# PACS Data Reduction Guide: Photometry

**Issue user. Version 10**
**Apr. 2012**

# PACS Data Reduction Guide: Photometry

# Table of Contents


1. PACS Launch Pads ..... 1
    1.1. Introduction ..... 1
    1.2. PACS Data Launch Pad ..... 1
        1.2.1. A quick on terminology ..... 1
        1.2.2. Getting your data from the archive into HIPE ..... 1
        1.2.3. Looking at your fully-reduced data ..... 4
    1.3. PACS Photometry Launch Pad ..... 5
        1.3.1. Does the observation data need re-processing? ..... 5
        1.3.2. Re-processing with the pipeline scripts ..... 5
        1.3.3. Considerations when running the pipeline ..... 6
        1.3.4. Further processing ..... 6
2. Setting up the pipeline ..... 8
    2.1. Terminology ..... 8
    2.2. Getting and saving your observation data ..... 8
        2.2.1. Getting ..... 8
        2.2.2. Saving ..... 9
    2.3. What and where are the pipeline scripts? ..... 10
    2.4. How much can you improve on the automatic pipeline? ..... 11
    2.5. Calibration files and the calibration tree ..... 11
        2.5.1. Installing and updating the calibration files ..... 11
        2.5.2. Checking what has been updated ..... 11
        2.5.3. The calibration tree ..... 12
        2.5.4. Comparing calibration file versions ..... 12
    2.6. Saving your *ObservationContext* and its calibration tree to pool ..... 13
3. In the Beginning is the Pipeline. *Photometry* ..... 14
    3.1. Introduction ..... 14
    3.2. Science case interactive pipeline scripts ..... 15
        3.2.1. Bright point source script ..... 15
        3.2.2. Deep survey maps ..... 26
        3.2.3. Extended sources ..... 30
    3.3. Multiple obsids ..... 31
4. Selected topics of data reduction. *Photometry* ..... 36
    4.1. Introduction ..... 36
    4.2. Used Masks ..... 36
    4.3. Second level deglitching ..... 36
        4.3.1. Pre-requisites and what is 2nd level deglitching? ..... 36
        4.3.2. Command-line syntax ..... 37
        4.3.3. The most important syntax options ..... 37
        4.3.4. A detailed look at the MapIndex task ..... 38
        4.3.5. A detailed look at the IIndLevelDeglitch task ..... 41
        4.3.6. Deglitching without MapIndex ..... 43
        4.3.7. MapIndexViewer: a useful tool for diagnostic and fine tuning ..... 44
    4.4. MMT Deglitching ..... 46
        4.4.1. Details and Results of the implementation ..... 48
    4.5. photRespFlatFieldCorrection ..... 50
    4.6. photHighPassfilter ..... 50
    4.7. photProject ..... 51
    4.8. photProjectPointSource ..... 53
    4.9. Features of the Map Monitor ..... 53
    4.10. Reducing minimaps (combining scan and cross-scan) ..... 54
    4.11. Dealing with Solar System objects (SSOs) ..... 59
        4.11.1. correctRaDec4Sso ..... 59
5. PACS scanmap reduction using MADmap ..... 61
    5.1. MADmap ..... 61
    5.2. MADmap pre-processing ..... 62

5.2.1. Pixel-to-pixel offset correction ..... 62
5.2.2. Module to module drift correction ..... 63
5.2.3. Correction for global signal drifts ..... 64
5.3. Optimising MADmap pre-processing ..... 67
5.3.1. Overriding defaults in option 1 ..... 67
5.3.2. Segmenting time-streams ..... 67
5.3.3. “long-pass” filtering ..... 67
5.3.4. Other global drift correctors ..... 68
5.4. Running MADmap ..... 68
5.4.1. Preparation ..... 68
5.4.2. makeTodArray ..... 69
5.4.3. runMadMap ..... 71
5.5. MADmap post-processing ..... 72
5.5.1. Introduction ..... 72
5.5.2. Map making basics ..... 72
5.5.3. Unrolling and Median filtering ..... 73
5.5.4. PGLS algorithm ..... 73
5.5.5. Results ..... 74
5.5.6. References ..... 75
5.5.7. Usage ..... 75
5.6. Open issues and known limitations ..... 75
5.6.1. Computing requirements ..... 75
5.7. Troubleshooting ..... 75
5.7.1. Glitches in the readout electronics ..... 75
5.7.2. Improper module-to-module drift correction ..... 76
5.7.3. Point source artifact ..... 76
6. Appendix ..... 77
6.1. Introduction ..... 77
6.2. PACS products: what they contain ..... 77
6.2.1. The ObservationContext ..... 77
6.2.2. Spectroscopy: *Frames, PacsCube, PacsRebinnedCube, SpectralSimpleCube* ..... 79
6.2.3. Spectroscopy: Sliced Products ..... 81
6.3. Information and organisation tables in PACS products ..... 82
6.3.1. Meta Data ..... 83
6.3.2. Status table ..... 87
6.3.3. BlockTable and MasterBlockTable ..... 89
6.4. PACS product viewers ..... 92
6.4.1. The PACS Spectral Footprint Viewer ..... 92
6.4.2. PACS product viewer (PPV) ..... 93

# Chapter 1. PACS Launch Pads

## 1.1. Introduction

*May 2012. The* PDRG *has been split into a Spectroscopy and Photometry part. The pipeline chapters have been updated. This has been written for track 9 of HIPE, but much will also be valid for track 10.*

*Welcome to the PACS data reduction guide (*PDRG*) #. We hope you have gotten some good data from PACS and want to get stuck in to working with them. This guide begins with a series of "launch pads" that from Chap. 1; essentially quick-start guides to working with PACS data. These will show you the fastest ways to get your data into HIPE, to inspect the HSA-pipeline reduced cubes (spectroscopy) or images (photometry), and will outline what you need to consider before you start to reduce the data yourself through the pipeline.* For PACS observations we recommend you always re-reduce your data: to include the latest calibrations, so that you can check the intermediate results (deglitching, map-making and masking, flat-fielding, etc), and because some pipeline tasks are not carried out by the automatic pipeline that the HSA-retrieved data were processed through.

*Contents*: Chap. 2 takes you through what you need to do and know before you start pipeline processing your data, Chap. 3 is dealing with the different PACS photometry pipelines and Chaps 4 and 5 contain more detailed information about photometry data processing (e.g. deglitching and MADmap). In the appendix are issues that are common to photometry and spectroscopy such as explained the organisation and contents of PACS data products.

Additional reading can be found on the HIPE help page, which you can access from the HIPE "*Help>Help Contents*" menu. This covers the topics of: HIPE itself, I/O, scripting in HIPE, and using the various data inspection and analysis tools provided in HIPE. We will link you to the most useful bits of this documentation—we do not repeat the information given there, only material that is PACS-specific is in this *PDRG*. You can also consult the PACS public wiki for the Observer's Manual and calibration information and documentation (herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWed?template=viewprint). This is also linked from the PACS section of the HIPE help page. Information on the calibration of PACS data is **not** covered in this *PDRG*, you have to look at the wiki for these.

## 1.2. PACS Data Launch Pad

### 1.2.1. A quick on terminology

The following Help documentation acronyms are used here (the names are links): **DAG**: the Data Analysis Guide; **SaDM**, the Scripting and Data Mining Guide.

**Level 0** products are raw and come straight from the satellite. **Level 0.5** products have been partially reduced, corrected for instrument effects generally by tasks for which no interaction is required by the user. **Level 1** products have been more fully reduced, some pipeline tasks requiring inspection and maybe interaction on the part of the user. **Level 2** products are fully reduced, including tasks that require the highest level of inspection and interaction on the part of the user. **Level 2.5** products, which are found for some of the pipelines, are generally those where observations have been combined or where simple manipulations have been done on the data.

Text written *like this* usually means we are referring to the class of a product (or referring to any product of that class). Different classes have different (java) methods that can be applied to them and different tasks will run (or not) on them, which it why it is useful to know the class of a product. See the SaDM to learn more about classes. Text written `like this` is used to refer to the parameter of a task.

### 1.2.2. Getting your data from the archive into HIPE

There are a number of ways to get Herschel data into HIPE, and those data can come in more than one format. The PACS pipelines use one method, which is documented in the pipeline script, but here

we give an overview of the simplest ways to get Herschel data into HIPE. This topic is covered in detail chap. 1 of the DAG.

Herschel data are stored in ESA's Herschel Science Archive (HSA),

- They are identified with a unique number known as the Observation ID (obsid)
- HIPE expects the data to be in the form of a **pool**, so HSA data must be *imported* into HIPE
- A **pool** is like a database, with observations organised as an **Observation Context**, containing links to all the data, calibration, and supplementary files, and including the raw, partially-, and fully-reduced data products

There are several ways to get observations imported from the HSA, or disk, into HIPE:

- Directly from the HIPE command line, using

```
obsid = 134....... # enter your own obsid
# to load into HIPE:
myobs = getObservation(obsid, useHsa=True)
# to load into HIPE and at the same time to save to disk:
myobs = getObservation(obsid, useHsa=True, save=True)

# You must be logged on to the HSA for this to work:
# See the DAG sec. 1.4.5.
```

  See the *DAG* sec. 1.4.5 for more information on getObservation (for example, how to log-on to the HSA before you can get the data). If you use the parameter save=True in getObservation then the data are at the same time saved to disk to your MyHSA: note that saving can take a while. Information and links about accessing MyHSA are also provided in sec. 1.4.5.

  This method is useful for single observations and brings the data directly into HIPE in the format that the PACS pipeline requires.

- Download a tar file (which is not a pool) from the HSA. See the *DAG* sec. 1.4.7 for more information on this. This is the method to use if you have several observations, or a very large one.

  If you are getting a tarfile, then you will have gone into the HSA, identified your observations, and asked for them to be send via ftp. From the tar file you get, you will need to import the data into HIPE. This is explained in the *DAG* sec. 1.5; to summarise, after having untarred the file you do the following in HIPE:

```
#   Get the data from the HSA as a tarball
#   On disk, untar the tarball, e.g.
cd /Users/me/fromHSA
tar xvf meme1342.tar

# in HIPE, then
myobsid=1342..... # enter your obsid
mypath="/Users/me/fromHSA/me1342
myobs=getObservation(obsid=myobsid,path=mypath)

# obsid is necessary only if more than one observation
# is in that directory, i.e. if your tarfile has several
# obsids in it
```

- Import directly into HIPE from the HSA via its GUI, which is useful for single observations and brings the data directly into HIPE in the format that the PACS pipeline requires. This is covered in the *DAG* sec. 1.4.5 ("using the GUI: HUI"). Briefly, you need to find your observation (e.g. by obsid), then in the query results tab of the HUI, click the download icon to access a "Retrieve Products" menu, from which select to download "All".

> **Tip**
>
> To get to the HSA from HIPE use the viewer from the HIPE menu *Window>Show View>Data Access>Herschel Science Archive*. Anything you download directly into HIPE from there will appear in the *Variables* pane of HIPE, and often the command will also be echoed to the *Console*.

After you have done this you can:

- If you loaded your data into HIPE using the first or third method above, then you must save the observation to disk.

  If using getObservation you can chose two ways to save the data to disk: either to your MyHSA, using the parameter save=True in getObservation as explained above; or to a pool on disk, using the command saveObservation.

  To use saveObservation:

  ```
  # "myobs" is the name of the ObservationContext
  pooln = "NGC3333"
  saveObservation(myobs, poolName=pooln)
  ```

  where the observation goes to your **local store** (HOME/.hcss/lstore) to a directory (pool) with the name that is either the obsid or as it given in `poolName`. The *DAG* does not explain saveObservation but it does explain the same parameters of getObservation: see sec. 1.7.

  **Note:** if using saveObservation, the *ObservationContext* is not saved by default with the calibration tree (calibration files) it was reduced with. If you wish to do that, see Sec. 2.6: you need to set the parameter saveCalTree=True, which will save whatever calibration tree is in your *ObservationContext*.

  f you used the second method you do not need to save your observation as a pool, you could keep them as they are. If you do want to convert your observation data on disk from the "HSA-format" to the pool format, you can simply use saveObservation as in the example above.

- Once the data are saved to this pool with saveObservation, on your disk, then any other time you want to access them you should use getObservation,

  ```
  obsid = 134...... # enter your obsid here
  pooln = "NGC3333"
  myobs=getObservation(obsid, poolName=pooln)
  ```

  The *DAG* explains these parameters of getObservation: see sec. 1.7 to know how to access the data if they were saved elsewhere than your local store. The command will get the data as located in the local store (HOME/.hcss/lstore) and with a directory (pool) name that is either the obsid (e.g. "342221212") or the value set by `poolName`. You can also use a GUI to get data, e.g. from your MyHSA: see sec. 1.7 of the *DAG*.

So, to summarise the command-line methods:

```
obsid = 134...... # enter your obsid here

# Direct I
#   Get the data from the HSA and then save to
#   /Users/me/.hcss/lstore/MyPoolName
myobs=getObservation(obsid, useHsa=True)
saveObservation(myobs, poolName="MyPoolName")
#   Then, to later get those data
myobs=getObservation(obsid, poolName="MyPoolName")
#
#   Get the data from the HSA and then save to
#   /Users/me/.hcss/lstore/[obsid as a string]
myobs=getObservation(obsid, useHsa=True)
saveObservation(myobs)
#   Then, to later get those data
```

```
myobs=getObservation(obsid)

# You must be logged on to the HSA for this to work:
# See the DAG sec. 1.4.5.
# See Sec. ??? to learn about saving and then
# restoring the caltree



# Direct II
#   Get the data from the HSA and immediately save
#   to /Users/me/.hcss/lstore/MyPoolName
myobs=getObservation(obsid, useHsa=True, save=True, poolName="MyPoolName")
#   Then to later get those data
myobs=getObservation(obsid, poolName="MyPoolName")



# DIRECT III
#   Get the data from the HSA as a tarball
#   On disk the data are in directories off of /Users/me/fromHSA
#   In HIPE, then
myobsid=1342..... # enter your obsid
mypath="/Users/me/fromHSA/me1342
myobs=getObservation(obsid=myobsid,path=mypath)
obsid = 134...... # enter your obsid here
mypath="/Users/me/fromHSA/me1342
myobs=getObservation(obsid=myobsid,path=mypath)
```

For the GUI-based methods, read chap. 1 of the DAG. For full parameters of getObservation, see its URM entry: here. (Note: there are two "getObservation"s in the URM. The one I link you to is the correct one, it is also the first in the URM list.)

# 1.2.3. Looking at your fully-reduced data

Once the data are in HIPE, the *ObservationContext* will appear in the HIPE Variables panel. To look at the fully-reduced, final Level 2 product (cubes for the spectrometer and images for the photometer) do the following,

- Double-click on your observation (or right-click and select the **Observation Viewer**)
- In the directory-like listing on the left of the Observation viewer (titled "Data"), click on the + next to the "level2"
- *For photometry*: go to HPPMAPB to get the blue map or the HPPMAPR to get the red Naive map. The map will open to the right of the directory-like listing, but if you want to view it in a new window then instead double-click on the "HPPMAPB" (or right-click to select the **Standard Image Viewer**)
- *For spectroscopy*: go to the + next to the HPS3DPR (red *projected* cube) or HPS3DPB (blue *projected* cube) or to the + next to the HPS3DRR (red *rebinned* cube) or HPS3DRB (blue *rebinned* cube)—see e.g. Sec. ??? to know what the difference between these two cubes are (basically, the projected cubes are the result of a mosaicking of the rebinned cubes, and the projected cubes have smaller, but more, spaxels). The list of numbers (+0, +1...) are the individual cubes, one per wavelength range specified in your AOR (observation request) and also one per pointing for the rebinned cubes. The tooltip that appears when you hover over a cube in the listing will help you work out what the contents of that cube is. Click on the number to see the cube open to the right of the directory-like listing, or to see it in a new window, right-click and chose the **Spectrum Explorer** (SE: see the *DAG* chap. 6). From the SE you can look at the spectra of your spaxels, perform mathematical operations, extract spectra or subcubes, make velocity and flux maps and fit your spectra. (Note, however, that you cannot perform all the mathematical operations if you open the SE from the Obervation viewer in this way; you need to drag and drop the cube to the *Variables* pane and open it from there.)

To learn more about the layers of the *ObservationContext* and what the products therein are, see Sec. ???.

# 1.3. PACS Photometry Launch Pad

The following Help documentation acronyms are used here: **DAG**: the Data Analysis Guide; **PDRG**: PACS Data Reduction Guide.

## 1.3.1. Does the observation data need re-processing?

The PACS ICC recommend that you always reprocess your data,

- The pipeline tasks and the calibrations are still undergoing improvement and the pipeline that the reduced data you got from the HSA may not have been the most recent
- There are some pipeline stages that for all but the most simple data you ought to inspect the results of the pipeline task yourself, to decide if you wish to change the default parameters

Information about calibration files held in the **calibration tree**:

- When you start HIPE, HIPE will begin by looking for a calibration file update: Sec. 2.5.1.
- To check what version of calibration files and the pipeline your HSA-gotten data were reduced with, and to compare that to the current version and to see what has changed, see Sec. 2.5.4.
- You can also look at the Meta data called calTreeVersion, see Sec. 2.5.4.
- To load the calibration tree into HIPE when you pipeline process, see Sec. 2.5.3.

## 1.3.2. Re-processing with the pipeline scripts

The subsequent chapter of the *PDRG*, linked to below, cover different pipelines each.

The pipeline script you will run will depend on the observing mode,

- *Chopped point source data*: see Sec. ??? for observations taken in chop-nod photometry mode (an old mode).
- *Bright Point sources*: see Sec. ??? for observations containing bright point sources for single observations, and Sec. ??? to combine multiple observations (usually a scan and cross-scan).
- *Deep survey*: see Sec. 3.2.2 for deep survey observations containing many faint sources for a single observation, and Sec. 3.3 to combine multiple observations (usually a scan and cross-scan).
- *Extended source using high pass filtering and photproject*: see Sec. 3.2.3 for single observations, and Sec. 3.3 to combine multiple observations (usually a scan and cross-scan)
- *Extended sources using MADMap*: see Chap. 5 for observations of extended sources (only use when scan and cross scan data are taken).
- *Mini Scan-map*: see the pipelines for bright, deep or extended sources, whichever your source(s) is(are).
- There is no major difference in the pipeline between point-sources and extended sources for the high-pass filtering+photproject pipeline, only the parameters of the masking task will be different. MADMap is good to use for extended sources.
- The pipeline scripts contain all the pipeline tasks and simple descriptions of what the task are doing. But if you want to know all the details you need to consult the pipeline chapters (links above). Individual pipeline tasks are also described in the PACS *User's Reference Manual* (PACS *URM*).
- The pipelines take you from Level 0 (raw) to Level 2 (fully-processed), although in some cases you can just begin from Level 0.5. If a Level 2.5 is done, that means maps have been combined.

**To access the scripts**, go to the HIPE menu *Pipelines>PACS>Photometer*. The scripts assume:

- The data are already on disk or you can get them from the HSA using getObservation (so you must know the Observation ID)
- You have the calibration files on disk; normally you will use the latest update, but you can run with any calibration tree version: see Sec. 2.5.3 to know how to change the version of the calibration tree you are using.
- You chose to do the red or the blue camera separately

**To run the scripts**,

- Read the instructions at the top, and at least skim-read the entire script before running it
- Although you can run most all in one go, it is *highly* recommended you run line by line at least for the first time
- If you are going to comment within the script or change parameters, then first copy the script to a new, personalised location and work on that one (HIPE menu *File>Save As*): otherwise you are changing the script that comes with your HIPE installation

**As you run the scripts**,

- Plotting and printing tasks are included with which you can inspect the images and masks themselves. The plots will open as separate windows
- The scripts will save the data into FITS files after each Level (this is a difference with the spectroscopy pipeline)

## 1.3.3. Considerations when running the pipeline

Considerations concerning the technicalities of running the pipeline are:

- If you chose to run the pipeline remotely or as part of bulk processing you might want to disable the plotting tasks by commenting out the lines starting with "Display(...)"
- **Memory vs speed**: the amount memory you assign to HIPE to run the pipeline depends on how much data you have, but >=4Gb for sure is recommended.

  If you wish to fiddle with your data (other than using the plotting tasks provided in the pipeline) it would be a good idea to do that in a separate running of HIPE.
- Save your data at least at the end of each Level, because if HIPE crashes you will lose everything that was held only in memory (the scripts, by default save your data after each Level so DO NOT modify that part)

Things to look out for in your data as you run the pipeline are:

- Saturated and Glitched data
- Non-smooth coverage map (the coverage map is not uniform but the transitions should be fairly smooth towards the edges)
- Up and down scan offsets (distorted Point Spread Function)
- Dark spots around bright point sources (sign of inappropriate high-pass filtering)

## 1.3.4. Further processing

There are a number of tasks that can be used to inspect and analyse your PACS Level 2 images. For a first quick-look inspection (and even for some image manipulation) we recommend the tasks' GUIs.

The tasks are listed in the *Tasks* panel under Applicable if the image is highlighted in the Variables panel. Double-click on the task will call up its GUI, except for the Standard Image Viewer which is invoked by a right-click on the image in the Variables panel and selecting *Open with>Standard Image Viewer*

- To simple look at the images you can use the **Standard Image Viewer**: see the *DAG*: note that this section is still being written but you can see the *DAG* sec. 2.2.1.
- # The **annularAperturePhotometry task**: see the *DAG*: note that this section is still being written but you can see the *DAG* sec. 4.12. Performs aperture photometry using simple circular aperture and a sky annulus. There are other aperture photometry tasks: fixedSky, pacsAnnularSky, rectangular.
- # The **sourceExtractorDaophot and sourceExtractorSussextractor**: see the *DAG*: note that this section is still being written but you can see the *DAG* sec. 4.13. Extracts sources from a simple image using different algorithms.
- # The **sourceFitter**: see the *DAG*: note that this section is still being written but you can see the *DAG* sec. 4.14. Fits a 2D Gaussian to a source in a specified rectangular region on an image.

See the image analysis chapter of the **Data Analysis Guide** *chap. 4* for more information on image processing in HIPE.

# Chapter 2. Setting up the pipeline

## 2.1. Terminology

**Level 0** products are raw and come straight from the satellite. **Level 0.5** products have been partially reduced and corrected for instrument effects generally by tasks for which no interaction is required by the user. **Level 1** products have been more fully reduced, some pipeline tasks requiring inspection and maybe interaction on the part of the user. **Level 2** products are fully reduced, including tasks that require the highest level of inspection and interaction on the part of the user. **Level 2.5** products, which are found for some of the pipelines, are generally those where observations have been combined or where simple manipulations have been done on the data.

The *ObservationContext* is the product class of the entity that contains your entire observation: raw data, HSC-reduced products (levels), calibration products the HSC reduced with, auxiliary products such as telescope pointing, and etc. You can think of it as a basket of data, and you can inspect it with the Observation Viewer. This viewer is explained in the *HOG* sec. 15.2, and what you are looking at when you inspect a PACS *ObservationContext* is explained in Sec. 6.2.

The Level 2 (and also 2.5) photometry product is a *SimpleImage* that contains a standard two-dimensional image, in particular the following arrays: "image" as an array 2D (e.g. double, integer); "error" as an array 2D (e.g. double, integer); "exposure" as an array 2D (e.g. double, integer); "flag" as a short integer array 2D. It also contains Meta data that provide unit and World Coordinate System information. The definition of *Frames* give above is valid also for photometry. The photometry pipeline does not push the products into *ListContexts* as it does not use slicing.

To learn more about what is contained in the *ObservationContext, Frames, SlicedXXX*, and the various cubes, see Sec. 6.2.

The following (Help) documentation acronyms are used here: *DAG*: the Data Analysis Guide; *PDRG*: PACS Data Reduction Guide; *HOG*: HIPE Owner's Guide.

## 2.2. Getting and saving your observation data

### 2.2.1. Getting

The fastest ways to get the *ObservationContext* into HIPE were explained in Sec. 1.2. We expand on that here, but do first read Sec. 1.2. If you get your data via the HSA-GUI as a "send to external application" then it should be an *ObservationContext* already.

*If you have the data already on disk but as gotten from the HSA as a tarball*:

```
# on disk, untar the tarball, e.g.
cd /Users/me/fromHSA
tar xvf meme1342.tar
# look at it: ls meme1342

# in HIPE,
myobsid=1342..... # enter your obsid
mypath="/Users/me/fromHSA/me1342
myobs=getObservation(obsid=myobsid,path=mypath)

# obsid is necessary only if more than one observation
# is in that directory, i.e. if your tarfile has several
# obsids in it
```

*Get the data from the HSA directly on the command line:*

```
obsid = 134..... # enter your obsid here
```

```
# Direct I
#   Get the data from the HSA and then save to
#   /Users/me/.hcss/lstore/MyPoolName
myobs=getObservation(obsid, useHsa=True)
saveObservation(myobs, poolName="MyPoolName")
#   Then, to later get those data
myobs=getObservation(obsid, poolName="MyPoolName")
#
#   Get the data from the HSA and then save to
#   /Users/me/.hcss/lstore/[obsid as a string]
myobs=getObservation(obsid, useHsa=True)
saveObservation(myobs)
#   Then, to later get those data
myobs=getObservation(obsid)

# You must be logged on to the HSA for this to work:
# See the DAG sec. 1.4.5.
# See Sec. ??? to learn about saving and then
# restoring the caltree

# Direct II
#   Get the data from the HSA and immediately save
#   to /Users/me/.hcss/lstore/MyPoolName
myobs=getObservation(obsid, useHsa=True, save=True, poolName="MyPoolName")
#   Then to later get those data
myobs=getObservation(obsid, poolName="MyPoolName")
```

*Or if the data are on a pool on disk (e.g. have already been converted from HSA-format to HIPE-format*), you use:

```
# for data in [HOME].hcss/lstore/me1234
obsid=1342..... # enter your obsid
myobs=getObservation(obsid,path=mypath)
```

The full set of parameters for getObservation can be found in its URM entry: here. (Note: there are two "getObservation"s in the URM. The one I link you to is the correct one, it is also the first in the URM list.)

## 2.2.2. Saving

You use the task saveObservation for this, and to run this task with all the parameters set:

```
# To save in /Users/me/bigDisk/NGC1 where "bigDisk" is a replacement for
# the "local store" default location (see below)
pooln="NGC1"
pool="/Users/me/bigDisk"
saveObservation(obs, poolName=pooln, poolLocation=pool,
saveCalTree=True|False, verbose=True|False)
```

Where the only parameter you *need* to set is the "obs"—by default the data is saved to HOME/.hcss/lstore/[obsid as a string]. All other parameters are optional. The data will be saved to a pool (directory) located in the local store, whether that local store is the default HOME/.hcss/lstore or /Users/me/bigDisk as in the example above.

Or, as already mentioned above, you can save as you get the data:

Or, as already mentioned above, you can save as you get the data:

```
# Direct II
#   Get the data from the HSA and immediately save
#   to /Users/me/.hcss/lstore/MyPoolName
myobs=getObservation(obsid, useHsa=True, save=True, poolName="MyPoolName")
#   Then to later get those data
myobs=getObservation(obsid, poolName="MyPoolName")
```

You can save to anywhere on disk, though by default the data go to [HOME]/.hcss/lstore with a `poolName` that is the obsid (observation number) as a string. If the directory does not exist, it will be cre-

ated. If it does, then new data are added to it. Note that if you add the same obsid to the same pool a second time, then using getObservation later to get the *ObservationContext* will get you only the latest saved data. There is a parameter, `saveCalTree`, which is a switch to ask to save the calibration tree that is contained in the *ObservationContext* (myobs): True will save it, and the default False will not. Saving with the caltree takes up more space on disk and more time to work, but if you want to be able to access the calibration tree that the data were reduced with by the pipeline (either that which the HSA ran or that which you run), you should first attach the calibration tree to the *ObservationContext* and then set this parameter to True. If you have gotten the data just now from the HSA then the calibration tree will be attached. Otherwise, see Sec. ???.

Alternatively, the task getObservation also has a parameter that will save the data to disk, to your MyHSA, and including the calibration tree. See the URM entry to learn more, and see also the *DAG* sec. 1.4 to learn more about getObservation, used on data from the HSA or from disk.

# 2.3. What and where are the pipeline scripts?

In the following chapters we describe how to run the photometry pipelines that are offered via the HIPE *Pipeline* menu. In this chapter we explain the setting up of the pipelines. You will then skip to the chapter that is of the pipeline appropriate for your AOT.

All the photometry pipelines are standalone and provide a full processing of your data, with all the necessary steps required to produce a FITS image of your science target. Here we give a short summary of the purpose of each pipeline, although their names are quite self explanatory.

- *Chopped point source data*: see Sec. ??? for observations taken in chop-nod photometry mode (an old mode).
- *Bright Point sources*: see Sec. 3.2.1 for observations containing bright point sources for single observations, and Sec. ??? to combine multiple observations (usually a scan and cross-scan).
- *Deep survey*: see Sec. 3.2.2 for deep survey observations containing many faint sources for a single observation, and Sec. 3.3 to combine multiple observations (usually a scan and cross-scan).
- *Extended source using high pass filtering and photproject*: see Sec. 3.2.3 for single observations, and Sec. 3.3 to combine multiple observations (usually a scan and cross-scan)
- *Extended sources using MADMap*: see Chap. 5 for observations of extended sources (only use when scan and cross scan data are taken).
- *Mini Scan-map*: see the pipelines for bright, deep or extended sources, whichever your source(s) is(are).

The pipeline scripts can be found in the HIPE *Pipelines>Photometry* menu. Load and copy (*File>Save As*) to a unique name/location the pipeline script you want to run, because otherwise if you make changes and save the file, you will be overwriting the HIPE default version of that pipeline script. Henceforth, to load your saved script you will use the HIPE *File>Open File* menu. Read the instructions at the beginning of the script and at least skim read the entire script before running it. They are designed such that they can be run all in one go, after you have set up some initial parameters, but it is recommended that you run them line by line, so you have better control over them.

**We remind you here that you should consult the AOT release notes and associated documentation** before reducing your data. These inform you of the current state of the instrument *and the calibration*. Information about the calibration of the instrument will be important for your pipeline reductions—any corrections you may need to apply to your data after pipeline processing will be written here. Information about spectral leakages, sensitivity, saturation limits, and PSFs can also be found here. These various documents can be found on the HSC website, in the PACS public wiki: here.

> **Note**
>
> Spacing/tabbing is very important in jython scripts, both present and missing spaces. Indentation is necessary in loops, and avoid having any spaces at the end of lines in loops,

especially after the start of the loop (the if or for statement). You can put comments in the script using # at the start of the line.

# 2.4. How much can you improve on the automatic pipeline?

Before you being pipeline reducing the data yourself, it is a valid question to ask: how much can I improve on what I have already seen in the HSA-obtained Level 2 product? The answer to this depends on when the data you have were reduced by the automatic pipeline run by the HSC, and on the type of observation you have. The HSC pipeline lags behind the current (track 9/10) pipeline, and therefore some of the calibration files it used may be older than those in your version of HIPE (how to check is shown in Sec. 2.5.4). The HSC pipeline may also not run all the pipeline tasks (the pipeline is continually being updated and at the time of writing does not apply the flatfielding, for example). It is possible—especially for the simplest observations, of single grating and nod repetitions and with bright lines—that your reduction will not improve much on the HSC reductions, but for longer observations, with more repetitions and/or with faint lines, you will be able to improve, and you will have a better idea of what lies behind the final product. Some tasks really do require the user to check the results and decide if the parameters need changing: in particular the flatfielding and the subtraction of the off-source data from the on-source data.

# 2.5. Calibration files and the calibration tree

## 2.5.1. Installing and updating the calibration files

**First, you should consult the AOT release notes and associated documentation (e.g. Observer's Manual and Performance and Calibration documents),** these being important for informing you of the current state of the instrument *and the calibration*. Information about spectral leakages, sensitivity, saturation limits, ghosts and PSFs can also be found there. These various documents can be found on the HSC website, in the PACS public wiki: here.

The calibration files are not provided with the HIPE build, rather you are offered to chance to update them only when they need to be updated. If you open HIPE and you get a pop-up telling you to install the calibration products, it means that the calibration file set has been updated by the PACS team and you are being offered the chance to get that update. Click on "Install" and the new calibration products will be downloaded and installed. They are placed in [HOME]/.hcss/data/pcal-community (or pcal-icc, but only for the PACS team).

If this is the very first time you are using HIPE and hence you have never installed any calibration files before, then you should select "Install", otherwise you will have no calibration files at all. If you have done this before, and hence you do have a calibration file set, then you can chose whether to update or not. Why would you not? Well, if you are in the middle of processing data you may want to continue with the calibration files you are already using, rather than downloading new files and possibly having to start again (for consistency's sake), although just because you update does not mean you need to use the updated calibration tree: see Sec. 2.5.3 for information about how to set the calibration tree version you use in the pipeline.

## 2.5.2. Checking what has been updated

The updater GUI tells you which calibration files have been changed. To see the "update" information in the calibration updater pop-up, first click on "Show details..." and in the new tabs that pop up you can read the "Release Notes" summary of the new set version, and look at the individual "Files release notes" to see what (if anything) has changed in them. If more than one version number of calibration files (e.g. V13,...) are listed, you will be most interested in the highest version number.

To check on which pipeline tasks this will affect, check the the pipeline scripts which state which calibration files are used by the tasks that use calibration files.

The calibration files take up about half a gigabyte, so you may need to install them in a directory other than the default [HOME]/.hcss/data/pcal-community. If you want to install them elsewhere then: in the pop-up click "Not Now"; go to the HIPE Preferences panel (from the Edit menu); click on *Data Access>Pacs Calibration*; in the "Updater" tab that is now in the main panel change the name of the directory in the space provided. Do not click on anything else—you do want to use the "Community Server" as these are the products that have been tested, the "ICC" ones are still in the process of being validated. Click to Apply and Close. Then go to the Tools menu of HIPE, and select *pacs-cal>run Updater*. Voilà.

You can also inspect the calibration sets and products with a **Calibration Sets View**. This allows you to inspect the calibration sets that have been installed on your system. You get to this view via the HIPE menu *Window>Show View>Workbench>Calibration sets*. The view will show the release notes for the selected set (numbered boxes at the top), or the calibration file list for the selected set (viewing the notes or the file list are chosen via the central drop-down menu). The calibration file list is just a list of what calibration files, and their version numbers, are included in the selected set, and the release note you will see is the general one for that set. A new release of a calibration set will include some updated calibration files and also all the rest that have not changed.

## 2.5.3. The calibration tree

Before beginning the pipeline you will need to define the calibration tree to use with your reductions. The calibration tree contains the information needed to calibrate your data, e.g. to translate grating position into wavelength, to correct for the spectral response of the pixels, to determine the limits above which flags for instrument movements are set. The calibration tree is simply a set of pointers to the calibration files in your installation, it is not the calibration files themselves. Tasks that use calibration files will have the parameter `calTree`, which you set to the name you have given to the calibration tree (see below).

To use the latest calibration tree you have in your installation is done with,

```
calTree=getCalTree(obs=myobs)
```

Where "obs=myobs" is setting the parameter `obs` to the *ObservationContext* you are going to be working on, here called "myobs". This is done so that those few calibrations that are *time-specific* will take, as their time, the time of your observation.

If you want to reduce your data with an older calibration tree, you can do this simply by typing

```
calTree=getCalTree(version=13) # to use version 13
```

If you want to use the calibration tree that is with the *ObservationContext* (assuming it has been saved there), you type,

```
calTree=myobs.calibration
```

This will always be present if you have just gotten the data from the HSA, and will be present if whoever saved the *ObservationContext* remembered to save it with the calTree (see Sec. 2.6).

## 2.5.4. Comparing calibration file versions

To compare the version of the calibration files you will use by default when you begin pipeline processing your data, to those used by the HSC when the automatic pipeline was run, you do the following: where "myobs" is the name of the *ObservationContext*, type,

```
# The caltree that comes with you data
print myobs.calibration
print myobs.calibration.spectrometer
# The caltree you have on disk, this is the command that loads
# the calibration tree
#   that you will later use when you run the pipeline
```

```
calTree=getCalTree(obs=myobs)
# And to then inspect it
print caltree
print caltree.spectrometer
# Now you can compare all the version numbers that are printed
# to Console
```

The parameter `obs` (set to myobs here) simply specifies that the calibration tree will take the versions of the calibration files that are from the time that your observation took place, for those few calibration files which are time-sensitive.

*Note* that to print out the information on the calibration tree from "myobs" (the first command in the script above) it is necessary that the calibration tree is there in "myobs". This will be the case for HSC-pipeline reduced data if you have only just gotten it from the HSA and loaded it into HIPE. But if you used saveObservation to save it first to disk, or if you are looking at an *ObservationContext* someone gave you, then to get hold of the calibration tree of that *ObservationContext* it must be that the calTree was attached to and saved with the ObservationContext when running saveObservation. This is done by using the `saveCalTree`=True option, as explained in the next section. For this reason it may also be worth saving the calibration tree you will use when *you* reduce your data.

You can also check the calibration version your HSA-data were reduced with by looking at the Meta data "calTreeVersion" in the *ObservationContext*. This gives you the "v"ersion number of the calibration tree used to reduce those data,

```
print obs.meta["calTreeVersion"].long
```

To find out what version of HIPE your data were reduced with, check the Meta data called "creator", it will tell you something like SPG V7.3.0, which means Standard Product Generator in HIPE v 7.3.0.

# 2.6. Saving your *ObservationContext* and its calibration tree to pool

As stated previously, and repeated here, if you wish to save the calibration tree with your ObservationContext, then you should follow these instructions for the command-line methods:

```
obsid = 134..... # enter your obsid here

# example I: save with saveObservation to $HOME/.hcss/lstore/MyFirst
myobs=getObservation(obsid, useHsa=True)
saveObservation(myobs,poolName="MyFirst",saveCalTree=True)
# followed later by
myobs=getObservation(obsid, poolName="MyFirst")
calTree=obs.calibration

# example II: save when you get from the HSA
myobs=getObservation(obsid,useHsa=True,save=True,poolName="MyFirst")
# then later:
myobs=getObservation(obsid,poolName="MyFirst")
calTree=myobs.calibration

# via tarfile
file = "/Users/me/me10445555"
myobs = loadObs(file, obsid)
calTree=myobs.calibration
```

Why you would want to save the calibration tree? Whether you are saving data you got directly from the HSA, or data you have pipeline reduced yourself with the latest calibration tree, it is worth saving the fully-reduced *ObservationContext* with the caltree so that if you later wish to compare the reductions to later ones you do, you can at least check that the calibration trees are the same; and so that when you write up your results, you can find out which calibration tree you used. But otherwise you do not need to: the calibration files themselves are held on disc, all you need to know is the calibration tree version that was used to reduce the data.

# Chapter 3. In the Beginning is the Pipeline. *Photometry*

## 3.1. Introduction

The purpose of this and the next few chapters is to tutor users in running the PACS photometry pipeline. In Chap. 1 we showed you how to extract and look at Level 2 automatically pipeline-processed data; if you are now reading this chapter we assume you wish to reprocess the data and check the intermediate stages. To this end we explain the interactive pipeline scripts that have been provided for you, accessible from the Pipeline menu of HIPE. These are the scripts that you will be following as you process your data. The details of the pipeline tasks—their parameters, algorithms and the explanation of how they work—are given in the PACS *URM* (with software details). In Chap. 6 we explain issues that are slightly more advanced but are still necessary for pipeline-processing your data.

In the Pipeline menu the scripts are separated by the AOT type (e.g. mini scan map or chopped point source; although the chop-nod mode is no longer recommeded and we provide only the standard pipline script) and then by astronomical case (point source vs. extended source, e.g. scanmap_Extended_emission.py for a scanmap on extended emission). The difference in the pipeline between different astronomical cases (point source vs. extended source) is found only when using certain pipeline tasks, e.g. the filter width you will employ when removing the 1/f background noise. You will also see "Standard pipeline" scripts, which are those that the automatic processing (SPG) use, and these differ from the interactive in being older, less friendly to use, and based on the so-called slicing pipeline (slicing=splitting your observation up into sections based on their scan leg, or sequence in a repetition, or other similar criteria). We do not recommend that you try to run these.

When you load a pipeline script (it goes into the Editor panel of HIPE), copy it ("save to"), otherwise any edits you make to it will overwrite your reference version of that script! You can run the pipeline via these scripts, rather than entirely on the Console command line, in this way you will have an instant record of what you have done. You can then run it either in one go (double green arrow in the Editor tool bar) or line by line (single green arrow). This latter is recommended if you want to inspect the intermediate products and because you will need to make choices as you proceed.

> **Note**
>
> Spacing/tabbing is very important in jython scripts, both present and missing spaces. Indentation is necessary in loops, and avoid having any spaces at the end of lines in loops, especially after the start of the loop (the if or for statement). You can put comments in the script using # at the start of the line.

> **Note**
>
> Syntax: *Frames* are how the *PDRG* indicates the "class" of a data product. "Frame" is what we use to refer to any particular *Frames* product. A frame is also an image (a 2D array) corresponding to 1/40s of integration time.

*We remind you here that you should consult the AOT release notes and associated documentation* before reducing your data. These inform you of the current state of the instrument *and the calibration*. Information about the calibration of the instrument will be important for your pipeline reductions—any corrections you may need to apply to your data after pipeline processing will be written here. Information about sensitivity, saturation limits, and PSFs can also be found here. These various documents can be found on the HSC website, on the "AOT Release Status" link (currently here). Any "temporary" corrections you have to apply to your data are **not** detailed in this data reduction guide, as these corrections vary with time.

Before you run the pipeline you need to load the calibration tree that you want. What the "calibration tree" is, how you grab it, and how to change, check, and save the calibration tree are explained in Sec. 2.5.

# 3.2. Science case interactive pipeline scripts

Here we describe the various interative pipeline scripts that you are offered. We repeat: the actual pipeline tasks are described the PACS *URM* (with software details). Within the pipeline scripts we do explain where you need to set parameters yourself, but you can also read these other documents to learn more about the pipeline tasks and their parameters.

Because the chopped point source mode is no longer supported we do not offer an interactive pipeline script; in the Pipeline menu you will only find the Standard (SPG) scripts there. So from now on, all the scripts we describe are the scan map and mini map AOT.

The handling of data obtained in scan map mode depends strongly on the scientific goal. There are three distinguishable cases:

- Bright point (and slightly extended) sources
- Deep survey (faint point sources)
- Extended source
  - Phot project
  - MADMap

For extended sources we provide two processing modes; they give similar results so it is up to the user to decide which one to use. All the scripts can be found under the HIPE Pipeline menu (PACS#Photometer#Scan map and minimap). In the sections that follow we explain these three cases, however as the pipeline scripts differ only in certain places, *the full explanation is given only for the bright point source script, for the others you will be told how they differ but you will need to follow this next section, as you process your data, to understand what you are doing.*

## 3.2.1. Bright point source script

This script processes scan map and mini scan map observations containing bright point source(s) from Level 0 (raw data) up to Level 2 (final map). This script uses the high-pass filter method to remove the 1/f noise. This method is recommended for point sources (the usual case for mini scan map and deep fields) and relatively small extended sources; for extended sources it is not the ideal method as it removes large-scale structures, which cannot be properly protected (masked) from the filtering.

This script can be used to reduce a single obsid or it can be called in a loop. You would use the loop if you wanted to reduce several obsids (particularly of the same source). If you want to do this then you should first read the description here of the single-obsid pipeline, or even better actually follow it on a single obsid, so you understand what is going on, and then you can run the multiple-obsid description (Sec. 3.3).

The bright point source script is divided into sections. The user can comment or uncomment several sections to start the data reduction from different product Levels; they will have to do this when using the script in a loop (Sec. 3.3).

### 3.2.1.1. SECTION 0: setting up

This section sets up the obsid number, camera, hpfradius and some other parameters. These parameters will be required later in the pipeline. If the the user wants to use this script in a loop (Sec. 3.3) then these parameters need to be set in the multiple-obsid script. In this case the paremters settings in this scipt will be ignored. The input_setting variable is introduced to check whether the settings are already given in the multiple-obsid script in order to avoid overwriting. If input_setting is set to True the script will keep the settings given in the corresponding multiple-obsid script. If input_setting is set to False, the script will read the settings in this section.

```
try:
  input_setting
  print "settings given in the multiple obsid script"
except:
  input_setting=False
```

First you need to set the directory name where you want to save your files (and to read them). Then give your obsid number and camera that you want to reduce first.

```
direc = "/your_favourite_output_directory/"
obsid=0000000000  #give here the obsid number
camera = "red" # or "blue"
print "Reducing OBSID:", obsid, "camera:", camera
```

Then you set the output map pixel size: the choice of the output pixel size depends on the redundancy of your observation (the number of times a sky pixel is visited, which is almost always >1 because the detector pixels are larger than the recommended sky pixel sizes, and otherwise it is a factor of the scanning speed and scan leg overlap of your AOT). It also depends on the "drop size", here called "pixfrac" (see also Sec. ???) and Sec. 3.2.1.4). that you want for the drizzling method implemented in the task photProject, which is the task that actually creates your maps. Here we set the output pixel size to the recommended 2 arcsec for the blue channel and 3 arcsec for the red channel. These parameter values are found to work generally well with a drop size (pixfrac) of 1/10 of the input pixel size (this is a rather conservative value, but for the general case this provides a very good combination for reducing the correlated noise). You should use these values when you first reduce your data, because then later if you want to change them you have a "default" case to test against.

```
if camera=='blue':
 outpixsz=2.0
elif camera=='red':
 outpixsz=3.0
# and for both cameras
pixfrac=0.1
```

These values will be used in SECTION 3 (Sec. 3.2.1.4) of this script.

You also need to set the high-pass filter radius in number of readouts (Sec. 3.2.1.4). The numbers in the script are suited for point sources. Larger radii should be set for very bright or slightly extended sources with the caveat that the 1/f noise will then not be so well removed. As a general rule, the smaller the high-pass filter radius, the better you remove the 1/f noise. The suggested values allow one to remove as much 1/f noise as possible without removing the wings of point sources. Alternative values should be chosen on the basis of the source dimensions: for data taken at a medium scan speed (20 arcsec/s) a width of 20 is a good compromise for the blue camera and 30 for the red camera, thgese correspond to 40/60 arcsec on the sky respectively. At high speed (60"/s) a width of 10 can be used for the blue, corresponding to a length in the sky of 1 arcmin. As a rule of thumb the hpfradius should be as large as the source size. However, if the interpolation method is going used in the high-pass filter task, then hpfradius can be smaller then the source size. In general, values larger that 100 readouts are not recommended. The first time you run this script, adopt these values; you can test how the maps change with them later.

```
if camera=='blue':
  hpfradius=15
elif camera=='red':
 hpfradius=25
```

The hpfradius will be used in SECTION 3 of this scripts (Sec. 3.2.1.4).

Next you need to chose between masking options and chose radius values. This is a crucial point. This script shows you how to reduce the data via the high-pass filtering + PhotProject method. The high-pass filtering task removes from the signal a median value that is calculated within a box around each readout in the timeline. This means that if astronomical sources are not properly masked, they will be included in the median calculation and so some of the source signal will be removed and in the resulting map you will probably see troughs around the source. By masking you are providing to the

high-pass filtering task information about the location of sources, along the timeline. The masking radius values should be similar to the hpfradius value.

```
masking_radius=15. #arcsec
option = 3

# If you have already a mask of the sources, you can provide
# here the file name and choose the option=2
maskfile=direc+ "mask_file_name"
option = 2
```

Option: this scripts provides here several options, suggestions, and tricks that the user may want to try to optimise the data reduction for her/his own observation:

**option = 1** The observation is a mini scan map of a bright point source or a small extended source of known coordinates, the user might want to mask blindly all the pixels within a given radius around the source coordinates. In this case the script will show how to create the mask and attach it to the *Frames* before running the high-pass filtering task.

**option = 2** If the user has already a mask, derived from a pre-existing map at a different wavelength or maybe from a prior catalogue (for deep surveys), then she/he needs only to attach to the *Frames* the desired mask via a dedicated task, run then the high-pass filtering and the projection tasks. How to do this will be explained in the script.

**option = 3** The source is masked via a sigma threshold. This is for when the user needs to create the mask directly from the observation. One or more iterations will be required:

- First run the high-pass filtering task,
- create the final map via PhotProject,
- use this map to create a mask of the source(s),
- go back to the original (saved) Level 1 products (not high-pass filtered),
- attach the mask via the dedicated task,
- run again high-pass filtering task this time by providing the dedicated mask,
- re-run the projection.

In all the three cases the mask used to masking the sources is called "HighpassMask" throughout the script. We will discuss the details later in this section.

> **Tip**
>
> In the bright source case, the S/N of the source should be high enough to base the source mask on the individual map, from a single obsid, without the need to combine maps from many obsids. Combining many maps may be the case for deep surveys. Thus, if the user knows the size of the source in advance, we suggest they use option 1 and adjust the "masking-radius" parameter according to the source size. If the user does not know anything about the source size and precise position, we suggest they use the option 3 which allows one to mask everything above a given sigma threshold.

The option and masking_radius will be used in SECTION 3 of this script (Sec. 3.2.1.4). Here you simply set which option you will use.

## 3.2.1.2. SECTION 1: instrument corrections, taking you to Level 0.5

Now you have set your parameters you need to get the data, either from the HSA or from disc. How to get your data, your OservationContext, was explained in Sec. ???. If you will later want to compare the HSA-reduced results with your own you could copy the HSA-obtained *ObservationContext* to a

pool that has a unique name (e.g. "NGC111_from_the_HSA") using saveObservation (Sec. ???, and do use the `saveCalTree` parameter), so you can load that into HIPE once you have reduced the data yourself, and compare them.

In this section the data are reduced up Level 0.5. After this level the auxiliary data are not needed and the Level 0.5 product is saved in a FITS file.

> **Tip**
>
> sometimes it can happen that previously stored data in the local pool conflicts with a new download. For instance it might happen that an observation reduced without problem the first time can not be reduced anymore because part of the data, e.g. the auxiliary data, can not be retrieved anymore. If this happens, it might be useful to remove or (simply move to somewhere else) the .hcss/pal_cashe/ and the /favorite_local_store_directory/has_cashe/ directories, and repeat the data reduction.

Now, you can extract the *Frames*, which contains the raw data that you will process, from the *ObservationContext* called "obs"

```
if camera=='blue':
  frames=obs.level0.refs["HPPAVGB"].product.refs[0].product
elif camera=='red':
  frames=obs.level0.refs["HPPAVGR"].product.refs[0].product
```

Then we need to check the observing mode and the scanspeed in which the observations were carried out so we can set the limits for the FilterOnSpeed task to select frames at constant velocity.

```
if obs.cusMode=="SpirePacsParallel":
   speed = frames.meta["mapScanRate"].value
   if speed=="slow":
      lowscanspeed  = 15.
      highscanspeed = 25.
   elif speed == "fast":
      lowscanspeed  = 54.
      highscanspeed = 66.
elif obs.cusMode=="PacsPhoto":
   speed = frames.meta["mapScanSpeed"].value
   if speed=="medium":
      lowscanspeed  = 15.
      highscanspeed = 25.
   elif speed =="high":
      lowscanspeed  = 54.
      highscanspeed = 66.
```

If you observe a Solar Sytem object you need some special corrections in the astrometry due to the movemnt of the objects duriing the observation. So you need to check whether your object belongs to the Solar System or not and set the sso variable accordingly:

```
sso = isSolarSystemObject(obs)
```

Grab the "pointing product" from obs: this contains all the information about where PACS was pointing during your observation:

```
pp = obs.auxiliary.pointing
```

Then load the calibration tree for this observation (see also Sec. ???). The calTree contains all the calibration files needed for the data processing. The setting "time=frames.startDate" ensures that the correct calibration files are attached to your observation. In particular, the so-called SIAM calibration file, which is necessary for the pointing calibration, changes with time. The SIAM product contains a matrix which provides the position of the PACS bolometer virtual aperture with respect to the spacecraft pointing. The "date" of the observation is needed to attach the correct SIAM to the data.

```
calTree = getCalTree(time=frames.startDate)
```

Then extract housekeeping parameters (contains information about PACS itself)

```
photHK=obs.level0.refs["HPPHK"].product.refs[0].product["HPPHKS"]
```

Finally get the orbit ephemeris, and the horizons Product which are necessary for the correct aberration correction when calculating the astrometry

```
oep = obs.auxiliary.orbitEphemeris
horizons = None
horizonsProduct = obs.auxiliary.horizons
```

(horizons = None is so set because this product is not yet ready. It is however necessary for solar system objects: see Sec. ???.

Now you have everything you need to start processing your data. This little section of the script collects all the tasks you need to go from Level 0 up to Level 0.5. The tasks do not need any interaction by the user. These tasks need as input data taken from the auxiliary products you just grabbed. After this section the auxiliary products are not needed for the further data processing, and you can delete them to save on memory. Thus, before saving "frames" in a FITS file for future processing, we recommend that you reach at least Level 0.5. In this level of the pipeline you will also remove the calibration block from the data.

First you need to identify the blocks in the observation. A block is simply a chunk of the data stream where the instrument settings remain constant and so those readouts can be grouped together. This task creates a BlockTable from which the user can check the structure of the observation. The BlockTable is better explained in Sec. 6.3.3.

```
frames = findBlocks(frames, calTree=calTree)
```

Then remove the calibration block and keep only the science frames.

```
frames = detectCalibrationBlock(frames)
frames = removeCalBlocks(frames)
```

Flag the known bad pixels

```
frames = photFlagBadPixels(frames, calTree=calTree)
```

If you have your own bad pixel mask and you want to insert it in the calTree, you can define your own calTree (mycal):

```
mycal=getCalTree()
fa=FitsArchive()
bad_pixel_prod=fa.load("/home/my_own_bad_pixel_mask.fits")
mycal.photometer.badPixelMask=bad_pixel_prod
```

The product bad_pixel_prod, defined above, contains the bad pixel mask you want to use. The last line puts this mask in the bad pixel mask space in mycal. If you want to use this mask in the photFlag-BadPixels task you simply call the task with your calibration tree as input:

```
frames = photFlagBadPixels(frames, calTree=mycal)
```

The phenomenon of electronic crosstalk was identified, in particular in the red bolometer (column 0 of any bolometer) subarray, during the testing phase and it is still present in the in-flight data. We reccommend to flag those pixels in order to remove artifacts from your map. For this purpose we use the task photMaskCrosstalk.

```
frames = photMaskCrosstalk(frames)
```

Now flag saturated pixels

```
frames = photFlagSaturation(frames, calTree=calTree, hkdata=photHK)
```

Convert from ADU to Volts

```
frames = convertChopper2Angle(frames, calTree=calTree)
```

Compute the coordinates for the reference pixel (detector centre) including the aberration correction

```
frames = photAddInstantPointing(frames,pp,orbitEphem = oep)
```

In case you object is a Solar System object, the processing of the observations requires some extra attention since the execution of one individual observation does not account for moving objects in real time, but recentres the telescope with each new pointing request. Thus, by default, the assignment of pointing information to the individual *Frames* assumes a non-moving target. Here we present a little piece of code that can be included in any photometer processing script to take into the account the motion of a SSO during the observation.

```
if (sso) :
  frames = correctRaDec4Sso(frames, timeOffset=0, orbitEphem=oep,  \
     horizonsProduct=horizonsProduct, linear=0)
```

At this stage the auxiliary products can be deleted since they will not be used anymore

```
del (pp,photHK,oep,horizons)
```

Now you can safely save the Level 0.5 *Frames* you have created (called frames) in a FITS file. You can also save to pool or even to the ObservationContext, but saving to FITS is easier to explain. We propose a very simple output filename convention based on the obsid number and the channel (camera). You can choose to change the proposed name convention in your favorite way. In this case, pay attention to edit also the parts of the scripts where the files are read back to continue the data processing.

```
savefile = direc+"frame_"+"_" + str(obsid) + "_" + camera + "Level_0.5.fits"
simpleFitsWriter(frames,savefile)
```

## 3.2.1.3. SECTION 2: flatfielding

If you have a saved Level 0.5 product you can start your processing here. You can then obviously skip SECTION 1. You can use the following code to read your files saved at the end of SECTION 1.

```
savefile = direc+"frame_"+"_" + str(obsid) + "_" + camera + "Level_0.5.fits"
frames=simpleFitsReader(savefile)
calTree = getCalTree(time=frames.startDate)
```

The only step in this section is applying the flat-field and convert Volts into Jy/pixel

```
frames = photRespFlatfieldCorrection(frames, calTree = calTree)
```

At this point you got to the Level 1 product, which is calibrated for most of the instrumental effects. You might want to save this product before running the high-pass filter task, so you can go back and forth as you try to optimise your data reduction. The script stores your data as a FITS file in the output directory "direc" which you set in SECTION 0. We propose here the same name convention used for saving the Level 0.5 products.

```
savefile = direc+"frame_"+"_" + str(obsid) + "_" + camera + "Level_1.fits"
simpleFitsWriter(frames,savefile)
```

## 3.2.1.4. SECTION 3: mask, filter out the noise, deglitching, make the map

This section begins with the Level 1 product and reduces the data up to Level 2. It is here that the "option" that were you asked to select in SECTION 0 will be used. The user can choose to start the data reduction from this section if the Level 1 data were saved in FITS file. This can be easily done by commenting out SECTIONs 1 and 2 and by uncommenting two lines of code where the FITS file of the Level 1 data is read. The script assumes you saved the Level 1 product with the name convention suggested above. Please change the savefile variable if you used a different convention.

```
savefile = direc+"frame_"+"_" + str(obsid) + "_" + camera + "Level_1.fits"
frames=simpleFitsReader(savefile)
```

```
calTree = getCalTree(time=frames.startDate)
```

At the end of this section the final map is saved in FITS file.

At this point we arrive at the issue of masking, and we offer several masking methods. We give a little introduction at the beginning of the section and here we will provide a detailed description of each masking option:

- **OPTION=1**: the mask HighpassMask is defined by masking everything within a given radius from the known source coordinates.

  The following command defines the "on_source" as a cube of the same dimension as frames but with value 0 (NOT masked) at the readouts with coordinates outside the given radius (masking_radius) from the source coordinates, and with value 1 at the readouts within the radius. So the task translates between the coordinate frame and the time frame that the data are held in. The masking_radius is set to 15 arcsec in the script (Sec. 3.2.1.1). You should change the radius according to the dimension of the source, following the advice previously given. A look at the Level 2 map provided by the automatic pipeline can help in defining the region to mask. We propose here also an automatic way to get the source coordinates from the Meta data. However, because of a pointing errors, you should check from the Level 2 map that these coordinates correspond to the source center. For this option, running the task phot AssignRaDec is mandatory.

```
if option == 1:
  frames = photAssignRaDec(frames, calTree=calTree)
  rasource  = obs.meta["ra"].value
  decsource = obs.meta["dec"].value
  cosdec=COS(decsource*Math.PI/180.)
  on_source=SQRT(((frames.ra-rasource)*cosdec)**2+(frames.dec-decsource)**2)\
    < masking_radius/3600.
```

  the following commands show you how do define, attach and set the "HighpassMask" in frames

```
  if (frames.getMask().containsMask("HighpassMask") == False):
     frames.addMaskType("HighpassMask","Masking the source for High pass")
     frames.setMask('HighpassMask',on_source)
```

  **A TIP** We do not recommend using MMT deglitching in this mode however if you decided to use it then be aware that in many cases the MMT deglitching task can detect very bright sources as glitches. If this is your case it is often useful to disable the deglitching mask within the on-target readouts. To do that you can use the 'HighpassMask' which provides information on the source location.

```
  mask=frames.getMask('MMT_Glitchmask')
  off_source=SQRT(((frames.ra-rasource)*cosdec)**2+(frames.dec-decsource)**2)\
     > masking_radius/3600.
  frames.setMask('MMT_Glitchmask',mask & off_source)
```

- **OPTION=2**: the user has already a mask called HighpassMask.

  In this option the script assumes the user has already a mask called HighpassMask. The file containing the mask is loaded as "maskfile"

```
elif option == 2:
  mask=simpleFitsReader(maskfile)
```

  the following task attaches the mask to the frames.

```
frames  = photReadMaskFromImage(frames, si=mask, \
   extendedMasking=True, maskname="HighpassMask")
```

  **A TIP** Similarly to option == 1, if you decided to use MMT deglitching you can also disable the deglitching on target in this option but a slightly different way.

```
  maskMMT=frames.getMask('MMT_Glitchmask')
```

```
  maskHPF=frames.getMask('HighpassMask')
  frames.setMask('MMT_Glitchmask',maskMMT & (maskHPF == 0))
```

- **OPTION=3**: create the mask from the final map.

  In this option the user creates the mask from the final map, i.e. from the data themselves. A preliminary data reduction is performed without masking the sources in the high-pass filter task. The mask is created using the first preliminary map. The data reduction from Level 1 is repeated but this time using the mask. So the first step is high-pass filetring without mask.

```
elif option == 3:
  frames = highpassFilter(frames,hpfradius)
```

  The next command selects only those readouts taken while the telescope is slewing at a constant speed, i.e. it removes the turnover loops between scan legs. It does it using the scanspeed and the limits defined in Section 3.1.2

```
  frames =
 filterOnScanSpeed(frames,lowScanSpeed=lowscanspeed,highScanSpeed=highscanspeed)
```

  Then a first preliminary projection is done using default settings, to create the preliminary map.

> **Tip**
>
> Deglitching is not absolutely necessary at this stage. Glitches are very short and affect just one or two pixels. In addition, bright sources could be easily misidentified as glitches due to the effect of the high-pass filtering residuals. On the other hand, the presence of many glitches could affect the mask quality. The user should check if this is the case.

```
  map = photProject(frames, calTree=calTree, calibration=True)
  Display(map)
```

  At this point a mask can be created. We propose here a simple method to create the mask. The key point is to define a threshold above which signal is identified as being from a source. To define this threshold we propose the following steps:

  - identify the regions of the map with high coverage: calculate the 1-sigma stddev value of the coverage, including in this all the map pixels where the coverage is >0 (this includes all pixels where the flux is >0 and exludes the edge regions of you maps, where Herschel was doing a turnover between scan legs).

```
    med=STDDEV(map.coverage[map.coverage.where(map.coverage > 0.)])
```

  - use this region to estimate the signal standard deviation (stdev)

```
    signal_stdev=STDDEV(map.image[map.image.where(map.coverage > med)])
```

  - define the threshold as cutlevel*stdev: we set the cutlevel to 3.0 here

```
    cutlevel=3.0
    threshold=cutlevel*signal_stdev
```

  - mask everything above that threshold

```
    mask=map.copy()
    mask.image[mask.image.where(map.image > threshold)] = 1.0
    mask.image[mask.image.where(map.image < threshold)] = 0.0
    Display(mask)
```

  To disable the MMT deglitching on target, follow the example of OPTION=2.

Masking is one of the key points for obtaining a good quality final map. An optimal masking method does not really exist because the masking depends strongly on the scientific case. Thus, we invite you

to play with the methods proposed here by changing e.g. the cutlevel parameter to understand how to mask properly your sources. In the case of deep field survey it can be convenient to smooth the preliminary map (e.g. with a gaussian smoothing) in order to avoid masking many individual noisy pixels. (Smoothing an image is a HIPE-general task, rather than a PACS-specific one).

If many individual pixels in the noisy region with low coverage are over-masked, you might want to unmask those pixels in the following way.

```
mask.image[mask.image.where(mask.coverage < med)] = 0.0
```

At this point the data reduction can be started again from Level 1 and the high-pass filter task can be executed with the proper mask.

```
savefile = direc+"frame_"+"_" + str(obsid) + "_" + camera + "Level_1.fits"
frames=simpleFitsReader(savefile)
```

The next task masks in the timeline all the readouts at the map coordinates where the signal is above the threshold (bona fide sources).

```
frames  = photReadMaskFromImage(frames, si=mask, \
  extendedMasking=True, maskname="HighpassMask")
```

The parameter `extendMasking` is this: in some cases, for example if you have many one-pixel-only masked points, the mask cannot be written perfectly back in to frames. This is because a singl frames (detector) pixel is projected on to several map pixels. If only one map pixel is masked, writing it back into the frames object may be inaccurate.Extension of the masked region" that this parameter sets may solve the problem by extending the mask to neighbouring pixels. (Its default value is False.)

Now the high-pass filter task can be executed with the proper mask.

**AN IMPORTANT TIP ON HOW TO SET THE HPFRADIUS** We will set the interpolateMaskValues to True for the cases where the hpfradius is smaller than the mask radius: i.e. if you masked-out sources are larger than the hpfradius. If the hpfradius is smaller than the source size, then when the task gets close to your masked source it will calculate the median over the given hpfradius including parts of your masked area, and this will result in the removal of some source flux. In these case the "interpolateMaskedValues" parameter should be set to True to let the task interpolate between the closest values of the median over the timeline but jumping over the masked area.

The hpfradius is set in SECTION 0, where we put all the settings. The default parameter is 15 readouts for the blue channel and 25 for the red channel. This value is usually adopted in the case of deep surveys with medium speed of 20"/s. The 15 readouts used in the blue channel corresponds to a radius of 30", and the 25 readouts used in the red channel corresponds to a radius of 50" for this scan speed. Extensive analysis of the effects of the high-pass filtering on the PSF (i.e. on the wings of point sources) have shown that these values of the hpfradius allow one to remove most of the 1/f noise without damaging the point source profile. Indeed, the effect is to remove part of the external lobes of the point source profile without affecting the core, where 95% of the light is enclosed. Smaller values for hpfradius would remove also part of the flux in the core. Thus, we suggest you do not use smaller values than those proposed here. If the observation was performed at a different speed or the source is particularly big or bright, you are invited to enlarge the hpfradius to be as large and the source size. We do not recommend values larger than 100 and 150 readouts for the blue and red camera, respectively, since this would not allow one to properly remove the 1/f noise.

```
frames  = highpassFilter(frames,hpfradius,maskname="HighpassMask", \
  interpolateMaskedValues=True)
```

We then need to remove the turnover loops and other non-constant speed parts of the data, as seen before

```
frames=filterOnScanSpeed(frames,lowScanSpeed=lowscanspeed,highScanSpeed=highscanspeed)
```

At this point we perform the deglitching using a method called "second level deglitching". The concept of the second level deglitching method is very simple and it is explained in more detail in Sec. 4.3. The method builds up the contributions of all input pixel to the output pixel and flags the outliers as glitches. The tasks involved in this step are IIndLevelDeglitchTask and MapIndexTask. In few word, MapIndexTask collects the information about the contributions of the individual input pixels to any output pixel and IIndLevelDeglitchTask flag the outliers and eventually builds the map. These tasks, together, uses exactly the same algorithm of PhotProject and the same functionalities with the addition of flagging the outliers. As mentioned, the IIndLevelDeglitchTask can also produce a map. However, due to the large amount of memory used and also the difficulty in controlling the large number of input parameters, we strongly suggest to keep the second level deglitching and the projection itself separated. Indeed, performing the projection within the IIndLevelDeglitchTask does not always reduce the amount of time needed for the data reduction and can be memory expensive.

How the MapIndexTask works? The way PhotProject and, thus, this task work is to estimate on how many output pixels the input pixel is falling and to estimate the geometrical weights, i.e. the fractional area of the input pixel overlapping the different output pixels. So MapIndexTask collects for any output pixel the fluxes, pixel coordinates and time index (position in the timeline) and, if desired, also the geometrical weights of all input pixels contributing to it. With the option slimindex=True the MapIndexTask stores the least possible info about the input pixel contributions (just fluxes and coordinates) and let the IIndLevelDeglitchTask performs only the sigma clipping to flag outliers. With the option slimindex=False the MapIndexTask stores all info and makes the IIndLevelDeglitchTask able to perform the full projection. As you can imagine the amount of information stored in the memory by this task is huge especially is the parameter slimindex is set to False and all information, including the geometrical weights, are stored in memory. For this reason we suggest to keep deglitching and projection separated. If you want to perform deglitching and projection in one go, we suggest you to divide your observation in chunks and deglich/project the separated chunks in order to not go out of memory. You can find all information how to do that by setting several MapIndexTask & IIndLevelDeglitchTask input parameters in Chapter 6 of the PDRG.

Here is how you call the deglitching tasks

```
from herschel.pacs.signal import MapIndex
from herschel.pacs.spg.phot import MapIndexTask
from herschel.pacs.spg.phot import IIndLevelDeglitchTask
mi = MapIndexTask()
iind = IIndLevelDeglitchTask()
```

Since we want to perform only the sigmaclip to flag outliers without creating a map yet, we set the slimindex value to True.

```
mapToCubeIdx = mi(frames,slimindex=True)
```

Before calling the IIndLevelDeglitchTask task we can set the algorithm for performing the sigma-clipping and flag the outliers. To do that we define the input parameters of the Sigclip() task. The first input (nsigma) is the level of sigma where we want to define the threshold for the sigma-clipping. We set here our threshold at 10 sigma. The parameter behavior is set to "clip" in order to perform a sigma clipping in one go. Alternatively, you might set it to filter, to use a box filter (see the chapter 6 of the PDRG for more details) . The parameter "outliers" is set to both in order to let the IIndLevelDeglitchTask task flag positive and negative outliers. The parameter "mode" define the method used to perform the sigma clipping. The default is MEAN and the standard deviation is used to define the threshold. Here we use MEDIAN since it is less affected by outliers. In this case the Median Absolute deviation is used to define the threshold. Since the Median Absolute Deviation can be much smaller than the standard deviation, be aware that the value of nsigma has to be higher when using the MEDIAN mode with respect to the MEAN mode.

```
s = Sigclip(nsigma=10,behavior="clip",outliers="both",mode=Sigclip.MEDIAN)
```

Let's deglitch the data! Now we have all in hands and we can perform the deglitching. Since we do not want to perform the projection and only obtain a mask with the deglitched readouts, we set map=False and mask=True. The mask SecondGlitchmask, as defined in the maskname parameter will be attached to the input frames.

```
deg =
 iind(mapToCubeIdx,frames,map=False,mask=True,maskname='SecondGlitchmask',algo=s)
```

If you prefer to let IIndLevelDeglitchTask task perform deglitching and projection in one go, you need to set the the map parameter to True. In this case, since we set the slimindex=True and no geometrical weight are stored by the MapIndexTask task, those will be re-calculated on the fly by the IIndLevelDeglitchTask exactly as photProject would do. To do that you just need to use the following line and omit the projection done by PhotProject in the next line

```
map =
 iind(mapToCubeIdx,frames,map=True,mask=True,maskname='SecondGlitchmask',algo=s)
```

IIndLevelDeglitchTask might need a large amount of RAM. If this becomes a problem for your data processing, you might want to try a similar method, which uses a significantly smaller amount of memory, but it is more time-consuming. This alternative task is called MapDeglitchTask and is similar in concept to the IIndLevelDeglitchTask. The main difference between the two is that MapDeglitchTask does not store information in memory, thus, reducing the need of memory. The commands needed to call this task are listed below. We suggest to use a nsigma parameter equal to 30 for an effective deglicthing. IMPORTANT NOTE:

> **Note**
>
> be aware that for executing the MapDeglitchTask task there is no need to execute MapIndexTask task.

```
from herschel.pacs.spg.phot.deglitching.map import MapDeglitchTask
s = Sigclip(nsigma=30,behavior="clip",outliers="both",mode=Sigclip.MEDIAN)
mdt = MapDeglitchTask()

deg = mdt(frames,deglitchvector='timeordered',maskname='SecondGlitchmask',algo=s)
```

And finally we project to create our final map

```
  map=photProject(frames, outputPixelsize=outpixsz, calTree=calTree, \
    pixfrac=pixfrac)
```

The photProject task performs a simple coaddition of images using the drizzle method (*Fruchter and Hook, 2002, PASP, 114, 144*).

**NOISE**: There is no particular treatment of the signal in terms of noise removal. The 1/f noise is supposed to have been removed by the high-pass filtering task. The key parameters of the photProject task are the the output pixel size and the drop size (pixfrac). A small drop size can help in reducing the correlated noise that you get due to the projection itself (for a quantitative treatment, see the appendix in *Casertano et al. 2000, AJ, 120,2747*). Bear in mind that any 1/f noise not removed by the high-pass filter task is still a source of correlated noise in the map. Thus, the formulae provided by Casertano et al. 2000, which account only for the correlated noise due to the projection, do not provide a real estimate of the total correlated noise of the final map. Indeed, this is a function of the high-pass filter radius, the output pixel and the drop size. Nevertheless, those formulae can be used to reduce the correlated noise at least due to the projection.

The `outputPixelsize` and `pixfrac` parameters values were set in SECTION 0. We set the output pixel size to 2 and 3 arcsec for the blue and the red camera. The pixfrac parameter is expressed as the ratio between the drop and the input pixel size. A drop size of 1/10 the input pixel size is found to give very good quality in the final map.

We stress here that the values of output pixel size and drop size (pixfrac) depend strongly on the redundancy of the data (e.g. the repetition factor). For instance, a too small a drop size value would create holes in the final map if the redundancy is not high enough (see *Fruchter and Hook, 2002*). The values proposed here for the output pixel and drop size are intended to be optimal for a standard mini scan map. Since these parameters have to be adjusted case by case, we invite you to play with them to find the right balance between noise, correlated noise and S/N of the source for creating the optimal map.

**A TIP** you might want to use a specific WCS for your final map (world coordinate system). Here we show how to set the WCS parameters.

```
  ra_reference=0.000
  dec_reference=0.000
```

The reference pixel set here is at the the center of the map (so you change the 0.000 to your own central decimal postions). To set these parameters you should have an idea of the map dimensions for the given output pixel size. We propose an automatic way to retrieve the centre of the map from the Meta data. However, you might want to check that from the Level 2 map provided by the automatic pipeline. Setting a fixed WCS could be very useful if your observation comprises many obsids and you want to produce maps all with the same WCS (e.g. so you can mosaic or add them together). In addition, the standard deviation of the output pixel signal can provide a good estimate of the noise. In this way you can obtain a meaningful noise map associated to your final map, which is still not otherwise available as an output of the current pipeline.

Build the WCS:

```
pixsize=outpixsz
rad1=240
rad2=240

my_wcs=Wcs(cunit1="Degrees",cunit2="Degrees",cdelt1=-pixsize/3600.,\
      cdelt2=pixsize/3600.,crota2=0.,crpix1=rad1,crpix2=rad2,\
      crval1=ra_reference,crval2=dec_reference,ctype1="RA---TAN",\
        ctype2="DEC--TAN", equinox=2000.0)
my_wcs.setParameter("naxis1",2*rad1,"naxis1")
my_wcs.setParameter("naxis2",2*rad2,"naxis2")
```

where rad1 and rad2 are half the dimension of the map in the x and y direction in pixel units. To set the wcs in photProject you need just to set the wcs parameterin photProject:

```
map=photProject(frames, outputPixelsize=outpixsz, calTree=calTree, \
  pixfrac=pixfrac,wcs=my_wcs
```

Finally you can save your map as a fits file:

```
outfile = direc+ "map_"+"_" + str(obsid) + "_" + camera + ".fits"
print "Saving file: " + outfile
simpleFitsWriter(map,outfile)
```

## 3.2.2. Deep survey maps

By "deep survey" what we mean is looking for very faint sources. In fact you follow the same script as for point source with three exception. So you should follow Sec. 3.2.1 to understand what the deep survey script is doing. In this section we just describe the differences between the two cases.

- the masking options are a bit different than in the case of bright point sources
  - **Option 1:** If your scientific case is a deep survey of a blank field, chances are that your sources will not be detectable on the single obsid map due to the low S/N. We suggest to perform a preliminary reduction without masking any source. You need to reduce all obsids, combine the individual maps and base the mask on a first preliminary mosaic (see the multiple_obsid_scanmap_Deep_survey_miniscan_ Pointsource.py script for the details how to make the mosaic). Although the preliminary map will show strong high-pass filtering residuals, you can use it to obtain a good mask of the sources. You can, then, reduce once again the individual obsid by using the mask based on the much deeper mosaic. This option offers to use a "unmasked" high-pass filtering for obtaining a preliminary map for building the mosaic.

    > **Note**
    >
    > USE THIS OPTION ONLY FOR A PRELIMINARY DATA REDUCTION TO OBTAIN A FIRST MOSAIC TO GET THE MASK.

> **Tip**
>
> for this particular usage of the highpassFilter task, we suggest to use a relatively large hp width in order to limit the damages of the sources. A smaller hp width can be used for the defnitive reduction when the highpassFilter is used with a mask.

```
frames  = highpassFilter(frames,hpfradius)
```

After you created your map, follow the method described in the previous section to create a mask from your data.

- **Option 2:** If the user has already a mask derived from a preexisting map at different wavelength or from a preliminary reduction, then she/he needs only to attach to the frames the desired mask via a dedicated task, run then the high-pass filtering and the projection tasks. This option is exactly the same as Option 2 in the case of bright point sources

- **Option 3:** In this option the script assumes the user provides an ascii file with a list of sky coordinates to create a mask with circular patches of a given radius (specified by the user) at the given coordinates. The radius of the circular patches should be as large as the FWHM of the PSF to limit the damages due to the highpassFiletr task. The radius provided to MaskFromCatalogueTask task must be a 1d array. In this way the user has the possibility to specify different radii for different sources. This is particularly useful if the deep field contains few slightly extended sources (nearby galaxies). If radius contains just one value, that is used for all sources of the prior catalog. The radius is expressed in arcsec.

```
if input_setting==False:
     file='sky_coordinates.txt'
     ascii=AsciiTableTool()

 ascii.template=TableTemplate(2,names=["ra","dec"],types=["Double","Double"])
     ascii.parser=FixedWidthParser(sizes=[10,10])
     table=ascii.load(file)
     radius=Double1d(1)
     ra=table["ra"].data
     dec=table["dec"].data
     radius[0]=9.0
```

Here the sky_coordinates.txt contains the ra and dec values of the center of the patches in decimal degree format and we assume that the radius of the patches is 9.0 arcsec. if you want to specify a different radius for each source, load, instead, a catalog with the following format, where the third column is the radius of the patches in arcsecond.

```

 ascii.template=TableTemplate(3,names=["ra","dec","radius"],types=["Double","Double","Double"])
     ascii.parser=FixedWidthParser(sizes=[10,10,4])
```

The task needs in input also a map with the correct WCS. For this purpose we use the map Level2 product taken from the observation context.

```
map=obs.refs["level2"].product.refs["HPPPMAPR"].product
```

Now you can create your mask using the MaskFromCatalogueTask

```
from herschel.pacs.spg.phot import MaskFromCatalogueTask
mfc = MaskFromCatalogueTask()
mask = mfc(map, ra, dec, radius, copy = 1)
```

the following task attaches the mask to the frames.

```
frames  = photReadMaskFromImage(frames, si=mask,
 extendedMasking=True,maskname="HighpassMask")
```

and now you can run the masked high-pass filter

```
frames  = highpassFilter(frames,hpfradius,maskname="HighpassMask",
 interpolateMaskedValues=True)
```

- we recommend a slightly different parameter sets for the photProject taks

  Instead of using

```
map=photProject(frames, outputPixelsize=outpixsz, calTree=calTree, \
   pixfrac=pixfrac
```

  one should use

```
map=photProject(frames, outputPixelsize=outpixsz, calTree=calTree,\
   weightedsignal=True, pixfrac=pixfrac)
```

  The only difference is the parameter `weighted signal` being set to True. The photProject task comes in two flavours: a simple average of the input pixel contributions to the output pixel, as done in the first call example above, or a weighted mean of those contributions, as should be done here. The weights are estimated as the inverse of the square error. *However, since noise propagation is not yet done properly in the PACS pipeline, a proper error cube must be provided to obtain good maps*. We stress here that this method can not rely on the errors provided by the pipeline, so we propose a trick to estimate a reliable error cube to perform the weighted mean. *However, be aware that this trick will work only for background-noise dominated maps*, as should be the case for deep surveys. For the mini scan map, if you are observing a bright source then this method will not produce correct results.

  The error cube proposed here is intended to be used only for the estimate of the weights in the weighted mean of photProject, it is not a real error map that you can use in your science analysis. The error for each readout is estimated as the stdev in a box as large as 10 times the hpfradius, centered on the readouts along the *timeline*. To avoid the occurrence of NaNs, for readouts where the error is zero (e.g. no coverage) the error is set manually to 1.e6 (to get negligible weights).

```
noisecube=Condense(2,10*hpfradius,STDDEV, "boxcar")(frames["Signal"].data)
noisecube[noisecube.where(noisecube==0)] = 1e6
frames.setNoise(noisecube)
```

  Now you can continue and run photProject.

- we recommend MMT deglitching instead of second level deglitching

  The main difference from the previous script is the place and method of the deglitching. Here you need to apply the deglitching before the flatfielding right after level 0.5. So basically the first step in Section 2 is deglitching the data with MMT method. This task works pretty well for deep surveys without using an enormous amount of memory as for the IIndLevelDeglitchTask. It deglitches the data along the time domain, i.e. looks for outliers along the detector pixels time-line rather than along the map spatial grid. We provide here the parameters that are found to work generally well for this specific scientific case.

```
frames = photMMTDeglitching(frames, incr_fact=2,mmt_mode='multiply', \
  scales=3, nsigma=5)
```

> **Tip**
>
> if your map contains very bright sources then this task may misidentify those sources as glitches. You can verify this case by looking at the coverage part of the Level 2 map produced by later-explained task photProject (you can use e.g. DS9 on the FITS image if you save the map to disc, or within HIPE you can use
>
> ```
> Display(map.coverage)
> ```
>
> This does mean that you will have to continue processing your data to the photProject stage (Sec. 3.2.1.4) before you can look at the map. If the MMT task has deglitched

any astronomical sources, you will find holes at the position of the source peaks in the coverage map. There are several tricks to solve this problem:

- **trick 1** you can provide the MMT task a "source mask" which lets it know where the sources are (it sets the regions that are source to a mask value of 1: IS masked and hence DO avoid). You can obtain this mask with the same method described below for creating the high-pass filter mask (Sec. 3.2.1.4), but with a somewhat higher cut level so you mask only bright sources. Then you need to read in the FITS file containing the source mask (if it is not already stored in memory):

```
source_mask=simpleFitsReader("file_name_of_source_mask.fits")
```

  and finally you provide this mask as an additional input to photMMTDeglitching:

```
frames = photMMTDeglitching(frames, incr_fact=2,
 mmt_mode='multiply', \
  scales=3, nsigma=5, sourcemask=source_mask)
```

  You can then run once again the task with different "relaxed" parameters to flag also the glitches on source. If you do not change the name of the mask created by the MMT task, the mask will be updated by adding the newly detected glitches. If you prefer to keep the two masks separated you can change the name of the mask by adding the additional input maskname="new_mask_name" when calling photMMT-Deglitching. You can, then, re-run the task with new and less stringent parameters to deglitch also the sources without damage

```
frames =
 photMMTDeglitching(frames,scales=2,nsigma=9,mmt_mode='multiply',incr_fact=2)
```

- **trick 2** as in the previous trick, you can provide the MMT task a "source mask" created the same way. In order to then deglitch within the astronomical sources (after all, they may also have glitches in them) you can use the second level deligthing task (this is explained in Sec. 3.2.3), and for this you can use the reverse of this mask so that this second level deglitching is done on the sources only.

  You can check the masks with the MaskViewer

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

> **Tip**
>
> another way to check for deglitching of sources is to project the mask to see which regions of the real map have been masked. This is a bit time consuming but is very useful if you have a preliminary map, it allows for a direct check on whether real sources have been systematically deglitched. To do this exercise we copy "frames" to a new name ("frames_masked") and replace its signal with the mask values. Then we remove the mask under consideration (MMT_GlitchMask), otherwise you will get an empty map, and then we project:
>
> ```
> frames_masked = frames.copy()
> objectMask =
>  frames_masked.getMask('MMT_Glitchmask').copy()
> frames_masked.setSignal(Double3d(objectMask))
> frames_masked =
>  deactivateMasks(frames_masked,String1d(['MMT_Glitchmask']))
> map_mask =
>  photProject(frames_masked,calTree=calTree)
> Display(map_mask)
> ```
>
> you could delete the frames_masked and the map_mask after the check
>
> ```
> del(frames_masked,map_mask,objectMask)
> ```

# 3.2.3. Extended sources

This script is also very similar to the the bright point source so we explain here only the differences. You will need to follow Sec. 3.2.1 to understand what the extended source script is doing. For large extended sources we recommend MADmap processing: see Chap. 5.

Basically the only difference with respect to the bright point source script is that we need to prepare our data using high pass filtering with a large hp radius to avoid removal of the large scale structures. So, in SECTION 0 of the script we will set two hpfradius values for each camera: a larger one for preparation and a smaller one for the final filtering

```
if camera=='blue':
  hpfradius_1=150
elif camera=='red':
  hpfradius_1=250
#
# second hp filter pass
#
if camera=='blue':
 hpfradius_2=25
elif camera=='red':
 hpfradius_2=40
```

The run the high-pass filtering without a mask using a large radius and filter on scan speed to remove the turnover loops as we did in the previous cases

```
frames = highpassFilter(frames,hpfradius_1)
frames=filterOnScanSpeed(frames,lowScanSpeed=18.0,highScanSpeed=22.0)
```

Now we can perform the second level deglitching:

```
from herschel.pacs.signal import MapIndex
from herschel.pacs.spg.phot import MapIndexTask
from herschel.pacs.spg.phot import IIndLevelDeglitchTask
mi = MapIndexTask()
iind = IIndLevelDeglitchTask()
```

since we want to perform only the sigma clipping to flag outliers, without also creating a map, we set the `slimindex` value to True.

```
mapToCubeIdx = mi(frames,slimindex=True)
```

Before calling the IIndLevelDeglitchTask task we can set the algorithm we want to use for the sigma-clipping and flagging the outliers. To do that we define the input parameters of the Sigclip() task. The first input (`nsigma`) is the level of sigma that is the threshold for the sigma-clipping. We set here our threshold at 10 sigma. The parameter `behavior` is set to "clip" in order to perform a sigma clipping in one go. Alternatively, you might set it to filter, to use a box filter (see Sec. ???). The parameter `outliers` is set to both in order to let the IIndLevelDeglitchTask task flag positive and negative outliers. The parameter `mode` defines the method used to perform the sigma clipping. The default is MEAN and the standard deviation is used to define the threshold. Here we use MEDIAN since it is less affected by outliers. In this case the Median Absolute deviation is used to define the threshold. Since the Median Absolute Deviation can be much smaller than the standard deviation, be aware that the value of `nsigma` has to be higher when using the MEDIAN rather than the MEAN .

```
s = Sigclip(nsigma=10,behavior="clip",outliers="both",mode=Sigclip.MEDIAN)
```

Now, let's deglitch the data! Since we do not want to perform the projection and only obtain a mask with the deglitched readouts, we set `map`=False and `mask`=True. The mask SecondGlitchmask, as defined in the maskname parameter, will be attached to frames.

```
deg =
 iind(mapToCubeIdx,frames,map=False,mask=True,maskname='SecondGlitchmask',algo=s)
```

If you prefer to let IIndLevelDeglitchTask task perform deglitching and projection in one go, you need to set the the `map` parameter to True. In this case, since we set `slimindex`=True and no geometrical weights are stored by the MapIndexTask task, those will be re-calculated on the fly by the IIndLevelDeglitchTask, exactly as photProject would do. To do that you just need to uncomment the following line and comment the projection done by PhotProject in the next line

```
map =
 iind(mapToCubeIdx,frames,map=True,mask=True,maskname='SecondGlitchmask',algo=s)
```

Now we can use PhotProject to create a first preliminary map using default settings. This will be used only to create the mask for the high-pass filtering.

```
map = photProject(frames, calTree=calTree,calibration=True)
Display(map)
```

From this point on the processing is identical with the bright point source case if you choose option 3 for mask making. The only deference is that the default threshold for extended sources should be a little bit lower: 2 sigma instead of 3. The second run of high-pass filtering should be performed using the smaller radius (hpfradius_2)

# 3.3. Multiple obsids

For multiple obsids that you wish to combine, you can process all the data in one wrapper script. This script sets up the necessary parameters and then calls whichever pipeline script (point source, extended source, etc) you want to use.

Since the observations in scan map mode are designed such that the same area is covered by two separate passes of the telescope (scan and cross-scan), it is always the case that one has to process multiple obsids using the same method and these then need to be combined into one map. In this case you need to use one of the following three scripts: multiple_obsid_scanmap_BrightPointsource.py, multiple_obsid_scanmap_Deep_survey_miniscan_Pointsource.py, multiple_obsid_scanmap_Extended_emission.py. The scripts can be found in /scripts/pacs/spg/pipeline/ipipe/phot/, located off of where-ever the HIPE software is on your computer. Thse scripts are capable of handling any number of obsids in a loop, and combining the results into a single map. We explain here multiple_obsid_scanmap_Deep_survey_miniscan_Pointsource.py as an example the others are very similar.

NOTE: if you have copied any of these pipeline scripts to a new directory and you are running from your copy, then the execfile command below will change. Note down the place where you have placed your copy and what name you gave it. The easiest way is to put every script into one directory and set a variable at the begining of the script containing the name of this directory:

```
scriptsDir = Configuration.getProperty("var.hcss.dir") + "/scripts/pacs/spg/
pipeline/ipipe/phot/"
```

First you need to set the input_settings parameter to True so the script reads the settings here and will ingnore any other setting given in the individual obsid script. If the input_setting variable is set to False, the script will read the settings in the Section 0 of the individual obsid script.

```
input_setting=True
```

In the example above the "Configuration.getProperty("var.hcss.dir")" simply reads the directory where your HIPE is installed. So your script directory in this case is where all the ipipe scripts are located.

Then set the other parameters that you would set in the singel obsid scripts. The syntax is exactly the same, there is only one difference: instead of giving only one obsid you give and array of obsids in the following way

```
obsidall=[obsid_number1,obsid_number2,...]
```

Then start the loop

```
for i in range(len(obsidall)):
    obsid= obsidall[i]
    print "Reducing OBSID:", obsid, "camera:", camera
    execfile(scriptsDir+'scanmap_Deep_Survey_miniscan_Pointsource.py')
```

this creates one map for each obsid as a FITS file. See the script for the names of these files, and don't change the names otherwise the rest of the script will return with an error. NOTE: if you copied the script you are now running (the multiple obsid script called multiple_obsid_scanmap_Deep_survey_miniscan_Pointsource.py here) and/or the script you wish to process each obsid with (scanmap_Deep_Survey_miniscan_Pointsource.py here), then you may have to change the exefile command. This command executes a script that is located in the same directory that the script you are running from is located.

At this point a preliminary map, still with the high-pass filtering residuals, is available for any obsid. If you did not provide a common WCS for all maps (Sec. 3.2.1.4), you can combine them using the MakeMosaicTask as we show here. The script assumes that you used the same name convention adopted in the individual obsid script for saving the maps in FITS files. If you used a different name convention, please, edit the script and change the map_file variable that contains the map file names. If you did provide a common WCS you can in fact still use the MaskMosaicTask (you can also use other options, but we do not explain that here).

```
from java.util import ArrayList
from herschel.ia.toolbox.image import MosaicTask

images=ArrayList()

for i in range(len(obsidall)):
 obsid=obsidall[i]
 map_file = direc+ "map_"+"_" + str(obsid) + "_" + camera + ".fits"
 map=simpleFitsReader(map_file)
 map.exposure=map.coverage
 images.add(map)
```

the variable "images" will the contains all the individual maps you want to combine. The mosaic is done in the following way:

```
mosaic=MosaicTask()(images=images,oversample=0)
Display(mosaic)
```

**AN IMPORTANT TIP** The MakeMosaic task relies on the error maps provided by the pipeline to estimate the error map of the final mosaic. We stress here that the error propagation of the PACS pipeline is still under construction and that the current available error maps produced by the pipeline are not reliable. To create a reliable error map in the final mosaic we suggest you project (create) all the individual maps with the same WCS. The way to set the WCS is explained in Sec. 3.2.1.4, and how to force PhotProject to use the desired WCS is described in the individual obsid script that you execute. Once the maps are created with the same WCS, the final map can be obtained by taking a simple average or weighted mean (weighted by coverage) of the individual output pixels. The standard deviation of the mean or weighted mean can provide a meaningful final error map. If the number of obsid is not high enough to let you estimate the mean (weighted mean) and standard deviation, you can slice the single obsid frames per repetition or odd and even scan legs to produce a sufficient number of individual maps to accurately estimate mean (weighted mean) and standard deviation.

If you chose the option = 3 in the script that you executed, and you are now happy with the result, you can simply save the final mosaic and skip the rest of the script.

```
outfile_name = direc+ "final_map"+ "_" + camera + ".fits"
print "Saving file: " + outfile_name
simpleFitsWriter(mosaic,outfile_name)
```

If you chose the option = 1, you just did a quick-look data reduction. Thus, you need to proceed to improve the data reduction with the following scheme. You can use the preliminary mosaic to detect

also the faint sources and create the mask. You need then to re-reduce the data starting from Level 1, and used the new mask to perform the masked high-pass filtering and projection. Change then your option!

We propose here the same method used in the single obsid script for masking the sources. Be aware that here we use the exposure and not the coverage because the MakeMosaicTask produces an exposure map that has the same meaning as the coverage map. So the steps of the mask making are the following:

```
med=STDDEV(mosaic.exposure[mosaic.exposure.where(mosaic.exposure > 0.)])
signal_stdev=STDDEV(mosaic.image[mosaic.image.where(mosaic.exposure > med)])
cutlevel=3.0
threshold=cutlevel*signal_stdev
mask=mosaic.copy()
mask.image[mask.image.where(mosaic.image > threshold)] = 1.0
mask.image[mask.image.where(mosaic.image < threshold)] = 0.0
Display(mask)
```

you can save the mask in a FITS file to be used for any further re-processing

```
maskfile = direc+ "mask"+ "_" + camera + ".fits"
simpleFitsWriter(mask,maskfile)
```

if you commented out SECTION 0 of the individual obsid script, the maskfile name will be passed to the script for the masked high-pass filtering.

If you are satisfied by the mask (to check you can display the mosaic and the mask and blink between the two, e.g. with DS9 on the FITS files or with Display in HIPE), then you can re-run the final processing by providing to the individual obsid script the new source mask and choosing option =2. We suggest you comment out the whole of SECTIONs 0, 1, and 2 in the individual obsid script and restart the data processing from the Level 1.

```
option = 2

for i in range(len(obsidall)):
    obsid= obsidall[i]
    print "re-reducing OBSID:", obsid, "camera:", camera
    execfile('scanmap_Deep_Survey_miniscan_Pointsource.py')

from java.util import ArrayList
from herschel.ia.toolbox.image import MosaicTask

images=ArrayList()

for i in range(len(obsidall)):
 obsid=obsidall[i]
 map_file = direc+ "map_"+"_" + str(obsid) + "_" + camera + ".fits"
 map=simpleFitsReader(map_file)
 map.exposure=map.coverage
 images.add(map)

mosaic=MosaicTask()(images=images,oversample=0)

Display(mosaic)
```

> **Tip**
>
> The MakeMosaic task relies on the error maps provided by the pipeline to estimate the error map of the final mosaic. We stress here that the error propagation of the PACS pipeline is still under construction and that the current available error maps produced by the pipeline are not reliable. To create a reliable error map in the final mosaic we suggest to project all the individual obsid maps with the same wcs. The way to set the wcs is suggested in the SECTION 0 of this script and how to force PhotProject to use the desired wcs is described in the individual obsid script. Once the maps are created with the same wcs, the final map can be obtained by taking a simple average or weighted mean (weighted by coverage) of the individual output pixels. The standard deviation of the mean or weighted mean can provide a meaningful final error map. If the number of obsid is not high enough to let you estimate the mean (weighted mean) and standard deviation, you can slice the

> single obsid frames per repetition or odd and even scan legs to produce a sufficient number of individual maps to accurately estimate mean (weighted mean) and standard deviation.

If you chose the option = 3 and you are now happy with the result, you can simply save the final mosaic and skip the rest of the script.

```
outfile_name = direc+ "final_map"+ "_" + camera + ".fits"
print "Saving file: " + outfile_name
simpleFitsWriter(mosaic,outfile_name)
```

If you chose the option = 1, you just did a quicklook data reduction. Thus, you need to proceed to improve the data reduction with the following scheme. You can use the preliminary and deep mosaic to detect also the faint sources and create the mask. You need, then, to re-reduce the data, starting from Level 1, and used the new mask to perform the masked high-pass filtering and projection.

We propose here the same method used in the single obsid script for masking the sources. Be aware that here we use the exposure and not the coverage because the MakeMosaicTask produce only the exposure map with the same meaning of the coverage map. This method follows the following steps:

1. identifying the region of the map with high coverage: this is done by taking the region where the coverage is above 1sigma

   ```
   med=STDDEV(mosaic.exposure[mosaic.exposure.where(mosaic.exposure > 0.)])
   ```

2. using this region to estimate the signal standard deviation (stdev)

   ```
   signal_stdev=STDDEV(mosaic.image[mosaic.image.where(mosaic.exposure > med)])
   ```

3. defining the threshold as cutlevel*stdev: we set the cutlevel equal to 3.0 by default.

   ```
   cutlevel=3.0
   threshold=cutlevel*signal_stdev
   ```

4. masking everything above the threshold

   ```
   mask=mosaic.copy()
   mask.image[mask.image.where(mosaic.image > threshold)] = 1.0
   mask.image[mask.image.where(mosaic.image < threshold)] = 0.0
   Display(mask)
   ```

> **Tip**
>
> masking is one of the key point for obtaining a good quality final map. The optimal masking method does not exist. The masking method strongly depends on the scientific case. Thus, we invite you to play with the method proposed here by changing the cutlevel parameter and to try the tips below to undertand how to mask properly your sources.
>
> 1. If many individual pixels in the noisy region with low coverage are over-masked, you might want to unmask those pixels in the following way.
>
>    ```
>    mask.image[mask.image.where(mask.exposure < med)] = 0.0
>    Display(mask)
>    ```
>
> 2. Alternatively you might want to smooth your map, e.g. with a Guassian filter, before masking in order to avoid masking many individual noisy pixels. In the example below we smooth the image with a Gaussian filter and perform the masking on the smoothed image
>
>    ```
>    nsigma=1.0
>    gf=GaussianFilter(nsigma)
>    mask=map.copy()
>    mask.image=gf(map.image)
>    mask.image[mask.image.where(map.image > threshold)] = 1.0
>    mask.image[mask.image.where(map.image < threshold)] = 0.0
>    ```

If also after smoothing the image, the pixels in the low coverage region are over-masked and you are unhappy about that, you can also use the tip number 1 to unmask those pixels.

You can save the mask in a fits file to be used for any further re-processing

```
maskfile = direc+ "mask"+ "_" + camera + ".fits"
print "Saving file: " + maskfile
simpleFitsWriter(mask,maskfile)
```

if you commented properly the SECTION 0 of the individual obsid script, the maskfile name will be passed to the script for the masked high-pass filtering

> **Tip**
>
> the cut level to use in order to identify the sources depends on the scientific case. Generally a 3sigma level is sufficient for the deep survey/mini scan map case. You can also chose to refine the method by recalculating iteratively the noise and the threshold by avoiding the masked pixels. Indeed, if you discard the sources in the map the standard deviation should be less affected by the source flux and should be smaller.

If you are satisfied by the mask, then you can re-run the final processing by providing to the individual obsid script the new source mask by choosing option =2. We suggest to comment the whole section 0, 1, and 2 in the individual obsid script and restart the data processing from the Level 1.

```
for i in range(len(obsidall)):
    obsid= obsidall[i]
    print "re-reducing OBSID:", obsid, "camera:", camera
    execfile(scriptsDir + 'scanmap_Deep_Survey_miniscan_Pointsource.py')
```

If you did not provide a common wcs for all maps, you can simply combine them by using the MakeMosaicTask to obtain the final map as in the following few lines of code. The script assumes that you used the same name convention adopted in the individual obsid script for saving the maps in fits files. If you used a different name convention, please, edit the script and change the map_file variable that contains the map file name.

```
from java.util import ArrayList
from herschel.ia.toolbox.image import MosaicTask

images=ArrayList()

for i in range(len(obsidall)):
 obsid=obsidall[i]
 map_file = direc+ "map_"+"_" + str(obsid) + "_" + camera + ".fits"
 map=simpleFitsReader(map_file)
 map.exposure=map.coverage
 images.add(map)

mosaic=MosaicTask()(images=images,oversample=0)

Display(mosaic)
```

Now you can finally save your final map in a fits file:

```
outfile_name = direc+ "final_map"+ "_" + camera + ".fits"
print "Saving file: " + outfile_name
simpleFitsWriter(mosaic,outfile_name)
```

# Chapter 4. Selected topics of data reduction. *Photometry*

## 4.1. Introduction

The main purpose of this chapter is to give a deeper insight into the fine-tuning of some of the pipeline tasks to achieve better results, and to provide a detailed description of advanced scripts used for different scientific purposes. It is recommended that you read this chapter after you have familiarised yourself with the basic reduction steps of the data processing of the PACS photometer (see Chap. 5).

This chapter is divided into:

- The second level deglitching
- Reducing mini-maps
- Solar System objects

## 4.2. 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.

## 4.3. Second level deglitching

Second level deglitching is a glitch-removal technique that works on the final map of PACS photometer data. Second level deglitching collects the data contributions to each map pixel in one dimensional data arrays (we call these "vectors", following mathematical conventions). Unlike the first level deglitching (such as the MMT technique), which works by clipping on the vectors of each *detector* pixel (and hence operating along the temporal dimension), the second level deglitching works on the *map/sky* pixels (and hence on the spatial plane). The detector pixels contain all the data taken over the course of the observation, but, because during an observation PACS (and hence all of its detectors) is moving around the sky, any single detector pixel will contain data for a range of projected map/sky pixels. The idea behind the deglitching is that without glitches all the signal contributions to a map pixel should be roughly the same. Glitches will introduce significantly outstanding values and can thus be detected by sigma-clipping. See this figure for an example:

**Figure 4.1. Vector of roughly 250 signal contributions to map pixel (85, 98)**

### 4.3.1. Pre-requisites and what is 2nd level deglitching?

To apply second level deglitching you need a *Frames* object that contains PACS photometer data. The *Frames* object should be processed up to the stage where the data can be mapped (i.e. to the end of

Level 1). For a proper deglitching it is especially important that flux calibration has been done using photRespFlatfieldCorrection.

The actual deglitching process takes place in two steps:

1. Calculate the data contributions to each map pixel and store them in a vector for every map pixel. This data is collected in an object called *MapIndex*.

2. Use the *MapIndex* object to loop over the data vectors and apply sigma clipping to each of the vectors. This is done by the task IIndLevelDeglitchingTask.

# 4.3.2. Command-line syntax

Here is the most basic application of second level deglitching. The syntax is the jython command line syntax. Lines starting with "#" are comments and usually don't have to be executed. During the startup of HIPE, most of the import commands are carried out automatically; these imports are:

```
# import the software classes
from herschel.pacs.signal import MapIndex
from herschel.pacs.spg.phot import MapIndexTask
from herschel.pacs.spg.phot import IIndLevelDeglitchTask
```

Then you continue with constructing the tasks, which can be done with the following syntax:

```
mapIndex = MapIndexTask()
IIndLevelDeglitch = IIndLevelDeglitchTask()
```

Note that this is not the only way to call on these tasks—as you use HIPE more and more you will learn yourself all the variations on a theme. What we have done here is set up a sort of alias for the task, useful if only so you have less to write (you could write "mit = MapIndexTask(), for example, for a really short alias). You need the *Frames* (which will be called "frames" here) coming out of the pipeline to run mapIndex (i.e. the MapIndexTask). The output of the task is a *MapIndex* object which we will call “mi”:

```
mi = mapIndex(frames)
```

The deglitching is the second step of the processing

```
map = IIndLevelDeglitch(mi, frames)
```

Now we will describe these tasks and their parameters.

# 4.3.3. The most important syntax options

The second step—map = IIndLevelDeglitch(mi, frames)—can be fine-tuned to optimise the deglitching process. The most significant values to specify are these:

1). Do you want to produce a map as output or only flag the glitches and write the flags in form of a mask back into the *Frames*? The options to do this are `map` = True or False, and `mask` = True or False. By default, both options are True (produce a map and also produce a mask).

2). You may customise the deglitching parameters by specifying a sigma clipping algorithm and using it as an input in the deglitching task. To set up the algorithm:

```
s = Sigclip(nsigma = 3)
```

defines a new Sigclip algorithm, with a 3-sigma threshold. You can set other parameters of this algorithm now in the following way:

```
s.setOutliers("both")
```

tells the algorithm to detect both positive and negative outliers. Alternatives are "positive" (default for positive glitches only) and "negative". Another parameter is:

```
s.setBehavior("clip")
```

telling it to apply the clip to the data vector in one go, i.e. don't use a box filter. If you prefer to use a filter, use

```
s.setBehavior("filter")
s.setEnv(10)
```

where 10 means that the boxsize will be 2*10+1 = 21 vector positions long. Another parameter

```
s.setMode(Sigclip.MEDIAN)
```

defines the algorithm used for detecting outliers: either median (with median absolute deviation: our recommendation) or mean (with standard deviation: Sigclip.MEAN). You can find more details in the Sigclip documentation (this particular algorithm is part of the general HIPE software; you can look at it, and others, in the PACS *URM*, the HCSS *URM* and the *SaDM*).

Now apply the newly configured Sigclip algorithm to the deglitching:

```
map = IIndLevelDeglitch(mi, frames, algo = s, map = True, \
  mask = False)
```

An interesting option is also algo=None. This does not apply any deglitching, it simply co-adds the data in the vector. This way, it creates an undeglitched map from the MapIndex (which you could, if you wanted to, compare to the degliched map). This is a relatively fast algorithm. So if you already have a MapIndex and just want to map, using the algo=None option is faster than the alternative map-making task PhotProject. If you don't specify the Sigclip alorithm, the default for second level deglitching is used, which is: clip with nsigma = 3, use a median algorithm, and both outliers (more or less what we have specified above). You may test your Sigclip parameters interactively with the MapIndexViewer, so you don't have to guess the best Sigclip parameters. Please read the description of the MapIndexViewer (Sec. ???) further down to learn how this is done.

```
mi = mapIndex(frames)
```

The deglitching is again the second step of the processing, and can be called simply as:

```
map = IIndLevelDeglitch(mi, frames)
```

# 4.3.4. A detailed look at the MapIndex task

The *MapIndex* is a 3-dimensional data array that has the same width and height as the resulting map. The third dimension contains references to the data in the *Frames*, so that every flux contribution to a map pixel can be retrieved from the *Frames*. In other words, the 3rd dimension contains information about where the data of each map pixel came from.

This third dimension is non-rectangular, because the number of detector pixel contributions differs from map pixel to map pixel. If you want to retrieve the data from the *MapIndex*, it is returned as an array of *MapElements*. Please have a look at the image to see what kind of data is stored in a *MapIndex*:

**Figure 4.2. MapIndex and MapElements**

The layout of the *MapIndex* determines the layout of the map itself. This is why the MapIndexTask uses many of the options that are also used by the photProject task (which is the task for a simple projection and was used in Chap. 5).

The most important parameters of the MapIndexTask and how to use them:

- `inframes`: the input *Frames*.
- `outputPixelsize`: this is the desired size of the map pixel in arcseconds (a square geometry is assumed). By default it is the same size as the *Frames* data array (3.2 arcsecs for the blue photometer and 6.4 for the red).
- `pixfrac`: this is the fraction of the input pixel size. If you shrink the input pixel, you apply a kind of drizzle. pixfrac should be between 0 (non inclusive) and 1. A value of 0.5 means that the pixel area is reduced to 0.5*0.5=0.25 of the original area, ie 1/4th of the unmodified pixelsize.
- `optimizeOrientation`: set this value to True if you want the map rotated for an optimised fit of the data and thus smallest mapsize. Of course, after rotation North may no longer be pointing upwards. By default this parameter is set to False.
- `wcs`: for a customised World Coordinate System. By default the wcs is constructed for a complete fit of the data. If you want a special wcs, for example if you want to fit the map into another map afterwards, you may specify your own wcs. A good starting point for this would be the Wcs4MapTask, which creates the default wcs for the MapIndexTask (and also PhotProject). You can take the default wcs and modifiy it according to your needs; find the full set of options in the documentation of the wcs (the PACS *URM* and the HCSS *URM*). For example:

```
from herschel.pacs.spg.phot import Wcs4mapTask
wcs4map = Wcs4mapTask()
wcs = wcs4map(frames, optimizeOrientation = False)
#the above gives the default world coordinate system (wcs)
#you can reset this:
wcs.setCrota2(45)  #rotate the map by 45 degees
wcs.setNaxis1(400) #force the map to 400X400 pixels.
wcs.setNaxis2(400)
wcs.setCrpix1(200) #The data may not fit anymore—make sure the centre of
wcs.setCrpix2(200) #your data remains in the centre of your map
```

- `slimindex`: this is a memory-saving option. Please read details in Sec. ???. In a nutshell, slimindex stores the least possible amount of data in the *MapIndex* (hence, a "slim" *MapIndex*), saving on memory use. This means that as you create a map, some values have to be recalculated on-the-fly during the deglitching task, which costs processing time. As a rule of thumb: use slimindex=True if you want to create only a glitchmask (map=False, mask=True in the deglitching). For this, the slim-MapIndex contains all necessary data. If you want to create a map use slimindex=False. This will enlarge the size of the *MapIndex* product from a slim-MapIndex to a full-MapIndex containing all bells and whistles—this can be heavy on memory use, but it will deglitch and map more quickely. If you don't have much memory, you may safely use slimindex=True and also create a map. The default value is slimindex=True.

The full call of the MapIndexTask with the described options looks like this:

```
mi = mapIndex(frames, optimizeOrientation = False, wcs = mywcs,\
   slimindex = False)
```

As soon as you execute the MapIndexTask, you will realize that it needs a lot of memory. There are reasons for this and it helps to understand the concept of the MapIndex in order to find the most efficient way to process your data.

# 4.3.4.1. Memory-saving options

## The size of the MapIndex

If we want to store the full amount of data needed to project the flux values from the *Frames* product to the map, this is what we need for every flux contribution to a map pixel:

- the row, column and time index of the data in the *Frames* product
- the relative overlapping area of the *Frames* pixel that falls onto the map pixel (= the poids or weight value)
- the size of the *Frames* pixel in units of the map pixel size

Since this is more than one value, the set of information is put into a *MapElement*, which is a container. The *MapElement* is what you get from the *MapIndex* when you want to extract this information.

This full set of information is contained in a *MapIndex* product called the *FullMapIndex*. The *FullMapIndex* is one of the two flavours a *MapIndex* can have. You get it when you use the non-default option slimindex=False in the MapIndexTask.

Now, what is the size? Assume a *Frames* product with 30000 time indices and a blue array with 2048 detector pixels. Assume further that every *Frames* pixel overlaps 9 map pixels when the flux is projected. We get a MapIndex size of nearly 12 gigabyte storing this information. Tweaks are obviously necessary!

Besides compressing the data while it is stored, the most significant tweak is the slim MapIndex (the second flavour of the MapIndex. The java class is called *SlimMapIndex*). This contains a reduced set of information, namely only the row, column (encoded in one value: the detector number) and time information of the data. While the *FullMapIndex* uses 12 gigabyte per hour of observation, the *SlimMapIndex* needs only 2 gigabytes.

## What can you do with the slim MapIndex: working with little memory

It is possible to perform the second level deglitching without creating a map. For the default deglitching, without the **timeordered** option, no information about pixel sizes or overlaps are necessary because only the values from the *Frames* product without weighting are used. In this way a glitch mask can be created.

Although the second level deglitching task can also create a map out of the slim MapIndex, the necessary information about pixel sizes and overlaps have to be recalculated on-the-fly. This is inefficient. So, with the slimIndex the best mapping strategy is

- deglitch and create only the glitch mask and no map
- use the regular projection with PhotProject to create a map. PhotProject will take into account the glitch mask

## Iteratively deglitch large observations

Another way to handle large, memory-hogging, observations is to divide the map into tiles and process one tile one after the other. The MapIndexTask supports this loop-based processing. To define the tiles, you overlay a chessboard like pattern over your final map and tell the MapIndexTask how many rows and how many columns this pattern should have. Then, according to a numbering scheme, you may also tell the MapIndexTask which of those tiles should be processed—the default is all tiles. Have a look at the following image to understand the numbering scheme:

**Figure 4.3. The numbering scheme of the tiles for iterative deglitching. This map is sliced into 4 rows and 4 columns. This results in 16 tiles.**

If you want to initiate a tile-based processing, you have to know about four additional parameters. The first three are MapIndexTask parameters:

- `no_slicerows`: the number of rows for the chessboard pattern (default:1)

- `no_slicecols`: the number of columns for the chessboard pattern (default:1)
- `slices`: the numbers of the slices that you want to be processed. By default, all slices (= no_slicerows*no_slicecols) will be processed. But you can command to process only a subset.
- `partialmap`: this is a parameter that has to be passed to the IIndLevelDeglitchTask. The value for this parameter is always None (at first). In the first loop, None tells the IIndLevelDeglitchTask that it has to construct a new image with the correct size for your final map. In the following loops, the partialmap indicates that the new data of the loop are added to the existing map, instead of creating a new one. See in the example below for how to achieve this:

```
from herschel.pacs.spg.phot import MapIndexTask
from herschel.pacs.spg.phot import IIndLevelDeglitchTask
mapindex = MapIndexTask()
deg = IIndLevelDeglitchTask()
img = None
# img will contain your final map. This is important: The first
# image MUST be None!
d = Display()
counter = 0 #only if you want to save the MapIndices

for mi in mapindex(inframes=frames,slimindex=False,\
no_slicerows=4,no_slicecols=4,slices=Int1d({5,6,10})):
    ## you can save the mapindex here if you want (this is optional,
    ## which is why it is commented out)
    #name = "".join(["mapindex_slice_",String.valueOf(counter),".fits"])
    #fa.save(name, mi)
    #counter = counter+1
    ## otherwise continue from here
    ## (note the parameter "partialmap = img" in the deglitch task)
    img = deg(mi, frames, algo = None, map = True, mask = False, \
       partialmap = img)
    ## after the deg-step, the new data has been added to img
    del(mi)  # free your memory before the next loop is carried out
    d.setImage(img)  # this allows to monitor the progress
```

(note the line-wrap in the for-loop: you will write this on one line, the break is here only to fit the text on the page.) At the end of this loop, the variable "img" contains your deglitched map.

**Figure 4.4. Deglitching slices 5,6 and 10**

If you don't want a map but only a mask and also don't require a progress display, the code simplifies even more. In this example, we also show how to apply a customised Sigclip algorithm:

```
from herschel.pacs.spg.phot import MapIndexTask
from herschel.pacs.spg.phot import IIndLevelDeglitchTask
mapindex = MapIndexTask()
deg = IIndLevelDeglitchTask()
s = Sigclip(10, 4)
s.mode = Sigclip.MEDIAN
s.behavior = Sigclip.CLIP

for mi in mapindex(inframes = frames,
slimindex = False, no_slicerows = 4, no_slicecols = 4):
    # for the mask only version, partialmap can be omitted
    deg(mi, frames, algo = s, map = False, mask = True)
```

(note the line-wrap: you will write this on one line, the break is here only to fit the text on the page.)

## 4.3.5. A detailed look at the IIndLevelDeglitch task

The deglitching applies sigma clipping to the data that is stored in the MapIndex. The options for IIndLevelDeglitch are:

- `index`: the *MapIndex* product that stores the map data

- `inframes`: the *Frames* product. Since the *MapIndex* only contains the references to the science data (and not the actual data themselves), the *Frames* product, which contains the data, is necessary (the MapIndex can then point to these data). If you ask to flag glitches in the task call, a corresponding glitch mask is written back into the *Frames* product.
- `map`: if you want a map as output of the deglitching task, use the default value "True". Otherwise set this option to False.
- `mask`: if you want a glitch mask as output of the deglitching task, use the default value "True". Otherwise use False.
- `maskname`: you may customise the name of the glitch mask that is written into the *Frames* product. The default is "2nd level glitchmask". You could perhaps change the name if you wished to create more than one version of the mask to compare.
- `submap`: specifies a rectangular subsection of the map and deglitching will only be done for that subsection. This implies that you know already what your map looks like. The values of submap are specified in units of map pixels. For example:

  First retrieve the size of the map from the *MapIndex*

  ```
  width = mapindex.getWidth()
  height = mapindex.getHeight()
  ```

  specify the first quarter of the map as [bottom left row index, bottom left column index, height (= number of rows), width (= number of columns)]

  ```
  submap = Int1d([height/4, width/4, height/2, width/2])
  ```

  Focusing the deglitching on a submap will accelerate the deglitching process. It can be useful to optimise the deglitching parameters on a submap first, before a long observation is completely processed.

**Figure 4.5. Deglitching performed with the submap option**

- `threshold`: a threshold value that is used in combination with the parameter sourcemask. Default value is 0.5.
- `sourcemask`: defines a submap that can have any shape. The sourcemask is a *SimpleImage* with the same dimensions as the final map. The values of the sourcemask are compared to the threshold parameter and are treated as:
  - value > threshold: this location is masked and will not be processed by the deglitching algorithm
  - value < threshold: the deglitching algorithm will treat this map location

  The sourcemask can be used to exclude bright sources from the deglitching process. A check on how the deglitching task has used the sourcemask is to use the output parameter "outmask". It returns the translation of the sourcemask as a 2d boolean array. You can extract this array with

  ```
  boolean_sourcemask = IIndLevelDeglitch.getValue("outmask")
  ```

  You can take a look at it with the Display tool if you convert the boolean values to 0 and 1 by putting them into a Int2d

  ```
  d = Display(Int2d(boolean_sourcemask))
  ```

- `deglitchvector`: allows one to treat strong flux gradients. Please read details in Sec. ???.
- `algo`: a customized Sigclip algorithm. See Sec. ??? for more details.

- `weightedsignal`: as for PhotProject, this value weights the signal contributions with the signal error (stored in the *Frames* product) when it co-adds the values to get a flux. Default value is False.

#### 4.3.5.1. Avoid deglitching strong gradients

If a small part of a source falls into one pixel, the default deglitching scheme may lead to wrong results. Look at the situation drawn in the following image: The grid in this image is the map you are projecting the *Frames* (detector) pixels, indicated with filled squares, on to. The yellow *Frames* pixel contains a small but bright source. Only a very small part of it maps onto the red-framed map pixel (the fractional overlap that we call "poids" = weight is small). All other *Frames* pixels in the vicinity have a small signal value (indicated as gray).

**Figure 4.6. Mapping a small but strong source**

For the deglitching of the red-framed map pixel, the signal coming from the strong source will very likely be found as a glitch, and hence be sigma clipped. This is because by default the full signal values of the contributing *Frames* pixels are used for deglitching, not the weighted values.

If we want to improve the situation, the signal vector for deglitching has to be filled with weighted values. As weights we use the relative overlap described by the poids value and normalise it to the pixel area: weighted signal=signal*poids/pixelarea. Because a glitch most likely appears only in one timeframe (i.e. within the *Frames* pixels, the glitches have sharp time profiles), the IIndLevelDeglitchingTask provides the option to co-add all weighted contributions of every timeframe that map in to a map pixel (the red-framed pixel). A source that should appear in all timeframes will very likely be protected by this scheme. A glitch that appears only in one or two timeframes should be found.

The downside of this process is that along with the found glitch you throw away all signal contributions to that map pixel coming from a time frame. Because of the co-addition of the weighted signal, the real location of the glitch cannot be confined with more precision.

Use the weighted glitch with the option `deglitchvector`="timeordered".

## 4.3.6. Deglitching without MapIndex

One of our latest achievements is the MapDeglitchTask. It virtually does the same as the IIndLevelDeglitchTask, but it does not need a MapIndex as input. This way, it runs with much less memory usage. On the other hand, the time it needs to finish is as long as creating a MapIndex and then IIndLevelDeglitch (plus a bit).

Internally, it implements a search algorithm, that collects the necessary data for deglitching Mappixel by Mappixel. Use it, if you don't have the memory to store the MapIndex. If you can store the Mapindex and need to deglitch and map more than once, it is more useful and efficient to still use the IIndLevelDeglitchTask with the Mapindex.

```
from herschel.pacs.spg.phot.deglitching.map import MapDeglitchTask
mdeg = MapDeglitchTask()
s = Sigclip(10, 4)
s.mode = Sigclip.MEDIAN
s.behavior = Sigclip.CLIP

mdeg(frames, algo = s, deglitchvector = "timeordered", pixfrac = 1.0,
 outputPixelsize = 3.2)
```

The glitches are put into the frames products mask (that is still called "2nd level glitchmask" by default).

The MapDeglitchTask also has two parameters that we know already from the MapIndex. They are:

- pixfrac: helps you to drizzle. Values should be > 0.0.

- outputPixelsize: the size of the map pixels. Although a map is not created by the task, internally we have to know the size of the pixels of the map, so the task can calculate the overlap between frames pixels and map pixel. The value is in arcseconds, just like for the MapIndex.

# 4.3.7. MapIndexViewer: a useful tool for diagnostic and fine tuning

A MapIndex contains a lot of interesting data that is very useful to be analysed. The tool for doing this is the MapIndexViewer. It is called like this:

```
from herschel.pacs.spg.phot.gui.mapindexview import MapIndexViewer
MapIndexViewer(mi, frames)
```

feeds in the *MapIndex* mi and the *Frames* product. Or

```
from herschel.pacs.spg.phot.gui.mapindexview import MapIndexViewer
MapIndexViewer(mi, frames, img)
```

provide also the map. If not, it can be calculated on-the-fly.

**Figure 4.7. The MapIndexViewer GUI**

In the top part of the GUI you see the map displayed by the Display tool. If you click on the map, a plot of the all values that a MapIndex contains for the selected map pixel plus the signal values from the *Frames* product is shown at the bottom. A click on the button "show table" displays the numerical values as a table on the right side of the GUI.

## 4.3.7.1. Optimising the Sigclip algorithm with the MapIndexViewer for best deglitching results

The bottom of the table contains a panel where you can test Sigclip parameters. Change the parameters and select the "apply" checkbox. Then the Sigclip will be applied to the plotted signal. You may even choose the timeordered option for the signal. That provides an easy way to find the best Sigclip parameters without running the deglitching repeatedly. Just find glitches in your map and optimise Sigclip before you deglitch the full map.

Maybe it is also worth noting at this point that you can use the sourcemask and/or the submap parameters to deglitch different sections of the map with different Sigclip parameters. Just use the sourcemask/submap to divide the map into multiple regions and create multiple glitchmasks with different sets of parameters instead of only one.

There is also a permanent preview of the "Mean +/- nsigma*Stddev" and the "Median +/- nsigma*Median Absolute Deviation" in the MapIndexViewer plot section. These are displayed at the x-axis index -5 as a value (Mean or Median) and an error bar (+/- nsigma*Stddev or +/- nsigma*Median Absolute Deviation).

The values nsigma and whether mean or median should be used are taken directly from the Sigclip panel.

**Figure 4.8. Preview of the signal arrays Mean, Median and nsigma values:**

## 4.3.7.2. How to write your own Sigclip algorithm in jython

You may write your own Sigclip algorithm that can be passed to and be used by the IIndLevelDeglitchingTask. Two things are important:

- your algorithm has to be a jython class that extends the numerics Sigclip
- the jython class must implement a method called "of". This way it overwrites the "of' method in the numerics Sigclip.

Look at the necessary code:

```
from herschel.ia.numeric.toolbox.basic import Sigclip

class My_Own_Sigclip(herschel.ia.numeric.toolbox.basic.Sigclip):

    def of(self, vector):
        #
        System.out.println("using MY OWN algorithm")
        #
        #here you may write any code that does your job
        #only make sure it returns a Bool1d with the same
        #length as vector
        bvector = Bool1d(vector.size)
        return bvector
```

You have to do your implementation of Sigclip in the of-function. The interesting input parameter of that function is vector. It will contain the array of signal values (framessignal or timeordered) of a MapIndex row/column pair. The second level deglitch task will loop over the MapIndex and call your Sigclip for every row/column pair.

Your implementation MUST return a Bool1d that has the same length as the input vector. The convention is: the signal values will be treated as glitches at the indices where the returned Bool1d is true. That is all.

After you have finished your implementation, you have to make the class available for your hipe session. Here are two possibilities to do it:

1. load the code by opening the file with hipe, place the green hipe arrow at the first import (from herschel.ia.numeric.toolbox.basic import Sigclip) and then click run (repeatedly, until the arrow jumps below the last line of your code ).
2. Save this jython class as a file. Use a directory that is already in the sys.path (like ./, your local working directory from which you start hipe). From there import it into your hipe session.

Use it with the deglitch task as follows:

```
mySclip = My_Own_Sigclip()
img = deg(mi, frames, algo = mySclip)
```

Actually, you could use the above code for My_Own_Sigclip. It will do no deglitching, because all returned boolean values are false (Bool1d(vector.size) contains only false by default), but it should run out of the box.

## 4.3.7.3. Command line options for the MapIndex

On the command line, you may get the same information by using the MapIndex interface. For map location (map_row, map_col) of the map, the array of all MapElements is retrieved like this:

```
mapelement_array = mi.get(map_row, map_col)
print mapelement_array[0]
#> MapElement: detector 32, timeindex 1579, poids 0.0, pixelsurface 0.0
```

or

```
print mapelement_array[0].detector # >32
print mapelement_array[0].timeindex # >1579
```

The mapelements are sorted in timeindex. This means mapelement_array[0] contains data with the smallest timeindex, the last mapelement in the mapelement_array contains values with the highest

timeindex. To save memory, the detector_row and detector_column information of the data origin in the Frames product is encoded in the value detector. For the blue Photometer array the detector has values between 0 and 2047, for the red array the values are between 0 and 511. This is because the detector_row, detector_column values are compressed into single values.

**Figure 4.9. Detector numbering scheme for the blue photometer array:**

The detector value can be translated to the actual detector_row and detector_column of the Frames data array using the method det2rowCol of the MapIndex product (FullMapIndex or SlimMapIndex):

```
detector_rowcol = MapIndex.det2rowCol(detector, frames.dimensions[1])
detector_row = detector_rowcol[0]
detector_column = detector_rowcol[1]
print detector_row, detector_column
```

Note: this is very complicated. A simplification, where detector_row and detector_column are directly stored in the Mapelements is on the way.

Please keep in mind that frames.dimensions[1] returns the size of the full detector array (32 or 64). This may not be the case if you work with a sliced Frames product. If you use the FullMapIndex, the surface and poids information is different from 0 (Remember? The SlimMapIndex stores only time and detector, the FullMapIndex also poids and pixelsurfaces ).

```
print full_mapelement_array[0]
# >MapElement: detector 32, timeindex 1579, poids 3.1238432786076314E-4,
 pixelsurface 0.7317940044760934
print full_mapelement_array[0].poids
print full_mapelement_array[0].pixelsurface
```

The corresponding signal value is still stored in the frames product. To get access to the full set of the projection parameters, including the signal, please use this code:

```
detector_rowcol_n = MapIndex.det2rowCol(full_mapelement_array[n].detector,
 frames.dimensions[1])
signal_n = frames.getSignal(detector_rowcol_n[0], detector_rowcol_n[1],
 full_mapelement_array[n].timeindex)
```

What do you do, if you have created a SlimMapIndex and have to know the poids and surface values for some pixels? There is a way to get to these values without recalculating a FullMapIndex. The values can be calculated on-the-fly with a task named GetMapIndexDataTask. You use it like this:

```
from herschel.pacs.spg.phot import GetMapIndexDataTask
data_acccess = GetMapIndexDataTask()
full_mapelement_array = data_access(mapindex, map_row, map_column, frames, command =
 "mapelements")
```

Note: the use of the GetMapIndexDataTask is also very ugly and complicated. A way to access the MapElement array directly from the MapIndex product is on the way. If it is done, you will find it documented here.

# 4.4. MMT Deglitching

This task detects, masks and removes the effects of cosmic rays on the bolometer. The photMMT-Deglitching task is based on the multiresolution median transforms (MMT) proposed by Stark et al (1996). This task is applied when the astrometric calibration has still to be performed. Thus, it does not rely on data redundancy, as the Second Level Deglitching method, but only on the time line noise properties.

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:

**Figure 4.10.**

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

**Figure 4.11.**

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:

**Figure 4.12.**

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

**Figure 4.13.**

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 task creates a mask, MMT_glitch mask, which flag all the readouts identified as glitches. By default the task mask only the glitches, but it can also replace the signal of the readouts affected by glitches by interpolating the signal before and after the glitch event.

**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)

# 4.4.1. Details and Results of the implementation

This is the signature of the task:

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

Task Parameters

- **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 for mmt_mode == "add" and 2 for mmt_mode == "multiply"
- **mmt_mode** (String): Defines how the environment should be modified between the scales. Possible values: "add" or "multiply". Default is "add"
  - example: the environmentsize for the subsequent median transform environment boxes will be

    ```
    env(0) = mmt_startenv, env(n) = env(n-1) mmt_mode incr/fact
    ```

  - default for mode "add" means then:

    ```
    env(0) = 1, env(1) = 1 + 1; env(2) = 2 + 1; etc.
    ```

  - default for mode "multiply" means:

    ```
    env(0) = 1, env(1) = 1*2; env(2) = 2*2; env(3) = 4*2; etc
    ```

- **noiseDetectionLimit** (double): Threshold for determining the image noise. values between 0.0 and 1.0. Default is 0.1
- **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
- **onlyMask** (boolean): If set to true, the deglitching will only create a glitchmasks and not remove the glitches from the signal. If false, the glitchmask will be created and the detected glitches will be removed from the signal. Default value: true
- **maskname**: This paramter allows to set a custom maskname for the glitchmask, that is added to the frames object by this task. Default value: "MMT_Glitchmask"
- **sourcemask** (String): It is the name of a mask. mmt deglitching is not only sensitive to glitches but also to pointsources. To avoid deglitching of sources, the sourcemask may mask the locations of sources. If this mask is provided in the frames object, the masked locations will not be deglitched by this task. After the task is executed, the sourcemask will be deactivated. Use the PhotReadMask-FromImageTask to write a mask into the frames object. Instruction how to do it are provided in the documentation of the task. Default value: ""
- **use_masks** (boolean): this paramter determines, whether the masks are used to calculate the noise. If set to true, the standard deviation will only be calculated from the unmasked samples. Default value: false
- **noiseByMeanFilter** (boolean): this paramter has effect, if neither a noise array or a noise value is submitted by the user. In that case, MMTDeglitching has to calculate image noise internally. This can be done by simply subtract all timelines by their mean filtered values (noiseByMeanFilter = true). If noiseByMeanFilter = false (default) the noise will be calculated in the wavelet space as describe by Starck and Murtagh. In both cases, the noise will be calculated separately for every pixel. After execution of the task, the noise can be inspected: MMTDeglitching.getImageNoiseArray() returns the noise array
- **noiseModel** (Double1d): a Double1d that models the noise. The standard internal approach is to use a Gaussian noise with a standard deviation of 1. This value is needed to calculate the noise in wavelet space (see (3)). Default: a Gaussian noise created by the class herschel.pacs.share.math.GaussianNoise with the standard deviation of 1. The length of the default data is 100.000

The method works well till the maximum number of readouts of a glitch is much smaller than the one of a real source. This method works particularly well for observations containing mostly point sources (e.g. deep field observations). Indeed, in these cases the sources do not affect the estimate of the noise image and the sources are still large enough to be distinguished from glitches than usually last one or two readouts. The following example works pretty well for this case:

```
frames =
 photMMTDeglitching(frames,scales=3,nsigma=5,mmt_mode='multiply',incr_fact=2,onlyMask=0)
```

However, if the observations includes particularly bright sources, the task may deglitch their central brightest core. In this case several tricks can be applied to avoid deglitching the sources. For instance, the user might choose use the previous settings parameters and provide also the "sourcemask". This will let the task mask all the glitches with the exclusion of the masked source readouts. In a second pass, the user can re-run the task with "relaxed" parameters and without providing the sourcemask:

```
frames =
 photMMTDeglitching(frames,scales=2,nsigma=9,mmt_mode='multiply',incr_fact=2,onlyMask=0)
```

This deglitching method is not recommended for extended emission observation. Indeed, in most of the cases the task is not able to properly recover the noise image due to the extended emission and the brightest regions of the extended source are misclassified as glitches. In this case the second level deglitching approach is recommended.

# 4.5. photRespFlatFieldCorrection

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

# 4.6. photHighPassfilter

This task is used to remove the 1/f noise of the PACS data in the branch of the pipeline that uses PhotProject. The task is removing from the timeline a running median filter. the median is estimated within a box of a given width around each readout. The user must specify the box width depending on the scientific case (see the ipipe scripts for different examples). Is is also higly recommended to properly mask the sources by providing a source mask.

```
outFrame = highPassFilter(inFrame [,copy=copy, maskname = "sourcemask"] )
```

- **inFrame** (Frames): input Frames object
- **environment** (int) default value = 20. The filterwidth is the number of values in the filterbox. It is 2*environement +1.
- **algo** (String): The algorithm for high pass filtering default value = "median". Alternative algorithms: "mean" : HighPassFilter applies a MeanFilter
- **copy** (int):
  - 0 - return reference: overwrites the input frames (default)
  - 1 - return copy : creates a new output without overwriting the input
- **maskname** (String): If maskname is specified and the mask is contained in the inframes object, a MaskedMedianFilter will be used for processing.
- **deactivatemask** (Boolean): If a mask covers the sources, it can be critical to assure deactivation of the mask before the projection is done! if a mask has been provided, deactivatemask leaves the mask untouched (deactivatemask = False), or deactivates the mask after processing the Highpassfilter (deactivatemask = True). Default value = True
- **severeEvents** (Bool1d): A Bool1d array that defines the time positions, where completely masked environment boxes have severe impact on the processing. These incidents are reported by the task.
- i**nterpolateMaskedValues** (Boolean): Usually masked values are not included in the Median calculation. In the cases, where the median box is completely masked, the mask will be ignored and

the median will be taken from all box values, as if nothing were masked. If the parameter "interpolateMaskedValues" is true (default), these completely masked filter boxes will be linearly interpolated over all masked values. This is done by taking the first unmasked values right and left from the masked environment and use them for the interpolation. If it is false, the Median vector contains Medians of all values in the environment box, as if no mask were there. Default = true.

# 4.7. photProject

This task accepts a frames object as input that has been processed by a pacs pipeline. The minimum contents of the frames object must be a signal array and the pointing status entries provided by the AddInstantPointingTask. PhotProject maps the images contained in the frames class into one single map and returns it as a SimpleImage object that contains the map, the coordinates (in form of a world coordinate system wcs) and inputexposure, noise (error) and weight maps. The mapping process by default uses all activated masks from the input frames object. A second functionality of PhotProject is the preparation for the second level deglitching. This is done with the paramter deglitch = true. In that case, a MapIndex Object is created (instead of the SimpleImage) that contains the necessary information to deglitch the signal contributions into each output pixel with a Sigma clipping. This is done with the IIndLevelDeglitchTask, which needs the MapIndex as input.

```
image = photProject(inFrames, [outPixelSize=outPixelsize,] [copy=1,] [monitor=1,]
 [optimizeOrientation=optimizeOrientation,]\
                          [wcs=wcs,] [pixfrac=pixfrac,] [calibration=calibration])
```

- **inframes** (Frames): the input frames class
- **calTree** (PacsCalibrationTree): calibration tree containing all calibration products used by the pipeline
- **copy** (int): 0 if the original frames class should be used, 1 if a copied version should be use.
  - 0 - return reference: overwrites the input frames (default)
  - 1 - return copy : creates a new output without overwriting the input
- **weightedsignal** (Boolean): set this to True, if the signal contributions to the map pixels should be weighted with the signal error. The error is taken from the noise array in the inframes object. Default value: False.
- **pixfrac** (float): ratio of drop and input pixel size.
- **outputPixelsize** (float): 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).
- **monitor** (Boolean): If True, shows the map monitor that allows in situ monitoring of the map building process. Default value: false.
- **optimizeOrientation** (Boolean): rotates the map by an angle between 0 and 89 degrees in a way that the coverage of the outputimage is maximized as a result, north points no longer upwards. Possible values:
  - false (default): no automatic rotation
  - true: automatic rotation
- **wcs** : allows to specify a customized wcs that is used for projection. Usually photProject calculates and optimizes its own wcs. The wcs parameter allows to overwrite the internally created wcs. The easiest way to create a Wcs is to use the Wcs4mapTask (that is also used internally by this task), modify its parameters. The necessary paramters to modify are:
  - *wcs.setCrota2(angle)*: the rotation angle in decimal degrees (45.0 for 45 degrees)

- *wcs.setCrpix1(crpix1)*:The reference pixel position of axis 1. Use the center of your map: mapwidth/2 (the Ra-coordiante)
- *wcs.setCrpix2(crpix2)*:The reference pixel position of axis 2. Use the center of your map: mapheight/2 (the Dec-coordinate)
- *wcs.setCrval1(crval1)*:The coordinate of crpix1 (ra-value in decimal degrees - use Maptools.ra_2_decimal(int hours, int arcmin, double arcsec) for conversion)
- *wcs.setCrval2(crval2)*:The coordinate of crpix2 (dec-value in decimal degrees - use Maptools.dec_2_decimal(int degree, int arcmin, double arcsec) for conversion)
- *wcs.setParameter("NAXIS1", mapwidth, "Number of Pixels along axis 1." )* :crpix1*2, if you follow these instructions
- *wcs.setParameter("NAXIS2", mapheight, "Number of Pixels along axis 2" )* :crpix2*2, if you follow these instruction

- **deglitch** (Boolean): It specifies, that PhotProject does not create a map, but writes all contributions to the map into a MapIndex object instead. After PhotProject is finsished, the MapIndex can be obtained with mstack = photProject.getValue("index"). The PacsMapstack is the input object for the second level deglitching with IIndLevelDeglitchingTask. Possible values: false (default) or true.
- **slimindex** (Boolean): together with deglitch = true instructs PhotProject to build a memory efficient index for deglitching. Building an index first means that second level deglitching will take longer, but can be processed with significantly less memory requirements (ca. 20% compared to the default slimindex = false).
- **image** (SimpleImage): the returned SimpleImage that contains a map and a Wcs (World Coordinate System).
- **index**: if the option deglitch is used, the SimpleImage is not created. Instead a MapIndex object is produced that must be used as input to the IIndLevelDeglitchingTask. The index is not returned by index = photProject(....) like the SimpleImage. Instead, photProject has to be executed (photProject(...) and then index = photProject.getValue("index") must be called.

The photProject task performs a simple coaddition of images, by using the drizzle method (Fruchter and Hook, 2002,PASP, 114, 144). There is not particular treatment of the signal in terms of noise removal. The 1/f noise is supposed to be removed by the high-pass filtering task. The drizzle algorithm is conceptually simple. Pixels in the original input images are mapped into pixels in the subsampled output image, taking into account shifts and rotations between images and the optical distortion of the camera. However, in order to avoid convolving the image with the large pixel "footprint" of the camera, we allow the user to shrink the pixel before it is averaged into the output image, as shown in the figure below.

**Figure 4.14.**

The new shrunken pixels, or "drops", rain down upon the subsampled output image.

**Figure 4.15.**

The flux in each drop is divided up among the overlapping output pixels in proportion to the areas of overlap. Note that if the drop size is sufficiently small not all output pixels have data added to them from each input image. One must therefore choose a drop size that is small enough to avoid degrading the image, but large enough that after all images are "dripped" the covera ge is fairly uniform. Due to the very high redundancy of PACS scan map data, even in the mini-map case, a very small drop size

can be chosen (1/10 of the detector pixel size). Indeed, a small drop size can help in reducing the cross correlated noise du e to the projection itself (see for a quantitative treatment the appendix in Casertano et al. 2000, AJ, 120,2747). The size of the drop size is usually fixed through the ``pixfrac'' parameter, which given the ratio between the drop and the input pixel si ze. The mathematical Formulation of the drizzling method is described below:

**Figure 4.16.**

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, so $0 < a(xy) < 1$. The weight w(xy) of the pixel can be zero if it is a bad pixel (hot pixels, dead pixels, cosmic rays event, ...), or can be adjusted according to the local noise (the value is then inversely proportional to the variance maps of the input image). 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 key parameters of this task are the the output pixel size and the drop size. A small drop size can help in reducing the cross correlated noise due to the projection itself (see for a quantitative treatment the appendix in Casertano et al. 2000, AJ, 120,2747). However, the remaining 1/f noise not removed by the high-pass filter task is still a source of cross-correlated noise in the map. Thus, the formulas provided by Casertano et al. 2000, which account only for the cross correlated noise due to the projection, do not provide a real estimate of the total cross correlated noise of the final map. Indeed, this is a function of the high-pass filter radius, the output pixel and the drop size. Nevertheless, those formulas can be used to reduce as much as possible the cross-correlated noise due to the projection. We stress here that the values of output pixel size and drop size strongly depend on the redundancy of the data (e.g. the repetition factor). For instance, a too small drop size would create holes in the final map if the redundancy is not high enough (see Fruchter and Hook, 2002 and the PDRG for a clear explanation). The drop size is set in the pixfrac parameter in the photProject input. The pixfrac parameter is expressed as the ratio between the drop and in input pixel size. A drop size of 1/10 the input pixel size is found to give very good quality in the final map. Since these parameters have to be adjusted case by case, we invite you to play with them to find the right balance between noise, cross-correlated noise and S/N of the source for creating the optimal map. The photProject task comes in two flavors: a simple average of the input pixel contributions to the given output pixel as done in the previous line, or a weighted mean of those contributions. The weights are estimated as the inverse of the error square. However, since the noise propagation is not properly done in the PACS pipeline a proper error cube must be provided to obtain good maps.

# 4.8. photProjectPointSource

Creates the final map in chop-nod mode. See the corresponding *photProjectPointSource* entry in the URM for the description of this task. The description of the projection method can be found under photProject earlier in this document.

# 4.9. Features of the Map Monitor

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).

# 4.10. Reducing minimaps (combining scan and cross-scan)

The minimap mode is a special way to execute scanmap observations. A minimap always consists of two obsids (the scan and cross scan), which is why they need to be handled together. In the following we will provide a sample script that can be used to reduce the minimap observations. First you need to place your obsid in an array which we can loop over. Basically they can be put into a single line but it is better to organise them in pairs so it is easier to oversee them. You also need to define the directory where you want to put your images and set in the coordinates of your source (rasource, decsource) in decimal degrees.

If you wish to follow these instructions in real-time, we suggest you cut and paste the entirety of this loop into a python script file in the Editor tab of HIPE. You can also look at the Pipeline menu scripts, which present the pipeline reduction for different types of AORs.

```
# create an array of names—all being of the same source
# the \ is a line break you can employ
# the numbers here are the obsids of your scan (odd entries)
# and associated cross scan (even entries)
Nobs = [\
13421111,13421112,\
13422222,13422223,\
...]

# and an array of cameras (these are the only possible)
channel = ["blue","red"]

# where the data are
direc = "/yourDirectory/"

# coordinates
rasource =      # RA of the source in degrees
decsource =     # DEC of the source in degrees
cosdecCOS(decsource*Math.PI/180.)
```

Then you can start looping over your observation. Note here: the scripts below that are indented from the very first line, are indented because they are still part of the for loop that begins here:

```
for k in range(len(Nobs)):
  if k % 2: continue
  OBSIDS=[Nobs[k],Nobs[k+1]]
```

This part makes sure that the scan (k) and cross-scan (k+1) are processed together. Then you need to do the standard pipeline processing up to a level for each individual obsid and channel. The following "for" loop does this. You can also set the pixel size of your final map using variables. The frames=[] is needed to let HIPE know that we are using an object array (we are holding each *Frames* created in a list). It will contain an array of images as they are used in the single observation case. Continue with the script:

```
  for j in range(len(channel)):
    print "\nReducing OBSID:", OBSIDS[i], " (", i+1, "/", len(Nobs), ")"
    #
    for i in range(len(OBSIDS)):
      frames=[]
      obsid=OBSIDS[i]
      print channel[j]
```

```
# channel :
camera = channel[j]
# output map pixel size:
if camera=='blue':
  outpixsz=3.2
elif camera=='red':
  outpixsz=6.4
```

Then you need to get your ObservationContext. In this example we use the simplest method using the Herschel Science Archive. For this you need the login properties set, which is explained in Sec. ???.

```
obs = getObservation(obsid, useHsa = True)
```

For parallel mode observations use:

```
obs = getObservation(obsid, useHsa = True, instrument='PACS')
```

otherwise it will return a SPIRE ObservationContext (which you cannot process unless you have the SPIRE/all version of HIPE).

*However please note that the HSA do not encourage the use of getObservation with useHsa=True within a loop, as it places great stress on the archive—and especially if the loop crashes, you will have to retrieve the data all over again! It is far better to have the data already on your disc.* To get your observation from your private pool also (as was also explained in Sec. ???):

```
dir = '/yourDirectory/' #this you can also put outside the loop
obs = getObservation(obsid, poolLocation=dir)
```

Then you need to extract some important information from your observation, such as operational day (OD) number and scan speed. These are going to be used later.

```
OD = obs.meta.get("odNumber").value
object = obs.meta.get("object").value
scanSpeed = obs.meta.get('mapScanSpeed').value
if scanSpeed == 'medium':
  speed = 20.
elif scanSpeed == 'low':
  speed = 10.
elif scanSpeed == 'fast':
  speed = 60.
```

Using the scan speed you can set up highpass filter half-width value (in units of readouts). You might want to EDIT the widths and TEST the photometry for different values here. For point sources at medium scan speed, values of the highpass filter half-width of 20 in the blue and 25 in the red provide good results. But of course the width depends on the science case and the scan speed. A little example generalising the scan speed dependence can be found below, which you can chose to include in this big for-loop script if you like:

```
if camera=='blue':
  hpfwidth=int(CEIL(20 * 20./speed))
elif camera=='red':
  hpfwidth=int(CEIL(25 * 20./speed))
```

Get the pointing product from the ObservationContext, get the calTree, extract houskeeping parameters and the orbit ephemeris product

```
pp = obs.auxiliary.pointing
calTree = getCalTree()
photHK=obs.level0.refs["HPPHK"].product.refs[0].product["HPPHKS"]
oep = obs.auxiliary.orbitEphemeris
```

Then you get the Level 0 data cube (frames) for the blue or red channel *Frames*. We can also obtain the "filter" information from the meta data at this point and convert it into human-readable form. It is important if we observe in both wavelength regimes of the blue camera.

```
      if camera=='blue':
        frames[i]=obs.level0.refs["HPPAVGB"].product.refs[0].product
        if (frames[i].meta["blue"].getValue() == "blue1"):
          filter="blue"
        else:
          filter="green"
      elif camera=='red':
        frames=obs.level0.refs["HPPAVGR"].product.refs[0].product
        filter="red"
```

Now we need to identify instrument configuraiton blocks in the observation and remove the calibration block keeping only the science frames.

```
      frames[i] = findBlocks(frames[i], calTree=calTree)
      frames[i] = removeCalBlocks(frames)
```

After that we need to execute the tasks already described in Chap. 3.

```
      frames[i] = photFlagBadPixels(frames[i], calTree=calTree)
      frames[i] = photFlagSaturation(frames[i], calTree=calTree, \
         hkdata=photHK)
      frames[i] = photConvDigit2Volts(frames[i], calTree=calTree)
      frames[i] = convertChopper2Angle(frames[i], calTree=calTree)
      frames[i] = photAddInstantPointing(frames[i],pp,orbitEphem = oep)
      if (isSso):
           print "Correcting coordinates for SSO: ", Date()
           if (obsid == obsids[0]):
               timeOffset = ft.microsecondsSince1958()
           frames =
 correctRaDec4Sso(frames,horizons,timeOffset,orbitEphem=oep,linear=False)
      frames[i] = photAssignRaDec(frames[i], calTree=calTree)
      frames = cleanPlateauFrames(frames, calTree=calTree)
      frames = addUtc(frames, timeCorr)
      frames[i] = photRespFlatfieldCorrection(frames[i], calTree = calTree)
```

These tasks flag the known bad pixels, flag saturated pixels, convert from ADUs to Volts, convert chopper angles into angles on the sky apply the flat-field, compute the coordinates for the reference pixel (detector centre) with aberration correction (and correct for motions of Solar System objects if necessary), and assign ra/dec to every pixel and convert Volts into Jy/pixel,

At this point you have gotten to the Level 1 product, which is calibrated for most of the instrumental effects. You might want to save this product before the highpass filter task to be able to go back and optimise your data reduction afterwards. There are three ways to save your *Frames*.

**FIRST OPTION** to save the data: use the 'save' task to store your data locally:

```
      savefile = direc+"frame_"+"_" + obsid.toString() + "_" \
         + camera + "Level_1.save"
      print "Saving file: " + savefile
      save(savefile,"frames[i]")
```

**SECOND OPTION** to save the data: store your data locally as a FITS file, specify the output directory as you did for the first option:

```
      savefile = direc+"frame_"+"_" + obsid.toString() + "_" \
         + camera + "Level_1.fits"
      print "Saving file: " + savefile
      simpleFitsWriter(frames[i],savefile)
```

**THIRD OPTION**: store frames in memory by copying to a new *Frames* (not a good idea if you do not have a big-memory machine)

```
      frames_original[i]=frames[i].copy()
```

The next step would be high pass filtering your data to remove the 1/f noise. However this filtering can severely affect the flux of your source. That is why you need to mask your astronomical object(s) before filtering. Here we propose two ways of masking your sources.

**FIRST MASKING OPTION:** mask the source blindly within 25 (or whatever you like) arcsec radius of the source coordinates (rasource and decsource). This is appropriate when you have only one source and you know its coordinates (the typical minimap case). If you have many sources with known coordinates, you can still use this method by looping over a prior source list and masking within the desidered aperture around the source position. Just be aware that looping in Jython is quite time consuming.

First you need to define the mask HighpassMask to use for masking sources in the highpass filtering.

```
awaysource=SQRT(((frames[i].ra-rasource)*cosdec)**2 \
  +(frames[i].dec-decsource)**2) < 25./3600.
if (frames[i].getMask().containsMask("HighpassMask") == False):
  frames[i].addMaskType("HighpassMask","Masking source for Highpass")
  frames[i].setMask('HighpassMask',awaysource)
```

The first line selects the pixel coordinates that are closer to the source than 25 arcsec. The second line examines if the HighpassMask already exists in frames and if not then creates it from the pixels selected in the first line. Then we can run highpass filtering while masking the source:

```
frames[i] = highpassFilter(frames[i],hpfwidth,\
    maskname="HighpassMask")
```

> **Tip**
>
> In many cases the MMT deglitching might detect bright sources as glitches (check this). It is often useful to disable the deglitching on the target in frames' mask. You do this by overwriting the mask within your desired region:
>
> ```
> mask=frames[i].getMask('MMT_Glitchmask')
> awaysource=SQRT(((frames[i].ra-rasource)*cosdec)**2\
>    +(frames[i].dec-decsource)**2) > 25./3600.
> frames[i].setMask('MMT_Glitchmask',mask & awaysource)
> ```
>
> Here the first line reads the mask created by the MMT deglitching task. Then it finds the pixel coordinates that are farther than 25 arcsec from the source, and finally it puts back the mask for each and all pixels that were masked originally by MMT deglitching and are is farther from the source than 25 arcsec.

**SECOND MASKING OPTION:** mask the source based on sigma-clipping of the map. This is more general than previous one but it might need some tuning to set the threshold properly (use MaskViewer to do this).

So, first we create a map with high pass filter without masking anything as we saw it in Chap 5.

```
frames[i] = highpassFilter(frames[i],hpfwidth)
frames[i] = filterOnScanSpeed(frames[i],limit=10.0,scical="sci",copy=True)
map1 = photProject(frames[i], calTree=calTree,calibration=True,\
   outputPixelsize=outpixsz)
```

Now we need to restore the frames saved before the high pass filtering

```
restore(savefile)
```

We will then find a threshold to mask, find out where the sources are, and then mask everything above that threshold. The choice of the threshold depends on the science case and there is no general criterium. This part of the pipeline needs a high level of interaction. That is why having the Level 1 product saved before is a good way to avoid repeating the whole prior data reduction several times.

Define the threshold on the basis of the map's standard deviation value. The following line derive the standard deviation of the map where there is signal

```
threshold=STDDEV(map1.image[map1.image.where(ABS(map1.image)\
   > 1e-6)])
```

Then we masks all readouts in the timeline at the same coordinates of the map pixels with signal above the threshold (bonafide sources) using the photReadMaskFromImage task:

```
maskMap = map1
frames[i] = photReadMaskFromImage(frames[i], maskMap, extendedMasking=True,\
  maskname="HighpassMask", threshold=threshold,calTree = calTree)
```

This task basically goes over the final map (here it called maskMap) pixel by pixel and along their (time-ordered data) vectors, and if the value of vector data is larger than the threshold it masks that point. It is possible to check the HighpassMask with MaskViewer:

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

And now you have set the mask, you can again run the highpass filter on the *Frames*.

*Of course if you intend to execute this step you cannot use the loops but process only one obsid at a time.*

**Now you have masked your source (using option 1 or 2), you can continue.**

Begin by running the highpass filter on the data.

```
frames[i] = highpassFilter(frames[i],hpfwidth,maskname="HighpassMask",\
   ,interpolateMaskedValues=True)
```

> **Tip**
>
> If the hpfwidth is smaller than the source size, the whole hpfwidth could be masked. In this case the task will calculate the median over the given hpfwidth as if it was not masked. Thus it will remove also source flux. In these case the "interpolation" parameter should be set to let the task interpolate between the closest values of the median over the timeline—True as we have here.

Now we can perform the second level deglitching (see Chap. 4). We can also check what fraction of our data were masked by the second level deglitching. If the values is larger than 1% you might want to investigate the deglitching a little further to make sure that only real glitches are masked.

```
frames[i]=photMapDeglitch(frames[i])

mask = frames[i].getMask('MapGlitchmask')
nMask = mask.where(mask == True).length()
frac = 100.*nMask/len(frames.signal)
print "   Second level deglitching has masked "+str(nMask)+" pixels."
print '   Second level deglitching has masked %.2f'%frac+'% of the data.'
```

Then we select the frames taken at constant scan speed (allowing 10% uncertainty with the keyword "limit=10.0"). Finally we can create our map using the highpass filtered data as we saw it before.

```
frames[i] = filterOnScanSpeed(frames[i],limit=10.0,scical="sci",copy=True)
map2 = photProject(frames[i], calTree=calTree,calibration=True,\
   outputPixelsize=outpixsz)
```

We can look at our map using the Display command.

```
Display(map2)
```

then we can save our map into a fits file using a simple fits writer

```
outfile = direc+ "map_"+"_"+ obsid.toString() + "_" + filter + ".fits"
print "Saving file: " + outfile
```

```
      simpleFitsWriter(map2,outfile)
```

Of course we can choose any name for our output file (outfile); if you reduce more than one obsids with this script then it is advisable to use a filename that includes at least the obsid and the filter (red, green or blue) information (giving the same name means the next loop's result will overwrite the previous).

Now we are finished with one obsid and our loop will start on the second one. After the second loop is finished we can join our frames and create a map using the scan and the cross-scan. First we join the frames that are stored in frames object array

```
    if i == 0:
       frames_all=frames[0].copy()
    else:
       frames_all.join(frames[i])
```

Then we use photProject to simply project all the frames onto a common map, delete the joined frames to save space, and write out our final map in a fits file.

```
    map = photProject(frames_all,calibration=True,calTree=calTree,\
       outputPixelsize=pixsize)
    Display(map)
    del(frames_all)
    outfile = direc+ "map_"+"_"+ OBSIDS[0].toString() + \
       OBSIDS[1].toString + "_" + filter + ".fits"
    simpleFitsWriter(map,outfile)
```

# 4.11. Dealing with Solar System objects (SSOs)

The processing of observations of SSOs require some extra attention since the execution of one individual observation does not account for moving objects in real time, but recentres the telescope with each new pointing request. Thus, by default, the assignment of pointing information to the individual *Frames* assumes a non-moving target. Here we present a little piece of code that can be included in any photometer processing script to take into the account the motion of a SSO during the observation.

## 4.11.1. correctRaDec4Sso

During the process of the data reduction, the task correctRaDec4Sso is able the reassign coordinates to the pixels of each frame by using the calculated position of the target as the reference instead of the centre of the FOV. It uses the horizon product that is available for data that have been processed with the SPG (standard product generation, i.e. the automatic pipeline processing done on the data as you got them from the HSA) version 4 or higher. The version is listed in the metadata of the ObservationContext under the keyword "creator". Any attempt to correct the pointing of SSOs with data that have been processed with an earlier version will crash the session. This is why the code listed here checks whether the horizons product can be found in the ObservationContext.

```
#Necessary module imports:
from herschel.ia.obs.auxiliary.fltdyn import Horizons
from herschel.ia.obs.auxiliary.fltdyn import Ephemerides

#Extraction of SSO relevant products:
# Is it a solar System Object ?
isSso = isSolarSystemObject(obs)
if (isSso):
    try:
        hp = obs.refs["auxiliary"].product.refs["HorizonsProduct"].product
        ephem = Ephemerides(oep)
        print "Extracting horizon product ..."
        if hp.isEmpty():
            print "ATTENTION! Horizon product is empty. Cannot correct SSO proper
 motion!"
```

```
            horizons = None
        else:
            horizons = Horizons(hp, ephem)
    except:
        print "ATTENTION! No horizon product available. Cannot correct SSO proper
 motion!"
        horizons = None
else:
    horizons = None
```

Note that for now the correctRaDec4Sso task can only be applied also to multiple associated observation (e.g. scan and cross-scan). In our example below we will show the general method:

```
OBSID=[1342199515,1342199516]
for obsid in OBSID:

    ...

    if (isSso)
        print "Correcting coordinates for SSO ..."
        if (obsid == OBSID[0]):
            timeOffset = frames.getStatus("FINETIME")[0]
        frames = correctRaDec4Sso(frames, horizons,
 timeOffset,orbitEphem=oep,linear=False)
    frames = photAssignRaDec(frames, calTree=calTree)
```

The correctRaDec4Sso() task first determines the start and end time of the observation and then extracts the theoretical position of the SSO from the horizons product for the two time stamps. The second half of the task interpolates the coordinates for each frame. Some of the code performs necessary time-format conversions. The interpolation is not done relative to the start of the observation of the given OBSID, but to the time that is handed to the task via the "timeOffset" option. In the script this is set to the start time of the first in the list of observations. The trick is that all subsequent scan maps are referenced to the time frame of the first scan map of the sequence. As a result, all individual frames of all obsids will be referenced to the calculated position of the SSO.

# Chapter 5. PACS scanmap reduction using MADmap

## 5.1. MADmap

The Microwave Anisotropy Dataset mapper (MADmap) is an optimal map-making algorithm, which is designed to remove the uncorrelated one-over-frequency (1/f) noise from bolometer Time Ordered Data (TOD) while preserving the sky signal on large spatial scales. The removal of 1/f noise creates final mosaics without any so-called “banding” or “striping” effects. MADmap uses a maximum-likelihood technique to build a map from a TOD set by solving a system of linear equations.

**Figure 5.1. The function of MADmap is to remove the effect of 1/f noise. Left: image created without MADmap processing. Right: The same image after MADmap processing. The central object has been masked out.**

For Herschel data processing, the original C#language version of the algorithm has been translated to java. Additional interfaces are in place to allow PACS (and SPIRE) data to be processed by MADmap. This implementation requires that the noise properties of the detectors are determined a-priori. These are passed to MADmap as PACS calibration files and referred to as the "INVNTT files" or “noise filters”.

The time streams must be free of any instrument artifacts and must be calibrated before MADmap can be used to create the final mosaic. This is a two#part process. The first part is the PACS Level 0 (raw data) to Level 1 (cleaned and calibrated images) pipeline processing. This is discussed in this section. The second part is MADmap pre-processing and how to run MADmap, which are discussed in the next section.

For most users standard Level 0 to Level 1 processing is normally sufficient. However, the method used for deglitching the data may have a significant and adverse impact on MADmap processing. For MADmap, we recommend and prefer the IInd level deglitching option. This option is not part of the automated (“standard”) pipelines; the alternative, the “standard” wavelet based “MMTdeglitcher”, does not perform optimally when it is allowed to interpolate the masked values. If the MMTdeglitcher is used, we recommend selecting the ‘maskOnly’ option to prevent replacing masked values with interpolated ones.

The HIPE tasks 'L05_phot' and 'L1_scanMapMadMap' are the recommended tasks for processing raw PACS data from L0 to L1 for the MADmap pipeline, and you can access them via the HIPE Pipeline menu: Pipeline->PACS-Photometer->Scan map and minimap->Extended source Madmap.

The HIPE task 'L25_scanMapMadMap' is the recommended task for processing PACS data from L1 to L2.5 for the MADmap map making, and you can access this via the HIPE Pipeline menu: Pipeline->PACS-Photometer->Scan map and minimap->Extended source Madmap. For optimal map-making with bolometer arrays, a scan and cross-scan observation is required (exceptions are when no extended emission is present or significant and one can rely on high-pass filtering techniques to remove bolometer noise). There is no L2 MADmap products for the very reason that the standard data processing works on a single OBSID. The Level 2.5 products are designed to combine scan and cross-scan observations belonging to multiple OBSIDs into a single final map, using L25_scanMapMadMap task.

*In this chapter we do not explain all the details of MADmap itself, rather we explain what you need to do to your PACS data to prepare it for MADmap. We then take you through the HIPE tasks that implement MADmap; but to learn more about MADmap itself, you should read that documentation. Some of the terminology we employ here comes from MADmap, and so we encourage you to read the MADmap documentation to understand this terminology, as well as to appreciate why the pre-processing steps we take you through here are necessary. You can look at http://crd.lbl.gov/~cmc/*

*MADmap/doc/man/MADmap.html and http://arxiv.org/pdf/0906.1775 or http://adsabs.harvard.edu/cgi-bin/bib_query?arXiv:0906.1775.*

# 5.2. MADmap pre-processing

The point of using MADmap is to account for signal drift due to 1/f noise while preserving emission at all spatial scales in the final mosaic. This is fundamentally different from the high#pass filter reduction, which subtracts the signal at scales larger than the size of the high#pass filter window. However, the MADmap algorithm, indeed most optimal map makers, assume and expect that the noise in the time streams is entirely due to the so#called 1/f variation of the detectors. The PACS bolometers show correlated drifts in the signal and these must be mitigated before MADmap can be used. The MADmap algorithm assumes that the noise is not correlated and so will (incorrectly) interpret the any systematic non-1/f-like drifts as real signal. Additionally, the PACS bolometers have pixel-to-pixel electronic offsets in signal values. These offsets must also be homogenised to a single base level for all pixels.

The mitigation of all of the above effects is referred to as MADmap preprocessing. In all, there are four types of corrections. We discuss each step below.

> **Warning**
>
> The MADmap preprocessing critically determines the quality of the final maps. Care must be taken to ensure that each step is optimally applied to achieve the best possible reduction. This may require repeating step(s) after interactively examining the results. Further, not all steps may be necessary. This is also discussed below.

## 5.2.1. Pixel-to-pixel offset correction

This is the most dominant effect seen in all PACS (photometry) signal readouts. For most single channel bolometers the offset is electronically set to approximately 0. The PACS bolometers are, however, multiplexed, and only the mean signal level for individual modules or array can be set to 0, leading to variations in the pixel-to-pixel signal level. This is purely an electronic and design effect.

Mitigation of this effect entails subtracting an estimate of what the zero level should be per pixel from all of the readouts of the pixel. There are two ways to estimate the zero level.

**(1)** Use a calibration zero-level image. The following snippet shows one example, for the blue filter, of how to access the zero#level image In HIPE.

```
calTree = getCalTree() # if not already done
zimage = calTree.photometer.corrZeroLevel["blue"].data
```

Where, “blue” in the square brackets can be replaced with another filter name. The resulting variable ‘zimage’ is a 2D floating#point array, which has been taken from the calibration tree (i.e. a calibration file provided by the PACS team), and this should be subtracted from each of the signal readout. The units for the zero#level image are volts. Therefore, the readouts must also be in volts.

**(2)** Use the median of signals in each pixel. This method works in any units (digital readout units or volts). The idea is to compute the median of the entire history of signal values per pixel and subtract this median from each signal. The task photOffsetCorr applies this correction in HIPE. For example,

```
frames = photOffsetCorr(frames, copy=<Integer>, zeroLevelCorr=<Integer>,
 regionSelection=<Selection>)
```

and the parameters of this task are:

- `frames` — Input *Frames*, MANDATORY, NO default value
- `copy` — default 0: input Frame is changed; 1: Frame is copied and that is returned
- `zeroLevelCorr` — default 0: median removal; 1: zero level calibration value subtracted

- `regionSelection` — default None: nothing is done; otherwise, median offset (for zeroLevelCorr=0) or zero level (for zeroLevelCorr=1) removal is doing using the region here specified.

Figure 2 shows the image that is the median of the signals in each pixel, which is what is done when you use photOffsetCorr.

**Figure 5.2. The map of the median of the signals in each pixel showing the pixel to pixel electronic offset.**

Figure 3 shows a single readout slice (one single time point) of an input Frame after the pixel#to# pixel offset correction using photOffsetCorr.

**Figure 5.3. A single slice (one single time-point) of a raw signal Frame after the offset (pixel to pixel) image removal step (i.e. after subtracting Figure 2 from the original Frames). While the pixel to pixel variation is mitigated, the result shows two modules are systematically at a different signal level than the rest.**

**Tip**

The photOffsetCorr task uses the on#target Status flag (OTF) to determine which readouts are to be used to estimate the offset values. This flag can be manipulated to specify, among other possibilities, which part of the sky (bound by right ascension and declination values) is suitable for estimating the offset values. Simply set the OTF flag to false. Then, set the OTF flag as true for all readouts that are within the boundaries of the sky region. An example for doing this is:

```
# Swath of sky within a certain ra boundary.
# The min and max acceptable ra values are specified by ra_1 and
 ra_2
#
ra_1 = 326.36
ra_2 = 326.42
#
ra = frames.getStatus("RaArray")
dec = frames.getStatus("DecArray")
sel = (ra>ra_1)&(ra<ra_2)
ind = sel.where(sel==True)
otf=frames.getStatus("OnTarget")
n = len(otf)
otf = Bool1d(n,False)
otf[ind]=True
frames.setStatus("OnTarget",otf)
# Do photOffsetCorr
#
# IMPORTANT: Reset the OTF flag after the photOffsetCorr
frames.setStatus("OnTarget",Bool1d(n,True))
```

# 5.2.2. Module to module drift correction

Figure 3 also shows that the blue focal plane readout, after the offset correction, has a systematically different signal level in the two modules appearing on the bottom left of the Figure than the remaining modules. (This is not something we expect you to see for red-detector data, please note.) If we had used a different offset correction, different pairs of modules may appear errant, but it will always be in pairs (1&2, 3&4, 5&6, or 7&8: the readout electronics are set up that way). Generally, the pair 1&2 is always systematically off. Mitigating this systematic difference in the signal level between individual modules is what we refer to as ‘module#to#module’ drift correction.

Figure 4 shows why drift is an apt description for this correction. In this figure we show the result from a more quantitative investigation, in which the median (over the pixels) of module 1 has been subtracted from the median (over all pixels) of module 5 (here the reference module); this difference is plotted against the readout index (or time) in Figure 4. The deviant signal “spikes” in the data are due to actual variations in the sky signal (i.e. sources). The apparent “break” in the trend around index

14800 is the difference between scan and cross-scan readouts that have been merged in this figure. Individual modules are colour-coded as labelled.

**Figure 5.4. The systematic module-to-module drift. The data are here are the median (over all pixels) of module 1 minus the median (over all pixels) of module 5, as a function of readout index/time. The most discrepant modules are 1&2 (shown in reddish tints) but all appear to show some drift in the median level of the frame.**

Figure 4 shows that the relative median levels of the modules are: (i) not the same, and (ii) change systematically as a function of readout (or time). As is the case for the pixel#to#pixel variation, the MADmap code will also not handle such correlated module#to#module variations.

At present, the recommended mitigation is as follows: (i) select only data with median level between +/-50 counts, or +/-0.1 volts, depending on the signal units. This will more or less ignore the real astrophysical variations in the signal. This is a practical, but not necessarily the optimal way to detrend module#to#module variations. (ii) Fit a straight line to the data as a function of the reset index and then subtract the fit from all pixels of the module.

In HIPE, the task photModuleDriftCorrection applies the steps listed above automatically. It is called, for example, as follows:

```
frames = photModuleDriftCorrection(frames)
```

See the documentation within the photModuleDriftCorrection task for information on optional parameters.

Figure 5 shows an example of linear fits to the module drifts. Some fits were affected by the larger variations caused by the real sky brightness variation over the scanned region. Even for these, the resulting discrepancy is small. There is no evidence that a higher order model is needed.

**Figure 5.5. The best-fit linear model to the module-to-module drifts (red line). Only one module example is shown.**

Figure 6 shows a single slice (a single time-point) of a Frames after the best-fit signal drift has been subtracted from each of the module's pixels.

**Figure 5.6. A single slice (time-point) of a Frame (from Figure 3) after the module-to-module correction as described in the text.**

After the module-to-module drift correction, the data should be such that systematic differences in the median level of the frame are not significant.

# 5.2.3. Correction for global signal drifts

Since the signal level for individual modules can drift in relation to the reference module, most assuredly we expect the reference module itself to drift as well. The effect of such a drift produces detector readouts that are systematically offset in signal from the beginning to the end of the scan in a monotonic way. Once again, this is a global correlated trend of the entire array and must therefore be removed prior to MADmap.

**Figure 5.7. The global median of the signal level as a function of the readout (here showing scan and cross-scan data). The question to answer is whether there a real trend downwards in the signal?**

Determining whether there is a systematic drift in the signal of the reference module is a difficult task, as there is no data that is constant to compare to. Figure 7 shows the median signal (of the entire array) as a function of the readout index (time). Does a trend exists, despite sources coming on and off the array? In this figure, the larger variations are from actual source/sky signals, which happen to dominate the readouts for this observation. The trend is more clearly visible in Figure 8 because the sky/source signal is much weaker compared to the background.

**Figure 5.8. The median value of the entire PACS array as a function of readout index (time). The monotonic trend seen here is due to the correlated drift in the signal of the entire array. The 1/f noise variations are much less significant compared to the overall signal drift.**

For data that are dominated by the background (as in Figure 8) the trend is relatively easy to model as source contamination is negligible. However, a more generic approach is needed that is also able to account for data that are similar to the one shown in Figure 7. To do this, we create 1000 readout-wide bins and assume that the minimum values in these bins will correspond to the few actual blank-sky/ background measurements. The idea is that over these 1000 readouts, at some point the scan pattern has observed a source#free part of the sky.

Figure 9 illustrates these minimum values plotted as a function of the readout index where they are found. The dots in Figure 9 are the actual minimum median values in each of the 1000 readout bins. The trend is now much easier to model. Figure 9 shows a 2nd order polynomial model to the data. The best#fit model (polynomial in this case) is then assumed to describe the correlated global drift and is subtracted from the each readout.

**Figure 5.9. The minimum value of the 1000-readout bin as a function of its corresponding readout index number (see text for details). The dots show the actual minima of the median. The line is a modelled fit to the data.**

The task photGlobalDriftCorrection applies this technique automatically. In HIPE the calling sequence is, for example:

```
frames = photGlobalDriftCorrection(inFrames=frames, model=<Integer>, copy=<Integer>,
 \
binsize=<Integer>, datadir=<String>, outprefix=<String>, doPlot=<Boolean>, \
slowMedian=<Boolean>, useMedian=<Boolean>, subFit=<Boolean>, order=<Integer>, \
verbose=<Boolean>)
```

With parameters:

- `frames` — a Frames type, input PACS frames with units of Jy/pixel (which will be the case if you have pipelined the data as explained in this, MANDATORY
- `model` — a Integer type, the model number to be used, default to 1, see below for more information
- `copy` — a Integer type, 1 to preserve input frames otherwise modified, default to 0
- `binsize` — a Integer type, the size of the bin to use for MIN/MEDIAN estimate, default to 1000
- `datadir` — a String type, specify where to store the plot, default to "./"
- `outprefix` — a String type, prefix to add to the filename for the plot when saving, default to "plot"
- `doPlot` — a Boolean type, show and save a plot of the baseline signal drift correction, default to False
- `slowMedian` — a Boolean type, compute the median by looping in jython. The alternative implementation is faster but may crash HIPE (see PACS-2279), default to False

- `useMedian` — a Boolean type, use MEDIAN instead of MIN. Only applicable to data where most of the signal is sky not source, default to False
- `subFit` — a Boolean type, for model 3, can either subtract the FIT or optionally the data itself, default to False
- `order` — a Integer type, order of the polynomial to fit to the baseline in all applicable models, default to 3
- `verbose` — a Boolean type, set verbose mode, currently not used, default to False

See the documentation for corrGlobalDriftCorrection for more information about running it.

> **Tip**
>
> To reduce (or prevent) frequent reprocessing of the same steps, it is recommended that you save your data prior to applying global signal drift corrections. This way, if you wish to examine the relative merits of various correction options, the processing up to this stage does not have to be repeated. Saving the Frames as FITS or to pool are both equally good options, or you can simply copy it into memory (newframe=frame.copy()).

## 5.2.3.1. Global drift correction models

The global signal mitigation affects the quality of final maps much more significantly than does the module#to#module signal drift correction. In this section we discuss what options exist for modelling the global drifts.

**Model 1**: This is the approach already discussed above. The fit to the baseline is a polynomial. For most data a 2nd order polynomial provides an accurate fit. For a significant fraction of the data, particularly those with less than few thousand readouts, a linear model has sufficient accuracy. This approach is usable for all data. This option corresponds to model=1 in photGlobalDriftCorrection().

**Model 2**: For data that are similar to the one shown in Figure 8, i.e., the signal is dominated by the background and not source emission, another option is to simply subtract the median value of the entire array from each pixel. This approach will fail when the sky emission is significant, even for a few readouts. This option corresponds to model=3 in photGlobalDriftCorrection().

A significant disadvantage for both options is that individual pixels are known to show small variations in their signal drift that is different from the global signal drift of the array. This leads to noisier maps than would be possible if drifts from individual pixels could be modelled. However, the magnitude of the additional noise component is relatively small. There is, unfortunately, no real cure for this for option 1. However, in some cases, option 2 can be modified as described below in option 3.

**Model 3**: Again for background-dominated data, individual pixel time-lines can be fit with a polynomial and subtracted. However, this approach has the disadvantage that for individual pixels the trend is less obvious and is also affected by the 1/f noise, so the fits may be worse (and also less correct). This option corresponds to model=2 in photGlobalDriftCorrection().

## 5.2.3.2. Which option to use?

For observations that are a few thousand readouts or less in size, we do not recommend any global signal drift correction. Over such a short time the detectors have not sufficiently drifted to cause a significant variation and so drifts will not adversely affect the final maps. The user can judge the need for this correction by examining the median values of the array for their data set. However, having stated that, we also note that observations taken at the beginning of the PACS observation day (OD) are much more susceptible to drifting signal than those taken at the end of the OD. (We are working on providing a task will tell you if this is the case.)

For observations longer than a few thousand readouts, we recommend option 1. The default setting should apply to most observations. For the best possible results, we recommend interactive optimisation described below. This is also the setting recommended for the default automatic pipeline.

If the observations are dominated by background (not source) emission, options 2 and 3 provide more accurate rendering of the final maps. However, care must be taken to first establish that the assumption is valid. Hence, this approach is not recommended for automated processing.

# 5.3. Optimising MADmap pre-processing

As mentioned earlier, the quality of the final map is strongly dependent on the quality of the preprocessing steps. The default settings now programmed for the various HIPE tasks/modules already introduced produce reasonable maps. However, baseline removal is a tricky art and the user can further optimise their maps by following the following suggested interactive steps.

## 5.3.1. Overriding defaults in option 1

Option 1 is implemented as model 1 in photGlobalDriftCorrection task. The bin size over which the minimum is calculated, and the order of the polynomial, are controlled by the user. Smaller bins more accurately model the drift provided that source emission does not creep into individual bins. The user must decide this for themselves by investigating (e.g. playing with) values for both parameters and examining the final fits.

## 5.3.2. Segmenting time-streams

For observations longer than ~2 hours, more global inflections become visible in the data. Figure 10 illustrates this behaviour.

**Figure 5.10. Option 1 and a 2nd order fit applied to an 8 hour observation. The best fit is reasonable towards the end of the observations but fails to match the inflections at the beginning of the observation.**

The first part of the observation is not well fit with the 2nd order polynomial model used. In fact, while not shown, we find no satisfactory fit for the data with any order polynomial. The best solution in this case came from segmenting the data into 30,000 readout groups. An example of the fit to one individual group of 30,000 was shown in Figure 9. This approach produces much more sensitive maps than a forced fit to all data. Figure 11 shows a comparison of the resulting maps with and without segmented fitting. The “structure” visible in the map without segmented fitting disappears when the drift correction is applied to individual segments. We attribute the difference to inaccuracies in modelling the signal drift without segmenting. This approach is not recommended for smaller datasets because: (i) not enough data are available for grouping, and (ii) when the segments become smaller than the scan length, actual spatial structure will be removed, hence, negating the use of MADmap.

**Figure 5.11. Comparison of final maps from default option 1 setting (top) and segmented option 1 fitting (bottom). The global signal drifts are much better corrected when large data sets are fit in smaller segments**

### 5.3.2.1. Optimal segmenting

We recommend that the minimum segment should be on the order of a few scan legs to ensure that spatial structure of the size of the map length are preserved, and enough data are present for proper binning. In reality, a combination of segment and bin size is best derived empirically be interactively examining the fits. A way to work out how long a scan leg is in readouts is to take your scan leg length in arcminutes and the scan speed in arcseconds (i.e. as was programmed in HSPOT when the observation template was written), and apply the following: (scan leg in arc-minutes * 60.0 / scan_speed in arcseconds per sec ) * 10 Hz.

## 5.3.3. “long-pass” filtering

Another way to combat the global drifts is to preprocess with a so-called “long-pass” filter. This is a variation of the high#pass filter except that filter window sizes of 500—3000 readouts are used.

The primary advantage is a more accurate removal of the global signal drift. However, as for the high-pass filter, the primary disadvantage of this approach is the removal of all spatial structures of sizes larger than the longpass filter width. The disadvantage is such that for data similar to those of Figure 7, significant filtering artifacts remain in the final maps. If a long#pass filter is used, then the photGlobalDriftCorrection task should be omitted from MADmap processing.

### 5.3.4. Other global drift correctors

The above methods are already available in HIPE. However, it should be noted that users are free to choose any other established or customised drift correction model that they deem appropriate, provided it can be programed in HIPE. Examples include: interpolated or tabulated drift correction per readout based on the minimum median values. In this case, the actual minimum median values (the dots in Figure 9, for example) are used to create an interpolated lookup table for each readout.

## 5.4. Running MADmap

After all that pre-processing, you can now create your map with MADmap.

### 5.4.1. Preparation

After pre#processing, in theory the only drift left in the signal should be due to the 1/f noise. Figure 12 shows the power spectrum of the data cube (averaged for all pixels) after the drifts have been accounted for. The a-priori selected noise correlation matrix for PACS is estimated from the Fourier power spectrum of the noise. The current MADmap implementation requires the following data to be present in the Frames object: *Camera, (RA, Dec) datasets, BAND, onTarget flag*. These data are generated during the Level 0 to Level 1 processing, or in the Level 0 product generation. MADmap will not work if any of the above dataset or status keywords are missing.

*Camera*. You should start with ‘Blue’ or ‘Red’. To check, use the following command in HIPE:

```
print frames.getMeta()["camName"]
{description="Name of the Camera", string="Blue Photometer"}
```

The *(RA, Dec) datasets* are the 3#dimensional double precision cubes of Right Ascension and Declination values generated during level 0#1 processing. In HIPE:

```
print frames[“Ra”].data.class
```

If an error occurs (provided no typos are present) then the Ra and/or Dec cubes simply has not been generated.

The *BAND* status keyword must have one of ‘BS’, ‘BL’, or ‘R’ values. If BAND does not have one of these values, MADmap will not work.

```
print frames[“Status”][“BAND”].data[0:10]
```

*OnTarget* status keyword. This in a Boolean flag under status and must have the value ‘true’ or ‘1’ for all valid sky pixels for MADmap to work. E.g.:

```
print frames[“Status”][“OnTarget”].data[0:10]
```

The data are now ready for MADmap.

**Figure 5.12. The power spectrum of the full data stream after the drift removals (averaged for all pixels). Some structure is expected due to the astrophysical sources and from the unremoved glitches (not the final figure, just a placeholder until I get the real one).**

# 5.4.2. makeTodArray

This task builds time-ordered data (TOD) stream for input into MADmap (basically it is just a reorganisation of the signal dataset of your Frames), and it creates meta header information for the output skymap. Input data is assumed to be calibrated and flat-fielded, i.e. it is assumed you have run the pipeline on the data, as specified in this chapter. The task also takes the "to's" and "from's" header information from the InvNtt calibration file, which is in our calibration tree. Finally, the task assumes that the BAND and OnTarget flags have been set in the Status, which will be the case if you have run the pipeline.

Terminology: "to's" and "from's" are terminology from MADmap (the details of which are not explained here, see the reference at the beginning of this chapter), and they are the starting and ending index identifiers in the TOD. "InvNtt" stands for the inverse time-time noise covariance matrix (it is written as N_tt*^-1). It is part of the maximum likelihood solution.

## 5.4.2.1. Usage

```
# Creates a PacsTodProduct
todProd = makeTodArray(frames=frames, scale=<a Double>, crota2=<a Double>,
 optimizedOrientation=<Boolean>, minRotation=<a Double>, chunkScanLegs=<Boolean>,
 calTree=<PacsCal>, wcs=<Wcs>)
```

With parameters:

- `frames` — A Frames type, Data frames with units of Jy/pixel (which will be the case if you have pipelined the data as explained in this chapter). Required inputs are: (1) RA,Dec datasets associated with the Frames including the effects of distortion; this would have been done by the pipeline task PhotAssignRaDec, (2) Mask which identifies bad data (in fact, a combination of all the masks that are in the Frames), (3) BAND Status information (BS,BL,R), AOT (observing) mode (scan/chopped raster). If you pipelined your data then these will all be in the Frames and there is nothing more for you to do
- `scale` — A Double type, Default = 1.0. Multiplication pixel scale factor for the output sky pixels compared to the nominal PACS pixel sizes (3.2" for the blue/green and 6.4" for the red). For scale = 1.0, the skymap has square pixels equal to nominal PACS detector size; for scale = 0.5, the sizes are 1.6" for the blue and 3.2" for the red
- `crota2` — A Double type, Default = 0.0 degree. CROTA2 of output skymap (position angle; see below)
- `optimizeOrientation` — A Boolean type, Default = false. If true, the projection will automatically rotate the map to optimise its orientations with respect to the array, and if false the rotation angle is 0 (north is up and east is to the left)
- `minRotation` minRotation — A Double type, Default = 15.0 degrees. Minimum angle for auto rotation if optimizeOrientation=true
- `chunkScanLegs` — A Boolean type, Default = true, on-target flags are used to chunk scan legs, i.e. to ensure that off-target data are not used
- `calTree` — A PacsCal type, Default = none, PACS calibration tree
- `wcs` — A Wcs type, Default = none, when users need to use a predefined Wcs
- `todProd` — A PacsTodProduct type, Output product, containing the TOD file name, the final output map's WCS, the so-called to and from indices for each good data segment and the correspondence between TOD indices and sky map pixels (i.e. so it is know what time-ordered indices came from which part of the sky)

The intermediate output TOD file is saved in a directory specified by the property var.hcss.workdir. As you do not need to interact personally with this file, it is not expected you will change this property

from the default (the file is removed after MADmap has been run and you exit HIPE), but if you wish to do so you can edit the Properties of HIPE yourself (via the HIPE menu).

The body of the TOD file is a byte stream binary data file consisting of header information and TOD data (see the MADmap references at the top of this chapter).

The output TOD product includes the astrometry of output map using the WCS, in particular meta data keywords such as:

```
CRVAL1 RA Reference position of skymap
CRVAL2 Dec Reference position of skymap
RADESYS ICRS 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)
```

## 5.4.2.2. Functionality

This is what happens as the task runs (i.e. this is what the task does, not what you do):

1. Build a TOD binary data file with format given above.

2. Define the astrometry of output map and save this as the keywords give above. Default CROTA2=0.0, but if optimizedOrientation=True, then call Maptools to compute the optimal rotation (i.e. for elongated maps). If rotation less than `minRotation`, then leave map un-rotated with `crota2`=0.

3. *Badpixels* — Dead/bad detector pixels are not included the detector in TOD calculations; they will not have good InvNtt data and hence are discarded for MADmap.

4. *Skypix indices* — Compute the skypix indices 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 of 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.

5. *Glitches* — Set weights to a BADwtval (bad weight value) for bad data as indicated in the masks (BADPIXEL, SATURATION, GLITCH, UNCLEANCHOP) of the Frames. For the BLINDPIXELs the default BADwtval is 0.0, but one may need to use a small non-zero value (e.g., 0.001) in practise (to avoid MADmap precondition that requires non-zero weights and data for each observed skypixel). Good data should have weights set to 1.0 for initial versions with `nnOBS`=1.

6. Chunk per scan leg — Use the OnTarget status flag to define the boundaries for data chunking per scan leg (i.e. separate data that is on- and off-source). The start and end boundaries of the TOD indices of each data chunk per detector pixel are needed for the InvNtt headers (the "tos" and "froms"). TOD rules: (1) Chunk for large gaps (>maxGap) defined by OnTarget (it is assumed that the telescope turn-around time will be larger than maxGap, but this is not a requirement). (2) for small gaps (<=maxGap) defined by OnTarget, use the data values for the TOD, but set the TOD weights=BADwtval (i.e. these data are effectively not used for the map, but are needed for a continuous noise estimate for MADmap). This condition is not expected for properly handled data products upstream, but could exists if there are issues with the pointing products. ??? (3) Include an option to turn-off chunking by scan leg.

7. Chunk as function of time per detector pixel, based on the mask information — Check the TOD stream per detector pixel. For "gaps" of bad data in samples larger than maxGap, then chunk the data. Ignore data streams that have a number of samples less than minGOOD, i.e., each TOD chuck should be larger or equal to minGOOD samples. For gaps smaller or equal to maxGap, linearly interpolate the TOD values across the gap and set TOD weights=BADwtval (i.e. these data are

not used for the map, but are needed for a continuous noise estimate for MADmap). TOD rules (in this order):

a. throw out initial and end samples that are bad.

b. fill in the small bad GAPS (<=maxGap), weights=BADwtval

c. chuck for large bad GAPS (>maxGap)

d. throw out small chucks (<minGOOD)

The initial default `maxGap` value is 5 and `minGOOD`=correlation length of the InvNtt calibration data. The locations defining the boundaries of the good chunks are stored on a per detector pixel basis (the "tos" and "froms"). Note that (6) and (7) have a similar logic [(6) is based on OnTarget and (7) is based on the mask information]. In both cases, chuck for gaps>maxGap. For gaps<=maxGap, linearly interpolate data values across the gaps for (7) [i.e. glitches], but use the data values for (6) [i.e. nothing is wrong with the data values].

# 5.4.3. runMadMap

This is a wrapper script that runs the MADmap module (a java code). Input TOD data is assumed to be calibrated and flat-fielded and input InvNtt is from calibration tree.

## 5.4.3.1. Usage

```
# Creates a SimpleImage
map = runMadMap(todproduct=tod, calTree=calTree, maxerror=<a Double>,
 maxiterations=<an Integer>, runNaiveMapper=<Boolean>, useAvgInvntt=<Boolean>)
```

With parameters:

- `tod` — (class PacsTodProduct) the result of makeTodArray, MANDATORY
- `calTree` — Pacs calibration tree. For your information: the particular calibration file used has the 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 pixel. MANDATORY.
- `maxerror` — Default = 1e-5, maximum relative error allowed in PCG routine (the conjugate gradient routine, which is part of MADmap)
- `maxiterations` — Default = 200, maximum number of iterations in PCG routine
- `runNaiveMapper` — Default = false, run MadMapper; when true, run NaiveMapper (i.e. something that is similar to what you would get with the pipeline task photProject, but less accurate.
- `useAvgInvntt` — Default = false, use InvNtt data for each pixel; when true, use InvNtt data averaged for all pixels in a detector

Two calls are needed if you want to produce both the MadMap and the NaiveMap simple image products (runNaiveMapper=true yields Naivemap product and runNaivMapper=false yields MadMap product). The NaiveMap image will be similar, but less accurate, than what is produced by the pipeline task photProject. The output from a single run on runMadMap is:

- map — Output product consisting of following:
  1. image — Sky map image (either a Naive map or MADmap) with WCS information
  2. coverage — Coverage map corresponding to sky map, with WCS (units are seconds, values are exposure time)
  3. error — Uncertainty image associated with the map, with WCS

The error map is currently only made properly for NaiveMapper, although note also that error maps do not reflect all the uncertainties in the data: this is an issue we are still working on. As this task runs there is an intermediate output TOD file created, this is saved in a directory specified by the property var.hcss.workdir (as mentioned before, there is not need for the user to interact with this file, it is removed after MADmap finishes and at the exit of HIPE, but if you wish to have it saved somewhere else, you will need to change properties via the HIPE preferences panel).

### 5.4.3.2. Functionality

1. Build InvNtt from input calibration tree.

```
Format of InvNtt chunk:
Header-LONG:
min_sample starting sample index
max_sample last sample index
n_correlation correlation width of matrix
Data-DOUBLE:
invntt(n_correlation+1)
```

   The min/max samples are the "tos" and "froms" calculated from a method within makeTodArray. The sample indices need to be consistent between the TOD chunked files and the InvNtt file.

2. Run MADmap module.

3. If runNaiveMapper = true run NaiveMapper module.

4. Put astrometric header information into output products.

# 5.5. MADmap post-processing

## 5.5.1. Introduction

When Genearlised Least Square (GLS) approaches, like MADmap or ROMA, are employed to produce sky maps from PACS data, the resulting maps are affected by artifacts in the form of crosses centered on bright point sources. These artifacts are annoying and make the GLS images difficult to use for astrophysical analysis. This is a pity since GLS methods are otehrwise excelent map makers. This problem is tackled by the Post processing for GLS (PGLS) algorithm, that will be briefly described in the following sections. The algorithm effectively removes the artifacts by estimating them and by subtracting the estimate from the GLS image, thereby producing a clean image. PGLS was devised within the Herschel Open Time Key Program HIGAL project and exploited in the HIGAL processing pipeline. The development was funded by the Italian Space Agency (ASI). A more detailed description of PGLS can be found in [R1, R2].

## 5.5.2. Map making basics

The output of a PACS scan is a set of bolometers' readouts with attached pointing information. Each readout gives the power measured at the corresponding pointing plus a noise term. The readouts can be organised into a matrix d = { d(n,k) } where the the element d(n,k) is the k-th readout of the n-th bolometer. The matrix d is termed the Time Ordered Data (TOD).

The map making problem is that of constructing an image (map) of the observed sky from the TOD. The typical approach is that of defining a pixelization of the observed sky, i.e. partitioning the sky into a grid of non overlapping squares (pixels). The map maker has to produce a map which is a matrix m = { m(i,j) } where m(i,j) is a measure of the power received by the pixel in the i-th row and j-th column of the sky grid.

A simple and important map making technique is the rebinning. In the rebinned map the value of each pixel is set equal to the mean value of all the readouts falling in the pixel. Rebinning is a computa-

tionally cheap technique and is capable of removing well white, uncorrelated noise. Unfortunately the PACS data is normally affected by 1/f, correlated noise too. As a result, rebinning is a poor map maker for PACS data. An example of rebinned map is shown in figure 13, where the impact of the correlated noise can be seen in the form of stripes following the scan lines.

**Figure 5.13. Impact of the correlated noise in the form of stripes following the scan lines.**

The GLS approach is an effective map making technique, exploited by map makers like MADmap and ROMA. The GLS approach is effective in removing both white and correlated noise and has a feasible computational complexity, albeit higher than simple rebinning. Unfortunately, when used to process PACS data, the technique introduces artifacts, normally in the form of crosses placed on bright point sources. An example of GLS map where such an artifact is clearly visible is shown in figure 14. The artifacts are due to the mismatches between the GLS assumptions and the actual physical process, e.g. the error affecting the pointing information of each readout, which are better analysed in [R2]. The artifacts are annoying and make the GLS image less usable in astrophysical analysis.

**Figure 5.14. Point source artifact in a form of crosses places on bright point sources.**

Notation: in the following we write m = R(d) when the map m is obtained from the TOD d by rebinning. And m = G(d) when the map m is obtained from the TOD d by the GLS approach.

## 5.5.3. Unrolling and Median filtering

Let us introduce two additional operations that are needed to describe the PGLS algorithm. The first operation is termed the unrolling and produces a TOD d from a map m. In particular, given the pointing information and a map m, the unrolling of the map amounts at producing a TOD where each readout is set equal to the value of the corresponding pixel in the map, as specified by the pointing information. In other words, the resulting TOD is the data that would be obtained if the sky was equal to the map.

The second operation is termed the residual filtering and is based on the median filtering of a sequence. Given a sequence x[n] and an integer h, the median filtering of x[n] is the sequence y[n] = median( x[n-h], x[n-h+1], ..., x[n], ..., x[n+h-1], x[n+h] ). In words y[n] is the median value of a window of 2h +1 samples from the x sequence, centered on the n-th sample. Now the the residual filtering can be defined as r[n] = x[n] - y[n], that is r[n] is obtained by subtracting from x[n] the corresponding median. Residual filtering is a non-linear form of high-pass filtering and is very effective in removing correlated noise. Specifically, all the harmonics of x[n] below the normalised frequncy of 1/(2h+1), that is with a period longer that 2h+1 samples, will be greatly attenuated. Finally note that residual fitering can be applied to a whole TOD d, by applying it separately to the data sequence of each bolometer.

Notation: in the following we write d = U(m) when the TOD d is obtained by unrolling the map m. And t = F(d) when the TOD t is obtained by residual filtering the TOD d.

## 5.5.4. PGLS algorithm

The Post-processed GLS (PGLS) map making algorithm starts from the TOD d and the GLS map m_g = G(d). It is based on the following basic steps, that aim to produce an estimate of the artifacts affecting the GLS map:

1. Unroll the GLS map: d_g = U( m_g ).

2. Remove signal: d_n = d_g - d.

3. Remove correlated noise: d_w = F( d_n ).

4. Compute artifacts estimate: m_a = R( t_w ).

The functioning of the basic steps can be explained as follows. The original TOD contains the signal S, the correlated noise Nc and the white noise Nw, so that, using the symbol # to indicate the components, we write d # S + Nc + Nw. Assuming that the GLS map maker perfectly removes the noise but introduces artifacts, m_g contains the signal S and the artifacts A, m_g # S+A. The same components are there in the unrolled TOD computed in step 1, d_g # S+A$. By subtracting the original TOD from d_g in step 2 we are left with a TOD d_n where the signal is removed and the noise (with changed polarity) and the artifacts are introduced, d_n # A - Nc - Nw. By performing the residual filtering of d_n with a proper window length (to be discussed soon), we eliminate the correlated noise while the artifacts and the white noise are preserved, d_w # A - Nw. By rebinning d_w in step 4, we eliminate the white noise so that m_a # A$ is an estimate of the artifacts. In practice, since the GLS map maker, the residual filtering and the rebinning do not perfectly remove the noise, m_a # A + Na where Na is the noise affecting the artifact estimate.

The basic steps just described can be iterated in order to improve the artifact estimate. In this way we obtain the PGLS algorithm, producing a PGLS map m_p from the TOD d:

1. Initialize PGLS map and artifacts estimate: m_p = G(d), m_c = 0.

2. Repeat following 1-5 steps until convergence:

   a. Unroll: d_g = U( m_p ).

   b. Remove signal: d_n = d_g - d.

   c. Remove correlated noise: d_w = F( d_n ).

   d. Estimate artifacts: m_a = R( d_w ).

   e. Improve PGLS map: m_p = m_p - m_a.

In the procedure, at each iteration more artifacts are removed from the map and eventually the PGLS map is obtained. Examples of the PGLS map m_p and of the artifacts estimate are shown in figure 15 and 16 respectively. One sees that the artifacts are removed and that the only drawback is a slight increase in the background noise, barely visible in the figures. This is due to the artifact noise, Na, which is injected into the PGLS map in step 2.5.

**Figure 5.15. Post-processed image with the artifacts removed.**

**Figure 5.16. The point source artifacts that were removed.**

# 5.5.5. Results

The PGLS was analysed by means of simulations, using a known, synthetic sky and the corresponding target map. It was verified that PGLS effectively removes the artifacts without introducing too much noise, because the PGLS map is closer to the target map than the GLS map in the mean square sense.

It was verified that the convergence criterion is not critical, because the artifact estimate rapidly approaches zero meaning that both the corrections and the noise injection decrease. Usually some four or five iterations are enough to get most of the improvment and the convergence criterion can be when the mean square value of the estimate is low enough.

It was verified that the only parameter of the algorithm, namely the median filter window length, is not critical. While shorter/longer windows cause less/more noise to be injected and less/more artifacts to be removed, good results are obtained in a wide range of values for the window length. A rule of thumb for this parameter is to set it equal to the width of the arms of the artifacts' crosses, which is easily estimated by visual inspection of the GLS image.

### 5.5.6. References

[R1] L. Piazzo: "Artifacts removal for GLS map makers", University of Rome "La Sapienza", DIET Dept., Internal Report no. 001-04-11, January 2011.

[R2] Lorenzo Piazzo, David Ikhenaode, P. Natoli, M. Pestalozzi , F. Piacentini and A. Traficante: "Artifact removal for GLS map makers", Submitted to the IEEE Trans. on Image Proc., June 2011.

### 5.5.7. Usage

A Java task PhotCorrMADmapArtifactsTask is used for this purpose. Its Jython syntax is shown below.

```
# Create a MapContext that contains both the corrected image and the artifacsts map
result = photCorrMadmapArtifacts(frames=frames, tod=tod, image=image, niter=<a
 Integer>, copy=<an Integer>)
```

With parameters:

- `result` — a SimpleImage type, the result of photCorrMadmapArtifacts, contains both the point source artifacts correted image and the artifacts map, MANDATORY
- `frames` — a Frames type, the input PACS frames, MANDATORY
- `tod` — a PacsTodProduct type, the input time ordered data (tod), MANDATORY
- `image` — a SimpleImage type, the input MADmap image, MANDATORY
- `niter` — a Integer type, the number of iteration, default value 1, OPTIONAL
- `copy` — a Integer type, copy=1 is to preserve input arguments, otherwise input arguments will be modified, default value 0, OPTIONAL

## 5.6. Open issues and known limitations

The following items are known limitations of MADmap processing:

### 5.6.1. Computing requirements

As a general rule of thumb, MADmap requires a computer of M * 1GB * Duration of observation in hours. M is dependent upon many factors, including the preprocessing and deglitching steps and the output pixel size of the final maps. For nominal pixel scales (3.2”/pix and 6.4”/pix in the blue and red channel, respectively), the value of M is typically 8 for the blue channel, and significantly less for the red channel.

## 5.7. Troubleshooting

The accuracy of this map compared to high-pass filtered maps is discussed in a companion report RD2. In summary, MADmap-produced maps have absolute flux levels consistent with those produced with photProject.

### 5.7.1. Glitches in the readout electronics

If a cosmic ray (or a charged particle) impacts the readout electronics, the result may be a significant change in the drift correction for the array (or the module) as a whole (the detector response is affected not just at the time of the strike, but for a time thereafter also). Figure 17 illustrates this.

**Figure 5.17. The minimum median (see Figure 9) plotted versus readout index. There appears to be a change in the magnitude of the drift, likely caused by a cosmic ray or charged particle impact on the readout electronics. You can see this by the break in the lines that fit the data: the scan direction data are described by a green and pale brown line ("scanA fit" and "scanB fit"), which do not have the same slope; and similarly for the cross-scan reddish and purple lines ("XscanA fit and "XscanB fit").**

Figure 18 shows the smoking gun that indicates a glitch caused the global drift to change.

**Figure 5.18. An expanded region of time-ordered data, near where the drift shows an abrupt change in magnitude in Figure 17. There is a clear break in the signal near readout value 9900**

The only possible remedy is to segment the data before and after the glitch even and fit the global drift separately. Segmenting of the data was explained in Sec. 7.3.2.

## 5.7.2. Improper module-to-module drift correction

If the inter-module drift is not corrected, the results will look similar to what is shown in Figure 19.

**Figure 5.19. The final mosaic with a clearly visible "checkered" noise pattern super imposed on the sky. This artifact is due to improper correction for the module-to-module drift**

## 5.7.3. Point source artifact

Figure 20 shows an example of what is typically referred to as the “point source artifact.” The artifact appears around bright point sources only and manifests itself as dark bands in the scan and cross-scan directions. The origin of the artifact is not well understood except as possibly due to misalignment of the PSF with respect to the final projected sky grid. This misalignment results in incorrectly assigning the peak and the wings of the PSF to the same sky pixel, resulting in a large deviation in the sky signal. When MADmap attempts to solve for the maximum likelihood solution to such a distribution of the signal values, it fails to achieve the optimal solution at the PSF +/- the correlation length.

The point source artifact has been successfully corrected using the method described in section 5.

**Figure 5.20. An example of the point source artifact around a very bright source. The MADmap reduction creates regions of negative (dark) stripes in the scan and cross-scan direction centred on the point source.**

# Chapter 6. Appendix

## 6.1. Introduction

In this appendix we explain:

- PACS *ObservationContext* and other products: what they are, what they contain, and how to work with them: Sec. 6.2.
- The Meta data, Status table and BlockTables of PACS products: what they are, what they contain, and how to look at them: Sec. 6.3.
- the PACS product viewers which work for photometry and spectroscopy: Sec. 6.4.

The following (Help) documentation acronyms are used here: *URM*: User's Reference Manual; *PDRG*: PACS Data Reduction Guide; *DRM*: Developer's Reference Guide; *PDD*: Product Description Document.

Note that this appendix is continually being updated, and is currently a little lacking for photometry.

## 6.2. PACS products: what they contain

Here we introduce you to the way PACS products are arranged. For spectroscopy and photometry the idea is the same, although the details of what the datasets hold naturally differ. *Unless otherwise stated, the following explanations are based on spectroscopy*. For spectroscopy we will explain: **ObservationContext**, **SlicedFrames**, **SlicedPacsCube**, **SlicedPacsRebinnedCube**, **Frames**, **PacsCube**, **PacsRebinnedCube**, **SpectrumSimpleCube**.

You can also check the *PDD*, which is a general HCSS document and confined mainly to detailed tabular listing of the contents of the various observation products you will find in HIPE.

Italics are used to indicate a Product class: so *ObservationContext* is used to indicate a product of class "ObservationContext". Different product classes have different "capabilities", hold information of different types and in different ways, and can be accessed by different task and GUIs. For the user, the main reason to know the class of a product is so that you know what tasks and tools will work on them, and you can know how to manipulate them on the command line. To learn how to use different classes in their full glory, you need to read the PACS *DRM*, which lists all their methods.

### 6.2.1. The ObservationContext

The first product you will work on is the *ObservationContext*, which is what you get from the HSA if you import the data directly or if you get it from a pool on disk (Sec. ???). An *ObservationContext* is a grand container of all the individual products and datasets that you get with each observation. To get an overview of its contents you should open the *ObservationContext* with the Observation viewer (see the *HOG* sec 15.2): right-click on it in the Variables panel or the Outline panel, and the viewer will open in the Editor panel:

**Figure 6.1. An ObservationContext viewed with the Observation Viewer**

In the upper part you see a Summary tab and the Meta data tab, at the lower left you see the directory-like listing of the layers in the *ObservationContext*, and on the right of that is an area where the default viewer for the selected layer, if such a viewer exists, will open (note that the default viewer

will open if you even single click on an appropriate product on the left side, and some products may take a while to load).

Let us go through these **products** one by one. In the lower left directory listing you will see something like this:

**Figure 6.2. The Levels directory listing in an ObservationContext**

What you see here is the directory of contents of the *ObservationContext* (called "obs" in the figure). Listings in red have not yet been loaded into memory, those in black have been; to load a product into memory you simply have to access it (e.g. click on it). The data that you will be interested are located in "+level0", "+level0_5", "+level1", and "+level2", these being the various raw or reduced Levels of your source at different stages of processing (0=raw, 0_5=partially processed, 1=more processed, 2=final). The other listings are additional data that you get with your observation, some of which are used in the pipeline and others of which are mostly of interest to the instrument scientists. The calibration layer contains the calibration tree that was used to reduce these data (see Sec. 2.5 for more information on the calibration tree). The auxiliary layer contains satellite and instrument data used in the pipeline, these mainly being to do with calibrating the pointing and timing. The History contains a history of the reductions, and there should be a history for every layer you can look at, and its history will be specific to that layer.

When you click on + next to the "level0/0_5/1" you will see this listing of contents of these observation data:

**Figure 6.3. The level contents**

If you hover with your cursor over these various HPSXXX layers (HPS=Herschel Pacs Spectroscopy), a banner comes up telling you what class the products are: for example, the level 0.5 HPSFITR|B are *SlicedFrames*: this being a type of *ListContext*, which is a special container (list) of other products (in this case, a list of *Frames*). There are several products held in the different levels (calibration, auxiliary, satellite, housekeeping, pointing...), and some products have several datasets which contain the actual information.

The level-layers the astronomer will be interested in are: HPSFIT[B|R], HPS3D[B|R], HPS3DP[B|R] and HPS3DP[B|R], which respectively are the *ListContext*s that contain the *Frames* (held in a *SlicedFrames* flavour of a *ListContext*), *PacsCubes* (held in a *SlicedPacsCube* list), *PacsRebinnedCube* (held in a *SlicedPacsRebinnedCube* list) and the final projected cubes which are of class *SpectralSimpleCube* (held in a standard *ListContext*), all for the blue and the red (B|R) camera data. The *Frames* are Level 0 to 1 products, the *PacsCubes* are Level 1 products, and the *PacsRebinnedCube* and the *SpectralSimpleCube* are the two final, Level 2 products. The other layers you can see in the screenshot above contain engineering (HPSENG), satellite housekeeping (HPSGENK and HPSHK), DecMec data (HPSDMC) which are used in some of the masking pipeline tasks, and data of the central detector pixel which is downlinked in raw format (HPSRAW). (You can also see the *PDD* to find definitions of the various HPSXXX and HPPXXX.)

By looking into the level0 HPSFITB (click on the + next to it) you can see the layers inside of it. For this product at Level 0 there will always only be one layer (numbered 0), which is a single *Frames* class product. The Level 1 HPSFITB, in the screenshot above, has several *Frames* since these data are "sliced", at the end of Level 0.5, by the pipeline into separate bits. A *Frames* itself hold various other products containing your astronomical and associated data: you can see in the screenshot below some of these, being the Status table, Mask, the BlockTable (see Sec. 6.3 for these) and the Signal. At Level 0.5 and 1, also shown in the screenshot, some pipeline processing has been carried out and so (i) there are more *Frames* in the listing, these being the original *Frames* sliced, according to an astronomical logic into smaller *Frames* units (see the spectroscopy pipeline chapters to learn more,

or look at the *URM* for "pacsSlicedContext"), and (ii) inside each *Frames* there will be new contents, e.g. the datasets such as Ra, Dec, Wave.

**Figure 6.4. The layers in the Level 0 and 0.5 product of an ObservationContext**

At Level 1, shown in the figure below, you see the same slices and much the same datasets. In addition at Level 1 there is also a *PacsCube* layer—HPS3D[B|R]—and it will have its own datasets.

**Figure 6.5. The layers in the Level 1 product of an ObservationContext**

As you can see in the figure above, next to the HPSXXX is text, this being a short listing of what slices are in them: L[number] means line(or range) number, not Level; N[number] means nod repetition number; A/B mean nod position A or B; R[number number] is the raster X and Y position. You can see a fuller version of the text in the tooltip (hover the mouse over the e.g. +1 and a tooltip will appear. These information are based on the BlockTable (see Sec. 6.3.3) and will be similar to the listing you see when you use the task slicedSummary on your product.

Finally there is the Level 2, inside which you will find two cubes: the rebinned cube HPS3DRR|B (*PacsRebinnedCube*) and the projected cube HPS3DPR|B (*SpectralSimpleCube*). These cubes have their own datasets, as you can see in the image below (this is a somewhat old screenshot but the organisation it demonstrates has not changed). In this screenshot you can see that there are two layers, called +0 and +1, for the HSP3DRB, and each of these contain another nine layers; within these final layers are the actual *PacsRebinnedCube*s. The reason for this structure is that the observation that this particular screenshot was taken from is a raster of nine pointings with two spectral lines observed at each pointing: the HPS3DRB layers are organised by wavelength and then by raster-pointing. However, the HPS3DPB entry has only two layers (+0 and +1), which are the two wavelength ranges, and each contains a single *SpectrumSimpleCube*, this cube being the mosaicked combination of the nine raster pointings.

**Figure 6.6. The layers in the Level 2 product of an ObservationContext**

In the following sections we explain the individual products for spectroscopy and photometry, and define their datasets.

If you want to open a GUI on "my observation data" to inspect it, at Level 0 and 0.5 it is the individual *Frames* that you will look at , and at Level 1 and 2 it will be the individual cubes.

## 6.2.2. Spectroscopy: *Frames, PacsCube, PacsRebinnedCube, SpectralSimpleCube*

The spectroscopy pipeline products are *Frames*, *PacsCube*s, *PacsRebinnedCube*s and *SpectralSimpleCube*s. In the pipeline they are held in *ListContext*s, see Sec. 6.2.3 to understand this lists. The PACS pipeline products, as you would have seen in some of the figures in Sec. 6.2.1, are multi-layered, these variously including: a BlockTable, Meta data and a Status table (Sec. 6.3), Masks (Sec. ???) and various datasets that have been produced by the pipeline.

**Table 6.1. The contents of PACS pipeline products**

| Name | What it is / task that made it | What product it is found in |
|---|---|---|
| BlockTable (Sec. 6.3.3) | organisation table of data blocks / created by findBlocks and added to by a few other tasks | *Frames, PacsCube* |
| coverage | a measure of the amount of data from the input *PacsRebinnedCubes* that became data of the *SpectralSimpleCube*, one value per wavelength and per spaxel / specProject | *SpectralSimpleCube* |
| exposure | a measure of the amount of data from the input *PacsCube* that became data of the output *PacsRebinnedCube*, one value per wavelength bin and per and spaxel / specWaveRebin | *PacsRebinnedCube* |
| fittedflux | to be ignored | *PacsCube* |
| flag | the saturation and rawsaturation masks carried forward by specWaveRebin, but held as an HCSS-defined "flag" rather than being of the "Mask" type of the prior products / specProject | *PacsRebinnedCube* |
| flux | the flux dataset in units of Jy / specRespCal | *Frames, PacsCube* |
| image | the flux dataset in Jy (taken from the "flux" dataset) / specRespCal | *PacsRebinnedCube* |
| ImageIndex | the wavelength dataset in micrometres (created from the "wave" and "waveGrid" datasets) / specWaveRebin | *PacsRebinnedCube* |
| Mask (Sec. ???) | contains all the individual masks / there from the Level0 and added to as pipeline proceeds | *Frames, PacsCube* |
| qualityControl | to be ignored | *PacsRebinnedCube* |
| ra, dec | RA and Dec datasets / specAssignRaDec | *Frames, PacsCube, PacsRebinnedCube* |
| signal | signal dataset, in units of counts / already there at Level0 but changed by specConvDigit2VoltsPerSecFrames and rsrfCal | *Frames* |
| startAndEndtime | to be ignored | *PacsCube* |
| Status (Sec. 6.3.2) | an table of the status of PACS during the entire observation / there from the Level0 and added to as pipeline proceeds | *Frames, PacsCube* |
| stddev (Sec. ???) | the standard deviation dataset (not an error dataset, but a measure of the scatter in the spectrum) / specWaveRebin | *PacsRebinnedCube* |
| wave | wavelength dataset in micrometres / waveCalc | *Frames, PacsCube* |
| waveGrid | the wavelength grid dataset created by wavelengthGrid and used to create the *PacsRebinnedCube* / wavelengthGrid | *PacsRebinnedCube* |

The methods you will use to access these parts of these products can be found in their PACS *DRM* entries, and in Chaps ??? and ???, where we show you various ways of interacting with your pipeline products, you will find examples of the methods you are most likely to use.

# 6.2.3. Spectroscopy: Sliced Products

In the spectroscopy pipeline the various products (*Frames* etc., Sec 6.2.2) are sliced, that is split into separate products wherever changes in pointing, wavelength and etc. occur, this being done to make handling them in the pipeline easier. The containers of the *Frames* etc. are a flavour of *ListContext* (which is a product class that is simply a list that holds products). Our *ListContext*s that contain the *Frames* or *PacsCube*, *PacsRebinnedCube* or *SpectralSimpleCube* slices are respectively of flavour *SlicedFrames*, *SlicedPacsCube*, *SlicedPacsRebinnedCube*, and just plain *ListContext*.

To work out what slice contains what you can use the pipeline task slicedSummary (see its *URM* entry learn more):

```
# For a SlicedFrames taken straight from the ObservationContext
HIPE> slicedSummary(obs_level0_5_HPSFITB)
noSlices: 5
noCalSlices: 1
noScienceSlices: 4
slice#  isScience  onSource  offSource  rasterId lineId band  dimensions
 wavelengths
0   false  no    no     0 0   [0,1]  ["B2A","B3A"]   [18,25,1680]  50.389 - 76.325
1   true   yes   no     0 0   [2]    ["B2A"]         [18,25,1500]  71.932 - 73.922
2   true   no    yes    0 0   [2]    ["B2A"]         [18,25,1500]  71.932 - 73.922
3   true   no    yes    0 0   [2]    ["B2A"]         [18,25,1500]  71.932 - 73.922
4   true   yes   no     0 0   [2]    ["B2A"]         [18,25,1500]  71.932 - 73.922
```

You can work with these SlicedXXX as a single unit or you can interact with the slices therein.

Our PACS SlicedXXX, since they are products, can contain more that just the slices: this means a History listing and Meta data, and for *SlicedFrames* also a MasterBlockTable. The "BlockTable" is explained in Sec. 6.3.3—essentially it is a table of the data blocks, where a "block" is a group of data-points taken while the instrument was in a particular setting: a particular wavelength, or nod position, or position in a raster pointing pattern, for example. While a BlockTable is the organisation table for a single *Frames* or cube, the MasterBlockTable is the concatenation of all the individual BlockTables of the *Frames* or cubes held in a *ListContext*. The output from slicedSummary is based on the MasterBlockTable. Sec. 6.3.3 shows you how to inspect the BlockTable that each *Frames* or cube product has, and the MasterBlockTable of your SlicedProduct can be inspected in the same way.

**Figure 6.7. That you can find a MasterBlockTable and Meta data in a SlicedFrames**

There are various methods you can use to interact with these sliced products (see the PACS *DRM* entries for the particular product to learn more). For the average user the most commonly-used methods will be those that allow you to identify and extract slices. A summary is provided here:

```
# SlicedFrames, SlicedPacsCube, SlicedPacsRebinnedCube: extract a slice
# The example is from a sliceFrames, but replace "Frames" with "PacsCube"
#  or "PacsRebinnedCube" and the same will work on the cubes

slice = 1
# get slice number "slice" (where 0 is the first slice)
# product returned is a Frames
frames = slicedFrames.get(slice)
# get science (as opposed to calibration) slice number "slice"
# product returned is a Frames
frames = slicedFrames.getScience(slice)
# manual method to get slice number "slice"
# product returned is a Frames
frames = slicedFrames.refs[slice].product
```

```
# get a slice another way
# product returned is a SlicedFrames
slicedFrames_sub=slicedFrames.selectAsSliced(slice)
# Using a task; see the URM to see all the
# parameters you can use. ASlicedFrames is returned
slicedFrames_sub = selectSlices(slicedFrames, sliceNumber=[slice])

# replace, add etc. (will work for all PACS sliced products)
slicedFrames.add(myframe1) # adds a new frame to the end
# inserts a new frame before the given slice no. (1 here):
slicedFrames.insert(1,myframe1)
# removed slice no. 1 (which is the second slice, not the first):
slicedFrames.remove(1)
# replaces slice no. 1 with a new frames:
slicedFrames.replace(1,myframe)

# concatenate any number of SlicedXXX using a task taken from the pipeline
sliceFrames_big = concatenateSliced([slicedFrames1, slicedFrames2])

ListContext:
# For the SpectralSimpleCube, the Level 2 "projected" cube, the
# result is held in a basic ListContext

slice = 1
# manual method to get slice number "slice"
# product returned is a SpectralSimpleCube
pcube = slicedPCubes.refs[slice].product
```

For the full details of the methods that work on PACS sliced products, look up "SlicedFrames" etc. in the PACS *DRM* and also look up *SlicedPacsProduct*, which is a class that these other sliced product come under. For the *ListContext* you need to consult the HCSS *URM* for its entry. You can also extract from a SlicedXXXX in a more complicated way by selecting on the MasterBlockTable, see Sec. 6.3.3.

> **Note**
>
> As with all other products in HIPE, their names (e.g. obs_level0_5_HPSFITB in the example above) are not the actual "thing" itself, but rather a pointer to the address in memory where the "thing" is held. That means that if you extract a slice out of a SlicedXXX and change it, you are not changing the copy only but also the original, because they are both pointing to the same address. Thus, if you want to work on a copy and not have any changes you make be done also to the original, you need to make a "deep copy" of the original:
>
> ```
> # A deep copy of a SlicedFrames
> slicedFrames_new = slicedFrames.copy()
> ```
>
> This will work also for a *SlicedPacsCube* and *SlicedPacsRebinnedCube* but not for the *ListContext* that contains the projected cubes, the *SpectralSimpleCube*s that are the output of the pipeline task specProject.
>
> The output of the pipeline task selectSlices, which reads a SlicedXXX in and writes a SlicedXXX out, also creates a fully independent product.
>
> The same applies for *Frames* and all the cubes: if you copy/take a slice out of a sliced product and want that to be independent you also need to make a deep copy, following this syntax:
>
> ```
> myframes1=slicedFrames.get(1).copy()
> myframes1=slicedFrames.refs.[1].copy()
> ```

# 6.3. Information and organisation tables in PACS products

In Sec. 6.2 the layout of the various PACS products is explained, and there you will see that many of our products contain: Meta data, [Master]BlockTable and a Status table. Here we explain what these are.

# 6.3.1. Meta Data

The Meta data are header data that most products in HIPE have. The longest set of Meta data is that attached to the *ObservationContext* that you begin your pipeline processing on. This includes information taken from the AOR and information on the configuration of PACS during your observation. From the *ObservationContext* you will extract your Level 0 product, and from there you will extract the *SlicedFrames* (spectroscopy) or *Frames* (photometry)—all of these layers have their own Meta data as well. Some of these Meta data are used by pipeline (and other) tasks. To see the most plentiful Meta data, you need to look at the *individual slices*, e.g. the *Frames* of the *SlicedFrames*. Note that if (as is the default now) the first slice is a calibration block, it will not have slice-relevant meta data in it; only the science slices do.

The Meta data of the layers in the *ObservationContext*, in the *SlicedFrames*, the *Frames*, and etc., will not be the same. Some are repeated, some are not. To see the Meta data of a particular product, you can click on it in the ObservationViewer, and the Meta data panel at the top will contain its Meta data:

**Figure 6.8. Meta data for a slice**

Clicking on the top-level product, i.e. that which you opened the Observation viewer on ("slicedFrames" in the figure above) then shows you its Meta data. You can also type Meta data out on the console. In the pipeline we add some Meta data keywords that are associated with the pipeline and the slicing (if the Meta data are not there you will simply get an error message),

```
# on the individual slices (i.e. Frames)
print myframe.meta["nodPosition"]
print myframe.meta["lineId"]
print myframe.meta["rasterId"]
print myframe.meta["lineDescription"]
print myframe.meta["band"]
# or also
print myframe.meta["bad"].string
# or, on slice 1 of the entire SlicedFrames (or Cubes)
print slicedFrames.getScience(0).meta["lineId"]
```

To inspect the Meta data of the *ObservationContext*, which is probably the first product you will look at, you can use the Observation viewer (*HOG* sec. 15.2). This viewer, accessible via a right-click on your *ObservationContext* in the Variables panel of HIPE, shows you the layers of the *ObservationContext* presented with a directory-like structure. With this you can see how the *ObservationContext* is organised, and you can also extract out, using drag-and-drop, some of the layers, or you can access the viewers that will work on these layers.

**Figure 6.9. Meta data of the ObservationContext**

Some of the entries can be understood without explanation, and those that are taken from the AOR (the description should say what they are) will have been explained in the HSPOT and AOT documentation (available from the HSC website). However, to help you further we here include a table listing the Meta data entries of an *ObservationContext*; this you will see when you click on the name of the *ObservationContext* in the Observation viewer (i.e. on "obs" in the figure above). Some Meta data are also explained in the *PDD*. Note that many of the Meta data are only interesting for the PACS project scientists.

| | |
|---|---|
| mapSquare | whether or not the map is square |
| missionConfig/uration | internal: mission configuration identifier |
| modelName | internal: Flight=PACS in space |
| n | number of raster lines (taken from HSPOT) (note: the default value is >0 even if the observation is not a raster) |

**Table 6.2. Meta Data of an ObservationContext**

| | |
|---|---|
| naidif | Solar system object NAIF identifier |
| object | target name |
| observer | P.I. name |
| obsid | observation identifier |
| obsMode | HSPOT entry: observing mode, e.g. pointed or mapping |
| obsOverhead | observing overheads |
| obsState | Level processed to |
| odnumber | operational day number |
| odStartTime | start of the operational day |
| orderSel | grating order |
| origin | who/what provided these data |
| PACS_PHOT_GAIN | gain settings for the bolometer |
| PACS_PHOT_MODE | Bolomter readout mode |
| pmDEC/RA | proper motion |
| pointingMode | pointing mode keyword |
| pointStep | raster step (point=column) in arcsec |
| posAngle | Position angle of pointing |
| processingMode | pipeline processing mode, SYSTEMATIC means "by the HSC" |
| proposal | name given to the programme when proposal was submitted |
| ra/raNominal | ignore/RA requested |
| raDeSys | RA, Dec coordinate reference frame |
| radialVelocity | spacecraft velocity along the line of sight to the target |
| raOff | HSPOT value: requested RA off position |
| redshiftType/Value | HSPOT value: the redshift given by the astronomer |
| refSelected | if true, then allow the use of raoff and decoff |
| repFactor | how many times the map was repeated |
| slewTime | time of start of slew to the target |
| source | HSPOT entry: type of observation, e.g. point(ed observation) or large (mapping observation) for spectroscopy |
| startDate/Time | start of observation |
| subType | internal |
| telescope | obviously: Herschel! |
| throw | chopper throw (spectroscopy) |
| type | product type identifier |
| userNODCycles | HSPOT value: number of nod cycles requested |
| velocityDefinition | which model was used to compute Herschel's radial velocity |
| widthUnit | HSPOT value: line width unit |

Next we list the Meta data of various layers of the *ObservationContext*: the level0, level0.5, level1, level2 layers; the *SlicedFrames* (HPSFIT[B|R] for spectroscopy or HPPAVG[B]|R for photometry), *SlicedPacsCube*s (HPS3D[B|R]), *SlicedPacsRebinnedCube*s (HPS3DR[B|R]), and the *ListContext*s holding photometry *SimpleImage* maps (HPPPMAP[B|R]) and projected *SpectralSimpleCube*s (HPS3DP[B|R]); and of the individual slices that are *Frames*, *SimpleImage*, *PacsCubes*, *PacsRebinnedCube* and the projected cubes (class *SpectralSimpleCube*). You will see these Meta data listed in the Observation viewer when you click on one of the names ("level0", "HPSFITB" etc.). These products should also have all of the Meta data of the *ObservationContext* also, so we list only the unique entries.

**Table 6.3. Meta Data of the other layers of a PACS observation**

| Name | Description |
|---|---|
| apid | internal |
| band | what filter/band was chosen |
| camera/camName/camera_signature | camera this product is for (red/blue spectrometer) |
| detRow/Col | number of detector rows and columns (but may be recorded incorrectly for the HPS3DP[B\|R] cube) |
| fileName | FITS file name associated with this product |
| herschelVelocityApplied | was Herschel's velocity corrected for (in the pipeline)? |
| isInterlaced | a mode of operation for the satellite's StarTracker |
| level | Level of this product (0,05,1,2) |
| nodPosition | nod position (A/B) that this product belongs to (mainly spectroscopy) |
| obsCount | internal |
| obsType | internal |
| productNotes | text describing the product |
| qflag_[R\|B]SSCGLIT_p/_p_v | description and percentage value of the number of glitched data-points found |
| qflag_[R\|B]SSCISAT_p/_p_v | description and percentage value of the number of saturated data-points found |
| qflag_[R\|B]SVALPIX_p/_p_v | internal |
| spec_[blue\|red]_FailedSPUBuffer | internal: percentage of failed SPU (signal processing unit) |
| testDescription/Name/Execution | internal |
| type | dataset type, e.g. HPSFITB for spectroscopy blue fit ramps (see the *PDD* for definitions) |
| verbose | internal |

Finally, there are some meta data that are added by the interactive spectroscopy pipeline, and these are described in the next table:

**Table 6.4. Meta Data added by the interactive spectroscopy pipeline**

| Name | Description |
|---|---|
| aotMode | aot mode, concatenation of other AOT mode Meta data |
| band | band name |
| calBlock | false or true for whether this is a calBlock |
| Chopper Throw | small, medium or large |
| Duration | of the observation in sec |
| isOffPosition | off or on position slice? |
| Line 1/2/... | details of this the line/band in the product |
| lineDescription | summary of other Meta data line descriptions |
| min/maxWave | minimum/maximum wavelength |
| nodCycleNum | nodding cycle number |
| nodPosition | nod B or A slice? |
| Number of lines | number of spectral lines |
| Number of Nod Cycles | exactly what it says |
| Observing mode | summary of all observing mode Meta data entries |
| onOffSource | on or off source: 0=not science, 1=on source, 2=off source |
| pacsSliceInfoUpdated | sliced information keywords were updated? |
| productNotes | dataset type, e.g. HPSFITB for spectroscopy blue fit ramps (see the *PDD* for definitions) |
| rasterId | number of raster lines, raster columns |
| Starting time | start of observation in a friendly format |

To query the meta data on the command line you can follow these examples:

```
# From an ObservationContext called obs
# - print all the meta data, so you know what are there
print obs.meta
# - print the meta data entry for "lines"
print obs.meta["lines"]
# giving:
#   {description="ObservationParameter",
#   string="[{O I 3P1-3P2", 63.18,4,0.0,0.0,1.0,"fluxWatt","kms"}]"}
# - print only the value of the "lineWidth" (which is held as a string)
print obs.meta["lineWidth"].string
# giving: [30.0]
# - what was the requested RA and Dec
print obs.meta["raNominal"]
# giving:
#   {description="Requested Right Ascencion of pointing",
#   double=63.2351234, unit=deg [0.01745 rad]}
# - print RA and Dec as numbers only
print obs.meta["raNominal"].double, obs.meta["decNominal"].double

# From a SlicedFrames product taken from level 1
# (e.g. slicedFrame=obs.level1.refs["HPSFITB"].product)
print slicedFrame.meta # and etc, as above

# From a Frames product, the first one in the SlicedFrames of level 1
# (e.g. frame=obs.level1.refs["HPSFITB"].product.refs[0].product)
print frame.meta # and etc, as above
```

# 6.3.2. Status table

The Status is a attached to your *Frames* and *PacsCube* products (not the *ListContext* they are contained within, but to the individual slices) and holds information about the instrument status—where the different parts of PACS were pointing and what it was doing—all listed in time order. The information held in the Status table are used by and added to by some pipeline tasks, and they can also be used to extract out particular parts of a product—although in HIPE track 8 it is no longer necessary to do such extractions using the Status table, in most cases this can be done more easily using the BlockTable or the pipeline task selectSlices. You, as an astronomy user, will never have to add to or change anything in the Status table and there is also almost nothing that you really need to check. However, since the Status table is there in your PACS products, we endeavour to explain it here, and then to show you how to select out parts of your spectra to plot by using Status parameters.

## 6.3.2.1. What is it?

The Status generally has the same dimensions as the readout/time-axis of the flux dataset, that is for each data-point, each Status column will have an entry. Hence, most (all?) of the Status entries are 1 dimensional. The Status is added to as the pipeline processing proceeds, and so what it contains will change as you proceed along the pipeline: e.g. when the chop- and chop+ are subtracted from each other in the spectroscopy pipeline task specDiffChop, then the readout dimension is reduced, and the Status is reduced appropriately.

To inspect the Status you can use the Dataset viewer or one of the plotters (access via the usual right-click on "Status" when you view your product via the ObservationViewer [right-click on the product in the Variables panel to access this viewer]). The entries in the Status table are of mixed type—integer, double, boolean, and string. The ones that are plain numbers can be viewed with the Table/OverPlotter, the others you need to look at with the Dataset Viewer. (Overplotting to see clearly the entries that have very different Y-ranges can be a bit difficult.)

Most Status entries are listed in the following table:

| | | |
|---|---|---|
| FINETIME | S+P Long 1 | Time in units of microsecond. Atomic time (SI seconds) elapsed since the TAI epoch of 1 Jan. 1958 UT2. |
| GPR | S Int 1 | grating position an encoded by the MEC |
| IsAPosition/IsBPosition | S+P Bool 1 | at B or A nod position> |
| IsConstantVelocity | S+P Bool 1 | is the satellite velocity constant? |
| IsOutOfField | S+P Bool 1 | out of field? |
| IsOffPos | S+P Bool 1 | off position flag |
| IsSerendipity | S+P Bool 1 | serendipity mode; was the instrument active during e.g. slew periods |
| IsSlew | S+P Bool 1 | slew of the satellite |
| LBL | S+P Int 1 | Label |
| Mode | S+P String 1 | pointing modes |
| NodCycleNum | S+P Long 1 | pointing nod cycle number |
| NrReadouts | Int 1 | number of readouts that were used for the on-board-reduced value |
| OBSID | S+P Long 1 | Observation identifier |
| OnTarget | S+P Bool 1 | on target flag |
| PaArray/Err | S+P Double 1 | Position angle of the boresight (centre of the detector), from pointing product and FINETIME |
| PIX | S+P Int 1 | - |
| RaArray/Err | S+P Double 1 | RA of the boresight (centre of the detector), from pointing product and FINETIME |
| RasterLine/ColunmNum | S+P Long 1 | raster line and column number |
| Repetition | S+P Int 1 | repetition number |
| RESETINDEX | S+P Int 1 | a counter of resets |
| RCX | S+P Int 1 | (detector) Raw Channel Index |
| RollArray | S+P | obsolete; see PaArray |
| RRR | S Int 1 | Readouts in Ramp (number) |
| SCANDIR | S Int 1 | grating scanning direction |
| ScanLineNumber | S+P Long 1 | scan line number |
| TMP1 | S+P Int 1 | Timing parameter: "time stamp" of the SET_TIME to the DMC, set only at the beginning of the observation. Unit of seconds |
| TMP2 | S+P Int 1 | see TMP1. In 1/65536 fractional seconds |
| Utc | S+P Long 1 | UTC (not used during pipeline processing but is set by pipeline task) |
| VLD | S+P Int 1 | Science data is value (0xff) or is not (0x00) |
| WASWITCH | S Bool | wavelength switching mode or not |
| WPR | S+P Int 1 | Wheel position encoded by the MEC |

**Table 6.5. Status information**

There are some Status parameters that are interesting to look at just to see what was going on while the observation was proceeding. You can also look at these Status parameters with one of the viewers that work on the Status (right-click on the Status, when you are looking at your *Frames* or etc. in the Observation Viewer, to access its viewers).

### 6.3.2.2. Selecting use the Status

A few ways to work with Status data, including selecting on your astronomical data:

```
# Spectroscopy

# What are the dimensions of a Status column
print myframe.getStatus("RaArray").dimensions

# Two ways to read Status data
# what are all the "BAND"s in your Frames
print UNIQ(myframe.status["BAND"].data)
print UNIQ(myframe.getStatus("BAND"))

# Extract the Status column called CHOPPOS from a Frames
# You can do the same for other columns
# Result will be a Double1d, Int1d, or String1d, in most cases
stat = myframe.getStatus("CHOPPOS")

# Extract out certain data using Status values
# get the on chops: here the throw is "large";
# "small" and "median" are also values
on = (stat == "+large") # a boolean, true where +large
on = on.where(on) # turn that into a Selection array
# get the off chops
off = (stat == "-large")
off = off.where(off)
wave = myframe.getwave(8,12)
offwave = wave[off]
# To check this, you could plot the selected data
p=PlotXY(myframe.getWave(8,12)[on],myframe.getSignal(8,12)[on])
p.addLayer(LayerXY(myframe.getWave(8,12)[off],myframe.getSignal(8,12)[off])
```

In fact there is much more you can do—HIPE is a scripting environment after all—but this is a data reduction, not a software guide. You can also learn more about scripting in HIPE in the *SaDM*. You can also look up the *Frames* and *PacsCubes* classes in the PACS *DRM* to see what methods that work with the Status within these products are available. And if you completely extract the Status from a *Frames* (>status=myframe.getStatus()), the PACS *DRM* entry for the *PacsStatus* class should be consulted to see what methods you can use there.

We show you more on selecting on the Status in order to plot the data, for example the grating and chopper movements plotted against the signal or the masks GRATMOVE and UNCLEANCHOP, in Sec. ???.

## 6.3.3. BlockTable and MasterBlockTable

The BlockTable is used more for spectroscopy than it is for photometry, although both types of data have this.

### 6.3.3.1. Spectroscopy

The BlockTable is a listing of the blocks of your *Frames* and *PacsCube*s. In the spectroscopy pipeline we slice our data according to a (mainly) astronomical logic (pointing, wavelength, etc.), therefore, as well as the BlockTables of the individual slices of the *SlicedFrames* or *SlicedPacsCube*s, there is a MasterBlockTable that is a concatenation of the individual BlockTables of all the slices.

Blocks are groups of data, following an instrument and/or astronomical logic: for example two grating scans will be two blocks, and two pointings in a raster will be two blocks. The data themselves are not

organised in this block structure, rather the BlockTable contains pointers to the data so that tasks can identify where the blocks are simply by querying with the BlockTable. It also gives a handy overview of your observation. Blocks are created in and used by the pipeline.

To find the BlockTable to look at it, open your *Frames* or *PacsCube* with the Observation Viewer, or look at them in the Outline panel (right-click to get a menu of viewers). Click on the Frames of PacsCube to see its contents and the BlockTable will be there...You will see the BlockTable in the directory-like listing. To see the MasterBlockTable for a *SlicedFrames* you do the same.

A screenshot of a MasterBlockTable viewed with the Dataset Inspector (the BlockTable will look very similar):

**Figure 6.10. The MasterBlockTable**

In the MasterBlockTable, the entry that identifies the slice number is "FramesNo". The entries that identify nods, grating scan (direction), raster pointing, wavelength range, etc. can be found in the table below. Many of the BlockTable columns come directly from the Status, and so can be similarly understood. If you hover with the cursor over the column titles, a banner explanation will pop up. The BlockTable entries are:

**Table 6.6. BlockTable columns**

| Name | Description |
|---|---|
| Band | spectral band |
| CalSource | 0,1,2 for neither, calsource 1, calsource 2 |
| ChopperPlateau | taken from the Status table |
| Description | explanation of keyword |
| DMSActive | DecMec sequence active? 0=no 1=yes |
| Filter | filter |
| FramesNo | slice number, the *Frames* number in the *SlicedFrames* that this block comes from |
| GenericOnOffSource | 0=not science, 1=on, 2=off, -1=not defined (i.e. the block has chopping, so both on and off are included) |
| GPRMin/Max | smallest and largest grating position |
| Id | keyword describing this block |
| index | a simple counter |
| IsOutOfField | true=out of field |
| LineDescription | line description taken from the Meta data |
| LineId | line identification number (a simple counter) |
| Min/MaxWave | smallest and largest wavelength |
| NodCycleNum | nod cycle number |
| NoddingPosition | 0=none, 1=A, 2=B |
| NrIdx | number of indices in this block |
| Obcp | on board control procedure number |
| OffSource1/2 | first and second off-source position label (some AOTs have 2 off positions) |
| OnSource | on-source position label |
| Raster | raster number |
| RasterColumn/LineNum | raster column and line numbers |
| Repetition | repetition cycle number (used in photometry) |
| ResLen | reset length (number of resets that fed into each readout) |
| ScanDir | scanning direction (0,1) |
| Start/EndFineTime | start and end FINETIME of this block |
| Start/EndIdx | starting/end index of this block |
| WaSwitch | wavelength switching active or not |

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 use the [Master]BlockTable to query your data you can use the following examples:

```
# get listing of all columns of the BlockTable of a Frames
# and the MasterBlockTable of a SlicedFrames
print myframe.blockTable
print slicedFrames.masterBlockTable
```

```
# extract the BlockTable from a Frames
block=myframe.getBlockTable()
# extract raster information from it
col=block.getRasterColumnNumber()
line=block.getRasterLineNumber()
# select col and line 1
sel=col.where((col == 1) & (line == l))
list=sel.toInt1d() # move to integer array
# now select those blocks
myframe_sub=myframe.select(myframe.getBlockSelection(list,None))

# Two ways to get a listing of a BlockTable column
# List of all the raster line number
line=block.getRasterLineNumber()
# list of the unique raster line numbers
line=UNIQ(myframe.getBlockTable()["RasterLineNumber"].data)

# by far the easiest way to select from a SlicedFrames
# or SlicedPacsCubes|RebinnedCubes
# is the pipeline helper task "selectSlices"
lineId      = []
wavelength  = []
rasterLine  = []
rasterCol   = []
nodPosition = ""
nodCycle    = []
band        = ""
scical      = ""
sliceNumber = []
sCubes = selectSlices(slicedCubes, lineId=lineId, wavelength=wavelength,\
         rasterLine=rasterLine,\
         rasterCol=rasterCol, nodPosition=nodPosition, nodCycle=nodCycle,\
         band=band, scical=scical,\
         sliceNumber=sliceNumber, verbose=verbose)

# Manual examples:
# Get the raster column numbers from a SlicedFrames
nodSlicesCol = slicedFrames.getMasterBlockTable().getRasterColumnNumber()
# select on it, e.g.:
select=nodSlicesCol.where(nodSlicesCol == 2)
select=select.toInt1d()
i = select[0] # and etc for the other elements of "select"
newSlicedFrames = slicedFrames.refs[i].product

# get the frame numbers, with a slightly different syntax to the above
framenum=slicedFrames.getMasterBlockTable()["FramesNo"].data
```

You can look up the *Frames* and *PacsCubes* classes in the PACS *DRM* to see what methods that work with the BlockTables are available. And if you completely extract the BlockTable from a *Frames* (>status=myframe.getBlockTable()), the PACS *DRM* entry for the *ObcpBlockTable* class should be consulted to see what methods you can use there.

# 6.4. PACS product viewers

Here we describe PACS viewers that can be used on both images and cubes. The PACS product viewer (PPV) is a general tool that both spectroscopy and photometry can use. The PACS spectral footprint viewer (SFV) is aimed more at spectroscopy but is useful if you have PACS images and cubes of the same source.

## 6.4.1. The PACS Spectral Footprint Viewer

This viewer will overplot the footprint of the spectrometer cube on an image, including on a PACS photometry image. It will open on an image or a cube, but it makes no sense to use it unless you have a cube. This task, pacsSpectralFootprint, will become an "Applicable Task" in the Task tab of HIPE when you click on a PACS cube or an image. It can be used to display the footprint of the IFU either on a blank background or on an existing image (with a WCS, e.g. one retrieved with the VO Tool

Aladin, or any Herschel image, most like a PACS image). To use it you need to have a cube (of which you are going to see the footprint) and an image (on which the footprint will be plotted).

- Open the task in the Tasks View. The GUI appears in the Editor area. If you have already selected a cube or an image, then this will open up in the viewer (you may need to zoom in with the buttons on the lower-left of the display area of the GUI)
- Optionally drag-and-drop an image or a cube with a WCS from the Variables panel into the display area of the footprint GUI. Drag it into the display area of the SFV (i.e. the PACS image or cube, or an image from elsewhere).
- If the product is a *ListContext* of cubes (i.e. the container that hold the cubes as you pipeline process your data), then the footprints of all the cubes will be drawn. You can select to view one or more footprints in the table which is below the display.
- Command line usage: You can run the task from the console as follows:

```
fp = pacsSpectralFootprint(some_image_with_wcs)
# Then the fp variable has some methods to add more footprints, or to
# access the Display inside for e.g. custom annotations:
fp.addFootprint(some_cube)
fp.display.zoomFactor = 2
```

  consult its *URM* entry to learn more, and the PACS *DRM* to find out about the methods.

**Figure 6.11. The PACS spectral footprint viewer**

The viewer should work on all PACS cubes.

## 6.4.2. PACS product viewer (PPV)

With the PACS product viewer, PPV (aka. the Mask Viewer), you an see the detector layout and the time-line dataset in all the detector pixels for the spectrometer and photometer (we explain the spectrometer here but it works the same way for the photometer). This GUI can be called up either with a right click on a *Frames* in the Variables panel (it does not work on maps or cubes), or on the command line:

```
MaskViewer(myframe)
```

In this way you can look at the data, in the form of signal vs readout (=time), for each detector pixel, one by one. It also allows you to plot flagged data and overplot Status values on the signal data-points. A screenshot of the PPV:

**Figure 6.12. The PACS Product Viewer**

At the top you see a layout of the detector with colour-coded values, from which you can select pixels to see their time-line spectrum in the plot at the bottom of the viewer. (For spectroscopy, the 25 columns are the 25 modules/spaxels, and in each are 18 pixels containing the spectral data, although pixel row 0 and 17 contain no astronomical data.) In the middle is a (time-line) scroll-bar. Above the plot panel is a mask selection box. If you select from the Masks list, you will see in the plot below the data-points that are so-masked turn red (the red dots may be small, so look carefully). If you select the "collapse mask" radio button, you will see data-points highlighted in red that are masked for anything. At the same time, in the image at the top of the GUI, the pixels for which the masked data are the current readout/time-point will also be outlined in red. The current readout is selected with the scroll-bar, and a yellow circle will follow the scroll-bar movements in the plot below. The "autoscale" button is to help with the colour gradient in the detector view. At the very bottom of the PPV you can select some Status parameters to overplot on the spectrum.

The PACS product viewer will not plot wavelength on the X-axis.

You can also edit the masks with the PPV: see Sec. ???.