The PACS Advanced User Manual

Herschel Data Processing

Issue 3.0

The PACS Advanced User Manual: Herschel Data Processing

Table of Contents

Chapter 1. Introduction

This PACS Advanced User Manual is the original PACS pipeline document. It has been written by and for internal users, and hence the version on the 3.0 release of HIPE will be, in places, difficult for general astronomy users to understand. There are sections of both chapters that include tasks that were used only for data collected before Herschel was launched, and sections that discuss pipeline tasks that are still not ready for general users to use. The photometry section is also a little out of date.

However, the detailed pipeline descriptions—the task parameters and the algorithms—are explained here and these will be useful for general astronomy users to read, and they are for the most part well explained. Consider this to be a reference document, where you look up the details for a pipeline task you want to know more about. If you are starting to work on your PACS data for the first time and want to learn how to run the pipeline, what order to run the tasks in, and how to check on the intermediate products, it is the PACS Data Reduction Guide that you should be reading.

By the 4.0 release the PAUM will be up-to-date and easier for all to understand.

Chapter 2. PACS spectroscopy standard data processing

2.1. Introduction

This chapter describes the standard processing steps (current and some old) for the various spectroscopy observation modes of the PACS instrument. Details about how the procedures work, why they are necessary, and the calibration tables used are given. This chapter is intended for advanced (or at least intermediate) users of HIPE and those used to working with PACS spectroscopic data. A first-time user should rather be reading the PACS Data Reduction Guide (the location of which in the HIPE help is still TBD). Some of the processes described here will only be used by the PACS calibration team, not the astronomy end-users, and the level of expertise assumed of the reader is fairly advanced. However, as the descriptions of the tasks are more complete here than provided in the basic data reduction guide, this chapter (indeed, this complete guide) could still be useful to the beginning user. Additional information can be found in the URM (User's Reference Manual) and the DRM (Developers Reference Manual, aka API), both accessible from the HIPE help page; these are, please note, advanced-level documents.

We recommend you begin from Level 0 data extracted from the Observation Context, rather than directly from telemetry (tm) files, although we do include here some processing steps for tm files, which are mostly prior to the Level 0 stage. Working with tm files—extracting them out, investigating them, etc.—was explained in previous chapters of this User Manual (Houskeeping; Summarising TM files).

2.2. Quick notes/to come

Still to be covered: AOTs other than chop–nod; slicing; noise calculation and propagation; new spatial calibration concept; flux calibration including flatfielding and drift correction; tools for inspecting and "editing" individual spectral time-lines before converting to a rebinned cube.

Some brief notes on some of these items (pipeline workshop 19–21 Oct 2009):

- Spatial Calibration: there will be a completely new jython script that will attach the correct RA and Dec to the central pixel and then the other pixels. Currently the positions coming out of the pipeline are wrong. So at present (Nov 10 2009) the spatial calibration in the cubes should be considered "browse" quality, rather than quantitative science quality. In addition, it is realised that for all (except the central?) spaxels the nod A and nod B may never be 100% aligned, this having consequences for specAddNod and the subsequent pipeline processing.
- Noise: Noise is calculated and propagated through the pipeline, however there are still issues over the details of the calculation of errors, and in particular that from first source of noise (from Level 0/0.5 product). Currently the dominant noise is from the response+dark. There is a script called specEstimateNoise in CVS that you could look at if you like; one could also inspect the workshop presentations on the twiki.

2.3. Summary of the pipeline

Here we give an itemised overview of the pipeline data processing steps. The Level 0 data to work on will usually be averaged ramps (*Ramps* class) or on-board fitted ramps (*Frames* class). Raw ramps are unlikely to be downlinked from the spacecraft (S/C) during routine operations, but will sometimes be downlinked during CoP and PV, and of course may be worked on from flight spare tests.

Level 0 to 0.5 processing is the same for all AOTs (points 1 to 8) and many of the subsequent tasks are also performed for most AOTs. (An explanation of the terminology is provided later.)

1. If working on *Ramps* data, flag for saturation. Then fit the slopes and convert the data to a *Frames* product
2. Signal is converted from digits/s to Volts/s
3. Status entry for calibration blocks is added to
4. S/C time is converted to UTC
5. S/C pointing is added to the Status table for the central pixel of the detector and chopper units are converted to sky angle
6. Wavelengths for each pixel are calculated; Herschel's velocity is corrected for
7. Data "blocks" are recognised and the information organised in a table; Status table is updated.
8. Masking. Bad pixels will have already been masked. Masking for readouts taken during grating and chopper movements is performed, and for saturation if the data reduction began on a *Frames* product
9. Masking for glitches is performed
10. Signal non-linearities are corrected for
11. Signal is converted to a level that would be the minimum capacitance setting
12. The dark current and pixel response levels (their individual sensitivities) are calculated using differential (internal) calibration source measurements to populate the absolute response arrays [(V/s)]; A response drift is then calculated
13. Chop#nod AOT: the up- and down-chops are combined (i.e. a background+dark subtraction); the signal is divided by the relative spectral response function and then pixel responses (and their drift) are corrected for; the nods are averaged, such that each nod-cycle (not each nod) becomes one.
14. Wavelength-switching AOT: *TBD*
15. Off-map AOT: *TBD*
16. Calibrated 5x5xlambda data cubes are generated
17. The wavelength grid is created
18. Outliers are flagged (another glitch detection)
19. The data cube is spectrally resampled
20. The data cube is spatially rebinned, different pointings combined and (mosaicked), or 3D drizzled (*the 3d Drizzle task is not yet ready*)

The steps described here follow those in the "ipipe" pipeline scripts (in the directory with the HIPE software, these are located in /scripts/pacs/toolboxes/spg/ipipe [pipeline scripts including slicing are in scripts/pacs/spg/pipeline/phot|spec/ipipe, but the slicing concept is still under construction]). In addition, if you downloaded the complete HIPE software (asked for the upack=yes option when installing), you could also consult the HSC pipeline scripts (/src/herschel/pacs/toolboxes/spg/pacsspectro), although these are not intended to be interactive and should not be your first source of information about the pipeline.

For large datasets the data will probably have been sliced, that is organised in distinct and separate, but linked parts using an "astronomical" logic (e.g. separate the different rasters of a single observation; keep together all data of the same spectral line). *Once this logic has been worked out and incorporated in the pipeline scripts, that information will be included here.*

2.4. The sliced-products pipeline

The sliced pipeline, that is where the input data are sliced according to line and raster in order to reduct the memory load when running the pipeline, is more or less the same as the standard pipeline except that the tasks have a "sliced" in front of them and you need to add in a few extra commands near the beginning of each level's reductions. If you want to see the scripts then go to the ipipe directory of your HIPE installation: /scripts/pacs/spg/pipeline/spec|phot; we will not list the contents of these scripts here, because the tasks being used are exactly the same as already here-described.

Note that the slicing concept and implimentation are in a beta stage.

These "sliced" tasks are simply a wrapper that loops the tasks over the individual slices in the input product, which rather than being a *Frames* or *Ramps* or cube, are a *SlicedFrames*, or a *ListContext* of *Ramps* or cubes (we do not yet have a *SlicedRamps* or *SlicedPacsCube*).

The ipipe scripts FramesL05.py or Ramps05.py are first, then chopNodStarL1.py and chopNodStarL2.py (and similarly for the other AOTs). The following is a description of the chop—nod scripts: it is intended that you read these scripts as you go through this description.

- FramesL05.py: first you get out from the ObservationContext your data as a *PacsContext* (level0=PacsContext(obs.level0)), where a *PacsContext* is a class that is a wrapper for PACS products. Extract out the various calibration information the pipeline needs (pointing product etc.) and get a calibration tree. Then extract out from the PacsContext the Level 0 products (slicedFrames=SlicedFrames(level0.fitted.getCamera(red).product)) and put them in a *SlicedFrames*. Then you run the Level 0 to 0.5 pipeline tasks, but using the syntax

  ```
  slicedFrames = slicedWaveCalc(slicedFrames, calTree=calTree)
  ```

  instead of

  ```
  Frames = waveCalc(slicedFrames, calTree=calTree)
  ```

  Looking at the ipipe script you will notice that after the task slicedSpecFlagBadPixelsFrames the slicedFrames product is then Level0.5-sliced, this being done according to a hard-wired logic

  ```
  slicedFrames = pacsSliceContext(slicedFrames,level='0.5')
  slicedDmcHead = pacsSliceContextByReference(slicedDmcHead,slicedFrames)
  ```

  The logic at Level 0.5 is currently that the slicing is done by raster, i.e. each new raster pointing is a new slice. If there was only one raster position in the observation, there will only be one slice in the *slicedFrames*. The first command above does this, the second then slices the DecMec header according to the logic of the *slicedFrames*. In pacsSliceContextByReference, the first parameter can be, in principle, any *ListContext* but it makes most sense if it is the DecMec header, and the second must be a *slicedFrames* product. This is necessary because you will used this sliced DecMec header in subsequent pipeline tasks and they must be sliced as your *slicedFrames* is.

  Near the end the pipeline script shows you how to save the *slicedFrames* back to pool as an ObservationContext. You do not need to do this step if you do not want to, you could continue straight on to the next ipipe script. At the very end of the script are two tasks that add information to the meta data to explain what slicing has been done. These are pacsPropagateMetaKeywords and addSliceMetaData. The first adds the meta data of the orginal ObservationContext to the new one (the sliced one) and the second adds meta keywords to the meta header of the slices, such as aotMode, lineDescription, and rasterId. Very useful for you to later know what the slices are made of. These tasks work on any *List/MapContext*.

  RampL05.py is so similar to FramesL05.py that it is not separately explained here.

- chopNodStarL1.py: the script begins by extracting from the ObservationContext that you saved at the end of FramesL05.py the level0_5 product and then putting it in a *slicedFrames*. You then are asked to slice the data according to the hard-wired logic at Level 1, where now the slicing is done on the on raster and (spectral) line. Then the pipeline tasks are run through, and the *slicedFrame* and/or *slicedCube* is saved to an ObservationContext. At the very end of the script are two tasks that add information to the meta data to explain what slicing has been done. These are pacsPropagateMetaKeywords and addSliceMetaData. The first adds the meta data of the orginal ObservationContext to the new one (the sliced one) and the second adds meta keywords to the meta header of the slices, such as aotMode, lineDescription, and rasterId. Very useful for you to later know what the slices are made of. These tasks work on any *List/MapContext*.

- chopNodStarL2.py begins by grabbing from the ObservationContext the *slicedCubes* and *slicedFrames*. Soon there will be a looping over line ID, so these IDs are extracted from the MasterBlockTable (this being unique to a *slicedFrames*). You then loop over these lines, taking them out of the *slicedCubes* and running the Level 1 to 2 pipeline tasks on them. This is done because we do not want to combine lines in the rebinned cube, so we make a cube per line and per raster. But we do want to combine the raster positions in the final, projected cube, which is then made one per line. Finally the products are again saved to an ObservationContext.

Note

The calibration tree that the ipipe script grabs is that from the data (the commands: calTree = obs.calibration and GetPacsCalDataTask.setDefaultCalTree(calTree)). This is necessary in this script so that it can be run as a whole. Of course, if running manually you can grab whichever calibration tree you want

The MasterBlockTable is a combination of the individual BlockTables of the *Frames* in the *slicedFrames*. It can be inspected with e.g.

```
mbt=slicedFrames.masterBlockTable
```

In the MasterBlockTable, be aware that, per block, the StartIdx and EndIdx entries are relative to the slice that is that block. The FramesNo entry contains the slice numbers relative to the whole Frames product.

To extract a single Frames from a slicedFrames you use the syntax of a ListContext

```
frame=slicedFrames.refs[0].product
```

But there is a method that also does this.

For slicedFrames the following methods will work

- get(i), to get the i-th *Frames* product; getScience(i) to get the i-th Science *Frames* product; getCal(i) to get the i-th calibration *Frames* product; getNumberOfFrame(), getNumberOfScienceFrames(), getNumberOfCalFrames() to find out the number of *Frames* of the specified type. e.g.:

  ```
  slicesFrames.get(0)
  ```

For the slicedTasks that slice data one can specify the following parameters

- scical="cal" to only use calibration slices, scical="sci" to only use science slices, and sliceSelection=[python list] to select particular slices,

  ```
  slicedFrames = slicedWaveCalc(slicedFrames, calTree=calTree, scical="cal")
  # OR -- do not use these two parameters together
  slicedFrames = slicedWaveCalc(slicedFrames, calTree=calTree,
  sliceSelection=[0,1,2,3,6])
  ```

We now can store slices automatically in a temporary pool. This can be turned on by setting the following properties:

```
hcss.pacs.spg.usesink = true
```

```
hcss.ia.pal.store.tempstore = { temppool -}
hcss.ia.pg.tmpstore = tempstore
```

Without setting these the slices will still stay in memory. The user will be informed about this with a one-time pop-up when they start to use slicing. Note that the user has to delete the temppool from time to time. It can grow quite fast and it will be not be deleted by the system.

To get the data from the ObservationContext use the following:

```
slicedFrames=SlicedFrames(level0.fitted.getCamera('red').product
# or, shorter
slicedFrames = SlicedFrames(level0.fitted.red.product)
```

Note

There is implemented an iterator in SlicedFrames, but that one does not work properly in Jython. Therefore, you have to loop over the frames as follows:

```
or i in range(0,slicedFrames.numberOfFrames):
   print slicedFrames.get(i).mask.activeMaskTypes
```

As soon as the iterator works properly this should also work:

```
for frames in slicedFrames: print frames.mask.activeMaskTypes
```

2.5. Processing levels

There is a Herschel-wide convention on the processing levels of its instruments.

- *Raw Telemetry:*

  Telemetry packets produced (and relayed to us) by the instrument in the course of the observation. In PACS we work with this level as a class *PacketSequence*.

- *Decompressed Science data:*

  This is an artificial level. The data are not stored and not visible to general user but are held in a format that can be analysed for debugging purposes.

  Telemetry data as measured by the instrument are minimally manipulated and stored as DataFrames interface. For PACS spectroscopy this level is stored/manipulated as class *DataFrameSequence*: a sequence of PACS DataFrames, which are decompressed SPU (Signal Processing Unit) buffers. What is contained in every decompressed SPU buffer depends on the SPU reduction mode. Typically there are several reduced readouts for every active detector (pixel), i.e. averaged ramp readouts and/or fitted slopes plus full ("raw") 256Hz readouts for a few selected pixels and mechanism/status information sampled at 256Hz by the DecMec (detector and mechanism controller), the so-called DMC Header. Raw readouts for every pixel will probably only be available for flight spare data or special flight model tests.

- *Level 0 data:*

  Level 0 data is a complete set of minimally processed data, and includes that held within Observation Contexts. After Level 0 data generation there is no connection to the database from which the raw data was extracted (that is the database from where the tm files come; this database is unstructured, being a series of packets with timestamps). Therefore the Level 0 data contain all the information required.

  - Science Data

    Science data are organised in user-friendly classes. The *Ramps* class contain (i) raw channel data (but usually only for a certain number of detector pixels, as these data are huge) (ii) averaged channel data, for all pixels, and the *Frames* class, for which on-board fitting of the slopes of the raw ramps has already been done.

  - Auxillary data

    Auxiliary data for the time-span covered by the Level 0 data, such as the spacecraft pointing (attitude history), the time correlation, selected spacecraft housekeeping, etc. The information are partly held as status entries attached to the basic science classes (*Ramp* and *Frame*) and the rest are available as separate products (e.g. the "pointing product") which you can access. It is possible that the auxiliary pool will be held in a different store to where the Observation Contexts can be found.

  - Decoded HK Data

    HK (housekeeping) data tables with raw and converted HK values (converted from raw to physical engineering values).

  - Level 0 data of associated observations, e.g. flatfields or photometric checks or other Trend Analysis products taken through the operational day or before. Provision of this is still under discussion.

- *Level 0.5 data:*

  Processing until Level 0.5 is AOT independent. These data are also saved in the product pool that you get e.g. from the HSA. At this level additional information has been added to the class (masks for saturation and bad pixels, RA and Dec, the BlockTable,...) and basic unit conversions have been applied (digital values to volts, chopper position to sky angle). We recommend that astronomy users who wanted to reduce their data themselves begin from this level.

- *Level 1 data:*

  Level 1 data generation is AOT dependent (although there will be much overlap between the AOTs). Level 1 data are also saved in the product pool to be extracted if you so wish. Data processing at this level is concerned with cleaning and calibrating, and as the end the data are converted to a basic cube, of *PacsCube* class (the 16x25 useful pixels have been converted to 5x5 spaxels, each holding 16 individual spectra).

- *Level 2 data:*

  Going from Level 1 to Level 2 the cube is spectrally and spatially rebinned and the cube is now of class *PacsRebinnedCube*. At this level scientific analysis can be performed. Level 2 work is highly AOT dependent.

- *Level 3 data:*

  This is simply a level where the scientific analysis has been done by the data users (e.g. spectral cubes converted to velocity maps, source catalogues), and it is hoped that users will import these products back into the HSA.

2.6. Masks

Masks are created by various tasks to flag individual data-points that are or may be bad. Subsequent tasks may use these masks. The masks that we currently have are:

- BLINDPIXELS: pixel masked out by the DetectorSelectionTable (applied already at Level 0)
- BADPIXELS: bad pixel masked during pipeline processing
- SATURATION: entire saturated pixels or individual saturated readouts
- GLITCH: readouts/signals affected by glitches (e.g. cosmic rays)

- UNCLEANCHOP: masking of unreliable readouts/signals taken during chopper transitions
- DEVIATINGOPENDUMMY: masking of an entire pixel column if the dummy or open channel of that column shows deviating ramps or `weird' signals
- OBSWERR: masking if randomly checked deviations of onboard to onground reductions larger than the expected noise occur
- BADFITPIX: masking of resets where the signal determination failed during fitting of the ramps
- GRATMOVE: masking of readouts/signals taken during grating movements
- OUTLIERS: masking of outliers for each wavelength bin in a *PacsCube*
- NOISYPIXELS: noisy pixels mask is to be added: by the task specFlagBadPixelsFrames, a warning mask based on GeGa analysis

We recommend that all masking be done on the *Frames* product, except if you begin with a *Ramps* product, in which case the saturation mask can be applied before fitRamps.

Oct 2009, the new masking concept: It is necessary to activate masks for them be used in any pipeline task that uses masks. A task may still only use certain masks (i.e. those that make sense for the task's job), but any inactive masks will certainly not be used. Once a mask has been created (by a mask-creating task) it is automatically activated, so if you do not want to use it in a subsequent mask-using task, you need to deactivate it.

To check if a mask is active:

```
print frames.mask.activeMaskTypes
# response will be something like
["BLINDPIXELS","BADPIXELS","UNCLEANCHOP","GRATMOVE"]
```

It is probably safest to simply always activate all the masks you positively do want a task to use, and deactivate all others at the same time, before running each pipeline task that uses masks. In the pipeline description that follows we will include the recommended activateMasks in each pipeline task description where masks are used. If you stick to what we write here then you will be O.K (or for a more rapidly updated pipeline description, see the ipipe scripts with your HIPE installation).

Activating and deactivate are done with the same task, in the following way:

```
# for a ramp, deactivate all masks (by activating none)
ramp = activateMasks(ramp, String1d([" -"]), exclusive = True)
# for a frame, activate the masks listed in the String1d, all
# others become inactive
frame = activateMasks(frame, String1d(["UNCLEANCHOP","GLITCH"]),
  exclusive = True)
```

This task works on *Frames*, *Ramps*, and *PacsCubes*. The parameter `exclusive` set to True means that the masks specified will be set to active and the rest to inactive (default), False means the specified masks are activated and for the others their state is not altered.

There is also a deactivateMask task, but to be honest it is best to stick to the activateMask task.

```
# for ramp, deactivate the masks listed in the String1d, all others
# become active
ramp = deactivateMasks(ramp, String1d(["UNCLEANCHOP","GLITCH"]),
   exclusive = False)
```

For deactivateMasks the default False for `exclusive` means that the specified masks are deactivated and the rest untouched, while True means the specified masks are deactivated and all others activated. One use for deactivateMask is if you then want to deactivate a single mask:

```
frames = deactivateMasks(frames,String1d(['GRATMOVE']))
```

which deactivates GRATMOVE and leaves all others untouched.

To inspect the masks you can use the MaskViewer GUI, which displays an image of the detectors at the top, where pixels are selectable for displaying as spectra (signal vs readout number) below,

```
from herschel.pacs.signal import MaskViewer
MaskViewer(myframe)
```

It is possible to set your own mask or edit current masks using the MaskViewer, the description for this is in the MaskViewer entry of this manual (currently Chap. 10). *There will also be a more complex mask GUI released in the near future*

To inspect a mask as an array from the command line you can use the .getMask() method to extract the data out, with syntax such as:

```
print ramp.getMaskTypes() # see what masks are present
# extract data to e.g. plot it
glitch=Int4d(ramp.mask["GLITCH"].data)
# or
glitch=Int4d(ramp.getMask("GLITCH"))
```

This gives you an Int4d dataset (glitch). The default output from the getMask method is a Bool4d. It is also possible to set or edit masks yourself on the command line. However, while this may be more convenient for you (e.g. you want to identify bad readouts using an algorithm rather than via the GUI), it is more complicated because the syntax for *Frames* and *Ramps* differs, and there are many ways to do this. Find a method you can understand and stick to it! A few examples:

```
# to add a new type of mask
frame.addMasktype("MYMASK","explanation of mymask")
# or to remove a mask
frame.getMask().removeMask("MYMASK")

# add/change the mask value for a frame product (3 dimensions)
frame.setMask("MYMASK",4,5,2,1)
# will set to -"True" the mask value for pixel row 4, column 5,
# and readout 2 (where the dimensions of a frame is
# 18,25,z: z being the time-line dimension)

# to add/change a mask value for a ramp product (4 dimensions)
true=Bool1d([1])
ramp.setMask("MYMASK",4,5,2,true)
# where 4,5,2 are the same as before (where the dimensions
# of a ramp are usually 18,25,64,z or 18,25,32,z). This sets to
# -"True" ALL the readouts (all z) of ramp number 2
```

True means that this particular pixel and readout is bad, i.e. you want a flag raised for it. We recommend you read the API entry for Mask (not Masks) to learn more, in particular the setMask and set attributes which tell you how to set flags for any range of data-points in 3 or 4 dimensions of a frame or ramp.

Note

as used in this chapter, X for a pixel follows the "row" direction (e.g. as seen when inspecting data with the MaskViewer) and Y follows the "column" direction. There are 18 Xs and 25 Ys.

2.7. Accessing data as an Observation Context

If you got your data (e.g. from the HSA) as a tar file, you should untar it into a "pool", into a directory located off of your "lstore" directory; by default HIPE expects your lstore directory to be located at e.g.

[/Users/me/].hcss/lstore/. Your data will then be in the form in which you can access the Observation Context. If instead you have a tm file, you will need to run a task to decompress it into a pool off of /lstore.

The Observation Context can be thought of as a container of products (such as *Ramps*) that belong to a specific observation. It provides associations between all the products you need to process that single observation (e.g. pointing products, HK data). You will need to extract out your data from the Observation Context and can then run the pipeline tasks on them. This will be explained here, but first we will start with the few steps necessary when beginning from tm files.

Note

Syntax: things that are parameters of tasks will be written like `this`; the names you give them will be written in normal font; class names will be written like *This*; and the word ramp, frame, and cube refers to products that are of class *Ramps*, *Frames* or *Cube*, but when written in normal font we are referring to a generic product, rather than its class type or the name you gave it.

2.7.1. Populating the pool from tm files

Telemetry files are a format where a whole lot of data is held in one file, e.g. all the data collected for one observation. It is necessary to decompress the file in order for the software (and the human) to access the individual parts. When you decompress a tm file to populate a pool, among the directories of files that are created will be something with the name "ObservationContext" in it.

If you have a single tm file you can populate a pool with the command:

```
populatePacsPoolFromFiles(filename, obsid, poolName [,startDirectory=<string>]
                          [,pattern=<string>] [,walk=<boolean>)] [,hklimit=<number>]
                          [,sclimit=<number>]
```

`filename` is a string and is the name of your tm file; `obsid` is the observation ID (a number with "l"—el, not one#at the end); `poolName` is a string and is the name that you want the pool to have (e.g. mypool); `startDirectory` is the starting directory for a pattern search; `pattern` is the pattern for a file search (default is *.tm); `walk` (default, False) is whether subdirectories will be considered in the pattern searching; `hklimit` is the number (default 10000) of HK TmSourcePackets used for a pool entity; `sclimit` is the number (default 10000) of science SPU buffers used for a pool entity. It is perfectly possible to run this task with the default parameters.

One can populate the same pool sequentially with the contents of more than one tm file (and it is very useful to do that if you later want to extract into a single product #more than one dataset)

If you have a tm database that is full of tm packets rather than tm files (a packet is the basic unit that the spacecraft sends down, where a tm file is a collection of packets) and you want to extract out all the data that lie within certain limits, then you can use:

```
populatePacsPoolFromDatabase(obsid, database, poolName
                            [,starttime=<number>, stoptime=<number>]
                            [,hklimit=<number>] [,sclimit=<number>])
```

Where `obsid` , `database` , `poolName`, `hklimit` and `sclimit` are as above; the optional time parameters are in units of finetime and are, obviously, the start and stop time to extract data between.

Note

Optional parameters in task calls: there are usually several optional parameters for tasks. The syntax we use to indicate that is [,hklimit=<number>], where the word inside the brackets is what you enter: a string (needing therefore ""); a number (integer or real); a calibration file (calfile); a boolean (True or False). Generally you can either use the wording hklimit=100 or just write 100 in the appropriate place in the call (the parameters listed in the call examples for the tasks discussed in this chapter should be in the correct order).

2.7.2. Getting the data

The ObservationContext can be extracted from your pool by (1) defining the pool, (2) identifying and extracting out the ObservationContext you want from that pool, (3) inspecting the ObservationContext and pulling out the bit of it you want, e.g. the Level 0 or 1 or 2; red or blue; observation data; auxiliary data; housekeeping data....

There are in fact several ways to do all of this, from the command line or GUIs. Here we explain some of the methods.

If you know the observation identifier (obsid) then you can use the command

```
obs=getObservation(obsid [,od=<number>] [,poolName=<string>]
    [,poolLocation=<string>] [,verbose=<boolean>] [,useHsa=<boolean>)
```

where obs is an ObservationContext, `obsid` is a number ending in "L", `od` is the number of the observation day, `poolName` is the name of the pool, `poolLocation` is the complete directory path to the location of the pool, `useHsa` is for external users, it will read the ObservationContext from the HSA. By default, if you only specify the `obsid`, it will search for the pool in a number of predefined locations with predefined naming conventions which are specific to MPE or Leuven. One of the default locations searched is also your home directory, e.g. .hcss/lstore. If an error "could not find OD number" is returned, you need to also specify the OD number. If you have your pool elsewhere entirely you can: specify `poolName` only, where it will look for that pool name in the default directories, or `poolLocation`, and it will look for the default pool names in that pool location, or both `poolName` and `poolLocation` for a unique pool name in a unique place. (Everything in [] is optional.)

The parameter `useHsa` is to be used if you wish to extract the data from the HSA. To do this your username and password must have been set, before you start HIPE, by placing into the file user.props, located in .hcss/, the following

```
hcss.ia.pal.pool.hsa.haio.login_usr = <string>
hcss.ia.pal.pool.hsa.haio.login_pwd = <string>
```

Now, let's say you want to first inspect all the ObservationContexts in a pool:

```
mypool = LocalPool(poolName) # a string
obslist = mypool.allObservations
```

and then click on obslist in the Variable panel to see what is in there, and to extract any out (into an ObservationContext). Alternatively you can extract parts out with:

```
myobs = obslist[0].product
# or [1] or [2]... depending on which part of obslist you want

# or get it straight from the pool, with the obsid you found from
# looking at obslist
myobs = mypool.loadObservation(obsid)
```

Or, more straightforward is to use

```
allObs = LocalPool("Kul_Pacs_Data_Pool_90_1", -"/STER/118/pacsman/
PacsPools").allObservations
```

And the click on allObs in the Variables panel to inspect it.

Note that the directory name, the pool name, specified by `poolName` will be created if it does not already exist. You can save an observation with

```
mypool.saveObservation(myobs [,verbose=<boolean>] [,poolLocation=<directory>]
   [,poolName=<poolname>] [,saveCalTree=<boolean>])
```

Where the final parameter allows you to save the calibration tree with the observation. Be aware that saving an observation may take a long time, because there are a lot of associated products to save. If instead you want to inspect everything that is in your pool via a GUI you can use the product browser GUI (although I have heard that this GUI will be deactivated soon):

```
obs = browseProduct(ProductStorage(mypool))
```

(Noting that another product browser GUI, for extracting out ObservationContexts, is the Data Access View, but to use that you need to have defined a storage, using the command mystorage=ProductStorage("poolname").)

Next you want to extract the particular part of the ObservationContext that you want. You can inspect "myobs" by clicking on it in the Variables panel to locate what bit of it you want. In this example we then extract out the 1st set of averaged red ramps that are in level 0.

```
rampr=myobs.level["level0"].refs["HPSAVGR"].product.refs[0].product
```

If you could see more than one HPSAVGR when you clicked on it in the Editor panel (when you click on myobs in the Variables panel it is sent to the Editor panel for further inspection) then you access these by specifying 1 or 2.. in the refs[0] part of the command.

It is necessary to extract a few other products in order for the pipeline processing steps to be carried out. Currently these are:

```
pp=myobs.auxiliary.pointing
orbitephem = myobs.auxiliary.orbitEphemeris
dmcHead=myobs.refs["level0"].product.refs["HPSDMCR"].product.refs[0].product
# for the DMC header for red data
# OR, if you are working from tm files (see 6.2 for what -"dfs" is)
dmcHead=extractDmc(dfs,channel=<"red" or -"blue">)
```

where again, refs[0] refers to the first thing that you put into "observation", and the number in the [] should correspond to the number in the [] that you extracted into "rampr" above. dmcHead is a *PacsDmcProduct* and is required for some of the masking tasks; pp is a *PointingProduct* and is require for the RA/Dec setting tasks; orbitephem is an *OrbitEphemerisProduct* and is the orbit ephemeris required for correcting Herschel's velocity. If these products are not available it simply means you can not run the relevant tasks on your dataset.

If your auxiliary products were not in your data pool, but held in a different one, then use the following commands to extract the pp:

```
# if you know where the auxiliary pool is and what it is called
myauxpool = LocalPool(poolName, poolLocation)
# for the calibration pool this syntax is necessary
mycalpool = getCalPool([verbose=True])
```

(For the last command, see Sec "Level 0 to Level 0.5".) And then inspect the pools and extract out your products e.g. with the product browser

```
pp = browseProduct(ProductStorage([mydatapool,myauxpool]))
```

this allows you to inspect the data pool and the auxpool, and you can locate the "pointing product" and extract it in the same way you do the ObservationContext.

2.7.3. Raw telemetry to Level 0

This is something that general users will never do. They can skip through to the next section. The steps here are things you will need to do if you work straight from tm files rather than the ObservationContext.

2.7.3.1. readtm - reading raw telemetry

Reading raw telemetry from a PacketRecorder archive file (tm file) is done as follows:

```
from herschel.pacs.share.util   import FileSelection

filename = FileSelection.getFilename()
seq = readtm(filename)
# or just
seq=readtm()
```

Where seq is a *PacketSequence* containing raw telemetry and/or TC SourcePackets. FileSelection will open a file selector box showing all files in your working directory; readtm() also does that, allowing you to select all files that end with "tm". If the `pacs.tm.datapath` property has been set (in .hcss/user.props) to an existing directory, the file selector box will be opened in that directory instead of the default home director. Alternatively you can use the command

```
filename=FileSelection.getFilenameInDirectory(dir, pattern) # strings
```

which does not need the `pacs.tm.datapath` property to have been set.

2.7.3.2. extractDataframes - decompress the science tm packets

This step generates the intermediate product decompressed science data. A compressed entity is distributed over many TmSourcePackets. This task collects all these, combines them appropriately, decompresses them and generates a dataframe. This dataframe is the raw result of the task, and is in a raw format, containing the compressed entitys' information and more.

```
dfs = extractDataframes(seq)
```

The result, dfs, is a *DataFrameSequence*, a collection of PacsDataFrame objects. These are the decompressed buffers of the two SPUs.

2.7.3.3. decomposeDataframes

This task organises the raw decompressed data into *Frames* and *Ramps* data structures.

```
pacsMix = decomposeDataframes(dfs [,channel=<string>] [,mode=<string>]
          [,fullDmc=<boolean>])
```

pacsMix is a product container. What is in here depends on your selection (as defined by the optional parameters `mode` and `channel`) and the instrument algorithms that were enforced**.** It can contain *Frames* and *Ramps* products. For *Frames* class the DecMec data are collapsed from the full readout sampling to the frequency of the reduced data (the frame). So the following applies:

- OBSID, BBID: get the first entry of the associated block of DecMec data
- LBL: gets the median (over the readouts) value and checks whether it is unique within a ramp
- VLD: gets the first value and checks whether it is unique within a ramp
- TMP1, TMP2, FINETIME, CRDC, CRCRMP, DBID, BSID: get the first value
- CPR, WPR, BOLST: get the mean value

The default for `fullDmc` is False. The frame is in the form of a data cube (note, cube here means in 3D, not a *Cube* class object), with the collapsed DecMec information, where the decoded Label information have been placed in the Status table (note: a Label is a tag that is given to the data sequence by the DecMec controller to mark where it is within the observing sequence, e.g. "chopper has moved", "grating has moved"). The *Ramps* product contain channel data e.g. raw or averaged ramps and that of the (rotating or maybe fixed) few additional raw channels, and the fully sampled DecMec data**.** The

BLINDPIXELS mask has already been applied to data extracted with this task; these are pixels that have been deselected with the Detector Selection Table.

The optional parameters of this task are: `channel`, a string with values "red", "blue" or "both" (default); `mode`, a string with values "frames", "ramps", "subramps" or "rawramps" and where the default is "all"; `fullDmc`, a boolean, True or False, to ask for the full resolution DecMec header.

Calibration file used (the use of which is hidden from the user): filterBandConversion. This task calls the tasks ExtractFrames which in turn calls photDfs2Frames, and it is there that the calfile filterWheel2Band (Common.FILTER_BAND_CONVERSION) is accessed.

2.7.3.4. readAttitudeHistory

Not yet available so information not complete

Read the attitude history.

```
attitude = readAttitudeHistory(pdf)
```

Reads the instantaneous pointing product covering the same time as the dataframes in pdf. "attitude" is a pointing product.

2.7.3.5. readTimeCorrelation

Not yet available so information not complete

Reads the time correlation information.

```
timecor = readTimeCorrelation(pdf)
```

Reads the time correlation product covering the same time as the dataframes in pdf. "timecor" is a *TableDataset* containing the time correction values.

2.7.3.6. Extract out the raw or averaged ramps

To get the *Frames* or *Ramps* product(s) which you will need in the next stage of the data reduction, type:

```
print pacsMix
# Is a listing of everything in pacsMix (see decomposeDataframes)
# Identify which frame or ramp you want; say Frame1 and Ramp1
frame=mix["Frame1"]
ramp=mix["Ramp1"]
```

In this way you can extract out the frame (fitted ramps), averaged ramps and/or raw ramps product, depending on what you want and what is present. In the listing of pacsMix: a *Frame* product will be described as a "Frames"; a raw *Ramps* product will be described as a "Raw Ramps"; an averaged *Ramps* product will be described as a "Complete (Sub-) Ramps".

2.7.4. Level 0 to Level 0.5

The PACS SPU has two onboard data reduction modes for spectroscopy: slope fitting of the integration ramps (to create a *Frames* class product) and averaging the samples of the integration ramps (which is a *Ramps* class product). The reduction steps in this section start with averaged or raw ramps but rapidly move on to fit ramps; therefore those starting with Level 0 fit ramps should also start reading here.

Please note that in the task calls described below we will often use the convention outRamps=Task(inRamp). However, note that for "outRamp" you could also type "inRamp", i.e. it is not necessary to place the result of the task in a different product to the product the task was run on.

We use this convention simply to make clear what went in and what came out. Hence, the outRamp for one task is the same inRamp for the next task.

The very first thing to do is to define the calibration tree:

```
mycalTree=getCalTree("FM")
```

FM here means flight model, and it is also the default entry. FS would be if you were working on flight spare data. You will notice that for many of the flagging tasks which are described next, passing the calTree is an option, where you either pass the calTree or individual calfiles. *We strongly recommend you always pass the calTree parameter anyway.*

Note: HIPE contains a function to load any version of a calibration product into memory: getCalProduct(). The following example should make this clear. If you load a calibration tree and print the photometer branch, the current versions of the products are printed. So,

```
fm=getCalTree()
print fm.photometer
```

will give you output similar to

```
PacsCalPhot Calibration Products:
...
invntt -: FM, 1
invnttBL -: FM, 2
invnttBS -: FM, 2
invnttRed -: FM, 2
...
```

Loading a previous version of a calibration product can be done as follows:

```
cal = getCalProduct("Photometer", -"InvnttBL", 1)
# and calFileVersion can be used to check which versions of calibration
# products you have
print cal.calFileVersion 1
print fm.photometer.invnttBL.calFileVersion 2
```

If you want to replace the calibration product temporarily in your calibration tree, e.g. for passing into pipeline steps, you can simply replace the appropriate product as follows:

```
fm.photometer.invnttBL = cal
print fm.photometer.invnttBL.calFileVersion 1
```

2.7.4.1. compareRawWithReducedDataRamps

Note a pipeline task

This task is a quality control step and creates an information mask. It performs a check on the onboard (SPU) reduction, and not something general users will use. It requires the raw and averaged ramps to be present.

```
outRamp = compareRawWithReducedDataRamps(inRawramp, inAverageramp [,copy=<number>])
# where inRawramp and inAverageramp were extracted as was the -"ramp" in Sec. 6.6
```

The inRawramp is a *TRamps* class product, being raw ramps, and inAverageramp is of class *ARamps* (both of these are types of *Ramps*), being averaged ramps. When you get averaged (or fit) ramps data, for free for 3 pixels you will also get raw ramps, and these will be held in a raw ramps *Ramps* product of your ObservationContext (which you must naturally extract out). For these 3 pixels the task averages their raw ramps and compares them to their on-board averaged ramps: a check that the on-board averaging was done as expected. It sets a flag in the mask OBSWERR if a deviation is found between the two. The mask is set for a detector pixel when a difference of more than 1 in the integer-converted values of the two ramps is found for that pixel; such a difference could be due to rounding accuracy. This mask is quality control oriented and should not be applied during the pipeline processing but carried through, and one should follow up for the reason for the deviation.

There is also a compareRawWithReducedDataFrames task, in which inFitramp will be compared to inRawramp.

Depending on whether `copy`=0 (default) or 1, outRamp is, respectively, a copy of inAverageramp with a new mask added, or a copy of the inAverageramp with no mask added.

2.7.4.2. specFlagSaturationRamps

```
outRamp = specFlagSaturationRamps(inRamp [,calTree=<mycalTree>]
          [,rawRamp=<Ramps>] [,rampSatLimits=<calfile>]
          [,copy=<number>] [,qualityContext = qualityContext])
```

Detects ramp readout values that are above the saturation limit and puts a flag in the corresponding SATURATION mask for those readouts. Reads the saturation ADU value from a calfile, where the four capacitance values and the power supply groups are distinguished (using data taken from the FM-ILT test report PICC-MA-TR-043). The calfile contains conservative values close to the saturation limit because sometimes the ramp values start rising again after hitting the saturation limit. `qualityContext` is a quality control product; currently it is by default a null file and you can ignore it. It is also possible to ask the task to check for saturation from the few downlinked raw pixels, by specifying the parameter `rawRamp`. This fills the boolean Status column "RAWSAT" (true if saturation is there detected) and creates an additional mask "RAWSATURATION", which is set to true for all pixels of a reset interval if the raw ramp is found to be saturated for that readout, and within the task the mask is set to inactivate.

Raw ramps: The task loops through all pixels and resets and find the readouts which ADU digits are just below the corresponding saturation limit and flags them in the SATURATION mask.

Averaged sub-ramps: 1) Calculates the median of pairwise differences and adds half of the pairwise difference to each ramp value (each datapoint)#find the first ramp readout index where the signal is just below the saturation limit, flag this and the rest of the ramp. 2) Assume that first pairwise difference of ramp is not saturation affected, construct a model ramp using this pairwise difference and add half of the pairwise difference to each ramp value#find the first ramp readout index where the signal is just below the saturation limit, flag this and the rest of the ramp. The task tries method 1 and if that does not work, then method 2.

