SOP for Data Analysis – LR83

Jen Ruddock, March 22, 2018

So you have data! Yay! Now you want to look at it, and look at it properly. Here's what to do. PLEASE READ BEFORE YOU BEGIN: Make sure you put in the file paths you want for writing/loading files, and that you use the correct run numbers and the correct time bins, etc.

Notes on common errors seen with Python:

If you see a message like this:

```
IOError: Unable to open file or directory
```

There are a few possibilities for this: (1) You did not ssh psana so you don't have access to experiment directories. (2) You are writing/reading from a directory that does not exist (so you have to make the directory!) (3) The experiment you want to access is stored on tape and you need to go into the elog and restore it to disk.

Other common issues:

IndentationError: usually means you are mixing up tabs and spaces. If you use spaces, every line must use spaces, etc.

All your output is zeros! This probably means you set something to an integer when it should be a floating point number. In Python, any whole number is assumed to be an integer, and math performed on an integer returns an integer, so 5200/5312.35 = 0. To prevent this, put a period at the end of the whole number: 5200./ 5312.35 = 0.97885. Below, `foo` is an integer while `bar` is a floating point number. You can also explicitly make a value a floating point value.

```
In [1]: foo = 1

In [2]: type(foo)
Out[2]: int

In [3]: bar = 1.

In [4]: type(bar)
Out[4]: float
```

I. Making a mask

Scripts: statistics_nouv.py, masking.py, buildamask.py, i_nouv.py

a. run statistics_nouv.py on a vacuum run

This script is meant for quickly collecting data from runs, and saving it in an hdf5 file. The data collected are:

```
front_intensity: the integrated intensity of the cspad
diode1_intensity: the channel 1 diode reading
diode2_intensity: the channel 3 diode reading
sample_pressure: the sample line manometer reading
xray_energy: the fee detector reading
xray_front: the summed cspad images for xray-on shots
xray_shots: the number of xray-on shots
dark_front: the summed cspad images for xray-off shots
dark_shots: the number of xray-off shots
```

sidenote: running i_nouv.py allows you to see histograms of the point data.

b. Run buildamask.py twice.

This script calls masking.py and allows you to build a mask! You want to change the `filename` variable to the path and hdf5 file you made with statistics_nouv.py

The first time you run it, `whichmask` should be set to `'dark'`, which allows you to make your initial mask from the xray-off shots. The script will show you a histogram of the pixel values of the averaged xray-off shots (`dark_front/dark_shots`), which should be one main peak centered at 0, and some outliers. Look closely at this and decide what you want your thresholds to be for the mask. The script will then ask for those lower and upper bounds for you to fill in. From there, it builds a mask and saves it. Loose values for the lower and upper bounds would be -2 and 2, while stricter ones are -1 and 1.

The second time you run it, set `whichmask` to `'xray'`, which allows you to make a mask *on top of* the mask you just built using the dark shots, but this time using the averaged xray-on images (`xray_front/xray_shots`).

WARNING: This method is only good for making a *basic* mask with a **vacuum image** only. It's mostly based on looking at threshold pixel values (hot and dead pixels), which doesn't work well if you're imaging non-vacuum. For LO92, I had to go in and manually mask pixels using uv-off SF6/TMA, and comparing to with IAM, to remove the Pt rings on the detector. It was essentially an iteration between geometry optimization and masking pixels with certain q values.

Note: These scripts make reference to 'front' and 'back' in some of the variable names and options – this is because I used older versions in LO92, when we had both a front and back cspad. If it really bothers you, you can change the variable names for your own personal use.

