroscopic_Indices
  ├ Spectroscopic_Values
├ Some_Group
  ----------
  ├ Some_Dataset
```

write_reduced_spec_dsets()

Now we make the new spectroscopic indices and values datasets while removing the `Bias` dimension

```
h5_spec_inds, h5_spec_vals = usid.hdf_utils.write_reduced_spec_dsets(h5_analysis_
↪group,
                                                                     h5_norm.h5_spec_
↪inds,
                                                                     h5_norm.h5_spec_
↪vals,
                                                                     'Bias')
print(h5_spec_inds)
```

Out:

```
<HDF5 dataset "Spectroscopic_Indices": shape (1, 2), type "<u4">
```

Let us take a look at the contents only inside h5_analysis_group now. Clearly, we have created two new spectroscopic ancillary datasets.

```
usid.hdf_utils.print_tree(h5_analysis_group)
```

Out:

```
/Measurement_000/Raw_Data-Normalization_000/Normalized_Data-Fitting_000
├ Spectroscopic_Indices
├ Spectroscopic_Values
```

write_ind_val_dsets()

Similar to `write_reduced_spec_dsets()`, `hdf_utils` also has another function called `write_ind_val_dsets()` that is handy when one needs to create the ancillary datasets before `write_main_dataset()` is called. For example, consider a data processing algorithm that may or may not change the position dimensions. You may need to structure your code this way:

```
if position dimensions are unchanged:
  # get links to datasets from the source dataset
  h5_pos_inds, h5_pos_vals = h5_source.h5_pos_inds, h5_source.h5_pos_vals
else:
  # Need to create fresh HDF5 datasets
  h5_pos_inds, h5_pos_vals = write_ind_val_dsets()

# At this point, it does not matter how we got h5_pos_inds, h5_pos_vals. We can␣
↪simply link them when we
# create the main dataset.
h5_new_main = write_main_dataset(...., h5_pos_inds=h5_pos_inds, h5_pos_vals=h5_pos_
↪vals)
```

Even though we already decided that we would not be changing the position dimensions for this particular example, we will demonstrate the usage of `write_ind_val_dsets()` to make `position indices` and `values` HDF5 datasets (that are identical to the ones already linked to `h5_norm`)

```
h5_pos_inds, h5_pos_vals = usid.hdf_utils.write_ind_val_dsets(h5_analysis_group, pos_
↪dims, is_spectral=False)
```

Looking at the contents of `Normalized_Data-Fitting_000` now reveals that we have added the `Position` datasets as well. However, we still do not have the `Main dataset`.

```
usid.hdf_utils.print_tree(h5_analysis_group)
```

Out:

```
/Measurement_000/Raw_Data-Normalization_000/Normalized_Data-Fitting_000
├ Position_Indices
├ Position_Values
├ Spectroscopic_Indices
├ Spectroscopic_Values
```

Finally, we can create and write a Main dataset with some results using the trusty write_main_dataset function. Since we have created both the Spectroscopic and Position HDF5 dataset pairs, we simply ask write_main_dataset() to re-use + link them. This is why the `pos_dims` and `spec_dims` arguments are None (we don't want to create new datasets).

```
reduced_main = np.random.rand(num_rows * num_cols, num_cycles)
h5_cap_1 = usid.hdf_utils.write_main_dataset(h5_analysis_group,  # parent HDF5 group
                                            reduced_main,  # data for Main dataset
                                            'Capacitance',  # Name of Main dataset
                                            'Capacitance',  # Quantity
                                            'pF',  # units
                                            None,  # position dimensions
                                            None,  # spectroscopic dimensions
                                            h5_spec_inds=h5_spec_inds,
                                            h5_spec_vals=h5_spec_vals,
                                            h5_pos_inds=h5_pos_inds,
                                            h5_pos_vals=h5_pos_vals,
                                              compression='gzip')
print(h5_cap_1)
```

Out:

```
<HDF5 dataset "Capacitance": shape (15, 2), type "<f8">
located at:
        /Measurement_000/Raw_Data-Normalization_000/Normalized_Data-Fitting_000/
↪Capacitance
Data contains:
        Capacitance (pF)
Data dimensions and original shape:
Position Dimensions:
        Cols - size: 5
        Rows - size: 3
Spectroscopic Dimensions:
        Cycle - size: 2
Data Type:
        float64
```

Multiple Main Datasets

Let's say that we need to create a new `Main dataset` within the same folder as `Capacitance` called `Mean_Capacitance`. `Mean_Capacitance` would just be a spatial map with average capacitance, so it would not even have the `Cycle` spectroscopic dimension. This means that we can reuse the newly created `Position ancillary datasets` but we would need to create new `Spectroscopic_Indices` and `Spectroscopic_Values` datasets in the same folder to express the 0 dimensions in the spectroscopic axis for this new dataset. However, we already have datasets of this name that we created above using the write_reduced_spec_dsets() function. Recall, that the criterion for a Main dataset is that it should have attributes of name Spectroscopic_Indices and Spectroscopic_Values. **It does not matter what the actual name of the linked datasets are**. Coming back to the current example, we could simply ask write_main_dataset() to name the spectroscopic datasets with a different prefix - Empty_Spec instead of Spectroscopic (which is the default) via the aux_spec_prefix keyword argument (last line). This allows the creation of the new Main Dataset without any name clashes with existing datasets:

```
h5_cap_2 = usid.hdf_utils.write_main_dataset(h5_analysis_group,  # Parent HDF5 group
                                            np.random.rand(num_rows * num_cols, 1),  #␣
↪Main Data
                                            'Mean_Capacitance',  # Name of Main Dataset
                                            'Capacitance',  # Physical quantity
                                            'pF',  # Units
                                            None,  # Position dimensions
                                            usid.write_utils.Dimension('Capacitance',
↪'pF', 1),  # Spectroscopic dimensions
                                            h5_pos_inds=h5_pos_inds,
                                            h5_pos_vals=h5_pos_vals,
                                            aux_spec_prefix='Empty_Spec')
print(h5_cap_2)
```

Out:

```
<HDF5 dataset "Mean_Capacitance": shape (15, 1), type "<f8">
located at:
        /Measurement_000/Raw_Data-Normalization_000/Normalized_Data-Fitting_000/Mean_
↪Capacitance
Data contains:
        Capacitance (pF)
Data dimensions and original shape:
Position Dimensions:
        Cols - size: 5
        Rows - size: 3
Spectroscopic Dimensions:
        Capacitance - size: 1
Data Type:
        float64
```

The compression argument need not be specified for small datasets such as Mean Capacitance. Clearly, Mean_Capacitance and Capacitance are two Main datasets that coexist in the same HDF5 group along with their necessary ancillary datasets.

Now, let us look at the contents of the group: Normalized_Data-Fitting_000 to verify this:

```
usid.hdf_utils.print_tree(h5_analysis_group)
```

Out:

```
/Measurement_000/Raw_Data-Normalization_000/Normalized_Data-Fitting_000
├ Capacitance
├ Empty_Spec_Indices
├ Empty_Spec_Values
├ Mean_Capacitance
├ Position_Indices
├ Position_Values
├ Spectroscopic_Indices
├ Spectroscopic_Values
```

File status

is_editable_h5()

When writing a class or a function that modifies or adds data to an existing HDF5 file, it is a good idea to check to make sure that it is indeed possible to write the new data to the file. `is_editable_h5()` is a handy function for this very purpose:

```
print('Is the file editable?: {}'.format(usid.hdf_utils.is_editable_h5(h5_file)))
```

Out:

```
Is the file editable?: True
```

If we close the file and try again we should expect runtime and Value errors. You can try this by yourself if you like

```
h5_file.close()
# print('Is the file editable?: {}'.format(usid.hdf_utils.is_editable_h5(h5_file)))
```

Let us try again by opening this file in read-only mode. We should see that the file will not be editable:

```
h5_file = h5py.File('test.h5', mode='r')
print('Is the file editable?: {}'.format(usid.hdf_utils.is_editable_h5(h5_file)))
```

Out:

```
Is the file editable?: False
```

Closing and deleting the file

```
h5_file.close()
os.remove(file_path)
```

Total running time of the script: (0 minutes 0.168 seconds)

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

10. Formalizing Data Processing

Suhas Somnath

4/26/2018

In this example, we will learn how to write a simple yet formal pyUSID class for processing data.

Introduction

Most of code written for scientific research is in the form of single-use / one-off scripts due to a few common reasons:

- the author feels that it is the fastest mode to accomplishing a research task
- the author feels that they are unlikely to perform the same operation again
- the author does not anticipate the possibility that others may need to run their code

However, more often than not, nearly all researchers have found that one or more of these assumptions fail and a lot of time is spent on fixing bugs and generalizing / formalizing code such that it can be shared or reused. Moreover, we live in an era of open science where the scientific community and an ever-increasing number of scientific journals are moving towards a paradigm where the data and code need to be made available with journal papers. Therefore, in the interest of saving time, energy, and reputation (you do not want to show ugly code / data. Instead you want to be the paragon of clean intelligible data and code), it makes a lot more sense to formalize (parts of) one's data analysis code.

For many researchers, formalizing data processing or analysis may seem like a daunting task due to the complexity of and the number of sub-operations that need to performed. `pyUSID.Process` greatly simplifies the process of formalizing code by lifting or reducing the burden of implementing important, yet tedious tasks and considerations such as:

- **memory management** - reading chunks of datasets that can be processed with the available memory, something very crucial for very large datasets that cannot entirely fit into the computer's memory
- **considerate CPU usage for parallel computing** - using all but one or two CPU cores for the (parallel) computation , which allows the user to continue using the computer for other activities such as reading mail, etc.
- **pausing and resuming computation** - interrupting and resuming the computation at a more convenient time, something that is especially valuable for lengthy computations.
- **avoiding repeated computation and returning existing results** - pyUSID.Process will return existing results computed using the exact same parameters instead of re-computing and storing duplicate copies of the same results.
- **testing before computation** - checking the processing / analysis on a single unit (typically a single pixel) of data before the entire data is processed. This is particularly useful for lengthy computations.

Using `pyUSID.Process`, the user only needs to address the following basic operations:

1. Reading data from file
2. Computation on a single unit of data
3. Writing results to disk

Components of pyUSID.Process

The most important functions in the Process class are:

- `__init__()` - instantiates a 'Process' object of this class after validating the inputs.
- `_create_results_datasets()` - creates the HDF5 datasets and Group(s) to store the results.
- `_map_function()` - the operation that will per be performed on each element in the dataset.
- `test()` - This simple function lets the user test the `map_function` on a unit of data (a single pixel typically) to see if it returns the desired results. It saves a lot of computational time by allowing the user to spot-check results before computing on the entire dataset
- `_read_data_chunk()` - reads the input data from one or more datasets.
- `_write_results_chunk()` - writes the computed results back to the file
- `_unit_computation()` - Defines how to process a chunk (multiple units) of data. This allows room for pre-processing of input data and post-processing of results if necessary. If neither are required, this function essentially applies the parallel computation on `_map_function()`.
- `compute()` - this does the bulk of the work of (iteratively) reading a chunk of data >> processing in parallel via `_unit_computation()` >> calling `write_results_chunk()` to write data. Most sub-classes, including the one below, do not need to extend / modify this function.

Recommended pre-requisite reading

- Universal Spectroscopic and Imaging Data (USID) model
- Crash course on HDF5 and h5py
- Utilities for reading and writing h5USID files using pyUSID
- Crash course on parallel processing

Example problem

For this example, we will be working with 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 USID model.

This example is based on the parallel computing primer where we searched for the peak of each spectra in a dataset. While that example focused on comparing serial and parallel computing, we will focus on demonstrating the simplicity with which such a data analysis algorithm can be formalized.

This example is a simplification of the pycroscopy.analysis.BESHOFitter class in our sister project - Pycroscopy.

Import necessary packages

```
from __future__ import division, print_function, absolute_import, unicode_literals

# The package for accessing files in directories, etc.:
import os

# Warning package in case something goes wrong
from warnings import warn
import subprocess
import sys

def install(package):
    subprocess.call([sys.executable, "-m", "pip", "install", package])
# Package for downloading online files:
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

# The mathematical computation package:
import numpy as np

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

# Packages for plotting:
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
# the scientific function
import sys
sys.path.append('./supporting_docs/')
from peak_finding import find_all_peaks

# Finally import pyUSID:
try:
    import pyUSID as usid
except ImportError:
    warn('pyUSID not found.  Will install with pip.')
    import pip
    install('pyUSID')
    import pyUSID as usid
```

The goal is to **find the amplitude at the peak in each spectra**. Clearly, the operation of finding the peak in one spectra is independent of the same operation on another spectra. Thus, we could divide the dataset in to N parts and use N CPU cores to compute the results much faster than it would take a single core to compute the results. Such problems are ideally suited for making use of all the advanced functionalities in the Process class.

Defining the class

In order to solve our problem, we would need to implement a `sub-class` of pyUSID.Process or in other words - **extend pyUSID.Process**. As mentioned above, the pyUSID.Process class already generalizes several important components of data processing. We only need to extend this class by implementing the science-specific functionality. The rest of the capabilities will be **inherited** from pyUSID.Process.

Lets think about what operations need be performed for each of the core Process functions listed above.

map_function()

The most important component in our new Process class is the unit computation that needs to be performed on each spectra. `map_function()` needs to take as input a single spectra and return the amplitude at the peak (a single value). The `compute()` and `unit_computation()` will handle the parallelization.

The scipy package has a very handy function called *find_peaks_cwt()* that facilitates the search for one or more peaks in a spectrum. We will be using a simplified function called *find_all_peaks()*. The exact methodology for finding the peaks is not of interest for this particular example. However, this function finds the index of 0 or more peaks in the spectra. We only expect one peak at the center of the spectra. Therefore, we can use the `find_all_peaks()` function to find the peaks and address those situations when too few or too many (> 1) peaks are found in a single spectra. Finally, we need to use the index of the peak to find the amplitude from the spectra.

test()

A useful test function should be able to find the peak amplitude for any single spectra in the dataset. So, given the index of a pixel (provided by the user), we should perform two operations:

- read the spectra corresponding to that index from the HDF5 dataset
- apply the `map_function()` to this spectra and return the result.

The goal here is to load the smallest necessary portion of data from the HDF5 dataset to memory and test it against the `map_function()`

create_results_datasets()

Every Process involves a few tasks for this function:

- the creation of a HDF5 group to hold the datasets containing the results - pyUSID.hdf_utils has a handy function that takes care of this.
- storing any relevant metadata regarding this processing as attributes of the HDF5 group for provenance, traceability , and reproducibility.
    - `last_pixel` is a reserved attribute that serves as a flag indicating the last pixel that was successfully processed and written to the results dataset. This attribute is key for resuming partial computations.
- the creation of HDF5 dataset(s) to hold the results. `map_function()` takes a spectra (1D array) and returns the amplitude (a single value). Thus the input dataset (position, spectra) will be reduced to (position, 1). So, we only need to create a single empty dataset to hold the results.

We just need to ensure that we have a reference to the results dataset so that we can populate it with the results.

write_results_chunk()

The result of `compute()` will be a list of amplitude values. All we need to do is:

- write the results into the HDF5 dataset
- Set the `last_pixel` attribute to the value of the `end_pos` internal variable to indicate that pixels upto `end_pos` were successfully processed. Should the computation be interrupted after this point, we could resume from `end_pos` instead of starting from `0` again.
- update the `start_pos` internal variable to guide compute() to process the next batch of positions / pixels

```
class PeakFinder(usid.Process):

    def test(self, pixel_ind):
        """
        Test the algorithm on a single pixel

        Parameters
        ----------
        pixel_ind : uint
            Index of the pixel in the dataset that the process needs to be tested on.
        """
        # First read the HDF5 dataset to get the spectra for this pixel
        spectra = self.h5_main[pixel_ind]
        # Next, apply the map function to the spectra. done!
        return self._map_function(spectra)

    def _create_results_datasets(self):
        """
        Creates the datasets an Groups necessary to store the results.
        There are only THREE operations happening in this function:
        1. Creation of HDF5 group to hold results
        2. Writing relevant metadata to this HDF5 group
        3. Creation of a HDF5 dataset to hold results

        Please see examples on utilities for writing h5USID files for more information
        """
        self.process_name = 'Peak_Finding'
```

(continues on next page)

(continued from previous page)

```
        # 1. create a HDF5 group to hold the results
        self.h5_results_grp = usid.hdf_utils.create_results_group(self.h5_main, self.
↪process_name)

        # 2. Write relevant metadata to the group
        usid.hdf_utils.write_simple_attrs(self.h5_results_grp,
                                        {'last_pixel': 0, 'algorithm': 'find_all_peaks
↪'})

        # Explicitly stating all the inputs to write_main_dataset
        # The process will reduce the spectra at each position to a single value
        # Therefore, the result is a 2D dataset with the same number of positions as␣
↪self.h5_main
        results_shape = (self.h5_main.shape[0], 1)
        results_dset_name = 'Peak_Response'
        results_quantity = 'Amplitude'
        results_units = 'V'
        pos_dims = None # Reusing those linked to self.h5_main
        spec_dims = usid.write_utils.Dimension('Empty', 'a. u.', 1)

        # 3. Create an empty results dataset that will hold all the results
        self.h5_results = usid.hdf_utils.write_main_dataset(self.h5_results_grp,␣
↪results_shape, results_dset_name,
                                                          results_quantity, results_
↪units, pos_dims, spec_dims,
                                                          dtype=np.float32,
                                                          h5_pos_inds=self.h5_main.h5_
↪pos_inds,
                                                          h5_pos_vals=self.h5_main.h5_
↪pos_vals)
        # Note that this function automatically creates the ancillary datasets and␣
↪links them.

        print('Finshed creating datasets')

    def _write_results_chunk(self):
        """
        Write the computed results back to the H5
        In this case, there isn't any more additional post-processing required
        """
        # write the results to the file
        self.h5_results[self._start_pos: self._end_pos, 0] = np.array(self._results)

        # Flush the results to ensure that they have indeed been written to the file
        self.h5_main.file.flush()

        # update the 'last_pixel' to indicate that the process was succesfully␣
↪completed on this many positions:
        self.h5_results_grp.attrs['last_pixel'] = self._end_pos

        # Now update the start position
        self._start_pos = self._end_pos

    @staticmethod
    def _map_function(spectra, *args, **kwargs):
        """
```

(continues on next page)

(continued from previous page)

```
        This is the function that will be applied to each pixel in the dataset.
        It's job is to demonstrate what needs to be done for each pixel in the␣
↪dataset.
        pyUSID.Process will handle the parallel computation and memory management

        As in typical scientific problems, the results from find_all_peaks() need to␣
↪be
        post-processed

        In this case, the find_all_peaks() function can sometimes return 0 or more␣
↪than one peak
        for spectra that are very noisy

        Knowing that the peak is typically at the center of the spectra,
        we return the central index when no peaks were found
        Or the index closest to the center when multiple peaks are found

        Finally once we have a single index, we need to index the spectra by that␣
↪index
        in order to get the amplitude at that frequency.
        """

        peak_inds = find_all_peaks(spectra, [20, 60], num_steps=30)

        central_ind = len(spectra) // 2
        if len(peak_inds) == 0:
            # too few peaks
            # set peak to center of spectra
            val = central_ind
        elif len(peak_inds) > 1:
            # too many peaks
            # set to peak closest to center of spectra
            dist = np.abs(peak_inds - central_ind)
            val = peak_inds[np.argmin(dist)]
        else:
            # normal situation
            val = peak_inds[0]
        # Finally take the amplitude of the spectra at this index
        return np.abs(spectra[val])
```

Comments

- The class appears to be large mainly because of comments that explain what each line of code is doing.
- Several functions of pyUSID.Process such as `__init__()` and `compute()` were inherited from the pyUSID.Process class.
- In simple cases such as this, we don't even have to implement a function to read the data from the dataset since pyUSID.Process automatically calculates how much of the data iss safe to load into memory. In this case, the dataset is far smaller than the computer memory, so the entire dataset can be loaded and processed at once.
- In this example, we did not need any pre-processing or post-processing of results but those can be implemented too if necessary.
- The majority of the code in this class would have to be written regardless of whether the intention is formalize the data processing or not. In fact, we would argue that **more** code may need to be written than what is shown

below if one were **not** formalizing the data processing (data reading, parallel computing, memory management, etc.)

- This is the simplest possible implementation of Process. Certain features such as checking for existing results and resuming partial computations have not been shown in this example.

Use the class

Now that the class has been written, it can be applied to an actual dataset.

Load the dataset

In order to demonstrate this Process class, we will be using a real experimental dataset that is available on the pyUSID GitHub project. First, lets download this file from Github:

```
h5_path = 'temp.h5'
url = 'https://raw.githubusercontent.com/pycroscopy/pyUSID/master/data/BELine_0004.h5'
if os.path.exists(h5_path):
    os.remove(h5_path)
_ = wget.download(url, h5_path, bar=None)
```

Lets open the file in an editable (r+) mode and look at the contents:

```
h5_file = h5py.File(h5_path, mode='r+')
print('File contents:\n')
usid.hdf_utils.print_tree(h5_file)
```

Out:

```
File contents:

/
├ 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
```

The focus of this example is not on the data storage or formatting but rather on demonstrating our new Process class so lets dive straight into the main dataset that requires analysis of the spectra:

```
h5_chan_grp = h5_file['Measurement_000/Channel_000']

# Accessing the dataset of interest:
h5_main = usid.USIDataset(h5_chan_grp['Raw_Data'])
print('\nThe main dataset:\n------------------------------------')
print(h5_main)

# Extract some metadata:
num_rows, num_cols = h5_main.pos_dim_sizes
freq_vec = h5_main.get_spec_values('Frequency') * 1E-3
```

Out:

```
The main dataset:
------------------------------------
<HDF5 dataset "Raw_Data": shape (16384, 119), type "<c8">
located at:
        /Measurement_000/Channel_000/Raw_Data
Data contains:
        Cantilever Vertical Deflection (V)
Data dimensions and original shape:
Position Dimensions:
        X - size: 128
        Y - size: 128
Spectroscopic Dimensions:
        Frequency - size: 119
Data Type:
        complex64
```

Use the Process class

Instantiation

Note that the instantiation of the new `PeakFinder` Process class only requires that we supply the main dataset on which the computation will be performed:

```
fitter = PeakFinder(h5_main)
```

Out:

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

test()

As advised, lets test the `PeakFinder` on an example pixel:

```
row_ind, col_ind = 103, 19
pixel_ind = col_ind + row_ind * num_cols

```

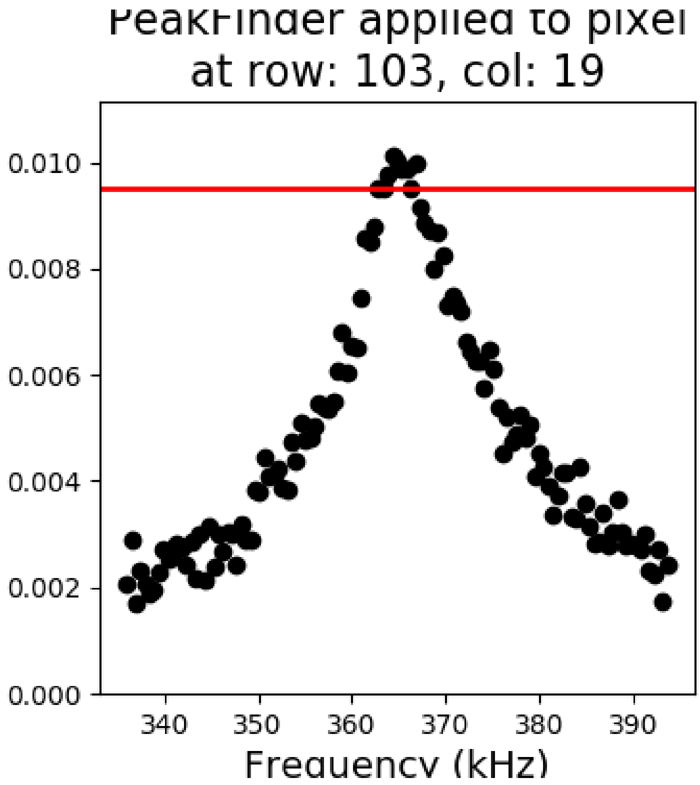
(continues on next page)

(continued from previous page)

```
# Testing is as simple as supplying a pixel index
amplitude = fitter.test(pixel_ind)
```

Now, let's visualize the results of the test:

```
spectra = h5_main[pixel_ind]

fig, axis = plt.subplots(figsize=(4, 4))
axis.scatter(freq_vec, np.abs(spectra), c='black')
axis.axhline(amplitude, color='r', linewidth=2)
axis.set_xlabel('Frequency (kHz)', fontsize=14)
axis.set_ylabel('Amplitude (V)')
axis.set_ylim([0, 1.1 * np.max(np.abs(spectra))])
axis.set_title('PeakFinder applied to pixel\nat row: {}, col: {}'.format(row_ind, col_
↪ind), fontsize=16);
```



If we weren't happy with the results, we could tweak some parameters when initializing the `PeakFinder` object and try again. However, for the sake of simplicity, we don't have any parameters we can / want to adjust in this case. So, lets proceed.

compute()

Now that we know that the `PeakFinder` appears to be performing as expected, we can apply the amplitude finding

```
h5_results_grp = fitter.compute()
print(h5_results_grp)
```

Out:

```
Finshed creating datasets
You maybe able to abort this computation at any time and resume at a later time!
        If you are operating in a python console, press Ctrl+C or Cmd+C to abort
        If you are in a Jupyter notebook, click on "Kernel">>"Interrupt"
Starting computing on 3 cores (requested 3 cores)
Finished parallel computation
Completed computation on chunk. Writing to file.
<HDF5 group "/Measurement_000/Channel_000/Raw_Data-Peak_Finding_000" (3 members)>
```

Lets take a look again at the file contents. We should be seeing a new HDF5 group called `Raw_Data-Peak_Finding_000` and three datasets within the group. Among the datasets is `Peak_Response` that contains the peak amplitudes we are interested in.

```
usid.hdf_utils.print_tree(h5_file)
```

Out:

```
/
├ 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
    ├ Raw_Data-Peak_Finding_000
      -------------------------
      ├ Peak_Response
      ├ Spectroscopic_Indices
      ├ Spectroscopic_Values
    ├ Spatially_Averaged_Plot_Group_000
      ---------------------------------
      ├ Bin_Frequencies
      ├ Mean_Spectrogram
      ├ Spectroscopic_Parameter
      ├ Step_Averaged_Response
    ├ Spectroscopic_Indices
    ├ Spectroscopic_Values
    ├ UDVS
    ├ UDVS_Indices
```

Lets look at this `Peak_Response` dataset:

```
h5_peak_amps = usid.USIDataset(h5_results_grp['Peak_Response'])
print(h5_peak_amps)
```

Out:

```
<HDF5 dataset "Peak_Response": shape (16384, 1), type "<f4">
located at:
```

(continues on next page)

(continued from previous page)

```
        /Measurement_000/Channel_000/Raw_Data-Peak_Finding_000/Peak_Response
Data contains:
        Amplitude (V)
Data dimensions and original shape:
Position Dimensions:
        X - size: 128
        Y - size: 128
Spectroscopic Dimensions:
        Empty - size: 1
Data Type:
        float32
```

Visualize

Since `Peak_Response` is a USIDataset, we could use its ability to provide its own N dimensional form:

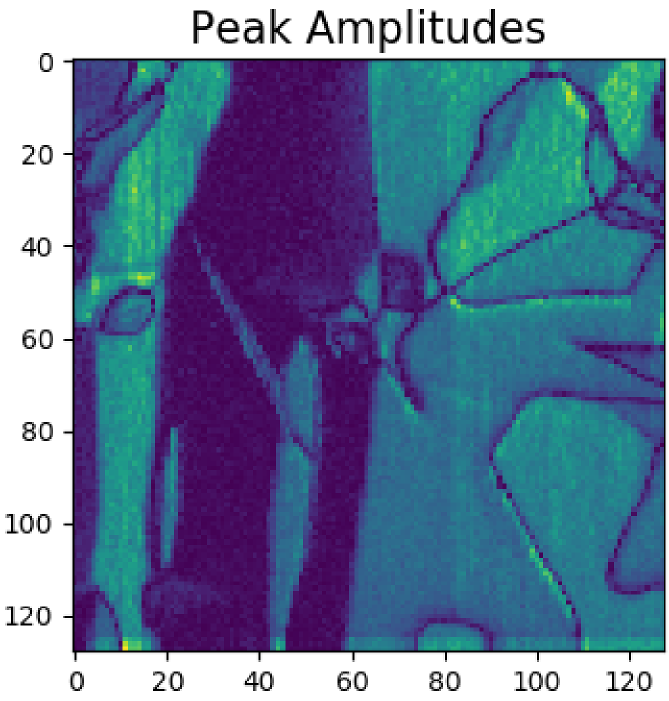
```
amplitudes = np.squeeze(h5_peak_amps.get_n_dim_form())
print('N dimensional shape of Peak_Response: {}'.format(amplitudes.shape))
```

Out:

```
N dimensional shape of Peak_Response: (128, 128)
```

Finally, we can look at a map of the peak amplitude for the entire dataset:

```
fig, axis = plt.subplots(figsize=(4, 4))
axis.imshow(amplitudes)
axis.set_title('Peak Amplitudes', fontsize=16)
```



Clean up

Finally lets close and delete the example HDF5 file

```
h5_file.close()
os.remove(h5_path)
```

Flow of functions

By default, very few functions (`test()`, `compute()`) are exposed to users. This means that one of these functions calls a chain of the other functions in the class.

init()

Instantiating the class via something like: `fitter = PeakFinder(h5_main)` happens in two parts:

1. First the subclass (`PeakFinder`) calls the initialization function in `Process` to let it run some checks:

- Check if the provided `h5_main` is indeed a `Main` dataset
- call `set_memory_and_cores()` to figure out how many pixels can be read into memory at any given time
- Initialize some basic variables

2. Next, the subclass continues any further validation / checks / initialization - this was not implemented for `PeakFinder` but here are some things that can be done:

   - Find HDF5 groups which either have partial or fully computed results already for the same parameters by calling `check_for_duplicates()`

test()

This function only calls the `map_function()` by definition

compute()

Here is how compute() works:

- Check if you can return existing results for the requested computation and return if available by calling either:
  - **`get_existing_datasets()` - reads all necessary parameters and gets references to the HDF5 datasets that shou**
    contain the results
  - `use_partial_computation()` - pick the first partially computed results group that was discovered by `check_for_duplicates()`
- call `create_results_datasets()` to create the HDF5 datasets and group objects
- read the first chunk of data via `read_data_chunk()` into `self._data`
- Until the source dataset is fully read (`self._data is not None`), do:
  - call `unit_computation()` on `self._data`
    - By default `unit_computation()` just maps `map_function()` onto `self._data`
  - call `write_results_chunk()` to write `self._results` into the HDF5 datasets

  - read the next chunk of data into `self._data`

use_partial_computation()

Not used in `PeakFinder` but this function can be called to manually specify an HDF5 group containing partial results

We encourage you to read the source code for more information.

Advanced examples

Please see the following pycroscopy classes to learn more about the advanced functionalities such as resuming computations, checking of existing results, using unit_computation(), etc.:

- `pycroscopy.processing.SignalFilter`
- `pycroscopy.analysis.GIVBayesian`

Tips and tricks

Here we will cover a few common use-cases that will hopefully guide you in structuring your computational problem

Juggling dimensions

We intentionally chose a simple example above to quickly illustrate the main components / philosophy of the Process class. The above example had two position dimensions collapsed into the first axis of the dataset and a single spectroscopic dimension (`Frequency`). What if the spectra were acquired as a function of other variables such as a `DC bias`? In other words, the dataset would now have N spectra per location. In such cases, the dataset would have 2 spectroscopic dimensions: `Frequency` and `DC bias`. We cannot therefore simply map the `map_function()` to the data in every pixel. This is because the `map_function()` expects to work over a single spectra whereas we now have N spectra per pixel. Contrary to what one would assume, we do not need to throw away all the code we wrote above. We only need to add code to juggle / move the dimensions around till the problem looks similar to what we had above.

In other words, the above problem was written for a dataset of shape `(P, S)` where `P` is the number of positions and `S` is the length of a single spectra. Now, we have data of shape `(P, N*S)` where `N` is the number of spectra per position. In order to use most of the code already written above, we need to reshape the data to the shape `(P*N, S)`. Now, we can easily map the existing `map_function()` on this `(P*N, S)` dataset.

As far as implementation is concerned, we would need to add the reshaping step to `_read_data_chunk()` as:

```python
def _read_data_chunk(self):
    super(PeakFinder, self)._read_data_chunk()
    # The above line causes the base Process class to read X pixels from the dataset into self.data
    # All we need to do now is reshape self.data from (X, N*S) to (X*N, S):
    # Assuming that we know N (num_spectra) through some metadata:
    self.data = self.data.reshape(self.data.shape[0]* num_spectra, -1)
```

Recall that `_read_data_chunk()` reads `X` pixels at a time where `X` is the largest number of pixels whose raw data, intermediate products, and results can simultaneously be held in memory. The dataset used for the example above is tractable enough that the entire data is loaded at once, meaning that `X = P` in this case.

From here, on, the computation would continue as is but as expected, the results would also consequently be of shape (P*N). We would have to reverse the reshape operation to get back the results in the form: (P, N). So we would prepend the reverse reshape operation to _write_results_chunk():

```
def _write_results_chunk(self):
    # Recall that the results from the computation are stored in a list called self._
↪results
    self._results = np.array(self._results)  # convert from list to numpy array
    self._results = self._results.reshape(-1, num_spectra)
    # Now self._results is of shape (P, N) and we can store it in the HDF5 dataset as␣
↪we did above.
```

Computing on chunks instead of mapping

In certain cases, the computation is a little more complex that the map_function() cannot directly be mapped to the data. Alternatively, in some cases the map_function() needs to mapped multiple times or different sections of the self.data. For such cases, the _unit_computation() in Process provides far more flexibility to the developer. Please see the pycroscopy.processing.SignalFilter and pycroscopy.analysis.GIVBayesian for examples.

By default, _unit_computation() maps the map_function() to self.data using parallel_compute() and stores the results in self._results. Recall that self.data contains data for X pixels. For example, _unit_computation() in pycroscopy.analysis.GIVBayesian breaks up the spectra (second axis) of self.data into two halves and computes the results separately for each half. _unit_computation() for this class calls parallel_compute() twice - to map the map_function() to each half of the data chunk. This is a functionality that is challenging to efficiently attain without _unit_computation(). Note that when the _unit_computation() is overridden, the developer is responsible for the correct usage of parallel_compute(), especially passing arguments and keyword arguments.

Other Notes

We are exploring Dask as a framework for embarrassingly parallel computation.

Total running time of the script: (0 minutes 37.397 seconds)

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

11. Input / Output / Computing utilities

Suhas Somnath

8/12/2017

This is a short walk-through of useful utilities in pyUSID.io_utils that simplify common i/o and computational tasks.

```
from __future__ import print_function, division, unicode_literals
from multiprocessing import cpu_count
import subprocess
import sys

def install(package):
```

(continues on next page)

(continued from previous page)

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

Computation related utilities

recommend_cpu_cores()

Time is of the essence and every developer wants to make the best use of all available cores in a CPU for massively parallel computations. `recommend_cpu_cores()` is a popular function that looks at the number of parallel operations, available CPU cores, duration of each computation to recommend the number of cores that should be used for any computation. If the developer / user requests the use of N CPU cores, this function will validate this number against the number of available cores and the nature (lengthy / quick) of each computation. Unless, a suggested number of cores is specified, `recommend_cpu_cores()` will always recommend the usage of N-2 CPU cores, where N is the total number of logical cores (Intel uses hyper-threading) on the CPU to avoid using up all computational resources and preventing the computation from making the computer otherwise unusable until the computation is complete Here, we demonstrate this function being used in a few use cases:

```
print('This CPU has {} cores available'.format(cpu_count()))
```

Out:

```
This CPU has 4 cores available
```

Case 1: several independent computations or jobs, each taking far less than 1 second. The number of desired cores is not specified. The function will return 2 lesser than the total number of cores on the CPU

```
num_jobs = 14035
recommeded_cores = usid.io_utils.recommend_cpu_cores(num_jobs, lengthy_
↪computation=False)
print('Recommended number of CPU cores for {} independent, FAST, and parallel '
      'computations is {}\n'.format(num_jobs, recommeded_cores))
```

Out:

```
Recommended number of CPU cores for 14035 independent, FAST, and parallel␣
↪computations is 3
```

Case 2: Several independent and fast computations, and the function is asked if 3 cores is OK. In this case, the function will allow the usage of the 3 cores so long as the CPU actually has 3 or more cores

```
requested_cores = 3
recommeded_cores = usid.io_utils.recommend_cpu_cores(num_jobs, requested_
↪cores=requested_cores, lengthy_computation=False)
print('Recommended number of CPU cores for {} independent, FAST, and parallel '
      'computations using the requested {} CPU cores is {}\n'.format(num_jobs,␣
↪requested_cores, recommeded_cores))
```

Out:

```
Recommended number of CPU cores for 14035 independent, FAST, and parallel␣
↪computations using the requested 3 CPU cores is 3
```

Case 3: Far fewer independent and fast computations, and the function is asked if 3 cores is OK. In this case, configuring multiple cores for parallel computations will probably be slower than serial computation with a single core. Hence, the function will recommend the use of only one core in this case.

```
num_jobs = 13
recommeded_cores = usid.io_utils.recommend_cpu_cores(num_jobs, requested_
↪cores=requested_cores, lengthy_computation=False)
print('Recommended number of CPU cores for {} independent, FAST, and parallel '
      'computations using the requested {} CPU cores is {}\n'.format(num_jobs,␣
↪requested_cores, recommeded_cores))
```

Out:

```
Recommended number of CPU cores for 13 independent, FAST, and parallel computations␣
↪using the requested 3 CPU cores is 1
```

Case 4: The same number of a few independent computations but eahc of these computations are expected to be lengthy. In this case, the overhead of configuring the CPU core for parallel computing is worth the benefit of parallel computation. Hence, the function will allow the use of the 3 cores even though the number of computations is small.

```
recommeded_cores = usid.io_utils.recommend_cpu_cores(num_jobs, requested_
↪cores=requested_cores, lengthy_computation=True)
print('Recommended number of CPU cores for {} independent, SLOW, and parallel '
      'computations using the requested {} CPU cores is {}'.format(num_jobs,␣
↪requested_cores, recommeded_cores))
```

Out:

```
Recommended number of CPU cores for 13 independent, SLOW, and parallel computations␣
↪using the requested 3 CPU cores is 3
```

get_available_memory()

Among the many best-practices we follow when developing a new data analysis or processing class is memory-safe computation. This handy function helps us quickly get the available memory. Note that this function returns the available memory in bytes. So, we have converted it to gigabytes here:

```
print('Available memory in this machine: {} GB'.format(usid.io_utils.get_available_
↪memory()/1024**3))
```

Out:

```
Available memory in this machine: 3.2607574462890625 GB
```

String formatting utilities

Frequently, there is a need to print out logs on the console to inform the user about the size of files, or estimated time remaining for a computation to complete, etc. pyUSID.io_utils has a few handy functions that help in formatting quantities in a human readable format.

format_size()

One function that uses this functionality to print the size of files etc. is format_size(). While one can manually print the available memory in gibibytes (see above), `format_size()` simplifies this substantially:

```
print('Available memory in this machine: {}'.format(usid.io_utils.format_size(usid.io_
↪utils.get_available_memory())))
```

Out:

```
Available memory in this machine: 3.26 GB
```

format_time()

On the same lines, `format_time()` is another handy function that is great at formatting time and is often used in Process and Fitter to print the remaining time

```
print('{} seconds = {}'.format(14497.34, usid.io_utils.format_time(14497.34)))
```

Out:

```
14497.34 seconds = 4.03 hours
```

format_quantity()

You can generate your own formatting function based using the generic function: `format_quantity()`. For example, if `format_time()` were not available, we could get the same functionality via:

```
units = ['msec', 'sec', 'mins', 'hours']
factors = [0.001, 1, 60, 3600]
time_value = 14497.34
print('{} seconds = {}'.format(14497.34, usid.io_utils.format_quantity(time_value,␣
↪units, factors)))
```

Out:

```
14497.34 seconds = 4.03 hours
```

formatted_str_to_number()

pyUSID also has a handy function for the inverse problem of getting a numeric value from a formatted string:

```
unit_names = ["MHz", "kHz"]
unit_magnitudes = [1E+6, 1E+3]
str_value = "4.32 MHz"
num_value = usid.io_utils.formatted_str_to_number(str_value, unit_names, unit_
↪magnitudes, separator=' ')
print('formatted_str_to_number says: {} = {}'.format(str_value, num_value))
```

Out:

```
formatted_str_to_number says: 4.32 MHz = 4320000.0
```

get_time_stamp()

We try to use a standardized format for storing time stamps in HDF5 files. The function below generates the time as a string that can be easily parsed if need be

```
print('Current time is: {}'.format(usid.io_utils.get_time_stamp()))
```

Out:

```
Current time is: 2018_07_31-13_16_52
```

Communication utilities

file_dialog()

This handy function generates a file window to select files. We encourage you to try this function out since it cannot demonstrated within this static document.

check_ssh()

When developing workflows that need to work on remote or virtual machines in addition to one's own personal computer such as a laptop, this function is handy at letting the developer know where the code is being executed

```
print('Running on remote machine: {}'.format(usid.io_utils.check_ssh()))
```

Out:

```
Running on remote machine: False
```

Total running time of the script: (0 minutes 0.002 seconds)

1.7 Data Model and File Format

Suhas Somnath

8/8/2017

In this document we aim to provide a comprehensive overview of the motivation for and specifications of the **Universal Spectroscopy and Imaging Data** (**USID**) Model and the file format (**h5USID**) used for storing spectroscopy and imaging data.

Pycroscopy uses the **USID** model and **h5USID** files.

Dr. Stephen Jesse conceived the **USID** while **Dr. Suhas Somnath** and **Chris R. Smith** implemented **USID** into hierarchical data format (**HDF5**) files using python in **pyUSID**

Contents

1.7.1 Nomenclature

Before we start off, lets clarify some nomenclature to avoid confusion.

Data model

Data model refers to the way the data is arranged. This does **not** depend on the implementation in a particular file format

File format

This corresponds to the kind of file, such as a spreadsheet (.CSV), an image (.PNG), a text file (.TXT) within which information is contained.

Data format

data format is actually a rather broad term. However, we have observed that people often refer to the combination of a data model implemented within a file format as a `data format`.

Measurements

In all measurements, some `quantity` such as voltage, resistance, current, amplitude, or intensity is collected as a function of (typically all combinations of) one or more `independent variables`. For example, a gray-scale image represents the quantity - intensity being recorded for all combinations of the variables - row and column. A (simple) spectrum represents a quantity such as amplitude or phase recorded as a function of a reference variable such as wavelength or frequency.

Data collected from measurements result in N-dimensional datasets where each `dimension` corresponds to a variable that was varied. Going back to the above examples a gray-scale image would be represented by a 2 dimensional dataset whose dimensions are row and column. Similarly, a simple spectrum wold be a 1 dimensional dataset whose sole dimension would be frequency for example.

Dimensionality

- We consider data recorded for all combinations of 2 or more variables as `multi-dimensional` datasets or `Nth order tensors`:
    - For example, if a single value of current is recorded as a function of driving / excitation bias or voltage having B values, the dataset is said to be `1 dimensional` and the dimension would be - `Bias`.

  - If the bias is cycled C times, the data is said to be `two dimensional` with dimensions - `(Bias, Cycle).`
  - If the bias is varied over B values over C cycles at X columns and Y rows in a 2D grid of positions, the resultant dataset would have `4 dimensions: (Rows, Columns, Cycle, Bias).`
- `Multi-feature`: As a different example, let us suppose that the `petal width`, `length`, and `weight` were measured for `F` different kinds of flowers. This would result in a `1 dimensional dataset` with the kind of flower being the sole dimension. Such a dataset is **not** a 3 dimensional dataset because the `petal width, length`, and `weight` are only different `features` for each measurement. Some quantity needs to be **measured for all combinations of** petal width, length, and weight to make this dataset 3 dimensional. Most examples observed in data mining, simple machine learning actually fall into this category

1.7.2 Why should you care?

The quest for understanding more about matter has necessitated the development of a multitude of instruments, each capable of numerous measurement modalities.

Proprietary file formats

Typically, each commercial instruments generates data files formatted in proprietary file formats by the instrument manufacturer. The proprietary nature of these file formats and the obfuscated data model within the files impede scientific progress in the following ways:

1. By making it challenging for researchers to extract data from these files
2. Impeding the correlation of data acquired from different instruments.
3. Inability to store results back into the same file
4. Inflexibility to accommodate few kilobytes to several gigabytes of data
5. Requiring different versions of analysis routines for each data format
6. In some cases, requiring proprietary software provided with the instrument to access the data

Future concerns

1. Several fields are moving towards the open science paradigm which will require journals and researchers to support journal papers with data and analysis software
2. US Federal agencies that support scientific research require curation of datasets in a clear and organized manner

Other problems

1. The vast majority of scientific software packages (e.g. X-array) aim to focus at information already available in memory. In other words they do not solve the problem of storing data in a self-describing manner and reading + processing this data.
2. There are a few file formatting packages and approaches (Nexus, NetCDF). However, they are typically narrow in scope and only solve the data formatting for specific communities
3. Commercial image analysis software are often woefully limited in their capabilities and only work on simple 1, 2, and in some cases- 3D datasets. There are barely any software for handling arbitrarily large multi-dimensional datasets.

4. In many cases, especially electron and ion based microscopy, the very act of probing the sample damages the sample. To minimize damage to the sample, researchers only sample data from a few random positions in the 2D grid and use advanced algorithms to reconstruct the missing data. We have not come across any robust solutions for storing such **Compressed sensing / sparse sampling** data. More in the **Advanced Topics** section.

1.7.3 Universal Spectroscopy and Imaging Data (USID) Model

To solve the above and many more problems, we have developed an **instrument agnostic data model** that can be used to represent data from any instrument, size, dimensionality, or complexity.

Information in **USID** are stored in three main kinds of datasets:

1. `Main` datasets that contain the raw measurements recorded from the instrument as well as results from processing or analysis routines applied to the data
2. Mandatory `Ancillary` datasets that are necessary to explain the `main` data
3. `Extra` datasets store any other data that may be of value

In addition to datasets, the data model is highly reliant on metadata that capture smaller pieces but critical pieces of information such as the `quantity` and `units` that describe every data point in the `main` dataset.

We acknowledge that this data model is not trivial to understand at first glance but we are making every effort to make is simple to understand. If you ever find anything complicated or unclear, please write to us **and we will improve our documentation.**

`Main` Datasets

Regardless of origin, modality or complexity, imaging data (and most scientific data for that matter) have one thing in common:

The same measurement / operation is performed at each spatial position

The **USID** model is based on this one simple ground-truth. The data always has some `spatial dimensions` (X, Y, Z) and some `spectroscopic dimensions` (time, frequency, intensity, wavelength, temperature, cycle, voltage, etc.). **In USID, the spatial dimensions are collapsed onto a single dimension and the spectroscopic dimensions are flattened into the second dimension.** Thus, all data are stored as **two dimensional arrays**. The data would be arranged in the same manner that reflects the sequence in which the individual data points were collected. Examples below will simplify this data-representation paradigm significantly.

In general, if a measurement of length `P` was recorded for each of `N` positions, it would be structured as shown in the table below here the prefixes `i` correspond to the positions and `j` for spectroscopic:

i0, j0	i0, j1	i0, j2	<..>	i0, jP-2	i0, jP-1
i1, j0	i1, j1	i1, j2	<..>	i1, jP-2	i1, jP-1
<. . . . . .>	<. . . . . .>	<. . . . . .>	<..>	<. . . . .>	<. . . . .>
iN-2, j0	iN-2, j1	iN-2, j2	<..>	iN-2, jP-2	iN-2, jP-1
iN-1, j0	iN-1, j1	iN-1, j2	<..>	iN-1, jP-1	iN-1, jP-1

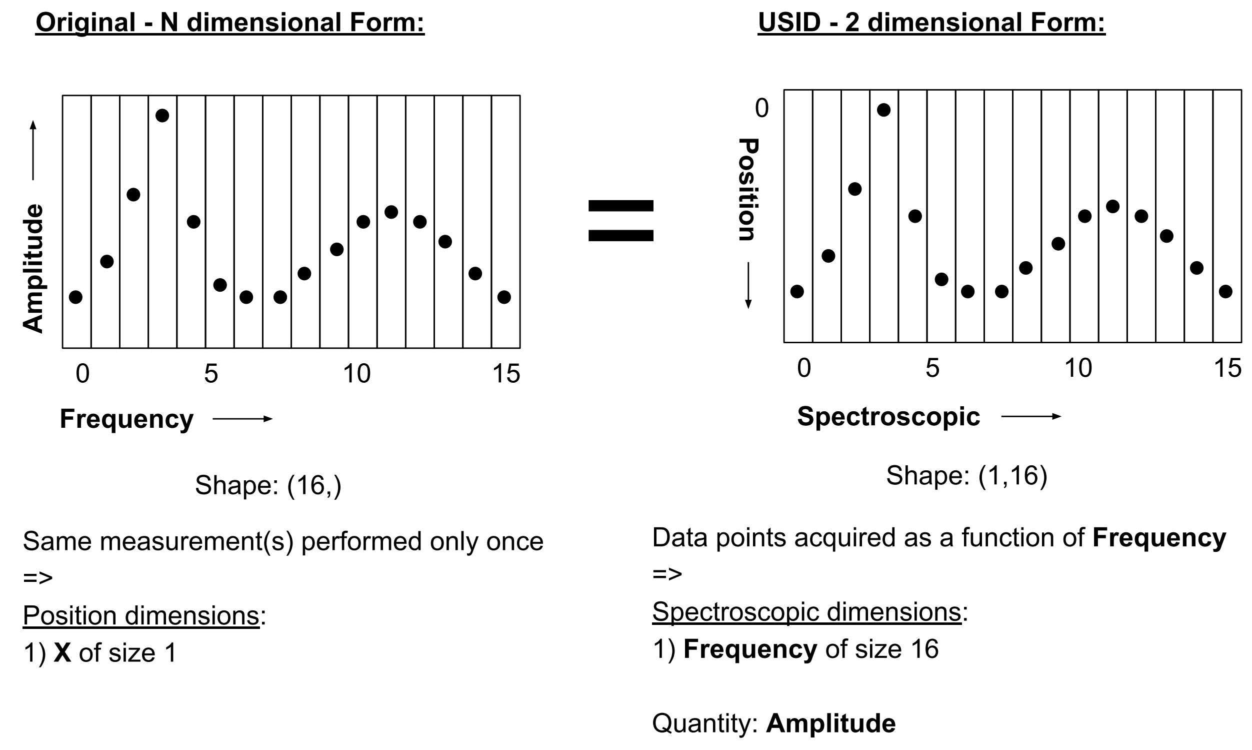
A notion of chronology is attached to both the position and spectroscopic axes. In other words, the data for the second location (second row in the above table) was acquired before the first location (first row). The same applies to the spectroscopic axis as well. This is an important point to remember especially when information is recorded from multiple sources or channels (e.g. - data from different sensors) or if two or more numbers are **necessary** to give a particular observation / data point its correct meaning (e.g. - color images). This point will be clarified via examples that follow.

While the data could indeed be stored in the original N-dimensional form, there are a few key **advantages to the 2D structuring**:

- The data is already of the **same structure expected by machine learning algorithms** and requires minimal to no pre-processing or post-processing. Briefly, the data is simply arranged in the standard form of `instances x features`, where `instances` makes up the locations and `features` which contains all the observables per entry.
- In certain cases, the data simply **cannot be represented in an N-dimensional form** since one of the dimensions has multiple sizes in different contexts.
- Researchers want to acquire ever larger datasets that take much longer to acquire. This has necessitated approaches such as **sparse sampling** or compressed sensing wherein measurements are acquired from a few randomly sampled positions and the data for the rest of the positions are inferred using complex algorithms. Storing such sparse sampled data in the N dimensional form would balloon the size of the stored data even though the majority of the data is actually empty. Two dimensional datasets would allow the random measurements to be written without any empty sections.
- When acquiring measurement data, users often adjust experimental parameters during the experiment that may affect the size of the data, especially the spectral sizes. Thus, **changes in experimental parameters** would mean that the existing N dimensional set would have to be left partially (in most cases largely) empty and a new N dimensional dataset would have to be allocated with the first few positions left empty. In the case of flattened datasets, the current dataset can be truncated at the point of the parameter change and a new dataset can be created to start from the current measurement. Thus, no space would be wasted.

Here are some examples of how some familiar data can be represented using this paradigm:

Spectrum



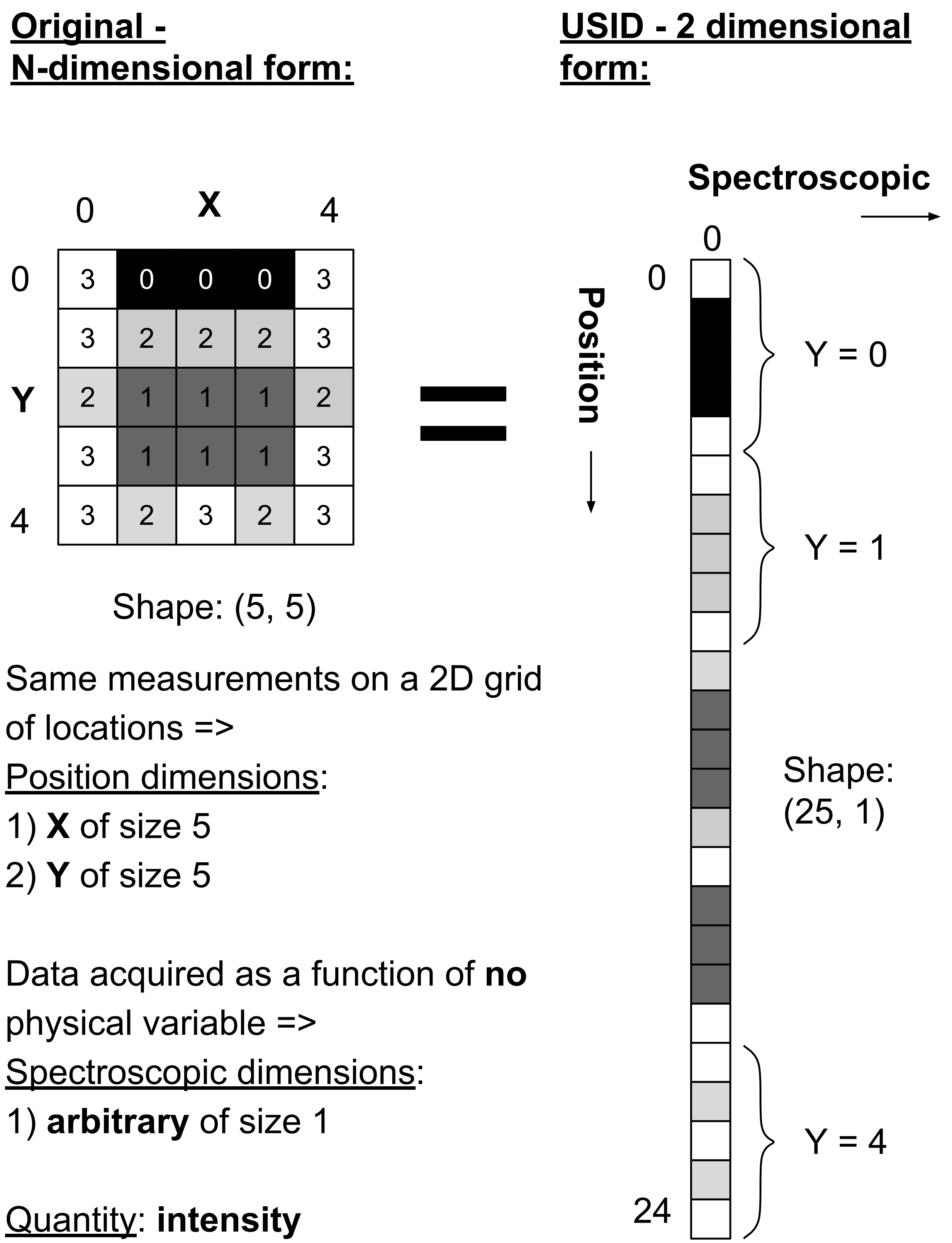
Same measurement(s) performed only once
=>
Position dimensions:
1) **X** of size 1

Data points acquired as a function of **Frequency**
=>
Spectroscopic dimensions:
1) **Frequency** of size 16

Quantity: **Amplitude**

This case encompasses examples such as a **single** Raman spectrum, force-distance curve in atomic force microscopy, current-voltage spectroscopy, etc. In this case, the measurement is recorded at a single location meaning that this dataset has a single *arbitrary* `position dimension` of size 1. At this position, data is recorded as a function of a single variable (`spectroscopic dimension`) such as *wavelength* or *frequency*. Thus, if the spectrum contained `S` data points, the **USID** representation of this data would be a `1 x S` matrix. The `quantity` represented in this data would be **Amplitude**.

Gray-scale images



Original - N-dimensional form:

USID - 2 dimensional form:

	0		X		4
0	3	0	0	0	3
	3	2	2	2	3
Y	2	1	1	1	2
	3	1	1	1	3
4	3	2	3	2	3

Shape: (5, 5)

Same measurements on a 2D grid of locations =>

Position dimensions:

1) **X** of size 5

2) **Y** of size 5

Data acquired as a function of **no** physical variable =>

Spectroscopic dimensions:

1) **arbitrary** of size 1

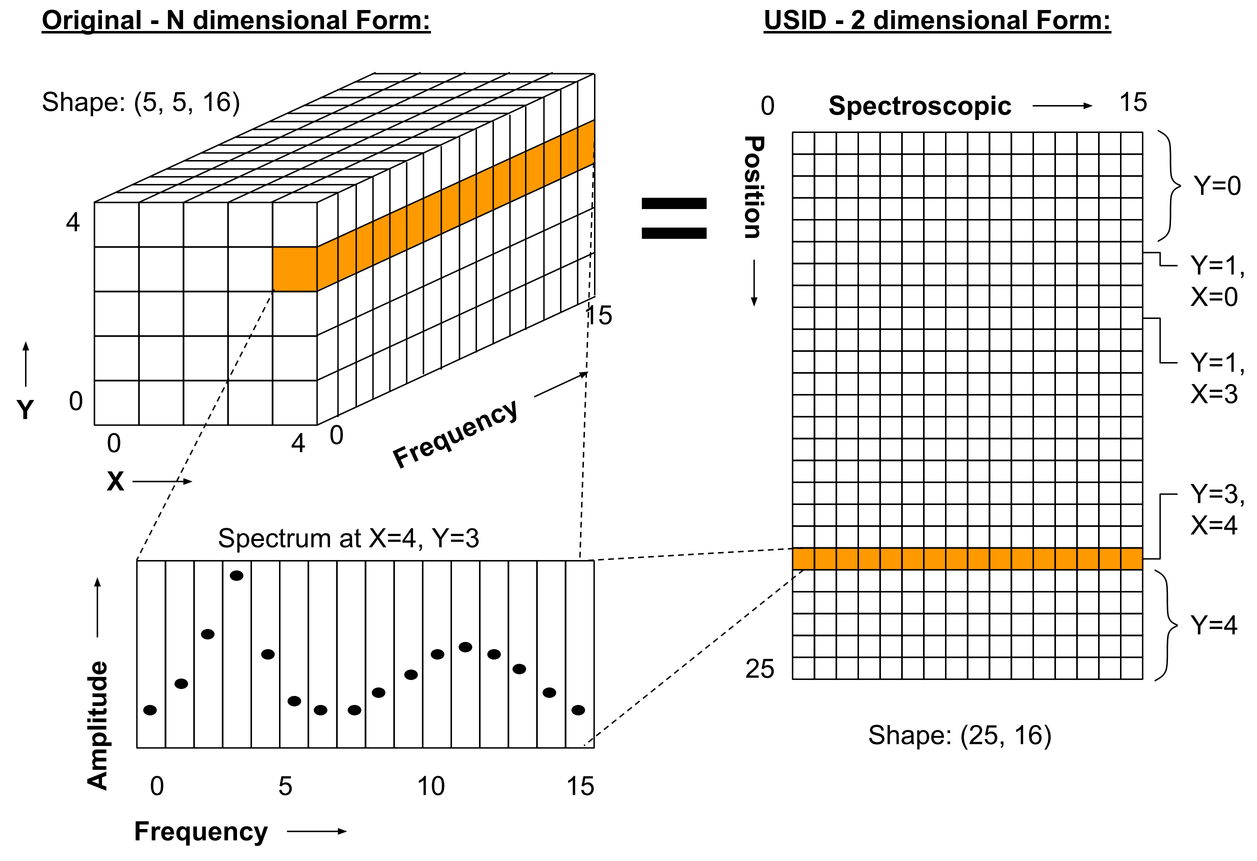
Quantity: **intensity**

In such data, a single value (`quantity` is *intensity*) in is recorded at each location in a two dimensional grid. Thus, there are are two `position dimensions` - *X*, *Y*. The value at each pixel was not really acquired as a function of any variable so the data has one *arbitrary* `spectroscopic dimension`. Thus, if the image had `P` rows and `Q` columns, it would have to be flattened and represented as a `P*Q x 1` array according to the **USID** model. The second axis has size of 1 since we only record one value (intensity) at each location. In theory, the flattened data could be arranged column-by-column (as in the figure above) and then row-by-row or vice-versa depending on how the data was (sequentially) captured. The sequence in this particular case is debatable in this particular example.

Popular examples of such data include imaging data from raster scans (e.g. - height channel in atomic force microscopy), black-and-white photographs, scanning electron microscopy (SEM) images. etc.

Color images will be discussed separately below due to some very important subtleties about the measurement.

Spectral maps



Same measurement(s) performed on a 2D grid of locations =>
Position dimensions:
1) **X** of size 5
2) **Y** of size 5

Data points acquired as a function of **Frequency** =>
Spectroscopic dimensions:
1) **Frequency** of size 16

Quantity: **Amplitude**

If a spectrum of length `S` were acquired at each location in a two dimensional grid of positions with `P` rows and `Q` columns, it would result in a three dimensional dataset. This example is a combination of the two examples above.

The above 3D dataset has two `position dimensions` - *X* and *Y*, and has one `spectroscopic dimension` - *Frequency*. Each data point in the dataset contains the same physical `quantity` - *Amplitude*. In order to represent this 3D dataset in the 2D **USID** form, the two `position dimensions` in such data would need to be flattened along the vertical axis and the spectrum at each position would be laid out along the horizontal axis or the spectroscopic axis. Thus the original `P x Q x S` 3D array would be flattened to a 2D array of shape - `P*Q x S`. Assuming that the data was acquired column-by-column and then row-by-row, the rows in the flattened 2D dataset would also be laid out in the same manner: $\mathrm{row}_0\mathrm{col}_0$, $\mathrm{row}_0\mathrm{col}_1$, $\mathrm{row}_0\mathrm{col}_2$, … , $\mathrm{row}_0\mathrm{col}_Q$, $\mathrm{row}_1\mathrm{col}_0$, $\mathrm{row}_1\mathrm{col}_1$, …

Popular examples of such datasets include Scanning Tunnelling Spectroscopy (STS) and current-voltage spectroscopy

High dimensional data

This general representation for data was developed to express datasets with 7, 8, 9, or higher dimensional datasets.

The **spectral map** example above only had one `spectroscopic dimension`. If spectra of length `S` were acquired for `T` different *Temperatures*, the resultant dataset would have two `spectroscopic dimensions` - *Frequency* and *Temperature* and would be of shape - `P x Q x T x S`. Just as the two `position dimensions` were flattened along the vertical axis in the example above, now the two spectroscopic dimensions would also need to be flattened along the horizontal axis. Thus the horizontal axis would be flattend as: $\mathrm{Temperature}_0\mathrm{Frequency}_0$, $\mathrm{Temperature}_0\mathrm{Frequency}_1$,$\mathrm{Temperature}_0\mathrm{Frequency}_2$, … , $\mathrm{Temperature}_0\mathrm{Frequency}_S$, $\mathrm{Temperature}_1\mathrm{Frequency}_0$, $\mathrm{Temperature}_1\mathrm{Frequency}_1$, … This four dimensional dataset would be flattened into a two dimensional array of shape `P*Q x T*S`.

In the same manner, one could keep adding additional dimensions to either the position or spectroscopic axis.

Non Measurements

This same flattened representation can also be applied to results of data analyses or data that were not directly recorded from an instrument. Here are some examples:

- A collection of `k` chosen spectra would also be considered `Main` datasets since the data is still structured as `[instance, features]`
- Similarly, the centroids obtained from a clustering algorithm like `k-Means clustering`
- The abundance maps obtained from decomposition algorithms like `Singular Value Decomposition (SVD)` or `Non-negative matrix factorization (NMF)`

Complicated?

This data model may seem unnecessarily complicated for very simple / rigid data such as 2D images or 1D spectra. However, bear in mind that **this paradigm was designed to represent any information regardless of dimensionality, origin, complexity**, etc. Thus, encoding data in this manner will allow seamless sharing, exchange, and interpretation of data.

Compound Datasets:

There are instances where multiple values are associate with a single position and spectroscopic value in a dataset. In these cases, we use the compound dataset functionality in HDF5 to store all of the values at each point. This also allows us to access any combination of the values without needing to read all of them. Pycroscopy actually uses compound datasets a lot more frequently than one would think. The need and utility of compound datasets are best described with examples:

- **Color images**: Each position in these datasets contain three (red, blue, green) or four (cyan, black, magenta, yellow) values. One would naturally be tempted to simply treat these datasets as `N x 3` or `N x 4` datasets, (where `N` is the product of the number of *rows* and *columns* as in the gray-scale image example above) and it certainly is not wrong to represent data this way. However, storing the data in this manner would mean that the *red* intensity was collected first, followed by the *green*, and finally by the *blue*. In other words, **a notion of chronology is attached to both the position and spectroscopic axes** according to the **USID** definition. While the intensities for each color may be acquired sequentially in detectors, since we are not aware of the exact sequence we will assume that the *red*, *green*, and *blue* values are acquired simultaneously for simultaneously.

  In these cases, we store data using `compound datasets` that allow the storage of multiple pieces of data within the same `cell`. While this may seem confusing or implausible, remember that computers store complex numbers in the same way. The complex numbers have a *real* and an *imaginary* component just like color images have *red*, *blue*, and *green* components that describe a single pixel. Therefore, color images in the **USID** representation would be represented by a `N x 1` matrix with compound values instead of a `N x 3` matrix with real or integer values. For example, one would refer to the *red* component at a particular position as:

  ```
  red_value = dataset_name[position_index, spectroscopic_index]['red']
  ```

- **Functional fits**: Let's take the example of a dataset flattened to shape - `N x P`, whose spectra at each location are fitted to a complicated equation. Now, the `P` points in the spectra will be represented by `S` coefficients that don't necessarily follow any order. Consequently, the result of the functional fit should actually be a `N x 1` dataset where each element is a compound value made up of the `S` coefficients. Note that while some form of sequence can be forced onto the coefficients if the spectra were fit to polynomial functions, the drawbacks outweigh the benefits:

  – **Slicing**: Storing data in compound datasets circumvents problems associated with getting a specific / the `kth` coefficient if the data were stored in a real-valued matrix instead.

  – **Visualization** also becomes a lot simpler since compound datasets cannot be plotted without specifying the component / coefficient of interest. This avoids plots with alternating coefficients that are several orders of magnitude larger / smaller than each other.

While one could represent multiple channels of information simultaneously acquired by instruments (for example - height, amplitude, phase channels in atomic force microscopy scan images) using compound datasets, this is **not** the intended purpose of compound datasets. We use recommend storing each channel of information separately for consistency across scientific disciplines. For example, there are modalities in microscopy where some channels provide high resolution topography data while others provide low-resolution but spectroscopy data.

For more information on compound datasets see the h5py Datasets documentation from the HDF Group.

`Ancillary` Datasets

So far we have explained how the (`main`) dataset of interest can be flattened and represented regardless of its origin, size, dimensionality, etc. In order to make this `main` dataset **self-explanatory**, additional pieces of information are required. For example, while the `main` dataset preserves the data of interest, information regarding the original dimensionality of the data or the combination of parameters corresponding to each observation is not captured.

In order to capture such vital information, each `main` dataset is always accompanied by **four** `ancillary` datasets. These are the:

- The `Position Values` and `Position Indices` that describe the index and value of any given row or spatial position in the `main` dataset.
- The `Spectroscopic Values` and `Spectroscopic Indices` that describe the index and values all columns in the `main` dataset for all spectroscopic dimensions.

The pair of `Values` datasets are analogous to legends for maps. In other words, the pair of `Values` datasets **provide the combination of the values for each dimension** / variable that correspond to a particular data point in the `main` dataset. For example, one would be able to understand readily that a particular data point in the `main` dataset was acquired for the reference values of *Frequency* of 315 kHz, *Temperature* of 400 K from the `Spectroscopic Values` dataset and location *X* of 7.125 microns and *Y* of 480 nanometers from the `Position Values` dataset.

The pair of `Indices` datasets are essentially **counters for each position and spectroscopic dimension** / variable. Continuing the example presented for the `Values` datasets, let's assume that the data was acquired as a function of all unique combinations of `37` *Frequency* values, `12` *Temperatures*, `64` locations in the *X* direction and `128` values in the *Y* direction. Then, the `Spectroscopic Indices` dataset would instruct that the given data point in the `main` dataset corresponds to the `13th` *Frequency* value and `5th` *Temperature* value. In the same way, the `Position Indices` dataset would show that the data point of interest corresponds to the `47th` value of *X* and `106th` value of *Y*.

The pair of `Indices` datasets are critical for explaining:

- the original dimensionality of the dataset
- how to reshape the data back to its N dimensional form

Much like `main` datasets, the `ancillary` datasets are also two dimensional matrices regardless of the number of `position` or `spectroscopic dimensions`. Given a `main` dataset with `N` positions, each containing `P` spectral values (shape = (`N x P`)), and having `U` `position dimensions` and `V` `spectroscopic dimensions`:

- The `Position Indices` and `Position Values` datasets would both of the same size of `N x U`, where `U` is the number of `position dimensions`. The **columns would be arranged in ascending order of rate of change**. In other words, the first column would be the fastest changing position dimension and the last column would be the slowest. **Each position dimension gets it's own column**.
- The `Spectroscopic Values` and `Spectroscopic Indices` dataset would both be `V x S` in shape, where `V` is the number of `spectroscopic dimensions`. Similarly to the `position dimensions`, the first row would be the fastest changing `spectroscopic dimension` while the last row would be the slowest varying dimension. **Each spectroscopic dimension gets it's own row**.

The `ancillary` datasets are better illustrated via a few examples. We will be continuing with the same examples used when illustrating the `main` dataset.

Spectrum

Let's assume that data points were collected as a function of 8 values of the (sole) variable / `spectroscopic dimension` - *Frequency*. In that case, the `Spectroscopic Values` dataset would be of size `1 x 5` (one row for the single `spectroscopic dimension` and eight columns for each of the reference *Frequency* steps. Let's assume that the data was collected as a function of *Frequency* over a band ranging from `300` to `320` kHz. In that case, the `Spectroscopic Values` would be as shown below:

Frequency	300	305	310	315	320

This means that for all positions in the `main` dataset, the `4th` column would always correspond to data collected for the *Frequency* of `315 kHz`.

As the name suggests, the `Spectroscopic Indices` dataset only shows the indices for the steps in the dimension. In this particular case, the dataset is trivial and just a linearly increasing array.

Note that indices start from `0` instead of `1` and end at `5-1` instead of `5` in line with common programming languages such as *C* or *python* as shown below:

Frequency	0	1	2	3	4

Given that the spectrum only had a single *arbitrary* `position dimension` which was varied over a single (arbitrary) value, the `Position Indices` and `Position Values` datasets would have a shape of `1 x 1`.

`Position Indices`:

arb.
0

`Position Values`:

arb.
0.0

Gray-scale image

A simple gray-scale image with `X` pixels in the horizontal and `Y` pixels in the vertical direction would have ancillary position datasets of shape `X*Y x 2`. The first column in the ancillary position datasets would correspond to the index / values of the dimension - `X` (assuming that it is the dimension that varies fastest) and the second column in the ancillary position dataset would be the dimension - `Y` assuming that the data was collected column-by-column and then row-by-row just as in the example above.

If the original image had 3 pixels in the horizontal direction and 2 pixels in the vertical direction, the corresponding `Position Indices` dataset would be:

X	Y
0	0
1	0
2	0
0	1
1	1
2	1

Notice that the index for `X` is reset to `0` when `Y` is incremented from `0` to `1` in the fourth row. As mentioned earlier, the data in such `Indices` datasets are essentially counters.

Correspondingly, if the measurements were performed at `X` locations: `0.0, 1.5, and 3.0` *microns* and `Y` locations: `-70` and `23` *nanometers*, the `Position Values` dataset may look like the table below:

X	Y
0.0	-70.0
1.5	-70.0
3.0	-70.0
0.0	23.0
1.5	23.0
3.0	23.0

Thus, the `5th` row in the `main dataset` for this gray-scale image would correspond to data collected at `X = 1.5 microns` and `Y = 23 nanometers` according to the `Position Values` dataset.

Note that `X` and `Y` dimensions have **different units** - microns and nanometers. Pycroscopy has been designed to handle variations in the units for each of these dimensions. Details regarding how and where to store the information regarding the `labels` ('X', 'Y') and `units` for these dimensions ('um', 'nm') will be discussed in the `Implementation` section.

Similar to the `position dimensions` for a spectrum, gray-scale images only have a single *arbitrary* `spectroscopic dimension`. Thus, both `Spectroscopic` datasets have shape of `1 x 1`:

`Spectroscopic Indices:`

arb.	0

`Spectroscopic Values:`

arb.	0

Spectral maps

Let's continue the example on **spectral maps**, which has two `position dimensions` - *X* and *Y*, and one `spectroscopic dimension` - *Frequency*. If the dataset was varied over `3` values of *X*, `2` values of *Y* and `5` values of *Frequency*, the `ancillary` datasets would be based on the solutions for the two examples above:

`Position Indices:`

X	Y
0	0
1	0
2	0
0	1
1	1
2	1

`Position Values:`

X	Y
0.0	-70.0
1.5	-70.0
3.0	-70.0
0.0	23.0
1.5	23.0
3.0	23.0

`Spectroscopic Indices:`

Frequency	0	1	2	3	4

`Spectroscopic Values:`

Frequency	300	305	310	315	320

High dimensional data

Continuing with the expansion of the **spectral maps** example - if the data was recorded as a function of `3` *Temperatures* in addition to recording data as a function of *Frequency* as in the above example, we wold have two `spectroscopic` dimensions - *Frequency*, and *Temperature*. Thus, the ancillary spectroscopic datasets would now have a shape of 2 x 5*3 instead of the simpler 1 x 5. The value 2 on the first index corresponds to the two spectroscopic dimensions and the longer (15 instead of 5) second axis corresponds to the fact that the spectra is now recorded thrice at each position (once for each *Frequency*). Assuming that the *Frequency* varies faster than the *Temperature* dimension (i.e.- the *Frequency* is varied from 300 to 320 for a *Temperature* of 30 C, **then** the *Frequency* is varied from 300 to 320 for a *Temperature* of 40 C and so on), the Spectroscopic Indices would be as follows:

Frequency	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
Temperature	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2

Correspondingly, the Spectroscopic Values would look like:

Frequency	300	305	310	315	320	300	305	310	315	320	300	305	310	315	320
Temperature	30	30	30	30	30	40	40	40	40	40	50	50	50	50	50

Since the manner and values over which the positions are varied remains unchanged from the *spectral maps* example, the Position Indices and Position Values datasets for this example would be identical those of the *spectral maps* example

A simple glance at the shape of the ancillary datasets for this (or any) example would be enough to reveal that the data has two position dimensions (two columns in the Position Indices dataset) and two spectroscopic dimensions (two rows in the Spectroscopic Indices dataset) dataset)

In the same manner, additional dimensions can be added to the main and appropriate ancillary datasets thus proving that this data model can indeed accommodate data of any size, complexity, or dimensionality.

Channels

The **USID** model also allows the representation and capture of **information acquired simultaneously from multiple sources** through Channels. Each Channel would contain a **separate** main dataset. Ancillary datasets can be shared across channels if the position or spectroscopic dimensions are identical.

As alluded to earlier, the most popular example many people can relate to are the various channels of information recorded during a conventional scanning probe microscopy raster scan (*Height*, *Amplitude*, *Phase*). For this example, all the channels could share the same set of four ancillary datasets.

It is not necessary that rate of acquisition match across channels. For example, one channel could be a high-resolution topography scan (similar to 2D gray-scale image) while another channel could contain spectra collected at each location on a **coarser** grid of positions (3D spectral-map dataset). In this case, the two channels may not be able to share ancillary datasets.

Specifics regarding the implementation of different channels will be discussed in a later section.

1.7.4 File Format

Requirements

No one really wants yet another file format in their lives. We wanted to adopt a file format that satisfies some basic requirements:

- already widely accepted in scientific research
- support parallel read and write capabilities.

- store multiple datasets of different shapes, dimensionalities, precision and sizes.
- scale very efficiently from few kilobytes to several terabytes
- can be (readily) read and modified using any language including Python, R, Matlab, C/C++, Java, Fortran, Igor Pro, etc. without requiring installation of modules that are hard to install
- store and organize data in a intuitive and familiar hierarchical / tree-like structure that is similar to files and folders in personal computers.
- facilitates storage of any number of experimental or analysis parameters in addition to regular data.
- highly flexible and poses minimal restrictions on how the data can and should be stored.
- readily compatible with high-performance computing (`HPC`) and (soon) cloud-computing.

Candidates

- We found that existing file formats in science such as the Nexus data format, XDMF, and NetCDF:
  - were designed for **specific / narrow scientific domains only** and we did not want to shoehorn our data structure into those formats.
  - Furthermore, despite being some of the more popular scientific data formats, it is **not immediately straightforward to read those files** on every computer using any programming language. For example - the Anaconda python distribution does not come with any packages for reading these file formats.
- Adios is perhaps the ultimate file format for storing petabyte sized data on supercomputers but it was specifically designed for simulations, check-pointing, and it trades flexibility, and ease-of-use for performance.
- The hierarchical data format (HDF5) is the implicitly or explicitly the de-facto standard in scientific research. In fact, Nexus, NetCDF, and even Matlab's .mat files are actually (now) just custom flavors of HDF5 thereby validating the statement that HDF5 is the **unanimous the file format of choice**
- The DREAM.3D is yet another group that uses HDF5 as the base container to store their data. We are currently evaluating compatibility with and feasibility of their data model.

We found that HDF5, works best for us compared to the alternatives. Hence, we have implemented the **USID** model into the HDF5 file format and such file will be referred to as **h5USID** files.

We acknowledge that it is nearly impossible to find the perfect file format and HDF5 too has its fair share of drawbacks. One common observation among file formats is that a file format optimized for the cloud or cluster computing often does not perform well (or at all) on HPC due to the conflicting nature of the computing paradigms. As of this writing, HDF5 is optimized for HPC and not for cloud-based applications. For cloud-based environments it is beneficial to in fact break up the data into small chunks that can be individually addressed and used. We think Zarr and N5 would be good alternatives; however, most of these file formats are very much in their infancy and have not proven themselves like HDF5 has. This being said, the HDF organization just announced a cloud flavor of HDF5 and we plan to look into this once h5py or other python packages support such capabilities.

1.7.5 h5USID - USID in HDF5

Here we discuss guidelines and specifications for implementing the **USID** model into HDF5 files. While we could impose that the file extension be changed from **.hdf5** to **.h5USID**, we choose to retain the **.hdf5** extension so that other software are aware of the general file type and can recognize / read them easily.

Quick basics of HDF5

Information can be stored in HDF5 files in several ways:

- `Datasets` allow the storage of data matrices and these are the vessels used for storing the `main`, `ancillary`, and any extra data matrices
- `Groups` are similar to folders in conventional file systems and can be used to store any number of datasets or groups themselves
- `Attributes` are small pieces of information, such as experimental or analytical parameters, that are stored in key-value pairs in the same way as dictionaries in python. Both groups and datasets can store attributes.
- While they are not means to store data, `Links` or `references` can be used to provide shortcuts and aliases to datasets and groups. This feature is especially useful for avoiding duplication of datasets when two `main` datasets use the same ancillary datasets.

`Main` data:

Dataset structured as (positions x time or spectroscopic values)

- `dtype` : uint8, float32, complex64, compound if necessary, etc.
- *Required* attributes:
  - `quantity` - Single string that explains the data. The physical quantity contained in each cell of the dataset – eg – 'Current' or 'Deflection'
  - `units` – Single string for units. The units for the physical quantity like 'nA', 'V', 'pF', etc.
  - `Position_Indices` - Reference to the position indices dataset
  - `Position_Values` - Reference to the position values dataset
  - `Spectroscopic_Indices` - Reference to the spectroscopic indices dataset
  - `Spectroscopic_Values` - Reference to the spectroscopic values dataset
- chunking : HDF group recommends that chunks be between 100 kB to 1 MB. We recommend chunking by whole number of positions since data is more likely to be read by position rather than by specific spectral indices.

Note that we are only storing references to the ancillary datasets. This allows multiple `main` datasets to share the same ancillary datasets without having to duplicate them.

`Ancillary` data:

`Position_Indices` structured as (`positions` x `spatial dimensions`)

- dimensions are arranged in ascending order of rate of change. In other words, the fastest changing dimension is in the first column and the slowest is in the last or rightmost column.
- `dtype` : uint32
- Required attributes:
  - `labels` - list of strings for the column names like ['X', 'Y']
  - `units` – list of strings for units like ['um', 'nm']
- Optional attributes: * Region references based on column names

`Position_Values` structured as (`positions` x `spatial dimensions`)

- dimensions are arranged in ascending order of rate of change. In other words, the fastest changing dimension is in the first column and the slowest is in the last or rightmost column.
- `dtype` : float32

- Required attributes:
  - `labels` - list of strings for the column names like [‘X’, ‘Y’]
  - `units` – list of strings for units like [‘um’, ‘nm’]
- Optional attributes: * Region references based on column names

`Spectroscopic_Indices` structured as (`spectroscopic dimensions` x `time`)

- dimensions are arranged in ascending order of rate of change. In other words, the fastest changing dimension is in the first row and the slowest is in the last or lowermost row.
- `dtype` : uint32
- Required attributes:
  - `labels` - list of strings for the column names like [‘Bias’, ‘Cycle’]
  - `units` – list of strings for units like [‘V’, ‘’]. Empty string for dimensionless quantities
- Optional attributes: * Region references based on row names

`Spectroscopic_Values` structured as (`spectroscopic dimensions` x `time`)

- dimensions are arranged in ascending order of rate of change. In other words, the fastest changing dimension is in the first row and the slowest is in the last or lowermost row.
- `dtype` : float32
- Required attributes:
  - `labels` - list of strings for the column names like [‘Bias’, ‘Cycle’]
  - `units` – list of strings for units like [‘V’, ‘’]. Empty string for dimensionless quantities
- Optional attributes:
  - Region references based on row names

Attributes

All groups and (at least `Main`) datasets must be created with the following **mandatory** attributes for better traceability:

- `time_stamp` : ‘2017_08_15-22_15_45’ (date and time of creation of the group or dataset formatted as ‘YYYY_MM_DD-HH_mm_ss’ as a string)
- `machine_id` : ‘mac1234.ornl.gov’ (a fully qualified domain name as a string)
- `pyUSID_version` : ‘0.0.1’
- `platform` : ‘Windows10. . . .’ or something like ‘Darwin-17.4.0-x86_64-i386-64bit’ (for Mac OS) - a long string providing detailed information about the operating system

Groups

HDF5 Groups in **h5USID** are used to organize categories of information (raw measurements from instruments, results from data analysis, etc.) in an intuitive manner.

Measurement data

- As mentioned earlier, instrument users may change experimental parameters during measurements. Even if these changes are minor, they can lead to misinterpretation of data if the changes are not handled robustly. To solve this problem, we recommend storing data under **indexed** groups named as `Measurement_00x`. Each time the parameters are changed, the dataset is truncated to the point until which data was collected and a new group is created to store the upcoming new measurement data.
- Each **channel** of information acquired during the measurement gets its own group.
- The `Main` datasets would reside within these channel groups.
- Similar to the measurement groups, the channel groups are named as `Channel_00x`. The index for the group is incremented according to the index of the information channel.
- Depending on the circumstances, the ancillary datasets can be shared among channels.
  - Instead of the main dataset in `Channel_001` having references to the ancillary datasets in `Channel_000`, we recommend placing the ancillary datasets outside the Channel groups in a area common to both channel groups. Typically, this is the `Measurement_00x` group.
- This is what the tree structure in the file looks like when experimental parameters were changed twice and there are two channels of information being acquired during the measurements.
- Datasets common to all measurement groups (perhaps some calibration data that is acquired only once before all measurements)
- `Measurement_000` (group)
  - `Channel_000` (group)
    - Datasets here
  - `Channel_001` (group)
    - Datasets here
  - Datasets common to `Channel_000` and `Channel_001`
- `Measurement_001` (group)
  - `Channel_000` (group)
    - Datasets here
  - `Channel_001` (group)
    - Datasets here
  - Datasets common to `Channel_000` and `Channel_001`
- ...

Tool (analysis / processing)

- Each time an analysis or processing routine, referred generally as `tool`, is performed on a dataset of interest, the results are stored in new HDF5 datasets within a new HSF5 group.
- A completely new dataset(s) and group are created even if a minor operation is being performed on the dataset. In other words, we **do NOT modify existing datasets**.
- Almost always, the tool is applied to one (or more) `main` datasets (referred to as the `source` dataset) and at least one of the results is typically also a `main` dataset. These new `main` datasets will either need to be linked to the ancillary matrices of the `source` or to new ancillary datasets that will need to be created.

- The resultant dataset(s) are always stored in a group whose name is derived from the names of the tool and the dataset. This makes the data **traceable**, meaning that the names of the datasets and groups are sufficient to understand what processing or analysis steps were applied to the data to bring it to a particular point.
- The group is named as `Source_Dataset-Tool_Name_00x`, where a `tool` named `Tool_Name` is applied to a `main` dataset named `Source_Dataset`.
  - Since there is a possibility that the same tool could be applied to the very same dataset multiple times, we store the results of each run of the tool in a separate group. These groups are differentiated by the index that is appended to the name of the group.
  - Note that a `-` separates the dataset name from the tool name and anything after the last _ will be assumed to be the index of the group
  - Please refer to the advanced topics section for tools that have **more than one** `source` datasets
- In general, the results from tools applied to datasets should be stored as:
  - `Source_Dataset`
  - `Source_Dataset-Tool_Name_000` (group containing results from first run of the `tool` on `Source_Dataset`)
    - Attributes:
      - all mandatory attributes
      - `algorithm`
      - Other tool-relevant attributes
      - `source_000` - reference to `Source_Dataset`
    - `Dataset_Result0`
    - `Dataset_Result1` . . .
  - `Source_Dataset-Tool_Name_001` (group containing results from second run of the `tool` on `Source_Dataset`)
- This methodology is illustrated with an example of applying `K-Means Clustering` on the `Raw_Data` acquired from a measurement:
  - `Raw_Data` (`main` dataset)
  - `Raw_Data-Cluster_000` (group)
  - Attributes:
    - all mandatory attributes
    - `algorithm` : 'K-Means'
    - `source_000` : reference to `Raw_Data`
  - `Label_Indices` (ancillary spectroscopic dataset with 1 dimension of size 1)
  - `Label_Values` (ancillary spectroscopic dataset with 1 dimension of size 1)
  - `Labels` (Main dataset)
    - Attributes:
      - `quantity` : 'Cluster labels'
      - `units` : 'a. u.'
      - `Position_Indices` : Reference to `Position_Indices` from attribute of `Raw_Data`

        - Position_Values : Reference to Position_Values from attribute of Raw_Data
        - Spectroscopic_Indices : Reference to Label_Indices
        - Spectroscopic_Values : Reference to Label_Values
        - all mandatory attributes
  - Cluster_Indices (ancillary positions dataset with 1 dimension of size equal to number of clusters)
  - Cluster_Values (ancillary positions dataset with 1 dimension of size equal to number of clusters)
  - Mean_Response (main dataset) <- This dataset stores the endmembers or mean response for each cluster
    - Attributes:
      - quantity : copy from the quantity attribute in Raw_Data
      - units : copy from the units attribute in Raw_Data
      - Position_Indices : Reference to Cluster_Indices
      - Position_Values : Reference to Cluster_Values
      - Spectroscopic_Indices : Reference to Spectroscopic_Indices from attribute of Raw_Data
      - Spectroscopic_Values : Reference to Spectroscopic_Values from attribute of Raw_Data
      - all mandatory attributes

- Note that the spectroscopic datasets that the Labels dataset link to are not called Spectroscopic_Indices or Spectroscopic_Values themselves. They only need to follow the specifications outlined above. The same is true for the position datasets for Mean_Response.

1.7.6 Advanced topics

Region references

These are references to sections of a main or ancillary dataset that make it easy to access data specific to a specific portion of the measurement, or each column or row in the ancillary datasets just by their alias (intuitive strings for names).

We have observed that the average **USID** user does not tend to use region references as much as we thought they might. Therefore, we do not require or enforce that region references be used

Processing on multiple Main datasets

One popular scientific workflow we anticipate involves the usage of multiple source datasets to create results. By definition, this breaks the current nomenclature of HDF5 groups that will contain results. This will be addressed by restructuring the code in such a way that the results group could be named as: Multi_Dataset-Tool_Name_000. To improve the robustness of the solution, we have already begun storing the necessary information as attributes of the HDF5 results groups. Here are the attributes of the group that we expect to capture the references to all the datasets along with the name of the tool while relaxing the restrictions on the aforementioned nomenclature:

- tool : <string> - Name of the tool / process applied to the datasets
- num_sources: <unsigned integer> - Number of source datasets that take part in the process
- source_000 : <HDF5 object reference> - reference to the first source dataset

- `source_001` : <HDF5 object reference> - reference to the second source dataset . . .

We would have to break the list of references to the source datasets into individual attributes since h5py / HDF5 currently does not allow the value of an attribute to be a list of object references.

Sparse Sampling / Compressed Sensing

In many cases, especially electron and ion based microscopy, the very act of probing the sample damages the sample. In order to minimize damage to the sample, researchers only sample data from a few random positions in the 2D grid of positions and use advanced algorithms to reconstruct the missing data. This scientific problem presents a data storage challenge. The naive approach would be to store a giant matrix of zeros with only a available positions filled in. This is highly inefficient since the space occupied by the data would be equal to that of the complete (non-sparse) dataset.

For such sparse sampling problems, we propose that the indices for each position be identical and still range from `0` to `N-1` for a dataset with `N` randomly sampled positions. Thus, for an example dataset with two position dimensions, the indices would be arranged as:

X	Y
0	0
1	1
2	2
.	.
N-2	N-2
N-1	N-1

However, the position values would contain the actual values:

X	Y
9.5	1.5
3.6	7.4
5.4	8.2
.	.
1.2	3.9
4.8	6.1

The spectroscopic ancillary datasets would be constructed and defined in the traditional methods since the sampling in the spectroscopic dimension is identical for all measurements.

The vast majority of the existing features including signal filtering, statistical machine learning algorithms, etc. in child packages like pycroscopy could still be applied to such datasets.

