Cheetah configuration and documentation

Table of contents

Running cheetah

Cheetah can be run on its own, but is most easily used in conjunction with the crystfinder script – which does a lot of useful housekeeping. We should describe how to set this up somewhere

Setting up cheetah and the crystfinder script

- Set up the crystfinder script to automate everything
- Configure cheetah.ini
    - Select the right detector
    - Select background processing options
    -

-

Tuning

Optimising crystal hit finding

- Set hitfinderADCthreshold low enough, but not too low.
-

Optimising processing speed

- Set nthreads to 72 (on cfelsgi) or 16 (on most other servers)
- Check I/O speed limit using ioSpeedTest
- Turn off powder pattern creation (which skips mutex locks around summation of powder patterns)
- Increase amount of time between calculation of running background (recalculation mutex blocks all worker threads)
-

Notes on the configuration files

- Keywords to be specified in ini files:
- Keywords are NOT case sensitive.
- If the same keyword is set twice, only the last occurence will be used by cheetah. No notification will be given if this is the case, SO CAREFULLY CHECK YOUR INI FILES!!
- Cheetah will now exit if unrecognized keywords are found (a useful note will be printed to the terminal).
- By default, cheetah no longer reads both cspad-cryst.ini and cheetah.ini files within the directory where cheetah is running.
- If you want to know all of the keywords and their values used in a particular run of cheetah (including both user-set and default values), have a look in the log.txt file ouput by cheetah. This will reflect keyword values which have been changed as a consequence of other keyword selections.

Most commonly adjusted keywords

These are the most important keywords you'll probably ever want to tweak – the rest can likely be left alone.

Detector configuration

- **detectorName (CxiDs1)**
- **geometry (geometry/cspad_pixelmap.h5)**

Calibration and masks

- **darkcal (darkcal.h5)**
- **badPixelmap (badpixels.h5)**

Background subtraction

- **cmModule (0)**
- **subtractBehindWires (0)**
- **useSubtractPersistentBackground (0)**
- **useLocalBackgroundSubtraction (0)**

Hit finding algorithms

- **hitfinderAlgorithm (3)**
- **hitfinderADC (100)**
- **hitfinderNPeaks (50)**
- **hitfinderMinPixCount (3)**
- **hitfinderMaxPixCount (20)**

What gets saved

- **saveHits (0)**

- **saveAssembled (1)**
- **saveRaw (0)**

Other

- **nThreads (16 or 72)**

Complete listing of keywords

Many of these are power user settings designed for turning on or off features in testing.

Detector configuration

detectorName (CxiDs1)

==========

Recognized options for **cspad** are:

- "CxiDs1": cspad near detector
- "CxiDs2": ?? But it's in the config_v3 files
- "CxiDsd": cspad far detector
- "XppGon": cspad on XPP beamline (not tested, and no idea why this is referenced as a goniometer???)

Recognized options for **pnccd** are:

- Hooks in place, bit not yet implemented

detectorType (cspad)

==========

Recognized options are:

- "cspad"
- "pnccd"

This field could be deleted – specifying detectorName should be enough to set this internally

geometry (geometry/cspad_pixelmap.h5)

==========

Path to an hdf5 file specifying the real-space coordinates of each pixel. The hdf5 data fields are /x /y /z. The x coordinate cooresponds to data fast scan coordinates. Units are meters, or pixels (values are first divided by pixel size variable to get coordinates of pixels in space).

- Does any part of the cheetah code need to know where pixels are located, other than peak density calculation?

Yes: (1) Assembly of the assembled images (which Tom happily ignores, but are used by others); (2) Calculation of radial line outs for images and powder; (3) Determination of 'resolution' of a pattern from the radius at which 80% of the peaks lie;

- Is the /z data currently being used by cheetah?

Not at the moment, but possibly in the future. I use the same detector geometry file in 3D assembly and elsewhere, so keep the generality in the reading routine.

- See the keyword pixelSize-- it is important that this is consistent with the units in the geometry!

Yes! Values in geometry file are first divided by pixel size variable to get coordinates of pixels in an assembled image.

pixelSize (0.000110)

==========

Size of pixels in meters, defaults to values for the cspad detector.