II. Run statistics on the data

Scripts: ana_stats.py, ana_pixels.py, combine_stats.py combine_pixels.py

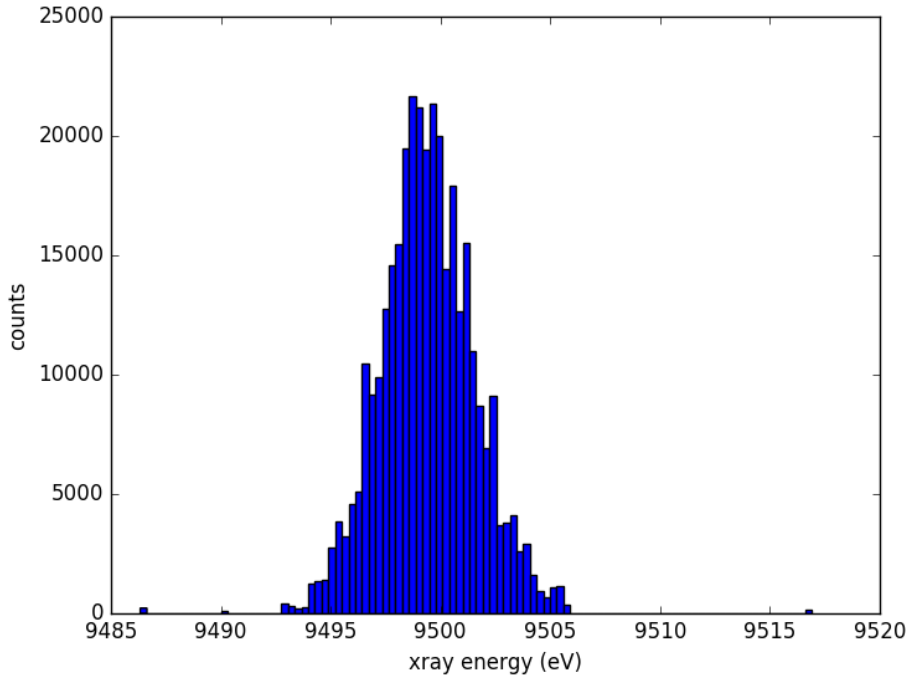
a. Ana_stats.py: This script gathers point data for each shot, so you can look at histograms and trends in the data.

For all data, the ana_stats.py script collects the xray energies (from the FEE), the sample pressures, the diode intensity, and the cspad intensity. For runs before run 84, the EVR alias should be 'evr0' not 'evr1'

For the longtime data, we didn't use the time tool, so we don't need timetool output. We did however get bucket jumps. We can check for bucket jumps using the uv arrival location on the acqiris. So for each uv-on shot, uv arrival time is taken, along with the corresponding uv delay.

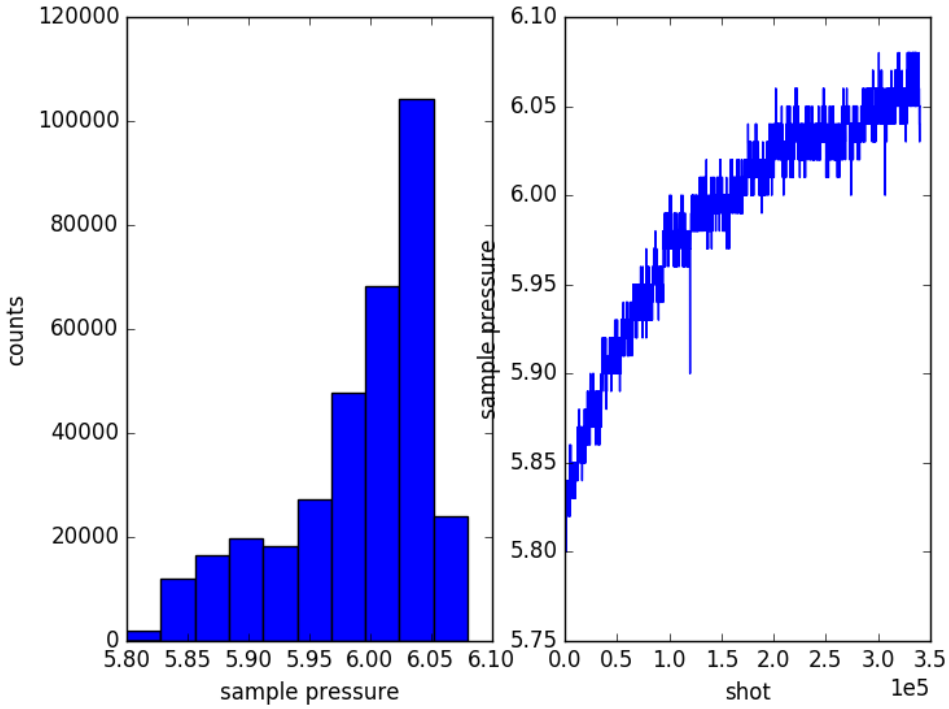
For the shorttime data, the timetool stats are also taken. I suggest reading the psana documentation on this. Make sure you input appropriate timebinmin and timebinmax (min and max times, in ps, for cutoffs of timetool data.)

After running ana_stats.py, run combine_stats.py. Here are what the results should look like. A histogram of xray energies:

