AGIPD 1.1 calibration software documentation

Abstract

This document describes the software for extracting the calibration constants from measurement data and for equalizing measurements using these constants.

Installation

Download the software from https://stash.desy.de/users/gevorkov/repos/agipdcalibration/browse e.g. via git clone https://stash.desy.de/scm/~gevorkov/agipdcalibration.git

To run the code, you need the latest version of Anaconda for Python 3. On the Maxwell cluster you can get it with the following command:

```
source /gpfs/cfel/cxi/common/cfelsoft-rh7/setup.sh
module load cfel-anaconda/py3-4.3.0
```

Define the environment variable AGIPDCALIBRATION_INSTALLATION_DIR to be the path to the root directory of the downloaded code. E.g.

```
export AGIPDCALIBRATION_INSTALLATION_DIR=/home/sampleUser/agipdCalibrationCode/
```

Source the file setup.sh from the root directory of the downloaded code.

The algorithms need (dependent on the size of the input data) a lot of RAM. A minimum of 256GB RAM is recommended. If possible, use the cores with 512GB (they are fastest anyway). Do not run scripts in parallel on one node, since you might hit the RAM limit this way. The scripts parallelize the processing internally. Running scripts in parallel (even if not hitting the RAM limit) will typically slow down the computation.

Workflow

The basic workflow can be seen in Figure 1. All the algorithms work on single modules.

The basic steps are as follows:

1. Calibration data is gathered, i.e. translated into a minimalistic, procession friendly format. If the data arrangement is changed, only these files have to be adapted, no change to the rest of the code has to be done. The relevant scripts start with “gather”.
2. The gathered data is processed by the scripts starting with “batchProcess”.
3. Processed data is collected into two files:
    a. combineCalibrationConstants.py gathers all the data that is needed to equalize the data

   b. createMask.py creates a bad pixel mask by the aid of the user
4. Finally the equalization can be done. The function equalizeRawData_oneBurst() takes a burst of data (352 images) and corrects it. The mask can be used to set masked cells to a specific value or can just be handed on to the user of the equalized data.

Gather scripts

All gather scripts save the data in the latest HDF5 format available in the Python version used. All data is saved in the root folder.

gatherXRayTubeData.py

The X-ray tube data is taken for one cell only. It is not chunked, so the entire data has to fit in RAM.

1. “/analog”: shape=(dataCount, 128, 512)
2. “/digital”: shape=(dataCount, 128, 512)

gatherCurrentSourceScanData.py

The current source data is saved chunked to allow fast access to subsets of the data. This ability is used in the batch processing scripts to reduce the memory requirements.

1. "/analog": shape=(dataCount, 352, 128, 512), chunks=(dataCount, 352, 64, 64)
2. "/digital": shape=(dataCount, 352, 128, 512), chunks=(dataCount, 352, 64, 64)

gatherDarkData.py

The dark data is not chunked, so the entire data has to fit in RAM.

1. “/analog”: shape=(dataCount, 352, 128, 512)
2. “/digital”: shape=(dataCount, 352, 128, 512)