Can also be set to 1.0 if geometry file is in units of pixels – but this may muck up other things.

Calibration and masks

darkcal (darkcal.h5)

==========

Path to an input hdf5 file containing dark current. Cheetah can create a darkcal from a dark run; see the generateDarkcal keyword. The hdf5 data field is "/data/data". Units are ADU. Darkcals should NOT be gain corrected.
Make sure the detector gain settings of the darkcal match that of the run.

gaincal (gaincal.h5)

==========

Path to an input hdf5 file containing the gainmap. By default the raw data will be multiplied by this map, although it can be inverted by setting xxxxxx. The hdf5 data field is "/data/data".

invertGain (0)

==========

Divide by the gain map, rather than multiplying (in case gain map is supplied as gain per pixel, rather than value to multiply pixel values by).
Conversion is performed when the gain map is read during program initialization. At the time of reading the gainmap, set the gainmap to its reciprocal value, which fixes possible divide by zero. In this way, pixels with value zero will specify a dead pixel.

peakMask (peakmask.h5)

==========

Path to an input hdf5 file indicating where to search for peaks.
Specifically, this operation multiplies the image data by this map prior to the usual peak search.
The hdf5 data field is "/data/data".

badPixelmap (badpixels.h5)

==========

Path to input hdf5 file indicating bad pixel. Specifically, this operation multiplies the image data by this map before carrying on with analsysis operations. Essentially, this has the same effect as a gainmap – but is a binary mask, meaning one doesn't have to keep messing with a gain calibration file.
The hdf5 data field is "/data/data"

Background subtraction

Proper subtraction of electronic and photon background is essential – there are many options.

useDarkcalSubtraction (1)

==========

Toggle the use of the darkcal map.

useBadPixelmap (0)

==========

Toggle the use of the bad pixel map.

useGaincal (0)

==========

Toggle gain corrections provided in the gaincal file. Data will be multiplied (not divided) by the gainmap unless invertGaincal=1 is set.

cmModule (0)

==========

One of three possible methods for subtracting common mode offsets from individual ASICs.
Subtract estimated common mode noise on each cdPAD ASIC. Common mode noise on each ASIC fluctuates randomly from frame to frame and must be estimated from the read out signal itself. Common mode is estimated as the lowest 10% of pixel values in each ASIC. 10% value can be set to something else by the user if desired.

This option assumes the lowest 10% of values represent only detector electronic noise - be careful of using this when there are no dark areas on the ASIC.

subtractcmModule (0)
==========
Old keyword – delete. This does exactly the same thing as the keyword subtractcmmodule. Remove one of them? Else, people might have both in their ini files.

cmFloor (0.100000)
==========
Use lowest x% of values as the offset to subtract (typically lowest 2%)

subtractUnbondedPixels (0)
==========
One of three possible methods for subtracting common mode offsets from individual ASICs.
Newer generations of cspad have some pixels unbonded for use in estimating ASIC electronic offsets. In csPAD revision 2 (August 2011) only a few ASICs have unbonded pixels for testing purposes, so this is of limited use at present. Pixel locations are hard coded for testing (generalize later)

subtractBehindWires (0)
==========
One of three possible methods for subtracting common mode offsets from individual ASICs.
For some experiments thin wires are placed in front of the detector and cast shadows; the counts behind shadows are used to estimate common mode offsets on each ASIC.

wireMaskFile (wiremask.h5)
==========
Path to input hdf5 file with binary mask specifying pixels behind wires to be used for background estimation. These wires are placed in front of the detector and cast shadows; the counts behind shadows to determine common mode noise?. Yes, exactly – We never used this in the crystal analysis, but I can show you data from some other runs for how this works.

useAutoHotpixel (1)
==========
Automatically identify and remove hot pixels. Hot pixels are identified by searching for pixels with intensities consitently above the threshold set by the keyword hotpixADC. In this case, "consistently" means that a certain fraction (user-set keyword hotpixFreq) of a certain number of buffered frames (number of frames set by the keyword hotpixMemory) are above threshold. The hot pixel map is updated every hotpixMemory frames.
Hot pixels within the corrected data will be set to zero. Note that the search for hot pixels is performed on the correcteddata (probably it should be performed on raw data instead?), so if you decide to change the corrections (e.g. darkcal, gainmap), the resulting hot pixel maps may differ.

