1.2 Area Detector treatment with DetObject

In order to keep the filesizes small to avoid issue when analysis the smallData files, we try to extract the important information gleaned from areaDetectors at an event-by-event basis and only save these pieces of data.

Analysis functions

The following functions are readily available for use in smalldata_tools:

- ROI
- Azimuthal integration
- Droplet
- Projection
- Autocorrelation

The funtions definition can be found in `<smalldata_tools>/smalldata_tools/ana_funcs`, in case one want to take a close look at how the data are processed. These analysis functions are sub-classes of `DetObjectFunc` and the computation are done in the custom `process` method defined in each subclass.

Comments on the masks for area detectors

Each area detector loaded in smalldata_tools has the following psana masks defined:

- `self.statusMask = self.det.mask(self.run, status=True)`: only accounts for the pixel status
- `self.mask = self.det.mask(self.run, unbond=True, unbondnbrs=True, status=True,  edges=True, central=True)`: accounts for the geometry pixels
- `self.cmask = self.det.mask(self.run, unbond=True, unbondnbrs=True, status=True,  edges=True, central=True, calib=True)`: generally the mask that fits most needs

Definitions of the psana mask's options:

- calib : bool - True/False = on/off mask from calib directory.
- status : bool - True/False = on/off mask generated from calib pixel_status.
- edges : bool - True/False = on/off mask of edges.
- central : bool - True/False = on/off mask of two central columns.
- unbond : bool - True/False = on/off mask of unbonded pixels.
- unbondnbrs : bool - True/False = on/off mask of unbonded pixel with four neighbors.
- unbondnbrs8: bool - True/False = on/off mask of unbonded pixel with eight neighbors.

Detector configuration data

When a DetObject has been declared, information used to extract the data will also be stored in the hdf5 file in the UserDataCfg dataset, among other things we store:

- pedestal (measured in a dark run)
- noise (measured in a dark run)
- gain (if applicable)
- geometry arrays (x/y/z positions for each pixel)
- mask (see above)

Common mode

The different common modes are described here

Interactive SmallDataAna session

For the use of the interactive features of SmallDataAna_psana, we recommend to start it in an ipython session as interactive grabbing of user input is currently not implemented via the notebook.

Creating an average image

In order to decide on the proper ROI, fit a team center or make a mask, the first step is always to create an image that you would like to use as base. This is achieved using the following function in the python terminal opened from:

`/sdf/data/lcls/ds/<hutch>/<expname>/results/smalldata_tools` via:

```
./producers/runSmallDataAna -r <#> [-e <expname>]
```

The following command will create an average image of 100 events of an area detector with a set of default parameters:

```
SDAna In: anaps.AvImage()
```

This will by default create an average image of 100 events of an area detector. The method will list and prompt for a detector name if more than one area detector is found in the dataset.

The full command with all its options can be looked via using the "?" feature of iPython:

```
SDAna In: anaps.AvImage?
```

If you would like to take a quick look at your average image before proceeding, call:

```
SDAna In: anaps.plotAvImage()
```

The SmallDataAna session can typically be used to help setting the parameters for the smalldata production.

Deprecated

Typical forms of userData

SmallDataProducer_userData.py has examples for the most standard requests. Parameters that are important for the production (ROI boundaries, centers for azimuthal projections,...) are stored in the "UserDataCfg" subdirectory.

A most current example of a producer file that show how to add both "pre packaged" feature extraction as well as more free form user data can be found in github:
https://github.com/slac-lcls/smalldata_tools/blob/master/examples/SmallDataProducer_userData.py
Each of the supported feature extraction/data reduction mechanism is described in their own subpages.
The common structure of adding user data is the addition of a "DetObject" for each detector you would like to extract information from to the producer python file. The relevant lines look like this:

```
    azIntParams = getAzIntParams(run)
    ROIs = getROIs(run)
    detnames=['epix10k135', 'epix10k2M']
    for detname in detnames:
        havedet = checkDet(ds.env(), detname)
        if havedet:
            common_mode=84
            if detname=='epix10k135': common_mode=80
            det = DetObject(detname ,ds.env(), int(run), common_mode=common_mode)
            #check for ROIs:
            if detname in ROIs:
                for iROI,ROI in enumerate(ROIs[detname]):
                    det.addFunc(ROIFunc(ROI=ROI, name='ROI_%d'%iROI))

            if detname in azIntParams:
                azint_params = azIntParams[detname]
                if 'center' in azIntParams[detname]:
                    try:
                        azav = azimuthalBinning(center=azint_params['center'],
                        dis_to_sam=azint_params['dis_to_sam'], phiBins=11,
                        Pplane=0, eBeam=azint_params['eBeam'],qbin=0.015)
                        det.addFunc(azav)
                    except:
                        print('could not define azimuthal integrating for %s'%detname)
                        pass

            det.storeSum(sumAlgo='calib')
            dets.append(det)

    return dets
```

'epix10k135' is a name given to the detector in the DAQ (called alias). You can get a list of detectors in your data by using the psana command "detnames exp=<expname>:run=<run#>" in a terminal.
The checkDet/if structure allows detectors to be only processed when they appear. We loop over the list of DetObjects in the main event loop and for each detector, we get the data and attach it to the detector object. This is followed by a loop through the list of data extraction methods that have been added in the code block above.
Only the first three parameters in the DetObject definition are necessary. The name will default to the alias of the detector. The common_mode will default to the typically best one. The currently recommended methods are listed below:

- CsPad, cs140k: of all tiles of detector will exhibit a zero-photon peak, use common_mode=1, otherwise we recommend no correction (common_mode=0). It is also possible to request the use of the unbounded pixels by using common_mode=5
- Opal, Zyla: no common-mode correction. Should a pedestal of the correct shape have been produced, then it will be subtracted. common_mode=-1 will return the raw data.
- Jungfrau: no common_mode correction is the default we now suggest to use common_mode=7
- Epix: we are using a method called "common_mode=46" which removed pixels with a lot of signal (10x noise) and the neighbors. Then, the common_mode is calculated for reach row & column and then subtracted. This is similar to method 7 and different will be tested hopefully soon.

The common mode corrections which are wrapped by DetObject are described on this page: Common mode correction algorithms.

When a DetObject has been declared, information used to extract the data will also be stored in the hdf5 file, among other things we store:

- pedestal (measured in a dark run)
- noise (measured in a dark run)
- gain (if applicable)
- geometry arrays (x/y/z positions for each pixel)

As mentioned above, to actually have event based data for your detector in the hdf5 file, you should add reduction/feature extraction methods to your detector. Several of them have been setup to allow for easy addition without any need to write own code. This and the tools to help you set the different algorithms up are described in the child pages listed at the bottom of this page. While it is possible to save the full image (using the ROI mechanism), this will result in big hdf5 files, possibly posing problems with memory management when these files are analyzed later. Full images are ideally only stood using the "cube/binned data" mechanism.

If you have needs that are not met by this, you can add your own code in the event loop and use the <det>.evt.dat array that has been created as input. You can then add your results as python dict to smldata, which will save it to the hdf5 file. This also allows to combine information from two detectors for your feature extraction. Unless you need to deal with big data (full images,...) or have very computationally expensive algorithms, I would recommend store simpler data in the first level hdf5 file and run the second level processing outside as described in 4. SmallData Analysis to Cube Production