PACS Data Reduction Guide: Photometry

Issue user. Version 15
March 2017

PACS Data Reduction Guide: Photometry

Table of Contents

List of Figures

Chapter 1. PACS Launch Pads

1.1. Introduction

*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 images, and will outline what you need to consider before you start to reduce the data yourself through the pipeline. A complete archive of PACS documentation, from data reduction and product advice, through calibration reports, to highly technical notes, can be found on the Herschel documentation archive: the HELL pages.*

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).

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*.

1.2. PACS Data Launch Pad

1.2.1. Terminology

The following Help documentation acronyms are used here (the names are links): **DAG**: the Data Analysis Guide; **SG**, the Scripting 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 SG to learn more about classes. Text written `like this` is used to refer to the parameter of a task.

1.2.2. Getting and saving PACS observations

Herschel data are stored in the **HSA**.

- They are identified with a unique number known as the Observation ID (**obsid**). You can find the obsid via the HSA.
- They can be downloaded directly into HIPE, or one at a time to disc, or many as a tarball.
- The data you get from the HSA is an **Observation Context**, which is a container for all the science data and all the auxiliary and calibration data that are associated with an observation, and includes

the **SPG** products. The entire observations is stored on disk as individual FITS files organised in a layered directory structure. The *ObservationContext* you load into HIPE contains links to all these files, and GUIs are provided to navigate through the layers.

There are several ways to **get and save observations from the HSA or disk** via HIPE. It does not matter which method you use.

- **Get the data directly from the HSA into HIPE on the command line, and then save to disk**:

  ```
  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
  # A: to save to the "MyHsa" directory (HOME/.hcss/MyHsa)
  myobs = getObservation(obsid, useHsa=True, save=True)
  # B: to save to your "local store" (usually HOME/.hcss/lstore)
  myobs = getObservation(obsid, useHsa=True)
  saveObservation(myobs)
  # C: to save to another disk location entirely, use:
  pooll = "/Volumes/BigDisk/"
  pooln = "NGC3333"
  myobs = getObservation(obsid, useHsa=True)
  saveObservation(myobs, poolLocation=pooll, poolName=pooln)
  ```

  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. For full parameters of getObservation, see its URM entry.

- **To get the data back from disk into HIPE:**

  A and B: If you saved the data to disk with the default name and location (either [HOME]/.hcss/MyHSA or [HOME]/.hcss/lstore) then you need only specify the obsid:

  ```
  obsid = 134...... # enter your obsid here
  myobs=getObservation(obsid)
  ```

  C: If you used saveObservation with a `poolName` and/or `poolLocation` specified:

  ```
  obsid = 134...... # enter your obsid here
  pooll = "/Volumes/BigDisk/"
  pooln = "NGC3333"
  myobs=getObservation(obsid, poolLocation=pooll, poolName=pooln)
  ```

To learn about the GUI methods for getting data, see chap. 1 of the DAG.

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 (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"
- 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**
- If there is a "level2_5" then you can also look at any of the maps in there, these in fact being a combination of level2 maps from related observations, and hence of better sensitivity.

To learn more about the layers of the *ObservationContext* and what the products therein are, see the PPE in *PACS Products Explained*.

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?

It is unlikely that you will need to reprocess you data: normally if there is something wrong with the data, (i) there will be a qualitySummary comment discussing that, and (ii) you will be able to see if directly in the images. The final data in the archive have the best possible calibration and for the majority of observations, the JScanam or Unimap maps at Level 2.5 or the maps at Level 2 (where there is no Level 2.5) cannot be bettered: possible exceptions are observations of a field of very low intrinsic SNR or with complex structure, and these may benefit from a personal data reduction.

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 and the science target,

- *Chopped point source data*: see Sec. 3.3 for observations taken in chop-nod photometry mode (an old mode).
- scan-map and mini scan-map for point sources*: see Sec. 3.2.1 for observations containing mainly point sources and small extended sources*
- Extended sources using MADMap*: see Chap. 3.2.2 for observations of extended sources (only use when scan and cross scan data are taken).*
- Extended source using JScanam *see Sec. 3.2.3 for observations of extended sources (only use when scan and cross scan data are taken).*
- *Extended source using Unimap* see Sec. 3.2.4.2 for observations of 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 1 ((calibrated data cubes in Jy/detector pixel)) to Level 2 (fully-processed). 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)

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.

1.3.3. 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*

- If you just want to look at the images you can use the **Standard Image Viewer**: see *Sec 4.4 of the DAG*:
- The **annularAperturePhotometry task**: (see *Sec 4.21 of the DAG*) 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 *Sec 4.19 of the DAG*) Extracts sources from a simple image using different algorithms.
- The **sourceFitter**: (see *Sec 4.20 of the DAG*) 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 *Herschel Owners Guide* chap. 15, and what you are looking at when you inspect a PACS *ObservationContext* is explained in the PPE in *PACS Products Explained*.

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 and Frames*, see the PPE in *PACS Products Explained*.

The following (Help) documentation acronyms are used here: *DAG*: the Data Analysis Guide; *PDRG*: this 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 later 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 (not ex-tarfile format, but 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=pooll,
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:

```
# 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 created. 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.

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. 3.3 for observations taken in chop-nod photometry mode (an old mode).
- *scan-map and mini scan-map for point sources*: see Sec. 3.2.1 for observations containing mainly point sources and small extended sources
- *Extended sources using MADMap*: see Chap. 3.2.2 for observations of extended sources (only use when scan and cross scan data are taken).
- *Extended source using JScanam* see Sec. 3.2.3 for observations of extended sources (only use when scan and cross scan data are taken).
- *Extended source using Unimap* see Sec. 3.2.4.2 for observations of extended sources.

The pipeline scripts can be found in the HIPE *Pipelines>PACS>Photometer* 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 (better known as the "SPG"—Standard Product Generation—data)? The answer to this depends on when the data you have were reduced by the PACS Standard Product Generation pipeline that is run by the Herschel Science Centre to populate the Herschel Science Archive, and on the type of observation you have. The data products contained in the Herschel Science Archive might be produced by a previous pipeline version, and therefore some of the algorithms and calibration files it used may be older than those in your version of HIPE(how to check is shown in Sec. 2.5.4). Note that you can always use the on-demand processing option provided by the Herschel Science Archive to run the latest version of the PACS Standard Product Generation pipeline. This option is especially interesting for those who do not have a machine with tens of Gigabytes of RAM that is needed to perform PACS data reduction. The pipeline is continually being updated. In any way it is always advisable to inspect your level2/level2.5 data to see whether the parameters with which the SPG pipeline was run are appropriate for your observations. To check which version of HIPE the SPG data were reduced with, type, in the Console of HIPE, the following: HIPE> print obs.getMeta()["creator"].string where "obs" is your ObservationContext; you can also look at the version of the calibration tree with: HIPE> print myobs.["calTreeVersion"].long. If you are reading this PDRG via HIPE then you will be working in Track 10|11 of HIPE. To figure out what calibration tree is the latest, simply load it and look:

```
calTree=getCalTre(obs-myobs)
print calTree
# version number is printed near top of listing
```

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.

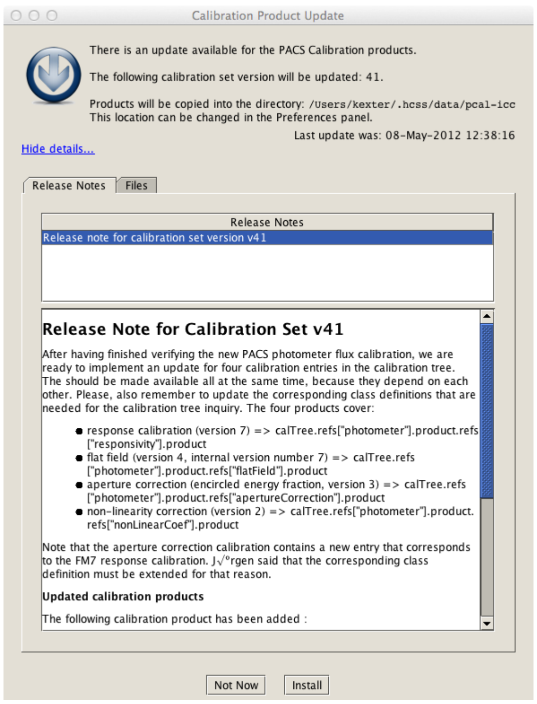
The calibration files are not provided with the HIPE build, rather you are offered to chance to update them only when they need to be updated. If you open HIPE and you get a pop-up telling you to install the calibration products, it means that the calibration file set has been updated by the PACS team 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 relevant information about the release, in the calibration updater pop-up click on "Show details...". In the new panel that appears, look at the "Release Notes" tab for a summary of the new set version. In there will be listed the calibration files (the FITS files) that have been included in the update and information about the changes made.

You can also look at the individual "Files" tab to see what (if anything) has changed in the individual files that are being updated. Some files will have no information in them, most of the information is in the Release Notes tab, and in the Files tab in the files called PCalBase_TimeDependency_FM_v#, which also contain a summary of the release. If more than one version number of calibration files are listed, you will be more interested in the highest version number.



There is an update available for the PACS Calibration products.

The following calibration set version will be updated: 41.

Products will be copied into the directory: `/Users/kexter/.hcss/data/pcal-icc`
This location can be changed in the Preferences panel.

Last update was: 08-May-2012 12:38:16

Release Notes
Release note for calibration set version v41

Release Note for Calibration Set v41

After having finished verifying the new PACS photometer flux calibration, we are ready to implement an update for four calibration entries in the calibration tree. The should be made available all at the same time, because they depend on each other. Please, also remember to update the corresponding class definitions that are needed for the calibration tree inquiry. The four products cover:

- response calibration (version 7) => calTree.refs["photometer"].product.refs["responsivity"].product
- flat field (version 4, internal version number 7) => calTree.refs["photometer"].product.refs["flatField"].product
- aperture correction (encircled energy fraction, version 3) => calTree.refs["photometer"].product.refs["apertureCorrection"].product
- non-linearity correction (version 2) => calTree.refs["photometer"].product.refs["nonLinearCoef"].product

Note that the aperture correction calibration contains a new entry that corresponds to the FM7 response calibration. J√°rgen said that the corresponding class definition must be extended for that reason.

Updated calibration products

The following calibration product has been added :

Figure 2.1. Updating the calibration files

To check on which pipeline tasks this will affect, check the pipeline scripts, as they comment on which calibration files are used by the tasks that use calibration files (e.g. you are told "# used cal files: observedResponse, calSourceFlux" for the task specDiffCs).

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 updater 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 SPG 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 = getObservation(obsid,file)
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. 4 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 was not a recommended observing mode and only few observation was taken using chop-nod technique at the beginning of the mission, we provide an ipipe script; see Sec. 3.3) and then by astronomical case (point source vs. extended source). You will also see "Standard pipeline" scripts, which are those that the automatic processing (SPG) use, and these differ from the interactive in being 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 thcan 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. They are not exchangeable, so use either tabs or spaces, not both. 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** described 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 interactive 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.

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

- High pass filtering and photProject: suitable for point (and slightly extended) sources.
- Generalized Least Square (GLS) mapmakers: we provided two mapmakers - Unimap and MadMad - that exploit the GLS method. Starting from SPG13, the maps from Unimap replace those from MadMap in the Herschel Science Archive.
- Destriper mapmaker: JScanam is the HIPE implementation of the IDL mapmaker called Scanmorphos.

Unimap (MadMap) and JScanam are well suited for point and extended sources: they give similar results so it is up to the user to decide which one to use. They exploit the redundancy provided by the observations (mainly scan and cross-scan) and they generate Level2.5 products. The high pass filter method is different, it is applied to a single observation (Level2) and then the individual maps (scan, cross-scan) are combined with the mosaic task to create Level2.5 products. 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 pipelines.

3.2.1. Point sources: high pass filtering and photProject

This script processes scan map and mini-scan map observations containing mostly point-like or relatively small extended sources. It starts from Level 1 (calibrated data cubes in Jy/detector pixel). The final map is in units of Jy/map pixel. This script uses the high-pass filter (HPF) method (see also Sec. 4.6 for a detailed description of the high-pass filtering method) to remove the 1/f noise. This processing is not adequate for extended sources as it removes large-scale structures, which cannot be properly protected (masked) from the filtering.

Note

As of HIPE 11.0 there is no need to start the reprocessing your data starting from the Level 0 (raw frames)

The script is able to combine several obsids but it is also useful for processing a single obsid. It performs iterative masked highpass filtering of the timelines and projects the final map using photProject: a first pass is done using a S/N filtering of the timeline, then a second pass is done using a full circular patch masking of the source(s). The placement of the circular patch is very important to ensure the correct positioning, we propose three options:

- Source fitting: if the source is the brightest object in the field source fitting will find the source and set the center of the patch to its fitted coordinates.
- Target coordinate list: the script reads the coordinates given in a text file and sets the coordinates to the center of the patch.
- Target source list and source fitting: if the coordinates are only approximate a source fitting is done on a sub-image centered on the given coordinates. The sub-image small size ensures that the fitting does not diverge seeking the brightest object in the image.

Beside the appropriate placement of the mask the correct filtering of the data also relies on the correct setup of the filter width. An optimum value is given as default but it can be modified as desired and/or necessary for specific science cases.

Note

if the HPF width (the half-width of the highpass filter) is too tight some extended features of the compact sources will be filtered out, on the other hand a wider HPF width will increase the left over 1/f noise.

An important parameter for the combination HPF and PhotProject is the ratio between the original pixel size, the re-gridding onto an output pixel (ouputpix) and the active pixel fraction (pixfrac). (see also Sec. 4.7 for a detailed description of the photProject task)

One last parameter is: a boolean to identify the source as being a solar system object or not. Several OBSIDS can be combined and as SSOs are moving across the sky between observations, a reference position and time should be set to be able to re-align the frames to the object.

In the following, we walk you through the process step by step.

3.2.1.1. Setting up for your Object

Fist we need to import some tasks to make our life easier:

```
import os
from java.util import ArrayList
from herschel.ia.toolbox.image import MosaicTask
from herschel.pacs.spg.phot import MaskFromCatalogueTask
from herschel.pacs.spg.pipeline.SaveProductToObservationContext import *
```

then we set up some important variables:

```
object = "yourObject"
obsids = [1342204327,1342204328]
camera = "blue"
sso = False
#direc = "/theDirectoryPath/here/"
direc = os.getcwd() + "/"
fileRoot = direc + object + "_" + camera
saveIntermediateSteps = False
calculateRaDec = False
doIIndLevelDeg = False
doPhotometry = True
doSourceFit = False
fromAFile = False
tfile = direc + "myTargets.txt"
```

1. object: Object name
2. obsids: list of obsids to combine (for reference: (scan + xscan))
3. camera: controls whether we want to process the data of the blue (70 and 100 micron) or the red (160 micron) side of the camera
4. sso: is True if the object is a "moving" target and the projection should be done onto the object and not according to sky position.camera: you may combine both both 70mic and 100mic "red" bolometers.
5. direc: it defines the directory where all the products are going to be saved. you can set is using the full path (recommended for the current User release) or just by using the os.getcwd which obtains the current working directory from which HIPE was started.
6. fileRoot: the root filename for the products to be saved

7. saveIntermediateSteps: allows to save intermediate maps and mask. This can be useful to check if the source(s) has/have been correctly masked.

8. calculateRaDec: this step allows to assign a RaDec to each frames. This allows the map projection to be run faster, however, this increase the memory load in HIPE by a factor of three.

Note

it is worth using it if one wants to make sevaral maps with slightly different parameters (e.g different pixel sizes or pixfrac values) to explore what is the optimal parameter settings for his/her dataset.

9. doIIndLevelDeg: if an additional second order deglitching is to be applied of not, (default = False) (see Sec. 4.3 for the detailed description of the second level deglitching task)

10. doPhotometry: if the photometry is to be performed on the given source

11. doSourceFit: if fitting of the source is required (usually if the source is not at the centre of the map.

12. fromAFile: if the source is not at the center of the map and/or too faint to be detected.

13. tfile: the name of the files containing the approximate position of the targets. It is only needed to be provided if fromAfile is set to True

3.2.1.2. Setting up your map output parameters

In this short section, you have to define parameters which influence on the image quality at the end of the processing:

```
#lowScanSpeed  = 15
#highScanSpeed = 22
limits = 10.

if camera == "blue":
  outpixsz = 1.2  # map pixel size (arcsec)
  pixfrac = 0.1   # bolometer pixel drop size
else:
  outpixsz = 2.4  # map pixel size (arcsec)
  pixfrac = 0.1   # bolometer pixel drop size
```

- • limits: either set as hard limits of as percentage of the scanning speed. A percentage is recommended for the Parallel mode.
  - outpixsz: the size of the output map pixels in arcsec
  - pixfrac: bolometer pixel drop size

Both the output pixel size and the drop pixel size influence strongly the noise in the resulting map. A smaller drop size or output pixel size will allow a better PSF sampling (see also Sec. 4.7 for a detailed description of the photProject task including the output pixel size and the drop size).

3.2.1.3. Building the list of observations and preparing for processing

First we need to build a list of the observations and the corresponding calibration trees (see also Sec. 2.5) and load them into an array.

```
obs = []
for i in range(len(obsids)):
  #obs.append(getObservation(obsids[i]))
  obs.append(getObservation(obsids[i], useHsa=True, instrument="PACS"))
calTree = []
```

```
for i in range(len(obsids)):
  calTree.append(getCalTree(obs=obs[i]))
```

The calTree contains all the calibration files needed for the data processing. The setting "obs=obs[i]" ensures that the correct calibration files are attached to each of 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.

If you observe a Solar System object you need some special corrections in the astrometry due to the movement of the objects during the observation. If you want ot combine more that one obsid the reference position is needed to be set to the first observation: the successive observations will be combined using the reference position and time. See also Sec. 4.12.1

```
if sso:
  for i in range(len(obsids)):
    if camera == "blue":
      frames = obs[i].level1.refs["HPPAVGB"].product
    else:
      frames = obs[i].level1.refs["HPPAVGR"].product
    pp = obs[i].auxiliary.pointing
    orbitEphem = obs[i].auxiliary.orbitEphemeris
    horizonsProduct = obs[i].auxiliary.horizons
    # Add again the pointing information to the status
    frames = photAddInstantPointing(frames, pp, calTree=calTree[i],
 orbitEphem=orbitEphem, horizonsProduct=horizonsProduct, copy=1)
    # The following task moves the SSO target to a fixed position in the map
    if i == 0:
      timeOffset = frames.refs[0].product.getStatus("FINETIME")[0]
    frames = correctRaDec4Sso(frames , timeOffset=timeOffset, orbitEphem=orbitEphem,
 horizonsProduct=horizonsProduct, linear=0)
    # Save the frames to the ObservationContext
    obs[i] = savePhotProductToObsContextL1(obs[i], "HPPT" , camera, frames)
  # Remove variables that are not needed anymore
  del frames, pp, orbitEphem, horizonsProduct, i, timeOffset
```

3.2.1.4. Processing

Now we can really start processing our data. The processing consists of three major steps.

- a first pass is done per observation masking the source(s) by signal-to-noise
- a second pass is done by building a better mask from the combined map of the first pass, again with a signal-to-noise thresholding
- a third pass is performed by masking the central source or a list of sources with circular patching

First processing pass: S/N mask

First we create a primary map for each individual obsids without highpass filtering then using this map we construct a signal-to-noise ratio mask. This mask is then used to create a first set of maps using the HPF with the new mask. As a start we define some arrays that we will need later for storing some parameters:

```
isBright = Bool1d(len(obsids), False)
iindDeg = Bool1d(len(obsids), doIIndLevelDeg)
hpfradius = Int1d(len(obsids), 0)
```

- isBright: if the source is too bright one might need a different filter width than for fainter sources, hence we need a variable to control this for each individual obsids.
- iindDeg: controls whether IInd level deglitching should be performed on the obsid

- hpfradius: it contains the values of the high pass filter widths for each individual obsid

then we cycling through the obsids and create the first map and the first mask based on the S/N of the map

First we extract the frames from the observation context, then remove the noise dataset because we won't need it anymore. We can recalculate the noise later (see Sec. 4.10 for a detailed description of the noise in PACS maps).