useSelfDarkcal (0)
==========
This keyword setting will do exactly what useSubtractPersistentBackground does. It has been removed.
Old keyword

useSubtractPersistentBackground (0)
==========
Subtract the pixel-by-pixel median background calculated from the previous N frames (N set by the keyword bgRecalc, but apparently it cannot exceed 50 frames – it can, if bgbuffer is made large enough).

useLocalBackgroundSubtraction (0)
==========

Prior to peak searching, transform the image by subtracting the median value of nearby pixels. The median is calculated from a box surrounding each pixel. The size of the box is equal to localBackgroundRadius*2 + 1
Bad pixels and detector edge effects are not accounted for (i.e., if most nearby pixels are bad, the local median will be equal to zero).
This is somewhat slower, but very effective for nanocrystal data.

localBackgroundRadius (3)
==========
See keyword useLocalBackgroundSubtraction.

subtractBg (0)
==========
This keyword influences nothing whatsoever. It has been removed from the list of recognized ini keywords.
An old keyword

Background calculation tuning

selfDarkMemory (50)
==========
Depth of the frame buffer used for automatic background and hot pixel corrections

bgMemory (50)
==========
See keyword useSubtractPersistentBackground.
Duplicates above

bgRecalc (50)
==========
Strange, this *almost* does the same thing as bgMemory, butif bgRecalc is less than the default value of bgMemory, that default value will be used?
This sets how often the program pauses to recalculate background and hot pixel values. It is typically the same as the buffer size, but since recalculation is a thread blocking process, setting this to happen less frequently (eg: every 200 or 500 frames) speeds up execution.

bgMedian (0.5)
==========
Rather than using the usual median value for background, you can optionally choose any arbitrary K-th smallest element equal to bgMedian*bgMemory. Neat!

bgIncludeHits (0)
==========
Include hits in the background running buffer.

bgNoBeamReset (0)
==========
This does nothing at the moment.
Used to reset the background buffer whenever LCLS beam dies, defined as when GMD<0.2mJ. (not implemented)

bgFiducialGlitchReset (0)

==========
This does nothing at the moment.
Used to restart backgrounds when LCLS unexpected changes operating frequency. This was a problem on hot days in the June 2010 data set. (not implemented)

scaleBackground (0)
==========
Haven't made sense of this bit of code. Scale the running background prior to subracting from current frame. Scale factor looks a bit like a normalized correlation between the two (ignoring values above hitfinderADC), but that's definitely not what it is...
If photon background is only caused by photon scattering, it will be stronger or weaker depending on the strength of the incident pulse. Assuming signal and background are orthogonal vectors, the background component in current image is found using an inner product between background and the current data frame. OK for crystals, use with caution on other data. This was more of a problem at FLASH than at LCLS, because FLASH is more unstable in intensity.

scaleDarkcal (0)
==========
This does exactly the same thing as scaleBackground. Keyword has been removed.
Old keyword

Automatic hot pixel calculation

hotpixFreq (0.900000)
==========
See keyword useAutoHotPixel.

hotpixADC (1000)
==========
See keyword useAutoHotPixel.

hotpixMemory (50)
==========
See keyword useAutoHotPixel.

Hit finding algorithms

hitfinder (0)
==========
Specify the hitfinder algorithm. Various flavours of hitfinder:
1 - Number of pixels above ADC threshold
2 - Total intensity above ADC threshold
3 - Count Bragg peaks
4 - Use TOF
Note that the choice of hitfinder influences what is reported in the log files.

First, do these steps (regardless of the hitfinder choice):
Build a buffer, which is a replica of the corrected data. Values in the buffer array will be set to zero as those pixels are analyzed and rejected. If hitfinderUsePeakmask != 0, then multiply the buffer by this array before moving on.

