swmr_tools

Release 0.8.1

Richard E Parke

Aug 24, 2023

CONTENTS:

CHAPTER

ONE

GETTING STARTED

swmr_tools is a python package for making live data processing of hdf5_files easy.

swmr_tools can be installed from conda-forge using:

```
conda install -c conda-forge swmr-tools
```

It can also be installed from PyPi:

```
pip install swmr_tools
```

Alternatively you can clone the git repository containing swmr_tools using:

```
git clone https://github.com/DiamondLightSource/python-swmrtools.git
```

CHAPTER

TWO

HDF5 FILE REQUIREMENTS

To live process HDF5 data using the swmr_tools package there are a few requirements on the file structure.

- The file must be created in swmr mode (see https://docs.h5py.org/en/stable/swmr.html)
- The file must have one (or more) *key* datasets (see below)
- (Optional) The file can have a *finished* dataset (see below)

2.1 Key Datasets

Although swmr allows HDF5 to be read while being written, it can be difficult to determine whether a slice of the data has been written to or is just the fill data HDF5 uses when a dataset is expanded. To determine whether real data is actually written, swmr_tools needs a *key* dataset. The *key* dataset is usually an integer dataset, with a fill value of zero, which is flushed with a non-zero integer value after the corresponding frame of the main dataset is flushed. By monitoring these *key* datasets, swmr_tools can determine when each data frame is readable.

2.2 Finished Dataset

Since HDF5 datasets can be expanded it can be difficult to tell whether a file is complete or whether more data is likely to be written. The swmr_tools library uses a time out to determine when to finish, but this can also be paired with a *finished* dataset. The *finished* dataset is a single integer dataset, with a value zero when the file is still being written to and non-zero when the file is complete. This allows a long time out to be used without wasting time waiting when the file is complete.

CHAPTER

THREE

BASIC USE

The DataSource class is the simplest way to interact with a live swmr file. The DataSource is an iterator that provides a map of data for each frame.

The DataSource class requires 2 arguments:

- A list of *key* datasets.
- A list of datasets containing the data you wish to process.

The DataSource also has an optional *timeout* argument, which defaults to 10 second unless otherwise specified, and *finished_dataset* argument, which is a *finished* dataset.

The DataSource works out the dimensions of the frame (whether scalar, vector or image) by looking at the difference between the rank of the key and data datasets. It assumes that the data is written row-major and the data frames are in the fastest dimensions.

3.1 Reading Data

As an example we will create two small datasets (of the same size but containing different values) and corresponding unique key dataset to use in our example. This example shows a 2 x 2 grid scan of a detector with shape [5,10]. The keys will all be non-zero so we should expect to receive every frame of the dataset

```
from swmr_tools import DataSource
import h5py
import numpy as np

#Create a small dataset to extract frames from
data_1 = np.random.randint(low = -10000, high = 10000, size = (2,2,5,10))
data_2 = np.random.randint(low = -10000, high = 10000, size = (2,2,5,10))
keys_1 = np.arange(1,5).reshape(2,2,1,1)

#Save data to an hdf5 File
with h5py.File("example.h5", "w", libver = "latest") as f:
    f.create_group("keys")
    f.create_group("data")
    f["keys"].create_dataset("keys_1", data = keys_1)
    f["data"].create_dataset("data_1", data = data_1)
    f["data"].create_dataset("data_2", data = data_2)
```