There is an equivalent task than runs on a *Frames* product, specFlagSaturationFrames, and if you extracted from the Level 0 the *Frames*, rather than *Ramps*, product, you should run this task after fitRamps (Sec. 7.3). The only difference is that since it runs on data with slopes values in them, it looks for where the signal, in V/s, exceeds the total possible dynamic range. Of course, the signal saturation limits depend on the reset interval (the ramplength); the saturation limits in the calfile refer to a 1 sec reset interval and the units are ADC (readouts/s). Unfortunately, this method applied to a *Frames* product can not absolutely guarantee that all/real saturation points will have been found, since after saturation the signal will start to decrease and these points will not be found with this method; however, a flagging of high values will serve as a warning that saturation may have occurred.

Note

The best way to find saturation is to look at saturated raw ramps and find the signal limits dependent on capacitance and reset interval (TBD) and put this information in a calfile; e.g. use a grating scan of bright emission line, because there the signal should drop rapidly after reaching saturation because the saturated part of a ramp is flat.

The use of open and dummy channel in saturation detection: deviations of their signals from normal noise is an indication of an FEE functional error, of which saturation is one likely cause; one could raise software warnings if this is detected

Calibration file used: rampSatLimits, the limit (in digits) per pixel and capacitance. As with the previous flagging tasks, either pass the `calTree` or pass the calfile.

2.7.4.3. fitRamps

```
outFrame = fitRamps(inRamp [,degree=<number>] [,firstReject=<number>]
           [,lastReject=<number>])
```

This task fits the slope of the integration ramp to convert to signal [readouts/s]. The output, outFrame, is a *Frames* class product, in which are placed the fit signals and the fit uncertainties. Propagates/rebins any attached masks from per ramp readout to a per reset interval. Propagates/rebins status words to a status word at the reset interval frequency.

- OBSID, BBID: get the first entry of the associated block of DecMec data
- LBL: gets the median (over the readouts) value and checks whether it is unique within a ramp
- VLD: gets the first value and checks whether it is unique within a ramp
- TMP1, TMP2, FINETIME, CRDC, CRCRMP, DBID, BSID: get the first value
- CPR, WPR, BOLST: get the mean value

All masks are also propagated by this task. In addition, it creates the mask BADFITPIX, flagging pixels where fits somehow failed (note that this mask is not created if the ramps were fitted on-board and you start working on the HPSFITR/B product). Masks are applied as follows: a master mask is constructed using the masks present in the *Ramps* object except for OBSWERR and DEVIATINGOPENDUMMY. Mask flag set to true are also not considered in the master mask. The master mask is then applied to the indices and readouts before the signal is calculated.

`first/lastReject` are the (integer) number of ramp readouts to exclude from the fit, e.g. it may be a good idea to exclude the first few readouts: more detail is given later in this chapter.

Note that for all and any masked pixels, masked ramps and masked readouts, even when the mask is considered by this task that entire ramp is still fit, individual readouts are not excluded from the fitting. However, the fitted slope value of the readout still carries the mask, so the user can later decide whether to ignore that data point in subsequent pipeline tasks.

Before running this task you should deactivate all masks:

```
inRamp = activateMasks(inRamp, String1d([" -"]), exclusive = True)
```

2.7.4.4. specConvDigit2VoltsPerSecFrames

```
outFrame = specConvDigit2VoltsPerSecFrames(inFrame [,calTree=<mycalTree>]
           [,readouts2Volts=<calfile>] [,copy=number])
```

This task converts the signal to V/s for inFrame:

frameSignal = frameSignal * 256Hz/rampLength

where the rampLength is the length of the ramps or subramps that were enforced during the observation (information which the frames product retains). The conversion for SPU signal values for the frame depends on the reset length (and, in case of subramp fitting, the number of subramps). The SPU takes the length of the ramp it fits (i.e. the complete ramp or the subramp) as unity. The slope numbers it produces are therefore digital units per second, or digital units per numberOfRampsPerSecond. The calfile SpecVolts is then used to convert the units further to Volt/s:

frameSignal = -1*frameSignal * (endVolt - startVolt) / (endDigit - startDigit)

the unit V/s of the signals is set by using the herschel.share.unit methods. Multiplication by -1 is so the units after fitRamps are correct.

The task takes into consideration differences between the four power supply units but only in the volt-to-readout conversion, for the red and blue arrays (each having two power supplies).

There is an equivalent task that runs on a Ramps product, specConvDigit2VoltsRamps. This converts the digital readouts to Volts and also multiplies by -1 so we will have positive signals after the ramps are fit.

Calibration file used: readouts2Volts (default value of `readouts2Volts`). Pass either this calfile or set the `calTree`.

2.7.4.5. detectCalibrationBlock

This task simply identifies the calibration blocks (i.e. where they lie in the data time-line) and fills the CALSOURCE entry in the status table.

```
outFrame = detectCalibrationBlock(inFrame)
```

2.7.4.6. specExtendStatus

```
outFrame = specExtendStatus(inFrame [,calTree=<mycalTree>]
           [,ChopperThrowDescription=<calfile>] [,copy=<number>])
```

This task adds useful information to the status table. These information are:

-GRATSCAN: a counter of grating scans, negative for down scans

-CHOPPER: a combination of CHOPPERPLATEAU (science plateau: 0, 1 [on] or 2 [off]) and CALSOURCE (plateau on a calibration source: 0, 1 [CS1] or 2 [CS2]) status entries that results in 0: no plateau, or 1: science on or CS1, or 2: science off or CS2

-CHOPPOS: +centre, small, medium, large, CS1, CS2

these being defined in a calfile as a String1d. `calTree` and `copy` are as described before.

Calibration file used: chopperThrowDescription; either set this parameter or set the `calTree`.

2.7.4.7. addUtc

```
outFrame = addUtc(inFrame, timecor, [,copy=<number>])
```

Converts from spacecraft on-board time (OBT) to coordinated universal time (UTC) using the time correlation table (UTC, correlation gradient, correlation offset). Fills the UTC field in the *Frames* dataset. `copy` means as before, and `timecor`, the time correlation table, is a *TableDataset.*

2.7.4.8. specAddInstantPointing

```
outFrame = specAddInstantPointing(inFrame, pp, [,calTree=<mycalTree>]
[,copy=<number>]
          [,siam = siam][,orbitEphem=orbitEphem] [,horizons=horizons][,isSso =
isSso]
          [,noInter=noInter][,useGyro = useGyro])
```

This task associates the PACS centre-of-field coordinates and the position angle (the RA and Dec of the central detector pixel) to the raster point counter and/or nod counter of the input frame (which it adds to depends on which are present, which depends on the AOT type), all of these being associates of the frame and are necessary to track which raster or nod the associated readouts belong to. The task is the same for photometer and spectrometer, and we refer you to the photometer chapter to learn more about how it works. `copy` means as before, and pp, the pointing product, is from Sec. 5.2.

By default the Filtered Pointing information is used, but also the gyro propagated pointing information may be used. This is done by using the Frames status entry FINETIME and extract the associated information from the PointingProduct. Also the SIAM matrix is applied and aberration is done (if the proper Products are passed). The result is added to the status entry of the Frames Product.

Parameters: The orbit Ephemeris Product needed for aberration correction of non SSO objects. Horizons Product is needed for the aberration correction of SSO objects. `isSso` is "is it solar system" (default False/0)? `noIter` is whether to interpolate or not. `useGyro` (False/0) is to use the Gyro propagated information instead of the filtered.

isSso (default False/0) is a switch for whether the target is solar system.

Calibration file used: siam

2.7.4.9. convXyStage2Pointing

Not a pipeline task

```
outFrame = convXyStage2Pointing(inFrame, seq [,noInter=<boolean>] [,copy=<number>])
```

This task is relevant only for ILTs (instrument level tests), where an "XY stage" was used to allow satellite pointing to be simulated. It is the same step for the photometer, so see photometer standard data processing chapter for a more detailed description. `copy` is as described before, `noInter` is a boolean to specify to adopt (True) or not (False: default) interpolation in the fitting, seq is the same as extracted from the data in Sec. 6.1.

2.7.4.10. convertChopper2Angle

```
outFrame = convertChopper2Angle(inFrame [,redundant=<number>] [,calTree=<mycalTree>]
           [,chopperSkyAngle=<calfile>] [,chopperAngle=<calfile>]
           [,chopperAngleRedundant=<calfile>] [,copy=<number>])
```

This task calculates the chopper position angle with respect to (i) the FPU optical zero and (ii) the angle on sky. It reads the status DecMec parameter "CPR" and then populates new Status words#CHOPFPUANGLE and CHOPSKYANGLE#in the returned frame. Angles are in units of armin. One should execute this task even if these was no chopping action during the observations, because the chopper would have been in at least one position and that angle needs to be recorded. `copy` means as it has always before, `redundant` takes on 0 (use FPI, the default nominal field plate calibration) or 1 (FPII, the redundant field plate).

Calibration files used: chopperSkyAngle, the linear conversion between FPU angle and position on the sky; chopperAngle (the non-linear conversion between CPR and FPU angle, containing the nominal readouts-to-angle calibration for the chopper); chopperAngleRedundant (ditto for the redundant unit, to be used only if the redundant unit ever is). As with previous tasks, either set these calfile parameters or set the `calTree` parameter.

2.7.4.11. specAssignRaDec

```
outFrame = specAssignRaDec(inFrame [,calTree=<mycalTree>]
[arrayInstrument=<calfile>]
           [,moduleArray=<calfile>] [,copy=<number>])
```

This task takes the pointing previously added for the central pixel, of the centre of the field-of-view coordinate, and assigns an RA and Dec to every pixel. If it finds the PV calibration which allows one to distinguish between the chopper throws small, medium, large, then while looping over the resets it picks the spatial calibration for that throw. The relative offsets of each spaxel with respect to module 12 is calculated. Then the absolute positions of the spaxels in RA/Dec on sky is determined using the jsky.coords.WCSTransform tool of ESO and taking into account the roll angle. It returns a frame with RA and Dec coordinates added.

Note that the spatial calibration is still being validated.

Calibration files used: arrayInstrument (coordinate conversion from array to sky); moduleArray (coordinate conversion from module to array). These are the default for their corresponding task parameters. Either set these parameters or set the `calTree`.

2.7.4.12. waveCalc

```
outFrame = waveCalc(inFrame [,filter=<string>] [,calTree=<mycalTree>]
           [,littrowPar=<calfile>] [,copy=<number>])
```

This is a task that calculates the wavelength corresponding to a grating position. For every pixel the Littrow equation is evaluated. A 3rd order polynomial relates the grating position to alpha (incoming angle in the Littrow equation) per pixel. `filter` is a string, being "R1", "B2", "B3" for the red array filter, blue array green filter, and blue array blue filter.

Calibration file used: littrowPolynomes, which contains the coefficients of fitted 3rd order polynomial of the conversion of grating position to wavelength. As with all other tasks, either specify this calfile or the `calTree`.

2.7.4.13. specCorrectHerschelVelocity

```
outFrame = specCorectHerschelVelocity(inFrame, orbitEphem, pp)
```

This task corrects the wavelengths for Herschel's velocity. The parameters you pass, the products pp and orbitEphem, were introduced in Sec. 5.2. If you were not able to extract out these products, then don't run this task. It wont affect anything except the accuracy, at a low level, of your wavelengths.

2.7.4.14. findBlocks

```
outFrame = findBlocks(inFrame [,copy=<number>])
```

The work of this task allows the subsequent steps to find the applicable calibration source measurement(s), the nod scans to differentiate, etc... A summary of the major 'blocks' in the observation is constructed and put into a BlockTable. This includes how many chop#nod cycles there were, how many raster were taken, etc. It summarises information already contained in the Status. In a first version this is done based on the LBL status word (the Label), the raster point counter and status information. `copy` is the same as in all other tasks described so far.

A new block is identified for every change in:

- nod/raster position
- grating scan direction
- detector parameters, e.g. reset interval, integrating capacitance, ...

For a more detailed description see the photometer standard data processing chapter, as step is the same for the photometer and spectrometer. (In the photometry chapter this is currently in Sec. 13.7.2, in the "Level 0 to 0.5" section)

2.7.4.15. specFlagBadPixelsFrames

```
outFrame = specFlagBadPixelsRamps(inFrame [,calTree=<mycalTree>]
          [,badPixelMask=<calfile>] [,copy=<number>])
```

This task flags the permanently damaged pixels and puts them in the BADPIXEL mask. outFrame, inFrame, and `copy` are the same as for previous task descriptions. If the BADPIXEL mask does not exist, it will be created. In principle, permanently damaged pixels should be known from module-level tests, but they could multiply during flight. At present this task only flags bad pixels read in from the calfile, it does not look for new ones. We expect three kinds of bad pixels (*different masks for different sorts of bad pixels? TBD*):

- dead pixels: behave like an open channel in that signal does not change with infalling flux#these should pop up in measurements of the calibration sources (as the pixel to pixel variations, the flat-field, should be known and be in the calfile)

- spiking pixels: show strong signal variations with time#could be identified in the same way that glitches are
- weird (noisy) pixels: high noise level and maybe varying with time#identify with the help of noise limits in the calfile

There is a difference between B and BADPIXEL masks:

- pixels selected out by the Detector Selection Table are flagged and put in the BLINDPIXELS mask during Level 0 data generation
- additional bad or damaged pixels (which may change with time) are flagged out during this step and put in the BADPIXEL mask

(There is also a specFlagBadPixelsRamps task, the details of which are exactly the same as here.)

Calibration files used: flatfield, noiseLimits, badPixelMask. The latter is the default name for the optional parameter `badPixelMask`, and you must either pass this parameter or pass the `calTree`.

2.7.4.16. cleanPlateauFrames or flagChopMoveFrames

```
outFrame = flagChopMoveFrames(inFrame, dmcHead=<dmcHead> [,calTree=<mycalTree>]
          [,redundant=<number>] [,chopperJitterThreshold=<calfile>]
          [,chopperAngle=<calfile>] [,chopperAngleRedundant=<calfile>]
          [,qualityContext=<calfile>] [,copy=<number>])
```

Note: I think that what was called cleanPlateauFrames has been renamed to flagChopMoveFrames.

This task masks unreliable readouts at the chopper transition phases, that is data taken while the chopper is still moving. It searches for chopper plateaux (i.e. where the chopper is not moving), and masks the readouts that deviate by more than specified thresholds from the median plateau position. The mask added called UNCLEANCHOP. The parameters `chopperJitterThreshold`, `chopperAngle`, `chopperAngleRedundanct` are optional inputs, and can be set to the names of calfiles to use: it is strongly advised you do not change these unless you know what you are doing (because the calfiles you specify have to be in the right format and contain correct information). `qualityContext` is a quality control product; currently it is by default a null file and you can ignore it. The parameter `copy` is as described before, and `redundant` is an integer which sets whether the nominal field plate calibration is to be used (0 for FPI: default) or the redundant (the on-board spare) is to be used (1 for FPII).

Be warned that this task will run without specifying `dmcHead` but (and it will not tell you this) the results will be wrong.

(There is also a cleanPlateauRamps task, the details of which are exactly the same as here.)

Calibration files used: chopperJitterThreshold (contains the specs of the allowed deviations from the final chopper positions for the science and calibration windows); chopperAngle (the calibration from chopper position values to FPU (focal plane unit) angle); chopperAngleRedundant (the calibration for redundant unit, the redundant unit being an on-board copy of PACS electronics...this task parameter is unlikely to ever be required). It is necessary to pass either the `calTree` or the `chopperJitterThreshold` and `chopperAngle` (or `chopperAngleRedundant`).

2.7.4.17. flagGratMoveFrames

```
outFrame = flagGratMoveRamps(inFrame, dmcHead=<dmcHead> [,calTree=<calTree>]
          [,gratingJitterThreshold=<calfile>] [,qualityContext=<calfile>]
          [,copy=<number>])
```

This task masks ramp readouts at grating transition phases, creating a mask called GRATMOVE. It calculates the median grating position for each reset/readouts using the full resolution status grating position information, and masks the individual readouts where a deviation of more than the specified threshold is found. The parameters `copy` and `qualityContext` are as described before.

Be warned that this task will run without specifying `dmcHead` but (and it will not tell you this) the results will be wrong.

(There is also a cleanPlateauRamps task, the details of which are exactly the same as here.)

Calibration file used: gratingJitterThreshold, which contains the specs for the allowed deviations from the final grating positions. You can either set it by passing the calfile or by passing the `calTree`.

2.7.4.18. flagDeviatingOpenDummyFrames

Not a pipeline task

```
outFrame = flagDeviatingOpenDummyFrames(inFrame [,sigma=<number>] [,copy=<number>])
```

This task looks at the behaviour of the open and dummy channel ramps and sets a mask flag (DEVIATINGOPENDUMMY) for the corresponding module if the ramps show a weird or deviating behaviour. (This is more an information mask than one to be considered in the subsequent data reduction steps.) Currently this is done by comparing the mean and standard deviation of the pairwise differences of each ramp with the overall mean and standard deviation of the pairwise differences of all ramps of all open and dummy pixels. If behaviour such as non-zero or oscillating ramps in the open channel are seen, this could indicate saturation or other problems in the FEE electronics.

Note

It may be necessary to introduce a calibration file that contains the normal noise thresholds of the ramps of the dummy and the open channels to be able to detect non-normal deviations; this has not yet been done. This could also be used for a long-term quality control of the ramp shape (deviating first readouts, debiasing).

However, ...a closer investigation of the open and dummy channel signals revealed cross-talk effects at the #3% level: chopped and non-chopped data on the open and dummy channels have different standard deviations (factor about 100!), which could mean that the chopper cycle is visible in the signal of the open and dummy channels (otherwise these would be no difference seen). Also, major differences in the signal level introduced by different power supply groups has been established: this means we cannot simply track, long-term, the noise values for each channel and store them in a calfile but rather we have to determine a global noise level for each measurement distinguished by power supply group, and track those. This presumes that most of the modules and/or resets show normal behaviour. These points also mean that if we wish to understand the cross-talk better, we need more tests.

The default value of `sigma`, the factor of the standard deviation outside of which signal variations are considered deviant, is 3. `copy` is as described before.

(There is also a flagDeviatingOpenDummyRamps task, the details of which are exactly the same as here.)

2.7.4.19. pairDiffSigClip

This is not part of the pipeline, but an independent task. It is an alternative to fitRamps, but it is not supported

```
outFrame = pairDiffSigClip(inRamp [,sigma=<number>]
[,ignoreSaturationMask=<boolean>]
           [,ignoreUncleanChopMask=<boolean>][,ignoreGlitchMask=<boolean>]
           [,ignoreGratMoveMask=<boolean>])
```

This is a task for calculating the signal of an averaged or raw ramp product to produce a frame [unit V/s], i.e. it is a parallel task to fitRamps. It does this by working out the sigma (which value [a Double] can be set by the user; default value 3) clipped pairwise differences of the input ramp. It stores these signals and the fit uncertainties (the standard deviation of the sigma-clipped array) in the output frame. It propagates/rebins masks from a flag per ramp readout to a flag per reset interval. Propagates/rebins masks per ramp readout to a flag per reset interval, and also propagates/rebins the status words to a status word at the reset interval frequency: this is as also done by fitRamps.

Masks are applied as follows: a master mask is constructed using the masks present in the Ramps object except OBSWERR and DEVIATINGOPENDUMMY. Additionally, those masks having `ignoreXXX` set to True are not considered in the master mask. The master mask is then applied to the indices and readouts before the signal is calculated.

2.7.4.20. pairDiffHodLehEst

This is not part of the pipeline, but an independent task. It is an alternative to fitRamps, but it is not supported

```
outFrame = pairDiffHodLehEst(inRamp [,ignoreSaturationMask=<boolean>]
           [,ignoreUncleanChopMask=<boolean>]
           [,ignoreGlitchMask=<boolean>][,ignoreGratMoveMask=<boolean>])
```

This task calculates the signal [V/s] by applying the Hodges#Lehmann estimator on pairwise differences of each ramp: it is also a task parallel to fitRamps. The H—L estimator is defined as the median of the mean of pairs of a pairwise differences array, or in other words it calculates a stable mean. These signals and the fit uncertainties (standard deviation) are stored in outFrame. Propagates/rebins masks per ramp readout to a flag per reset interval, and also propagates/rebins the status words to a status word at the reset interval frequency: this is as also done by fitRamps.

Masks are applied as follows: a master mask is constructed using the masks present in the Ramps object except OBSWERR and DEVIATINGOPENDUMMY. Additionally, those masks having `ignoreXXX` set to True are not considered in the master mask. The master mask is then applied to the indices and readouts before the signal is calculated. By default no mask is applied.

2.7.5. Level 0.5 to Level 1

The tasks taking data up to Level 0.5 are ramp fitting, flagging and organising of status information. No user interaction is necessary, thus it is more than possible to extract Level 0.5 products from your ObservationContext and begin your data reduction here.

2.7.5.1. specFlagGlitchFramesQTest

While it is not strictly necessary to run this#you could run your own glitch removal task#it is strongly recommended and is part of the pipeline

```
outFrame = specFlagGlitchFramesQTest(inFrame [,copy=<number>]
[,qtestwidth=<number>]
           [,thresholds=<number>] [,qtestlow=<number>] [,qtesthigh=<number>]
           [,splitChopPos=<boolean>)
```

This is a task that masks responsivity jumps that are (presumed to be) due to glitches (aka cosmic rays and other sudden and unwanted events). It works at the slope level i.e. on a frame. The task creates a "GLITCH" mask, which must not already exist. It works on the entire time sequence of the slopes (sl), for each pixel individually. The statistical test that is used to look for outliers is the Q-test. The basic way the task works is:

- Compute two differential signals: dsl = sl[i] - sl[i-1] and ds2 = sl[i+1] - sl[i-1], where i goes from first to last datapoint.
- Compute two contrast functions based on Q-tests of dsl and ds2, giving q1 and q2 respectively. The principle is to run the Q-test over a 'box' containing a fixed number of data points (of width w) and to slide the box over the whole time sequence (so each slope is visited w times). Each data point, i.e. each slope, gets w values of the Q-score. The value of the "contrast" is taken as the highest of those values: the result of this stage is an array of contrast values.

- Apply some thresholding, specific to the kind of events produced by glitches and responsivity jumps (spikes w/ and w/o decays, and staircases#details that have been worked out from ILTs and PV data). There are four threshold parameters currently used: t0, t1, t2, t3.
- For any detector, a given slope i is marked as affected by a responsivity jump in the following cases:

1. q1[i] > t0 and i-1 not flagged as jump

   this identifies the strongest events (computationally) quickly and where they would be missed because of other strong events close by

2. q1[i] or q1[i+1] > t1

   q2[i-1] or q2[i+1] > t2

   q2[i-1] and q2[i+1] > t3

   q2[i-1] and q2[i+1] have opposite sign

   these identify 'isolated' events, i.e. the classical spikes affecting only one ramp, with weak or no long-term effect on the responsivity (more typical for low stress detectors, i.e. the blue detector)

3. q1[i] > t1

   q2[i] or q2[i-1] > t2

   q2[i] and q2[i-1] > t3

   q2[i-1] and q2[i+1] have identical sign

   these identify 'step' events, i.e. those leading to a long term modification of the responsivity (more typical for high stress detectors, i.e. red detector)

4. if both neighbours of a slope are "events": because sometimes in these cases the central slope is not also flagged as a glitch (which it should be), we especially mark the central one

Examples of use of this RJ-flagging algorithm can be found in PICC-KL-TN-023. Although `qtestwidth` (integer: value of w), `thresholds` (Double1d: values of t0,t1,t2,t3), `qtestlow` (integer: the number of low slope values to exclude from the Q-test in the box) and `qtesthigh` (integer: the number of high slope values to exclude) are optional parameters, at least for the first two we strongly recommend you do not change them from default unless you are sure you know what you are doing.

The default assumption is that the frame this task is working on has chopping in it, and this is accounted for by the task as it considers the various chopper positions as separate timelines and applies the deglitching to each of those independently. If you are working on non-chopped data then you can set the optional parameter `splitChopPos` to False (the default is True, which means the data are with chopping). In any case, glitch detection is still something being tested. Testing with data from staring observations on a dark sky field, we find that this task works very well in finding slopes that in the raw ramps can be seen as obviously glitched. In addition it sometimes flags some post-glitch ramps, even if these ramps have a slope value is within the scatter of the rest of the slope values.

Currently this tasks requires that you deactivate all masks before calling it

```
inFrame = activateMasks(inFrame, String1d([" -"]), exclusive = True)
```

2.7.5.2. specEstimateNoise

```
 outFrame = specEstimateNoise(inFrame, [binWidth=<integer>] [,copy=<0|1>])
```

A jython task that estimates the noise at Level 1 for each pixel and fills the Noise dataset. First it selects the frames according to chopperplateau position 1, 2 or 0. It computes the median filtered signal (using bins with bin width `binWidth`) after discarding the readouts masked by the Master mask, as these could propagate and fake very high noise in the neighbouring readouts. Running over all readouts, the bin-medians are subtracted from each readout in the bins. The noise is then the square root of the signal.

2.7.5.3. specCorrectCrossTalk

currently not part of the pipeline, as the calibration information for it to work are not available.

```
outFrame = specCorrectCrossTalk(inFrame [,calTree=<mycalTree>]
           [,crossTalkMatrix=<calfile>] [,copy=<number>]
           [qualityContext=<smthng>])
```

This task subtracts the cross-talk contribution to a pixel from its neighbouring pixels, before any linearity correction is applied. It reads the cross-talk ratios from a calibration file and subtracts from every pixel the fraction of the signal that comes from a cross-talking neighbour. `copy` and `qualityContext` are as described before.

The calibration file should be created and edited with the script makeCalCrossTalkMatrix.py. The calfile will contain an *ArrayDataset* with two *Double2d* arrays in it, one for the red and one for the blue channel. The dimensions of the Double2d will be [number of cross-talks found, 5) where the values are:

[k,0] : the row value of the original pixel

[k,1] : the column value of the original pixel

[k,2] : the row value of the cross-talking pixel

[k,3] : the column value of the cross-talking pixel

[k,4] : the cross-talking ratio

Example: let us assume that 50% of the signal of pixel (4,7) spreads to pixel (0,0) and 13% of the signal of pixel (8, 10) appears in pixel (5,1). Then the array will look like:

[0,0] = 4

[0,1] = 7

[0,2] = 0

[0,3] = 0

[0,4] = 0.5

[1,0] = 8

[1,1] = 10

[1,2] = 5

[1,3] = 1

[1,4] = 0.13

Thus the crosstalk corrected values of pixel (0,0) and (5,1) would be

realsig(0,0) = realsig(0,0) - 0.5*sig(4,7)

realsig(5,1) = realsig(5,1) - 0.13*sig(8,10)

These simple subtractions are applied by this task for the whole signal array.

Calibration file used: CrossTalkMatrix (*which is currently a dummy filled with 0s*). Specify it or the `calTree`.

2.7.5.4. specCorrectSignalNonLinearities

```
outFrame = specCorrectSignalNonLinearities(inFrame [,calTree=<mycalTree>]
           [,nonLinearity=<calfile>] [,copy=<number>])
```

This task corrects for intrinsic non-linearities in the shapes of the raw ramps of each pixel. The correction is a 2nd order polynomial fit using the coefficients from the calfile. This calfile was based on testing done on the shapes of raw ramps, which produced a correction that could be applied to *Frames* data.

Calibration file used: nonLinearity. Specify it or the `calTree`.

2.7.5.5. convertSignal2StandardCap

```
outFrame = convertSignal2StandardCap(inFrame [,calTree=<mycalTree>]
           [,capacitanceRatios=<calfile>] [,copy=<number>])
```

This task reads the capacitance ratios calfile and scales all the signals in the frame to the lowest available integration capacitance, which is referred to as the standard capacitance. This is done because the subsequent flux calibration and dark subtraction tasks use calibrations based on data taken at the smallest capacitance value. It will also allow one to compare signals (from different observations, for example) that were recorded using different integration capacitances.

Calibration file used: capacitanceRatios, the measured capacitance ratios (capacitance w.r.t the smallest capacitance), for each pixel. Specify it or the `calTree`.

2.7.5.6. specDiffCs

```
csResponseAndDark = specDiffCs(inFrame [,calTree=<mycalTree>]
                    [,calSourceFlux=<calfile>]
                    [,relCalSourceFluxProduct=<calfile>])
```

For each pixel, this task computes, for the calibration blocks(s) of your inFrame, the statistics (mean and standard deviation) of the pairwise differences between the two chopper position (CS1 and CS2), for each grating position. (Adopting chopping AB for OBCP 13 and ABBA for OBCP 35, in the case of only 1 ramp per chopper plateau). You can chose yourself to say which masks to include (True) or ignore (False, the default). The output is a product which contains, for every pixel, the dark value at the key wavelength of the calibration block and one or three responses: three in the blue (one for each key wavelength for the bands B2A, B2B and B3A) and one in the red (which only has band R1). These values are calculated by comparing the flux difference between the calibration sources with their calfile recorded flux differences. We refer internal PACS members to Report PICCKLTN034 if they want to know more.

specDiffCs also computes the errors on the response and dark, even if the Noise dataset is absent in the input *Frames*.

We recommend the following activateMask call

```
inFrame = activateMasks(inFrame,
String1d(["UNCLEANCHOP", -"GLITCH", -"BADFITPIX"]),
   exclusive = True)
```

Calibration files used: calSourceFlux: the measured flux of calibration sources in Jy (*not yet available*); relCalsouceFlux; keyWavelengths

2.7.5.7. specFitSignalDrift

```
responseDrift = specFitSignalDrift(inFrame, csResponseAndDark=<product>)
```

This task will use the output of specDiffCs and establishes the pixels' responsivity during the entire observation. It uses the starting response values determined by specDiffCs, and using the signal of the off-source chopper plateaux, it tracks the drift in the response during the observation. It fits a background spectrum to all the off-source points together after deselecting actively masked datapoints. Therefore we recommend the following activateMask call

```
inFrame = activateMasks(inFrame, String1d(["BADPIXELS", -"GLITCH",
   -"SATURATION", -"BADFITPIX"]), exclusive = True)
```

2.7.5.8. decodeLabel

Not a pipeline task

```
outFrame/outRamp = decodeLabel(inFrame/inRamp [,copy=<number>])
```

This is not a task that is required any more, as it is applied inside decompseDataframe, but for completeness we keep the description. `copy` means the same as it has before, and this task will work on *Frames* or *Ramps* class objects.

Converts the Label entry into a verbose form and puts it into the Status; it is the same for the photometer and the spectrometer. (note: a Label is a tag that is given to the data sequence by the DecMec controller to mark where it is within the observing sequence, e.g. "chopper has moved", "grating has moved").

-DMCSEQACTIVE Bit1

-SCIENCEPLATEAU Bit 2-4

-CALPLATEAU Bit 7-8

-SCANDIR Bit 5, non-zero: positive for up, negative for down, counting the scans

-WASWITCH Bit 6

-WASWITCHPOS

2.7.5.9. addOBCP2Frames

Not a pipeline task

```
outFrame = addOBCP2Frames(inFrame, seq, [,copy=<number>] [,hkObcp=<calfile>])
```

Included here for completeness. The parameter `hkObcp` is a *TableDataset* and contains the OBCP (on-board control procedure) parameters gotten from HK data (default value for this is "hkObcp"). This task adds the DP_WHICH_OBCP numbers to the Status dataset of a *Frames* object. It is now done within decomposeDataframes. seq is as from Sec 6.1, and `copy` means the same as before.

2.7.5.10. specSubtractDark

Not yet mature

```
outFrame = specSubtractDark(inFrame [,csResponseAndDark=<product>]
          [,calTree=<mycalTree>][,copy=<number>])
```

This task is for AOTs of type wavelength-switching and off-mapping. It subtracts the dark current previously determined by specDiffCs.

2.7.5.11. subtractOffPosition

Not yet mature

```
outFrame = subtractOffPosition(inFrame)
```

This task subtracts the background (off-position) for off-mapping AOTs.

2.7.5.12. specAvgPlateau

No longer part of the pipeline, so do not run it

```
outFrame = specAvgPlateau(inFrame [,sigclip=<number>][,mean=<number>]
           [,ignoreUncleanChopMask=<boolean>] [,ignoreGratMoveMask=<boolean>]
           [,qualityContext=<smthng>] [,copy=<number>])
```

Averages all valid signals on chopper plateaux and resamples signals, status, mask, stdev, wave, ra/dec words. Calculates the noise. The result is a *Frames* class with one image per every single chopper plateau. `mean` is an integer and if 1 use the median, if 0 (default) use the mean. `sigclip` is 0 (default) to not do sigma clipping, otherwise use the sigma value input and do clip; it should only be set for long chopper plateaux (> 3 readouts).

- It reads the values of CHOPPERPLATEAU and CALSOURCE columns in the status table. Any plateau is identified as a sample sequence of equal value of CHOPPERPLATEAU.
- The mask UNCLEANCHOP and GRATMOVE is used to identified the samples to discard in the estimation of the signal median (see cleanPlateauFrames/Ramps module for more details) due to distortions by the chopper and grating transition
- After the 'cleaning' procedure, the median of the signal of each pixel is estimated over the chopper plateau length. If the chopper plateau contains no valid data the signal is set to zero and the UNCLEANCHOPMASK to true.
- The noise is calculated by the following equation :

  noise[x,y,p] = STDDEV(signal[x,y,validSelection[p]]) / SQRT(nn)

  p : one plateau

  nn : Number of valid measurements
- The noise result is stored in the Frames as Noise entry
- The module adds the NrChopperPlateau column to the status table, which contains the number of valid samples averaged over the Plateau. Additionally, an UnCleanChop column is added which contains the number of discarded samples of each plateau.
- The Status entries with different values over the chopper plateau length are modified with the following scheme:
  - RESETINDEX, OBSID, BBID, LBL, FINETIME, CRDC, DBID, DMCSEQACTIVE, CHOPPERPLATEAU, CALSOURCE, SCANDIR, WASWITCH, BLOCKIDX, BAND, BBTYPE, BBSEQCNT, DP_WHICH_OBCP, GRATSCAN, CHOPPER: value of the beginning of the chopper plateau
  - TMP1, TMP2, VLD, PIX, RCX, RESETCNT: removed
  - CPR, WPR, GPR, CPCRMP, RRR, CRECR, WASWITCHPOS, CHOPFPUANGLE, CHOPSKYANGLE, XY_Stage_X/Y_AXIS: median
  - CHOPPOS: value of the beginning of the chopper plateau, if this is "NoName" first valid name is taken
  - OnRasterCount, OffRasterCount, DithPos : Median
  - others: median, strings: beginning of plateau
- For all available masks a layer is associated to each chopper plateau. For each pixel the value is set to True if the value is True in one or more frames over the chopper plateau length (except for UNCLEANCHOP and GRATMOVE, see above)

2.7.5.13. specDiffChop

```
outFrame = specDiffChop(inFrame [,removeCalStr=True>]
             [,normalize=False])
```

This task subtracts every off-source (chopped) signal from every consecutive on-source (chopped) signal, at the same grating position and in the same scan. The result has one image per one chopper cycle. The task first scans the status LBL and the GRATMOVE mask to determine where the grating plateaux are and what the chopper pattern within these plateaux is (a variant of ABAB or ABBA). Then it creates an array of all frame indices which need to be subtracted from each other, and the total number of frames in the result. Then for each result frame it computes/stores the following:

- The signal difference A-B, which may be normalised by setting the "normalize" option to true, when rather (A-B)/(2*(A+B)) is computed.
- Masks are merged with the OR operator. (Also, when one of the signals is masked then the mask is transferred to the resulting frame).
- The RA/Dec and WAVE of the ON position are stored in the result.
- The Noise is propagated from both A and B noise.
- The ON reset indices are stored in the RESETINDEX status column and the OFF reset indices in the OFF_RESETIDX column. In this way users can check which frames the algorithm has subtracted from which.
- The ON and OFF LBL values of the original Status tables are merged into a new Int2d column LBL2.
- All other status columns are taken from the ON position frames.

The following may be old:

The task follows the scheme:

- data with no DecMec running sequence are skipped (?)
- the values of the columns CALSOURCE are read in the Status table to identify the consecutive calibration blocks 1 and 2
- consecutive calibration blocks are subtracted
- a warning message appears if the calibration blocks have different lengths or if one of the calibration block is incomplete; in the latter case the calibration block is ignored
- the values of the columns CHOPPERPLATEAU are read in the Status table to identify the on#off images (consecutive chopper positions)
- for every couple of on#off images, the off-image is subtracted from the on-image; in case of an asymmetric chopper cycle (an odd number of chopped images) the last image is skipped
- for every pair of ON and OFF chopper positions the noise is computed :

  noise [x,y,k] = SQRT(noise[x,y,pON]**2 + noise[x,y,pOFF]*+2)

  k : ON#OFF pair

  pON : Plateau ON chop

  pOFF : Plateau OFF chop (in general pON + 1)

The Status entries with different values over the chopper plateau length are modified with the following scheme:

- RESETINDEX: counter
- OBSID, BBID, FINETIME, CPR, CRDC, CRDCCP, DBID, DMCSEQACTIVE, CALSOURCE, BAND, BBTYPE, BBSEQCNT: on-source value
- WPR: value of the beginning of the chopper plateau
- RESETCNT, BLOCKIDX, CHOPPERPLATEAU, RCX, PIX, BOLST, BSID, LBL, TMP1, TMP2, VLD: removed
- DithPos, OnRasterCounts, OffRasterCount, others : on-source value

2.7.5.14. rsrfCal

```
outFrame = rsrfCal(inFrame [,calTree=<mycalTree>] [,rsrfR1=<calfile>]
           [,rsrfB2B=<calfile>] [,rsrfB2A=<calfile>] [,rsrfB3A=<calfile>]
           [,normalise=<number>] [,copy=<number>])
```

This task corrects for the wavelength-dependent response of the system as mapped in the Relative Spectral Response Function. Per band, it reads the RSRF calibration file; normalised the RSRFs over the prime key wavelengths of the band; loops over all pixels and interpolates the normalised RSRF to the wavelengths sampled in those pixels; divides the signal by the interpolated response. `copy` is as for all other tasks and `normalise` is an integer, 0 to not normalise to the key wavelength, 1 (the default) to normalise.

Calibration files used: rsrfB2B, rsrfR1, rsrfB2A, rsrfB3A, one calfile per pixel, containing the wavelength/response. Either specify `calTree` or the particular calfiles to use.

2.7.5.15. specRespCal

```
outFrame = specRespCal(inFrame [,calTree=<mycalTree>] [,responseDrift=<product>]
           [,csResponseAndDark=<product>] [nominalResponse=<calfile>]
           [,copy=<number>])
```

This task divides by the best known and most recent responsivity values. The optional inputs `responseDrift` and `csResponseAndDark` are the products that were created by the tasks specFitSignalDrift and specDiffCs. If you did not (or could not) run those tasks, then this will still work but will take standard calfile values for its work, rather than those worked out from the dataset you are working on.

Calibration file used: nominalResponse, containing the nominal response values. Either specify this or the `calTree`.

2.7.5.16. specAddNod

```
outFrame = specAddNod(inFrame [,useWeightedMean=<integer>])
```

At present this task does not use masks, but that will change in the future.

This task combines the nod positions for a chop#nod observation. It adds every upscan on nod A to the subsequent upscan on nod B. It retains the pointing and chopper positions of nod A. It then does exactly the same for the downscans. By default it combines using a non-weighted mean, i.e. it combines the average of nod A and B of each nod cycle, separately, it does not average the grating up and downscan (i.e. these are retained as separate time-lines). If there is an error array present it is added to as the standard deviation of the mean. If you want to use the error-weighted mean and you have an error array to do that, then specify the parameter `useWeightedMean`, with value >0 (value=0, the default, corresponds to using the standard mean), at the same tme the error array is propagated accordingly.

Warning: before running this task you should check that the nod A and nod B for the opposite chops are actually really pointing at the same place on the sky (by overplotting the signal from a spaxel/pixel for e.g. chop-nodB and chop+nodA and seeing if they are the same). As of Nov 2009 we have rarely found that these pointings are exactly the same. Depending on what you want to get from the data, a slight offset may not be important, but if it is then you should not run this task; rather you will need to separate the data of nod B and A and treat them separately until you create the cube: how you then combine the nods is something we have still not fully established.

2.7.5.17. specFrames2PacsCube

```
cube = specFrames2PacsCube(inFrame)
```

This task converts a frame to a fully calibrated, oversampled 5x5xn PacsCube, which is the end of the Level 1 stage. The cube dimensions are 5x5xlambda, where within each spaxel all the spectra of the 16 modules that contribute to that spaxel. It does no manipulation of the spectra.