Now, depending on the algorithm, do these steps:

Algorithm 1:

1) Count the pixels within the buffer that are above the (user-defined) hitfinderadc value.
2) Also, sum the values of the pixels that meet the criteria of step 1.
3) If the number of pixels is greater than the value of (user-defined) hitfindernat, count this frame as a hit.
4) Report the number of pixels in the log files as npeaks and as nPixels.
5) Report the total counts (intensity) as peakTotal

Algorithm 2:

Same as algorithm 1, except that the criteria for a hit is now that the *intensity* is greater than hitfindernat, rather than the pixel count.

Algorithm 3:

Briefly, this is what happens:
1) Scan the buffer, module-by-module, searching for "blobs" of connected pixels which all meet the criteria of being above the threshold defined by the keyword hitfinderADC. A pixel can be "connected" to any of its eight nearest neighbors. If its "connected" neighbor is "connected" to another pixel, then all three are mutually "connected" to each other.
2) If a blob contains more than hitfinderMinPixCount connected pixels, and less than hitfinderMaxPixCount pixels, it is counted as a peak.
3) The center of mass and integrated intensity is calculated for the blob (this is the peak position and integrated intensity).
4) If there are more than hitfinderNpeaks peaks, and less than hitfinderNpeaksMax, then count this as a hit.

Some important keywords:
- hitfinderNAT
- hitfinderADC
- hitfinderMinPixCount
- hitfinderMaxPixCount
- hitfinderNPeaks
- hitfinderNPeaksMax
- hitfinderCheckMinGradient: Before considering a peak candidate, check that the intensity gradient is above this threshold. Here, the "gradient" is the mean square derivative of the "above/below" and "upper/lower" pairs of pixels connnected to the pixel of interest.
- hitfinderMinGradient: Threshold for the above keyword.
- hitfinderCheckPeakSeparation: After locating peaks, throw out all the peak pairs that are too close together.
- hitfinderMaxPeakSeparation: Threshold for the above keyword.

This algorithm also calculates a quantity called "peak density"? What's being reported in the log files?

Algorithm 4:

Use time-of-flight data... details later...

Hitfinder tuning

hitfinderAlgorithm (3)
==========
See the keyword hitfinder.

hitfinderADC (100)
==========
See the keyword hitfinder.

hitfinderNAT (100)
==========
See the keyword hitfinder.

hitfindertit (1283604304)
==========
See the keyword hitfinder. Is it TAT or TIT?

hitfinderCluster (0)
==========
This does nothing at the moment.

hitfinderNPeaks (50)
==========
See the keyword hitfinder.

hitfinderNPeaksMax (100000)
==========
See the keyword hitfinder.

hitfinderMinPixCount (3)
==========
See the keyword hitfinder.

hitfinderMaxPixCount (20)
==========
See the keyword hitfinder.

hitfinderUsePeakMask (0)
==========
See the keyword hitfinder.

hitfinderUseTof (0)
==========
Does choosing hitfinding algorithm 4 accomplish the same thing?

hitfinderTofMinSample (0)
==========
?

hitfinderTofMaxSample (1000)
==========
?

hitfinderTofThresh=1283604304
==========
?

Specifying what gets saved in exported frame HDF5 files

saveHits (0)
==========
Save the hits to individual hdf5 files. Exactly what will be saved is
determined by the keywords saveRaw, saveAssembled, savePeakInfo
saveDetectorCorrectedOnly, saveDetectorRaw, and possibly more...

saveAssembled (1)
==========
Save the data which has been interpolated into a physically correct image (as would be seen on a sheet of film), based on the geometry file. Note that this will take up more space on disk, but provides an image which can be analysed/displayed as if the detector were one CCD.
Also, note that geometry is updated sometimes, and you will need to re-run all of your hitfinding if you intend to store the data only in assembled form.
The hdf5 field is /data/assembleddata, and has zeros where there is no data. If present, it will be symbolically linked to the field /data/data.

saveRaw (0)
==========
Save corrected data in the hdf5 files, in data layout as read from the detector, without interpolation into a physically realistic image.
The hdf5 data field is /data/rawdata. Note that the word "raw" does not mean uncorrected (!) as one might think; it just means that it has not been interpolated onto a larger (zero-padded) array based on the geometry file (this one is the "assembled" data set).
Maybe the field name is misleading? Would saveUnassembled be better?