By nature of its definition, such a dataset will certainly pose problems when attempting to reshape to its N-dimensional form among other things. Pycroscopy currently does not have any scientific algorithms or real datasets specifically written for such data but this will be addressed in the near future. This is section is presented to show that we have indeed thought about such advanced problems as well when designing the universal data structure.

1.8 Guidelines for Contribution

1.8.1 Structuring code

We would like to thank you and several others who have offered their code. We are more than happy to add your code to this project. Just as we strive to ensure that you get the best possible software from us, we ask that you do the same for others. We do NOT ask that your code be as efficient as possible. Instead, we have some simpler and easier requests. We have compiled a list of best practices below with links to additional information. If you are confused or need more help, please feel free to contact us:

- Encapsulate independent sections of your code into functions that can be used individually if required.
- Ensure that your code (functions) is well documented (numpy format) - expected inputs and outputs, purpose of functions
- Please avoid naming variables with single alphabets like `i` or `k`. This makes it challenging to find and fix bugs.
- Ensure that your code works in python 2.7 and python 3.5 (ideally using packages that are easy to install on Windows, Mac, and Linux). It is quite likely that packages included within Anaconda (upon which pyUSID is based and has a comprehensive list packages for science and data analysis + visualization) can handle most needs. If this is not possible, try to use packages that are easy to to install (pip install). If even this is not possible, try to use packages that at least have conda installers.
- Please ensure that your code files fit into our package structure (`io`, `processing`, `viz`)
- There is a good chance that pyUSID may already have some plotting or data analysis function that you need / already have. Please check against our tutorials and examples. In such cases, consider either improving the function already in pyUSID or reusing code already within pyUSID.
- Provide a few examples on how one might use your code - preferably via a jupyter notebook.
- Follow best practices for PEP8 compatibility. The easiest way to ensure compatibility is to set it up in your development environment. PyCharm does this by default. So, as long as PyCharm does not raise many warning, your code is beautiful!

You can look at our code in our GitHub project to get an idea of how we organize, document, and submit our code.

1.8.2 Contributing code

We recommend that you follow the steps below. Again, if you are ever need help, please contact us:

1. Learn `git` if you are not already familiar with it. See our compilation of tutorials and guides, especially this one.
2. Create a `fork` of pyUSID - this creates a separate copy of the entire pyUSID repository under your user ID. For more information see instructions here.
3. Once inside your own fork, you can either work directly off `master` or create a new branch.
4. Add / modify code
5. `Commit` your changes (equivalent to saving locally on your laptop). Do this regularly.
6. Repeat steps 4-5. After you reach a certain milestone, `push` your commits to your `remote branch`. To avoid losing work due to problems with your computer, consider `pushing commits` once at least every day / every few days.
7. Repeat steps 4-6 till you are ready to have your code added to the parent pyUSID repository. At this point, create a pull request. Someone on the development team will review your `pull request` and then `merge` these changes to `master`.

1.8.3 Writing tests

Unit tests are a good start for ensuring that you spend more time using code than fixing it. New functions / classes must be accompanied with unit tests. Additionally, examples on how to use the new code must also be added so others are aware about how to use the code. Fortunately, it is rather straightforward to turn unit tests into examples.

1.9 Frequently asked questions

Contents

1.9.1 pyUSID philosophy

Is pyUSID specific to any communities?

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

Also, please see our answer to 'Who uses pyUSID' below:

Who uses pyUSID?

- The Institute for Functional Imaging of Materials (IFIM) at Oak Ridge National Laboratory uses pycroscopy (built on pyUSID) 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.

Why is pyUSID written in python and not C / Fortran / Julia?

Here are some of the main reasons pyUSID is written in python:

- **Ease of use**: One of the main objectives of pyUSID is to **lower the barrier** to advanced data analytics for domain scientists such as material scientists. A C++ / Fortran version of pyUSID would certainly have been more efficient than the current python code-base. However, the learning curve for writing efficient C code is far steeper compared to python / Julia / Matlab for the average domain scientist. Focusing on science is a big enough job for domain scientists and we want to make it as easy as possible to adopt pyUSID even for those who are novices at programming.
- **Optimized core packages**: Furthermore, our code makes heavy use of highly efficient numerical and scientific libraries such as Numpy and Scipy that are comparable in speed to C so we do not expect our code to be substantially slower than C / Fortran.
- **Support**: Julia is a (relatively) new language similar to python that promises to be as fast as C and as easy as python and purpose-built for efficient computing. However, as of this writing, Julia unfortunately still does not have open-source package ecosystem that is as large or diverse (think of the many packages necessary to read obscure proprietary file formats generated by instruments as an example) as python.
- **Industry standard**: Furthermore, python's unchallenged leadership in the data analytics / deep learning field have only validated it as the language of choice.

We welcome you to develop application programming interfaces (APIs) for languages besides python.

pyUSID is written in python, so it is going to be slow since it cannot use all the cores on my CPU, right?

Actually, all data processing / analysis algorithms we have written using `pyUSID.Process` so far can use every single core on your CPU. Given N CPU cores, you should notice a nearly N-fold speed up in your computation. Note that the goal of pyUSID was never to maximize performance but rather to simplify and lower the barrier for the average scientist who may not be an expert programmer. By default, we set aside 1-2 cores for the operating system and other user applications such as an internet browser, Microsoft Word, etc.

1.9.2 Using pyUSID

I don't know programming. Does this preclude me from using pyUSID?

Not at all. One of the tenets of pyUSID is lowering the barrier for scientists and researchers. To this end, we have put together a list of useful tutorials and examples and examples to guide you. You should have no trouble getting started even if you do not know programming. That being said, you would be able to make the fullest use of pyUSID if you knew basic programming in python.

What sort of computer do I need to run pyUSID?

You can use practically any laptop / desktop / virtual machine running Windows / Mac OS / Linux. pyUSID is not tested on 32 bit operating systems (very rare).

I am not able to find an example on topic X / I find tutorial Y confusing / I need help!

We appreciate your feedback regarding the documentation. Please contact us and we will add / improve our documentation.

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.

How can I reference pyUSID?

For now, please use: *Somnath, Suhas, Chris R. Smith, and Stephen Jesse. pyUSID. Computer software. Vers. 0.0.1. Oak Ridge National Laboratory, 01 June 2018. Web. <https://pycroscopy.github.io/pyUSID/about.html>*.

We are writing a journal paper that you should be able to cite soon.

1.9.3 Data

What do you mean by multidimensional data?

Please refer to the nomenclature section of our data and file formatting document

Why not use established file formats from other domains?

In our documentation about the data structure and file format we discuss our requirements and our thoughts about the feasibility of established scientific formats including:

- Nexus data format
- Nearly Raw Raster Data (NRRD)

- XDMF,
- NetCDF
- Matlab's .mat
- Adios

We found that established community standards (like Nexus, XDMF, NetCDF, NRRD):

- were designed for specific / narrow scientific domains only and we did not want to shoehorn our data structure into those formats.
- it is not immediately straightforward to read those files on every computer using any programming language.

Unlike Nexus, NetCDF, Matlab's .mat files, the Universal Spectroscopy and Imaging Data (USID) Model does not impose any strict restrictions or requirements on the HDF5 file structure. Instead, implementing the USID model only increases the functionality of the very same datasets in pyUSID and other packages.

We are currently exploring collaboration / translators to and from DREAM.3D's HDF5 data model

1.9.4 Becoming a part of the effort

I don't know python / I don't think I write great python code. Does this preclude me from contributing to pyUSID?

Not really. Python is far easier to learn than many languages. If you know Matlab, Julia, C++, Fortran or any other programming language. You should not have a hard time reading our code or contributing to the codebase.

You can still contribute your code.

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 pyUSID.
- Tell us what you think of our documentation or share your own.
- Let us know what you would like to see in pyUSID.
- Put us in touch with others working on similar efforts so that we can join forces.

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 pyUSID 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.

There are several things we want to improve or add. Please get in touch to start a conversation.

Can you add my code to pyUSID?

Please see our guidelines for contributing code

1.10 Contact us

- Join our google group to discuss about pyUSID, ask questions, report bugs, get help, etc.
- Alternatively, raise an issue if you find any bugs or if you want a feature added to pyUSID. 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 the function you called that generated the error, details regarding your computer, operating system, python, pyUSID and package versions etc.

1.11 Credits

The core pyUSID team consists of:

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

USID was conceived by @stephenjesse (Stephen Jesse)

Substantial contributions from many developers including:

- @str-eat (Daniel Streater)
- @nmosto (Nick Mostovych)
- and many more

1.11.1 Acknowledgements

- 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 pyUSID, we would like to thank the developers of the following software packages:
  - Anaconda
  - PyCharm
  - GitKraken

1.11.2 Funding

- pyUSID and pycroscopy were initially developed through grants from:
  - Oak Ridge National Laboratory - Laboratory Director’s Research and Development fund
  - U.S. Department of Energy, Office of Science
- pyUSID is currently not officially or directly funded by any program and is developed by the contributors to the project

1.12 What's New

1.12.1 v 0.0.1 - Jun 22 2018: <https://github.com/pycroscopy/pyUSID/releases/tag/0.0.1

- PyUSID is a migration of the engineering-specific components out of a child project - pycroscopy (code was migrated from pycroscopy.core)

1.13 Upgrading from Matlab

Chris R. Smith

Here are some one-to-one translations for many popular functions in Matlab and python that should make it easier to switch from Matlab to Python

1.13.1 System functions

Matlab Function	Python Equivalent	Description
addpath	sys.path.append	Add to path

1.13.2 File I/O

Matlab Function	Python Equivalent	Description
dlmread	either read and parse or skimage.io.imread	Read ASCII-delimited file of numeric data into matrix
imread	pyplot.imread	read image file; N is number of files used

1.13.3 Data Type

Matlab Function	Python Equivalent	Description
int	numpy.int	Convert data to signed integer
double	numpy.float	Convert data to double
real	numpy.real	Return the real part of a complex number
imag	numpy.imag	Return the imaginary part of a complex number

1.13.4 Mathematics

Matlab Function	Python Equivalent	Description
sqrt	math.sqrt or numpy.sqrt	Square root
erf	math.erf or scipy.special.erf	Error function
atan2	math.erf or numpy.atan2	Four-quadrant inverse tangent
abs	abs or numpy.abs	Absolute value
exp	exp or numpy.exp	Exponential function
sin	sin or numpy.sin	Sine function

1.13.5 Array Creation

Matlab Function	Python Equivalent	Description
zeros	numpy.zeros	Create an array of zeros
meshgrid	numpy.meshgrid	Create grid of coordinates in 2 or 3 dimensions
ndgrid	numpy.mgrid or numpy.ogrid	Rectangular grid in N-D space

1.13.6 Advanced functions

Matlab Function	Python Equivalent	Description
permute	numpy.transpose	Rearrange dimensions of N-dimensional array
angle	numpy.angle	Phase angles for elements in complex array
max	numpy.max	Return the maximum element in an array
min	numpy.min	Return the minimum element in an array
reshape	numpy.reshape	Reshape array
mean	numpy.mean	Take mean along specified dimension
size	numpy.size	get the total number of entries in an array
cell2mat	numpy.vstack([numpy.hstack(cell) for cell in cells])	converts data structure from cell to mat; joins multiple arrays of different sizes into single array
repmat	numpy.tile	Repeat copies of an array
unwrap	np.unwrap	Shift the phase of an array so that there are no jumps of more than the desired angle (default pi)

1.13.7 Array Indexing

Matlab Function	Python Equivalent	Description
find	numpy.where	Find all indices of a matrix for which a logical statement is true
isnan	numpy.isnan	checks each array entry to see if it is NaN
isinf	numpy.isinf	checks each array entry to see if it is Inf
ischar	numpy.ischar	checks each array entry to see if it is a character

1.13.8 Advanced functions

Matlab Function	Python Equivalent	Description
fft2	numpy.fft.fft2	2D fast Fourier transform
fftshift	numpy.fft.fftshift	shift zero-frequency component to the center of the spectrum
ifftshift	numpy.fft.ifftshift	inverse fftshift
ifft2	numpy.fft.fifft2	inverse 2d fft
interp2	scipy.interpolate.RectBivariateSpline or scipy.interpolate.interp2	Interpolation for 2-D gridded data in meshgrid format
imshowpair	skimage.measure.structural_similarity	Compare differences between 2 images
imregconfig		Creates configurations to perform intensity-based image registration
imregister		Intensity-based image registration
imregtform	skimage.feature.register_translation or skimage.transform.estimate_transform	Estimate geometric transfomation to align two images
imwarp	skimage.transform.warp	Apply geometric transformation to an image
imref2d		Reference 2d image to xy-coordinates
corr2	scipy.signal.correlate2d	2d correlation coefficient
optimset		Create of edit optimizations options for passing to fminbnd, fminsearch, fzero, or lsqnonneg
lsqcurvefit	scipy.optimize.curve_fit	Solve nonlinear curve-fitting problems
fastica	sklearn.decomposition.FastICA	Fast fixed-point algorithm for independent component analysis and projection pursuit
kmeans	sklearn.cluster.Kmeans	kmeans clustering
fsolve	scipy.optimize.root(func, x0, method=’anderson’)	Root finding. Scipy does not have a trust-region dogleg method that functions exactly like Matlab’s fsolve. The ‘anderson’ method reproduces the results in many cases. Other methods may need to be explored for other problems.

1.13.9 Basic Plotting

Matlab Function	Python Equivalent	Description
figure	matplotlib.pyplot.figure	Create a new figure object
clf	figure.clf	clear figure; shouldn't be needed in Python since each figure will be a unique object
subplot	figure.subplots or figure.add_subplot	1stcreates a set of subplots in the figure, 2ndcreates one subplot and adds it to the figure
plot	figure.plot or axes.plot	Add lineplot to current figure
title	object.title	Title of plot; better to define on object creation if possible
xlabel	axes.xlabel	Label for the x-axis of plot
ylabel	axes.ylabel	Label for the y-axis of plot
imagesc	pyplot.imshow or pyplot.matshow	Scale image data to full range of colormap and display
axis	axes.axis	Axis properties
surf	axes3d.plot_surface or axes3d.plot_trisurf	Plot a 3d surface, need to uses mpl_toolkits.mplot3d and Axes3d; which you use depends on data format
shading		Set during plot creation as argument
view	axes3d.view_init	Change the viewing angle for a 3d plot
colormap	plot.colormap	Set the colormap; better to do so at plot creation if possible
colorbar	figure.add_colorbar(axes)	Add colorbar to selected axes

1.14 API Reference

The pyUSID package.

1.14.1 Submodules

—

pyUSID:

pyUSID.io	pyUSID's I/O module
pyUSID.processing	
pyUSID.viz	

1.14.2 pyUSID.io

pyUSID's I/O module

Submodules

hdf_utils	Created on Tue Nov 3 21:14:25 2015
image	Created on Feb 9, 2016
io_utils	Created on Tue Nov 3 21:14:25 2015

Continued on next page

Table 4 – continued from previous page

numpy_translator	Created on Fri Jan 27 17:58:35 2017
usi_data	Created on Thu Sep 7 21:14:25 2017
translator	Created on Tue Nov 3 15:07:16 2015
write_utils	Created on Thu Sep 7 21:14:25 2017

pyUSID.io.hdf_utils

Created on Tue Nov 3 21:14:25 2015

@author: Suhas Somnath, Chris Smith

Functions

assign_group_index(h5_parent_group, base_name)	Searches the parent h5 group to find the next available index for the group
attempt_reg_ref_build(h5_dset, dim_names[, …])	**param h5_dset** Dataset to which region references need to be added as attributes
check_and_link_ancillary(h5_dset, anc_names)	This function will add references to auxilliary datasets as attributes of an input dataset.
check_for_matching_attrs(h5_obj[, …])	Compares attributes in the given H5 object against those in the provided dictionary and returns True if the parameters match, and False otherwise
check_for_old(h5_base, tool_name[, …])	Check to see if the results of a tool already exist and if they were performed with the same parameters.
check_if_main(h5_main[, verbose])	Checks the input dataset to see if it has all the neccessary features to be considered a Main dataset.
clean_reg_ref(h5_dset, reg_ref_tuple[, verbose])	Makes sure that the provided instructions for a region reference are indeed valid This method has become necessary since h5py allows the writing of region references larger than the maxshape
copy_attributes(source, dest[, skip_refs])	Copy attributes from one h5object to another
copy_main_attributes(h5_main, h5_new)	Copies the units and quantity name from one dataset to another
copy_reg_ref_reduced_dim(h5_source, …)	Copies a region reference from one dataset to another taking into account that a dimension has been lost from source to target
copy_region_refs(h5_source, h5_target)	Check the input dataset for plot groups, copy them if they exist Also make references in the Spectroscopic Values and Indices tables
create_empty_dataset(source_dset, dtype, …)	Creates an empty dataset in the h5 file based on the provided dataset in the same or specified group
create_indexed_group(h5_parent_group, base_name)	Creates a group with an indexed name (eg - 'Measurement_012') under h5_parent_group using the provided base_name as a prefix for the group's name
create_region_reference(h5_main, ref_inds)	Create a region reference in the destination dataset using an iterable of pairs of indices representing the start and end points of a hyperslab block

Continued on next page

Table 5 – continued from previous page

`create_results_group`(h5_main, tool_name)	Creates a h5py.Group object autoindexed and named as ‘DatasetName-ToolName_00x’
`find_dataset`(h5_group, dset_name)	Uses visit() to find all datasets with the desired name
`find_results_groups`(h5_main, tool_name)	Finds a list of all groups containing results of the process of name tool_name being applied to the dataset
`get_all_main`(parent[, verbose])	Simple function to recursively print the contents of an hdf5 group
`get_attr`(h5_object, attr_name)	Returns the attribute from the h5py object
`get_attributes`(h5_object[, attr_names])	Returns attribute associated with some DataSet.
`get_auxiliary_datasets`(h5_object[, …])	Returns auxiliary dataset objects associated with some DataSet through its attributes.
`get_data_descriptor`(h5_dset)	Returns a string of the form ‘quantity (unit)’
`get_dimensionality`(ds_index[, index_sort])	Get the size of each index dimension in a specified sort order
`get_formatted_labels`(h5_dset)	Takes any dataset which has the labels and units attributes and returns a list of strings formatted as ‘label k (unit k)’
`get_group_refs`(group_name, h5_refs)	Given a list of H5 references and a group name, this method returns H5 Datagroup object corresponding to the names.
`get_h5_obj_refs`(obj_names, h5_refs)	Given a list of H5 references and a list of names, this method returns H5 objects corresponding to the names
`get_indices_for_region_ref`(h5_main, ref[, …])	Given an hdf5 region reference and the dataset it refers to, return an array of indices within that dataset that correspond to the reference.
`get_region`(h5_dset, reg_ref_name)	Gets the region in a dataset specified by a region reference
`get_sort_order`(ds_spec)	Find how quickly the spectroscopic values are changing in each row and the order of rows from fastest changing to slowest.
`get_source_dataset`(h5_group)	Find the name of the source dataset used to create the input *h5_group*
`get_unit_values`(h5_inds, h5_vals[, …])	Gets the unit arrays of values that describe the spectroscopic dimensions
`is_editable_h5`(h5_obj)	Returns True if the file containing the provided h5 object is in w or r+ modes
`link_as_main`(h5_main, h5_pos_inds, …[, …])	Links the object references to the four position and spectrosocpic datasets as attributes of *h5_main*
`link_h5_obj_as_alias`(h5_main, h5_ancillary, …)	Creates Dataset attributes that contain references to other Dataset Objects.
`link_h5_objects_as_attrs`(src, h5_objects)	Creates Dataset attributes that contain references to other Dataset Objects.
`print_tree`(parent[, rel_paths, main_dsets_only])	Simple function to recursively print the contents of an hdf5 group
`reshape_from_n_dims`(data_n_dim[, h5_pos, …])	Reshape the input 2D matrix to be N-dimensions based on the position and spectroscopic datasets.
`reshape_to_n_dims`(h5_main[, h5_pos, …])	Reshape the input 2D matrix to be N-dimensions based on the position and spectroscopic datasets.
`simple_region_ref_copy`(h5_source, h5_target, key)	Copies a region reference from one dataset to another without alteration

Continued on next page

Table 5 – continued from previous page

write_book_keeping_attrs(h5_obj)	Writes basic book-keeping and posterity related attributes to groups created in pyUSID such as machine id, pyUSID version, timestamp.
write_ind_val_dsets(h5_parent_group, dimensions)	Creates h5py.Datasets for the position OR spectroscopic indices and values of the data.
write_main_dataset(h5_parent_group, . . . [, . . .])	Writes the provided data as a 'Main' dataset with all appropriate linking.
write_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
write_region_references(h5_dset, reg_ref_dict)	Creates attributes of a h5py.Dataset that refer to regions in the dataset
write_simple_attrs(h5_obj, attrs[, . . .])	Writes attributes to a h5py object

get_attr(*h5_object*, *attr_name*)

Returns the attribute from the h5py object

Parameters

- **h5_object** (*h5py.Dataset, h5py.Group or h5py.File object*) – object whose attribute is desired
- **attr_name** (*str*) – Name of the attribute of interest

Returns **att_val** – value of attribute, in certain cases (byte strings or list of byte strings) reformatted to readily usable forms

Return type object

get_h5_obj_refs(*obj_names*, *h5_refs*)

Given a list of H5 references and a list of names, this method returns H5 objects corresponding to the names

Parameters

- **obj_names** (*string or List of strings*) – names of target h5py objects
- **h5_refs** (*H5 object reference or List of H5 object references*) – list containing the target reference

Returns **found_objects** – Corresponding references

Return type List of HDF5 dataset references

get_indices_for_region_ref(*h5_main*, *ref*, *return_method='slices'*)

Given an hdf5 region reference and the dataset it refers to, return an array of indices within that dataset that correspond to the reference.

Parameters

- **h5_main** (*HDF5 Dataset*) – dataset that the reference can be returned from
- **ref** (*HDF5 Region Reference*) – Region reference object
- **return_method** (*{'slices', 'corners', 'points'}*) – slices : the reference is return as pairs of slices

  corners : the reference is returned as pairs of corners representing the starting and ending indices of each block

  points : the reference is returns as a list of tuples of points

Returns **ref_inds** – array of indices in the source dataset that ref accesses

Return type Numpy Array

get_dimensionality(*ds_index*, *index_sort=None*)

Get the size of each index dimension in a specified sort order

Parameters

- **ds_index** (*2D HDF5 Dataset or numpy array*) – Row matrix of indices
- **index_sort** (*Iterable of unsigned integers (Optional)*) – Sort that can be applied to dimensionality. For example - Order of rows sorted from fastest to slowest

Returns **sorted_dims** – Dimensionality of each row in ds_index. If index_sort is supplied, it will be in the sorted order

Return type list of unsigned integers

get_sort_order(*ds_spec*)

Find how quickly the spectroscopic values are changing in each row and the order of rows from fastest changing to slowest.

Parameters **ds_spec** (*2D HDF5 dataset or numpy array*) – Rows of indices to be sorted from fastest changing to slowest

Returns **change_sort** – Order of rows sorted from fastest changing to slowest

Return type List of unsigned integers

get_auxiliary_datasets(*h5_object*, *aux_dset_name=None*)

Returns auxiliary dataset objects associated with some DataSet through its attributes. Note - region references will be ignored.

Parameters

- **h5_object** (*h5py.Dataset, h5py.Group or h5py.File object*) – Dataset object reference.
- **aux_dset_name** (*str or list of strings, optional, default = all (DataSet.attrs)*) – Name of auxiliary Dataset objects to return.

Returns

Return type list of h5py.Reference of auxiliary dataset objects.

get_attributes(*h5_object*, *attr_names=None*)

Returns attribute associated with some DataSet.

Parameters

- **h5_object** (*h5py.Dataset*) – Dataset object reference.
- **attr_names** (*string or list of strings, optional, default = all (DataSet.attrs)*) – Name of attribute object to return.

Returns

Return type Dictionary containing (name,value) pairs of attributes

get_group_refs(*group_name*, *h5_refs*)

Given a list of H5 references and a group name, this method returns H5 Datagroup object corresponding to the names. This function is especially useful when the suffix of the written group is unknown (due to the autoindexing in HDFwriter)

Parameters

- **group_name** (*unicode / string*) – Name of the datagroup. If the index suffix is left out, all groups matching the basename will be returned Example - provide 'SourceDataset_ProcessName' if a specific group is required, provide - 'SourceDataset_ProcessName_017'
- **h5_refs** (*list*) – List of h5 object references

Returns group_list – A list of h5py.Group objects whose name matched with the provided group_name

Return type list

check_if_main(*h5_main*, *verbose=False*)

Checks the input dataset to see if it has all the neccessary features to be considered a Main dataset. This means it is 2D and has the following attributes Position_Indices Position_Values Spectroscopic_Indices Spectroscopic_Values

In addition the shapes of the ancillary matricies should match with that of h5_main

In addition, it should have the 'quantity' and 'units' attributes

Parameters

- **h5_main** (*HDF5 Dataset*) – Dataset of interest
- **verbose** (*Boolean (Optional. Default = False)*) – Whether or not to print statements

Returns success – True if all tests pass

Return type Boolean

check_and_link_ancillary(*h5_dset*, *anc_names*, *h5_main=None*, *anc_refs=None*)

This function will add references to auxilliary datasets as attributes of an input dataset. If the entries in anc_refs are valid references, they will be added as attributes with the name taken from the corresponding entry in anc_names. If an entry in anc_refs is not a valid reference, the function will attempt to get the attribute with the same name from the h5_main dataset

Parameters

- **h5_dset** (*HDF5 Dataset*) – dataset to which the attributes will be written
- **anc_names** (*list of str*) – the attribute names to be used
- **h5_main** (*HDF5 Dataset, optional*) – dataset from which attributes will be copied if *anc_refs* is None
- **anc_refs** (*list of HDF5 Object References, optional*) – references that correspond to the strings in *anc_names*

Returns

Return type None

Notes

Either *h5_main* or *anc_refs* MUST be provided and *anc_refs* has the higher priority if both are present.

copy_region_refs(*h5_source*, *h5_target*)

Check the input dataset for plot groups, copy them if they exist Also make references in the Spectroscopic Values and Indices tables

Parameters

- **h5_source** (*HDF5 Dataset*) – source dataset to copy references from
- **h5_target** (*HDF5 Dataset*) – target dataset the references from h5_source are copied to

get_all_main (*parent*, *verbose=False*)

Simple function to recursively print the contents of an hdf5 group

Parameters

- **parent** (*h5py.Group*) – HDF5 Group to search within
- **verbose** (*bool*) – If true, extra print statements are enabled

Returns **main_list** – The datasets found in the file that meet the 'Main Data' criteria.