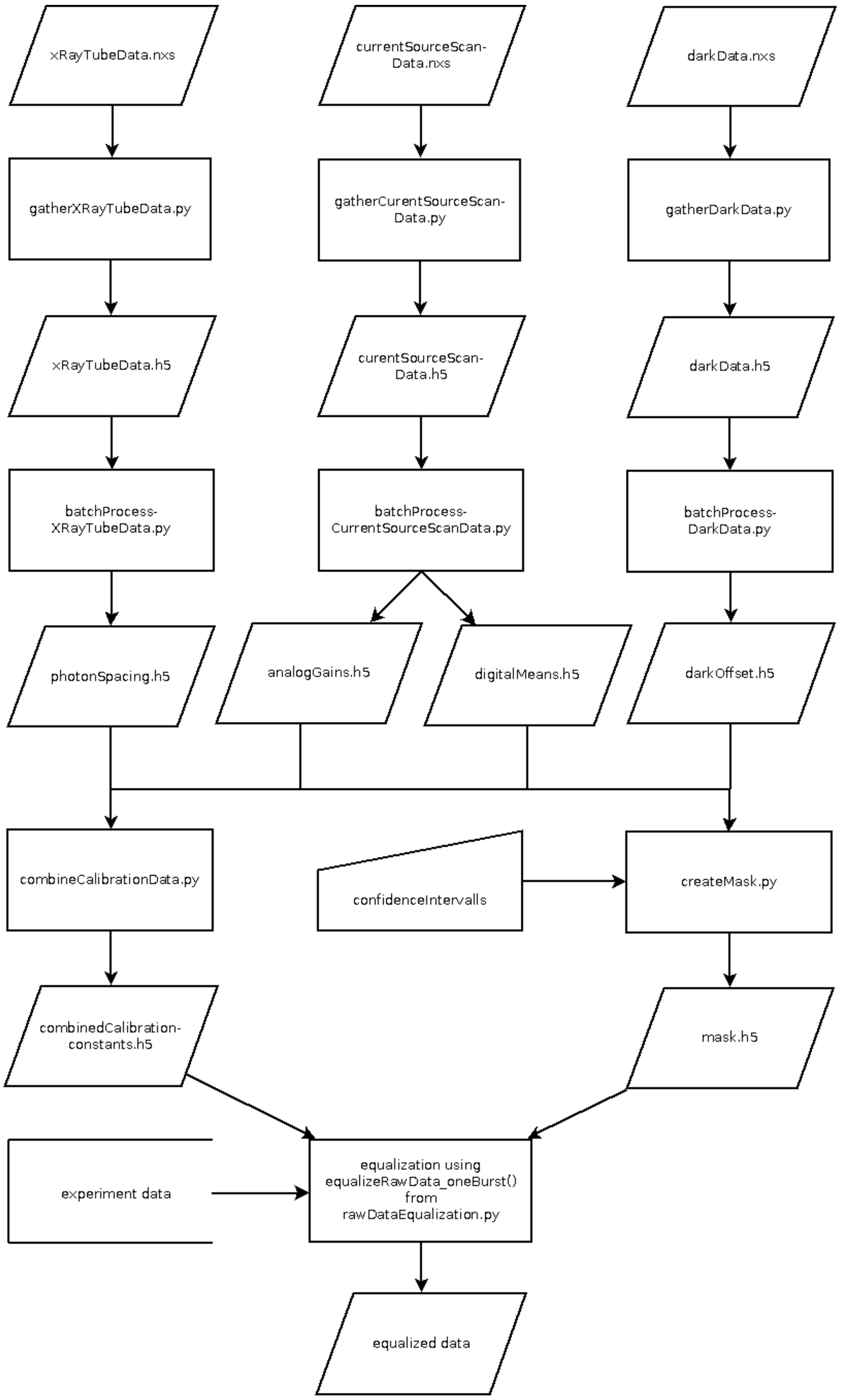

Figure 1: basic calibration workflow

Batch process scripts

batchProcessXRayTubeData.py

This script parallelizes the application of the function getOnePhotonAdcCountsXRayTubeData() to all the data.

The algorithm tries to identify the ADC count per photon. It does so by finding peaks in the histogram of the data. Test data used for developing this algorithm showed a strong drift over time. This drift is removed by subtraction of a lowpass. The lowpass is created by a (custom and fast) running median filter, see Figure 2.

The algorithm identifies peaks using the function find_peaks_cwt() from the scipy library. The ADC count per photon is computed using the distance between the two highest peaks in the histogram.

As a quality measure, the “valley depth” of the smoothened Histogram between the two highest peaks is taken.