savePeakInfo (1)
==========
Save the peak center of mass (two coordinates), intensity, and number
of pixels to the hdf5 and log files. These values are specified in
the "assembled" and "raw" coordinate systems. Look to the hdf5 fields
/processing/hitfinding/peakinfo* for this information. More details
later.

saveDetectorCorrectedOnly (0)
==========
Even if background subtraction is used for hit finding, back up to image with only detector corrections subtracted and save this instead. Useful for preserving the water ring, for example.

If set to non-zero value, save the data which has only the following operations done to it (in this order):

1) Subtract darkcal
2) Subtract common mode offsets
3) Apply gain correction
4) Multiply by bad pixel mask

If set to zero, then you get these additional corrections (in this order):

5) Subtract running (persistent) background
6) Subtract local background
7) Zero out hot pixels
8) Multiply by bad pixel mask (again)

If the keyword saveDetectorRaw is set, then none of the above corrections will be applied (therefore, this keyword has no effect).

saveDetectorRaw (0)

Image will be saved exactly as represented in the XTC data stream, regardless of what detector corrections and photon background subtraction is used for hit finding. This option is mainly of interest to detector groups who want to look at data in as raw a form as possible, or for low-level diagnostics on common mode, electronic noise, etc.

As the name suggests, save the intensity values with no corrections applied, even though the corrections were done prior to hitfinding. This keyword trumps the keyword saveDetectorCorrectedOnly. Note that this keyword may be confused with saveRaw, which has to do with the question of padding the array and interpolation, not data processing.

hdf5dump (0)

==========

Write every nth frame to an hdf5 file, regardless of whether it was found to be a hit.

Creation of calibration files

generateDarkcal (0)

==========

Create a darkcal from a given run (which should contain dark data).
Takes the average of all patterns, and output a "darkcal" hdf5 file named rXXXX-darkcal.h5 in the end.
Basically this option tricks cheetah into thinking every frame is a "hit". The darkcal is the average not he sum, unlike the usual "powder" patterns which are generated.
If you set generatedarkcal=1, the following keywords will be modified so everything works as expected:

cmModule = 0;
cmSubtractUnbondedPixels = 0;
subtractBg = 0;
useDarkcalSubtraction = 0;
useGaincal=0;
useAutoHotpixel = 0;
useSubtractPersistentBackground = 0;
hitfinder = 0;
savehits = 0;
hdf5dump = 0;
saveRaw = 0;
saveDetectorRaw = 1;
powderSumHits = 0;
powderSumBlanks = 0;
powderthresh = -30000;
startFrames = 0;
saveDetectorCorrectedOnly = 1;

generateGaincal (0)
==========
Automatically create a gain map file from flat field data. Works, but the output likely needs tweaking by hand in IDL/Matlab.
Sum all patterns (regardless of whether or not each frame is a "hit"), form an average, then divide this average by the median value of the image. The median value is therefore gain = 1. The gainmap will be saved as "rXXXX-gaincal.h5". At the moment, the gainmap is set to zero where it is outside of the bounds 0.1 and 10.

When setting generategaincal=1 the following keywords will be modified so everything works as expected:

```
cmModule = 0;
cmSubtractUnbondedPixels = 0;
subtractBg = 0;
useDarkcalSubtraction = 1;
useAutoHotpixel = 0;
useSubtractPersistentBackground = 0;
useGaincal=0;
hitfinder = 0;
savehits = 0;
hdf5dump = 0;
saveRaw = 0;
saveDetectorRaw = 1;
powderSumHits = 0;
powderSumBlanks = 0;
powderthresh = -30000;
startFrames = 0;
saveDetectorCorrectedOnly = 1;
```

saveInterval (1000)
==========
Periodically save running sums and update the log file at this interval.

