CATEGORY ARCHIVES: PYTHON

[Python] Read EMD

Posted on **November 8, 2016** by **Florian**

In this post we will see what it takes to read an EMD file with python. As a test case we consider the *test.emd* file written in the previous post. Don't forget to take a look at our collection of common pitfalls.

BASIC

In the easiest case you know the structure of the EMD file either because you have written it yourself or from browsing it with a different tool like the EMD viewer. In this case it takes only a few lines to access the data.

```
import h5py

# open the EMD file
f = h5py.File('test.emd', 'r')

# assuming you know the structure of the file
emdgrp = f['data/dataset_1']

# read data
data = emdgrp['data'][:]

# close the EMD file
f.close()
```

First, we import the `h5py` package to facilitate working with HDF5 files. We then open the EMD file by specifying its path. Here we use the readonly option to not accidentally change its content.

Remember that the HDF5 file works as its own small filesystem. We can therefore access the *dataset_1* group in our EMD file by specifying its full path inside the file. For convenience we save a reference to this group in a variable called `emdgrp`.

The dataset *data* saved in the *dataset_1* group can be accessed the very same way, using `emdgrp['data']` which is equivalent to `f['data/dataset_1/data']`. We use the `[:]` indexing here to copy all the values to a new `numpy.ndarray` object referenced by the `data` variable.

Finally, we close the EMD file calling `f.close()`, as we do not need it anymore.

MORE

Of course this is not the only thing one can read from the EMD file. For example one will generally want to access the dimensions information as well. The following lines read in the *dim1* datasets plus the metadata stored in the attributes. (Note that the following code examples have to be placed before `f.close()`, as they need to access information from the file.)

```
# read dimensions 1 and 3
dim1 = emdgrp['dim1'][:]
dim1_meta = [emdgrp['dim1'].attrs['name'], emdgrp['dim1'].a

dim2 = emdgrp['dim2'][:]
dim2_meta = [emdgrp['dim2'].attrs['name'], emdgrp['dim2'].a

dim3 = emdgrp['dim3'][:]
dim3_meta = [emdgrp['dim3'].attrs['name'], emdgrp['dim3'].a
```

This information can be used for further processing, for example to create coordinate arrays which are useful to evaluate mathematical functions at the same points we have data values for.

```
# create x and y coordinate arrays
import numpy as np
xx, yy = np.meshgrid(dim1, dim2)
```

The metadata becomes for example important when it comes to plotting. In the following line a string is created to potentially label the *z*-axis of the dataset.

```
# label for z axis
print('{} {}'.format(dim3_meta[0].decode('utf-8'), dim3_meta
```

Lets find out whom to contact, in case we have any questions about how the data has been acquired. Simply grap the *email* attribute from the user group.

```
# grap email from user
email = f['user'].attrs['email'].decode('utf-8')
print('In case of questions, let\'s ask {}'.format(email))
```

To review the changes made to the EMD file, we can have a look at all notes in the *comments* group:

```
# review changes logged in the comments section
changes = f['comments'].attrs
for key in changes:
    # iterating over dict
    print('{}:\t{}'.format(key, changes[key].decode('utf-8'
```

ADVANCED

In case you do not know the structure of your EMD file or are to lazy to look it up, you can iteratively search for things. The following lines go through the items in the file and test them for the *emd_group_type* attribute.

```
# recursive function to run and retrieve groups with emd_gr
def proc_group(group, emds):
    # take a look at each item in the group
    for item in group:
        # check if group
        if group.get(item, getclass=True) == h5py._hl.grou
            item = group.get(item)
            # check if emd_group_type
            if 'emd_group_type' in item.attrs:
                if item.attrs['emd_group_type'] == 1:
                    print("Found EMD group at: {}".forma
                    emds.append(item)
            # process subgroups
            proc_group(item, emds)

# run
emds = []
proc_group(f, emds)
```

We define a processing function, which we recursively run on all groups in the file. Given a parent `group` a for loop iterates over every `item` in the group. As these can be datasets or groups, the item is checked which type it is. Only in the case of a group, we check for the *emd_group_type* attribute and whether it is set to 1. If both applies, a message is printed out and a reference to this group is saved in a list. For the case `item` is a group, the function is recursively run on them, in case of a dataset, nothing further happens.
To execute the search, we create an empty list `emds` and start the recursion by running the `proc_group` function on the root of the EMD file.

Posted in **Developer**, **Python**

[Python] Write EMD

Posted on **November 5, 2016** by **Florian**

In this post we will go through what it takes to write a simple EMD file in python. If you want to learn how to read an EMD file in python, take a look here.

We will write an EMD file containing a 512x512x100 datacube filled with random numbers. The finished python script can be found here. Please note that the EMD file created by this script is about 200 MB in size.

```
import h5py
import numpy as np
import datetime
```

EMD files are based on the HDF5 format. Therefore we import `h5py` package containing the python interface to the HDF5 library. Further details on this package and how to install it can be found on the official website. `h5py` uses `numpy` arrays to handle the data contained in an HDF5 file, so the `numpy` package is imported. We will also use it to create our random test data. The `datetime` package is imported to create a timestamp for the comments metadata.

```
# create file
f = h5py.File('test.emd', 'w')
```

To create our EMD file let `h5py` create a new HDF5 file name *test.emd*. The *w* parameter opens the file in write mode. We save the reference to the root group in variable `f`.

```
# set version information
f.attrs['version_major'] = 0
f.attrs['version_minor'] = 2
```

To let the reader know which specification the data contained in this file follows, we set the version information as attributes of the root group. Note that every group and dataset in an HDF5 file interfaced by `h5py` has an attribute called `attrs` containing the HDF5 attributes of this group or dataset as a python dict.

```
# add a group
grp_exp = f.create_group('data')
```

Next we add a *data* group as a container for the datasets we are going to write in this EMD file. This is especially useful, if you want to put multiple datasets within a single EMD file.

```
# add an emd type subgroup for the dataset
grp_dat = grp_exp.create_group('dataset_1')
grp_dat.attrs['emd_group_type'] = 1
```

Our dataset itself will be contained in another subgroup comprising the actual data and the dimension vectors. This group is given a meaningful identifier best describing the dataset (*dataset_1* in our case). To make this group be recognized as an emd-type dataset, we add the attribute *emd_group_type* and set it to the integer value 1.

```
# create a 3D dataset with random floats
data = grp_dat.create_dataset('data', (512,512,100), dtype
data[:,:,:] = np.random.rand(512,512,100)
```

To this group we add the actual dataset using the `create_dataset` method. Its parameters are the label of the dataset, which has to be *data* in the EMD specification, the shape of the dataset and its datatype. Here we create a 512x512x100 three dimensional datacube of float values. To write data to this dataset, we use `numpy` indexing with the given handle. In this example we create a 512x512x100 dataset of random floats using the `random.rand()` method from `numpy` and set it to our EMD dataset.

```
# add dimension vectors
dim1 = grp_dat.create_dataset('dim1', (512,), dtype='int'
dim1[:,0] = np.arange(512)
dim1.attrs['name'] = np.string_('x')
dim1.attrs['units'] = np.string_('[px]')
```

In addition to the actual data we need to supply the dimensions for each axis in *dim1* datasets within the same group. These contain the values along this axis for each element in that direction. The datasets are created in the EMD file in the same way we created the datacube. The first dimension in this example is filled with integers indicating the nth pixel in the x direction. Attributes are used to indicate the label for this axis (*name*) and the *units* used for the values. See also the recommendation for consistent unit description in the specifications. To save data as strings using the HDF5 library, the `np.string_()` method is used to parse the string to fixed-width byte strings.

```
dim2 = grp_dat.create_dataset('dim2', (512,), dtype='int'
dim2[:,0] = np.arange(512)
dim2.attrs['name'] = np.string_('y')
dim2.attrs['units'] = np.string_('[px]')

dim3 = grp_dat.create_dataset('dim3', (100,), dtype='floa
dim3[:,0] = np.linspace(0.0, 3.14, num=100)
dim3.attrs['name'] = np.string_('angle')
dim3.attrs['units'] = np.string_('[rad]')
```

Dimension vectors have to be provided for each dimension of the original dataset. The above code creates the *dim2* and *dim3* datasets analogously to the *dim1* dataset. The *dim2* datasets contains integers indicating the nth pixel in y direction similar to *dim1*, *dim3* contains float values describing a fictional angular dimension running from 0 to pi.

The following groups and attributes are not necessary to create a valid EMD file. However it is good practice to use them to supply metadata in a standardized way, facilitating the exchange of scientific datasets. After all, this is what the EMD file format is all about.

```
# create microscope group for metadata
grp_mic = f.create_group('microscope')
grp_mic.attrs['magnification'] = 10
```

The *microscope* subgroup is recommended to store the experimental settings of the microscope which have led to the acquisition of the saved dataset. The metadata is stored in single attributes to this group, exemplarily shown here for a fictional *magnification* of 10x.

```
# create user group for user info
grp_usr = f.create_group('user')
grp_usr.attrs['operator'] = np.string_('me')
grp_usr.attrs['email'] = np.string_('me@mine')
```

The *user* subgroup is supposed to contain information about the operator of the microscope. It should contain contact information of whom to ask about the experiment or simulation whose results are provided in the EMD file.

```
# create sample group for information on sample
grp_spl = f.create_group('sample')
grp_spl.attrs['material'] = np.string_('random')
```

The *sample* group should contain information about the sample, e.g. a unique identifier and information of the material and preparation method.

```
# create comments group for log
grp_com = f.create_group('comments')

# add a comment on file creation with the current timestam
timestamp = datetime.datetime.utcnow().strftime('%Y-%m-%d
grp_com.attrs[timestamp] = np.string_('File created, fille
```

The *comments* group is recommended to hold timestamped information on the history of the EMD file. An attribute should be added on every change made to the file. Here we retrieve the current timestamp using the `datetime` package and use it to add a comment on the creation of this file.

```
# close the file
f.close()
```

Finally we close the EMD file. Congratulations, you have written your first EMD file using python!

Make sure to play around with this file using the EMD Viewer, or try to read it using this post. A number of common pitfalls witnessed when working with EMD files using the h5py package in python has been compiled here.

Posted in **Developer**, **Python**

[Python] Common Pitfalls

Posted on **November 4, 2016** by **Florian**

Do not make the same mistakes we did! Here you find a list of common pitfalls we have witnessed when working with EMD files in python.

- Note that the first dimension is saved in the dataset called *dim1*, there is no *dim0*.
- The dimensions of a dataset in `h5py` are in ascending order *1,2,...n* or *x,y,...n*. When working with images as numpy arrays however, the usual way to order the dimensions is as *y,x* corresponding to rows and columns on the screen. To interchange these with the EMD file, one has to flip *x* and *y* directions by using for example `np.transpose()`.
- To correctly save strings using `h5py` the use of fixed-width byte strings is encouraged. Saving python string objects can lead to encoding errors or worse. Just parse your string through `np.string_('example')`. To convert back just decode it to UTF8 like `b'example'.decode('utf-8')`
- Remember, that you can create significant memory leakages in python, if you are not careful about assigning variables. There is a difference between `h5py_dset = data` and `h5py_dset = data[:]`, which becomes interesting when you repeatedly read in *data* in a loop.
- There have been reports about performance issues in `h5py` related to `numpy` indexing. Feel free to do a quick internet research.

Posted in **Developer**, **Python**

EMD LINKS

Download ver. 0.4.2 [13.4 MB]
Windows 64-bit executable

Source Code (c++)
GitHub Repo