SACLA data processing

mender:donians

1 Data analysis basics

1.1 the data file

At SACLA, the data files are in hdf5 format. An hdf5 file object is composed of group and dataset objects. A dataset is similar to a numpy array, and a group is a hierarchical grouping of multiple datasets. You can think of groups as directories and datasets as file basenames. To open them in Python, use the h5py module

```
import h5py
fh5 = h5py.File( 'subset_178884.h5', 'r')
```

The file *subset_178884.h5* contains a portion of the images from run number 178884 of our last experiment. A run has a fixed number of exposures. In this case, the full run has 5050 exposures (the first 50 of which have the shutter closed). This subset file has only the first 199 exposures, yet data from other beam line monitors exists for all 5050 exposures.

There is a lot of information stored within the file, and you can explore its contents using the keys method:

```
print f.keys()
#  [u'file_info', u'run_178884']
print f['run_178884'].keys()
#  [u'detector_2d_assembled_1', u'event_info', u'exp_info', u'run_info']
```

You will probably want to focus on the data images themselves. You can find them in the fixed hdf5 path *'/run_178884/detector_2d_assembled_1'*. Each exposure will have a unique tag associated with it. Find the tags using the keys method (we will skip the *'detector_info'* group because it does not contain an image)

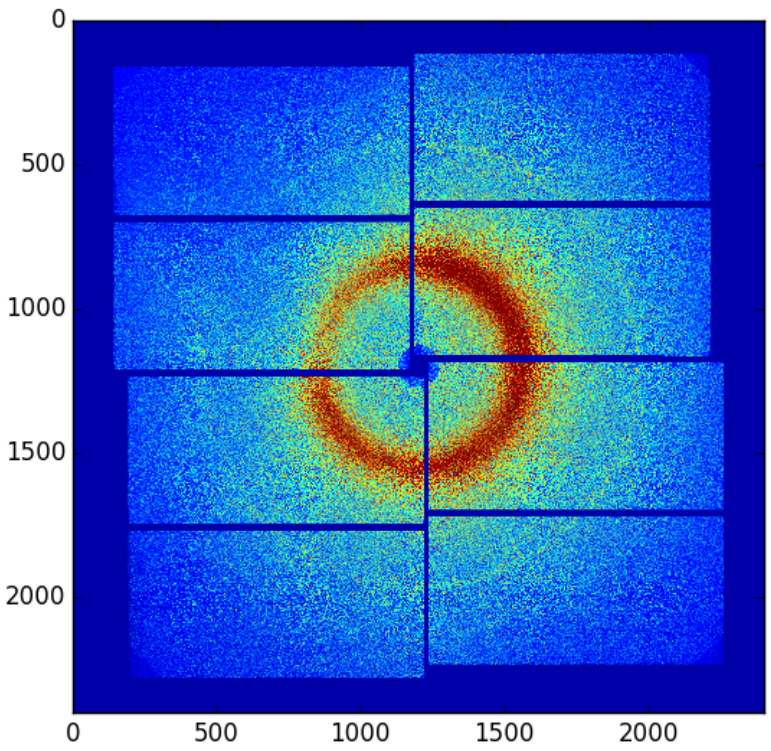
```
imgs_path = '/run_178884/detector_2d_assembled_1'
print fh5[ imgs_path].keys()
# [u'detector_info',
#  u'tag_166762140',
#  u'tag_166762142',
#  u'tag_166762144',
#  ...
#  u'tag_166762536']

exposure_tags = fh5[ imgs_path].keys()[1:]
print exposure_tags
```

```
# [u'tag_166762140',
#  u'tag_166762142',
#  u'tag_166762144',
#  ...
#  u'tag_166762536']

num_img = len( exposure_tags )
print num_img
# 199
print fh5[ imgs_path + '/' + exposure_tags[0] ].keys()
# [u'detector_data', u'detector_status']
```

Now, we have the full path to the data, and we can select the image of each tag using

```
img = fh5[ imgs_path + '/' + exposure_tags[60] + '/detector_data' ].value
from pylab import *
# now you have loaded all pylab and numpy packages
imshow( img, vmax = 1000 )
show()
```

which should display the image:



The full run files are huge and it is not advisable to unpack all images at once. The proper way to handle this is to use Python generators. We can create a generator and load the images in turn

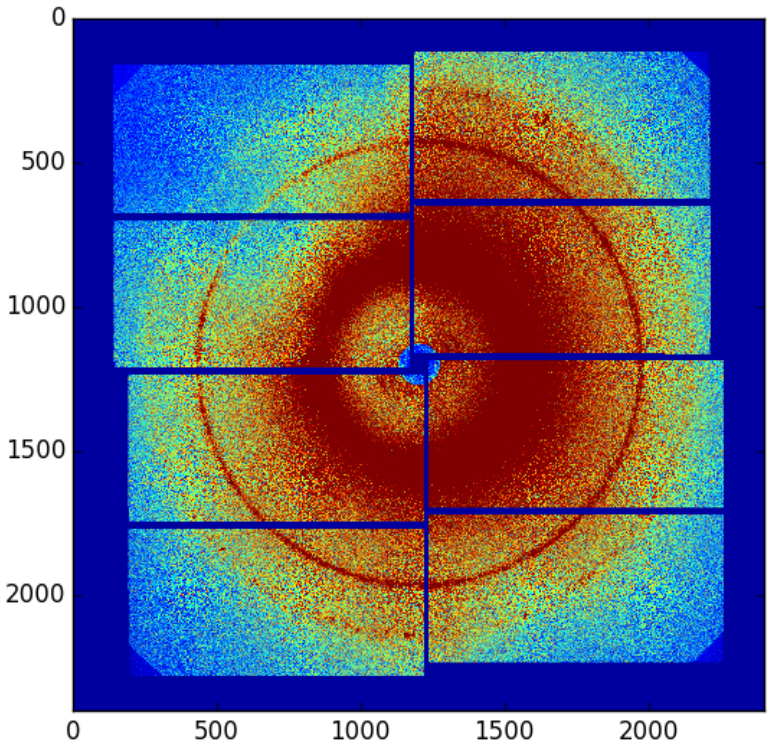
```
img_gen = ( fh5[ imgs_path + '/' + tag + '/detector_data' ].value for tag in
    exposure_tags )

for dummie in xrange( num_img ):
    imshow( img_gen.next() , vmax=1000)
    draw()
    pause(3)
```

It is important to use parentheses '( )' and not square brackets '[ ]' in the definition of img_gen above. One creates a Python generator (good), and the other creates a Python list, loading everything into memory (bad).

1.2 Using the RingData library

In order to correlate a detector image, one needs to first determine the pixel point where the forward x-ray beam would hit the detector in the absence of a beam stop or hole. This is commonly referred to as the "center" of the diffraction image. We can load an image with sharp diffraction rings for finding the center (i.e. try a different tag):

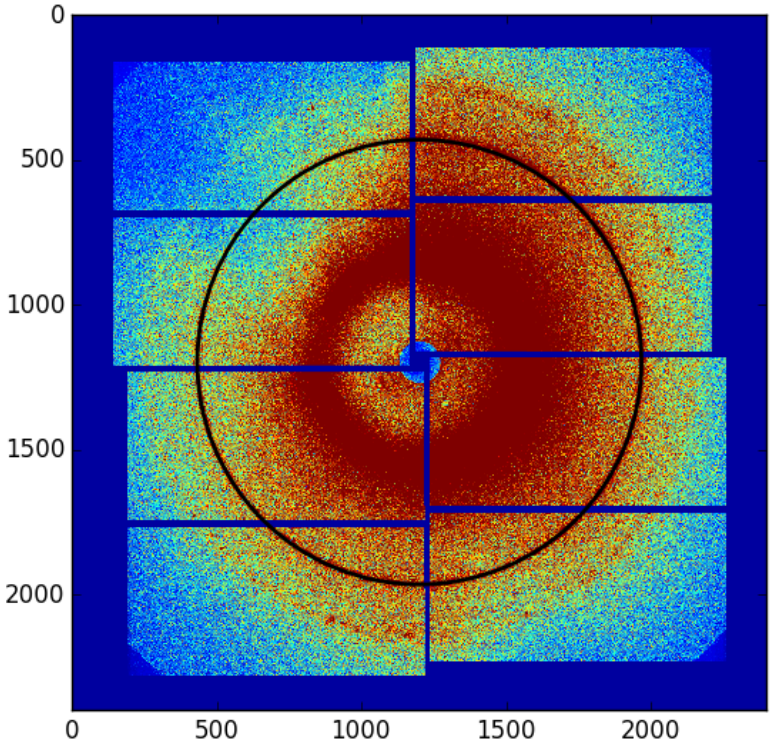
```
img = fh5[ imgs_path + '/' + exposure_tags[100] + '/detector_data' ].value
imshow(img, vmax = 1000)
show()
```



This looks like a nice ring pattern. Using the pylab GUI, one can guess the radial position of the brightest ring (measured from the center) to be 765 pixel units. A good guess for the

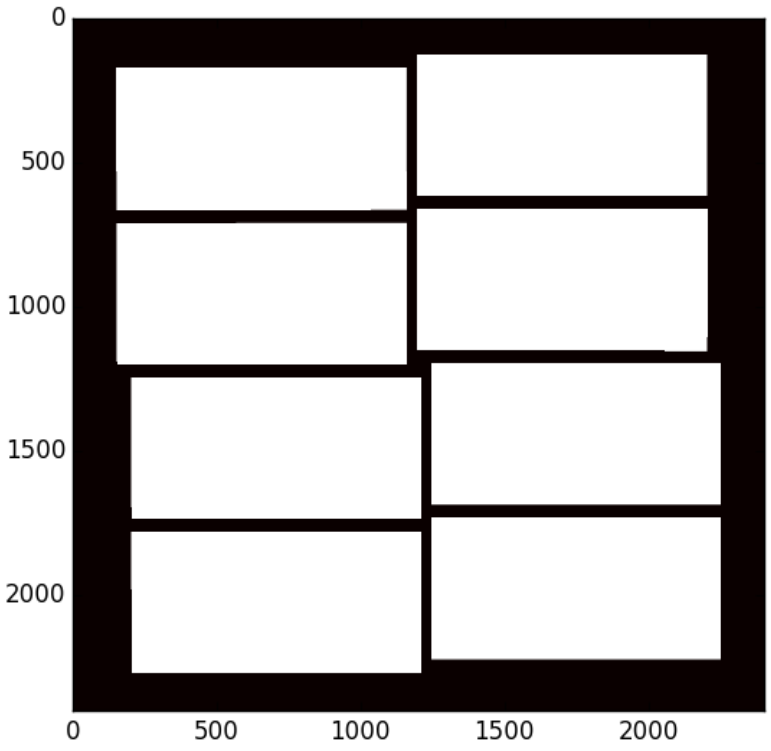
center of a symmetric diffraction image like this one is just the center of the detector:

```
x_i  = img.shape[1]/2. # fast dimension of image
y_i = img.shape[0]/2. # slow dimension of image
q_i = 765.

# change directories to wherever the RingData.py file is stored
import os
os.chdir( '/Users/mender/for_schengjun' )
from RingData import RingFit
RF = RingFit( img )
x_center, y_center, q_ring = RF.fit_circle( beta_i=( x_i ,y_i , q_i ) )

# plot the result
imshow(img, vmax = 1000)
gca().add_patch(Circle( xy=(x_center,y_center), radius=q_ring , ec = 'k', fc =
    'none', lw= 2,) )
show()
```



Now, I load a user defined mask file. In this case, the mask is saved as a numpy array, in the numpy binary format (.npy). A '1' is unmasked and a '0' is masked, meaning it won't be included in computations. It can be loaded using:

```
mask_file = '/Users/mender/for_schengjun/mask.npy'
mask = np.load( mask_file ) # pylab auto-loads the numpy module as np
imshow( mask, cmap='hot')
show()
```



For now I will include a basic outline of how to use the rest of the library, but it is still under development. This is why I will pre-set some of the parameters. I will add more to this section, and if you find yourself confused before I get the chance to update the document, please email (dermen@stanford.edu).

```
#######################
# GLOBAL PARAMETERS #
#######################
x_c, y_c, qR_c = x_center, y_center, q_ring
nq = 20 #width of the polar image, which is centered at qR_c
Y,X = img.shape
raw_img_shape = img.shape
nphi = int( 2 * pi * qR_c )
from RingData import Interp, DiffCorr
def corRun( imgen , num_shots):
    '''
    Example usage of RingData Methods
    =================================
    imgen        - generator, image generates 2d np.array images for a given run,
    num_shots    - int, the number of shots in run
    raw_img_shape - int tuple, shape of the 2d images
    '''
#   number of boxes around ring for fitting splines in
    num_phi_bin = raw_img_shape[0] / 5
#   interpolate intensities using magic
    print "interpolating..."
    runI    = Interp(x_c, y_c, qR_c, qR_c+nq, qR_c-nq, num_phi_bin, nphi,
        bin_fac=1, fastDim= X, slowDim=Y, raw_img_shape=raw_img_shape )
```

```
    pmask   = runI.fast_spline( mask, dtype=bool ).round()
    pols    = array( [ pmask* runI.fast_spline( imgen.next() , spline_order = 3 )
        for dummie in xrange( num_shots-1 ) ] )
#   normalize before differencing to account for sample thickness variation
    norms   = median( pols,axis=2 )
    polsnorm = rollaxis( rollaxis( pols, 2 ) / norms , 0,3 )
#   difference correlate (delta_shot is the spacing in time between shots, 1
    being the consecutive)
    print "correlating..."
    DC      = DiffCorr( polsnorm, delta_shot=1 )
    cors    = DC.autocorr(num_high=10, num_low=10)
    cors    = DC.normalize()
    return cors

# re-initialize the image generator
img_gen = ( fh5[ imgs_path + '/' + tag + '/detector_data' ].value for tag in
    exposure_tags )
cor_img = corRun ( img_gen, num_img )
# plot the result
cor_img_shotMean =cor_img.mean(axis=0)
imshow( cor_img_shotMean[: , 10:-10], aspect='auto' )
show()
```