The output file contains the following datasets:

1. "/photonSpacing": shape=(128, 512)
2. "/quality": shape=(128, 512)

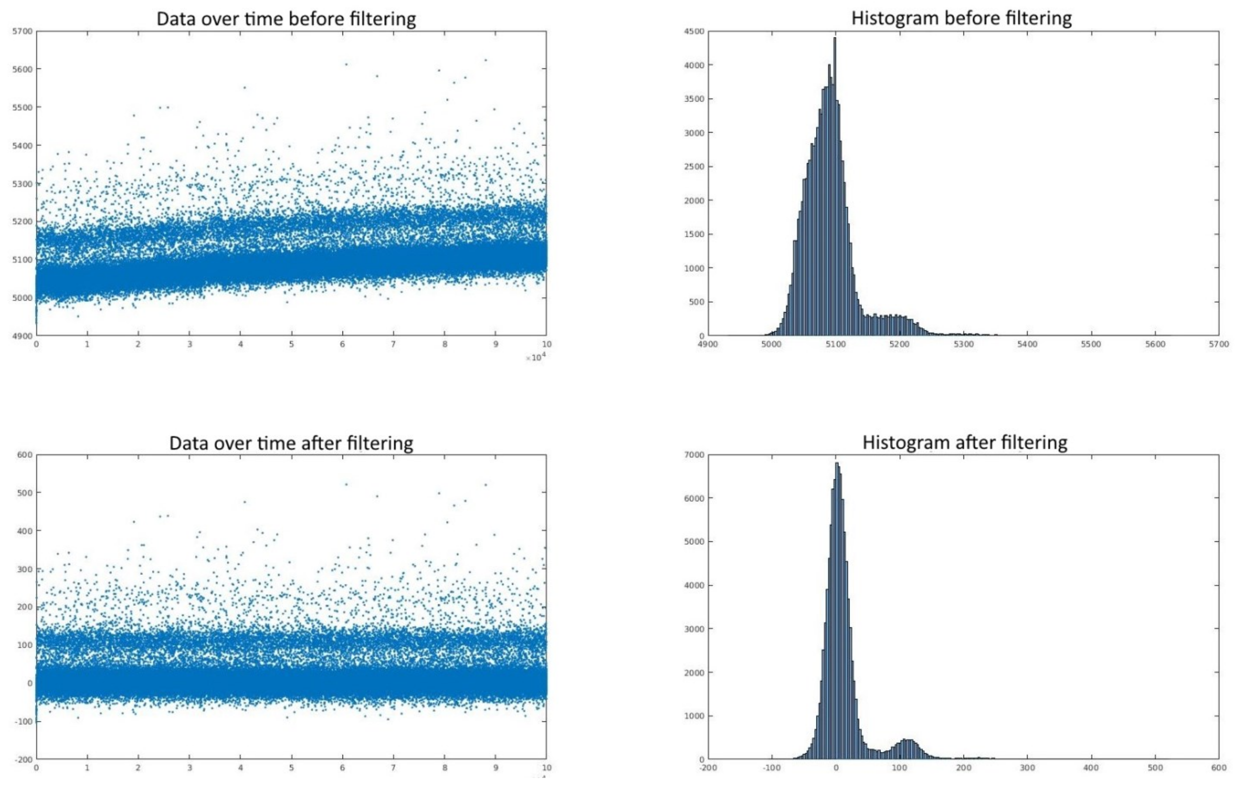

Figure 2: Drift correction for histogramming X-ray tube data

batchProcessCurrentSourceScanData.py

This script parallelizes the application of the function fit3DynamicScanSlopes() to all the data.

The algorithm works in two steps:

1. Fit mean digital values of the three gain stages
2. Fit a line to every gain stage

Since this script takes several hours to complete, optimization is an important issue here. This is the reason, why instead of the easy-to-use function find_peaks_cwt() from the scipy library a fast simplified adaptation of the k-means algorithm is used to determine the digital values of the gain stages.

The analog data is filtered to remove spikes above a hardcoded width (maxSpikeWidth = 2) and height (minSpkieHeigh = 300). Only spike values are changed, the rest of the data remains the same.