Return type list of h5py.Dataset

get_unit_values (*h5_inds*, *h5_vals*, *dim_names=None*, *all_dim_names=None*, *verbose=False*)

Gets the unit arrays of values that describe the spectroscopic dimensions

Parameters

- **h5_inds** (*h5py.Dataset or numpy.ndarray*) – Spectroscopic or Position Indices dataset
- **h5_vals** (*h5py.Dataset or numpy.ndarray*) – Spectroscopic or Position Values dataset
- **dim_names** (*str, or list of str, Optional*) – Names of the dimensions of interest. Default = all
- **all_dim_names** (*list of str, Optional*) – Names of all the dimensions in these datasets. Use this if supplying numpy arrays instead of h5py.Dataset objects for h5_inds, h5_vals since there is no other way of getting the dimension names.
- **verbose** (*bool, optional*) – Whether or not to print debugging statements. Default - off
- **- this function can be extended / modified for ancillary position dimensions as well** (*Note*) –

Returns **unit_values** – Dictionary containing the unit array for each dimension. The name of the dimensions are the keys.

Return type dict

get_data_descriptor (*h5_dset*)

Returns a string of the form 'quantity (unit)'

Parameters **h5_dset** (*h5py.Dataset object*) – A USID 'main' dataset

Returns **descriptor** – string of the form 'quantity (unit)'

Return type String

copy_attributes (*source*, *dest*, *skip_refs=True*)

Copy attributes from one h5object to another

Parameters

- **source** (*h5py.Dataset, h5py.Group, or h5py.File object*) – Object containing the desired attributes
- **dest** (*h5py.Dataset, h5py.Group, or h5py.File object*) – Object to which the attributes need to be copied to

- **skip_refs** (*bool, optional. default = True*) – Whether or not the references (dataset and region) should be skipped

reshape_to_n_dims (*h5_main*, *h5_pos=None*, *h5_spec=None*, *get_labels=False*, *verbose=False*, *sort_dims=False*)

Reshape the input 2D matrix to be N-dimensions based on the position and spectroscopic datasets.

Parameters

- **h5_main** (*HDF5 Dataset*) – 2D data to be reshaped
- **h5_pos** (*HDF5 Dataset, optional*) – Position indices corresponding to rows in *h5_main*
- **h5_spec** (*HDF5 Dataset, optional*) – Spectroscopic indices corresponding to columns in *h5_main*
- **get_labels** (*bool, optional*) – Whether or not to return the dimension labels. Default False
- **verbose** (*bool, optional*) – Whether or not to print debugging statements
- **sort_dims** (*bool*) – If True, the data is sorted so that the dimensions are in order from fastest to slowest If False, the data is kept in the original order If *get_labels* is also True, the labels are sorted as well.

Returns

- **ds_Nd** (*N-D numpy array*) – N dimensional numpy array arranged as [positions slowest to fastest, spectroscopic slowest to fastest]
- **success** (*boolean or string*) – True if full reshape was successful

  ”Positions” if it was only possible to reshape by the position dimensions

  False if no reshape was possible
- **ds_labels** (*list of str*) – List of the labels of each dimension of *ds_Nd*

Notes

If either *h5_pos* or *h5_spec* are not provided, the function will first attempt to find them as attributes of *h5_main*. If that fails, it will generate dummy values for them.

link_h5_objects_as_attrs (*src*, *h5_objects*)

Creates Dataset attributes that contain references to other Dataset Objects.

Parameters

- **src** (*Reference to h5.object*) – Reference to the the object to which attributes will be added
- **h5_objects** (*list of references to h5.objects*) – objects whose references that can be accessed from src.attrs

Returns

Return type None

link_h5_obj_as_alias (*h5_main*, *h5_ancillary*, *alias_name*)

Creates Dataset attributes that contain references to other Dataset Objects. This function is useful when the reference attribute must have a reserved name. Such as linking ‘SHO_Indices’ as ‘Spectroscopic_Indices’

Parameters

- **h5_main** (*h5py.Dataset*) – Reference to the the object to which attributes will be added
- **h5_ancillary** (*h5py.Dataset*) – object whose reference that can be accessed from src.attrs
- **alias_name** (*String*) – Alias / alternate name for trg

find_results_groups(*h5_main*, *tool_name*)

Finds a list of all groups containing results of the process of name tool_name being applied to the dataset

Parameters

- **h5_main** (*h5 dataset reference*) – Reference to the target dataset to which the tool was applied
- **tool_name** (*String / unicode*) – Name of the tool applied to the target dataset

Returns **groups** – groups whose name contains the tool name and the dataset name

Return type list of references to h5 group objects

get_formatted_labels(*h5_dset*)

Takes any dataset which has the labels and units attributes and returns a list of strings formatted as 'label k (unit k)'

Parameters **h5_dset** (*h5py.Dataset object*) – dataset which has labels and units attributes

Returns **labels** – list of strings formatted as 'label k (unit k)'

Return type list

reshape_from_n_dims(*data_n_dim*, *h5_pos=None*, *h5_spec=None*, *verbose=False*)

Reshape the input 2D matrix to be N-dimensions based on the position and spectroscopic datasets.

Parameters

- **data_n_dim** (*numpy.array*) – N dimensional numpy array arranged as [positions dimensions…, spectroscopic dimensions] If h5_pos and h5_spec are not provided, this function will have to assume that the dimensions are arranged as [positions slowest to fastest, spectroscopic slowest to fastest]. This restriction is removed if h5_pos and h5_spec are provided
- **h5_pos** (*HDF5 Dataset, numpy.array*) – Position indices corresponding to rows in the final 2d array The dimensions should be arranged in terms of rate of change corresponding to data_n_dim. In other words if data_n_dim had two position dimensions arranged as [pos_fast, pos_slow, spec_dim_1….], h5_pos should be arranged as [pos_fast, pos_slow]
- **h5_spec** (*HDF5 Dataset, numpy. array*) – Spectroscopic indices corresponding to columns in the final 2d array The dimensions should be arranged in terms of rate of change corresponding to data_n_dim. In other words if data_n_dim had two spectral dimensions arranged as [pos_dim_1,…, spec_fast, spec_slow], h5_spec should be arranged as [pos_slow, pos_fast]
- **verbose** (*bool, optional. Default = False*) – Whether or not to print log statements

Returns

- **ds_2d** (*numpy.array*) – 2 dimensional numpy array arranged as [positions, spectroscopic]
- **success** (*boolean or string*) – True if full reshape was successful

  "Positions" if it was only possible to reshape by the position dimensions

False if no reshape was possible

Notes

If either *h5_pos* or *h5_spec* are not provided, the function will assume the first dimension is position and the remaining are spectroscopic already in order from fastest to slowest.

find_dataset(*h5_group*, *dset_name*)

Uses visit() to find all datasets with the desired name

Parameters

- **h5_group** (*h5py.Group*) – Group to search within for the Dataset
- **dset_name** (*str*) – Name of the dataset to search for

Returns **datasets** – List of [Name, object] pairs corresponding to datasets that match *ds_name*.

Return type list

print_tree(*parent*, *rel_paths=False*, *main_dsets_only=False*)

Simple function to recursively print the contents of an hdf5 group

Parameters

- **parent** (*h5py.Group*) – HDF5 tree to print
- **rel_paths** (*(Optional) bool. Default = False*) – True - prints the relative paths for all elements. False - prints a tree-like structure with only the element names
- **main_dsets_only** (*bool, optional. default=False*) – True - prints only groups and Main datasets False - prints all dataset and group objects

Returns

Return type None

copy_main_attributes(*h5_main*, *h5_new*)

Copies the units and quantity name from one dataset to another

Parameters

- **h5_main** (*h5py.Dataset*) – Dataset containing the target attributes
- **h5_new** (*h5py.Dataset*) – Dataset to which the target attributes are to be copied

create_empty_dataset(*source_dset*, *dtype*, *dset_name*, *h5_group=None*, *new_attrs=None*, *skip_refs=False*)

Creates an empty dataset in the h5 file based on the provided dataset in the same or specified group

Parameters

- **source_dset** (*h5py.Dataset object*) – Source object that provides information on the group and shape of the dataset
- **dtype** (*dtype*) – Data type of the fit / guess datasets
- **dset_name** (*String / Unicode*) – Name of the dataset
- **h5_group** (*h5py.Group object, optional. Default = None*) – Group within which this dataset will be created
- **new_attrs** (*dictionary (Optional)*) – Any new attributes that need to be written to the dataset

- **skip_refs** (*boolean, optional*) – Should ObjectReferences and RegionReferences be skipped when copying attributes from the *source_dset*

Returns **h5_new_dset** – Newly created dataset

Return type h5py.Dataset object

check_for_old(*h5_base*, *tool_name*, *new_parms=None*, *target_dset=None*, *verbose=False*)

Check to see if the results of a tool already exist and if they were performed with the same parameters.

Parameters

- **h5_base** (*h5py.Dataset object*) – Dataset on which the tool is being applied to
- **tool_name** (*str*) – process or analysis name
- **new_parms** (*dict, optional*) – Parameters with which this tool will be performed.
- **target_dset** (*str, optional, default = None*) – Name of the dataset whose attributes will be compared against new_parms. Default - checking against the group
- **verbose** (*bool, optional, default = False*) – Whether or not to print debugging statements

Returns **group** – List of all groups with parameters matching those in *new_parms*

Return type list

get_source_dataset(*h5_group*)

Find the name of the source dataset used to create the input *h5_group*

Parameters **h5_group** (*h5py.Datagroup*) – Child group whose source dataset will be returned

Returns **h5_source** – Main dataset from which this group was generated

Return type Pycrodataset object

link_as_main(*h5_main*, *h5_pos_inds*, *h5_pos_vals*, *h5_spec_inds*, *h5_spec_vals*, *anc_dsets=None*)

Links the object references to the four position and spectrosocpic datasets as attributes of *h5_main*

Parameters

- **h5_main** (*h5py.Dataset*) – 2D Dataset which will have the references added as attributes
- **h5_pos_inds** (*h5py.Dataset*) – Dataset that will be linked with the name 'Position_Indices'
- **h5_pos_vals** (*h5py.Dataset*) – Dataset that will be linked with the name 'Position_Values'
- **h5_spec_inds** (*h5py.Dataset*) – Dataset that will be linked with the name 'Spectroscopic_Indices'
- **h5_spec_vals** (*h5py.Dataset*) – Dataset that will be linked with the name 'Spectroscopic_Values'
- **anc_dsets** (*(Optional) list of h5py.Dataset objects*) – Datasets that will be linked with their own names

copy_reg_ref_reduced_dim(*h5_source*, *h5_target*, *h5_source_inds*, *h5_target_inds*, *key*)

Copies a region reference from one dataset to another taking into account that a dimension has been lost from source to target

Parameters

- **h5_source** (*HDF5 Dataset*) – source dataset for region reference copy

- **h5_target** (*HDF5 Dataset*) – target dataset for region reference copy
- **h5_source_inds** (*HDF5 Dataset*) – indices of each dimension of the h5_source dataset
- **h5_target_inds** (*HDF5 Dataset*) – indices of each dimension of the h5_target dataset
- **key** (*String*) – Name of attribute in h5_source that contains the Region Reference to copy

Returns **ref_inds** – Array containing pairs of points that define the corners of each hyperslab in the region reference

Return type Nx2x2 array of unsigned integers

simple_region_ref_copy(*h5_source*, *h5_target*, *key*)

Copies a region reference from one dataset to another without alteration

Parameters

- **h5_source** (*HDF5 Dataset*) – source dataset for region reference copy
- **h5_target** (*HDF5 Dataset*) – target dataset for region reference copy
- **key** (*String*) – Name of attribute in h5_source that contains the Region Reference to copy

Returns **ref_inds** – Array containing pairs of points that define the corners of each hyperslab in the region reference

Return type Nx2x2 array of unsigned integers

write_book_keeping_attrs(*h5_obj*)

Writes basic book-keeping and posterity related attributes to groups created in pyUSID such as machine id, pyUSID version, timestamp.

Parameters **h5_obj** (*h5py.Object*) – Object to which basic book-keeping attributes need to be written

is_editable_h5(*h5_obj*)

Returns True if the file containing the provided h5 object is in w or r+ modes

Parameters **h5_obj** (*h5py.File, h5py.Group, or h5py.Dataset object*) – h5py object

Returns **mode** – True if the file containing the provided h5 object is in w or r+ modes

Return type bool

write_ind_val_dsets(*h5_parent_group*, *dimensions*, *is_spectral=True*, *verbose=False*, *base_name=None*)

Creates h5py.Datasets 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

- **h5_parent_group** (*h5py.Group or h5py.File*) – Group under which the indices and values datasets will be created
- **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

- **h5_spec_inds** (*h5py.Dataset*) – Dataset containing the position indices
- **h5_spec_vals** (*h5py.Dataset*) – Dataset containing the value at each position

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.

write_reduced_spec_dsets(*h5_parent_group, h5_spec_inds, h5_spec_vals, dim_name, 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
- **dim_name** (*str / unicode*) – Name of the dimension to remove
- **basename** (*str / unicode, Optional*) – String to which '_Indices' and '_Values' will be appended to get the names of the new datasets

Returns

- **h5_inds** (*h5py.Dataset*) – Reduced Spectroscopic indices dataset
- **h5_vals** (*h5py.Dataset*) – Reduces Spectroscopic values dataset

write_simple_attrs(*h5_obj, attrs, obj_type='', verbose=False*)

Writes attributes to a h5py object

Parameters

- **h5_obj** (*h5py.File, h5py.Group, or h5py.Dataset object*) – h5py object to which the attributes will be written to
- **attrs** (*dict*) – Dictionary containing the attributes as key-value pairs
- **obj_type** (*str / unicode, optional. Default = ''*) – type of h5py.obj. Examples include 'group', 'file', 'dataset
- **verbose** (*bool, optional. Default=False*) – Whether or not to print debugging statements

write_main_dataset(*h5_parent_group*, *main_data*, *main_data_name*, *quantity*, *units*, *pos_dims*, *spec_dims*, *main_dset_attrs=None*, *h5_pos_inds=None*, *h5_pos_vals=None*, *h5_spec_inds=None*, *h5_spec_vals=None*, *aux_spec_prefix='Spectroscopic_'*, *aux_pos_prefix='Position_'*, *verbose=False*, ***kwargs*)

Writes the provided data as a 'Main' dataset with all appropriate linking. By default, the instructions for generating the ancillary datasets should be specified using the pos_dims and spec_dims arguments as dictionary objects. Alternatively, if both the indices and values datasets are already available for either/or the positions / spectroscopic, they can be specified using the keyword arguments. In this case, fresh datasets will not be generated.

Parameters

- **h5_parent_group** (*h5py.Group*) – Parent group under which the datasets will be created
- **main_data** (*np.ndarray or list / tuple*) – 2D matrix formatted as [position, spectral] or a list / tuple with the shape for an empty dataset. If creating an empty dataset - the dtype must be specified via a kwarg.
- **main_data_name** (*String / Unicode*) – Name to give to the main dataset. This cannot contain the '-' character.
- **quantity** (*String / Unicode*) – Name of the physical quantity stored in the dataset. Example - 'Current'
- **units** (*String / Unicode*) – Name of units for the quantity stored in the dataset. Example - 'A' for amperes
- **pos_dims** (*Dimension or array-like of Dimension objects*) – Sequence of Dimension objects that provides all necessary instructions for constructing the indices and values datasets Object specifying the instructions necessary for building the Position indices and values datasets
- **spec_dims** (*Dimension or array-like of Dimension objects*) – Sequence of Dimension objects that provides all necessary instructions for constructing the indices and values datasets Object specifying the instructions necessary for building the Spectroscopic indices and values datasets
- **main_dset_attrs** (*dictionary, Optional*) – Dictionary of parameters that will be written to the main dataset. Do NOT include region references here.
- **h5_pos_inds** (*h5py.Dataset, Optional*) – Dataset that will be linked with the name 'Position_Indices'
- **h5_pos_vals** (*h5py.Dataset, Optional*) – Dataset that will be linked with the name 'Position_Values'
- **h5_spec_inds** (*h5py.Dataset, Optional*) – Dataset that will be linked with the name 'Spectroscopic_Indices'
- **h5_spec_vals** (*h5py.Dataset, Optional*) – Dataset that will be linked with the name 'Spectroscopic_Values'
- **aux_spec_prefix** (*str / unicode, Optional*) – Default prefix for Spectroscopic datasets. Default = '**Spectroscopic_**'
- **aux_pos_prefix** (*str / unicode, Optional*) – Default prefix for Position datasets. Default = '**Position_**'
- **verbose** (*bool, Optional, default=False*) – If set to true - prints debugging logs

- **will be passed onto the creation of the dataset. Please pass chunking, compression, dtype, and other** (*kwargs*) –
- **this way** (*arguments*) –

Returns **h5_main** – Reference to the main dataset

Return type *USIDataset*

attempt_reg_ref_build(*h5_dset*, *dim_names*, *verbose=False*)

Parameters

- **h5_dset** (*h5.Dataset instance*) – Dataset to which region references need to be added as attributes
- **dim_names** (*list or tuple*) – List of the names of the region references (typically names of dimensions)
- **verbose** (*bool, optional. Default=False*) – Whether or not to print debugging statements

Returns **labels_dict** – The slicing information must be formatted using tuples of slice objects. For example {'region_1':(slice(None, None), slice (0,1))}

Return type dict

write_region_references(*h5_dset*, *reg_ref_dict*, *add_labels_attr=True*, *verbose=False*)

Creates attributes of a h5py.Dataset that refer to regions in the dataset

Parameters

- **h5_dset** (*h5.Dataset instance*) – Dataset to which region references will be added as attributes
- **reg_ref_dict** (*dict*) – The slicing information must be formatted using tuples of slice objects. For example {'region_1':(slice(None, None), slice (0,1))}
- **add_labels_attr** (*bool, optional, default = True*) – Whether or not to write an attribute named 'labels' with the
- **verbose** (*Boolean (Optional. Default = False)*) – Whether or not to print status messages

assign_group_index(*h5_parent_group*, *base_name*, *verbose=False*)

Searches the parent h5 group to find the next available index for the group

Parameters

- **h5_parent_group** (*h5py.Group object*) – Parent group under which the new group object will be created
- **base_name** (*str / unicode*) – Base name of the new group without index
- **verbose** (*bool, optional. Default=False*) – Whether or not to print debugging statements

Returns **base_name** – Base name of the new group with the next available index as a suffix

Return type str / unicode

clean_reg_ref(*h5_dset*, *reg_ref_tuple*, *verbose=False*)

Makes sure that the provided instructions for a region reference are indeed valid This method has become necessary since h5py allows the writing of region references larger than the maxshape

Parameters

- **h5_dset** (*h5.Dataset instance*) – Dataset to which region references will be added as attributes
- **reg_ref_tuple** (*list / tuple*) – The slicing information formatted using tuples of slice objects.
- **verbose** (*Boolean (Optional. Default = False)*) – Whether or not to print status messages

Returns **is_valid** – Whether or not this

Return type bool

create_results_group(*h5_main*, *tool_name*)

Creates a h5py.Group object autoindexed and named as 'DatasetName-ToolName_00x'

Parameters

- **h5_main** (*h5py.Dataset object*) – Reference to the dataset based on which the process / analysis is being performed
- **tool_name** (*string / unicode*) – Name of the Process / Analysis applied to h5_main

Returns **h5_group** – Results group which can now house the results datasets

Return type h5py.Group object

create_indexed_group(*h5_parent_group*, *base_name*)

Creates a group with an indexed name (eg - 'Measurement_012') under h5_parent_group using the provided base_name as a prefix for the group's name

Parameters

- **h5_parent_group** (*h5py.Group or h5py.File object*) – File or group within which the new group will be created
- **base_name** (*str / unicode*) – Prefix for the group name. This need not end with a '_'. It will be added automatically

pyUSID.io.image

Created on Feb 9, 2016

@author: Chris Smith

Functions

read_image(image_path, *args, **kwargs)	Read the image file at *image_path* into a numpy array

Classes

ImageTranslator(*args, **kwargs)	Translates data from a set of image files to an HDF5 file

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

Translates data from a set of image files to an HDF5 file

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

read_image(*image_path*, **args*, ***kwargs*)

Read the image file at *image_path* into a numpy array

Parameters **image_path** (*str*) – Path to the image file

Returns

- **image** (*numpy.ndarray*) – Array containing the image from the file *image_path*
- **image_parms** (*dict*) – Dictionary containing image parameters. If image type does not have parameters then an empty dictionary is returned.

pyUSID.io.io_utils

Created on Tue Nov 3 21:14:25 2015

@author: Suhas Somnath, Chris Smith

Functions

check_ssh()	Checks whether or not the python kernel is running locally (False) or remotely (True)
file_dialog([file_filter, caption])	Presents a File dialog used for selecting the .mat file and returns the absolute filepath of the selecte file
format_quantity(value, unit_names, factors)	Formats the provided quantity such as time or size to appropriate strings
format_size(size_in_bytes[, decimals])	Formats the provided size in bytes to kB, MB, GB, TB etc.

Continued on next page

Table 8 – continued from previous page

format_time(time_in_seconds[, decimals])	Formats the provided time in seconds to seconds, minutes, or hours
formatted_str_to_number(str_val, . . . [, . . .])	Takes a formatted string like '4.32 MHz' to 4.32 E+6
get_available_memory()	Returns the available memory
get_time_stamp()	Teturns the current date and time as a string formatted as: Year_Month_Dat-Hour_Minute_Second
recommend_cpu_cores(num_jobs[, . . .])	Decides the number of cores to use for parallel computing

get_available_memory()

Returns the available memory

Chris Smith – csmith55@utk.edu

Returns **mem** – Memory in bytes

Return type unsigned int

get_time_stamp()

Teturns the current date and time as a string formatted as: Year_Month_Dat-Hour_Minute_Second

Returns

Return type String

recommend_cpu_cores(*num_jobs*, *requested_cores=None*, *lengthy_computation=False*, *min_free_cores=None*, *verbose=False*)

Decides the number of cores to use for parallel computing

Parameters

- **num_jobs** (*unsigned int*) – Number of times a parallel operation needs to be performed
- **requested_cores** (*unsigned int (Optional. Default = None)*) – Number of logical cores to use for computation
- **lengthy_computation** (*Boolean (Optional. Default = False)*) – Whether or not each computation takes a long time. If each computation is quick, it may not make sense to take a hit in terms of starting and using a larger number of cores, so use fewer cores instead. Eg- BE SHO fitting is fast (<1 sec) so set this value to False, Eg- Bayesian Inference is very slow (~ 10-20 sec)so set this to True
- **min_free_cores** (*uint (Optional, default = 1 if number of logical cores < 5 and 2 otherwise)*) – Number of CPU cores that should not be used)
- **verbose** (*Boolean (Optional. Default = False)*) – Whether or not to print statements that aid in debugging

Returns **requested_cores** – Number of logical cores to use for computation

Return type unsigned int

file_dialog(*file_filter='H5 file (*.h5)'*, *caption='Select File'*)

Presents a File dialog used for selecting the .mat file and returns the absolute filepath of the selecte file

Parameters

- **file_filter** (*String or list of strings*) – file extensions to look for
- **caption** (*(Optional) String*) – Title for the file browser window

Returns **file_path** – Absolute path of the chosen file

Return type String

format_quantity(*value*, *unit_names*, *factors*, *decimals=2*)
Formats the provided quantity such as time or size to appropriate strings

Parameters

- **value** (*number*) – value in some base units. For example - time in seconds
- **unit_names** (*array-like*) – List of names of units for each scale of the value
- **factors** (*array-like*) – List of scaling factors for each scale of the value
- **decimals** (*uint, optional. default = 2*) – Number of decimal places to which the value needs to be formatted

Returns String with value formatted correctly

Return type str

format_time(*time_in_seconds*, *decimals=2*)
Formats the provided time in seconds to seconds, minutes, or hours

Parameters

- **time_in_seconds** (*number*) – Time in seconds
- **decimals** (*uint, optional. default = 2*) – Number of decimal places to which the time needs to be formatted

Returns String with time formatted correctly

Return type str

format_size(*size_in_bytes*, *decimals=2*)
Formats the provided size in bytes to kB, MB, GB, TB etc.

Parameters

- **size_in_bytes** (*number*) – size in bytes
- **decimals** (*uint, optional. default = 2*) – Number of decimal places to which the size needs to be formatted

Returns String with size formatted correctly

Return type str

pyUSID.io.numpy_translator

Created on Fri Jan 27 17:58:35 2017

@author: Suhas Somnath

Classes

NumpyTranslator([max_mem_mb])	Writes a numpy array to .h5

class NumpyTranslator(*max_mem_mb=1024*, **args*, ***kwargs*)
Writes a numpy array 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(*h5_path*, *data_name*, *raw_data*, *quantity*, *units*, *pos_dims*, *spec_dims*, *translator_name='NumpyTranslator'*, *parm_dict=None*, *extra_dsets=None*, ***kwargs*)

Writes the provided datasets and parameters to an h5 file

Parameters

- **h5_path** (*String / Unicode*) – Absolute path of the h5 file to be written
- **data_name** (*String / Unicode*) – Name of the scientific data type. Example - 'SEM'
- **raw_data** (*np.ndarray*) – 2D matrix formatted as [position, spectral]
- **quantity** (*String / Unicode*) – Name of the physical quantity stored in the dataset. Example - 'Current'
- **units** (*String / Unicode*) – Name of units for the quantity stored in the dataset. Example - 'A' for amperes
- **pos_dims** (*Dimension or array-like of Dimension objects*) – Sequence of Dimension objects that provides all necessary instructions for constructing the indices and values datasets Object specifying the instructions necessary for building the Position indices and values datasets
- **spec_dims** (*Dimension or array-like of Dimension objects*) – Sequence of Dimension objects that provides all necessary instructions for constructing the indices and values datasets Object specifying the instructions necessary for building the Spectroscopic indices and values datasets
- **translator_name** (*String / unicode, Optional*) – Name of the translator. Example - 'HitachiSEMTranslator'
- **parm_dict** (*dictionary (Optional)*) – Dictionary of parameters that will be written under the group 'Measurement_000'
- **extra_dsets** (*dictionary (Optional)*) – Dictionary whose values will be written into individual HDF5 datasets and whose corresponding keys provide the names of the datasets. You are recommended to limit these to simple and small datasets.
- **kwargs** (*will be passed onto hdf_utils.write_main_dset() which will in turn will be passed onto the creation of*) – the dataset. Please pass chunking, compression, dtype, and other arguments this way

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

Return type String / unicode

pyUSID.io.usi_data

Created on Thu Sep 7 21:14:25 2017

@author: Suhas Somnath, Chris Smith

Functions

write_dset_to_txt(self[, output_file])	Output a USIDataset in csv format

Classes

USIDataset(h5_ref[, sort_dims])	New data object that extends the h5py.Dataset.

class USIDataset(*h5_ref*, *sort_dims=False*)

New data object that extends the h5py.Dataset.

Parameters

- **h5_ref** (*hdf5.Dataset*) – The base dataset to be extended
- **sort_dims** (*bool*) – Should the dimensions be sorted internally from fastest changing to slowest.

self.**get_current_sorting**()

self.**toggle_sorting**()

self.**get_pos_values**()

self.**get_spec_values**()

self.**get_n_dim_form**()

self.**slice**()

self.**h5_spec_vals**

h5py.Dataset – Associated Spectroscopic Values dataset

self.**h5_spec_inds**

h5py.Dataset – Associated Spectroscopic Indices dataset

self.**h5_pos_vals**

h5py.Dataset – Associated Position Values dataset

self.**h5_pos_inds**

h5py.Dataset – Associated Position Indices dataset

self.**pos_dim_labels**

list of str – The labels for the position dimensions.

self.**spec_dim_labels**

list of str – The labels for the spectroscopic dimensions.

self.**n_dim_labels**

list of str – The labels for the n-dimensional dataset.

self.**pos_dim_sizes**

list of int – A list of the sizes of each position dimension.

self.**spec_dim_sizes**

list of int – A list of the sizes of each spectroscopic dimension.

self.**n_dim_sizes**

list of int – A list of the sizes of each dimension.

Notes

The order of all labels and sizes attributes is determined by the current value of *sort_dims*.

astype(*dtype*)

Get a context manager allowing you to perform reads to a different destination type, e.g.:

```
>>> with dataset.astype('f8'):
...     double_precision = dataset[0:100:2]
```

attrs

Attributes attached to this object

chunks

Dataset chunks (or None)

compression

Compression strategy (or None)

compression_opts

Compression setting. Int(0-9) for gzip, 2-tuple for szip.

dims

Access dimension scales attached to this dataset.

dtype

Numpy dtype representing the datatype

file

Return a File instance associated with this object

fillvalue

Fill value for this dataset (0 by default)

fletcher32

Fletcher32 filter is present (T/F)

flush

Flush the dataset data and metadata to the file. If the dataset is chunked, raw data chunks are written to the file.

This is part of the SWMR features and only exist when the HDF5 library version >=1.9.178

get_current_sorting()

Prints the current sorting method.

get_n_dim_form(*as_scalar=False*)

Reshapes the dataset to an N-dimensional array

Returns **n_dim_data** – N-dimensional form of the dataset

Return type numpy.ndarray

get_pos_values(*dim_name*)

Extract the values for the specified position dimension

Parameters **dim_name** (*str*) – Name of one of the dimensions in *self.pos_dim_labels*

Returns **dim_values** – Array containing the unit values of the dimension *dim_name*

Return type numpy.ndarray

get_spec_values(*dim_name*)

Extract the values for the specified spectroscopic dimension

Parameters **dim_name** (*str*) – Name of one of the dimensions in *self.spec_dim_labels*

Returns **dim_values** – Array containing the unit values of the dimension *dim_name*

Return type numpy.ndarray

id
Low-level identifier appropriate for this object

len()
The size of the first axis. TypeError if scalar.

Use of this method is preferred to len(dset), as Python's built-in len() cannot handle values greater then 2**32 on 32-bit systems.

maxshape
Shape up to which this dataset can be resized. Axes with value None have no resize limit.

name
Return the full name of this object. None if anonymous.

ndim
Numpy-style attribute giving the number of dimensions

parent
Return the parent group of this object.

This is always equivalent to obj.file[posixpath.dirname(obj.name)]. ValueError if this object is anonymous.

read_direct(*dest*, *source_sel=None*, *dest_sel=None*)
Read data directly from HDF5 into an existing NumPy array.

The destination array must be C-contiguous and writable. Selections must be the output of numpy.s_[<args>].

Broadcasting is supported for simple indexing.

ref
An (opaque) HDF5 reference to this object

refresh
Refresh the dataset metadata by reloading from the file.

This is part of the SWMR features and only exist when the HDF5 library version >=1.9.178

regionref
Create a region reference (Datasets only).

The syntax is regionref[<slices>]. For example, dset.regionref[. . .] creates a region reference in which the whole dataset is selected.

Can also be used to determine the shape of the referenced dataset (via .shape property), or the shape of the selection (via the .selection property).

resize(*size*, *axis=None*)
Resize the dataset, or the specified axis.

The dataset must be stored in chunked format; it can be resized up to the "maximum shape" (keyword maxshape) specified at creation time. The rank of the dataset cannot be changed.

"Size" should be a shape tuple, or if an axis is specified, an integer.

BEWARE: This functions differently than the NumPy resize() method! The data is not "reshuffled" to fit in the new shape; each axis is grown or shrunk independently. The coordinates of existing data are fixed.

scaleoffset
Scale/offset filter settings. For integer data types, this is the number of bits stored, or 0 for auto-detected. For floating point data types, this is the number of decimal places retained. If the scale/offset filter is not in use, this is None.

shape
Numpy-style shape tuple giving dataset dimensions

shuffle
Shuffle filter present (T/F)

size
Numpy-style attribute giving the total dataset size

slice(*slice_dict*, *as_scalar=False*, *verbose=False*)
Slice the dataset based on an input dictionary of 'str': slice pairs. Each string should correspond to a dimension label. The slices can be array-likes or slice objects.

Parameters

- **slice_dict** (*dict*) – Dictionary of array-likes. for any dimension one needs to slice
- **as_scalar** (*bool, optional*) – Should the data be returned as scalar values only.
- **verbose** (*bool, optional*) – Whether or not to print debugging statements

Returns

- **data_slice** (*numpy.ndarray*) – Slice of the dataset. Dataset has been reshaped to N-dimensions if *success* is True, only by Position dimensions if *success* is 'Positions', or not reshape at all if *success* is False.
- **success** (*str or bool*) – Informs the user as to how the data_slice has been shaped.

toggle_sorting()
Toggles between sorting from the fastest changing dimension to the slowest and sorting based on the order of the labels

value
Alias for dataset[()]

visualize(*slice_dict=None*, *verbose=False*, ***kwargs*)
Interactive visualization of this dataset. Only available on jupyter notebooks

Parameters

- **slice_dict** (*dictionary, optional*) – Slicing instructions
- **verbose** (*bool, optional*) – Whether or not to print debugging statements. Default = Off

write_direct(*source*, *source_sel=None*, *dest_sel=None*)
Write data directly to HDF5 from a NumPy array.

The source array must be C-contiguous. Selections must be the output of numpy.s_[<args>].

Broadcasting is supported for simple indexing.

write_dset_to_txt(*self*, *output_file='output.csv'*)
Output a USIDataset in csv format

Parameters

- **self** (*USIDataset*) – the USIDataset that will be exported as a csv
- **output_file** (*str, optional*) – path that the output file should be written to

Returns

- **output_file** (*str*)
- *Author - Daniel Streater*

pyUSID.io.translator

Created on Tue Nov 3 15:07:16 2015

@author: Suhas Somnath

Functions

generate_dummy_main_parms()	Generates a (dummy) dictionary of parameters that will be used at the root level of the h5 file

Classes

Translator([max_mem_mb])	Abstract class that defines the most basic functionality of a data format translator.

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

Abstract class that defines the most basic functionality of a data format translator. A translator converts experimental data from binary / proprietary data formats to a single standardized HDF5 data file

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(*filepath*, **args*, ***kwargs*)

Abstract method. To be implemented by extensions of this class. God I miss Java!

generate_dummy_main_parms()

Generates a (dummy) dictionary of parameters that will be used at the root level of the h5 file

Returns main_parms – Dictionary containing basic descriptors that describe a dataset

Return type dictionary

pyUSID.io.write_utils

Created on Thu Sep 7 21:14:25 2017

@author: Suhas Somnath, Chris Smith

Functions

build_ind_val_matrices(unit_values[, . . .])	Builds indices and values matrices using given unit values for each dimension.
calc_chunks(dimensions, dtype_byte_size[, . . .])	Calculate the chunk size for the HDF5 dataset based on the dimensions and the maximum chunk size in memory
clean_string_att(att_val)	Replaces any unicode objects within lists with their string counterparts to ensure compatibility with python 3.
create_spec_inds_from_vals(ds_spec_val_mat)	Create new Spectroscopic Indices table from the changes in the Spectroscopic Values
get_aux_dset_slicing(dim_names[, last_ind, . . .])	Returns a dictionary of slice objects to help in creating region references in the position or spectroscopic indices and values datasets
make_indices_matrix(num_steps[, is_position])	Makes an ancillary indices matrix given the number of steps in each dimension.

Classes

Dimension(name, units, values)	Simple object that describes a dimension in a dataset by its name, units, and values :param name: Name of the dimension.

clean_string_att(*att_val*)

Replaces any unicode objects within lists with their string counterparts to ensure compatibility with python 3. If the attribute is indeed a list of unicodes, the changes will be made in-place

Parameters **att_val** (*object*) – Attribute object

Returns **att_val** – Attribute object

Return type object

get_aux_dset_slicing(*dim_names*, *last_ind=None*, *is_spectroscopic=False*)

Returns a dictionary of slice objects to help in creating region references in the position or spectroscopic indices and values datasets

Parameters

- **dim_names** (*iterable*) – List of strings denoting the names of the position axes or spectroscopic dimensions arranged in the same order that matches the dimensions in the indices / values dataset
- **last_ind** (*(Optional) unsigned int, default = None*) – Last pixel in the positon or spectroscopic matrix. Useful in experiments where the parameters have changed (eg. BEPS new data format) during the experiment.
- **is_spectroscopic** (*bool, optional. default = True*) – set to True for position datasets and False for spectroscopic datasets

Returns **slice_dict** – Dictionary of tuples containing slice objects corresponding to each position axis.

Return type dictionary

make_indices_matrix(*num_steps*, *is_position=True*)

Makes an ancillary indices matrix given the number of steps in each dimension. In other words, this function builds a matrix whose rows correspond to unique combinations of the multiple dimensions provided.

Parameters

- **num_steps** (*List / numpy array*) – Number of steps in each spatial or spectral dimension Note that the axes must be ordered from fastest varying to slowest varying
- **is_position** (*bool, optional, default = True*) – Whether the returned matrix is meant for position (True) indices (tall and skinny) or spectroscopic (False) indices (short and wide)

Returns **indices_matrix** – arranged as [steps, spatial dimension]

Return type 2D unsigned int numpy array

INDICES_DTYPE
alias of `numpy.uint32`

VALUES_DTYPE
alias of `numpy.float32`

build_ind_val_matrices (*unit_values*, *is_spectral=True*)
Builds indices and values matrices using given unit values for each dimension.

Parameters

- **unit_values** (*list / tuple*) – Sequence of values vectors for each dimension
- **is_spectral** (*bool (optional), default = True*) – If true, returns matrices for spectroscopic datasets, else returns matrices for Position datasets

Returns

- **ind_mat** (*2D numpy array*) – Indices matrix
- **val_mat** (*2D numpy array*) – Values matrix

calc_chunks (*dimensions*, *dtype_byte_size*, *unit_chunks=None*, *max_chunk_mem=10240*)
Calculate the chunk size for the HDF5 dataset based on the dimensions and the maximum chunk size in memory

Parameters

- **dimensions** (*array_like of int*) – Shape of the data to be chunked
- **dtype_byte_size** (*unsigned int*) – Size of an entry in the data in bytes
- **unit_chunks** (*array_like of int, optional*) – Unit size of the chunking in each dimension. Must be the same size as the shape of *ds_main*. Default None, *unit_chunks* is set to 1 in all dimensions
- **max_chunk_mem** (*int, optional*) – Maximum size of the chunk in memory in bytes. Default 10240b or 10kb per h5py recommendations

Returns **chunking** – Calculated maximum size of a chunk in each dimension that is as close to the requested *max_chunk_mem* as posible while having steps based on the input *unit_chunks*.

Return type tuple of int

create_spec_inds_from_vals (*ds_spec_val_mat*)
Create new Spectroscopic Indices table from the changes in the Spectroscopic Values

Parameters **ds_spec_val_mat** (*array-like,*) – Holds the spectroscopic values to be indexed

Returns **ds_spec_inds_mat** – Indices corresponding to the values in ds_spec_val_mat

Return type numpy array of uints the same shape as ds_spec_val_mat

class USIDataset (*h5_ref*, *sort_dims=False*)

New data object that extends the h5py.Dataset.

Parameters

- **h5_ref** (*hdf5.Dataset*) – The base dataset to be extended
- **sort_dims** (*bool*) – Should the dimensions be sorted internally from fastest changing to slowest.

self.**get_current_sorting**()

self.**toggle_sorting**()

self.**get_pos_values**()

self.**get_spec_values**()

self.**get_n_dim_form**()

self.**slice**()

self.**h5_spec_vals**

h5py.Dataset – Associated Spectroscopic Values dataset

self.**h5_spec_inds**

h5py.Dataset – Associated Spectroscopic Indices dataset

self.**h5_pos_vals**

h5py.Dataset – Associated Position Values dataset

self.**h5_pos_inds**

h5py.Dataset – Associated Position Indices dataset

self.**pos_dim_labels**

list of str – The labels for the position dimensions.

self.**spec_dim_labels**

list of str – The labels for the spectroscopic dimensions.

self.**n_dim_labels**

list of str – The labels for the n-dimensional dataset.

self.**pos_dim_sizes**

list of int – A list of the sizes of each position dimension.

self.**spec_dim_sizes**

list of int – A list of the sizes of each spectroscopic dimension.

self.**n_dim_sizes**

list of int – A list of the sizes of each dimension.

Notes

The order of all labels and sizes attributes is determined by the current value of *sort_dims*.

astype (*dtype*)

Get a context manager allowing you to perform reads to a different destination type, e.g.:

```
>>> with dataset.astype('f8'):
...     double_precision = dataset[0:100:2]
```

attrs
Attributes attached to this object

chunks
Dataset chunks (or None)

compression
Compression strategy (or None)

compression_opts
Compression setting. Int(0-9) for gzip, 2-tuple for szip.

dims
Access dimension scales attached to this dataset.

dtype
Numpy dtype representing the datatype

file
Return a File instance associated with this object

fillvalue
Fill value for this dataset (0 by default)

fletcher32
Fletcher32 filter is present (T/F)

flush
Flush the dataset data and metadata to the file. If the dataset is chunked, raw data chunks are written to the file.

This is part of the SWMR features and only exist when the HDF5 library version >=1.9.178

get_current_sorting()
Prints the current sorting method.

get_n_dim_form(*as_scalar=False*)
Reshapes the dataset to an N-dimensional array

Returns **n_dim_data** – N-dimensional form of the dataset

Return type numpy.ndarray

get_pos_values(*dim_name*)
Extract the values for the specified position dimension

Parameters **dim_name** (*str*) – Name of one of the dimensions in *self.pos_dim_labels*

Returns **dim_values** – Array containing the unit values of the dimension *dim_name*

Return type numpy.ndarray

get_spec_values(*dim_name*)
Extract the values for the specified spectroscopic dimension

Parameters **dim_name** (*str*) – Name of one of the dimensions in *self.spec_dim_labels*

Returns **dim_values** – Array containing the unit values of the dimension *dim_name*

Return type numpy.ndarray

id
Low-level identifier appropriate for this object

len()

The size of the first axis. TypeError if scalar.

Use of this method is preferred to len(dset), as Python's built-in len() cannot handle values greater then 2**32 on 32-bit systems.

maxshape

Shape up to which this dataset can be resized. Axes with value None have no resize limit.

name

Return the full name of this object. None if anonymous.

ndim

Numpy-style attribute giving the number of dimensions

parent

Return the parent group of this object.

This is always equivalent to obj.file[posixpath.dirname(obj.name)]. ValueError if this object is anonymous.

read_direct(*dest*, *source_sel=None*, *dest_sel=None*)

Read data directly from HDF5 into an existing NumPy array.

The destination array must be C-contiguous and writable. Selections must be the output of numpy.s_[<args>].

Broadcasting is supported for simple indexing.

ref

An (opaque) HDF5 reference to this object

refresh

Refresh the dataset metadata by reloading from the file.

This is part of the SWMR features and only exist when the HDF5 library version >=1.9.178

regionref

Create a region reference (Datasets only).

The syntax is regionref[<slices>]. For example, dset.regionref[. . .] creates a region reference in which the whole dataset is selected.

Can also be used to determine the shape of the referenced dataset (via .shape property), or the shape of the selection (via the .selection property).

resize(*size*, *axis=None*)

Resize the dataset, or the specified axis.

The dataset must be stored in chunked format; it can be resized up to the "maximum shape" (keyword maxshape) specified at creation time. The rank of the dataset cannot be changed.

"Size" should be a shape tuple, or if an axis is specified, an integer.

BEWARE: This functions differently than the NumPy resize() method! The data is not "reshuffled" to fit in the new shape; each axis is grown or shrunk independently. The coordinates of existing data are fixed.

scaleoffset

Scale/offset filter settings. For integer data types, this is the number of bits stored, or 0 for auto-detected. For floating point data types, this is the number of decimal places retained. If the scale/offset filter is not in use, this is None.

shape

Numpy-style shape tuple giving dataset dimensions

shuffle
Shuffle filter present (T/F)

size
Numpy-style attribute giving the total dataset size

slice (*slice_dict*, *as_scalar=False*, *verbose=False*)
Slice the dataset based on an input dictionary of 'str': slice pairs. Each string should correspond to a dimension label. The slices can be array-likes or slice objects.

Parameters

- **slice_dict** (*dict*) – Dictionary of array-likes. for any dimension one needs to slice
- **as_scalar** (*bool, optional*) – Should the data be returned as scalar values only.
- **verbose** (*bool, optional*) – Whether or not to print debugging statements

Returns

- **data_slice** (*numpy.ndarray*) – Slice of the dataset. Dataset has been reshaped to N-dimensions if *success* is True, only by Position dimensions if *success* is 'Positions', or not reshape at all if *success* is False.
- **success** (*str or bool*) – Informs the user as to how the data_slice has been shaped.

toggle_sorting ()
Toggles between sorting from the fastest changing dimension to the slowest and sorting based on the order of the labels

value
Alias for dataset[()]

visualize (*slice_dict=None*, *verbose=False*, ***kwargs*)
Interactive visualization of this dataset. Only available on jupyter notebooks

Parameters

- **slice_dict** (*dictionary, optional*) – Slicing instructions
- **verbose** (*bool, optional*) – Whether or not to print debugging statements. Default = Off

write_direct (*source*, *source_sel=None*, *dest_sel=None*)
Write data directly to HDF5 from a NumPy array.

The source array must be C-contiguous. Selections must be the output of numpy.s_[<args>].

Broadcasting is supported for simple indexing.

class NumpyTranslator (*max_mem_mb=1024*, **args*, ***kwargs*)
Writes a numpy array 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 (*h5_path*, *data_name*, *raw_data*, *quantity*, *units*, *pos_dims*, *spec_dims*, *translator_name='NumpyTranslator'*, *parm_dict=None*, *extra_dsets=None*, ***kwargs*)
Writes the provided datasets and parameters to an h5 file

Parameters

- **h5_path** (*String / Unicode*) – Absolute path of the h5 file to be written

- **data_name** (*String / Unicode*) – Name of the scientific data type. Example - ‘SEM’
- **raw_data** (*np.ndarray*) – 2D matrix formatted as [position, spectral]
- **quantity** (*String / Unicode*) – Name of the physical quantity stored in the dataset. Example - ‘Current’
- **units** (*String / Unicode*) – Name of units for the quantity stored in the dataset. Example - ‘A’ for amperes
- **pos_dims** (*Dimension or array-like of Dimension objects*) – Sequence of Dimension objects that provides all necessary instructions for constructing the indices and values datasets Object specifying the instructions necessary for building the Position indices and values datasets
- **spec_dims** (*Dimension or array-like of Dimension objects*) – Sequence of Dimension objects that provides all necessary instructions for constructing the indices and values datasets Object specifying the instructions necessary for building the Spectroscopic indices and values datasets
- **translator_name** (*String / unicode, Optional*) – Name of the translator. Example - ‘HitachiSEMTranslator’
- **parm_dict** (*dictionary (Optional)*) – Dictionary of parameters that will be written under the group ‘Measurement_000’
- **extra_dsets** (*dictionary (Optional)*) – Dictionary whose values will be written into individual HDF5 datasets and whose corresponding keys provide the names of the datasets. You are recommended to limit these to simple and small datasets.
- **kwargs** (*will be passed onto hdf_utils.write_main_dset() which will in turn will be passed onto the creation of*) – the dataset. Please pass chunking, compression, dtype, and other arguments this way

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

Return type String / unicode

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

Translates data from a set of image files to an HDF5 file

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

1.14.3 pyUSID.processing

parallel_compute(*data*, *func*, *cores=1*, *lengthy_computation=False*, *func_args=None*, *func_kwargs=None*, *verbose=False*)

Computes the guess function using multiple cores

Parameters

- **data** (*numpy.ndarray*) – Data to map function to. Function will be mapped to the first axis of data
- **func** (*callable*) – Function to map to data
- **cores** (*uint, optional*) – Number of logical cores to use to compute Default - 1 (serial computation)
- **lengthy_computation** (*bool, optional*) – Whether or not each computation is expected to take substantial time. Sometimes the time for adding more cores can outweigh the time per core Default - False
- **func_args** (*list, optional*) – arguments to be passed to the function
- **func_kwargs** (*dict, optional*) – keyword arguments to be passed onto function
- **verbose** (*bool, optional. default = False*) – Whether or not to print statements that aid in debugging

Returns **results** – List of computational results

Return type list

class Process(*h5_main*, *cores=None*, *max_mem_mb=4096*, *verbose=False*)

Encapsulates the typical steps performed when applying a processing function to a dataset.

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.
- **cores** (*uint, optional*) – Default - all available cores - 2 How many cores to use for the computation
- **max_mem_mb** (*uint, optional*) – How much memory to use for the computation. Default 1024 Mb
- **verbose** (*Boolean, (Optional, default = False)*) – Whether or not to print debugging statements

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(***kwargs*)

Tests the process on a subset (for example a pixel) of the whole data. The class can be reinstantiated with improved parameters and tested repeatedly until the user is content, at which point the user can call compute() on the whole dataset.

Parameters **- dict, optional** (*kwargs*) – keyword arguments to test the process

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

1.14.4 pyUSID.viz

CHAPTER TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX