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 fh5.keys()
#  [u'file_info', u'run_178884']
print fh5['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)

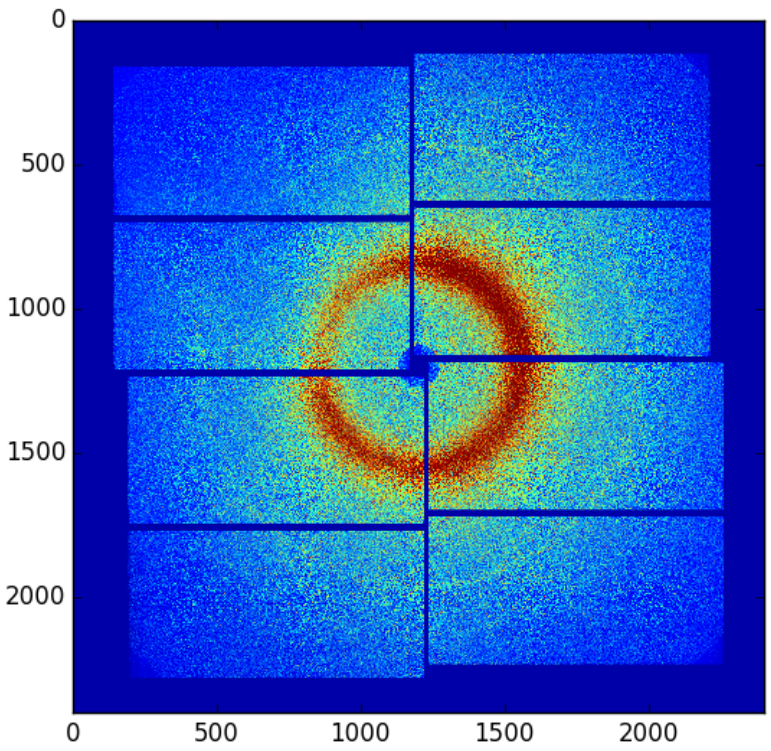
```
imgs_path = '/run_178884/detector_2d_assembled_1'
print fh5[ imgs_path].keys()
# [u'detector_info',
#  u'tag_166762140',
#  u'tag_166762142',
#  u'tag_166762144',
#  ...
#  u'tag_166762536']

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

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

num_imgs = len( exposure_tags )
print num_imgs
# 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

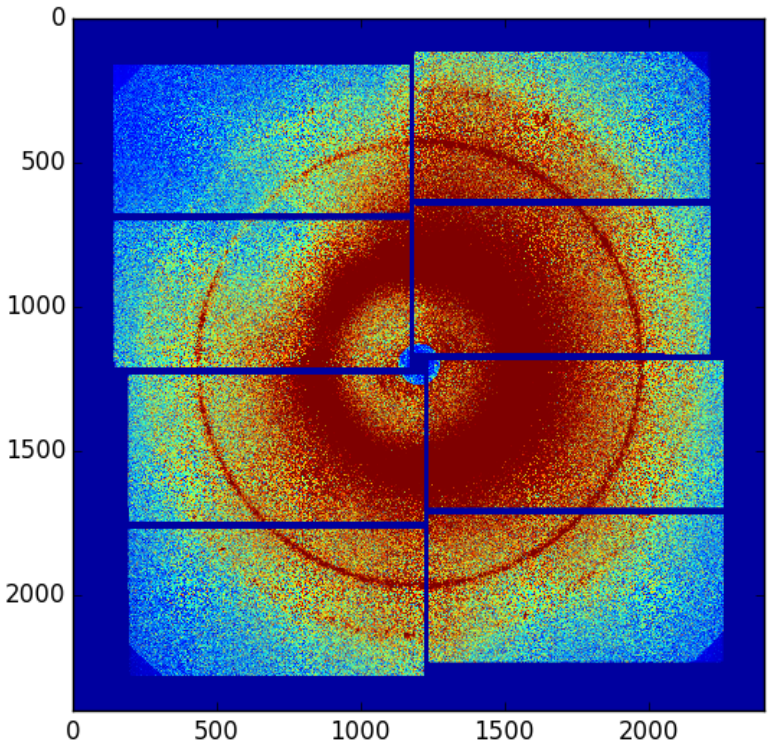
```
img_gen = ( fh5[ imgs_path + '/' + tag + '/detector_data' ].value for tag in
    exposure_tags )

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):

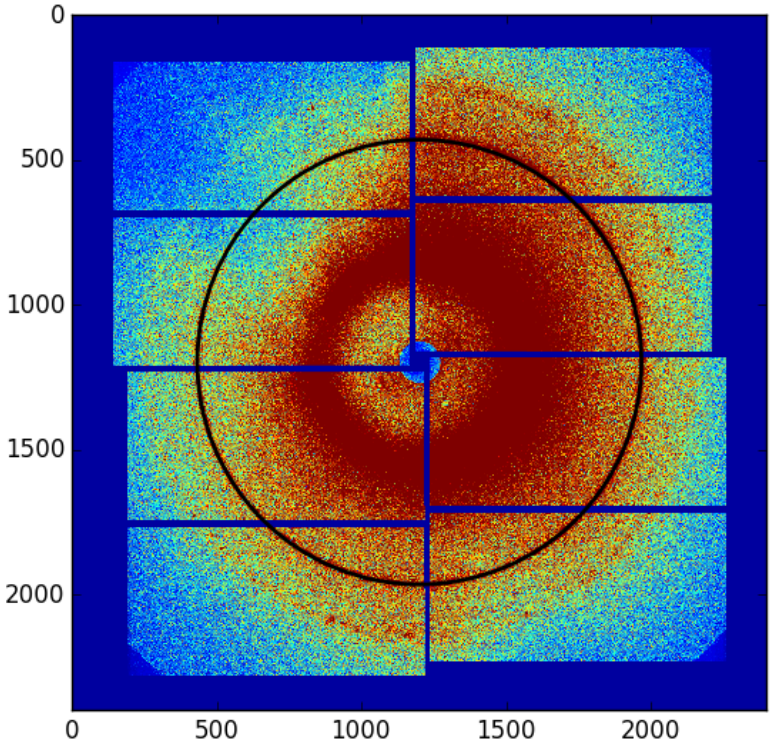
```
img = fh5[ imgs_path + '/' + exposure_tags[100] + '/detector_data' ].value
imshow(img, vmax = 1000)
show()
```



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

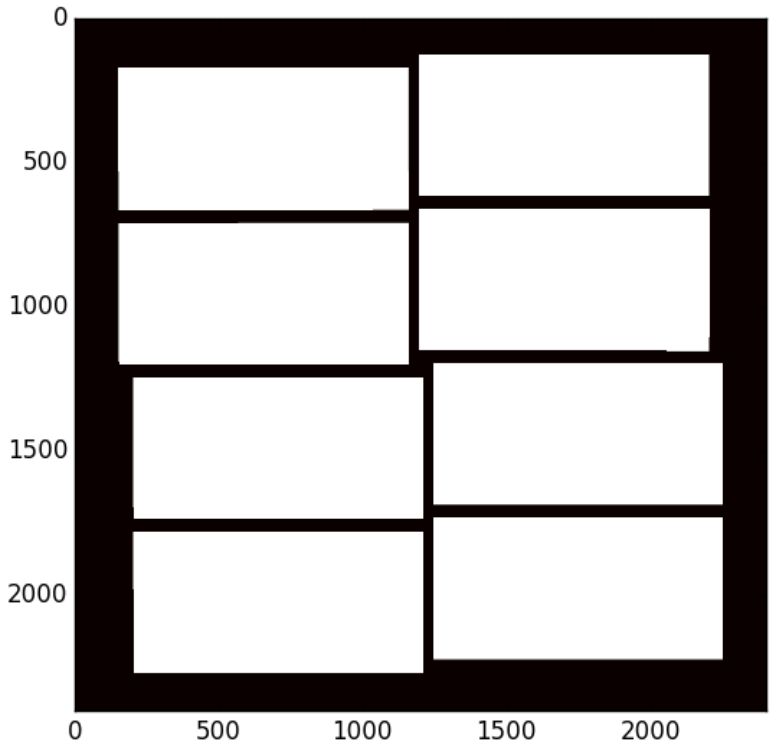
center of a symmetric diffraction image like this one is just the center of the detector:

```
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).