Warning: As of Nov 2009 you should consider these cubes to be of "browse", not astronimical quality as the spatial calibration (and hence pixel–spaxel combining) is still incorrect.

2.7.6. level 1 to level 2

The generation of Level 2 data products starting from Level 1 products will be dependent on the AOT. The AOTs for which the pipeline tasks will work are:

- Line spectroscopy
  - pointed: chop/nod or wavelength switching
  - pointed with dither: chop/nod or wavelength switching
  - mapping: chop/nod or wavelength switching
- Range spectroscopy
  - pointed: chop/nod
  - pointed with dither: chop/nod
  - mapping: chop/nod or off-position
- SED Mode
  - pointed: chop/nod
  - pointed with dither: chop/nod
  - mapping: chop/nod or off-position

This information will most certainly change during and after PV phase.

2.7.6.1. wavelengthGrid

```
grid = wavelengthGrid(pacsCube [,oversample=<number>]
       [,upsample=<number>] [,calTree=<mycalTree>])
```

This task calculates the wavelength bins for your dataset, which are dependent on the actual wavelengths present and the requested `oversampling` factor (the default value of which is 2.0; type: double, and can be sub-integer in value). `upsample` (type: double) is how much you shift forward by when creating the bins; the default value is 3.0 and it can take on values 1.0, 2.0, or 3.0. The grid created by this task is a product. Note: if you make `oversample` > `upsample` the resulting spectral binning will be horrible.

The oversample factor is used to increase the number of wavelength bins by the formula bins*oversample, where the number of bins is based on the theoretical resolution of your observation.

The upsample factor specifies how many shifts per wavelength bin to make while rebinning. Each bin is sampled "upsample" times, shifting forwards by 1/upsample. An upsample value of 2 means sample, shift by binwidth/2, and sample again. In this example, since both samples are the width in wavelength of the original wavelength bin, the second sample will overlap the next bin.

Note

up and oversampling: if you know that a the data series contains no frequencies higher than X, you know that you don't lose information if you (over)sample it more than every 1/(2*X) (Shannon or Nyquist sampling). You may want to sample it in smaller intervals, e.g. to verify that there are indeed no higher frequencies in the data. So in general, you sample the data every 1/(n*2*X), where n is the oversampling factor. Or, in other words, if you know the instrumental resolution (FWHM) is dl, you would rebin to wavelength intervals of width dl/n*2, where n is the oversampling factor. For practical purposes, rebinning (or grid making) routines take n*2 as a parameter (i.e. the width of the bin size as a fraction of FWHM rather than a fraction of half the FWHM).

A spectrum rebinned with an oversampling factor >2 has all the information that is in the unrebinned spectrum. But if you shift the bins by half a bin width, do the rebinning again, and plot the data together with the first rebin result, you get a spectrum where spectral profiles are more clearly recognised, making it easier to see the difference between e.g. a spectral line and a glitch. This is upsampling with a factor 2.

2.7.6.2. specFlagOutliers

```
cube = specFlagOutliers(cube, grid [,nSigma=<number>] [,nIter=<number>]
       [,ignoreMasks=<string>] [,saveStatus=<boolean>])
```

This task flags outliers in each wavelength bin and introduces the mask OUTLIERS. It should not mask data already masked (hence: see below). `nSigma` (default value 5.0) is the sigma value to flag at, `nIter` is how many repeats (iterations) of the outlier hunting you want to do (default value 1 but 2 would be a better first try value). `ignoreMasks` is a String1d of mask names that you want the task not to take in to account. The task gets the flux and wavelengths, for each spaxel, sorts the wavelengths, applies the masks, calculates the median and median absolute deviation of the flux in each wavelength bin, and clips outliers (+ and -) using that information. `saveStats` set to True (not the default) will save the median and deviation values calculated as *ArrayDatasets* attached to the cube.

The activating of masks recommended is

```
inFrame = activateMasks(inFrame, String1d(["GLITCH","UNCLEANCHOP",
   -"SATURATION","GRATMOVE", -"BADFITPIX"]), exclusive = True)
```

2.7.6.3. specWaveRebin

```
rebinnedCube = specWaveRebin(cube, grid)
```

This task constructs 5x5xlambda data cube which is the integral field view of the PACS spectrograph. It rebins the fluxes of the spectra held in each spaxel of the input cube, using the grid constructed by the wavelengthGrid task. The end result of this task is a cube of 5x5xlambda, where lambda now is of dimensions on your input grid, and in the course of the rebinning the 16 spectra that were originally stored in each spaxel have been merged into 1 spectrum per spaxel. By default any masks that are present are considered, except DEVIATINGOPENDUMMY and OBSWERR.

The activating of masks recommended is

```
inFrame = activateMasks(inFrame, String1d(["GLITCH","UNCLEANCHOP",
   -"SATURATION","GRATMOVE","BADFITPIX","OUTLIERS"]), exclusive = True)
```

2.7.6.4. specProject

```
projectedCube = specProject(rebinnedCube [,outputPixelSize=<number>]
                    [,use_mindist=<boolean>] [,norm_flux=<boolean>]
                    [,threshold=<number>] [,filter_nans=<boolean>]
                    [,debug=<boolean>] [interactive=<boolean>]
                    [qualityContext=<smthng>])
```

This task projects a rebinned cube (the output of SpecWaveRebin) onto a regular RA/Dec grid on the sky. The grid (the corners and dx,dy) will be determined by the task using the RA and Dec information in rebinnedCube. Input and output both are *SimpleCubes*. The parameters are: `outputPixelSize` is the output spaxel side in arcsec (default 3.0); `use_mindist` tells the task whether it should use the minimum spaxel distance rather than the average (default False); `norm_flux` (default True) tells the task whether it should divide by the exposure map to normalise fluxes; `threshold` (default 2.0) is used only if a *PacsCube* is input, rather than a *PacsRebinnedCube*, and is the minimum jump, in arcsec, which triggers a new raster position; `filter_nans` (default False) if True all frames with one or more NaN values will be discarded; `debug` (default False) set to True will create extra datasets in the output product for debugging purposes; `interactive` (default False) set to true will produce several plots while running; `qualityContext` (default None) is only used in SPG mode.

The task:

1. scans all the RA/Dec values in the input cube and selects (all) the unique scan position(s). Store for each scan position the frame numbers which match these positions, the RA and Dec and rotation matrix of the spaxels (method: selectUniquePositions).

2. computes a regular RA/Dec grid which encompasses all the raster positions from the previous step (method: computeGrid).

3. loops over all raster positions and do for each position the following: i) compute the weights for projecting the input spaxels to the output grid. These weights determine which input spaxel(s) the output spaxel(s) overlap and by how much. The results are stored in two 3D arrays, one containing the overlapping modules for each output RA/Dec, and one with their corresponding weights. ii) compute for each frame at each position in the cube the output fluxes on the new regular grid. This is done by adding up for each spaxel the fluxes of the contributing spaxels multiplied by their overlap weights.

4. Combine the projected images from different raster positions and normalise by dividing with the sum of the weights of all positions.

5. Write the resulting projection to the output cube.

This task is worth running even if you only have one pointing in your observation because it does not just add together, or mosaic, multiple pointings, but also sets the correct spatial grid for each wavelength of your cube. For the PACS spectrograph, each wavelength sees a slightly different spatial position, even for spectra within a single spaxel.

2.7.6.5. 3dDrizzling

Still to come

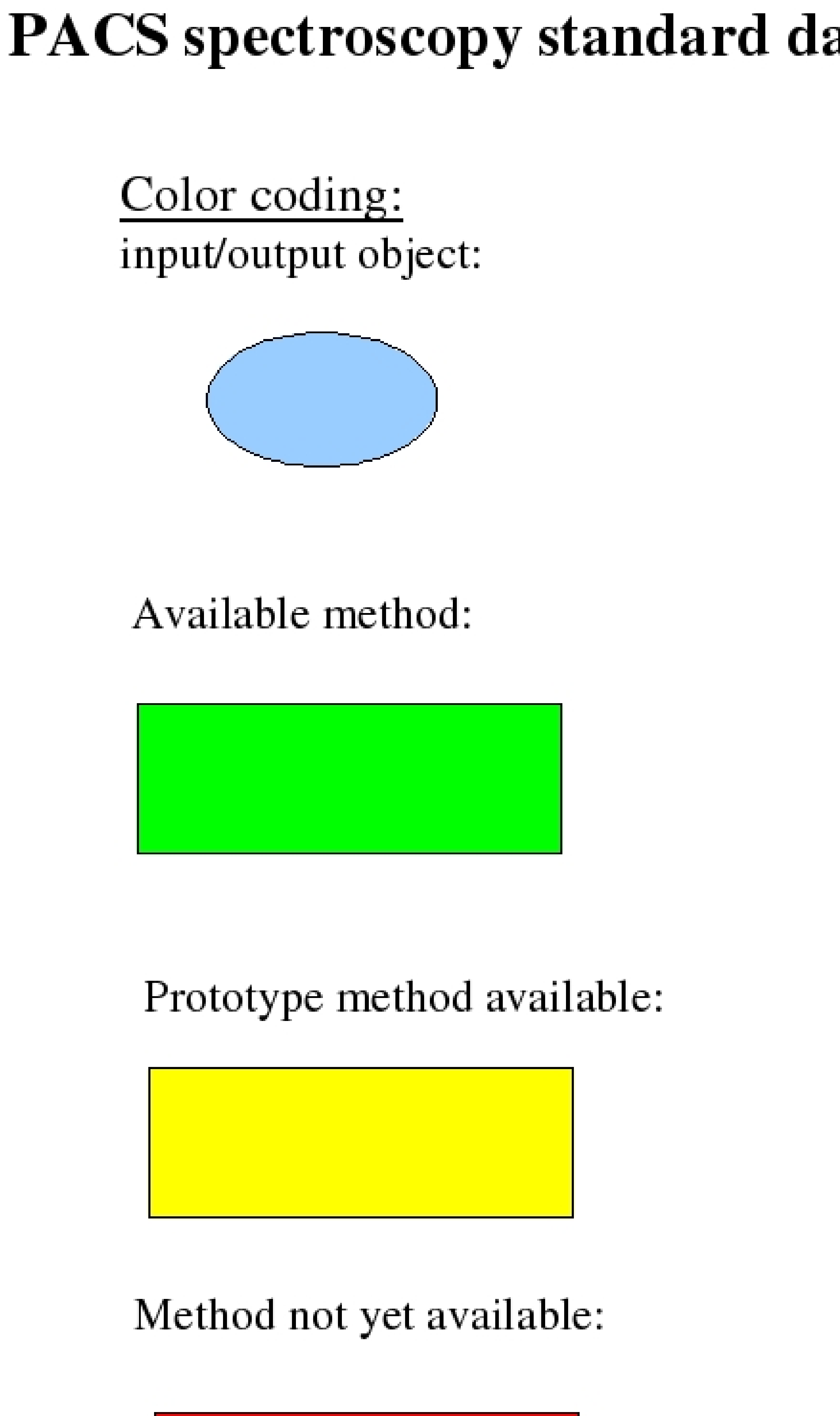
```
drizzledCube = 3dDrizzling(cube)
```

More to come.

2.7.7. SPG Pipeline chart

Note: these are updated less frequently than the text

2.7.7.1. color coding



PACS spectroscopy standard da

Color coding:

input/output object:

🔵

Available method:

🟩

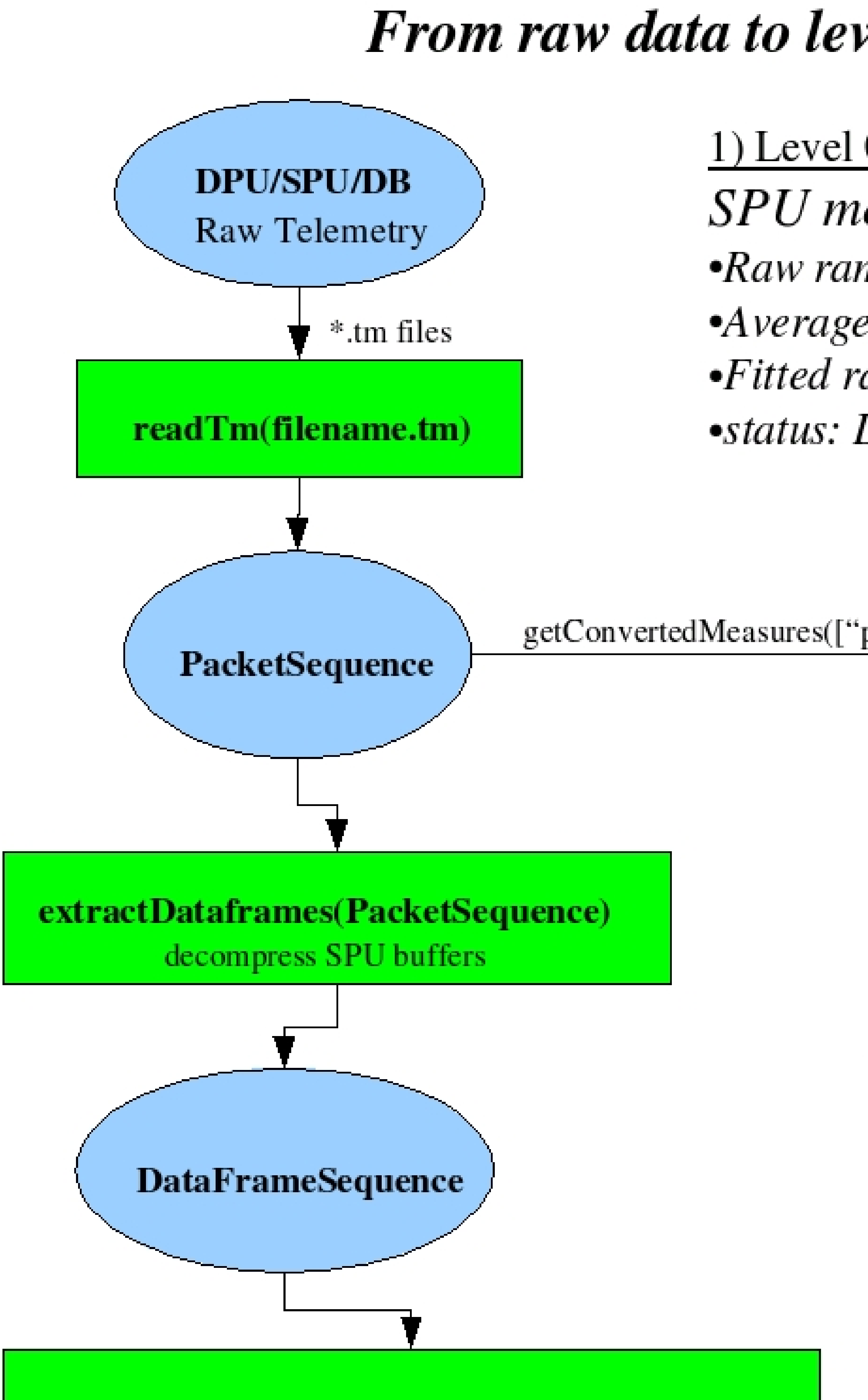
Prototype method available:

🟨

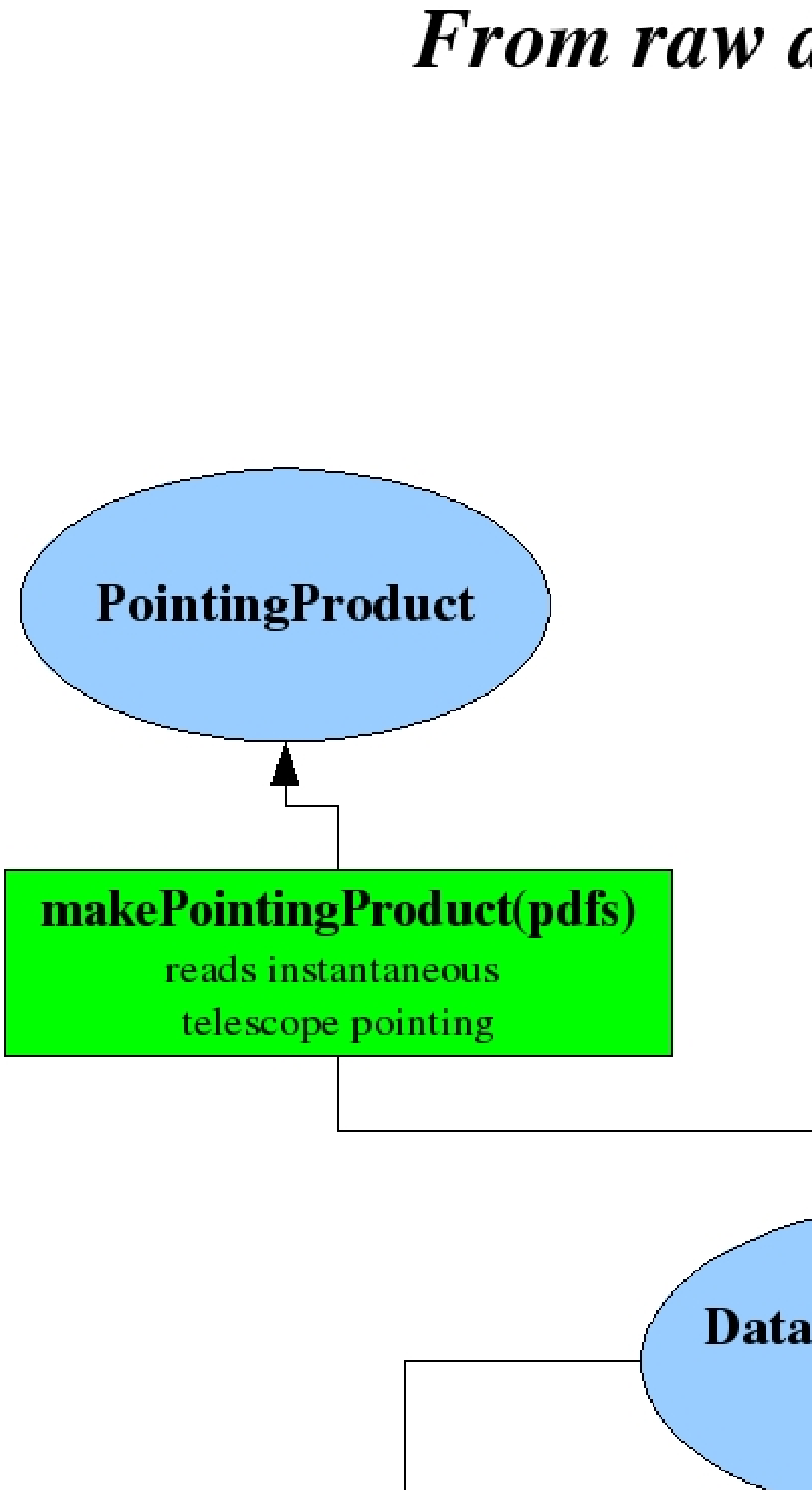
Method not yet available:

🟥

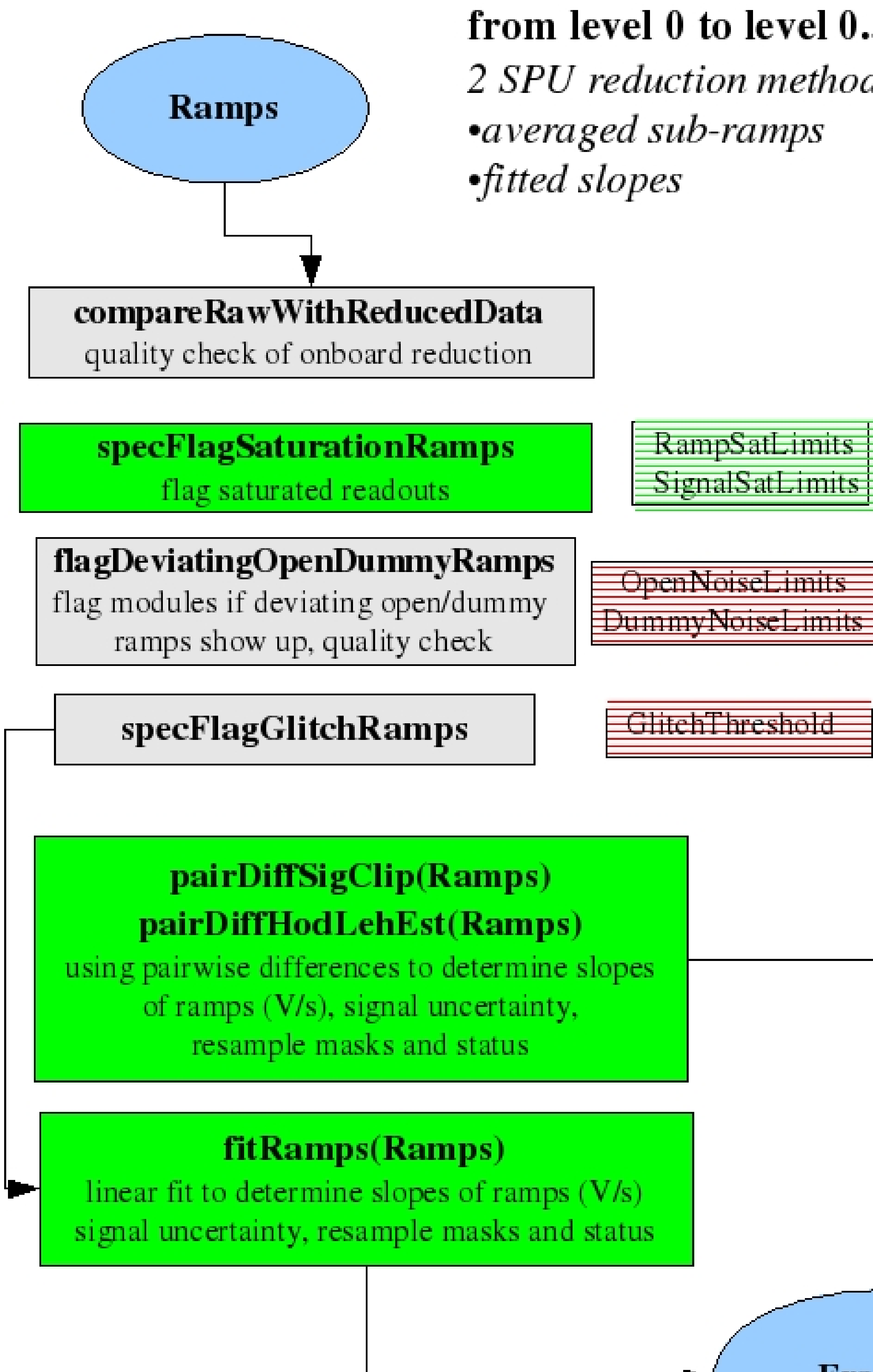
2.7.7.2. from raw telemetry to level 0



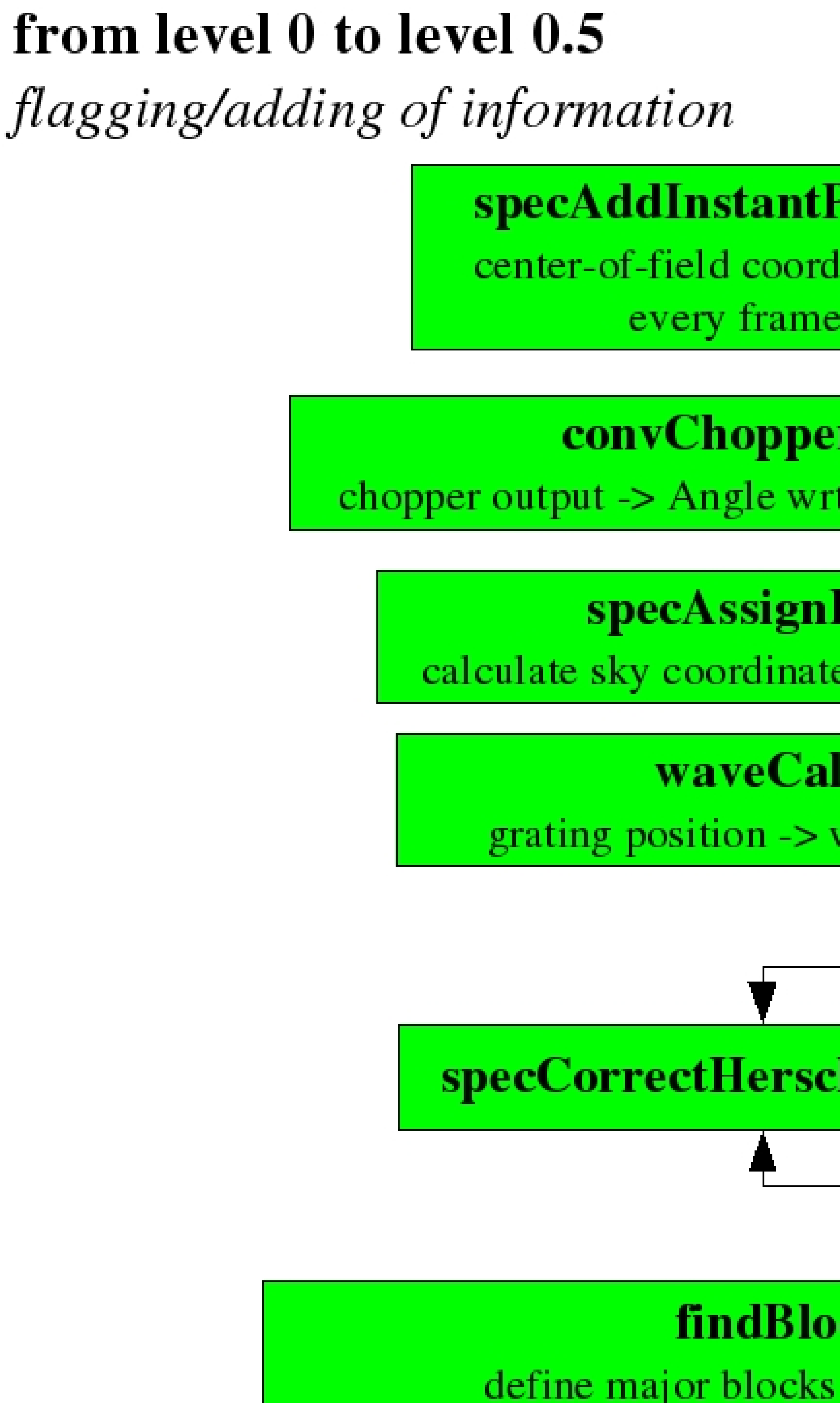
2.7.7.3. from raw telemetry to level 0



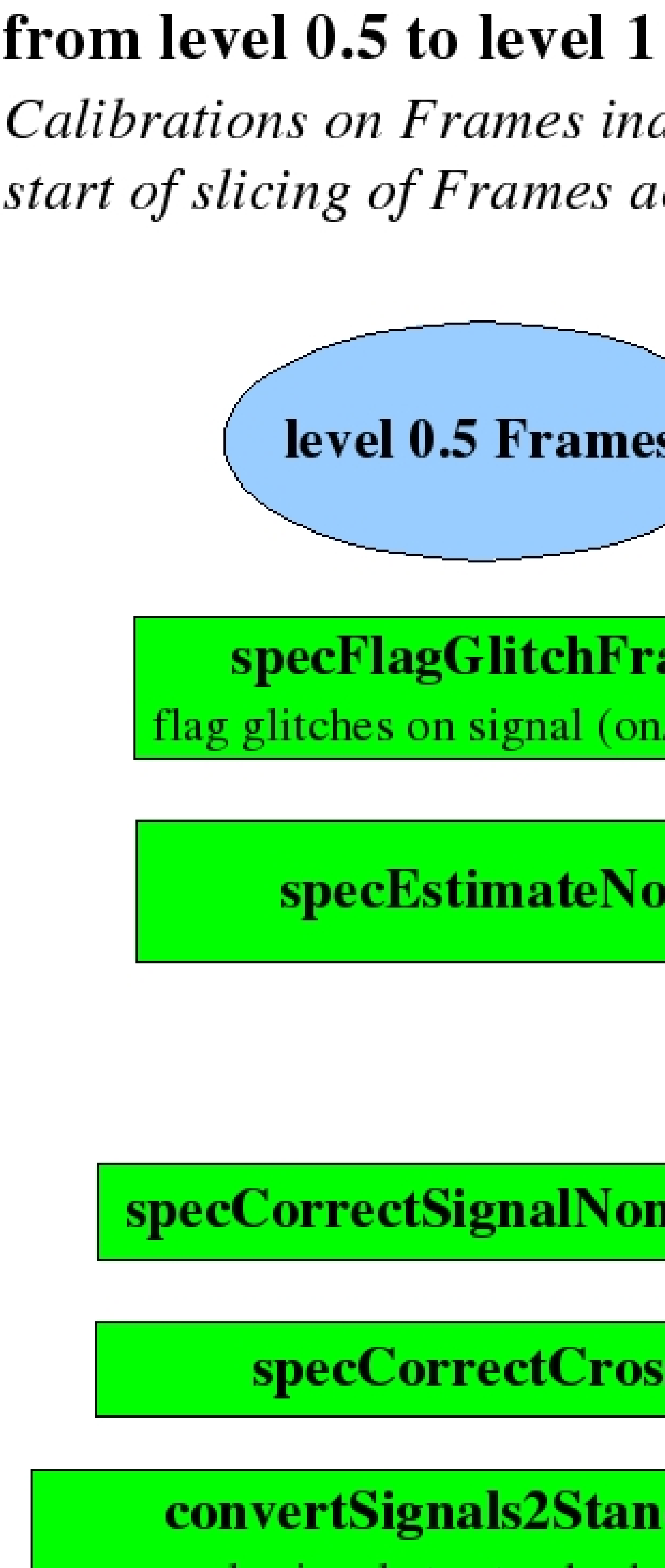
2.7.7.4. from level 0 to level 0.5


from level 0 to level 0.
2 SPU reduction methoc
•averaged sub-ramps
•fitted slopes
Ramps
compareRawWithReducedData
quality check of onboard reduction
specFlagSaturationRamps
flag saturated readouts
RampSatLimits
SignalSatLimits
flagDeviatingOpenDummyRamps
flag modules if deviating open/dummy ramps show up, quality check
OpenNoiseLimits
DummyNoiseLimits
specFlagGlitchRamps
GlitchThreshold
pairDiffSigClip(Ramps)
pairDiffHodLehEst(Ramps)
using pairwise differences to determine slopes of ramps (V/s), signal uncertainty, resample masks and status
fitRamps(Ramps)
linear fit to determine slopes of ramps (V/s) signal uncertainty, resample masks and status

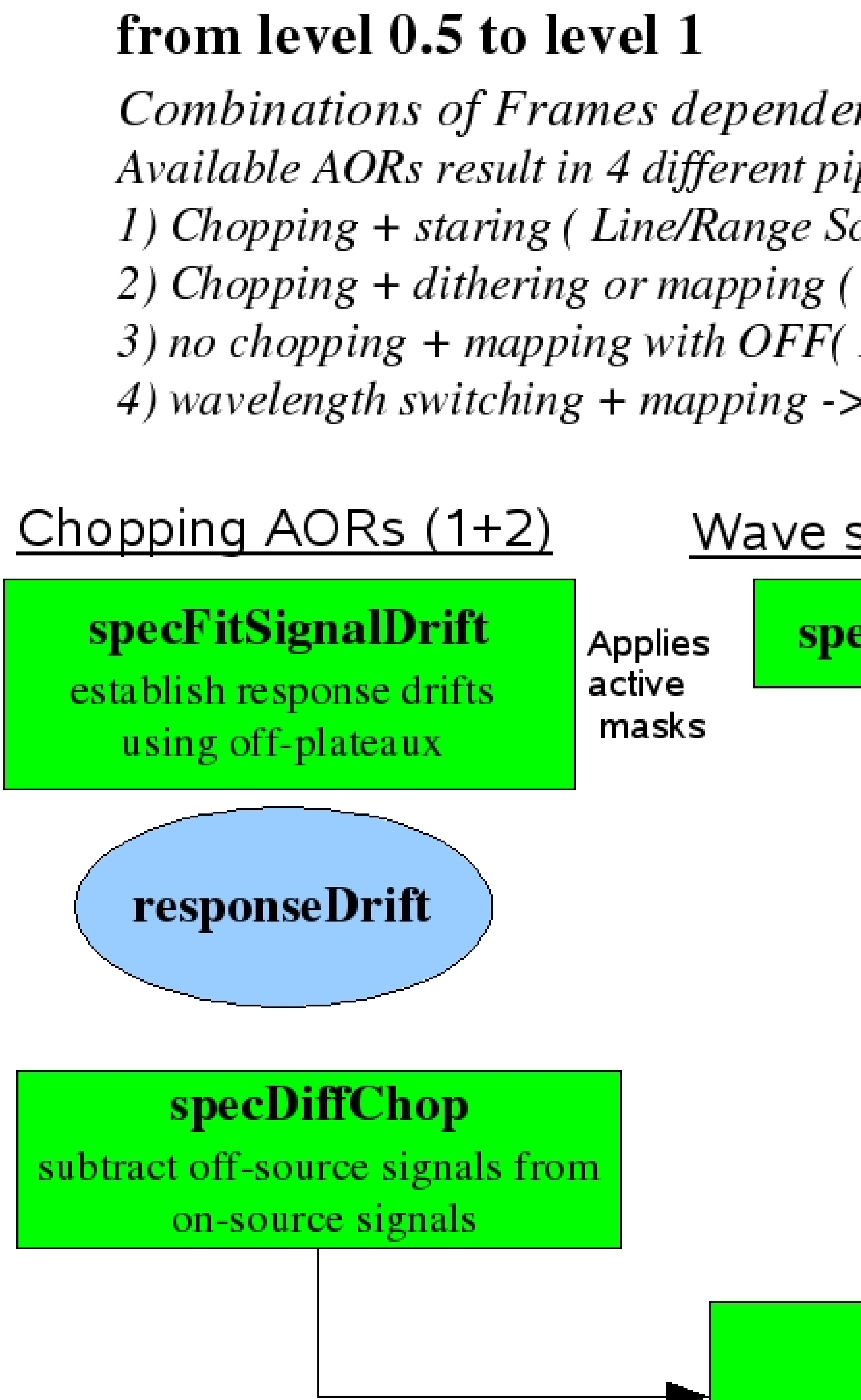
2.7.7.5. from level 0 to level 0.5



2.7.7.6. from level 0.5 to level 1


from level 0.5 to level 1
Calibrations on Frames inc
start of slicing of Frames a
level 0.5 Frames
specFlagGlitchFra
flag glitches on signal (on
specEstimateNo
specCorrectSignalNon
specCorrectCros
convertSignals2Stan

2.7.7.7. from level 0.5 to level 1

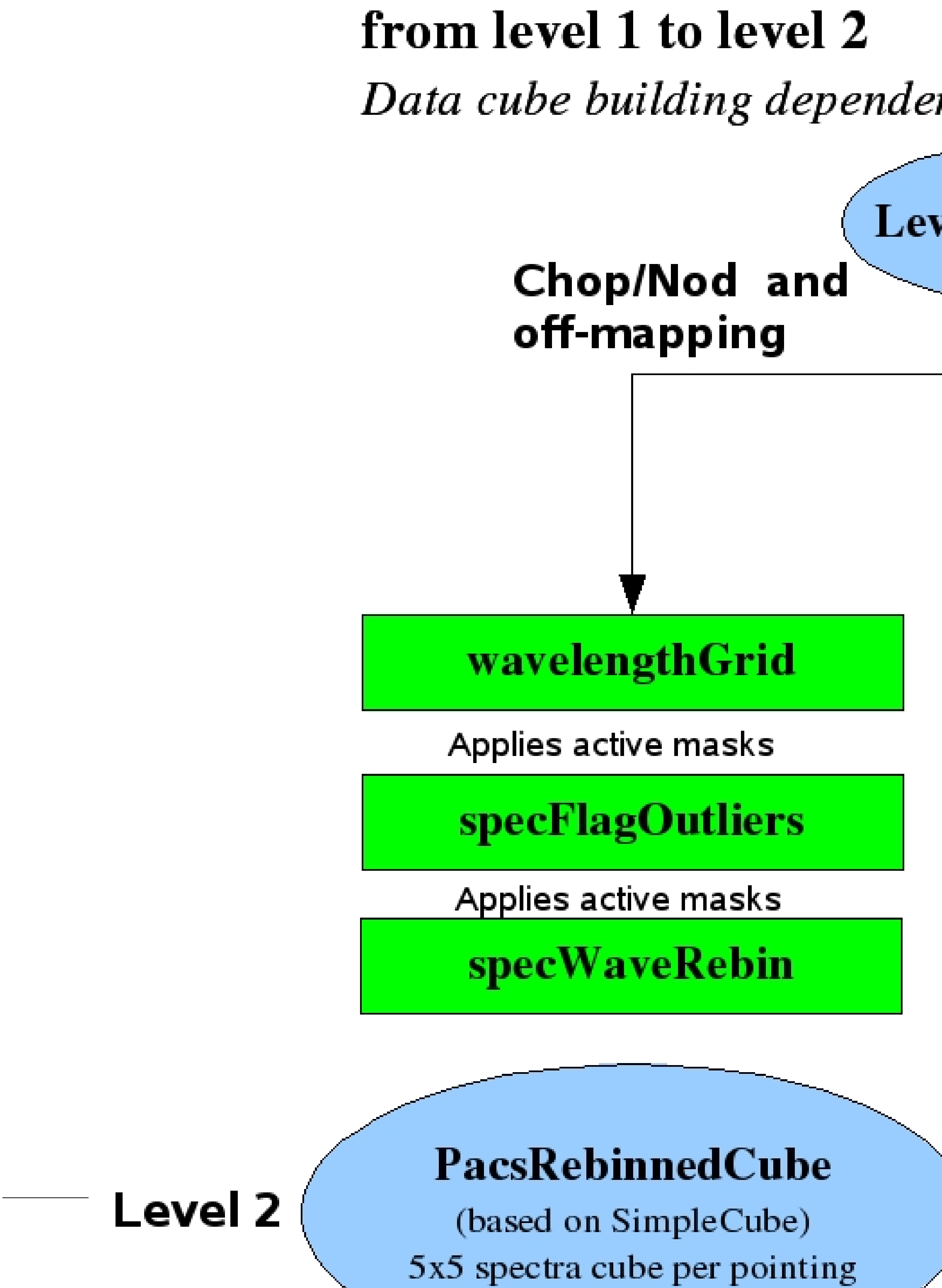


from level 0.5 to level 1

Combinations of Frames depende
Available AORs result in 4 different pi
1) Chopping + staring (Line/Range Sc
2) Chopping + dithering or mapping (
3) no chopping + mapping with OFF(
4) wavelength switching + mapping ->

Chopping AORs (1+2)
Wave s
specFitSignalDrift
establish response drifts using off-plateaux
Applies active masks
spe
responseDrift
specDiffChop
subtract off-source signals from on-source signals

2.7.7.8. from level 1 to level 2



2.7.8. Appendix: Spectrometer Flux Calibration Concept

In the sections below we outline the spg steps necessary to get to absolutel flux densities. We briefly outline the overal flux calibration concept for PACS spectroscopic observations. *This information will probably change afer PV phase.*

A PACS spectroscopic observation typically consists of the following sequence:

- Internal cal source measurement: short grating scan around a key wavelength in the band observed. At every grating position the chopper is moved to the two calibration sources.
- At Nod position A: grating scan up (increasing wavelengths) over the desired wavelength range. At every grating position the chopper moves between the source (S) and the background (B).
- At Nod position A: grating scan down (decreasing wavelengths) over the desired wavelength range. At every grating position the chopper moves between the source (S) and the background (B).
- At Nod position B: (nod position is choosen such that the source is in the previous background position in the field of view). grating scan up (increasing wavelengths) over the desired wavelength range. At every grating position the chopper moves between the source (S) and the background (B).
- At Nod position B: grating scan down (decreasing wavelengths) over the desired wavelength range. At every grating position the chopper moves between the source (S) and the background (B).

The signal we measure during the different sky grating scans is the following:

- Nod A / chop 1 : A1 = R [T1 + S + B]
- Nod A / chop 2 : A2 = R [T2 + B]
- Nod B / chop 1 : B1 = R [T1 + B]
- Nod B / chop 2 : B2 = R [T2 + S + B]

with R the system response at the observed wavelength, T1 the flux from the telescope thermal background at chop position 1, T2 the flux from the telescope thermal background at chop position 2, S the flux from the source, and B the flux from the sky background.

In order to get the flux from the source from A1, A2, B1 and B2, we calculate:

[A1 - A2] - [B1 - B2] = R [T1 + S + B - T2 - B - T1 - B + T2 + S + B] = 2 R S

The SPG steps therefore differentiate the scans at the on/off chopper position in the two nod positions, and then differentiate between the two nod positions.

Obviously, only signals sampled at the same wavelength can be differentiated. Per grating position there are typically a few signals on source and a few signals off source. There are two extreme approaches to subtracting these.

The extreme reduction approach would be to average the signals on source, average the signals off source and subtract the two averages. If we have 4 samples per grating position, this would reduce the size of the Frames dataset by a factor 8 after the chopper plateau subtraction, and a factor 16 after the nod subtraction. A noise filter could be applied before averageing, but the redundancy in the data is low at this point, so the filter needs to be conservative.

The other extreme is to subtract every off-source signal from every on-source signal. In the case of 4 samples per grating position, this increases the size of the Frames dataset by a factor of 2 after the chopper plateau subtraction, and a factor of 16 after the nod subtraction. The advantage of this approach is that the noise filter / rebinning can be applied at a later stage, when combining data from several scans/observations, over wavelength bins that are tuned to the instrument resolution.

The pipeline steps allow to choose between these two extreme approaches, and different flavours in-between (e.g. average the signals of the off positions only). For practical reasons, we therefore calculate [A1 - A2] + [B2 -B1], i.e. we always subtract off source from on-source and co-add the two nod positions.

The response R is disentangled in a relative wavelength-dependent component which is stable and an absolute flux scaling to correct for detector drifts. The wavelength dependent response is determined per detector in the Relative Spectral Response Function (RSRF). As to get the absolute scale of the response, the differential signal of the internal cal source measurement at a key wavelength can be related to the same differential signal in calibration observations when celestial calibrators are observed. The differential calibration source signal measured in the observation is looked up in a calibration table and gives the conversion from Volts/Sec to Jansky.

Chapter 3. PACS Photometry standard data processing

3.1. Introduction

This chapter describes the standard processing steps for the different photometry observation modes of the PACS instrument. For every step it gives a rough algorithm (optimizations of complexity are beyond the scope of this document) and calibration tables that are needed as input. The different intermediate conceptual formats of the PACS photometry data throughout the reduction are described as well.

3.2. Definition of terms

3.3. Summary of the Photometry processing steps

We summarize here the basic steps of the PACS photometry data reduction. The aim of this chapter is to explain the user how to reduce the PACS photometry data starting by different "Level Product". We assume here that the user is familiar with the concept of the "ObservationContext". So we assume that the user will start the data processing by accessing different levels of data Products in her/his local store. Under these assumption the basic steps of the data processing starting from Level 0 Products are the following:

1. access the local store and retrieve the Frames of a given observation and the related pointing product
2. identify the structure of the observation and identify the main block (Calibration and Science blocks)
3. pre-process the calibration block and extract useful information for the further calibrations
4. perform data cosmetics: flag bad/saturated pixels and flag/correct cross talk and glitches
5. convert signal from digits to volts
6. covert chopper position from engineering units into angle
7. satellite pointing info are added to frames (sky coordinates of reference pixel for each readout)
8. the astrometry is calculated on the basis of spatial calibration files (spatial distortions are taken into account)
9. in case of chopped observation the chop-nod cycle is reduced to remove sky and telescope background
10. the flat field and flux calibration are applied and corrected for possible drifts
11. The spacecraft on-board time is converted to UTC
12. in case of scan map observation, the signal is filtered to remove 1/f noise
13. A stack/mosaic of frames is constructed

3.4. Processing levels

There is a Herschel-wide convention on processing levels of the different instruments. Here we list the content and the properties of the different Product Level for the PACS Photometry mode.

- *Raw Telemetry :* This is the format of the raw PACS photometry data. The telemetry file is composed of telemetry packets produced by the instrument in the course of the observation. These data are pre-processed and compressed on board of Herschel. For pre-processing we mean a simple averaging any 4 readouts for a final sampling of 10 Hz. This data product will not be visible in the pipeline processing and it will not be delivered to the end user.
- *Decompressed Science Data :*

  This is an "artificial level". The data are not stored and not visible for general user. But in the interactive step by step data analysis the data product can be analyzed for debugging purposes.

  Telemetry data as measured by the instrument, minimally manipulated and stored as Data Frames. For PACS photometry, this level is stored/manipulated in a *DataFrameSequence* : a sequence of PACS dataframes, which are decompressed SPU buffers.

  What is contained in every decompressed SPU buffer depends on the SPU reduction mode. Typically there are several reduced readouts for every active detector (averaged detector signals), 40Hz or 20Hz readouts for a few selected pixels and mechanism/status information sampled at 40Hz/20Hz by the DecMec, the so-called DMC Header.
- *Level 0 data:*

  Level 0 data is a complete set of data requested to do the scientific data reduction. It is saved in a Level 0 Data Pool in form of Fits files. After Level 0 data generation no connection to the Database is possible any more. Therefore Level 0 data need to contain all information needed from the Database (e.g. uplink information).

  - Science data

    Science data are organized in user friendly classes. The `Frames` class for reduced data and the `PhotRaw` class for additional raw channel data will be the basic data products for this processing steps. The so-called Status table of the Frames class stores the info carried by the DMC header which are necessary for the data reduction (chopper position, identification of internal calibration observation and scientific observations).
  - Auxiliary data

    Auxiliary data for the time span covered by the Level 0 data, such as the spacecraft pointing (attitude history), the time correlation, selected spacecraft housekeeping, etc

    The information is partly merged as status entries into the basic science classes Frames and PhotRaw or available as Products (Pointing)
  - Decoded HK Data

    HK data Tables with converted and raw HK values.
  - Calibration files and data of 'associated' observations - e.g. photometric checks or other Trend Analysis results taken throughout the operational day or even before (still to be clarified!).
- *Level 0.5 data :*

  Processing until Level 0.5 is AOT independent These data are saved in the Product Pool.

  On this Level additional information is added to the Frames class (Flags for Saturation, Flags Bad Pixel, BlockTable,...) and basic unit conversion are done (digital values to Volts, chopper angle).

- *Level 1 data:*

  Level 1 data generation is AOT dependent. Level 1 data are saved in the Product Pool.

  Detector readouts calibrated, converted to physical units and grouped into blocks. For PACS photometry this is a data cube with flux densities with associated sky coordinates. Mostly every step before actual Image construction is done.

  The `Frames or FramesStack` class will be the basic Level 1 product of photometer data

  Possibly the Level 1 data generation can be done automatically to a large extend after the instrument has been calibrated.

- *Level 2 data:*

  Further processed level-1 data to such a level that scientific analysis can be performed.

  The noise is filtered and the map is reconstructed with different methods/algorithms depending on the AOT mode.

  For optimal results many of the processing steps involved to generate level-2 data may require human interaction, based both on instrument understanding as well as understanding of the scientific aims of the observation.

  The result is an Image Product.

- *Level 3 data:*

  These are the publishable science products where level-2 data products are used as input. These products are not only from the specific instrument, but are usually combined with theoretical models, other observations, laboratory data, cataloguers, etc. Their formats should be VO compatible and these data products should be suitable for VO access.

3.5. Imports

To be able to execute the commands in this document, you need to import the necessary java classes and jython toolboxes:

```
>> from all import *
```

3.6. Used Masks

The following Masks are used by default during Pipeline processing, additional masks can be added by the user when necessary:

```
BLINDPIXEL  -: Pixel masked out by the DetectorSelectionTable (already in Level 0)
BADPIXEL    -: Bad Pixel masked during pipeline processing
SATURATION  -: Saturated Readouts
GLITCH      -: Glitched Readouts
UNCLEANCHOP -: Flagging unreliable signals at the begin and end of a ChopperPlateau
```

All the masks created by the pipeline are 3D masks. This is true even for the cases when it is not necessary such as in the BLINDPIXEL, BADPIXEL and UNCLEANCHOP masks. Moreover all the masks are boolean: unmasked pixels are saved as FALSE and masked pixels are saved as TRUE. Be careful that the DatasetInspector in jide and the Editor in hipe are not able to properly transform a boolean mask into integer mask. Due to this bug, the table shown by the jide DatasetInspector and the hipe Editor for each mask has wrong dimension. Only the MaskViewer is able to properly display the masks (see section 10.6.6.2 for details about the use of the MaskViewer).

3.7. Level 0 to Level 0.5

We assume that the reader is starting the data reduction from Level 0 Products. Tasks and procedures related to creation of pools from telemetry files or from database need a deeper knowledge of the system and are included at the end of this chapter. The PACS Photometer pipeline is composed of tasks written in java and jython. In this section we explain the individual steps of the pipeline up to Level 0.5. Up to this product level the data reduction is AOT independent. The only AOT dependent task executed in this part of the data reduction is the CleanPlateau task, which is executed only for chopped observations (Point-Source, Small source and Chopped Raster AOT).

3.7.1. Getting started: how to retrieve data in the Observation Context

We assume that the reader got a tar file containing all the chosen observations and associated data from the HSA. Unpacking this tar file should automatically create a 'so called' local store with one or more pools. Any pool contains a number of directories containing data products of different level (Level 0,1,2) and calibration files for each observation. A special pool contains the Auxiliary products with, in particular, the Pointing and the SIAM products which are needed for the astrometric calibration. We list here few commands that need to be executed to retrieve a given observation from a pool and start the data reduction:

```
lstore = LocalStoreFactory.getStore("test_pool")
store  = ProductStorage()
store.register(lstore)
result = browseProduct(store)
#in alternative:
query=MetaQuery(ObservationContext,'h','h.meta["obsid"].value == 3221226006l')
result=store.select(query)
```

The first three commands listed in the window above access and register a test pool ("test_pool"). The fourth command calls the Product Browser to inspect the content of the Observation Context and choose a given observation. We refer the reader to Chapter 12.1.10 for a detailed description of the Product Browser and its use. The observation chosen in the Product Browser is, then, stored in the variable "result". In alternative, if the content of the pool is already known, we can query a particular observation on the basis of its OBSID, which is a unique identification number. The result of the query is, then, stored in the result variable as done in the case of the Product Browser.

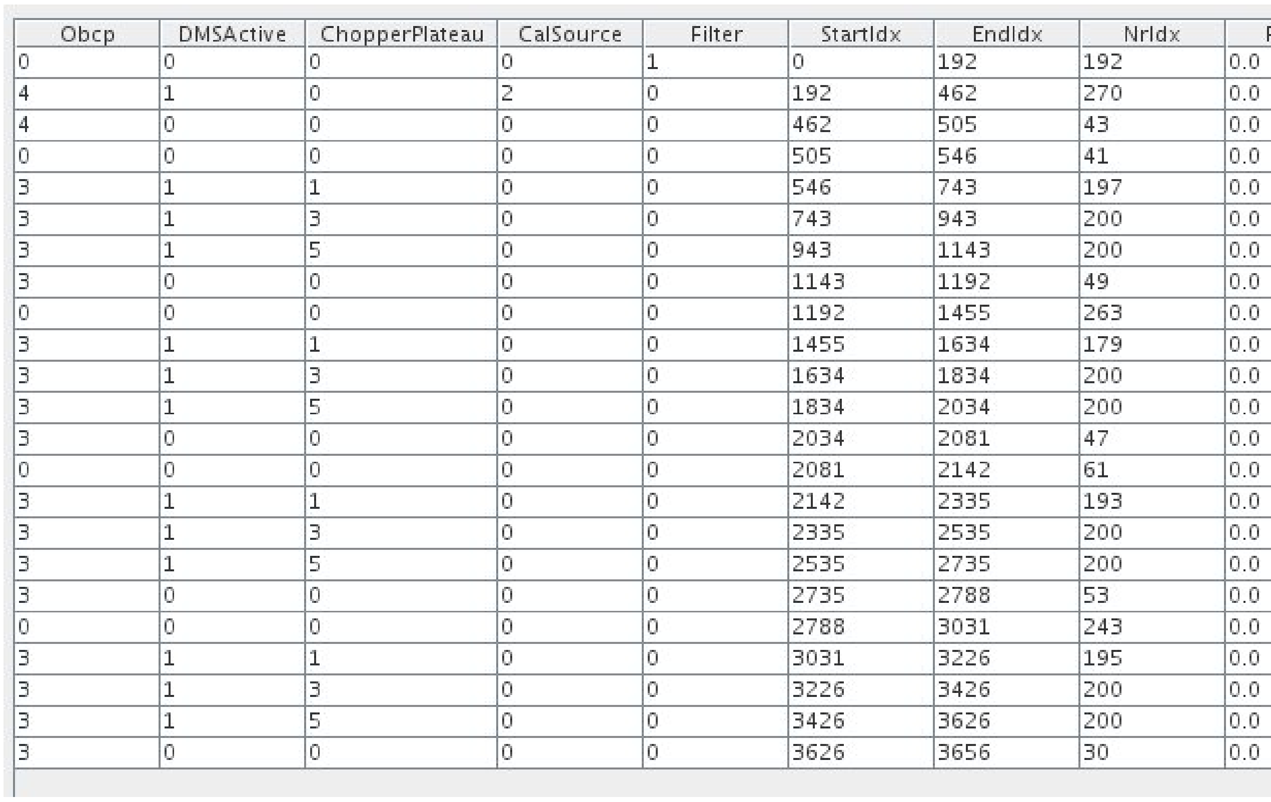
```
obs=result[0].product
frames=obs.level0.refs["HPPAVGB"].product.refs[0].product
hkdata = level0.refs["HPPHK"].product.refs[0].product["HPPHKS"]
pp=obs.auxiliary.pointing
calTree = getCalTree("FM", -"BASE")
```

After selecting our favorite observation, we can store a given product (Level 0, 1 or 2 if they exist in the considered pool) in the "obs" variable, as shown in the first command in the window above. In our example we select all the information relative to the Level 0 product to start with the first step of the data processing. The second command selects the frames of the Blue bolometer and store them in the variable "frames". The string "HPPAVGB" needs to be changed to "HPPAVGR" to select products of the Red bolometer. The frames class is composed of a data cube containing all the readouts of our observation, the so-called Status table with several entries for each readout, the BLIND-PIXEL mask and several metadata. With a similar syntax we store in the "hk" variable several House Keeping values of our observation. This variable will be used directly in the next step of the data reduction. This info is needed for the further data calibration. The last two commands store the Pointing Product in the variable "pp" and the Calibration Tree in the calTree variable.

3.7.2. The second step, understanding what there is in the observation: findBlocks (jython prototype available)

```
>> outFrames = findBlocks(inFrames)

outFrames -: Frames -: Frames out
inFrames  -: Frames -: Frames in
```

This task is not essential for the data reduction. We can reduce the data even without executing the task findblock. However, we suggest to execute this task just after getting the data to understand what there is in the dataset. The result of this task is a table, called BlockTable, containing info about the structure of the observation: how many calibration blocks were executed during the observation, how many chop-nod cycles or scan legs are contained in the data, etc. Basically, the BlockTable summarizes per observation block several info already contained, per readout, in the Status Table. This info can help us in checking if the data contains the observation as we have designed it, to select just part of the observation for a preliminary data reduction, to slice the data as we desire, etc.



Obcp	DMSActive	ChopperPlateau	CalSource	Filter	StartIdx	EndIdx	NrIdx	F
0	0	0	0	1	0	192	192	0.0
4	1	0	2	0	192	462	270	0.0
4	0	0	0	0	462	505	43	0.0
0	0	0	0	0	505	546	41	0.0
3	1	1	0	0	546	743	197	0.0
3	1	3	0	0	743	943	200	0.0
3	1	5	0	0	943	1143	200	0.0
3	0	0	0	0	1143	1192	49	0.0
0	0	0	0	0	1192	1455	263	0.0
3	1	1	0	0	1455	1634	179	0.0
3	1	3	0	0	1634	1834	200	0.0
3	1	5	0	0	1834	2034	200	0.0
3	0	0	0	0	2034	2081	47	0.0
0	0	0	0	0	2081	2142	61	0.0
3	1	1	0	0	2142	2335	193	0.0
3	1	3	0	0	2335	2535	200	0.0
3	1	5	0	0	2535	2735	200	0.0
3	0	0	0	0	2735	2788	53	0.0
0	0	0	0	0	2788	3031	243	0.0
3	1	1	0	0	3031	3226	195	0.0
3	1	3	0	0	3226	3426	200	0.0
3	1	5	0	0	3426	3626	200	0.0
3	0	0	0	0	3626	3656	30	0.0

Figure 3.1.

The figure above shows a BlockTable example for a chopped observation, in particular a point-source AOT. The main ingredient of the BlockTable is the OBCP/DMC number (first column in the example above). This number identifies what PACS photometer is doing for a given time, that is between the time indexes StartIdx and EndIdx of our observation. A verbal translation of the OBCP/DMC number is given in the Id and Description columns. For instance, we can easily see in which part the observation the calibration block was executed (id="PHOT_CHOP_CS"), or when the PACS photometer is preparing itself fo the next command (Id="OBCP and DMC preparation" or Id="Undefined") and when the real scientific data are taken and the first chopper sequence is executed (Id="PHOT_CHOP_TRG_1"). Satellite pointing information/mode (staring, nod-position A or B, raster point M-N, scan leg number, tracking, hold position, etc.) can also be included in the BlockTable if findBlockd is executed after the execution of the AddInstantPointing task which adds the pointing information to the frames class.

Typical AOT observations might contain several OBCPs and some of the OBCPs might be executed many times within one AOT observation (as in the example above). A block contained in a given OBCP/DMC sequence might correspond to different labels. In some cases a change in label does not necessarily mean a change in chopper position, e.g. see below, block 2 in OBCP 04 "chopper photometry sequence on target" can have labels 1,3, and 5. Here, labels 3 and 5 correspond very likely to the same chopper position.

We list here the OBCP/DMC sequences and the blocks associated to them (taken from PACS-ME-LI-005, Issue 1.1, 08-Mar-2005) :

- OBCP 01: Bolometer transition to IDLE state ---> no DMC sequence, no blocks
- OBCP 02: Bolometer operation for unregulated state ---> no DMC sequence, no blocks
- OBCP 03: Fixed-Fixed Chopped Photometry ---> DMC sequence 14 blocks:
  - OBCP and DMC preparation (label undefined, 0)
  - first chopper sequence on target (labels 3,5)
  - second chopper sequence on target (labels 7,9)
  - third chopper sequence on target (labels 11,13)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 04: Chopped Photometry ---> DMC sequence 1 blocks:
  - OBCP and DMC preparation (label undefined, 0)
  - chopper sequence on target (labels 1,3,5)
  - chopper sequence on CSs (labels 65,129)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 05: Chopped Photometry with Dither ---> DMC sequence 2 blocks:
  - OBCP and DMC preparation (label undefined, 0)
  - chopper sequence on target (labels 1,3,5)
  - chopper sequence on CSs (labels 65,129)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 06: Freeze Frame Chopping Photometry ---> DMC sequence 4 blocks:
  - OBCP and DMC preparation (label undefined, 0)
  - freeze frame sequence on target (label 63)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 07: Staring Photometry for Line Scans ---> DMC sequence 3 blocks:
  - OBCP and DMC preparation (label undefined, 0)
  - staring sequence on target (label 1)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 08: Grating Spectral Line Scan Chopped ---> DMC sequence 8 blocks:

  - OBCP and DMC preparation (label undefined, 0)
  - chopped up-scan sequence on target (labels 3,5,7)
  - chopped up-scan sequence on CSs (labels 65,129)
  - chopped down-scan sequence on target (labels 19,21,23)
  - chopped down-scan sequence on CSs (labels 81,145)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 09: Grating Spectral Line Scan Chopped with Dither ---> DMC sequence 9 blocks:
  - OBCP and DMC preparation (label undefined, 0)
  - chopped up-scan sequence on target (labels 3,5,7)
  - chopped up-scan sequence on CSs (labels 65,129)
  - chopped down-scan sequence on target (labels 19,21,23)
  - chopped down-scan sequence on CSs (labels 81,145)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 10: Photometry Calibration I ---> DMC sequence 5 blocks:
  - OBCP and DMC preparation (label undefined, 0)
  - variable-variable chopped sequence on CSs (labels 65, 129)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 11: Photometry Calibration II ---> DMC sequence 6 blocks:
  - OBCP and DMC preparation (label undefined, 0)
  - fixed-variable chopped sequence on CSs (labels 65, 129)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 12: Photometry Calibration III ---> DMC sequence 7 blocks:
  - OBCP and DMC preparation (label undefined, 0)
  - fixed-fixed chopped sequence on CSs (labels 65, 129)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 13: Internal Calibration Spectroscopy ---> DMC sequence 11 blocks:
  - OBCP and DMC preparation (label undefined, 0)
  - chopped up-scan sequence on CSs (labels 65,129)
  - chopped down-scan sequence on CSs (labels 81,145)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 14: Acquire Non-Sequencer Science Data ---> no DMC sequence, no blocks
- OBCP 15: DMC Test Mode ---> no DMC sequence, no blocks

- OBCP 16: Switch Spectroscopy to Photometry ---> no DMC sequence, no blocks
- OBCP 17: Switch Photometry to Spectroscopy ---> no DMC sequence, no blocks
- OBCP 18: Prepare for Switch-off ---> no DMC sequence, no blocks
- OBCP 19: Start 1355 link ---> no DMC sequence, no blocks
- OBCP 20: Write in EEPROM ---> no DMC sequence, no blocks
- OBCP 21: Start HLSW ---> no DMC sequence, no blocks
- OBCP 22: Wavelength Switch Grating ---> DMC sequence 10 blocks:
  - OBCP and DMC preparation (label undefined, 0)
  - first grating switch sequence (label 33)
  - second grating switch sequence (label 97)
  - third grating switch sequence (label 161)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 23: Ge:Ga Set-up ---> no DMC sequence, no blocks
- OBCP 24: Switch to SAFE ---> no DMC sequence, no blocks
- OBCP 25: Time Synchronisation Test 1 ---> no DMC sequence, no blocks
- OBCP 26: Time Synchronisation Test 2 ---> no DMC sequence, no blocks
- OBCP 27: Grating Line Scan Chopped 2 ---> DMC sequence 12 blocks:
  - OBCP and DMC preparation (label undefined, 0)
  - chopped up-scan sequence on target (labels 3,5)
  - chopped up-scan sequence on CSs (labels 65,129)
  - chopped down-scan sequence on target (labels 19,21)
  - chopped down-scan sequence on CSs (labels 81,145)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 28: Grating Line Scan Without Chopping ---> DMC sequence 13 blocks
  - OBCP and DMC preparation (label undefined, 0)
  - up-scan sequence on target + CSs (labels 3,65,129)
  - down-scan sequence on target + CSs (labels 19,81,145)
  - DMC and OBCP Reset (label 0, undefined)
- OBCP 29: Generate dummy science packets ---> no DMC sequence, no blocks
- OBCP 30: SPU test mode SPEC ---> no DMC sequence, no blocks
- OBCP 31: SPU test mode PHOT ---> no DMC sequence, no blocks
- OBCP 32: BION ---> no DMC sequence, no blocks
- OBCP 33: OBMO ---> no DMC sequence, no blocks

- OBCP 34: ACWE ---> no DMC sequence, no blocks

3.7.3. Pre-processing of the calibration blocks

In the current observation design strategy a calibration block is executed at the beginning of any observation. It is possible that in the future the current design will be changed to include more than one calibration block to be executed at different times during the observation. In order to take into account this possible change, the pipeline includes as a very first step a pre-processing of the calibration block(s) that is planned to work under any possible calibration blocks configuration. The calibration block pre-processing is done in three steps: a) the calibration block(s) is identified and extracted from the frames class, b) it is reduced by using appropriate and pre-existing pipeline steps, c) the result of the cal block data reduction is attached to the frames class to be used in the further steps of the data reduction.

3.7.3.1. photCSExtraction

This is the first step of the calibration block pre-processing. This task identifies the calibration block(s) of the given observation and it stores it in an additional Product class (csbasket example below). The input frames remain unchanged.

```
>> csbasket = photCSExtraction(Frames inFrames)
csbasket -: PhotTrendProducts -: list of frames containing calibration blocks. One
block per frame.
inFrames -: Frames     -: Frames in
```

PhotTrendProducts contains two kinds of container : the first one gathers frames related to the calibration blocks, second one keeps compiled housekeeping information for the further trend analysis. Here is some useful command to explore this product :

```
print csbasket.cs -: PhotCSProducts is a list of frames and give an overview of
the cs blocks
print csbasket.cs.get(0) -: Frames conatins information on the first calibration
block (index 0)
print csbasket.trend -: PhotTrendProducts is a list of PhotTrendProduct. This
product is available after PhotCsProcessing task execution
                       -"print" gives a summary of the available commands and an
overview on all calibration blocks
print csbasket.trend.get(0) -: PhoTrendProduct contains compiled hk information on
the first cs block (index 0)
print csbasket.trend.get(0).cs1Temperature is an average of the temperature of cs1
print csbacket.trend.get(0).cs2Temperature is an average of the temperature of cs2
print csbasket.trend.get(0).cs1Cpr is an average of the chopper position on cs1
print csbasket.trend.get(0).cs2Cpr is an average of the chopper position on cs2
print csbasket.trend.get(0).cs1TemperatureStdDev
print csbasket.trend.get(0).cs2TemperatureStdDev
print csbasket.trend.get(0).mode  {"Direct", -"DDCS"}
print csbasket.trend.get(0).bias  average of Vh-Vl in Volt
print csbasket.trend.get(0).gain  {"high","low"}
print csbasket.trend.get(0).startTime finetime at the beginning of the calibration
block
print csbasket.trend.get(0).endTime finetime at the end of the calibration block
```

Please see also Trend chapter.

3.7.3.2. photCSProcessing

Once the calibration block is identified and stored in a proper frames class the calibration data can be reduced. The input of this task are the product containing the calibration block (csbasket of previous task), the House Keeping (hkdata as in section 7.1). The output of the task is the differential image of the two calibration sources plus several House Keeping values extracted from the hkdata variable. Those info are necessary to correct any drift in the flat-field and flux calibration (see sections related to this tasks for more details).

```
>> outCsBasket = photCSProcessing(inCsBasket, hkdata[,calTree=calTree]
[,sigclip=True|False][,nsgima=n][,quality])
```

```
inCsBasket  -: PhotTrendProducts -: list of raw calibration blocks -- the result
of the module photCSExtraction
outCsBasket -: PhotTrendProducts -: list of processed frames containing
calibration blocks.
hkdata      -: TableDataset      -: housekeeping information extracted from the
observation context
calTree     -: CalibrationTree   -: calibration tree with current calibration
products
sigclip     -: int -:
                 0 -- sigma clipping is disable
                 1 -- sigma clipping is applied on the calibration blocks
nsigma      -: double -: if sigclip is activated nsigma gives the maximum standard
deviation authorized. Pixels out of range are flagged
              in DCsMask dataset and excluded from noise computation (DCsNoise) --
This operation is done by PhotCSDiff task.
quality     -: QualityContext -:
```

Since the calibration block is nothing else than a chopped observation, the calibration data are reduced in analogy to the Point-source data. Thus, this module call all the remaining tasks described in the current section up to level 0.5 and few specific tasks of the Point-source pipeline between level 0.5 and 1. We list below the tasks called in the execution of this module:

- photFlagBadPixels adds badpixel mask to the frames

- photFlagSaturation adds a mask containing pixels saturated according to the bolometer settings.

- photConvDigit2Volts converts the calibration block signals into Volt

- photCorrectCrossTalk corrects the crosstalk of each pixel

- photMMTDeglitching flags/removes glitches found in the signal

- photCleanPlateau identifies chopper plateau at the calibration source

- photAvgPlateau calculates the average of each plateau

- photCSDiff calculates the differential image of the two calibration sources CS1-CS2, in addition it collects the housekeeping data relative to the calibration, such as gain and bias settings of the observation. blocks

Output : after the execution of this task each frames stored in csFrames contains five more pieces of information :

```

DCs      differential image of the calibration sources (CS1-CS2)
DCsNoise noise image of the CS1-CS2 subtraction
DCsTable a TableDataset containing housekeeping data relative to the calibration
blocks
DCsMask  a mask of the pixel always masked in the calibration blocks
DCsCoverage a weight pixel image
```

We point out that DCs, DCsNoise, DCsTable, DCsMask are used by photDriftCorrection module only.

photCSDiff (photDiffCal former task)

Among the tasks used for the reduction of the calibration block and listed above, only photCSDiff is specific to this case. All the other tasks will be described in the point-source and small-source pipelines. Here we describe what this task is doing in detail.

photCSDiff adds a TableDataset named DCsTable to the frame. For each calibration block encountered new rows is added to the table. Each column contains one type of information such as the CPR position, start time and end time of the calibration block and so on). This table is reused later on by photDriftCorrection. Here is briefly the name of the columns:

- index : calibration block index for this obsid
- channel : channel currently processed by the pipeline

- startTime : start time of the calibration block (finetime)
- endTime : end time of the calibration block (finetime)
- middleTime : mean time of the calibration block (finetime)
- cs1Temperature : average of CS1 temperature found inside the calibration block
- cs2Temperature : average of CS2 temperature found inside the calibration block
- cs1TemperatureStdDev : gives the standard deviation of the temperature of CS1
- cs2TemperatureStdDev : gives the standard deviation of the temperature of CS2
- cs1Cpr : the average of the chopper position when PACS looks at its first calibration source
- cs2Cpr : the average of the chopper position when PACS looks at its second calibration source
- mode : median of the readout mode (Direct or DDCS)
- bias : average of <VH-VL> for all BU
- gain : low or high (1 or 0)
- cprFrequency : velocity of the chopping between CS1 and CS2 in Hz
- Filename : possible reusable file name "PTrendPhotometer_diffCS_'Date=YYMMDD_hhmmss'_FM_"

PhotCSDiff adds as well two additional computations to the frame : the difference of the calibration source (indexed with 'DCs' keyword) and the noise computed for each calibration block (indexed with 'DCsNoise' keyword).

Here are the formulas used to compute the noise "DCsNoise" :

* for each difference dnoise = SQRT(noise$_1{}^2$+noise$_2{}^2$) and globaly DCsNoise=SQRT(<dnoise^2>)/ SQRT(n)

*noise$_1$ and noise$_2$ parameters above, are the noises computed by photAvgPlateau during respectively CS1 and CS2 observation. While n parameter is the number of samples of cs1-cs2 measurements averaged on the calibration block interval.

PhotCSDiff adds as well a mask of the pixel always masked in the master mask inside the calibration block

At last, this task adds a dataset named DCsCoverage, this latter contains the weight of each pixel

3.7.3.3. photCSClean

We pointed out in the description of the PhotCSExtraction task, that the input frames remain unchanged. That frames is the class where we stored all the Level 0 products, including the calibration block. Since we already reduced the cal block in the previous task and we extracted the useful info from it, we now want to remove it from the original frames in order to keep only the scientific data and to store only the essential information about the calibration sources. Thus, this task simply removes the cal block from the original frames and replaces it with the output of the previous task. The output frames will contain the scientific data plus the DCs and DCsNoise images and DCsTable (see previous task).

```
>> outframes = photCSClean(inFrames)

outFrames -: Frames     -: Frames out
inFrames  -: Frames     -: Frames in
csFrames  -: PhotTrendProducts -: calibration block encapsulated into frame --
result of photCSProcessing module
clean     -: int        -:
                       0 -- keep calibration block
```

```
                        1 -- remove calibration blocks from the frames
```

3.7.4. photFlagBadPixels

The purpose of this task is to flag the damaged pixels in the BADPIXEL mask. The task should do a twofold job: a) reading the existing bad pixel mask provided by a calibration file ("PCalPhotometer_BadPixelMask_FM_v1.fits" for the first release), b) identifying additional bad pixels during the observation. In the current version of the pipeline only the first functionality is activated. The algorithm for the identification of additional bad pixels is not in place. So the task is just reading the bad pixel calibration file and transforming the 2D mask contained in it in the 3D BADPIXEL mask.

```
>> outFrames = photFlagBadPixels(inFrames [,calTree=calTree] [,copy=copy] -)


outFrames  -: Frames -: Frames out
inFrames   -: Frames -: Frames in
calTree    -: PacsCalibrationTree -: calibration tree containing all calibration
products used by the pipeline
copy       -: int    -:
                    0 -- return reference
                    1 -- return copy
```

3.7.5. photFlagSaturation

This task checks saturation based on the readout electronics saturation (called CL saturation) and ADC converter (called ADC saturation). By default both kinds of saturation are checked. Checking is control by the option "check" in the command line.

CL saturation : is based on the knowledge of the transfer function of the readout electronics. Electronic settings are extracted from housekeeping data and injects in the transfer function. Two saturation images are computed (low and high saturation) and compared to the signal. Thus two masks are generated respectively "CL_SATURATION_HIGH" and "CL_SATURATION_LOW".

ADC saturation checking is very straight forward :this tasks identifies the saturated pixels on the basis of saturation limits contained in a calibration file. Before doing that, the task identifies the reading mode led by the warm electronic BOLC (Direct or DDCS mode) and the gain (low or high) used during the observation. These information are provided for each sample of the science frames by the BOLST entry in the status table. The task compares the pixel signal at any time index to the dynamic range corresponding to the identified combination of reading mode and gain. Readout values above the saturation limit are flagged in two masks called respectively "ADC_SATURATION_HIGH" and "ADC_SATURATION_LOW".

At last, Saturation task merges all masks in a 3D "SATURATION" mask. For analysis purpose, additionnal masks are kept and contain intermediate merging operation: "SATURATION_HIGH" and "SATURATION_LOW"

```
>> outFrames = photFlagSaturation(inFrames[,calTree=calTree]
[,hkdata=hk][,check=check][,satLimits=satLimits][,clLimits=clLimits]
[,clTransferFunction=clTranferFunction][,copy=copy])
outFrames  -: Frames -: Frames out
inFrames   -: Frames -: Frames in
calTree    -: PacsCalibrationTree -: calibration tree containing all calibration
products used by the pipeline
hkdata     -: TableDataset        -: housekeeping information extracted from the
observation context
check      -: String              -: { -"adc","cl", -"full" (default)} is the kind
saturation checking done by the task
satLimits  -: SatLimits           -: is the ADC saturation limits (cal product)
used according to the mode used.
clLimits   -: ClSaturationLimits  -: is the CL saturation limits (cal product)
clTranferFunction -: ClTransferFunction -: is the CL transfer function (cal product)
copy       -: int    -:
                     0 -- return reference
                     1 -- return copy
```

```
Valid calls -:
frames = photFlagSaturation(frames,calTree)                     --- adc checking
by default
frames = photFlagSaturation(frames,calTree,hk,"cl")             --- cl checking only
frames = photFlagSaturation(frames,calTree,hk,"adc")            --- adc checking
only
frames = photFlagSaturation(frames,calTree,hk,"full")           --- cl & adc
checking
```

Literature reference :

Saturation limits for the PACS Photometer - M.Sauvage, N.Billot, K,Okumura - July 22, 2008

Detecting and flagging saturated pixels in the PACS pipeline - M.Sauvage - February 10, 2009

3.7.6. photConvDigit2Volts

The task converts the digital readouts to Volts. As in the previous task, as a first step the task identifies the reading mode and the gain on the basis of the the BOLST entry in the status table for each sample of the frame. This is redundant and this step will be skipped when mode and gain will be stored in the metadata of the Level 0 Product. The task extracts, then, the appropriate value of the gain (high or low) and the corresponding offset (positive for the direct mode and negative for the DDCS mode) from the calibration file (PCalPhotometer_Gain_FM_v1.fits for the first version). These values are used in the following formula to convert the signal from digital units to volts:

signal(volts) = (signal(ADU) - offset) * gain

```
>> outFrames = photConvDigit2Volts(inFrames [,calTree=calTree]
[,photGain=photgain] [,copy=copy])

outFrames  -: Frames -: Frames out
inFrames   -: Frames -: Frames in
calTree    -: PacsCalibrationTree -: calibration tree containing all calibration
products used by the pipeline
photGain   -: gain   -: nominal gain (1, 100 uV/step), low gain (5, 20 uV/step) or
high gain(20, 5uV/step)
copy       -: int    -:
                     0 -- return reference
                     1 -- return copy
```

Reference : BOLC TO DMC ELECTRICAL INTERFACE CONTROL DOCUMENT (SAp-PACS-cca-0046-01)

3.7.7. photCrossCorrection

The phenomenon of electronic crosstalk was identified, in particular in the red bolometer, during the testing phase. The working hypothesis of this task is that the amount of signal in the crosstalking pixel is a fixed percentage of the signal of the correlated pixel. A calibration file (PCal_PhotometerCrosstalkMatrix_FM_v2.fits in the current release) reports a table containing the coordinates of crosstalking and correlated pixels and the percentage of signal to be removed, for the red and the blue bolometer, respectively, . The task reads the calibration file and use the info stored in the appropriate table to apply the following formula:

Signal_correct(crosstalking pixel)) = Signal(crosstalking pixel) - a*Signal(correlated pixel)

where 'a' is the percentage of signal of the correlated pixel to be removed from the signal of the crosstalking pixel.The task is still under investigation, in the sense that invariability of 'a' is still an assumption to be tested in further tests.

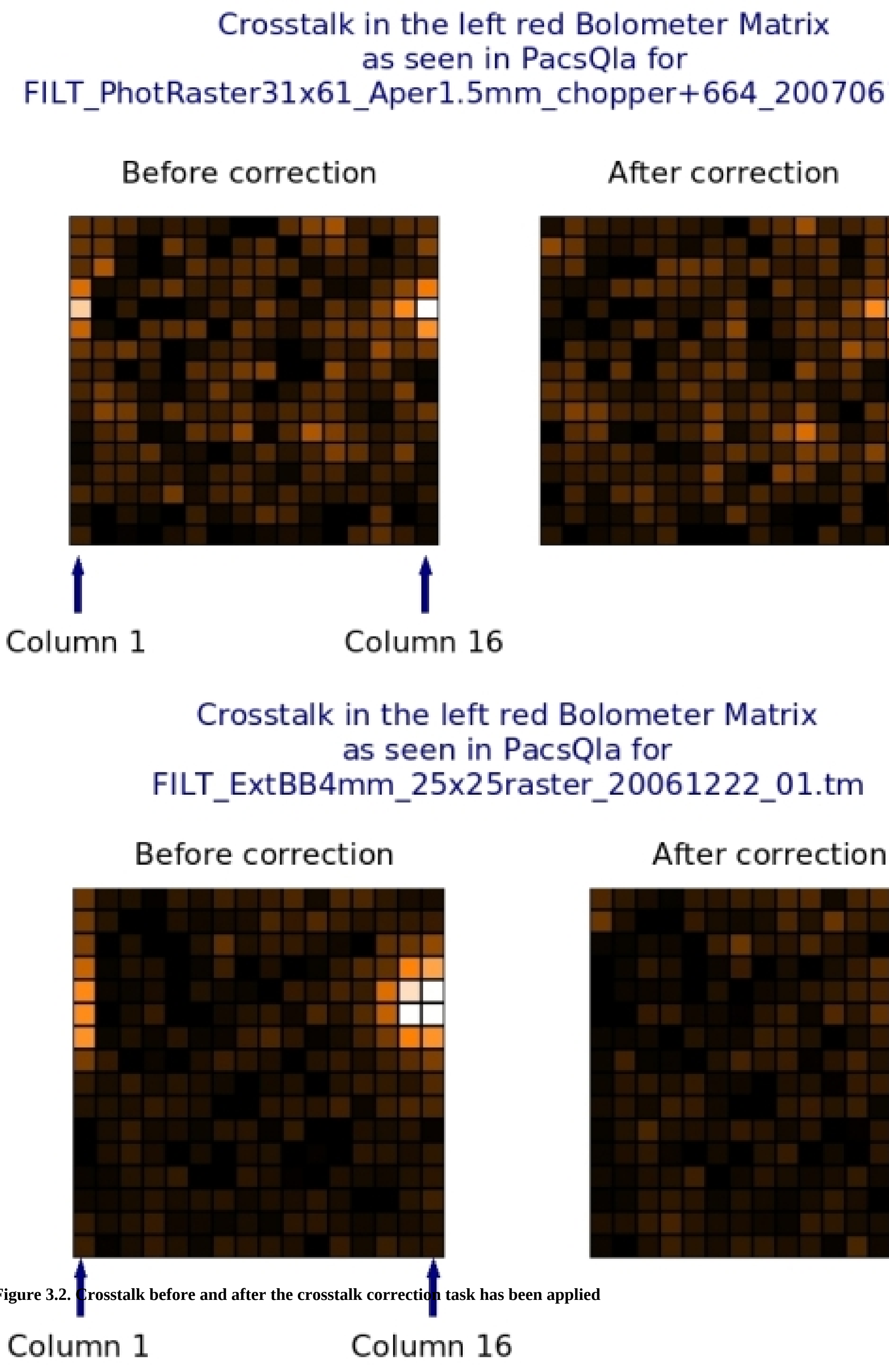


Figure 3.2. Crosstalk before and after the crosstalk correction task has been applied

In the above two images we show two examples of electronic crosstalk in the red bolometer for different source fluxes. The left side shows the situation before the correction. The right side shows the result after the correction. The task removes succefully the fraction of the signal in column 1 due to the correlated column 16. However, it is worth to notice that in the second case, the crosstalk is somehow over-corrected. This would imply that 'a' could depend also on the signal of the correlated pixel. Moreover it is known that the amount of crosstalk can be influenced also by the photometer bias voltage settings. Future tests are planned to explore all these possibilities, in particular, a) finding a bias settings able to minimize/avoid the crosstalk or, in alternative, b) studying the dependence of 'a' on the bias settings and providing a new calibration file which takes into account this dependence.

```
>> outFrames = PhotCorrectCrosstalk(inFrames [,copy=copy])

outFrames  -: Frames -: Frames out
inFrames   -: Frames -: Frames in
copy       -: int    -:
                     0 -- (False) return reference
                     1 -- (True) return copy
```

Reference: D. Lutz, P. Popesso. Bolometer Spatial Calibration, PACS Test Analysis Report FM-ILT Version 0.0 from October 25/2007

3.7.8. photMMTDeglitching and photWTMMLDeglitching

These tasks detect, mask and remove the effects of cosmic rays on the bolometer. Two different tasks are implemented for the same purpose: photMMTDeglitching is based on the multiresolution median transforms (MMT) proposed by Starck et al (1996), WTMMLDeglitching is based on the Wavelet Transform Modulus Maxima Lines Analysis (WTMML). The former task is in the testing phase. The tests aim at identifying suitable ranges of parameters for different scientific cases. The latter task is still under investigation and debugging phase. At this stage of the data reduction the astrometric calibration has still to be performed. Thus, the two tasks can not be based on redundancy. Both tasks have to overcome the following problems:

- signal fluctuation of each pixel,
- the movement of the telescope,
- the hits received by one pixel due to several cosmic rays having different signatures and arrival time,
- the non-linear nature of each glitch.

3.7.8.1. Deglitching using the Multiresolution Median Transform (photMMTDeglitching)

This task is based on the method developed by Starck et al. (1998) for the detection of faint sources in ISOCAM data. The method relies on the fact that the signal due to a real source and to a glich, respectively, when measured by a pixel, shows different signatures in its temporal evolution and can be identified using a multiscale transform which separates the various frequencies in the signal. Once the "bad" components due to the glitches are identified, they can be corrected in the temporal signal. Basically, the method is based on the multiresolution support. We say that a multiresolution support (Starck et al. 1995) of an image describes in a logical or boolean way if an image f contains information at a given scale j and at a given position (x,y). If the multiresolution support of f is M(j,x,y)=1 (or true), then f contains information at scale j and position (x,y). The way to create a multiresolution support is trough the wavelet transform. The wavelet transform is obtained by using the multiresolution median transform. The median transform is nonlinear and offers advantages for robust smoothing. Define the median transform of an image f, whit the square kernel of dimension n x n, as med(f,n). Let n=2s +1; initially s=1. The iteration counter will be denoted by j, and S is the user-specified number of resolution scales. The multiresolution median transform is obtained in the following way:

1. Let $c_j = f$ with $j = 1$.
2. Determine $c_{j+1} = \mathrm{med}(f, 2s+1)$.
3. The multiresolution coefficients w_{j+1} are defined as: $w_{j+1} = c_j - c_{j+1}$.
4. Let $j \longleftarrow j+1$; $s \longleftarrow 2s$. Return to step 2 if $j < S$.

Figure 3.3.

A straightforward expansion formula for the original image (per pixel) is given by:

$$c_0(x, y) = c_p(x, y) + \sum_{j=1}^{p} w_j(x, y).$$

Figure 3.4.

where, cp is the residual image. The multiresolution support is obtained by detecting at each scale j the significant coefficient wj. The multiresolution support is defined by:

$$M(j, x, y) = \begin{cases} 1, & \text{if } w_j(x, y) \text{ is significant;} \\ 0, & \text{if } w_j(x, y) \text{ is not significant.} \end{cases}$$

Figure 3.5.

Given stationary Gaussian noise, the significance of the w_j coefficients is set by the following conditions:

$$\text{if } |w_j| \geq k\sigma_j, \text{ then } w_j \text{ is significant;}$$

$$\text{if } |w_j| < k\sigma_j, \text{ then } w_j \text{ is not significant.}$$

Figure 3.6.

where sigma_j is the standard deviation at the scale j and k is a factor, often chosen as 3. The appropriate value of sigma_j in the succession of the wavelet planes is assessed from the standard deviation of the noise, sigma_f, in the original f image. The study of the properties of the wavelet transform in case of Gaussian noise, reveals that sigma_j=sigma_f*sigma_jG, where sigma_jG is the standard deviation at each scale of the wavelet transform of an image containing only Gaussian noise. The standard deviation of the noise at scale j of the image is equal to the standard deviation of the noise of the image multiplied by the standard deviation of the noise of the scale j of the wavelet transform. In order to properly calculate the standard deviation of the noise and, thus, the significant wj coefficients, the tasks applies an iterative method, as done in starck et al. 1998:

- calculate the Multiresolution Median Transform of the signal for every pixel
- calculate a first guess of the image noise. The noise is estimated using a MeanFilter with boxsize 3 (Olsens et al. 1993)
- calculate the standard deviation of the noise estimate
- calculate a first estimate of the noise in the wavelet space
- the standard deviation of the noise in the wavelet space of the image is then sigma(j) = sigma(f)*sigma(jG) (Starck 1998).
- the multiresolution support is calculated
- the image noise is recalculated over the pixels with M(j,x,y)=0 (containing only noise)

- the standard deviation of the noise in the wavelet space, the multiresolution support and the image noise are recalculated iteratively till (noise(n) - noise(n-1))/noise(n) < **noiseDetectionLimit**, where noiseDetectionLimit is a user specified parameter

  (Note: if your image does not contain pixels with only noise, this algorithm may not converge. The same is true, if the value **noiseDetectionLimit** is not well chosen. In this case the pixel with the smallest signal is taken and treated as if it were noise)

At the end of the iteration, the final multiresolution support is obtained. This is used to identify the significant coefficients and , thus, the pixels and scales of the significant signal. Of course, this identifies both glitches and real sources. According to Starck et al. (1998), at this stage a pattern recognition should be applied in order to separate the glitch from the real source components. This is done on the basis of the knowledge of the detector behavior when hit by a glitch and of the different effects caused in the wavelet space by the different glitches (short features, faders and dippers, see Starck at al. 1998 for more details). This knowledge is still not available for the PACS detectors. At the moment, a real pattern recognition is not applied and the only way to isolate glitches from real sources is to properly set the user-defined parameter **scales** (S in the description of the multiresolution median transform above). The method works reasonably well till the maximum number of readouts of a glitch is much smaller than the one of a real source (**scales** < 5, see the alpha irradiation example below). For higher value of **scales** (< 5), also part of the signal of a real bright source can be identified as a glitch (see proton irradiation example below).

When the glitches are identified the signal of the pixel affected is corrected by interpolating the signal before and after the glitch event. In addition, the task also produces the 3D GLITCH mask which flags the deglitched pixels at any time (due to a bug of the task the GLITCH mask is not always produced, this is under investigation). However, it is required that the task does not correct by default the signal of the pixels affected by glitches. It is foreseen that the task will provide the user the possibility to choose whether to correct or not the signal or to have only the GLITCH mask as a result.

Literature reference for this algorithm:

ISOCAM Data Processing, Stark, Abergel, Aussel, Sauvage, Gastaud et. al., Astron. Astrophys. Suppl. Ser. **134**, 135-148 (1999)

Automatic Noise Estimation from the Multiresolution Support, Starck, Murtagh, PASP, **110**, 193-199 (1998)

Estimation of Noise in Images: An Evaluation, Olsen, Graphical Models and Image Processing, **55**, 319-323 (1993)

Details and Results of the implementation

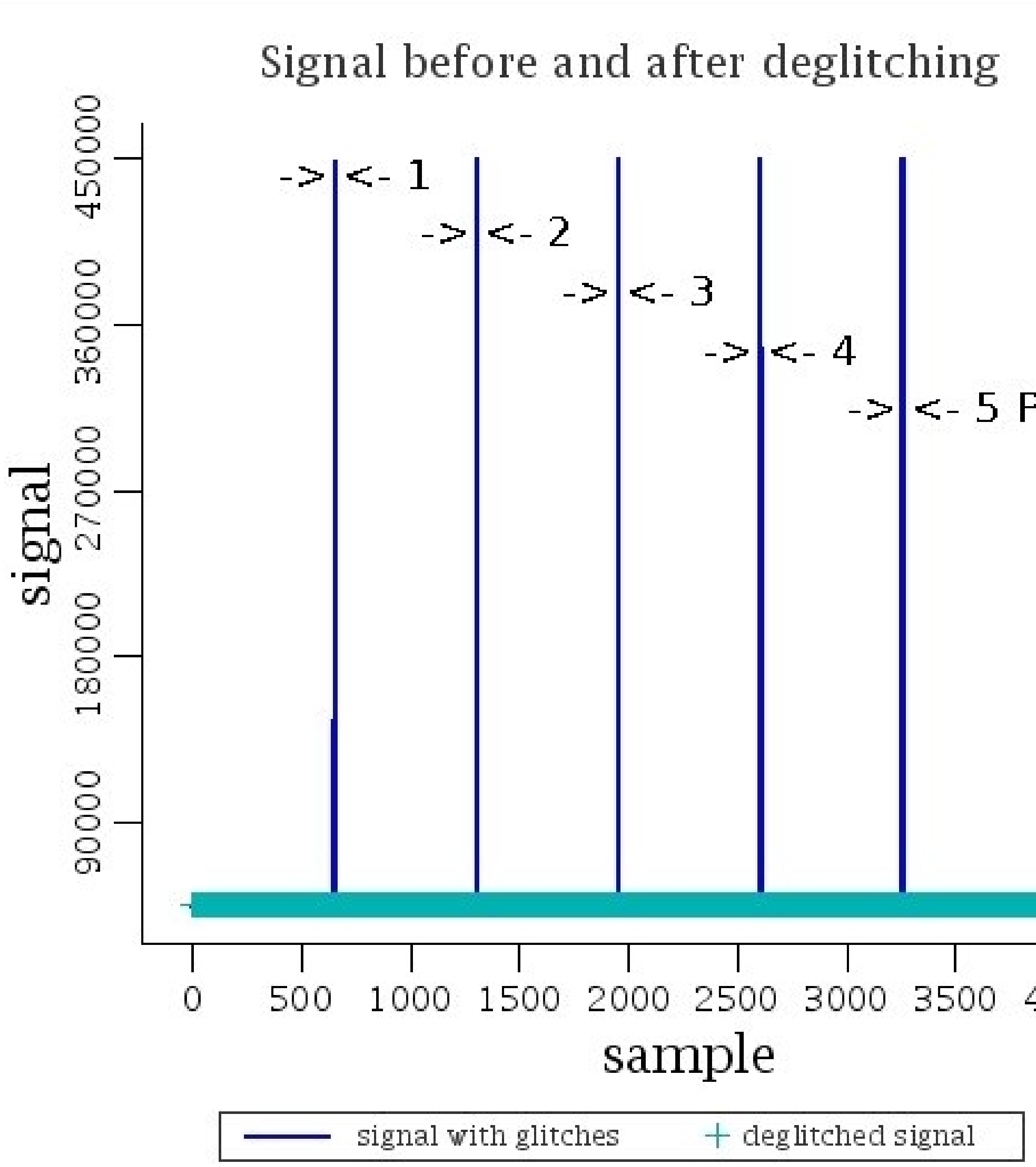
This is the signature of the task:

```
>> outframes = mMTDeglitching(inFrames [,copy=copy] [,scales=scales]
[,mmt_startenv=mmt_startenv] [,incr/fact=incr/fact] [,mmt_mode=mmt_mode]
[,mmt_scales=mmt_scales] [,nsigma=nsigma])

outFrames  -: the returned Frames object
inFrames   -: the Frames object with the data that should be deglitched
copy       -: boolean. Possible values: false (jython: 0) -- inFrames will be
modified and returned
                               true (jython: 1) -- a copy of inFrames will be
returned

scales     -: int. Number of wavelet scales. This should reflect the maximum
expected readout number of the glitches. Default is 5 readouts.
mmt_startenv -: int. The startsize of the environment box for the median
transform. Default is 1 readout (plus/minus).
incr/fact  -: float. Increment resp. factor to enhance the mmt_startenv. Default
is 1.
mmt_mode   -: int. Defines how the environment should be modified between the
scales. Possible values: 1 (ADD) or 0 (Multiply). Default is 1.
```

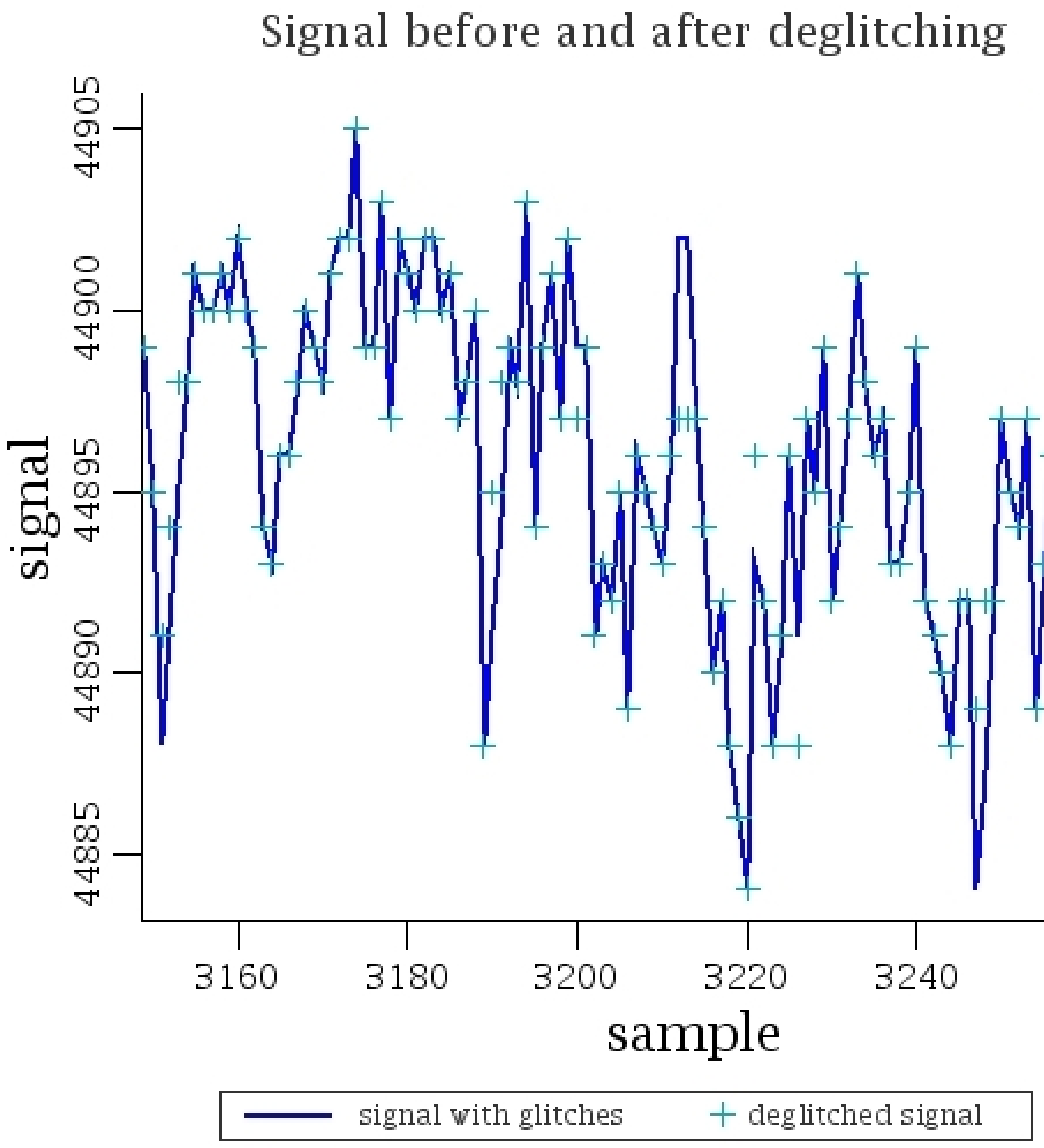
```
              example: the environment-size for the subsequent median transform
environment boxes will be  env(0) = mmt_startenv, env(n) = env(n-1) mmt_mode incr/
fact
                      default means then: env(0) = 1, env(1) = 1 + 1, env(2) = 3
etc.
noiseDetectionLimit -: double. Threshold for determining the image noise. values
between 0.0 and 1.0. Default is 0.3.
nsigma     -: int. Limit that defines the glitches on the wavelet level. Every
value larger than nsigma*sigma will be treated as glitch. Default is 5.
```

Results of example data

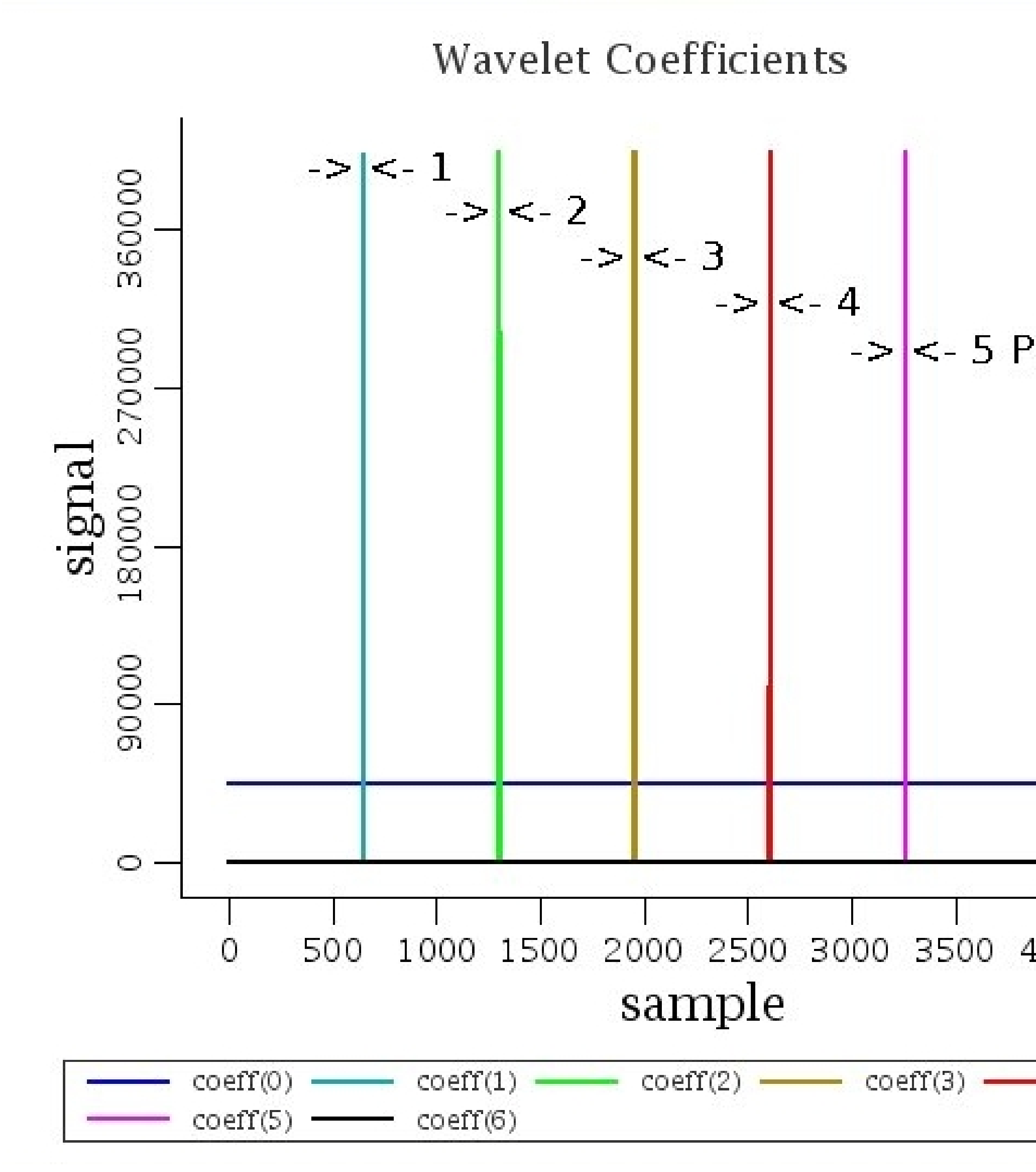
To examine the result of this algorithm, a real Bolometer signal has been taken and artificial glitches with a width of 1, 2, 3, 4 and 5 pixels have been added to the signal of one of the pixels. The analysis has then been done with 6 wavelet scales.



A closer look at the signal and the deglitched signal shows the quality of the processing.



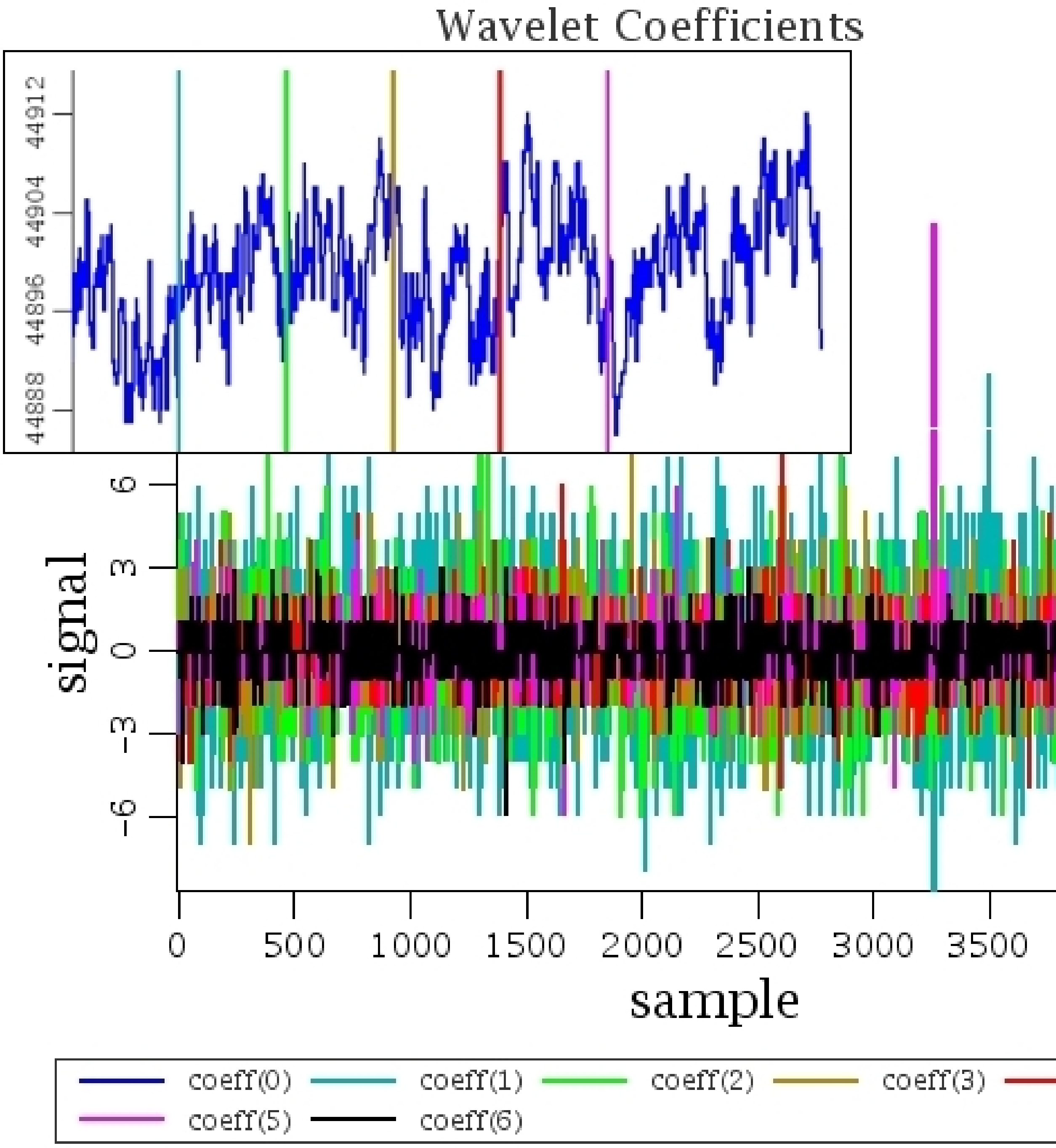
The wavelet coefficients are crucial for this deglitching process.



From the image it is obvious, that the glitches are placed according to their width into the wavelet coefficients. This fact is due to the choice of the median transform and the configuration of the environment (plus/minus 1 for coeff 1, plus/minus 2 for coeff 2 etc.). Please note also that coefficient 6 does not contain data (glitches have widths up to 5). Please find details of the Multiresolution Median Transform in Starck et al. (1999).

The baseline of the coefficients is zero, not the signal level. The signal level appears only in coefficient 0! Coefficient 0 has also been used to remove the background from the image while the noise has been estimated. From this plot it is obvious, why coeff(0) has been used.

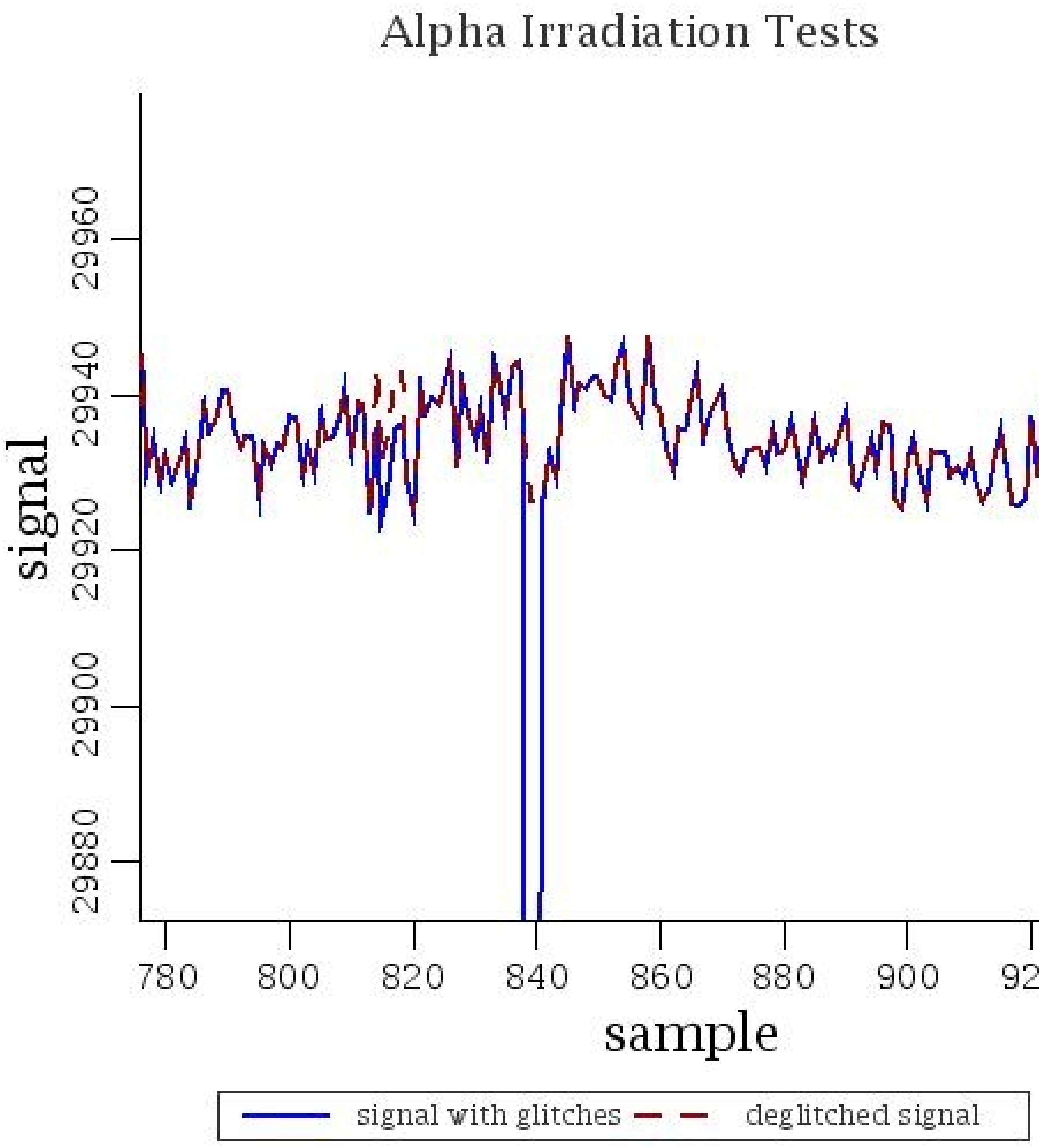
Here is a closer look at the wavelet coeficients:



Although the wavelets baseline is zero a little noise is there. Its even in the same order of magnitude as the original Signals noise. Thats why it is important to have a good noise estimate to remove the glitches.

Alpha and Proton Irradiation Tests

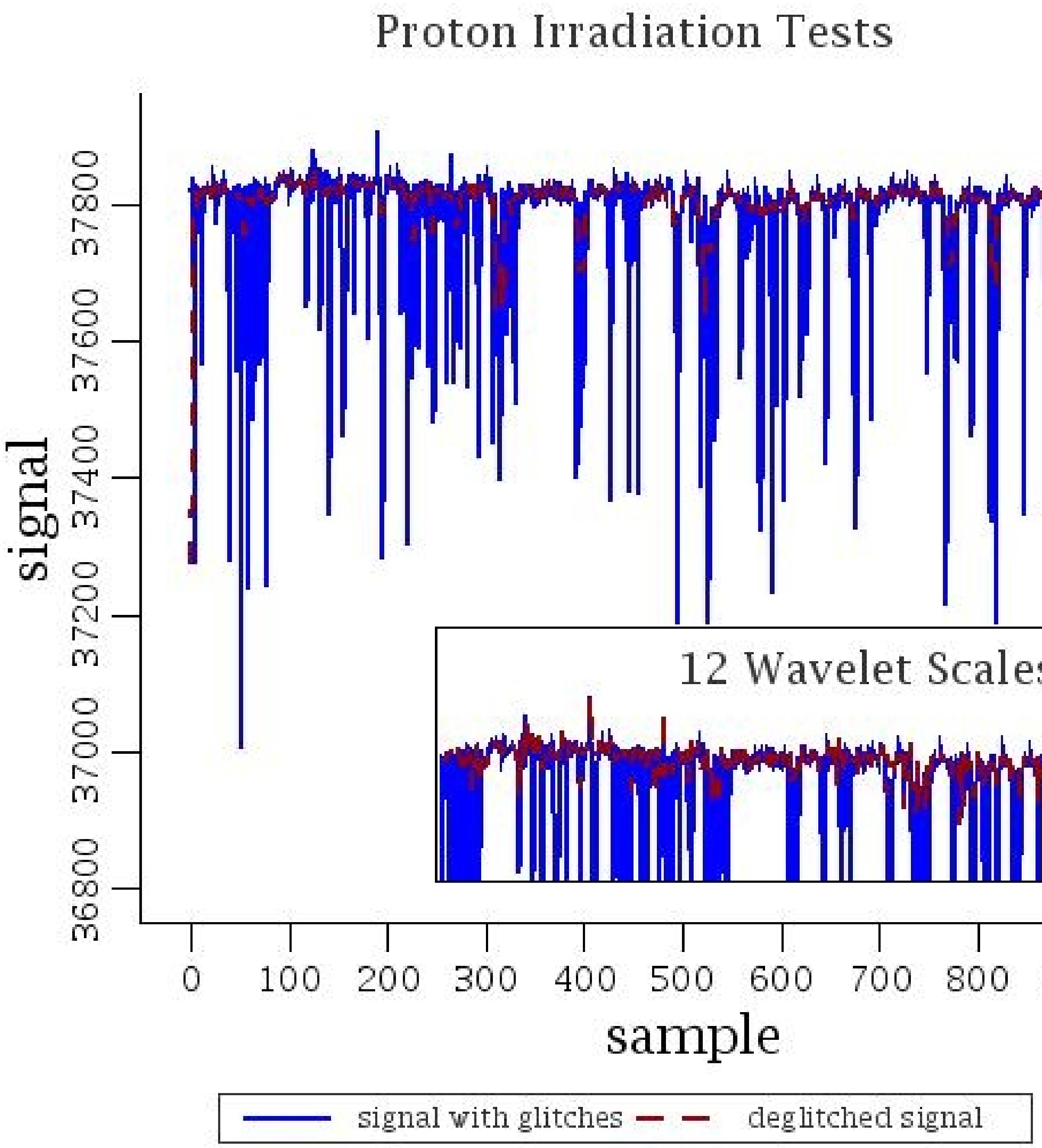
Here are the results of the Multiresolution Median Transform deglitching applied to the Bolometer irradiation test with alpha particles and protons from (CITATION NEEDED).



The glitches of the alpha particles do not differ significantly from the artificial glitches. Their width is around 1-3 readouts and their signal is much higher than the average signal, so the nsigma method works as well as expected.

The proton tests have a distinctly different pattern. The glitches are much higher in number and their width can also be large. The image shows, that increasing the wavelet scales from 6 to 12 leads to a better removal of the glitch structures. The problem in these measurements is that the number of the glitches is so high, that a good estimate of the noise is hard to do. There is no pixel without a glitch. Thus the estimated noise will be high and small glitches are not removed.

Anyhow, experimenting with the settings is in every case worth a try as the 12 wavelet scale inset shows.



3.7.8.2. Wavelet Transform Modulus Maxima Lines Analysis (photWTMMLDeglitching)

During the mission, Herschel observatory and PACS instrument will be irradiated by a lot of particles coming from any directions. When particles hit the telescope shield or materials near the detectors, PACS can see these impacts which pollute the interesting signal. These traces are called glitch signatures. Several kinds of signature can exist. In general glitch can be characterized by a variation of the signal where the raising time equals about the decay time. There are the 'fader' glitch types characterized by a raising time greater than the decay e-(t .Ï„) and at last the 'dipper' types where the energetic particles produce a very long time constant. In this case the signal stay polluted during a long time period. These kinds of signature and terminologies have come from ISOCAM papers.

In order to see the behavior of the detector, on May 26th and 27th,2005, irradiation tests have been carried out at Orsay/France with Tandem accelerator. Orthogonal irradiation with alpha and proton have been carried out. Data obtained during these tests are a good start to check deglitching algorithms but can be insufficient.

Thanks to previous missions, we have now some models of the cosmic particles into our considered space area. Despite of these models and the knowledge of the behavior of our detector, it isn't possible to know exactly the glitch signatures that will be encountered in the space, hence it is important to have a flexible deglitching algorithm. The MMT method showed above is well adapted because there is no assumption about the glitch signature.

Nevertheless, one can try to work on the shape of the glitch. This direction has been explored by the Spire's developers with the 'WTMML' analysis (Wavelet Transform Modulus Maxima Lines). This method tries to recognize the temporal shape of the glitches. With their courtesy permission, their algorithm has been adapted for PACS. This section tries to give an evaluation of the algorithm with its strength, its weakness and the limits of its applicability.

Overview

Status : first version - not ready yet. Author b.Marin based on WTMML software developed by C.Ordenovic, C.Surace, B.Torresani, A.Llebaria

Reference literature for this algorithm and data used:

- Faint source detection in ISOCAM images, J.L. Starck, H. Aussel, D. Elbaz, D. Fadda, and C. Cesarsky, A&A Suppl. Ser. 138,365-379 (1999)
- Glitches detection and signal reconstruction using HÃ¶lder and wavelet analysis, C.Ordenovic, C.Surace, B.Torresani, A.Llebaria STAMET-D-07-00048
- A wavelet tour of signal processing, Mallat
- Glitch effects in ISOCAM detectors A.Claret, H.Dzitko, J.Engelmann and J.-L. Starck
- Herschel/PACS Description des irradiations Tandem 2005 [Ref: SAp-PACS-BH-0470-05 ver 1.1] Benoit HOREAU

Principle

For each pixel(x,y), the method consists of doing the following steps :

1. set s(t) = D(x,y,t) ; where D represents sampling data and t the time
2. A multi-resolution decomposition of the signal is done using Mexican Hat wavelet. the result is Ws(b,a)
3. Signal irregularities are tagged by the study of the evolution of |Ws(b,a)| in the time-scale plane(b,a)
4. noise is estimated
5. Irregularities not identified as noise are tagged as glitches
6. Glitch contributions are estimated and removed from the decomposed signal
7. Signal is rebuilt

Details and results of the implementation

```
>> outFrames = wtmmlDeglitching(inFrames [,copy=copy][,scaleMin=2.0][,scaleMax=6.0]
              [,hmin=-1.3][,voices=5.][,holderThreshold=-0.6][,CorrThreshold=0.985]
              [,reconstruction=True]

inFrames  -: the input frame object containing signal to analyse
outFrames -: the returned Frame object containing signal deglitched and a mask of
pixels modified
```

```
copy      -: boolean with the possible values -:
               false (jython: 0) -- inFrames will be modified and returned
               true (jython: 1)  -- a copy of inFrames will be returned
scaleMin  -: signal continuous wavelet transform is computed from scaleMin
scaleMax  -: signal continuous wavelet transform is computed till scaleMax
voices    -: voices number by octave
hmin      -: the minimal holder value allowed
holderThreshold -: must be greater than hmin -- the threshold holder exponent
corrThreshold   -: correlation coefficient threshold (around 1.) is a criteria used
                  to identify an irregularity of the signal
reconstuction   -: Boolean: {true|false} = {inFrames is changed -| inFrames is not
changed}
```

More parameter descriptions

Wavelet transform will be computed from scaleMin to scaleMax. Octave number (nOct) is log(scaleMax)/log (2) (dyadic decomposition) and there are nVoice voices by octave. The scale a of the octave o and the voice v is a = 2^(nOct*(o-1)+v/nVoice)

Correlation threshold (close to value 1.) is a criteria used to identify a potentially irregularity of the signal as a possible glitch.

holderThreshold gives an upper limit of the acceptable holder exponent found

hmin gives a lower limit of the acceptable holder exponent found

hmin < holder exponent found < holderThreshold

Algorithm description

1. Multiresolution signal decomposition is performed from minScale to maxScale. The Mexican hat wavelet is used here.

2. Along each scale, locally maxima are identified. In other words, if dWs(b0,a0)/db = 0 the point b0,a0 is a locally maximum.

3. Across the scales, 'maxima lines' are researched. A maxima line is any curve a(b) in the scale-space plane (b,a) along which all points (previously identified) are modulus maxima.