Image summation (powder patterns)

powderSumHits (1)
==========
Record and save the summed (not averaged) intensities from frames determined to be hits. Will be saved as the file named powderSumHits.h5. The hdf5 data field is /data/data.

powderSumBlanks (1)
==========
Record and save the summed (not averaged) intensities from frames determined to be non-hits. Will be saved as the file named powderSumBlanks.h5. The hdf5 data field is /data/data.

powderthresh (-20000)
==========
Apply this intensity threshold before powder summation.
Setting to ~500 typically captures only peaks; setting to 0 sums only positive values; setting to -20,000 typically sums everything.

Time of flight spectrometer (Acqiris)

tofName (CxiSc1)
==========
Is this relevant to our pump laser diode trace?

tofName (CxiSc1)
==========
Name of Acqiris device in XTC data stream

tofChannel (1)
==========
Acqiris channel number

Multithread tuning and speed optimization

nThreads (16)
==========
Run this many worker threads in parallel (one worker thread per LCLS event). Set to 1x-2x the number of cores on the machine.

- 72 or 144 on cfelsgi (72 physical cores)
- 16 is more than adequate on most servers (eg: compute farm at SLAC uses 12 or 16 core machines)

Speed may saturate before all threads are busy if data transfer makes cheetah I/O limited (check with ipSpeedTest), or if competing access to shared variables results in mutex locks (happens when too many threads write to powder patterns or running background buffer at once).

useHelperThreads (0)
==========
This doesn't appear to do anything at the moment??
This currently does nothing. It was intended for computing backgrounds asynchronously with processing data frames.

threadPurge (10000)
==========
Periodically pause and let all threads finish. On cfelsgi we seem to get mutex-lockup on some threads if we don't do this.
But we should check again – this may have been something else, and we can drop (or simply not mention) this keyword.

Data processing flow: skipping XTC frames

startAtFrame (0)
==========
Skip all frames in the xtc file prior to this one (no processing done).
What happens when there are multiple xtc files? Does cheetah concatenate the events in all of the xtc files, or does cheetah start at this frame in each xtc file?
Interesting point, and I have not tested this - myana.cc and main.cc code dictates this behavior presently. I suspect it is in order – to test we could print the fiducuals and see whether they skip go in sequence as expected, or jump at the start of a new XTC file

stopAtFrame (0)
==========

Skip all frames in the xtc file after this one. Setting to 0 means to ignore this setting.
See question about startatframe above, regarding the case of multiple xtc files.
Currently, the program does not end when this frame is reached; it just keeps running until the end of the xtc file(s), doing nothing other than printing a message indicating that it is skipping to the end of the file. Maybe this can be fixed, but probably the change needs to happen in the myana.cc or main.cc code. Or we do an exit(1) and let the program simply die.

startFrames (0)
==========
Number of frames at the start of processing used for background estimates, etc, before starting hit finding etc.

ioSpeedTest (0)
==========
Run through events in xtc file, reading in all data, but do no data processing (don't spawn worker threads). Useful for checking raw I/O speed to determine whether the process is I/O bound or CPU/mutex bound.

Misc

debuglevel (2)
==========
Sets verbosity level of the output (how much diagnostic junk is printed to screen)

Cheetah output files

==========

rXXXX-sumBlanksRaw.h5

as the name suggest, this is simply the sumation of blank frames (non-hits)

rXXXX-sumBlanksRawSquared.h5:

As the name suggests...

rXXXX-sumBlanksRawSigma.h5:

NOT AS THE NAME SUGGESTS; this one is divided by the number of frames.

Same goes for rXXXX-sumHitsRaw and so on...

There is a symbolic link called /data/data, but what it links to depends on parameters in the ini file.

map log files to variables in the code:

framefp:

FrameNumber threadInfo->threadNum
UnixTime threadInfo->seconds
EventName threadInfo->eventname
npeaks threadInfo->nPeaks
nPixels threadInfo->peakNpix
totalIntensity threadInfo->peakTotal
peakResolution threadInfo->peakResolution

peakDensity threadInfo->peakDensity

cleanedfp

global->runNumber
Filename info->eventname
npeaks info->nPeaks
nPixels info->peakNpix
totalIntensity info->peakTotal
peakResolution info->peakResolution
peakDensity info->peakDensity