From the digital means, the analog values belonging to a gain stage are identified. Parts close to the borders are neglected (hardcoded constant linearFitDataShrinkFactor=0.3) to avoid fitting the line to nonlinearities. For the center part a least squares approach is used to fit the line. For an example fit see Figure 3.

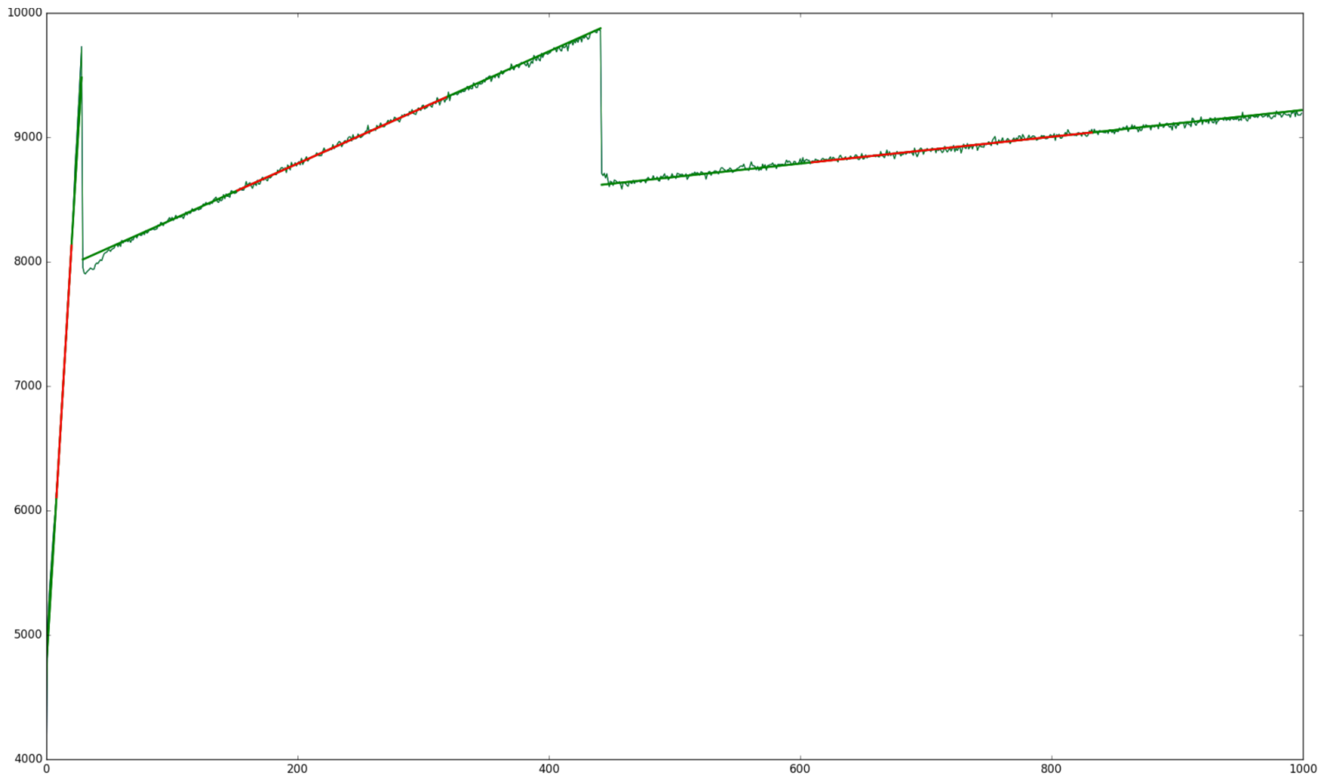

Figure 3: Current source line fitting. Data in the red region is used to fit the line that is extrapolated to the green region. Every gain stage has its own line

As a quality measurement for digital and analog, the standard deviation of the data in the region that is used for linear fitting is computed.

The thresholds for the digital values are computed as means of the respective two digital means. As a quality measure for a threshold, the spacing between the respective digital values divided by the sum of their standard deviations is used. This value is called “digitalSpacingsSafetyFactor”.