```
import numpy as np
import h5py
import json
from RingData import InterpSimple, DiffCorr

# User defined function for normalizing the polar images
# this can be a critical step when doing difference correlations
# (you don't want to subtract and correlate two shots if they have an intensity
    offset)
def normalize_polar_images( imgs, mask_val = -1 ):
    norms = ma.masked_equal( imgs, mask_val).mean(axis=2)
    imgs /= norms[:,:,None]
    imgs[ imgs < 0 ] = mask_val
    return imgs


#######################
# GLOBAL PARAMETERS #
#######################
x_c, y_c = x_center, y_center # pixel units
qmin        = 2.64          # inverse angstroms
qmax        =  2.71      # inverse angstrom
```

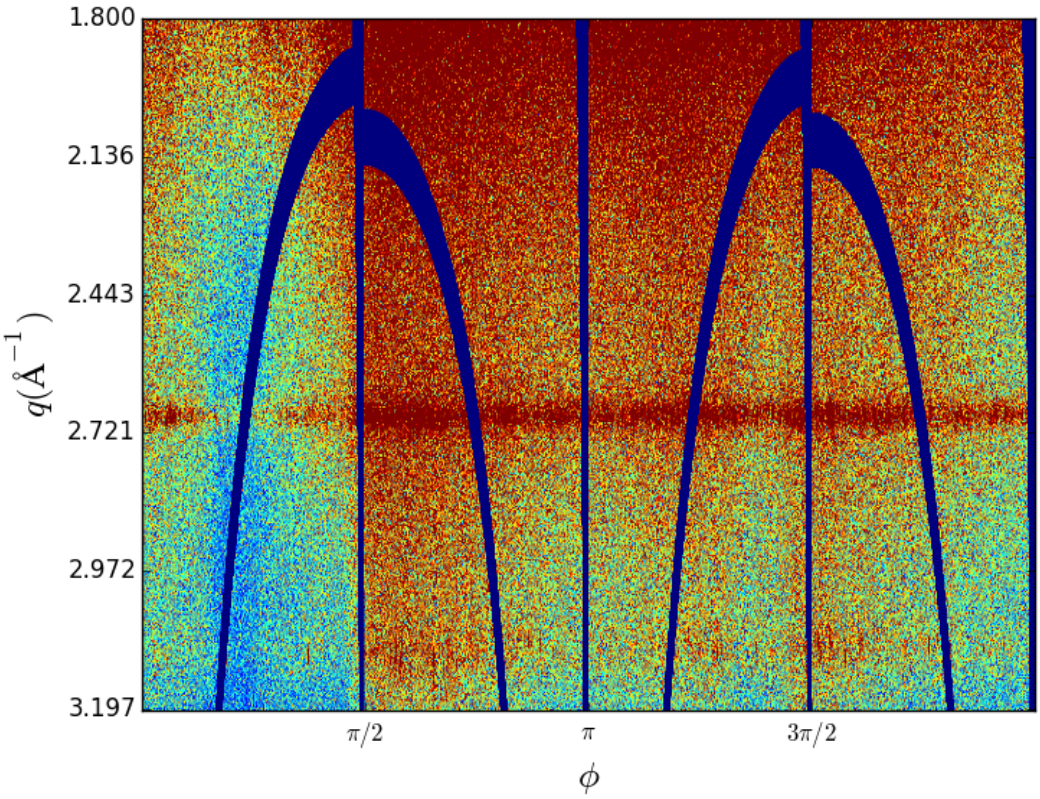
```
pixsize     = 0.00005 # meter
wavelen   = 1.44       #angstrom
detdist      = 0.053     #meter
nphi          =  int( 2 * pi * qmax ) # single pixel resolution at maximum q

prefix = '178884'
output_hdf = h5py.File( prefix + '.hdf5', 'w' )

#some useful functions
pix2invang = lambda qpix : sin(arctan(qpix*pixsize/detdist )/2)*4*pi/wavelen
invang2pix = lambda qia : tan(2*arcsin(qia*wavelen/4/pi))*detdist/pixsize

qmin_pix = invang2pix ( qmin )
qmax_pix = invang2pix ( qmax )

# initialize the interpolater
interpolater = InterpSimple( x_c, y_c, qmax_pix, qmin_pix, nphi, raw_img_shape =
    mask.shape )

# make a polar image mask
pmask  = interpolater.nearest( mask , dtype=bool ).round()

# re-initialize the image generator
img_gen = ( fh5[ imgs_path + '/' + tag + '/detector_data' ].value \
                    for tag in exposure_tags )

# Make the polar images
# (be reasonable with this list, if qmax - qmin is very large then this list will
    get rather large as well and
# you don't want to load it all into memory
# In such a case you should save it periodically to files depending
# on the amount of available system memory)

polar_imgs = np.array( [ pmask * interpolater.nearest( img_gen.next() ) for
    dummie in range( num_imgs) ] )