4. Singularities are detected by finding the abscissa where the wavelet modulus maxima converges at fine scale

   a. glitch signature can be characterized as a Lipschitz function (HÃ¶lder). HÃ¶lder exponent is evaluated thanks to the Mallat inequalities:

      log2[|Ws(b,a)|] <= log2 (A) + Î± log2(a) for a -> 0 ; Î± is the local HÃ¶lder exponent ; Ws(b,a) is the wavelet coefficient of the signal s at the scale a and the time b along the maxima line

   b. an irregularity of the signal can produce a cone in scale/time referential, so the coefficient correlation C is calculated on the set of points (log2(|Ws(bi,ai)|),log2 (ai))

   c. when C is greater than our corrThreshold, the linear regression is performed between minScale and maxScale and the slope of the linear regression can give a holder exponent

   d. if the holder exponent found is between hmin and holderThreshold, a singularity/detection has been found, and we know the contribution of the cone on the signal through wavelet coefficients

5. false detections provided by the noise are identified and removed by sigma clipping algorithm applied to the wavelet coefficients

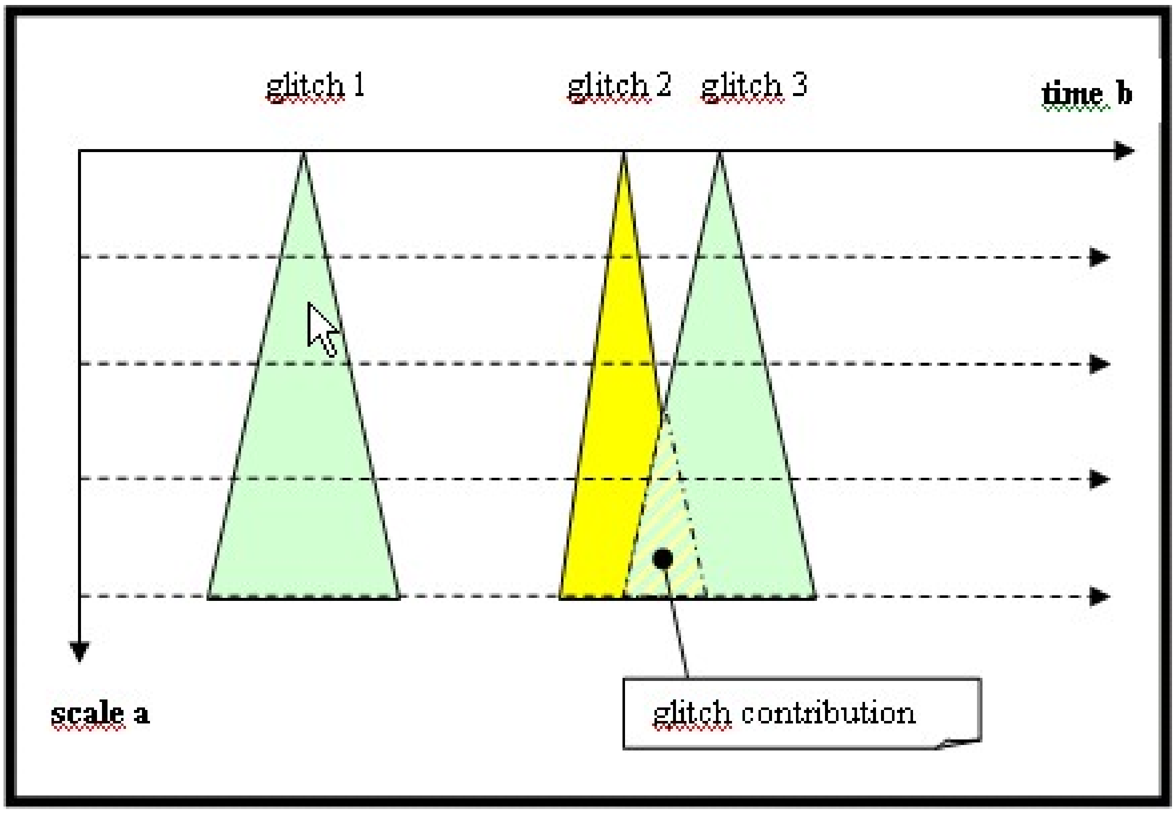
   a. white noise is a stationary process having the same spectral density whatever the frequencies. Thanks to Donoho, one can compute from the lowest scale (a=1) and wavelet coefficients found at this scale, an estimator of the noise variance :

$$\sigma^2 = \left(med\left(\frac{| W_s(b, a=1) |}{0.6745} \right) \right)^2$$

b. From the detections found by the maxima line analysis, one can considered white noise contribution when the wavelet coefficient are lower than 3Ïƒ.

$$|W_s(b,a)| < 3\sigma$$

6. Glitch wavelet coefficient contributions are calculated and removed from the signal

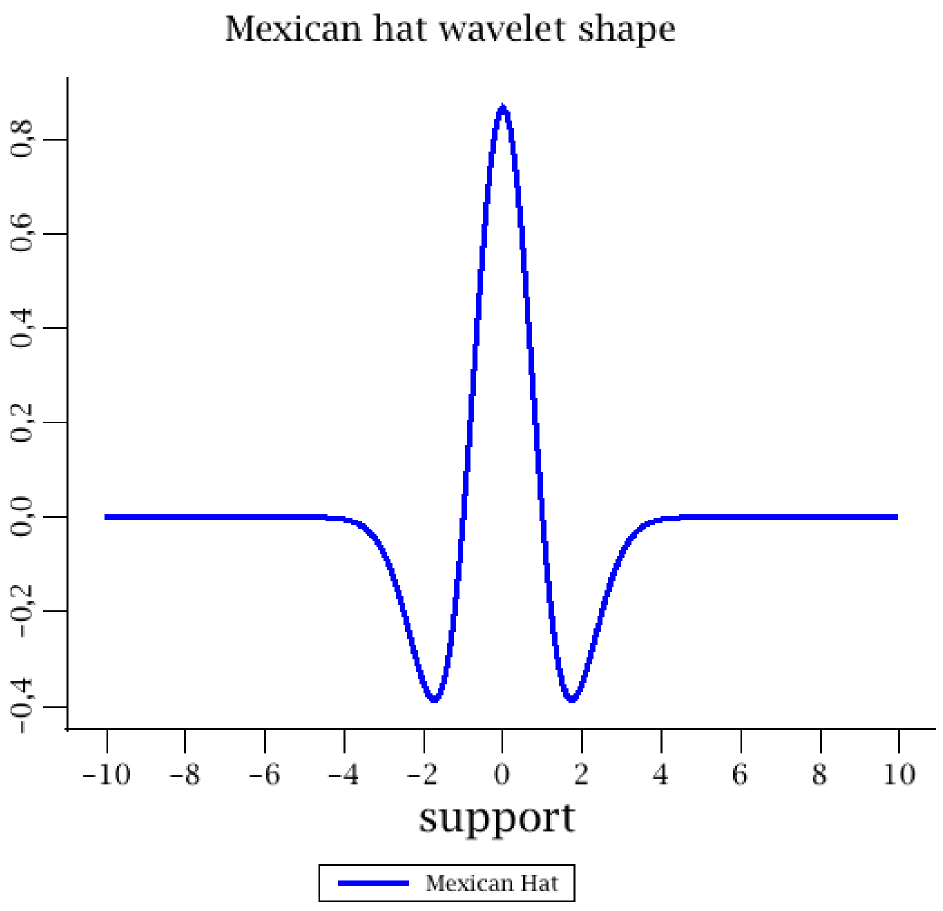


7. Signal is rebuilt with the following synthesis equation

Wgl : glitch contribution, Ws : wavelet coefficients of the signal

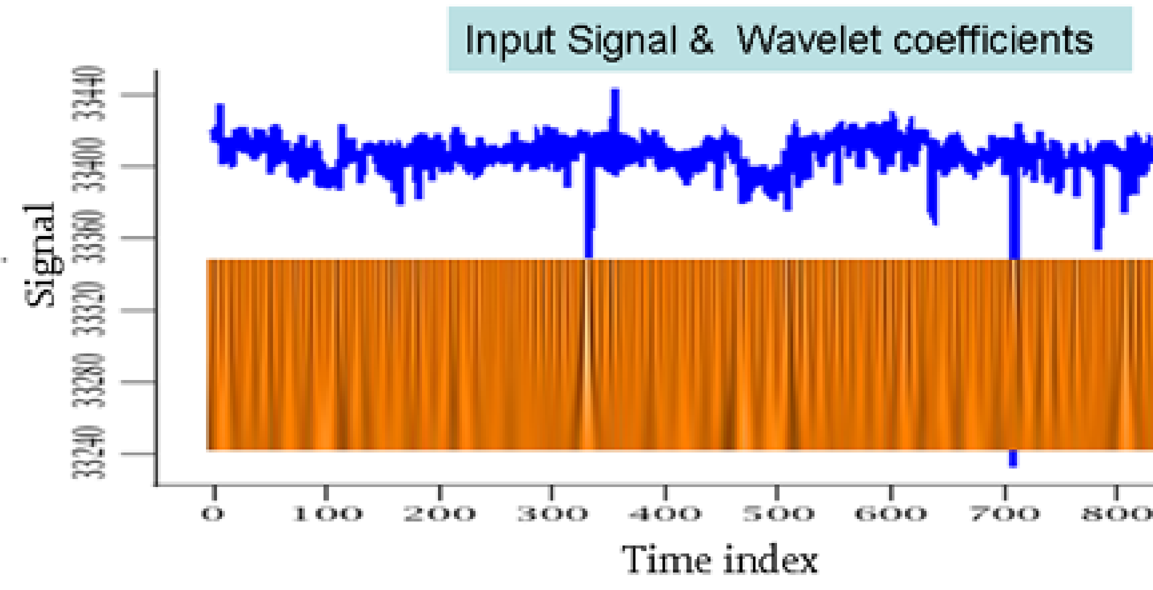
$$s(t) = \frac{1}{c'_{\psi}} \sum_i \left[W_s(t, a_i) - W_{gl}(t, a_i) \right] \frac{da_i}{a_i}$$

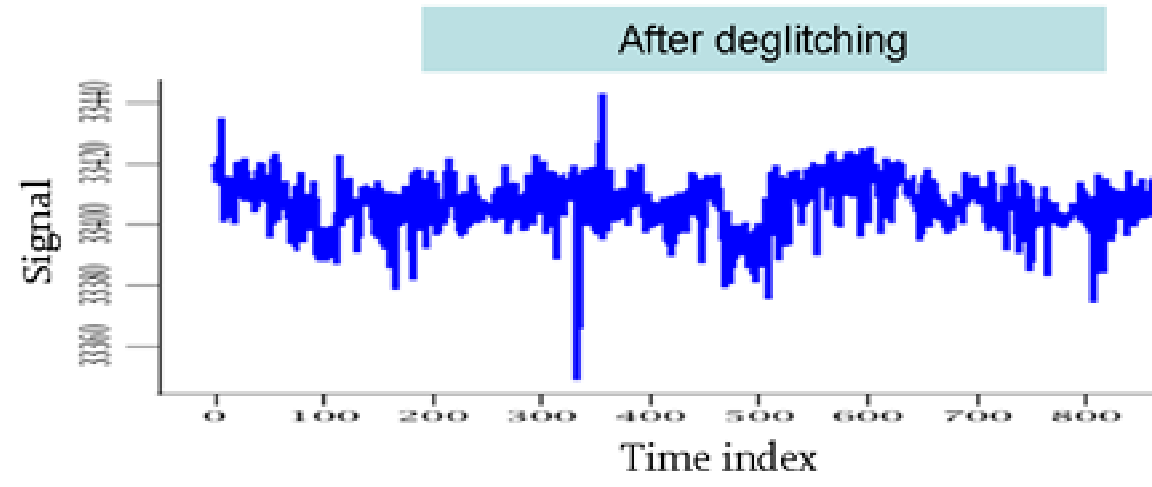
Wavelet used



Results

The following figure gives an overview of the deglitching process. wtmmlDeglitching task has been run with the default parameter. At the top, one can see the input signal extracted from the real data got during the alpha irradiation tests. In the middle (in orange), the wavelet coefficients according to the scale are plotted. At each signal variation, one can see a cone. This cone is analyzed, glitches are identified, removed, then the signal is rebuilt. At the bottom, the signal has been deglitched. On the right, the glitch located at position 710 has been removed, while the glitch at position 332 is still there. Beware that y axis at the top and at the bottom, are not identical.





Conclusion

The tuning of the parameters of this software is tricky. One's want to remove efficiently the glitches encountered, one can run several times the WTMML algorithm with various parameters.

Application domain

If the deglitching of the data delivered by the MMT method is considered as insufficient, one can use wtmmlDeglitching task with various parameters. This task will be useful for a limited usage, ran individually according to the glitch signature.

Test harness

Test harness carries out a lot of tests issued from real data obtained during Tandem irradiation tests and from data built with mathematical model.

Test 1 : from real data - remove alpha glitch signature

Test 2 : from real data - remove proton glitch signature

Test 3 : data are built with positive glitch effect (the detector wall has received the particle)

Test 4 : data are built with negative glitch effect (the detector grid has received the particle)

Test 5 : data are built with modulated signal, a gaussian noise gsigma higher than the signal and ng negative glitch effects , gaussian noise

Data location of the irradiation tests : [CVS]pacs/develop/data/pacs/data/glitchmodels

Status : is under development. Author : b.morin

```
>> success = wtmmlDeglitchingTest([signalLevel=1e-12][,signalShape=0]
[,noise="gaussian"][,noiseLevel=5.][,glitchNumber=3][,glitchLevel=8.]
[,glitchShape="lipschitz"))


success -: boolean: {true|false} = {successful tests|tests have failed}

Optional parameters are only for Test5 -:

signalLevel -: whatever value

signalShape -: {0,4} = {0:

noise -: {"gaussian","white"}

noiseLevel -: sigma value from the signal

glitchNumber -: number of glitch generated

glitchLevel -: sigma value from the signal (must be higher than the noiseLevel)

glitchShape -: {"lipschitz","amortizedSine"}
```

Data location : [CVS]pacs/develop/data/pacs/data/glitchmodels

3.7.9. addUTC

Convert from spacecraft on-board time (OBT) to UTC, using the time correlation table. Fill the UTC field in the frames dataset.

```
>> outFrames = addUTC(inFrames frames, timecor, [,copy=copy]  [,calVersion =
calVersion])

outFrames  -: Frames       -: Frames out
inFrames   -: Frames       -: Frames in
timecor    -: TableDataset -: Time corrections
copy       -: int          -:
                           0 -- return reference
                           1 -- return copy
calVersion -: String -: Version of the calibration files used
```

Calibration File: Time correlation table

3.7.10. convChopper2Angle (jython prototype available)

This task converts the Chopper position expressed in technical units to angles. This is done by reading the CPR entry in the Status table and express it in two ways: a) as angle with respect to the FPU (CHOPFPUANGLE entry in the Status table) and b) as angle in the sky (CHOPSKYANGLE). Both angle are in arcseconds. In particular, the CHOPFPUANGLE is a mandatory input for the PhotAssignaRaDec task, to be executed after Level 0.5 for the final step of the astrometric calibration. Thus, the convChopper2Angle task must be executed even if the chopper is not used at all as in the scan map (chopper maintained at the optical zero). CHOPFPUANGLE corresponds to the chopper throw in arcseconds in HSpot.

```
>> outFrames = convChopper2Angle(inFrames [,copy=copy]  [, calVersion = calVersion])

outFrames  -: Frames -: Frames out
inFrames   -: Frames -: Frames in
copy       -: int    -:
                     0 -- return reference
                     1 -- return copy
calVersion -: String -: Version of the calibration files used
```

The calibration between chopper position in technical units (voltages) and angles is give by a 6th oder polynomial. The calibration is based on the calibration file containing the Zeiss conversion table.

Reference: "Angular Calibration and zero-point offset determination of PACS FS Chopper for cold HeII (T=4.2 K) conditions.", PICC-MA-TR-009, U. Klaas, J. Screiber, M. Nielbock, H. Dannerbauer, J. Bouwman.

3.7.11. convXYStage2Pointing (available)

During the so-called PACS ILT tests in the lab, there was no info about satellite pointing information. So this step is used to simulate pointing information for this particular test-case. For real PACS Herschel data the next task, "photAddInstantPointing", should be used instead. The coordinates of the used point source, called XY stage, are included in the Status table and used later as input for a simulated astrometric calibration (photAssignRaDec).

```
>> outFrames = convXYStage2Pointing(inFrames, seq [, noInter=noInter] [,copy=copy])

outFrames -: Frames         -: Frames out
inFrames  -: Frames         -: Frames in
seq       -: PacketSequence -: PacketSequence holding the TmPackets of the period
of Frames
noInter   -: boolean        -: True  -: without Interpolation
                             False -: with Interpolation (default -!)
copy      -: int    -:
                     0 -- return reference
                     1 -- return copy
```

The coordinates of the XY stage are contained in the XY HK. This info is extracted from there and the internal time is used to merge the coordinates to the individual frames.

The HK packets have a readout frequency lower than the frames readout. So the task by default (noInter = false) interpolates between the available XY stage coordinates to obtain coordinates for each frame. With the keyword "noInter=true" no interpolation is done.

New entries in the Frames Status :

XY_Stage_EvType : Event Type (regular, start , stop)

XY_Stage_Mode : Mode (idle ,local single, local raster, single position move, single raster)

XY_Stage_TimeSec : Time seconds

XY_Stage_TimemS : Time miliSeconds

XY_Stage_LV_Sts :

XY_Stage_Status : XY Stage Status

XY_Stage_X_Axis : X axis position

XY_Stage_Y_Axis : Y Axis position

XY_Stage_X_idx

XY_Stage_Y_idx

XY_Stage_Stage_Nod_cnt : Nodding count

XY_Stage_Nod_pos : Nodding position (on raster , off raster)

XY_Stage_column : Column count

XY_Stage_line : Line count

This task allows also to include info about the nod cycle by adding a nod position counter (entry XY_Stage_Stage_Nod_cnt) and the a nod on or off position identifier (entry XY_Stage_Nod_pos).

3.7.12. photAddInstantPointing

The purpose of this task is to perform the first step of the astrometric calibration by adding the sky coordinates of the virtual aperture (center of the bolometer) and the position angle to each readout as entry in the status table. In addition the task associates to each readout raster point counter and nod counter for chopped observations and sky line scan counter for scan map observations.

```
outFrames = photAddInstantPointing(inFrames, scPointing, [,calTree=<mycalTree>]
[,copy=<number>]
          [,siam = siam][,orbitEphem=orbitEphem] [,horizons=horizons][,isSso =
isSso]
          [,noInter=noInter][,useGyro = useGyro])
```

This first part of the astrometric calibration deals with two elements: the satellite pointing product and the SIAM product. Both are auxiliary products of the observation and are contained in the Observation context delivered to the user. The satellite pointing product gives info about the Herschel pointing. The SIAM product contains the a matrix which provides the position of the PACS bolometer virtual aperture with respect to the spacecraft pointing. In the current version of the pipeline this task used a SIAM matrix contained in a calibration file and not the one of the SIAM product. However, this will be changed in the future and the SIAM product will be used for the astrometric calibration. The time is used to merge the pointing information to the individual frames. scPointing is the pointing product.

By default the Filtered Pointing information is used, but also the gyro propagated pointing information may be used. This is done by using the Frames status entry FINETIME and extract the associated information from the PointingProduct. Also the SIAM matrix is applied and aberration is done (if the proper Products are passed). The result is added to the status entry of the Frames Product.

Parameters: The orbit Ephemeris Product needed for aberration correction of non SSO objects. Horizons Product is needed for the aberration correction of SSO objects. `isSso` is "is it solar system" (default False/0)? `noIter` is whether to interpolate or not. `useGyro` (False/0) is to use the Gyro propagated information instead of the filtered.

isSso (default False/0) is a switch for whether the target is solar system.

The task adds the following entries to the status table:

- RaArray: ra coordinate of the virtual aperture (deg)

- DecArray: dec coordinate of the virtual aperture (deg)

- PaArray: position angle (deg)

- raArrayErr: ra coordinate inaccuracy of the virtual aperture (deg)

- decArrayErr: dec coordinate inaccuracy of the virtual aperture (deg)

- PaArrayErr: position angle inaccuracy(deg)

- Mode: PacsPhoto, in the bolometer case

- RasterLineNumber, for chooped observation only

- RasterColumnnumber, for chooped observation only

- NodcycleNum, for chopped observation only

- OnTarget, on source position identifier, for chooped observation only (flase or true)

- AbPosId, for chooped observation only (false or true)

- IsSlew, identifies satellite slewing (false or true)

- IsOffPos, identifies off position (false or true)

- ScanLineNumber, identifies the scan line number for scan map observation

- AcmcMode

- Aperture

- IsAposition, identifies A position in nod cycle

- IsBPosition, identifies B position in nod cycle

- IsOutOfField

- IsSerendipity

- RollArray, any difference with PAArray???

3.7.13. cleanPlateau (java prototype available)

This task is executed before Level 0.5 only for chopped observations (point-source, small-source, chopped raster modes).

```
>> outFrames = cleanPlateauFrames(Frames inFrames[,dmcHead=dmcHead][,copy=copy]
[,calVersion = calVersion])

outFrames  -: Frames -: Frames out with mask UNCLEANCHOP
inFrames   -: Frames -: Frames in
copy       -: int    -:
                     0 -- return reference -: overwrites the input frames by
adding the additional -'DithPos' column
                     1 -- return copy      -: creates a new output without
overwriting  the input
calVersion -: String -: Version of the calibration files used
```

The module flags the readouts at the beginning of a chopper plateau, if they correspond to the transition between two chopper positions. In the chopper transition phase, the chopper is still moving towards to proper position and the signal of this readouts does not correspond to the on or off position. Usually the chopper is moving so fast that only one readout needs to be masked out. The module just adds the 3D UNCLEANCHOP mask to the input frame.

The task identifies the chopper plateaus on the basis of the CHOPPERPLATEAU (for the science data) and CALSOURCE (for the calibration block) entries in the status table. For each chopper plateau the readouts with a chopper position deviating from the mean position (threshold provided by the calibration file ChopJitterThreshold) are flagged in the UNCLEANCHOP mask.

3.8. The AOT dependent pipelines

After level 0.5, the pipeline is AOT dependent. In the following sections we will describe separately the different AOT pipelines, point source, small source, chopped raster, scan map AOTs, up to Level 2.

3.9. Point Source AOR

3.9.1. Level 0.5 to Level 1

3.9.1.1. photMakeDithPos (jython prototype available)

The task just checks if exists a dithering pattern and identifies the dither positions. The task adds a dither position counter, "DithPos", to the Status table. Frames with the same value of 'DithPos' are at the same dither position.

```
>> outFrames = photMakeDithPos(inFrames [,copy=copy] -)

outFrames  -: Frames -: Frames out with one image per every single chopper plateau
inFrames   -: Frames -: Frames in
copy       -: int    -: This has to be done by
                     0 -- return reference -: overwrites the input frames
                     1 -- return copy      -: creates a new output without
overwriting  the input
```

3.9.1.2. photMakeRasPosCount (jython prototype available)

The task adds raster position counter to status table.

```
>> outFrames = photMakeRasPosCount(inFrames [,copy=copy])

outFrames  -: Frames -: Frames out with one image per every single chopper plateau
inFrames   -: Frames -: Frames in
copy       -: int    -: This has to be done by
                     0 -- return reference -: overwrites the input frames
                     1 -- return copy      -: creates a new output without
overwriting  the input
```

The task needs the output of the photAddInstantPointing task to be executed otherwise an error is raised saying that the pointing information are missing for the observation. The module uses the virtual aperture coordinates and the raster flags in the status table to identify different raster positions. The raster positions are identified in the Status table by the new entries 'OnRasterPosCount' and 'OffRasterPosCount'.

3.9.1.3. photAvgPlateau (java prototype available)

The task averages all valid signals on chopper plateau and resamples signals, status and mask words for the photometer. It calculate noise map but not the coverage map. The result is a Frames class with one image per every single chopper plateau.

```
>> outFrames = photAvgPlateau(inFrames [,sigclip=0] [,mean=0]
[,qualityContext=QualitxContext]  [,copy=0] -)

outFrames       -: Frames -: Frames out with one image per every single chopper
plateau
inFrames        -: Frames -: Frames in
sigclip         -: Value for sigma clipping (default = 0 -: no sigma clipping)
mean            -: mean = 1 -: use MEAN instead of AVERAGE (default mean = 0 -:
use AVERAGE)
qualityContext  -: QualityContext
copy            -: int    -: This has to be done by
                     0 -- return reference -: overwrites the input frames (default)
                     1 -- return copy      -: creates a new output without
overwriting the input
```

The module uses the status entry CHOPPERPLATEAU (CALSOURCE in case of calibration block pre-processing) to identify the chopper plateau in the same way as CleanPlateau. Then it computes the average (sigma clipping if sigclip > 0, and median if mean =1) for each pixel over the chopper plateau .

Figure 3.7. Simplified Example : Chopper Plateaus

The signal of the bad pixels, identified by the BADPIXEL mask, is set to 0. The pixels flagged in the other available masks (SATURATION, GLITCH, UNCLEANCHOP) are discarded in the average. If the chopper plateau contains no valid data (all pixels masked out) the signal is set to zero. The noise is calculated for each pixel (x,y) and each plateau (p) as:

noise[x,y,p] = STDDEV(signal[x,y,validSelection[p]]) / SQRT(nn)

where nn is the number of valid readouts in the chopper plateau. This number is then stored as addition entry (NrChopperPlateau)in the status table. The noise is stored in the Noisemap

The Status entries with different values over the chopper plateau length are modified with the following scheme:

- OBSID: value of the beginning of the chopper plateau
- BBID: value of the beginning of the chopper plateau
- LBL : removed
- TMP1: removed
- TMP2: removed
- FINETIME: value of the beginning of the chopper plateau
- VLD : removed
- WPR: value of the beginning of the chopper plateau
- BOLST: removed
- BSID : removed
- CRDC: value of the beginning of the chopper plateau
- CRDCCP: value of the beginning of the chopper plateau
- DBID: value of the beginning of the chopper plateau

- DMCSEQACTIVE: value of the beginning of the chopper plateau
- CHOPPERPLATEAU : Sum
- CALSOURCE : Sum
- PIX: removed
- RCX: removed
- RESETCNT: Just counting 1 to x
- BLOCKIDX: removed
- BAND: value of the beginning of the chopper plateau
- BBTYPE: value of the beginning of the chopper plateau
- BBSEQCNT: value of the beginning of the chopper plateau
- UnCleanChop: Sum
- DithPos : Median
- OnRasterCount : Median
- OffRasterCount : Median

NOTE: the masks are still not properly treated in the pipeline. This task (and the following as well) is somehow reducing the masks as the images. So the information carried by the individual mask is not propagated into the pipeline. At this point of the data reduction the masks should be combined in a master mask. photAvgPlateau should use the master mask to create the exposure (weight) map of any chopper plateau: that means, if a plateau has 10 frames and a pixel is 3 times flagged in the master mask(e.g. it is twice saturated and once hit by a glitch), its weight will be 0.7 instead of 1, if a pixel is a bad pixel, its weight is zero. The Product of photAvgPlateau has to contain the averaged plateau images with exposure and noise maps without any mask. The information carried sofar by the masks should now be condensed and transfered into the exposure map. Developers still working on that.

3.9.1.4. photDiffChop (java prototype available)

Subtract every off-source signal from every consecutive on-source signal. The result is a Frames class with one image per one chopper cycle.

```
>> outFrames = photDiffChop(inFrames -,hkdata=hkdata
[,qualityContext=QualityContext] [,copy=0] -)

outFrames         -: Frames -: Frames out with one image per one chopper cycle
inFrames          -: Frames -: Frames in
hkdata            -: TableDataset -: issued from HPPHK product (Herschel PACS
Photometer HK)
qualityContext    -: QualityContext
copy              -: int    -:
                       0 -- return reference -: overwrites the input frames
(default)
                       1 -- return copy      -: creates a new output without
overwriting the input
```

To better subtract the telescope background emission and the sky background the 'off-source' image is subtracted from the 'on-source' image (consecutive chopper positions). The module accepts as input the output of photAvgPlateau module. It returns as output a Frames class with the differential image of any couple of on-off chopped images. The module resamples the status table and the the masks accordingly (see NOTE in photAvgPlateau section).

The on and off images are identified on the basis of the status entries added by the photAddInstant-Pointing task. The noisemap is computed in the following way:

noise [x,y,k] = SQRT(noise[x,y,pON]**2 + noise[x,y,pOFF]*+2)

where k is the frame number of the differential on-off image, pOn is the frame number of the on source image, pOFF is the frame number of the off source image, and noise[x,y,pON] and noise[x,y,pOFF] are the error map at the on and off source images, respectively (output of the previous pipeline step).

Figure 3.8. Simplified Example : Chopper Plateaus

3.9.1.5. photAvgDith (jython prototype available)

The chop cycle is repeated several times per any A and B nod position. This task calculates the mean of the on-off differential chopped images per any A and B position within any Nod cycle. If the dithering is applied in the point-source mode as offered by HSpot, the average is done separately per dithered A and B nod positions.

```
>> outFrames = photAvgDith(inFrames [,qualityContext=QualityContext] [,copy=0] -)

outFrames      -: Frames -: Frames out with one image per chopper plateau per
nodding position
inFrames       -: Frames -: Frames in
qualityContext -: QualityContext
copy           -: int    -:
                       0 -- return reference -: overwrites the input frames
(default)
                       1 -- return copy      -: creates a new output without
overwriting the input
```

The task uses the entries in the status table regarding the dithering pattern (DithPos) and the ones regarding the A and B nod position and Nod cycle identifier (see entries added after AddInstantPoint-ing) to identified all the differential on-off images belonging to the A or B position with the same nod cycle and at the same dither position. Since only the average of the identified images is performed, the noise is propagated as follows:

For "c" chopper cycles (c=k), we average the n/2 differences

noise [x,y] = SQRT(MEAN(noise[x,y,:]**2)) / SQRT(n)

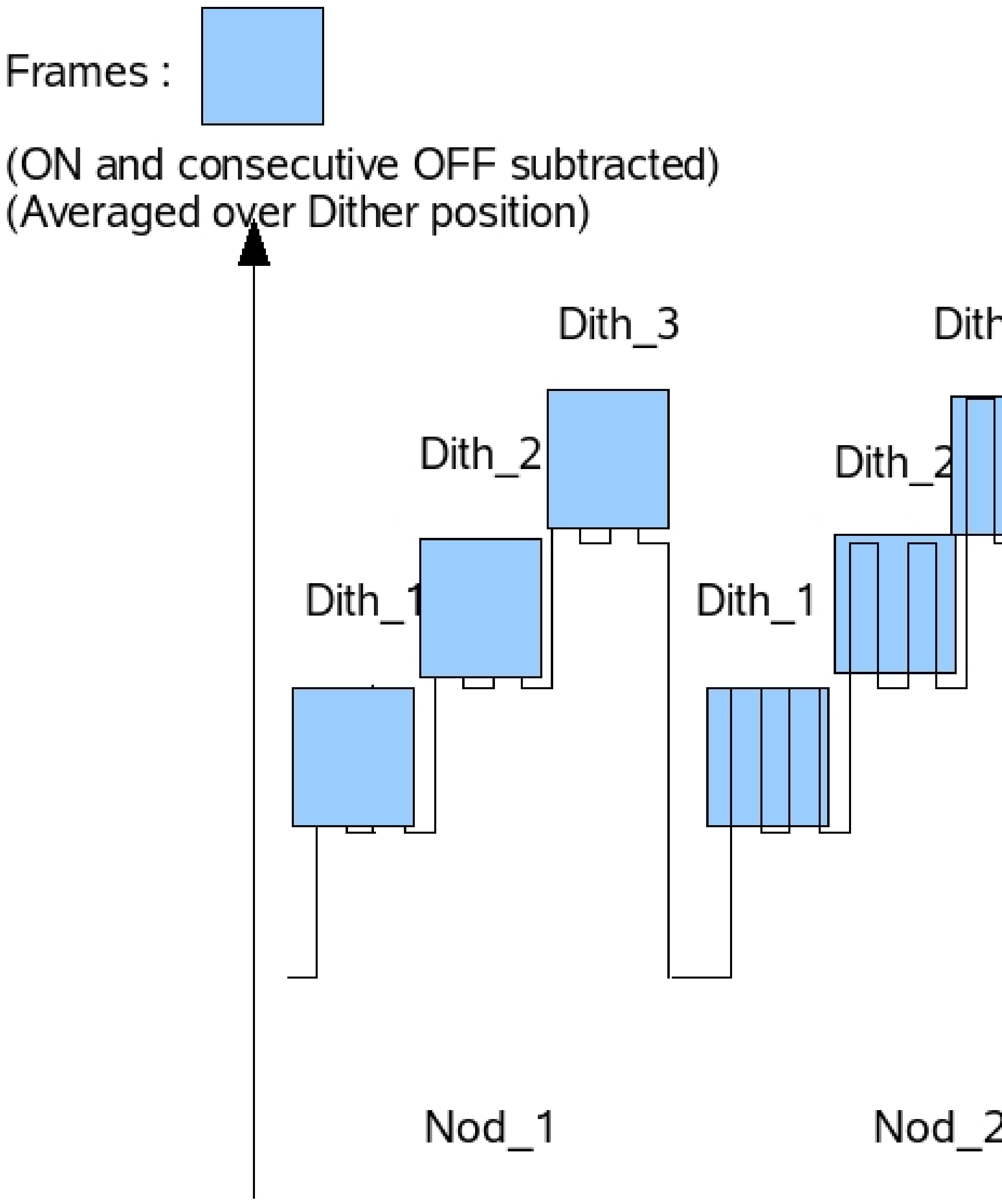

Frames :
(ON and consecutive OFF subtracted)
(Averaged over Dither position)
Dith_3
Dith
Dith_2
Dith_2
Dith_1
Dith_1
Nod_1
Nod_2

Figure 3.9. Simplified Example : Chopper Plateaus

3.9.1.6. photDiffNod (java prototype available)

This task is performing the last step of the background (sky+telescope) subtraction. It subtracts the images corresponding to the A and B positions of each nod cycle and per each dither position. The module needs as input the output of photAvgDith.

```
>> outFrames = photDiffNod(inFrames [qualityContext=QualityContext] [,copy=0] -)

outFrames      -: Frames  -: Frames out with one image per nod cycle
inFrames       -: Frames  -: Frames in
qualityContext -: QualityContext
copy           -: int     -:
                       0 -- return reference -: overwrites the input frames
(default)
                       1 -- return copy      -: creates a new output without
overwriting the input
```

The noise is propagated as follows:

noise [x,y,k] = SQRT(noise[x,y,A]**2 + noise[x,y,B]*+2)

where the A and B indexes refer to the A and B nod position.

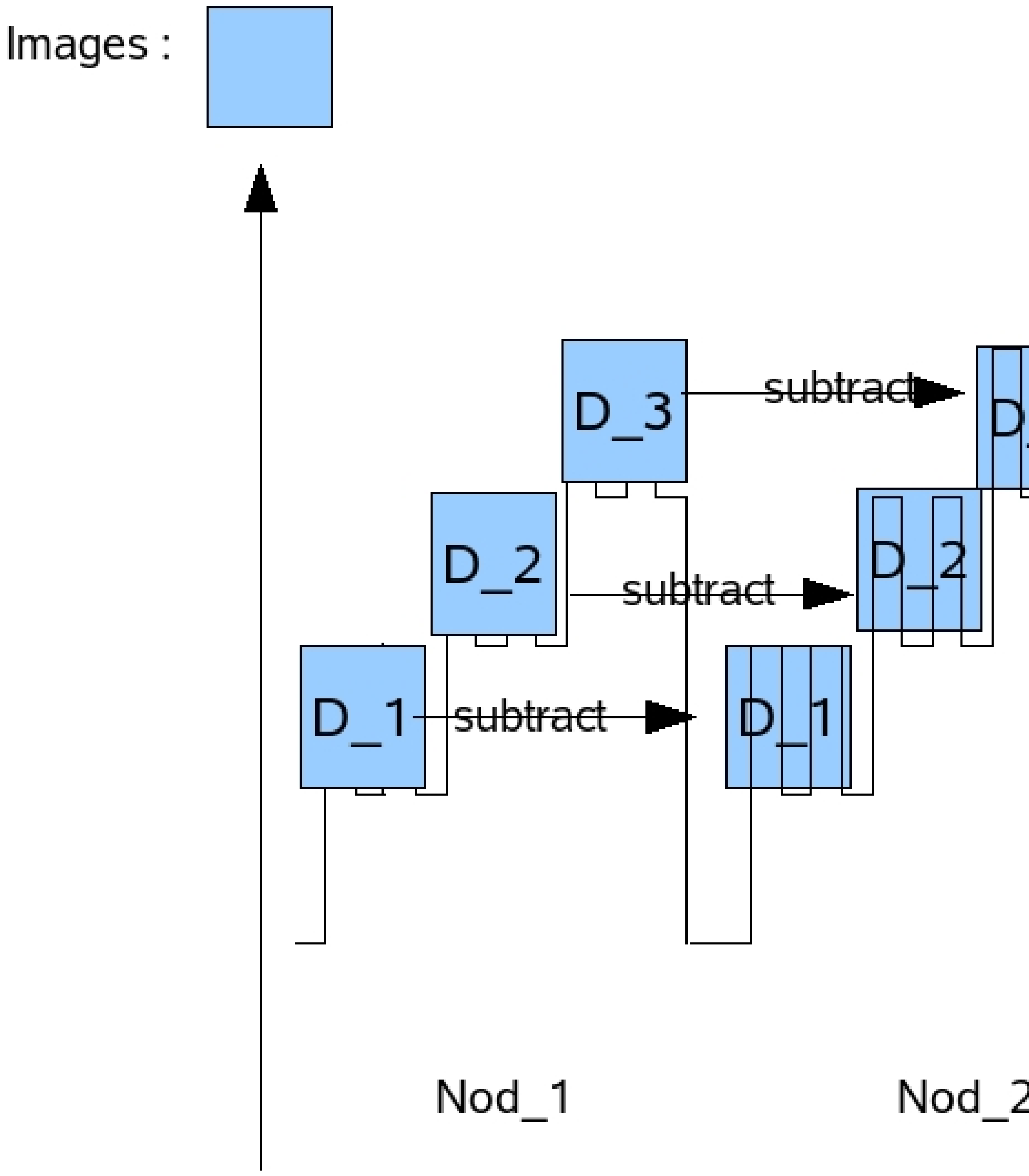


Figure 3.10. Simplified Example : Chopper Plateaus

3.9.1.7. photCombineNod (java prototype available)

The nod cycles are repeated many times per any dither position. This task is taking the average of the differential noda-nob images corresponding to any dither position. The results is a frames class containing a completely background subtracted point source image per any dither position.

```
>> outFrames = photCombineNod(inFrames [qualityContext=QualityContext] [copy=0] -)

outFrames      -: Frames  -: Frames out with one image
inFrames       -: Frames  -: Frames in
qualityContext -: QualityContext
copy           -: int    -:
                     0 -- return reference -: overwrites the input frames (default)
                     1 -- return copy
```

The noise is propagated as follows:

noise[x,y,d] = STDDEV(signal[x,y,nd]) / SQRT(nd)

where d is the index of the dither position and nd is the number nod cycles per dither position.

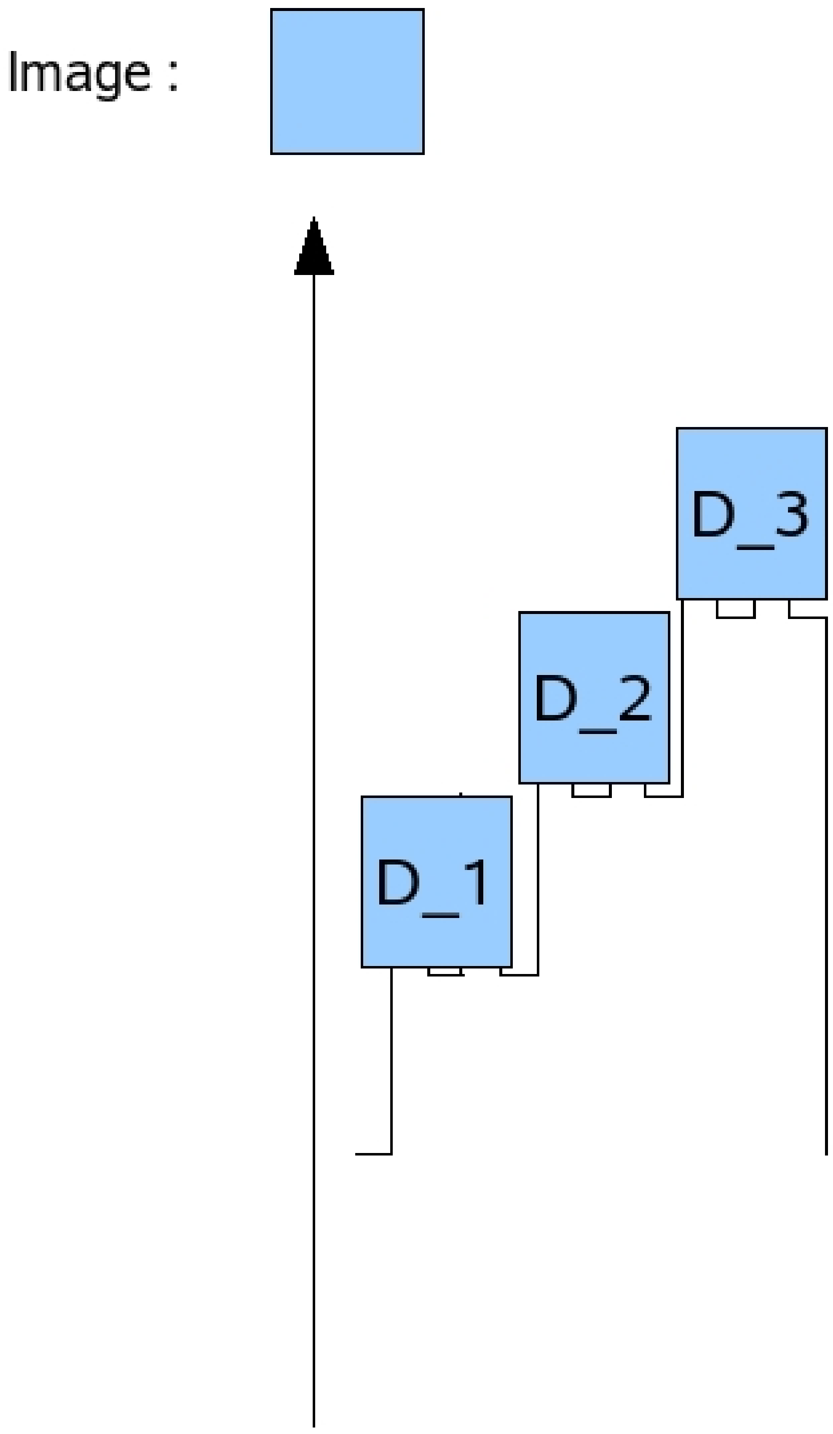


Figure 3.11. Simplified Example : Chopper Plateaus

3.9.2. Level 1 to Level 2

3.9.2.1. photDriftCorrection

Applies drift correction of the flat field and controls the photometric stability

```
>> outFrames = photDriftCorrection(inFrames [,calTree=calTree][,dCSsRef=dCSsRef]
[threshold=threshold][algo=algo][,copy=copy] -)

outFrames    -: Frames              -: Frames out in Jansky photometricaly
calibrated
inFrames     -: Frames              -: Frames with signal in Volt
calTree      -: PacsCalibrationTree -: calibration tree containing all calibration
products used by the pipeline
dCSsRef      -: DCSRef              -: Î´C0 reference point is the difference of
the internal calibration source computed during the determination of the flat field
threshold    -: PhotometricStabilityThreshold
algo         -: String              -: algorithm used when many calibration blocks
exists {"first","last","mean" [default] -,"median","interpolation"}
                                   -"first"  -- only the first calibration block
is applied on the science data block
                                   -"last"   -- only the last calibration block is
applied on the science data block
                                   -"mean"   -- the mean of calibration block is
applied on the science data block
                                   -"median" -- the median of the calibration
block is applied on the science data block
                                   -"interpolation" -- an
interpolation is done between the calibration block and applied at each
map.
copy         -: int                 -:
                                   0 -- return reference
                                   1 -- return copy
```

Literature

Photometric calibration of PACS bolometer - K.Okumura, D.Lutz, M.Sauvage and B.Morin - October 11,2007

Photometric calibration products - B.Morin, K.Okumura, D.Lutz, M.Sauvage - February 08,2008

Principle

PhotDriftCorrectionTask has the goal to multiply signal s(t) in Volt by the ratio Î´C0/Î´Cs. This factor corrects possible drift of the flat field Î¦ or CS emission. Reference point Î´C0 is computed during the determination of the flat field. Î´Cs coming with the frame gives an evaluation of the possible drift of the flat. This evolution can be either an alteration of the internal calibration sources or an evolution of the detector pixels. The drift is compared with photometric stability threshold parameters (stored in the calibration files). If the ratio overtakes these thresholds, a 'DriftAlert' keyword is added to the metadata . For each pixel having a ratio in error, the ratio is collected, the average done and stored in the metadata. Collected at the end of the pipeline, a specific action will be triggered with it (such as mail and so forth).

Hereafter the formula managing the photometric adjustement :

$$\delta f(t) = \delta s(t) * \frac{(\delta C_0)}{(\delta C)} * \frac{1}{(J\Phi)}$$

Î´f(t) is the flux in Jy

Î´s(t) is the signal in Volt

Î´C_0 is the difference of the calibration sources got during a calibration campaign. Computed during the flat field determination the unit is in Volt

Î´C is the difference of the calibration sources computed by the pipeline. Unit is in Volt

J is a flux calibration factor which contains the responsivity and the conversion factor to Jansky

Î¦ is the normalized flatfield - dimensionless

Ratio 1/JÎ¦ converts the signal s(t) in Volt to f(t) in Jansky

Ratio Î´C_0/Î´C corrects the drift of the ratio 1/JÎ¦

Quick chopping between calibration source will remove one part of the offset. CS1 and CS2 measurements are always done at the same chopper position (-21350 for CS1 and +21200 for CS2). The using of the same chopper reference, the ratio Î´C_0/Î´C is a good way to free of the pixel distortions.

Hereafter the formula used to compute the noise

*noise = SQRT(s_{out}^2 * [(ÏƒÎ´s^2/Î´s^2) + (ÏƒÎ´C_0^2/Î´C_0^2) + (ÏƒÎ´C_s^2/Î´C_s^2)])

```
where
Î´s is the input signal in Volt, ÏƒÎ´s is the input noise (frame.getNoise() -)
Î´C0 is our reference in Volt see above and ÏƒÎ´C0 is the noise of our reference
Î´Cs computed by photDriftCorrection contains the drift in Volt while ÏƒÎ´Cs is
the associated noise
sout is the output signal in Volt, modified by PhotDriftCorrection
noise is the new noise in Volt available with the command -: frame.getNoise()
```

Calibrations files : DCRef (Î´C0 in the formula) and PhotometricStabilityThreshold.

Addendum: the first Î´C_0 has been determined with data collected during ILT test campaign. The following biases have been used: 2.6 V for both the blue and green channel, 2.0 V for the red one. These bias are not used during flight operation so we pay attention for using PhotDriftCorrection. This latter normaly corrects the drift, but due to bias divergence between dCo and dcs, one part of the response is included in the ratio applied. For the photometry we should apply PhotDriftCorrection. Nevertheless, **PhotDriftCorrection shouldn't use when the reponsivity calibration product version 4.0 is applied.**

3.9.2.2. photRespFlatFieldCorrection

Applies flat field corrections and converts signal to a flux density

```

>> outFrames = photRespFlatFieldCorrection(inFrames [,calTree=calTree]
[,flatField=flatField][,responsivity=responsivity][,copy=copy] -)

outFrames    -: Frames              -: Frames out in Jansky photometricaly
calibrated
inFrames     -: Frames              -: Frames with signal in Volt
calTree      -: PacsCalibrationTree -: calibration tree containing all calibration
products used by the pipeline
flatField    -: FlatField           -: FlatField calibration product
responsivity -: Responsivity        -: Calibration product converting the signal
in Flux density (Jansky)
copy         -: int                 -:
                                   0 -- return reference
```

```
                  1 -- return copy
```

Literature

Photometric calibration of PACS bolometer - K.Okumura, D.Lutz, M.Sauvage and B.Morin - October 11,2007

Photometric calibration products - B.Morin,K.Okumura,D.Lutz, M.Sauvage - February 08,2008

Principle

photRespFlatFieldCorrection divides Î´s(t) by the flat field Î¦ and the responsivity J.

Hereafter the formula managing the photometric adjustement :

$$\mathcal{S}f(t) = \mathcal{S}(t) * \frac{(\delta C_0)}{(\delta C)} * \frac{1}{(J\Phi)}$$

(Please have a look also on the PhotDriftCorrectionTask)

1/JÎ¦ converts signal in Volt to jansky. The conversion is done for a Î»0 wavelength related to the central filter band used. Bandwith and Î»0 information is stored in the Responsivity calibration product available from the calibration Tree (calTree).

Calibration Files: FlatField and Responsivity

FlatField calibration product

This product contains the flat field and the noise for the red, the green and the blue channel. Here is briefly a description of the product :

- An image of the flat field - the physical unit is dimensionless and around 1
- an image of the noise - the physical unit is dimensionless - Noise has to be multiply by the responsivity in order to have a physical dimension
- The average of the fluxes used to compute the flat in Jansky
- The difference of the fluxes used to compute the flat in Jansky
- The creation date of the flat in microsecond since the 1 jan 1958
- Text field given the coordinates of the calibration source used to make the flat or during the ILT the OGSEs with their temperatures in Kelvin and filter.

Useful commands to explore this calibration product

```
from herschel.pacs.cal import *
caltree = getCalTree("FM")
flat = caltree.photometer.flatField
channel = -"red" # or -"green" or -"blue"
print flat[channel].getMeta().get("MeanFlux")
print flat[channel].getMeta().get("DeltaFlux")
print flat[channel].getMeta().get("Summary")
print flat[channel].getMeta().get("CreationDate")
# or simply
```

```
print flat[channel].getMeta()
Display(flat[channel]["FlatField"].data)
Display(flat[channel]["NoiseMap"].data)
or
resp = caltree.photometer.responsivity
Display(flat[channel]["NoiseMap"].data.multiply(resp[channel"]
[Responsivity].data[0]) # in V/Jy
```

Overview

Noise

$$\frac{\sigma^2 f}{f^2} = \frac{\sigma^2 s}{s^2} + \frac{\sigma^2 \delta C_0}{\delta C^2{}_0} + \frac{\sigma^2 \delta C}{\delta C^2} + \frac{\sigma^2 K}{K^2} + \frac{\sigma^2 r}{r^2} + \frac{\sigma^2 \phi}{\phi^2}$$

with Ïƒk = 0

Responsivity calibration product

This product contains the responsivity. Signal in volt multiply by the coefficient found in this product converts the signal in Volt into flux density in Jansky. Here is briefly the content of this product :

- responsivity coefficient V/Jy
- Reference wavelength Î»0 in micrometer used to compute the flux - in general the central wavelength of the filter
- Effective aperture in square meter - the diameter of the primary telescope miror
- Effective bandwidth in Hertz - see formula below

Effective bandwith :

$$\Delta\upsilon = M * \Delta'\nu * \int_{-\infty}^{+\infty} \frac{T(\lambda)}{\lambda} d\lambda$$

M = mirror's transmission
T(λ) is the filter transmission

Useful commands to explore this product

```
from herschel.pacs.cal import *
caltree = getCalTree("FM")
resp = caltree.photometer.responsivity
channel = -"red"    # or -"green" or -"blue"
print resp[channel].Responsivity.getMeta().get("RefWavelength")
print resp[channel].Responsivity.getMeta().get("EffectiveAperture")
print resp[channel].Responsivity.getMeta().get("EffectiveBandwidth")
# or simply
print resp[channel].Responsivity.getMeta()
print resp[channel].Responsivity.data[0] # in V/Jy
```

3.9.2.3. photShiftDith

The dithering pattern offered by HSpot is just a 1/3 pixel shift. Thus the coaddition of the 3 dithered double differential image is done only in pixel coordinates by this task. This is not the definitive result of the pipeline since also for the Point-source mode a final astrometric calibrated image should be provided. This is a work in progress and still under investigation.

3.10. Small Source AOR

Many of the tasks of this session are the same already described in the Point-Source pipeline. Thus, the description will not be repeated here. For those tasks the user can refer to the previous section.

3.10.1. Level 0.5 to Level 1

3.10.1.1. photMakeRasPosCount (jython prototype available)

See description of the same task in the Point-source pipeline

3.10.1.2. photAvgPlateau (java prototype available)

See description of the same task in the Point-source pipeline

3.10.1.3. photDiffChop (java prototype available)

See description of the same task in the Point-source pipeline

3.10.1.4. photAvgNod (jython prototype available)

See description of the same task in the Point-source pipeline

3.10.1.5. photDiffNod

See description of the same task in the Point-source pipeline

3.10.1.6. photDriftCorrection (java prototype available)

See description of the same task in the Point-source pipeline

3.10.1.7. photRespFlatFieldCorrection (java prototype available)

See description of the same task in the Point-source pipeline

3.10.2. Level 1 to Level 2

3.10.2.1. photAssignRaDec

This task performs the last step of the astrometric calibration. Sofar only the sky coordinates of the virtual aperture (center of the bolometer) and the position angle are available in the status table for each frame. the astrometric calibration is done by estimating the sky coordinates of the center of each pixel. This information is then stored into two cubes, one for RA and one for DEC, with the same dimensions of the frame class to be atrometrized.

```
>> outFrames = photAssignaRaDec(inFrames, calTree=calTree, [,copy=0]  -)

outFrames         -: Frames -: Frames out with one image per one chopper cycle
inFrames          -: Frames -: Frames in
calTree           -: PacsCalibrationTree -: calibration tree containing all
calibration products used by the pipeline
```

```
copy              -: int    -:
                       0 -- return reference -: overwrites the input frames
(default)
                       1 -- return copy      -: creates a new output without
overwriting the input
```

This step of the astrometric calibration is done in two steps. In the first step the subarray coordinate system, that is the the integer coordinates (p,q in the figure below) of the pixel centers as displayed in IA, have to be transformed into the the cartesian coordinate system of the PACS focal plane (u, v, respectively, in mm), which reproduces the real misalignment and rotation of the submatrices, as shown in the bottom figure below. The transformation coefficient between p,q to u,v coordinates are contained in the spatial calibration file PCalPhotometer_SubArrayArray_version.fits.

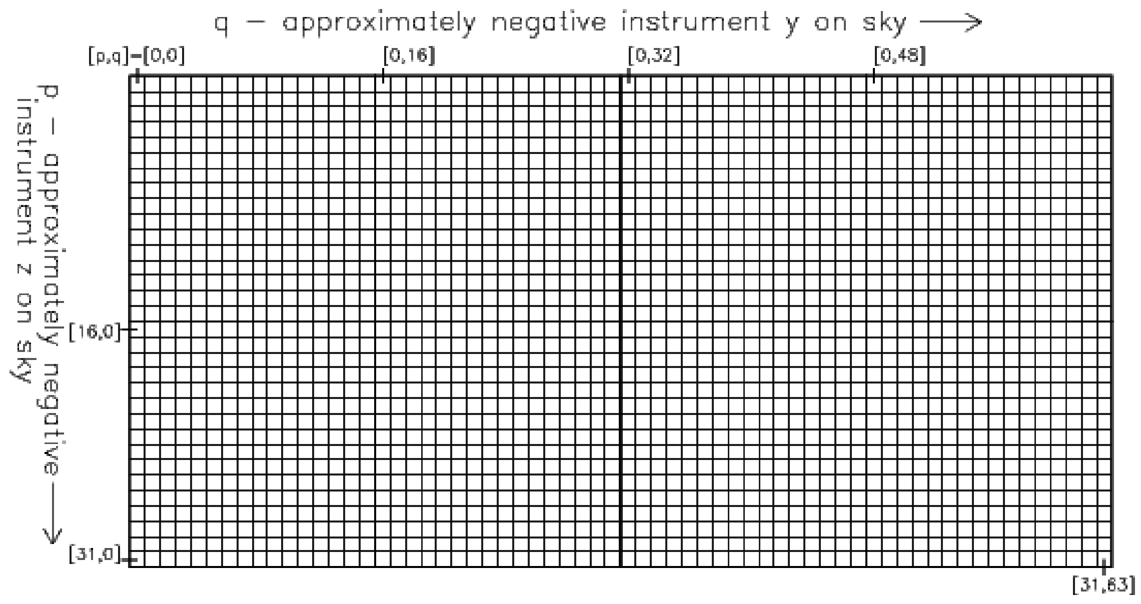

q — approximately negative instrument y on sky ⟶
[p,q]=[0,0]
[0,16]
[0,32]
[0,48]
p — approximately negative instrument z on sky ⟶
[16,0]
[31,0]
[31,63]

Figure 3.12.

As a second step the coordinates u,v on the PACS focal plane have to be transformed into orthogonal local coordinates on the tangential plane on the sky, y,z. The coordinates y,z, as shown in the figure below, correspond to the offset in arcsecond of the individual pixel coordinates with respect to the the virtual aperture. They are approximated by two polynomials in the three-dimensional space of u,v and chopper and alpha (CHOPFPUANGLE entry in the status table output of convertchopper2angle task):

$$y = \sum_{i=0}^{N} \sum_{j=0}^{M} \sum_{k=0}^{O} a_{ijk} u^i v^j \alpha^k$$

Figure 3.13.

the coefficients of the two polynomials are contained in the spatial calibration file PCalPhotometer_ArrayInstrument_version.fits. Once the Instrument coordinates are available, the sky coordinates of the center of each pixel are simply obtained by spherical trigonometric for any given RA and DEC of the virtual aperture and position angle PA (listed for each frames in the status table), as shown in the figure below.

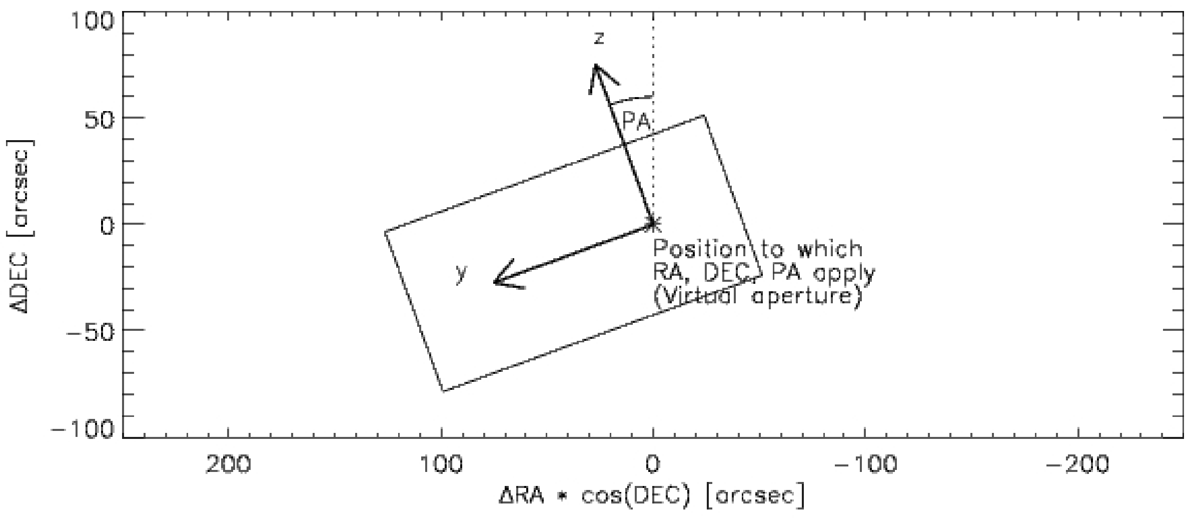


Figure 3.14.

3.10.2.2. photProject

The protProject task provides one of the two methods adopted for the map creation from a given set of images (in the PACS case, a frame class). The second method is MadMap which will be discussed in the ScanMap pipeline section.

```
>> si = photProject(inFrames, [outPixelSize=outPixelsize,] [copy=1,] [monitor=1]
[optimizeOrientation=optimizeOrientation]
                          [mapcoordinates=mapcoordinates] [calibration=calibration])

si              -: final map (SimpleImage) with WCS

inFrames        -: Frames  -: astrometric calibrated input frames

outPixelSize    -: double -:  the size of a pixel in the output dataset in
arcseconds.
                           Default is the same size as the input (6.4 arcsecs for
the red and 3.2 arcsecs for the blue photometer)
copy            -: default is 0 (no copy of inFrames). Option is 1, if inFrames
should be copied

optimizeOrientation:rotates the map by an angle between 1 and 89 degrees in order
to avoid huge output maps with lots of
                    zero-signal pixels. Possible vaules: false (default, no
automatic rotation), true (automatic rotation)

mapcoordinates: allows to specify the coordinates of the output map. Required
values: mapcenterra (deg), mapcenterdec (deg),
                mapwidthra, mapwidthdec, angle. If mapcoordinates are given the
optimazeOrientation will be ignored even if set to true.

monitor         -: shows the map montor that allows a close visual inspection of
the map building process.
                 default value is 0 (no map monitor). monitor = 1 shows the map
monitor

calibration -: default 0 to calculates pixel corners with standard bolometer
astrometric calibration; 1 for geometrical calculation
              (test purposes)
```

:

The task perform a simple coaddition of images. Thus it can be applied to raster and scan map observations without particular restrictions. The only requirement is that the input frame class must be astrometric calibrated, which means, in the PACS case, that it must include the cubes of ra and dec coordinates of the pixel centers. Thus, photAddInstantPointing and photAssignRaDec should be executed before PhotProject. There is not any particular treatment of the signal in terms of noise removal. The background noise and 1/f noise is supposed to be removed before the execution of this task, e.g. by the previous steps of the pipeline in the case of chooped-nodded observations and by the photHighPassFilter or similar tasks in the scan map case. The simple projection is shown in the following picture.

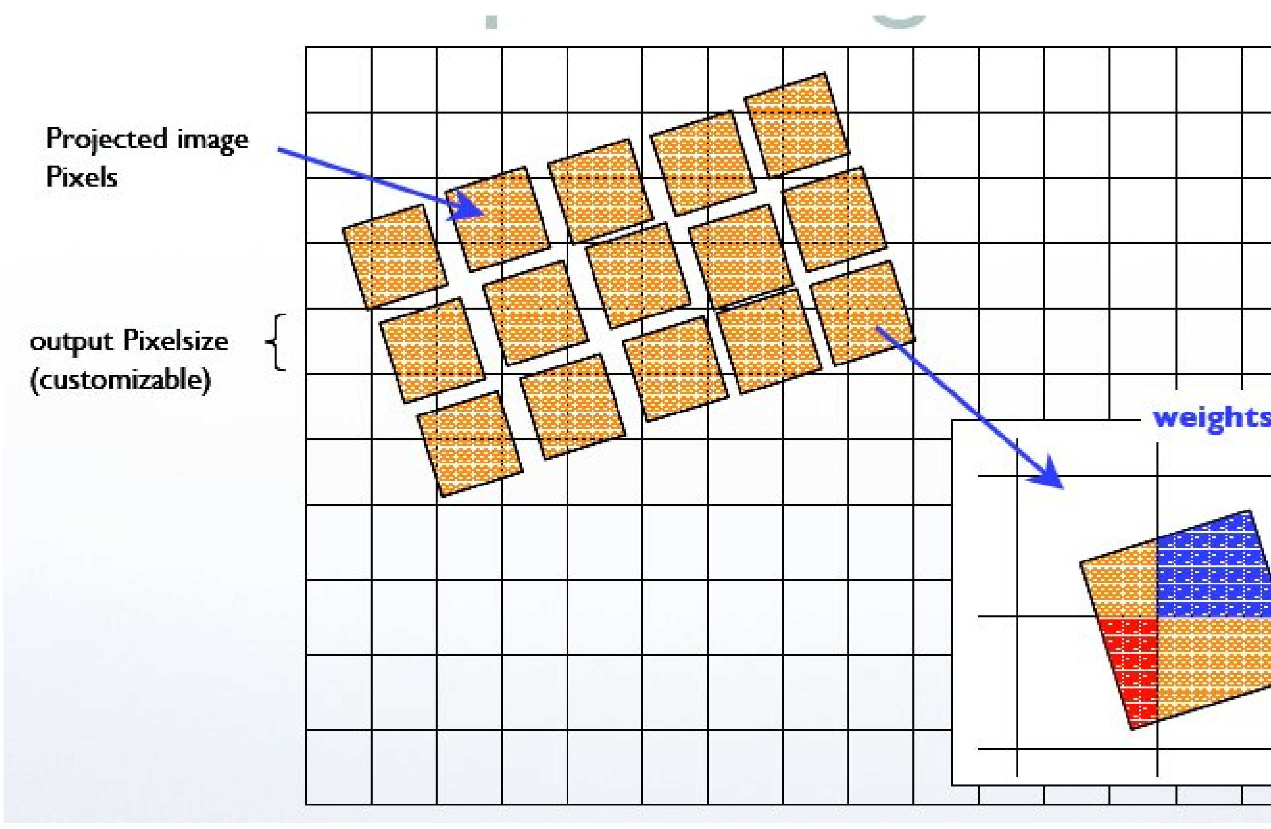


Figure 3.15.

First the task defines the dimensions of the output image on the basis of the input images. The size of the output pixel can be specified by the user in arcseconds by setting the outPixelsize parameter. By default this is the same as the input pixel (3.2" for the blue and 6.4" for the red bolometer, respectively). The user can set this parameter on the basis of the raster or dithering pattern and on the scan map speed. In order to map any input pixel into the output map as shown in the bottom-right corner of the figure above, the task calculates the sky coordinates of the pixel corners. If the parameter calibration is set to true (default), the task uses for this purpose the same method used by photAssignRaDec for calculating the sky coordinates of the pixel center, that is by using the distortion calibration files (see the description of that task for more detail). If calibration is set to false, than the input pixel is supposed to be a square and the coordinates of the corners are calculated by geometry on the basis of the pixel center coordinates and position angle. The first method is preferable under all points of view: it is much less time consuming and it takes into account the distortions of the PACS bolometers. The second method is still available for test purposes and it is very time consuming. Once the corner coordinates are available, first the task transform the signal from flux(Jy) per input pixel into flux(jy) per output pixel. This is done by dividing the input pixel signal by the area mapped by an input pixel in the output image (the colored region in the bottom-right corner of the figure above). After this step the coadded image is obtained with the following method:

$$I_{x'y'} = \frac{\sum_{i=1}^{N_p} a_{xy} w_{xy} i_{xy}}{W_{x'y'}}$$

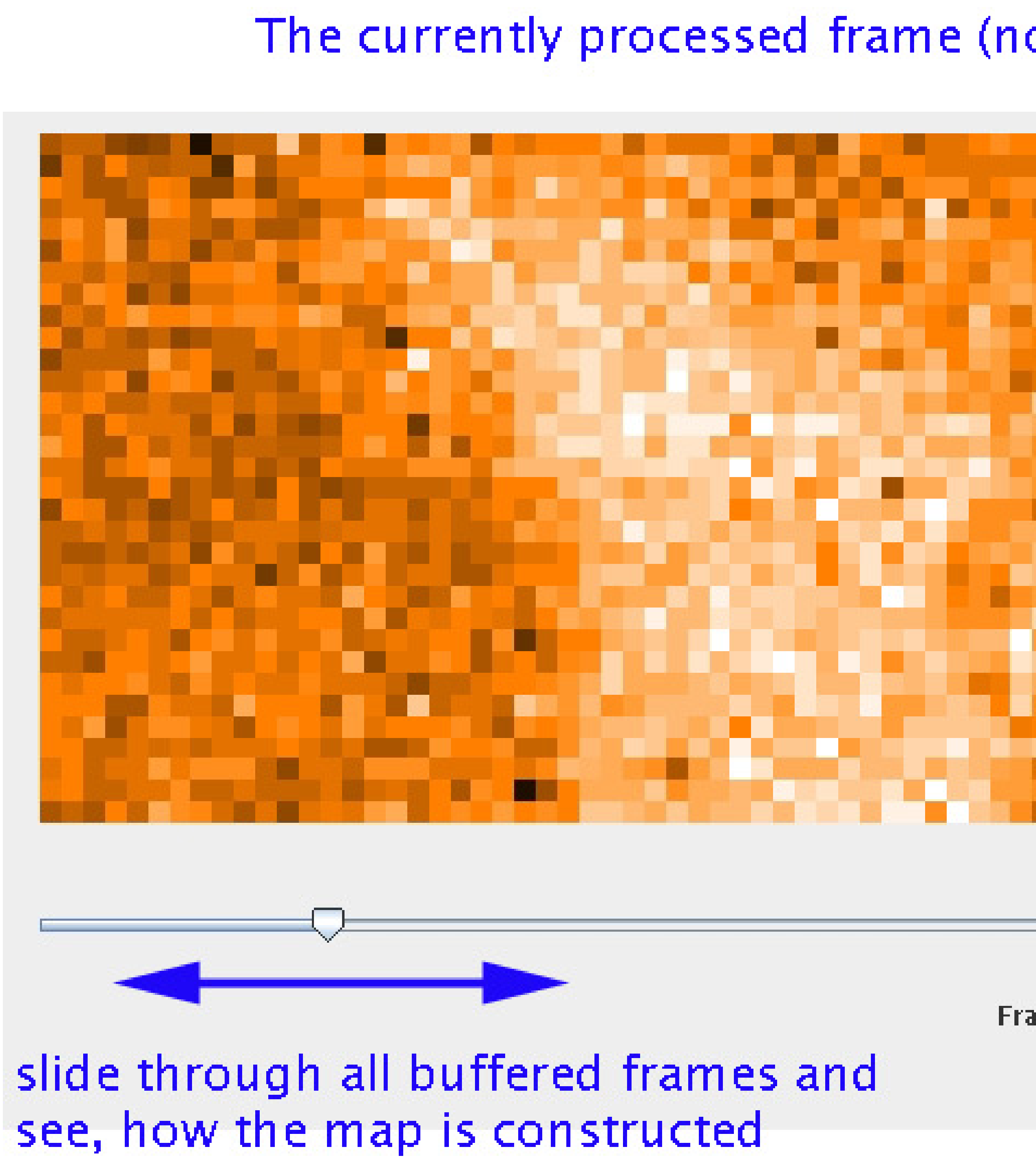
$$W_{x'y'} = \sum_{i=1}^{N_p} a_{xy} w_{xy}$$

Figure 3.16.

where I(x'y') is the flux of the output pixel (x',y'), a(xy) is the geometrical weight of the input pixel (x,y), w(xy) is the initial weight of the input pixel, i(xy) is the flux of the input pixel and W(x'y') is the weight of the output pixel (x'y'). The geometrical weight a(xy) is given by the fraction of ouptput pixel area overlapped by the mapped input pixel (the 4 regions with different colors shown in the bottom-right corner of the figure above), so 0 < a(xy) < 1. The initial weight w(xy) depends on the observation. In case of chooped-nodded observations (point-source, small-source and raster mode), w(xy) should be given by the coverage or exposure map which takes into account the different number of readouts used, pixel by pixel, in the previous averaging processes (averaging of the chopper plateau, averaging of differential on-off images, etc). In the case of scan map observations, w(xy) is just equal to 0 if a pixel is masked out in the available masks (BADPIXEL, SATURATION, GLITCH) and 1 in th opposite case. Thus, the signal Ix'y' of the output image at pixel (x',y') is given the sum of all input pixels with non zero geometrical (a(xy)) and initial weight w(xy), divided by the total weight (sum of the weight of all contributing pixels).

The task provides as output the final map, the coverage map and the noise map. Only the final map has a correct wcs (...well, almost!), the other images are not provided yet with WCS.

3.10.2.3. Features of the Map Monitor


The currently processed frame (no
Fra
slide through all buffered frames and see, how the map is constructed

The use of the Map Monitor is straight forward. After PhotProject is started with the option monitor=1, the Map Monitor appears and shows how the map is constructed. It has a buffer for all processed frames and maps. The slider moves through this buffer and displays the map in all stages of construction. Here are some remarks:

- autodisplay: if this is selected, the map is immediately displayed, while PhotProject processes the data. Uncheck this option and the buffer initially fills much faster.
- memory: depending on the size of the processed Frames class the buffer may use a lot of memory. Start PhotProject with all memory you can afford. If the Map Monitor runs out of memory, it will delete its buffer to avoid out of memory situations and go on showing only the currently processed map. In this low memory mode the slider is disabled (but it still indicates the number of the currently processed frame).

3.11. Chopped Raster AOR

3.11.1. Level 0.5 to Level 1

3.11.1.1. photMakeRasPosCount (jython prototype available)

See description of the same task in the Point-source pipeline

3.11.1.2. photAvgPlateau (java prototype available)

See description of the same task in the Point-source pipeline

3.11.1.3. photDiffChop (java prototype available)

See description of the same task in the Point-source pipeline

3.11.1.4. photDriftCorrection (java prototype available)

See description of the same task in the Point-source pipeline

3.11.1.5. photRespFlatFieldCorrection (java prototype available)

See description of the same task in the Point-source pipeline

NOTE: In the case of chopped raster mode, only the chop cycles and not the nod cycles are defined for any raster position. Usually, the user can specified the raster observations to reproduce also the nod cycle. This makes the observation dependent on the user observation design. Therefore, the data reduction after this step and up to Level 1 of the pipeline can not be generalized.

3.11.2. Level 1 to Level 2

3.11.2.1. photAssignRaDec

See description of the same task in the Point-source pipeline

3.11.2.2. photProject

See description of the same task in the Point-source pipeline

3.12. Scan Map AOR

3.12.1. Level 0.5 to Level 1

3.12.1.1. photDriftCorrection (java prototype available)

See description of the same task in the Point-source pipeline

3.12.1.2. photRespFlatFieldCorrection (java prototype available)

See description of the same task in the Point-source pipeline

3.12.2. Level 1 to Level 2

3.12.2.1. photAssignRaDec

See description of the same task given in the Small-source pipeline

3.12.2.2. The map reconstruction

At this stage of the data reduction the scan map pipeline is divided in two branches: a simple projection given by Photproject and the inversion given by MadMap. The two methods are implemented to satisfy the requirements of different scientific cases. See following subsections for more details.

The simple projection

filterSlew

This task is removing from the science data the readouts corresponding to the satellite slewing, e.g. at the beginning of the science block or between different adjacent scan legs. These readouts need to be discarded in the map reconstruction because they correspond to a satellite acceleration.

```
>> outFrames = filterSlew(inFrames, [,copy=0]  -)

outFrames         -: Frames -: Frames out with one image per one chopper cycle
inFrames          -: Frames -: Frames in
copy              -: int    -:
                       0 -- return reference -: overwrites the input frames
(default)
                       1 -- return copy      -: creates a new output without
overwriting the input
```

The task is just reading the Status table entry "IsSlew" (see description of "photAddInstantPointing" task for more details). This flag is set to "true" for readouts corresponding to the satellite slewing and "false" elsewhere. The readouts with IsSlew=true are removed from the frames class.

photHighPassfilter (jython prototype)

This task is only a prototype. The purpose is to remove the 1/f noise. Several methods are still under investigation. At the moment the task is just using a Median Filter by removing a running median from each readout. The filter box size can be set by the user (filterbox parameter in the scheme below). By default is is 200 readouts.

```
>> outFrames = photHighPassfilter(inFrames, [filterbox=filterbox], [,copy=0]  -)

outFrames         -: Frames -: Frames out with one image per one chopper cycle
```

```
inFrames          -: Frames -: Frames in
filterbox=        -: int    -: median filter box size, by default is 200 readouts
copy              -: int    -:
                       0 -- return reference -: overwrites the input frames
(default)
                       1 -- return copy      -: creates a new output without
overwriting the input
```

A real high pass filter is still under implementation and its use is under investigation.

photProjects

See description of the same task given in the Small-source pipeline

The MadMap case

makeTodArray

Builds time-ordered data (TOD) stream for input into MADmap and derives meta header information of the output skymap. Input data is assumed to be calibrated and flat-fielded. Also prepares the "to's" and "from's" header information for the InvNtt (inverse time-time noise covariance matrix) calibration file.

```

>> PacsTodProduct todProd = makeTodArray(inFrames [,scale=scale] [,crota2=crota2]
[,todname=todname] [,toddir=toddir])

inFrames  -: Data frames in units of mJy/pixel.   Required input meta-data:
            (1) RA,Dec cubes associated with the frames including the effects of
distortion.  Assume this step has
                been previously done by PhotAssignRaDec.
            (2) input mask cube which identifies bad pixels.
            (3) information on band (BS,BL,RED), mode (scan/chopped raster), and
locations between scan legs for data
                -"chunking".

scale     -: %pixel scale of output skymap in relation to nominal PACS detector
size, e.g., 3.2" for Blue and 6.4" Red.
            For scale = 1, the skymap has square pixels equal to nominal PACS
detector size.

crota2    -: CROTA2 of output skymap.  Default = 0.0 degree.

todname   -: Filename of TOD file.

toddir    -: Directory that contains the above TOD file.

todProd   -: Output product representing the TOD binary bit-stream and associated
meta data keywords.


Body of todProd is TOD bit stream binary data file consisting of binary header
information and TOD data
(Reference: http://crd.lbl.gov/~cmc/MADmap/doc/man/MADmap.html).  The binary
header is four 8byte integers

(1) First sample index for TOD data, set to 0.
(2) Last sample index for TOD data chunk, set to (n_good_detectors *
n_samples) --1.
(3) nnObs = Number of detector values per sky pixel during each time sample (for
default one-to-one mapping of
    detectors on to sky pixels, nnObs=1).
(4) total number of sky pixels with good data.

The binary header is followed by the data in the order of:

    For each input GOOD detector pixel ("observation"):
           value (double, 8-byte float) ==v
           For each sky pixel observed:
                  weight (4-byte float)  == w
                  skypixel index (4-byte int) == p

    (e.g.,.) for good detectors ii=1,nd and time samples kk=1,nt, TOD order is
given by:
    for ii=1,nd
          for kk=1,nt
               v[ii,kk]
               for jj=1,nnObs
                  w[ii,kk]
                  p[ii,kk]

Initially for the SPG, we will set nnOBS=1, i.e., use the default one-to-one
mapping of input detectors onto sky pixels.
```

The TOD binary data file is built with format given above and the tod product includes and the astrometry of output map using meta data keywords: CRVAL1 : RA Reference position of skymap

CRVAL2 : Dec Reference position of skymap

EQUINOX : 2000.

CTYPE1 : RA---TAN

CTYPE2 : DEC--TAN

CRPIX1 : Pixel x value corresponding to CRVAL1

CRPIX2 : Pixel y value corresponding to CRVAL2

CDELT1 : pixel scale of sky map (=input as default, user parameter)

CDELT2 : pixel scale of sky map (=input as default, user parameter)

CROTA2 : PA of image N-axis (=0 as default, user parameter)

The weights are set to 0 for bad data as flagged in the mask. Dead/bad detectors (detectors which are always {or usually} bad), are not included in TOD calculations.

The skypix indices are derived from the projection of each input pixel onto the output sky grid. The skypix indices are increasing integers representing the location in the sky map with good data. The skypixel indices of the output map must have some data with non-zero weights,must be continuous, must start with 0, and must be sorted with 0 first and the largest index last.

Future planned parameters that may be implemented include:

medianSub : True/False; Flag to subtract median value from input data (default = false).

nnObs : Number of detector values per sky pixel during each time sample (for default one-to-one mapping of detectors on to sky pixels this value is one (i.e., the value for each sky pixel for one time sample is based on only one detector value). If value for a sky pixel for one time sample is based on multiple values, then nnObs > 1 and one needs to assign the appropriate weights (e.g., fractional area of detector pixel seen by sky pixel, and conserve surface brightness.

maxGap : Maximum size of gap (in samples) before chunking is done.

otfName : On-target-flag name (ONTARGET, HSC-DOC-0662 [PACS-PTREQ-G08] which will be a status flag from the pointing product. Required for data chunking of scan data by scan leg.

runMadMap

The module runMadMap is the wrapper that runs the JAVA MADmap module. MADmap uses a maximum-likelihood technique to build a map from an input Time Order Data (TOD) set by solving a system of linear equations. It is used to remove low-frequency drift ("1/f") noise from bolometer data while preserving the sky signal on large spatial scales. (Reference: http://crd.lbl.gov/~cmc/MADmap/doc/man/MADmap.html). The input TOD data is assumed to be calibrated and flat-fielded and input InvNtt noise calibration file is from calibration tree.

```
>> SimpleImage map = runMadMap(todProd, [calTree=calTree]
[,filterLength=filterLength]
                               [,maxRelError=maxRelError]
[,maxIterarions=maxIterations] -)

todProd       -: The PacsTodProduct from makeTodArray

calTree       -: PacsCalibrationTree containing calibration InvNtt information
stored as an array of size
                max(n_correlation+1) x n_all_detectors.  Each row represents the
InvNtt information for each detector.

filterLength  -: Specifies the length of the FFT's that will be done; code will
make a best guess if not provided.

maxRelError   -: Maximum relative error allowed in PCG routine (default is 1e-6).

maxIterations -: Maximum number of iterations in PCG routine  (default is 50).


map           -: Simple image sky map including header information from the tod
product meta-data.  Products also include
                the naive map (map without corrections), a coverage map, and a
representative noise map (product definition is TBD).
```

The filterLength, which is calculated by the module, must be larger than 2*bandWidth, and can be much longer. For optimum performance filterLength should be the smallest power of two such that filterLength / (ln(filterLength) + 1) >= bandWidth - 1. But, note that for best performance filterLength should not be longer than the stationary time scale of the the noise.

Future planned parameters that may be implemented include:

bandWidth : Width of the non-zero band along the diagonal of inv(N); code will derive from noise file if not provided. The bandWidth is 2*n_correlation +1.

maxMemory : Maximum number of bytes of memory that each process can allocate (default is 1GB).

medianSub : Flag to subtract median of the input data values before MADmap computation, and then the median level is added back into the output sky map. May be helpful for data of limited dynamic range where background >> signal.

3.13. Trend Analysis Product generation

This section is dedicated to the trend analysis product generation. The concept and the scheme of this product generation has to be still finalized. At the moment only the calibration blocks and several HK of each observation are saved as trend analysis products. The tasks responsible for these products are listed and described below. However, it is worth to mention that the implementation of these tasks and their results is prone to change on the basis of calibration scientist requirements.

3.13.1. photTrendCS

This task is not mandatory for the pipeline.

For trend analysis purpose, photTrendCS collects, reduces and stores useful data about the internal calibration sources. This process is applied for each calibration block encountered.

Facultative, this task leaves the frames unchanged and is usualy called by the task PhotCSProcessing.

```
>> outFrames = photTrenCSTask(Frames inframes,trend=trend[,hkdata=hk][,seq=seq]
[,calTree=calTree][rtConverter=rtConverter][,copy=copy])

outFrames    -: Frames              -: unchanged input frames
inFrames     -: Frames              -: input frames
trend        -: PhotTrendCSProducts -: (csbasket) list of frames containing
calibration blocks -- Slots of this product are filled by the PhotTrendCS.
hkdata       -: TableDataset        -: housekeeping information extracted from the
observation context
seq          -: PacketSequence      -: alternative choice when hkdata is unavailable
calTree      -: PacsCalibrationTree -: calibration tree containing all calibration
products used by the pipeline
trConverter  -: CsResistanceTemperature -: Calibration product given resistor to
temperature conversion
copy         -: int                -:
                                   0 -- return reference (default)
                                   1 -- return copy
```

Content of CS product generated :

CS product contains three parts {red, green, blue}. While the red part is always filled, blue and green depends on the current observation. A keyword called 'channel' and stored in the metadata, keeps information on the valid filling part (green or blue). Here are the keywords and information stored:

- "Cs1" in volt, contains CS1 data stored in one calibration block.
- "Cs2" in volt, contains CS2 data stored in one calibration block.
- "Cs1Time" gives the time in microseconds (since 01Jan1958) of each layer of CS1 data cube.
- "Cs2Time" gives the time in microseconds (since 01Jan1958) of each layer of CS2 data cube .
- Meta data
  - "channel" = { red,green, blue} tell us the dataset currently filled.
  - "cs1Cpr" contains the mean of the chopper positions extracted during an observation of the CS1 in command unit (CU)
  - "cs2Cpr" contains the mean of the chopper positions extracted during an observation of the CS2 in command unit (CU)
  - "cs1Temperature" is the average of the cs1 temperature in Kelvin
  - "cs2Temperature" is the average of the cs2 temperature in Kelvin
  - "cs1TemperatureStdDev" is the standard deviation of cs1 temperature in Kelvin
  - "cs2TemperatureStdDev" is the standard deviation of cs2 temperature in Kelvin
  - "bias" is the average of Vh-Vl found on all BU (Buffer Unit).This quantity is in Volt
  - "mode" gives the reading mode led by the warm electronic BOLC .This quantity is a string = {Direct,DDCS} takes from the median of the calibration block.
  - "gain" of the warm electronic, possible values are { 0 = high gain, 1= low gain } . This value is based on the median of the value found in the calibration blocks

Exploring PhoTrendCSProducts : please have a look on PhotCSExtraction task

3.14. Raw Telemetry to Level 0

Usually the pipeline data reduction is supposed to start directly from the Level 0 products. However, this tasks can still be usefull for test purposes.

3.14.1. averageFrames

Average Photometer detector signals (avgNr readouts) for the raw data instrument modes .

```
>> Frames outFrames = averageFrames(Frames inFrames, int avgNr)

outFrames -: Frames -: Frames out
inFrames  -: Frames -: Frames in
avgNr     -: int    -: Number of samples to average
copy      -: int    -:
                     0 -- return reference
                     1 -- return copy
```

- Frame Signals (double) are just averaged
- Frame Wavelengths (double) are just averaged
- The Frame Masks (boolean) are reduced. Optional user may reduce this data by "AND" or "OR" operations against True.
- The Frame Status (int) are averaged. But the then rounded to the nearest Integer.
- The Frame Status (boolean) are treated as Frame Masks
- The Frame Status (string) are selected by majority, in case of no majority the first one is taken

3.14.2. readTm - reading Raw Telemetry

Reading raw telemetry from a PacketRecorder archive file (.tm file) .

```
>> seq = readtm()

seq -: PacketSequence -: PacketSequence containing raw Tm and/or TC SourcePackets
```

In the operational environment this steps will be hidden for the general user. But, of course, within interactive sessions it ois possible to execute every single step and examine the intermediate results.

This will open a file selector box showing all files in your working directory that end with "`.tm`". The telemetry is then loaded from the selected file into a PacketSequence variable called `seq`.

If the `pacs.tm.datapath` property is set to an existing directory, the file selector box will be opened in that directory which makes it easier to navigate from there to your data. The `readtm()` also accepts a filename in which case no file selector box will pop up.

3.14.3. extractDataframes - decompress the science tm packets

This step generate the intermediate Product Decompressed Science data.

```
>> dfs = extractDataframes(seq)

seq -: PacketSequence    -: PacketSequence containing raw Tm and/or TC
SourcePackets
```

```
dfs -: DataframeSequence -: Sequence containing the raw, decompressed DataFrames
```

Again this step is hidden in the pipeline Level 0 data generation.

User may use it as long as the pipeline Level 0 generation is not available or for debugging purposes.

The `extractDataframes` task groups the science telemetry packets per group that can be decompressed together and decompresses them.

The result is a DataFrameSequence, a collection of PacsDataFrame objects. These are decompressed buffers of the two Signal Processing Units (SPU).

3.14.4. decomposeDataframes - organize the raw decompressed data in `Frames` and `PhotRaw` data structures

`Decompose the raw DataFrames into Products suitable for further processing.`

```
>> pacsMix = decomposeDataframes(dfs [,channel=channel] [,mode=mode] [, calVersion
= calVersion])

pacsMix -: PacsMix           -: Container for the Products (Frames, Ramps, PhotRaw)
dfs     -: DataframeSequence -: Sequence of Pacs Dataframes
channel -: String  -:
                    -"red"  -- red channel only
                    -"blue" -- blue channel only
                    -"both" -- both channel (default)
mode    -: String  -: default is all modes
                    -"frames"   -- only frames
                    -"ramps"    -- only Ramps
                    -"subramps" -- only Subramps
                    -"rawramps" -- only Raw Ramps
calVersion -: String -: Version of the calibration files used
```

`PacsDataFrames` contain the result of the Decompression (reduced data, raw data, DecMec data and Compression Header data).

This pipeline step is restructuring the data in a proper format for further scientific analysis. This proper formatted products are `Frames` and `PhotRaw`, depending on the instrument algorithm and compression mode.

`Frames` contain the reduced data as data cube, collapsed DecMec information, and decoded Label information in the associated Status. The `PhotRaw` product contain raw channel data (non averaged data) e.g. of the rotating additional raw channels. `Pixel deselected with the Detector Selection Table are masked by the BLINDPIXEL mask.`

`decomposeDataframes` return a `PacsMix` . Depending on the (possibly different) :

- Instrument Algorithms
- Compression Modes
- User selections (Red channel and / or blue channel)

the user the `PacsMix` contain one or more `Frames` and/or `PhotRaw` products. For `Frames` the DecMec data are collapsed from the full readout sampling to the frequency of the reduced data :

- OBSID : first entry of the associated block of DecMec data
- BBID : first entry of the associated block of DecMec data
- LBL : first value + decoded value + check whether it change
- TMP1 : first value
- TMP2 : first value
- FINETIME : first value
- VLD : first value + check it is not changing
- CPR : mean value
- WPR : mean value
- BOLST : mean value
- CRDC : first value :
- CRDCCP : first value
- DBID : first value
- BSID : first value

Calibration Files :

- `FilterBandConversion` : Filter to Band Conversion.
- `LabelDescription` : label description.

Going from Level 0 to Level 0.5 implies extracting/collecting the necessary auxiliary data.

3.14.5. readAttitudeHistory

Read the attitude history.

```
>> attitude = readAttitudeHistory(pdfs)

attitude -:
pdfs     -: DataframeSequence -: Sequence of Dataframes
```

Reads the instantaneous pointing product covering the same time as the dataframes in the Dataframe-Sequence pdfs.

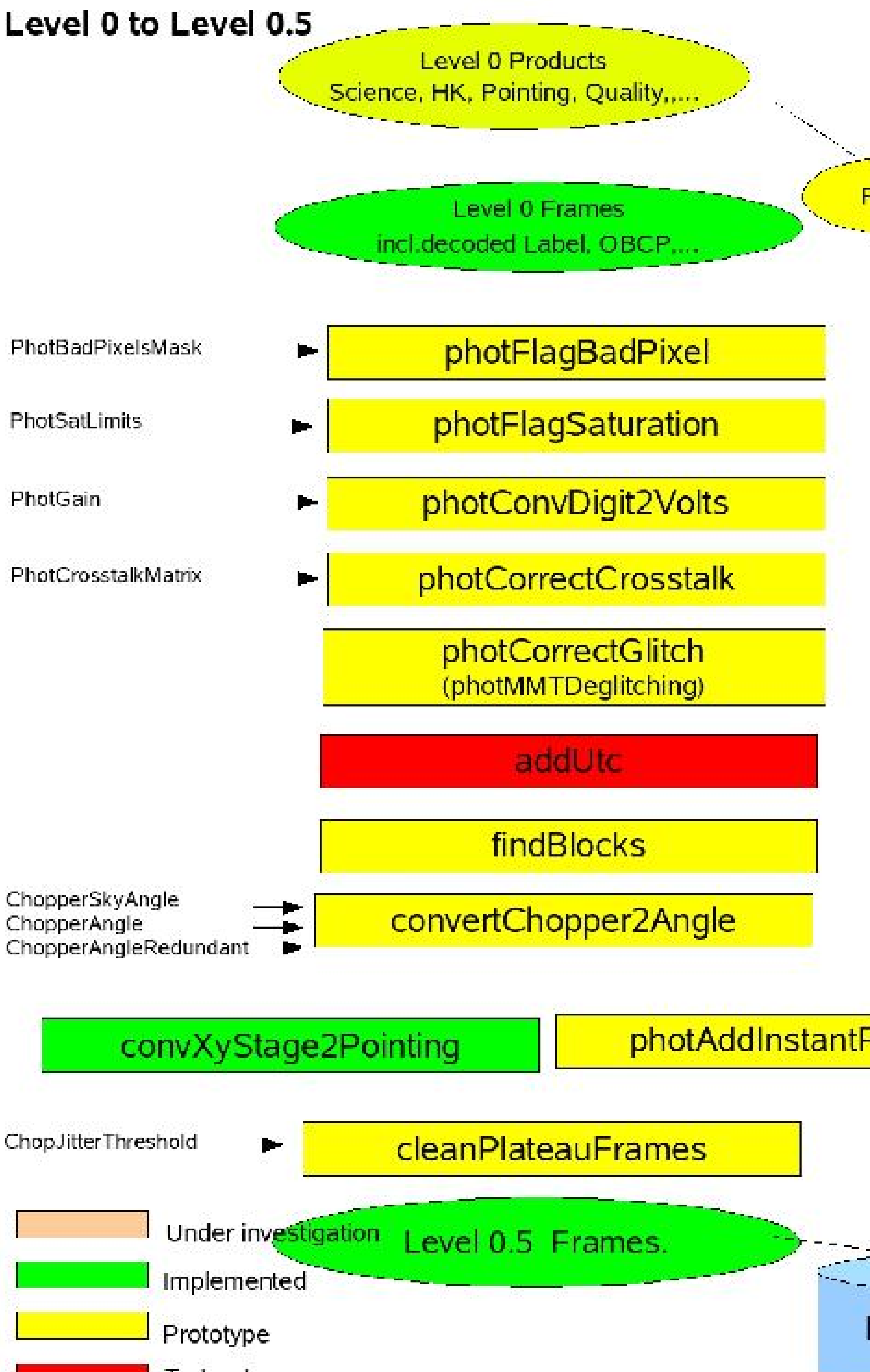
3.14.6. readTimeCorellation

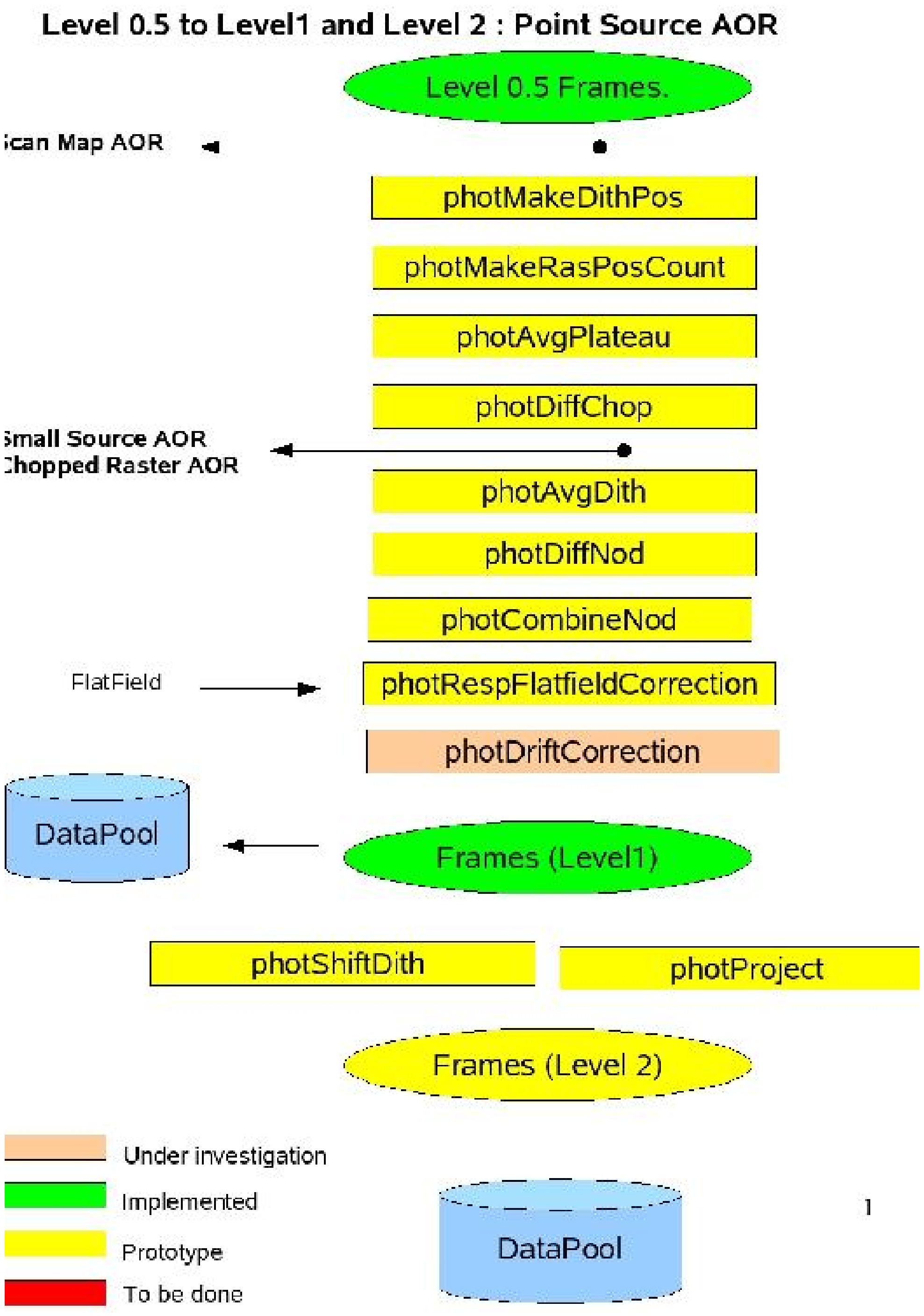
Read the Time Correlation information.

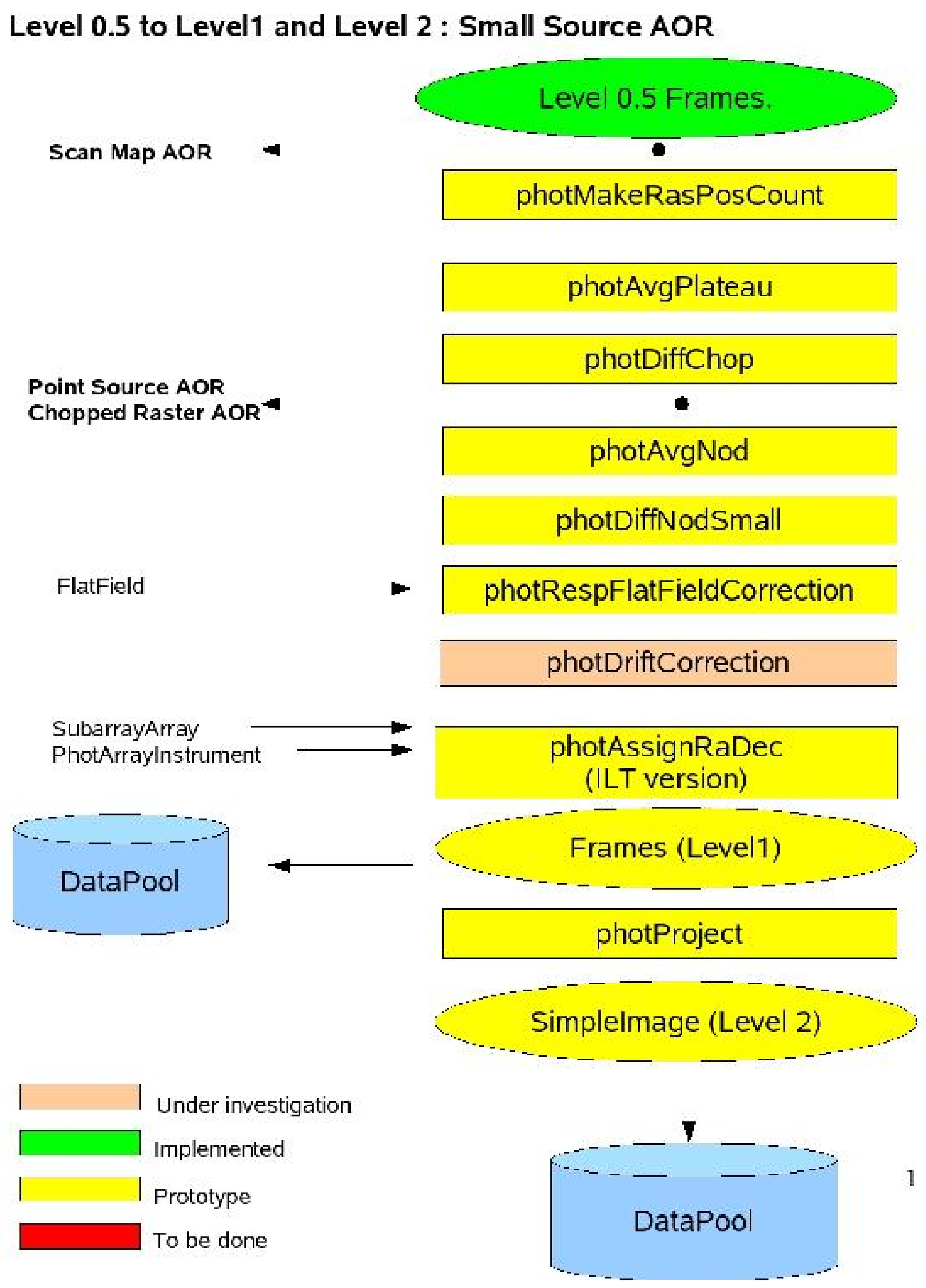
```
>> timecor = readTimeCorellation(pdfs)

timecor -: TableDataset -: Time correction values
pdfs    -: DataframeSequence -: Sequence of Dataframes
```

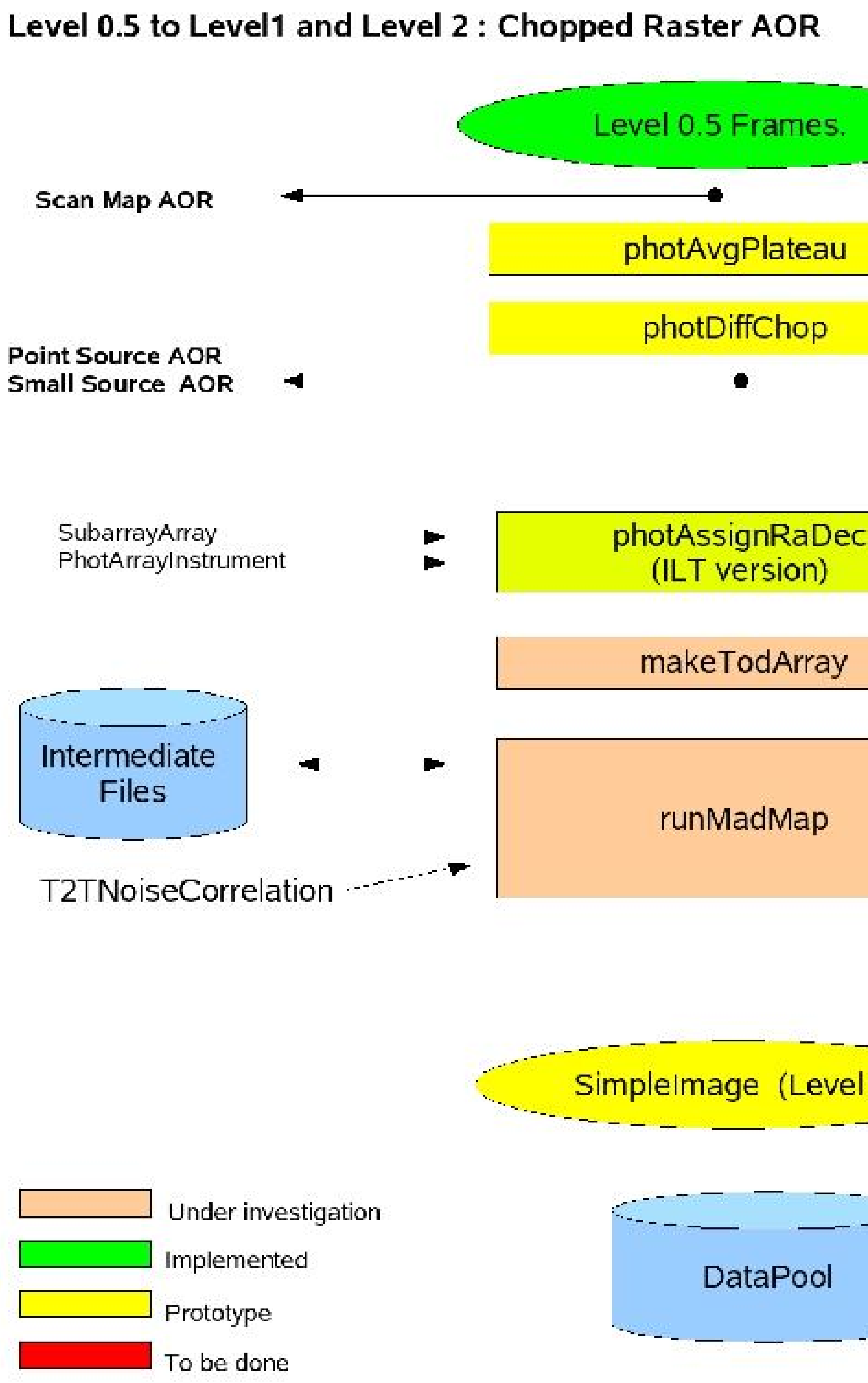
Reads the time corellation product covering the same time as the dataframes in the DataframeSequence pdfs.

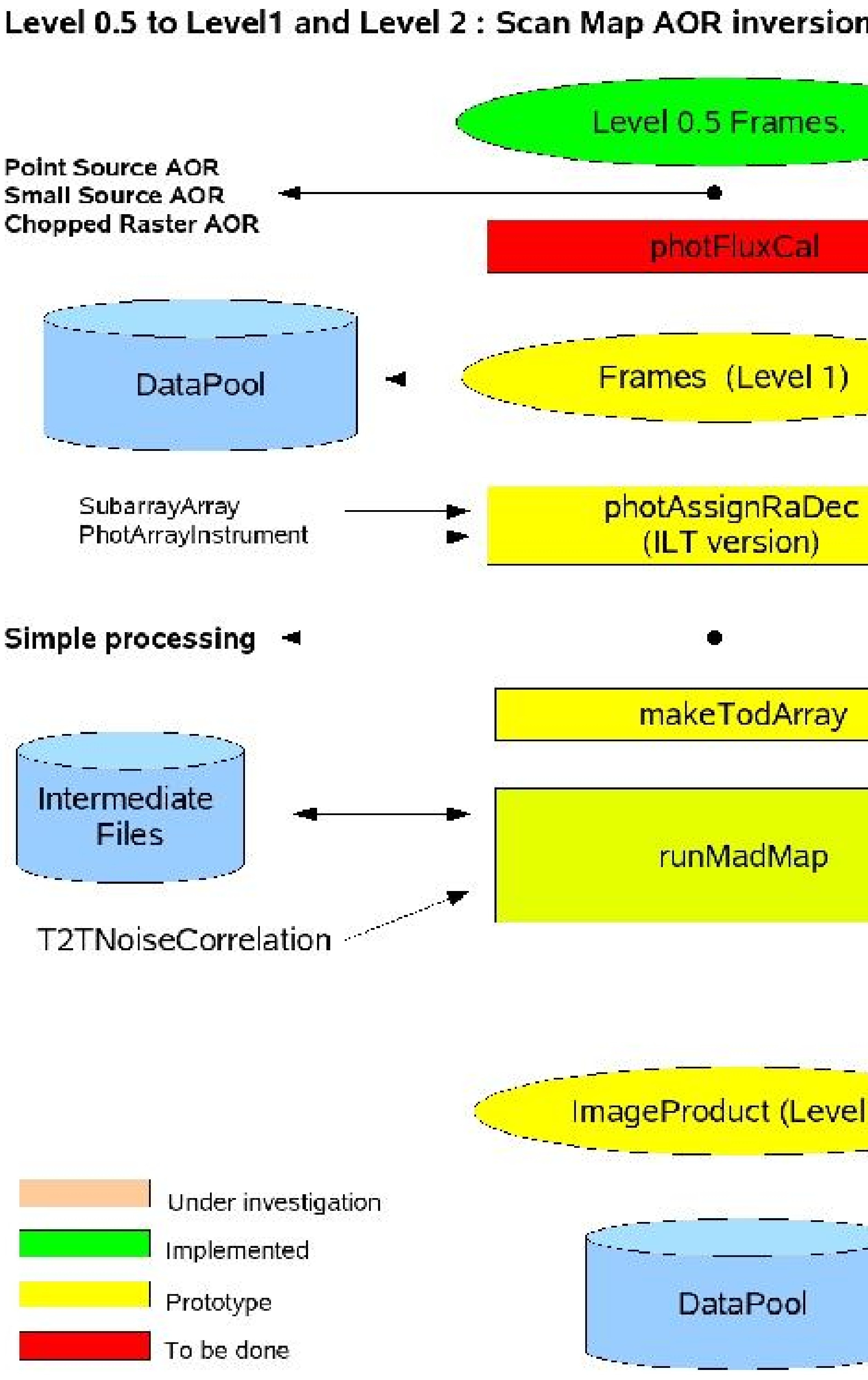
3.15. SPG Pipeline chart

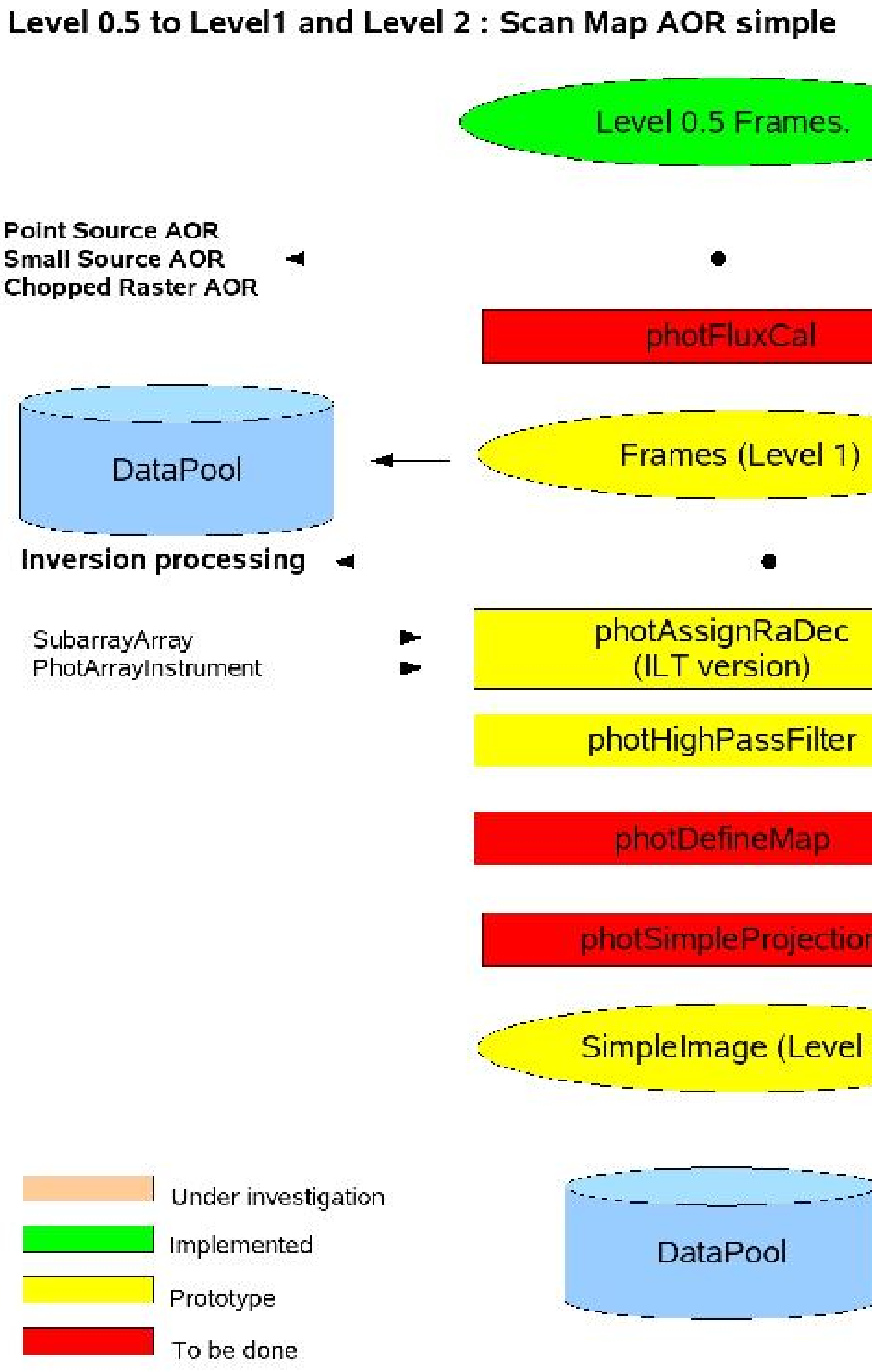



Level 0.5 to Level1 and Level 2 : Point Source AOR
Level 0.5 Frames.
ican Map AOR
photMakeDithPos
photMakeRasPosCount
photAvgPlateau
photDiffChop
imall Source AOR
Chopped Raster AOR
photAvgDith
photDiffNod
photCombineNod
FlatField
photRespFlatfieldCorrection
photDriftCorrection
DataPool
Frames (Level1)
photShiftDith
photProject
Frames (Level 2)
Under investigation
Implemented
Prototype
To be done
DataPool
1


Level 0.5 to Level1 and Level 2 : Small Source AOR
Level 0.5 Frames.
Scan Map AOR
photMakeRasPosCount
photAvgPlateau
photDiffChop
Point Source AOR
Chopped Raster AOR
photAvgNod
photDiffNodSmall
FlatField
photRespFlatFieldCorrection
photDriftCorrection
SubarrayArray
PhotArrayInstrument
photAssignRaDec
(ILT version)
DataPool
Frames (Level1)
photProject
SimpleImage (Level 2)
Under investigation
Implemented
Prototype
To be done
DataPool
1


Level 0.5 to Level1 and Level 2 : Chopped Raster AOR
Level 0.5 Frames.
Scan Map AOR
photAvgPlateau
photDiffChop
Point Source AOR
Small Source AOR
SubarrayArray
PhotArrayInstrument
photAssignRaDec
(ILT version)
makeTodArray
Intermediate Files
runMadMap
T2TNoiseCorrelation
SimpleImage (Level
Under investigation
Implemented
Prototype
To be done
DataPool


Level 0.5 to Level1 and Level 2 : Scan Map AOR inversion
Level 0.5 Frames.
Point Source AOR
Small Source AOR
Chopped Raster AOR
photFluxCal
DataPool
Frames (Level 1)
SubarrayArray
PhotArrayInstrument
photAssignRaDec
(ILT version)
Simple processing
makeTodArray
Intermediate Files
runMadMap
T2TNoiseCorrelation
ImageProduct (Level
Under investigation
Implemented
Prototype
To be done
DataPool


Level 0.5 to Level1 and Level 2 : Scan Map AOR simple
Level 0.5 Frames.
Point Source AOR
Small Source AOR
Chopped Raster AOR
photFluxCal
DataPool
Frames (Level 1)
Inversion processing
SubarrayArray
PhotArrayInstrument
photAssignRaDec (ILT version)
photHighPassFilter
photDefineMap
photSimpleProjectio
SimpleImage (Level
Under investigation
Implemented
Prototype
To be done
DataPool

3.16. Product summary

Table 3.1. Overview - last updated 2006/06/09

Level	Product name	Status
0	readTm	Done
0	extractDataframes	Done
0	decomposeDataframes	Done
0.5	readAttitudeHistory	toDo
0.5	readTimeCorrelation	toDo
0.5	extractDMC	toDo
0.5	extractFrames	Done
0.5	photFlagSaturation	Done
0.5	photConvDigit2Volts	Done
0.5	photFlagBadPixels	Done
0.5	photFlagGlitch	toDo
0.5 - 1	decodeLabel	prototype
0.5 - 1	findBlocks	prototype
1	convChopper2Angle	Prototype
1	convXYStage2Pointing	Done
1	photAddInstantPointing	toDo
1	photCorGlitch	toDo
1	cleanPlateau	Prototype
1	photAvgPlateau	Prototype
1	photSpatialCal	toDo
1	photDiffChop	toDo
1	photAvgNode	toDo
1	photFluxCal	toDo
1	photSkyRespCal	toDo
1	photCsRespCal	toDo
1	addUTC	toDo
1	constructStack	toDo
1	photCorrZeroLevel	toDo

3.17. Appendix

3.17.1. How to remove sky background and telescope emission

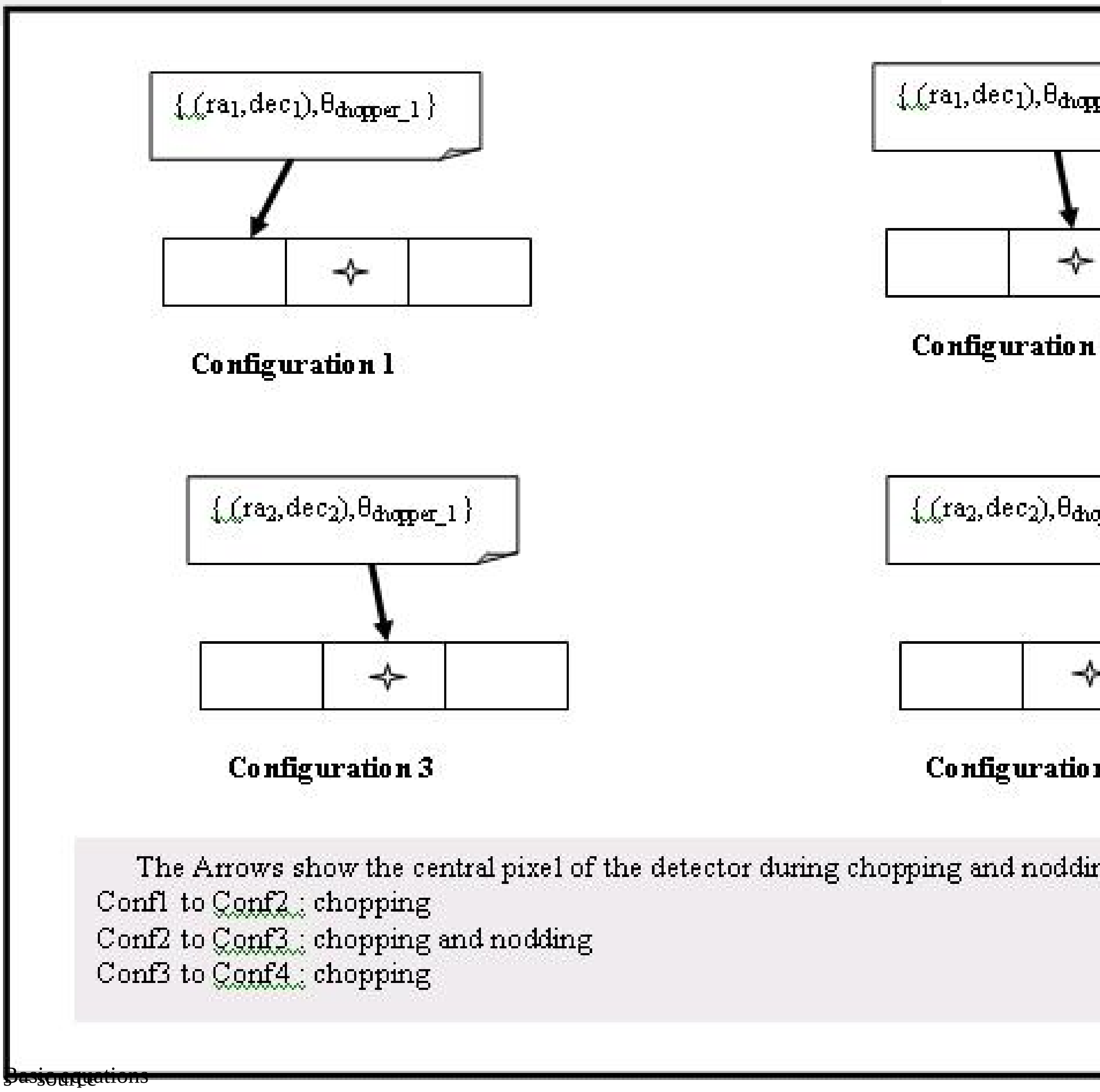
Telescope emission is the major flux received by the detector. During his life telescope temperature should be to 80Â°K on average and his emissivity to 4%.

In order to remove instrumental error, chopping and nodding mode are used. If the chopper doesnÂ’t move, the optical path in PACS doesnÂ’t change when the telescope pointing is running. Successive pointing positions in the sky are lead by the satellite pointing and are called nodes. If the chopper is moving, the optical path in PACS instrument change. So that, the flux received by the detector doesnÂ’t have the same telescope emission.

Three sky areas are used:

- one containing a brightness source,
- and two others with no source

```
Configuration 1 -: first position -: point-source is observed ❶
Configuration 2 -: after chopping                             ❷
Configuration 3 -: after nodding and chopping                 ❸
Configuration 4 -: after chopping                             ❹
```


{ (ra1,dec1), θchopper_1 }
{ (ra1,dec1), θchopp
Configuration 1
Configuration
{ (ra2,dec2), θchopper_1 }
{ (ra2,dec2), θcho
Configuration 3
Configuratio
The Arrows show the central pixel of the detector during chopping and noddi
Conf1 to Conf2 : chopping
Conf2 to Conf3 : chopping and nodding
Conf3 to Conf4 : chopping

Basic equations

s = source

f = foreground (telescope emission)

b = background

❶ In configuration (1) brightness source exists: I_1 (1)

2. Satellite pointing is unchanged; chopping angle is changed in order to have brightness source in the field of view;

❷ $I_2 = b_2$ (2)

3. Satellite pointing has been changed; chopping angle is changed in order to align the previous source in the same previous matrix area (case 2)

❸ (3)

4. Satellite pointing is unchanged; chopper angle is changed.

❹

Now it is possible to remove telescope background emission:

I(A1)=(A1-A2)... the mean of the two couple of images

If the sky background is different between two positions, it wonÂ´t be possible to correct the sky target properly.

Example 3.1. Four observing configurations exist:

Glossary

Stack	sorted list of the observations of an astronomical object got in a given context. All of these observations can be processed in the same way.
Plateau	data sequence got during an elapsed time while the chopper and telescope pointing are unchanged.
Cycle	one ON/OFF chopper sequence
Node	telescope pointing is fixed (only slight motion due to the jitter can be found)