The output files contain the following datasets:

Analog gains file:

1. "/analogGains": shape=(3, 352, 128, 512)
2. "/anlogLineOffsets", shape=(3, 352, 128, 512)
3. "/analogFitStdDevs", shape=(3, 352, 128, 512)

Digital means file:

1. "/digitalMeans", shape=(352, 3, 128, 512)
2. "/digitalThresholds", shape=(2, 352, 128, 512)
3. "/digitalStdDeviations", shape=(352, 3, 128, 512)
4. "/digitalSpacingsSafetyFactors", shape=(352, 2, 128, 512)

batchProcessDarkData.py

This script computes mean and standard deviation of the dark data for every cell.

The output files contain the following datasets:

1. "/darkOffset": shape=(352, 128, 512)
2. "/darkStandardDeviation": shape=(352, 128, 512)

Combining necessary constants for equalization

The script combineCalibrationData.py combines the constants needed for equalization in one file.

The analog gains given in $ADC/integrationTime$ are translated into $\mathrm{keV}/\mathrm{ADC}$ /. For this, the photonSpacing data computed by batchProcessXRayTubeData.py is used.

For the high gain dark offset the darkOffset produced by batchProcessDarkData.py is used. For the medium and low gain offset the anlogLineOffsets produced by batchProcessCurrentSourceScanData.py are used. Tests need to show, whether the latter two are trustworthy.

The digital thresholds are taken as-is from digitalThresholds produced by batchProcessCurrentSourceScanData.py. They should not be needed this way in the final version, because then global thresholds are planned to be used.

Mask creation

Mask creation is a very touchy business. It is highly dependent on the user and affects the produced data very much. This is the reason, why here user interaction is necessary. The provided script is meant to be edited by the user, while giving some guidelines.

The user is expected to provide a manual mask (e.g. for beam stop or shadows) and a systematic mask (e.g. masking the 32-tips bug in the cells). A template for masking the asic borders is also provided.

The then following code is meant to be unchanged in the typical cases. The user is shown histograms of the computed values and is asked to deliver confidence intervals. Cells outside these intervals are masked. When creating the histograms, the already masked cells from previous stages are ignored. This way, cleaner histograms are achieved.

This method of creating a mask is not perfect, but it gives an entry point for the complicated task of mask creation for such a large detector.

The generated file contains the following dataset:

1. "/badCellMask", shape=(352, 128, 512)

Equalization of the data

The final aim of the calibration is to enable equalization (i.e. correction) of the data. The equalization can be done with the function equalizeRawData_oneBurst() from the file rawDataEqualization.py.

This can be done in real-time or with a script that loads the data from disc. An example script is provided in equalizeAgipdData.py. The script equalizes a data file with the datasets “/analog” and “/digital” with shape=(dataCount, 352, 128, 512).

Running the scripts automatized

The repository provides sample bash scripts that automatize everything but mask creation and equalization.

1. jobProcessAgipdCalibration.sh: This is the script to be edited and executed by the user. Here all filenames are defined. As a naming convention it uses m_${moduleNumber} to identify files for different modules.
2. processAgipdCalibration_oneModule.sh: this script submits a batch job (to SLURM) that computes the calibration constants for one module. All modules called from jobProcessAgipdCalibration.sh are processed in parallel.
3. batchJob_processAgipdCalibration_oneModule.sh: This script runs serially on one cluster node. It follows the workflow described in this document. If necessary, parallel processing steps (see Figure 3) can be split to run on different nodes by replacing the call of this file from batchJob_processAgipdCalibration_oneModule.sh by submission of separate batch jobs with proper dependencies.

Running time

The creation of calibration data for one module takes ~1 night on a 32C/64T machine on the Maxwell cluster. Most time is consumed by batchProcessCurrentSourceScanData.py. Fortunately this script is rewritable to be parallelizable across different nodes. This way, using more computing power (and some manpower to rewrite the script), the computation time can be reduced to a 1-2 hours.