# save the data raw
output_hdf.create_dataset( 'polar_data',data = polar_imgs)

# Consider normalizing the images
polar_imgs = normalize_polar_images( polar_imgs, mask_val=0)

# save the data normalized
output_hdf.create_dataset( 'normalized_polar_data',data = polar_imgs)

#   save a lookup-map
tag_map = {}
for indx,tag in enumerate( exposure_tags ):
    tag_map[tag] = indx

#   save meta data
output_hdf.create_dataset( 'polar_mask',data = pmask.astype(int))
output_hdf.create_dataset( 'x_center', data = x_center)
output_hdf.create_dataset( 'y_center', data = y_center)
output_hdf.create_dataset( 'q_ring',   data = q_ring)
output_hdf.create_dataset( 'num_phi', data = nphi)
output_hdf.create_dataset( 'wavelen' , data = wavelen)
```

```
output_hdf.create_dataset( 'pixsize' , data = pixsize)
output_hdf.create_dataset( 'detdist' , data = detdist)

#   save the q-mapping
qrange_pix = np.arange( qmin_pix, qmax_pix )
q_map      =  np.array( [ [ ind, pix2invang(q) ] for ind,q in enumerate(
    qrange_pix) ] )
output_hdf.create_dataset( 'q_mapping' , data = q_map )

# save
output_hdf.close()

# save the lookup-map
dump_file = open( prefix + '.json', 'w')
json.dump( tag_map, dump_file )
dump_file.close()
```

Before we close this, lets have a look at the image we centered earlier, only this time in polar coordinates

```
tag = exposure_tags[100]

# load the polar data
data_hdf   = h5py.File( prefix + '.hdf5' )

polar_data = data_hdf['polar_data']

# load out map
tag_map = json.load( open( prefix + '.json') )
indx       = tag_map[ tag ]

polar_img = polar_data[ indx]
imshow( polar_img, vmax = 1000, aspect='auto' )

# adjust the x axis
xtic = [ nphi/4, nphi/2, 3*nphi/4 ]
xlab = [ r'$\pi/2$', r'$\pi$', r'$3\pi/2$' ]
xticks( xtic, xlab )

# adjust y axis
ytic = arange( 0, nq, nq/5 )
ylab = [ '%.3f'%(q_map[x]) for x in ytic ]
yticks( ytic, ylab )

ylabel(r'$q (\AA^{-1})$',fontsize=18)
xlabel(r'$\phi$',fontsize=18)
show()
```

This image is one of gold, and you can see the {111} Bragg ring at $2.67\mathring{A}^{-1}$. Now, for the angular correlations. We will be using the difference correlations routine, where pairs of exposures are subtracted and then correlated. This emphasizes the correlations unique to each exposure (e.g. photon correlations from multiple photons scattering off the same sample particle), while minimizing the correlations that are common to the exposures (e.g. detector artifacts). Assume you have a list of exposure tag pairs that will be good for



difference correlations. At first we can consider the list to just be consecutive exposures. but this list can become more complex, and it will probably become more complex as the analysis is repeated. (for instance, some shot pairs might give large correlation outliers, and these will need to be filtered). We are using a list of image tags because we already made a tag mapping to the polar images saved on disk.

```
# simply correlated every other exposure, except for the first 50 exposures as
    they are dark images in this run
simple_map = zip( exposure_tags[50:-1], exposure_tags[51:] )
# I will save this to disk
tmp_file = open( prefix + '_tag_pairs.json' , 'w')
json.dump( simple_map, tmp_file )
tmp_file.close()
```

and now we can use this object to do the correlations quite simply

```
tag_pairs = json.load( open( prefix + '_tag_pairs.json' ))

# already loaded the polar data and the tag_map
exposure_diffs = []
for i_, tags in enumerate( tag_pairs) :
    tagA, tagB = tags
    try: indxA = tag_map[ tagA]
    except KeyError: continue
    try: indxB = tag_map[ tagB]
    except KeyError: continue
```

```
    shotA = polar_data[ indxA ]
    shotB  = polar_data[ indxB]
    exposure_diffs.append(  shotA - shotB )

exposure_diffs = np.vstack( exposure_diffs )

# take the autocorrelation of each pair
DC     = DiffCorr( exposure_diffs )
cor      = DC.autocorr()
# take the mean over pairs
cor_m = cor.mean(0)

imshow( cor_m[:, int( nphi*0.05) :-int( nphi*0.05)], aspect='auto' )
show()
```