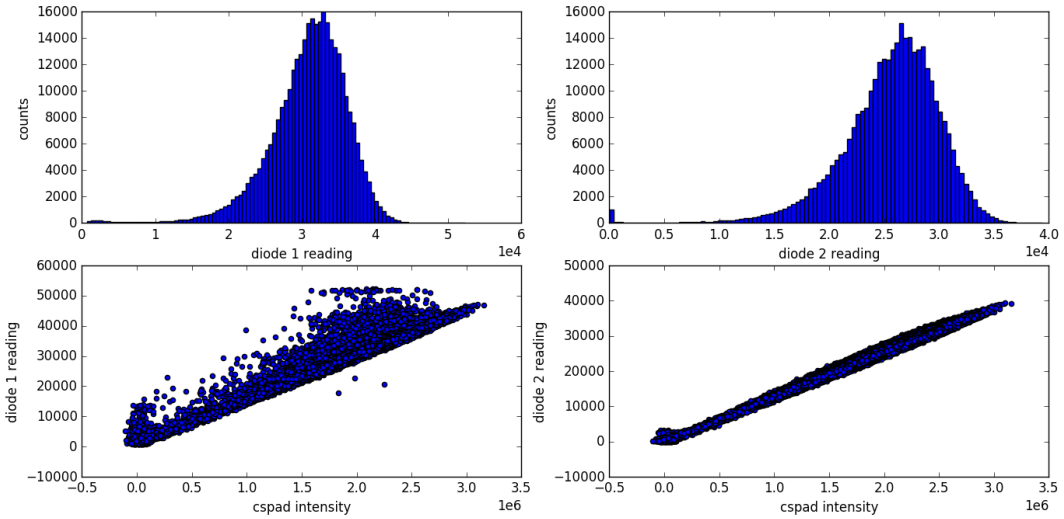


For the FEE I use the Median Absolute Deviation to determine thresholds – so for this it's at 9495.3 eV and 9503.4 eV. In future analysis, any shots with xray energies outside those bounds will be discarded.

A histogram of sample pressures, and also a plot of the pressures over time:

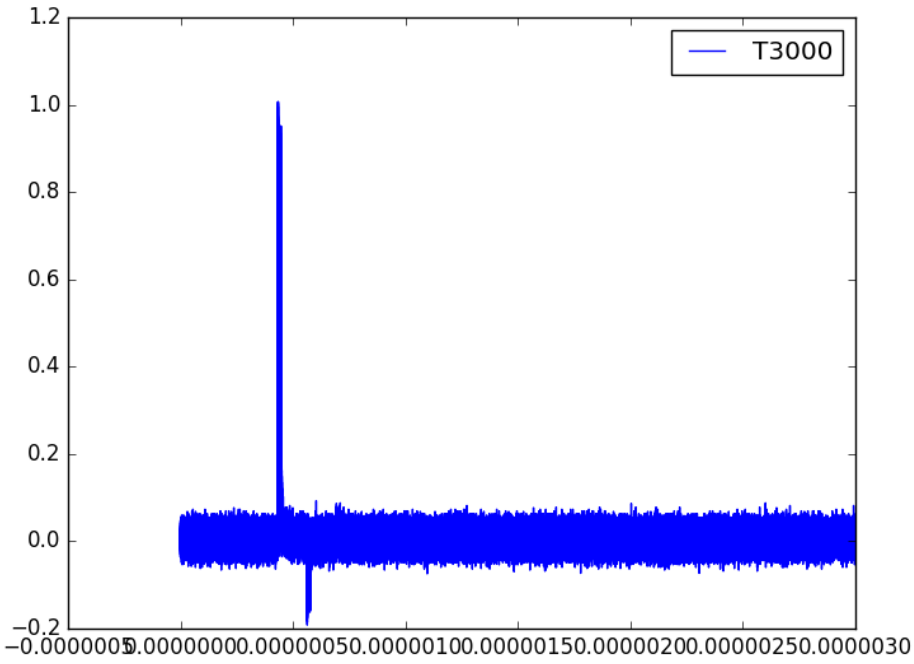


And histograms and scatterplots of diode and cspad intensities. In general it looks like the second diode reading is better:

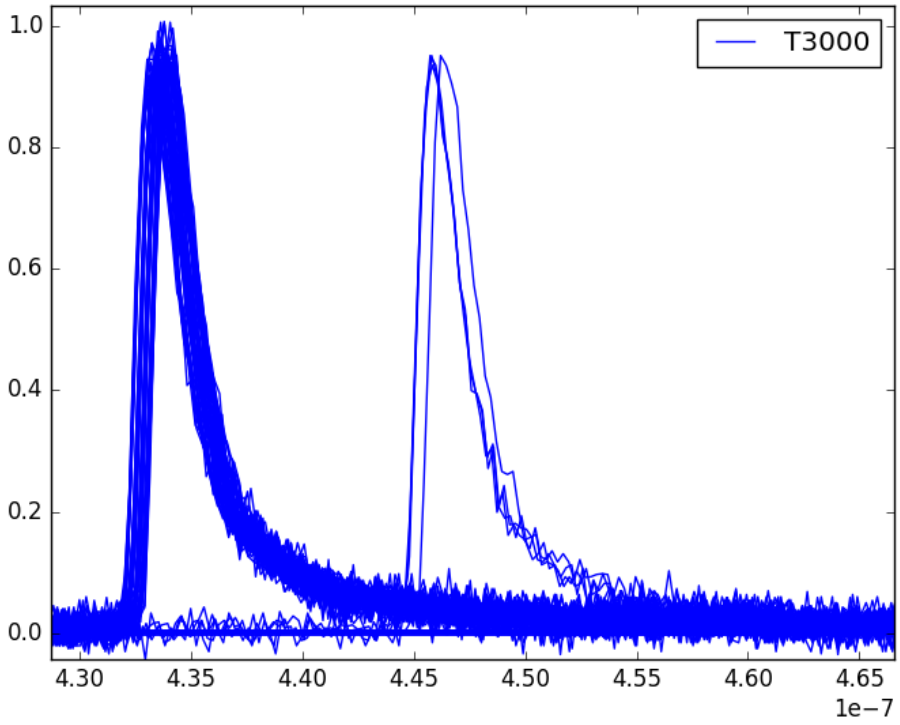


For later analysis, a lower bound on the diode intensity may be used. We've found that a fairly strict bound works better. (A stricter bound would be 22000 ADU as opposed to 17000 ADU)

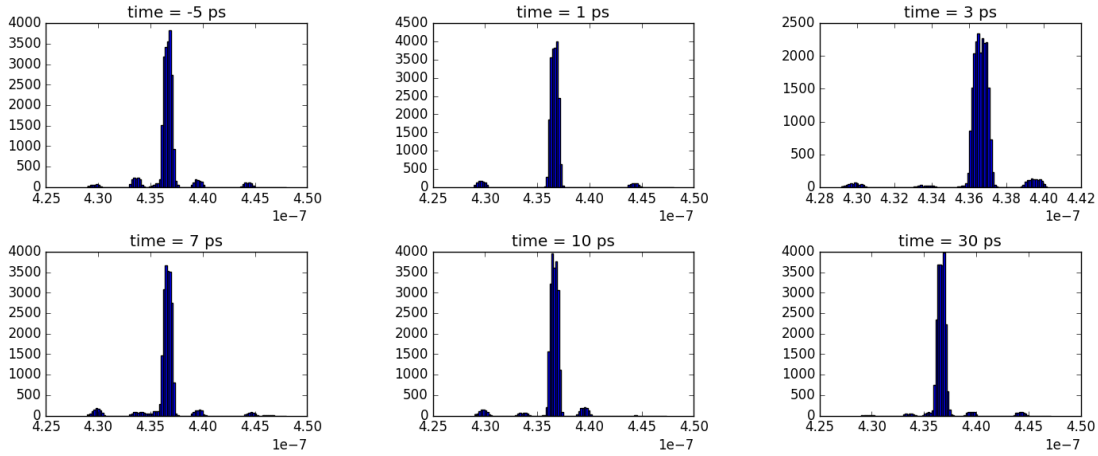
FOR LONGTIME DATA:
If you take the acqiris data and plot channel #3 like this, and plot the data, you can see the UV trace. Here are a bunch plotted for the delay setting of 3000 ps:



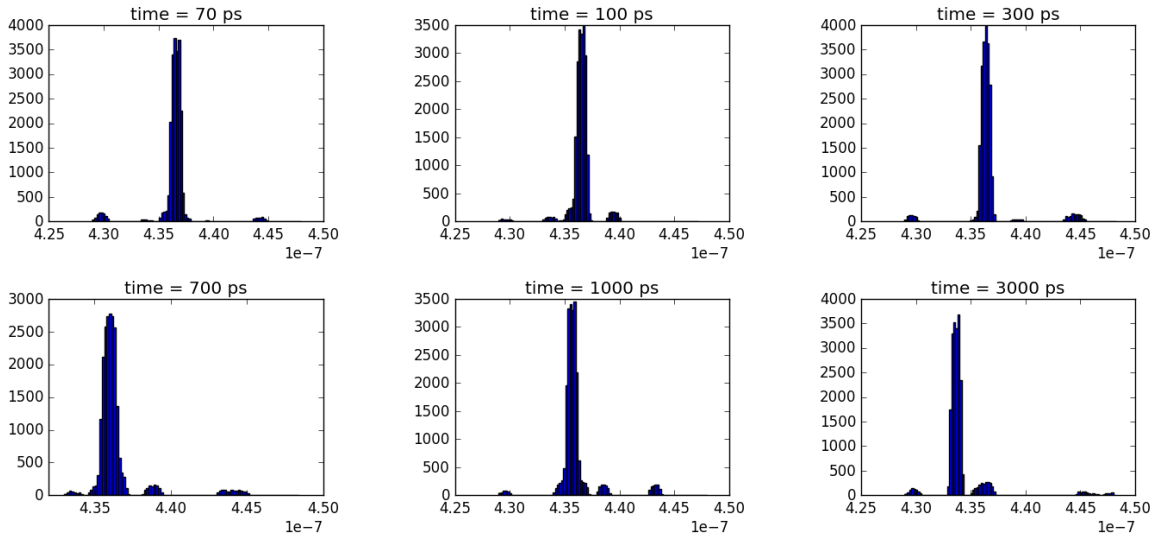
The peaks all appear to overlap, but if you zoom in, you see this:



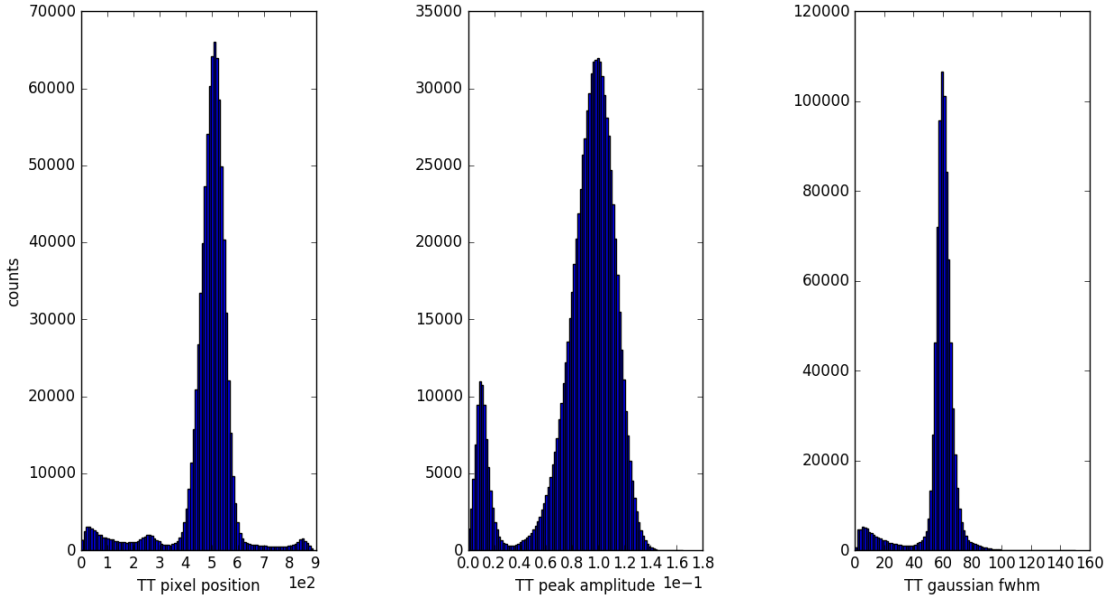
Some of the shots are clearly at a different real delay, despite all being set to 3000 ps on the uv delay. These are the bucket jumped shots. The ana_stats.py collects the location of the peak for each shot, and its corresponding time delay. Then it histograms the peak locations for each delay setting:



The main peaks are the good ones, and the other peaks need to be removed from further data analysis. As you can see, for short times the peaks tend to overlap, but for larger delay settings, the peaks actually start to shift to lower values:

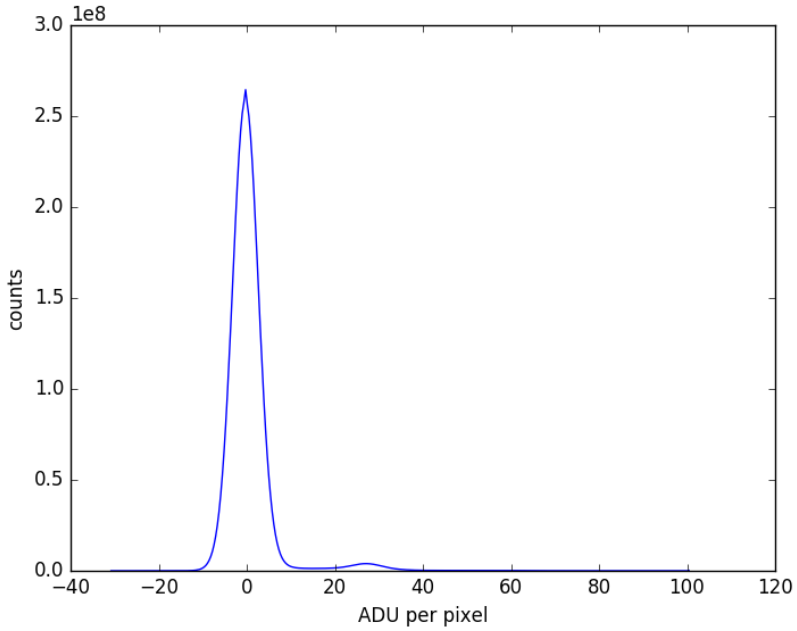


FOR SHORTTIME DATA: the combine_stats.py script should output some histograms of the timetool data