Then we simply setup a DataSource pointing at the keys and datasets and let it run:

```
with h5py.File("example.h5", "r") as f:
    keys = [f["/keys/keys_1"]]
    datasets = {"/data/data_1" : f["/data/data_1"],
                "/data/data_2" : f["/data/data_2"]}
    ds = DataSource(keys,datasets)

    for data_map in ds:
        frame = data_map["/data/data_1"]
        print(data_map.slice_metadata)
        print(str(frame))

(slice(0, 1, None), slice(0, 1, None))
[[[[ 3980 -3645 -5966  8665   360  1863  7697  -769 -5559 -2142]
   [ 4588 -9254  8550 -1948  1172  -886  5600 -4307 -3488  2684]
   [ 6961 -6236 -4299 -7908  4577  4358 -6297 -8586 -4147 -3344]
   [ 7149 -2261  1190 -6692  -828  4310  5177 -1239  8868 -4319]
   [ 2442  5367 -1959  6815  5524 -2185 -2171 -8405 -2000 -6897]]]]
(slice(0, 1, None), slice(1, 2, None))
[[[[-4746  9432  4913 -7990 -7969   508 -4400 -4904   749 -1777]
   [-5639 -6433   214 -9282   951 -9444  3568   147 -3306  3393]
   [-9036 -9871 -9149  3938 -4487  9919  -170  5348  3916   289]
   [-3024   237  6456  8663  3531  8984 -3129  9678  3566  1306]
   [ 1891 -6206  9541 -4270 -7572 -6388 -1389  7990 -9341  8785]]]]
(slice(1, 2, None), slice(0, 1, None))
[[[[ 5964  6778 -1285 -4820  1111  5613 -3506 -2496 -6278  2581]
   [ 5037 -1065 -5667  1903  -311 -3747  1912  8773  1429   459]
   [ 4058  6380 -8450 -6520  7715  2446  8190 -6177 -9543  5414]
   [-6701  -870 -7936 -1994  9943  7053  9467 -5751 -7643  1843]
   [ 5033  4083  4520 -3509  9507  1576  9728 -1245  3678 -9098]]]]
    ...
```

The data (as numpy arrays) can be accessed from the data_map for each point using the dataset path as a key in the map. The slice_metadata attribute on the data_map shows the slice the data was taken from.

The slice_metadata can be used to write processed data into a new hdf5 dataset, and the DataSource class has some convenience methods to help with this.

3.2 Writing Data

The DataSource class has two methods to assist with writing processed data back into a hdf5 file:

```
ds.create_dataset(result_data,file_handle,hdf5_path)
```

which creates a new hdf5 dataset, with the correct type and shape for the result_data numpy array, and:

```
ds.append_data(result_data,slice_metadata,output_dataset)
```

which adds new result datasets into this hdf5 dataset.

CHAPTER

FOUR

ADVANCED USE

The DataSource class is designed to be simple but because of this may not work for every method of processing (for example if for performance reasons you dont want to read every frame, or only want to read a region of each frame).

For more complicated use cases the KeyFollower and FrameReader classes can be used.

4.1 KeyFollower

The KeyFollower is the most fundamental class in swmrtools; it follows the key datasets and reports the highest index for which all the key datasets are non-zero.

As an example we will create a dataset of non-zero integers, respresenting a complete set of scans all flushed to disk

```
import h5py
from swmr_tools import KeyFollower
import numpy as np

#create a sequential array of the numbers 1-8 and reshape them into an array
# of shape (2,4,1,1)
complete_key_array = np.arange(8).reshape(2,4,1,1) + 1
```

We then create an empty hdf5 file, create a group called “keys” and create a dataset in that group called “key_1” where we will add our array of non-zero keys

```
with h5py.File("test_file.h5", "w", libver = "latest") as f:
    f.create_group("keys")
    f["/keys"].create_dataset("key_1", data = complete_key_array)
```

Next, we shall create an instance of the KeyFollower class and demonstrate a simple example of its use. At a minimum we must pass the key datasets we wish to read from.

Shown below is an example of using an instance of KeyFollower within a for loop, as you would with any standard iterable object. For this basic example of a dataset containing only non-zero values, the loop runs 8 times and stops as expected

```
# using an instance of Follower in a for loop
with h5py.File("test_file.h5", "r", swmr = True) as f:
    keys = [f["/keys/key_1"]]
    kf = KeyFollower(keys)
    for key in kf:
        print(key)
0
```

(continues on next page)

(continued from previous page)

```
1
2
3
4
5
6
7
```

As with the DataSource, the timeout and finished_dataset can be set on contruction of the KeyFollower.

Running the KeyFollower should not be computationally expensive, because all of the *key* datasets should be relatively small, allowing the KeyFollower to follow a very rapid scan.

The DataSource class is just a KeyFollower that uses a FrameReader to read a frame from each requested dataset. The FrameReader class can also be used outside the DataSource.

4.2 FrameReader

The FrameReader class is constructed using the dataset to read, and the rank of the scan (1 for a stack of images, 2 for a grid scan etc).

The read_frame(index) method then reads the frame corresponding to the index *i* which can be provided by the KeyFollower.

CHAPTER

FIVE

REFERENCES

5.1 swmr_tools package

5.1.1 Submodules

5.1.2 swmr_tools.DataSource module

5.1.3 swmr_tools.KeyFollower module

5.1.4 Module contents