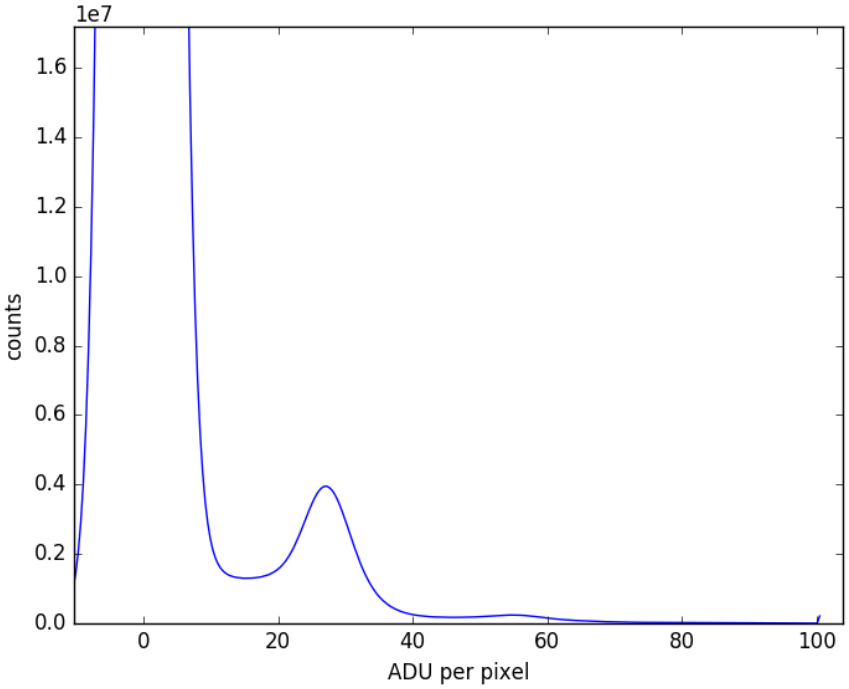


b. Ana_pixels.py: This script gathers the histograms of the cspad images – the adu per pixel. Because these histograms should only be affected by the xray energy and intensity, I only consider the fee and diode thresholds in this script and they are identical for both longtime and shorttime data. Because the xray intensity is so important to our data, I put a strict diode threshold when running this script, though in later analysis it may be ok to loosen it. Also I think taking only a few shots is probably plenty for getting a good idea of the adu/pixel, but I run this on all the runs anyway. I also only look at ADU values in the range (-30 to 100), because that seems good for the xray energies we look at. For LCLSII-HE, that range may need to be shifted or increased to account for the higher xray energies. Also, they may have a completely different and updated PSANA library for LCLSII!

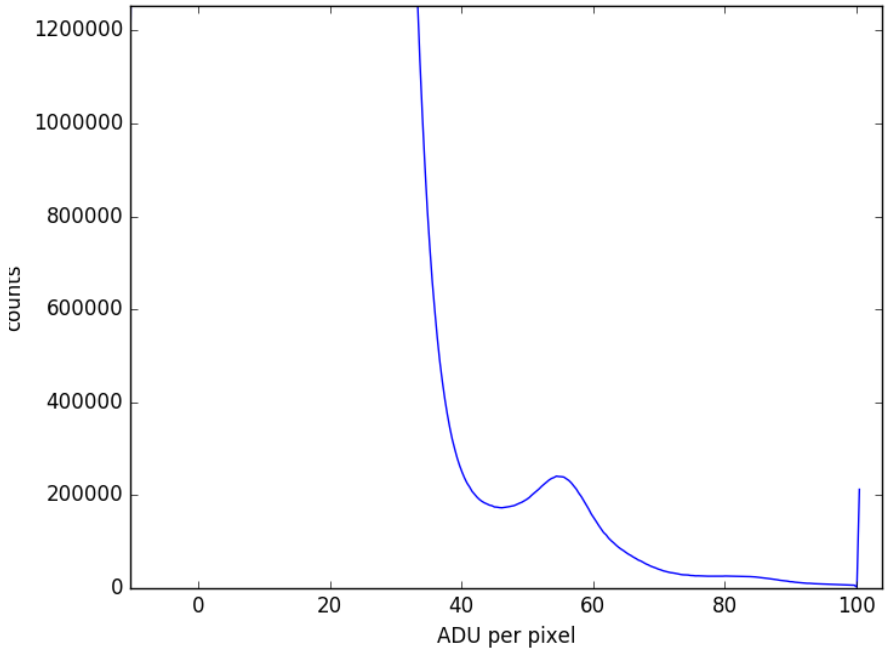
Here's what the histogram looks like:



Unfortunately, that giant peak is the zero-photon peak. None of those pixels are reading anything of use. If you zoom in you can see the one-photon peak centered at ~25:



For future analysis, I'll use a cutoff of ~16 ADU, so that the zero-peak isn't included.
If you zoom in further you can see the 2- and 3-photon peaks:



III. Time-bin the data

a. Ana_tbin_batches.py

These scripts take the data and analyze them in batches of size `batch_size`, so that any pressure fluctuations can be accounted for. We don't depend on the pressure gauge for this as we aren't sure how much the pressure reading is real and how much is electrical noise.

For both longtime and shorttime data, you need to specify what bounds you want to put on the various parameters, and also the time bins.

FOR LONGTIME DATA: Starting at Line 72, in the time-binning part of the script, you'll see this:

```
if time_delay < 31: # time bins -5 to 30 ps
    if (uv_time < 4.35e-7) or (uv_time > 4.38e-7): continue
elif time_delay < 101: # 70, 100 ps
    if (uv_time < 4.347e-7) or (uv_time > 4.38e-7): continue
elif time_delay < 701: #300, 700 ps
    if (uv_time < 4.347e-7) or (uv_time > 4.376e-7): continue
elif time_delay < 1001: #1000 ps
    if (uv_time < 4.336e-7) or (uv_time > 4.375e-7): continue
elif time_delay < 3001: #3000 ps
    if (uv_time < 4.325e-7) or (uv_time > 4.347e-7): continue
else: #7000 ps
    if (uv_time > 4.31e-7): continue
```

The `uv_time` variable is the uv delay as seen on the Acqiris, `time_delay` is the uv delay as seen by the laser delay. These lines are where I applied vets for the bucket-jumped shots. To make sure that the shot is actually in the `time_delay` timeslice, it has to be within the bounds I set for the Acqiris arrival time. These lines need to be changed to fit your own time-binning needs.

FOR SHORTTIME DATA: There are two ways to set up the time bins:

```
## TIMEBINNING
timebinmin, timebinmax = -1, 2
timebinsize = 0.025
time_bins = np.arange(timebinmin, timebinmax + timebinsize, timebinsize) #
#time_bins = np.hstack((np.arange(-1.2, -0.5, 0.04),
#        np.arange(-0.52, 2, 0.025),
#        np.arange(0.5, 1.5, 0.04),
#        np.arange(1.5, 3.01, 0.06)))
#timebinmin, timebinmax = time_bins.min(), time_bins.max()
#timebinsize = time_bins[1] - time_bins[0]
```

The un-commented lines do simple equally spaced time bins from -1 to 2 ps.
The commented lines allow you to switch up the time bin spacing. It's important to have `timebinmin`, `timebinmax`, and `timebinsize` to be specified so shots outside of the time range can be vetoed. Because the `time_bins` are upper bounds, to make the first time-bin the size of `timebinsize`, all shots time delays less than `timebinmin – timebinsize` are vetoed, so that the first time bin is [`timebinmin – timebinsize, timebinmin`]. If this script seems to get 'stuck' running on a node or quits on you suddenly, you may have too many time bins, and it's too much for the memory. You can also demand more memory using the following command:

```
>> bjobs –q [QUEUE] –o [OUTPUT FILE] –R "rusage[mem=MEMORY]" python [SCRIPT]
```

b. Combine_offs.py, combine_batches.py

This script takes all your saved files of batched data, and averages the uv-off shots. This average is used to set the `uvoff_average` variable in the combine_batches.py script. Each batch of shots will be scaled to this average.
The combine_batches.py script takes all the saved batches and scales and averages them. It then saves this file. This process can take a while depending on how many batches you have. Another variable to set is `num_batches = 23517/batch_size + 1`, the 23517 comes from the longest run I am looking at. You can find it in the elog:

DAQ Detector Totals

Number of damaged events	44	Damaged
Total number of events	23517	Events
Amount of data recorded [GB]	116.068	Size

WARNING: For these scripts, if a file is not there to be opened, it simply prints the run number and the batch number of the file that could not be opened. This means there is **no error message**. If you think that that file should exist, then makes sure the file path is correct, or check if there's an error message for the ana_tbin_batches.py script for that run.

c. Geometry_optimize.py, geometry_image.py

This file takes the uv-off shots that you saved to the one file, and optimized the detector geometry so that using a least-squares fit to a theoretical pattern. You basically just have to change the file paths to the `theory`, `mask`, and `data`. If SLSQP doesn't work, you can change the `method` parameter or the `x0` initial guesses. Also make sure the xray wavelength is accurate (I don't think it changes much throughout the experiment anyway but it's good to check). It

usually takes 10-15 minutes to run, so either do it as a batch job or expect to wait. When it's done, look for "success=True" in the output, if it says False, then the optimization did not work. The optimized parameters are the 'x' array:

```
 success: True
       x: array([  8.93237364e+02,  -2.24028423e+02,   8.62315972e+04,
         9.06806988e-02,   8.35487644e-01])
```

The x array returns the parameters in this order: the x offset, y offset, detector distance, (all in microns) and phi offset (in radians), and a scaling factor to scale the experiment to the theory (this value ultimately doesn't matter) – these end up making the `x0, y0,distance`, and `phi0` values in the combine_batches.py script. The geometry_image.py script takes the optimized parameters (`best_values`)and plots the theory and the experiment so you can see if they actually match up.

d. Error_propagation.py

This script takes the saved file from combine_batches.py, the optimized parameters from geometry_optimize.py, and the saved file from geometry_image.py, and gives you the final results: radially averaged, angularly averaged, the errors, and the q-and-phi-binned data. The error analysis is based off of the number of photons.

IV. Visualizing the data

If you prefer using Matlab, you can save all the data in .mat files (rather than .npz like I have in the scripts) and then export it to matlab using the following:

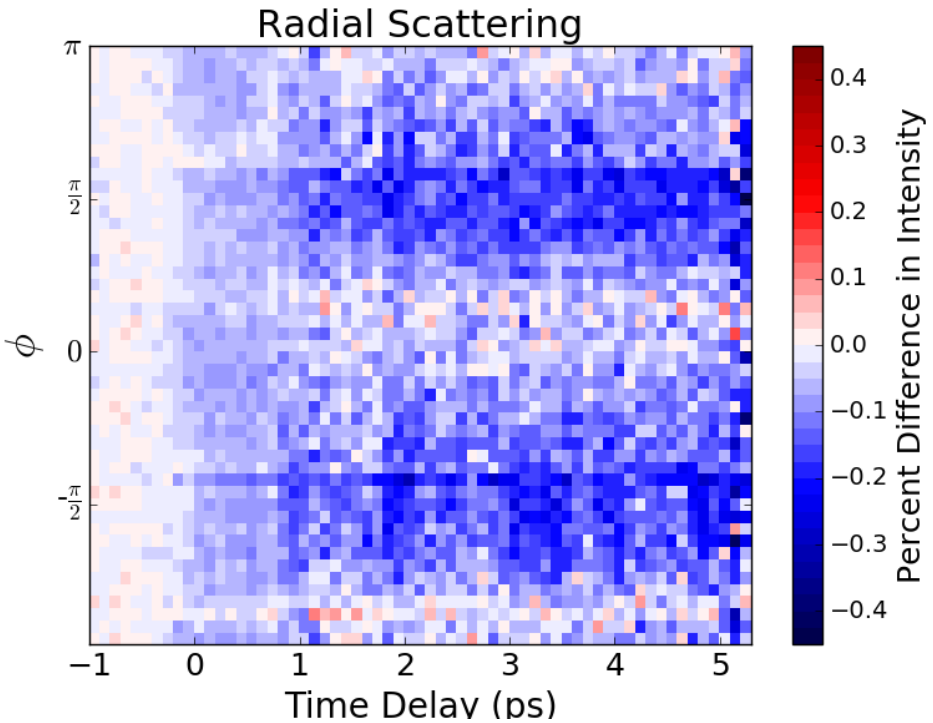
```
import scipy.io as sio

sio.savemat('mydata.mat', {'q_bins':q_bins, 'uvon_radial_scattering':rads, 'uvoff_radial_scattering' : radoffs,
    'radial_pdiffs' : radpdiffs, 'phi_bins' :phi_bins, 'angular_pdiffs': angpdiffs,
    'binned_pdiffs' : ibpdiffs,
    'time_bins':time_bins, 'cspad_ons':ons, 'cspad_offs':offs, 'shots_per_bin':onshots})
```

Brian has good Matlab scripts for this, probably.

If you'd rather use Python, I have a script for making contour plots (contourplot.py) and a script for making gifs (makeagif.py). The gifs require you to install the imageio library.
To do that, from whatever directory you would like to install the library do the following:

```
>> git clone https://github.com/imageio/imageio.git
>> cd imageio-2.2.0/
>> python setup.py install --user
```

For either, if you have access to the LO92 experiment you can run them as is and look at the output for yourself. Here's what you get for the contourplot.py script:



I set up the colorbar so it is white at 0 and diverges to red and blue. Another interesting thing I do here is use greek letters for the phi angle tick marks. For the most part you just set the x,y,z, and plot formatting, and let it run.