```
for i in range(len(obsids)):
  print "Start processing: ", obsids[i]
  if camera == "blue":
    frames = obs[i].level1.refs["HPPAVGB"].product.refs[0].product
  else:
    frames = obs[i].level1.refs["HPPAVGR"].product.refs[0].product
  # Remove the noise dataset
  frames.remove("Noise")
```

Then we check whether one or more of our obsids contain sources that are brighter than 10 Jy.

```
  try:
    if(obs[i].meta["fluxPntBlu"].value > 0.0) or (obs[i].meta["fluxPntRed"].value >
 0.0):
      if(obs[i].meta["fluxPntBlu"].value > 10000.0) or
 (obs[i].meta["fluxPntRed"].value > 10000.0):
        print "+++ IInd Level Deglitching will be set to true: for very bright
 sources MMTdeglitching tends"
        print "+++ to flag as glitches the signal in the center of the sources."
        isBright[i] = True
        iindDeg[i] = True
  except:
    print " Parallel mode "
    isBright[i] = False
```

and finally set the width of the highpass filter which depends on the brightness of the source, calculate the coordinates of each frames (if calcualteRADec was set to True) and save the frames variable as its current state so we can use it later.

```
  if camera == "blue":
    hpfradius[i] = 15
    if(isBright[i]): hpfradius[i] = 25
  else:
    hpfradius[i] = 25
    if(isBright[i]): hpfradius[i] = 40
  if calculateRaDec:
    frames = photAssignRaDec(frames)
  save(fileRoot + "_" + str(obsids[i]) + "_framesWithoutHPF.ser", "frames")
```

Then we can do our first pass with the highpass filter and construct our first source mask based on S/N.

```
  frames = highpassFilter(frames, hpfradius[i], interpolateMaskedValues=True)
  #frames = filterOnScanSpeed(frames, lowScanSpeed=lowScanSpeed,
 highScanSpeed=highScanSpeed)
  frames = filterOnScanSpeed(frames, limit=limits)
  map, mi = photProject(frames, pixfrac=pixfrac, outputPixelsize=outpixsz,
 calTree=calTree[i], __list__=True)
  d = Display(map, title="HighpassFilter task result on " + str(obsids[i]) +
 " (without source mask)")
  if saveIntermediateSteps :
    simpleFitsWriter(map, fileRoot + "_" + str(obsids[i]) +
 "_map_withoutMask_firstStep.fits")
  # Obtain an source mask from this initial map
  med = MEDIAN(map.coverage[map.coverage.where(map.coverage > 0.)])
  index = map.image.where((map.coverage > med) & (ABS(map.image) < 1e-2))
  signal_stdev = STDDEV(map.image[index])
  threshold = 3.0*signal_stdev
  print "Threshold for the source mask = ", threshold
  mask = map.copy()
  mask.image[mask.image.where(map.image > threshold)] = 1.0
  mask.image[mask.image.where(map.image < threshold)] = 0.0
  mask.image[mask.image.where(map.coverage < 0.5*med)] = 0.0
  d = Display(mask, title="Source mask for " + str(obsids[i]))
  if saveIntermediateSteps :
    simpleFitsWriter(mask, fileRoot + "_" + str(obsids[i]) + "_mask_firstStep.fits")
```

and perform a second pass using our mask to provide a better highpass filtering on the original frames saved before the first pass. At the end of this steps we can remove some variables that are not needed anymore to free some memory.

```
  restore(fileRoot + "_" + str(obsids[i]) + "_framesWithoutHPF.ser")
  # Introduce the mask in the frames and apply the highpass filtering
  frames, outMask = photReadMaskFromImage(frames, si=mask, maskname="HighpassMask",
 extendedMasking=True, calTree=calTree[i], __list__=True)
  frames = highpassFilter(frames, hpfradius[i], maskname="HighpassMask",
 interpolateMaskedValues=True)
  #frames = filterOnScanSpeed(frames, lowScanSpeed=lowScanSpeed,
 highScanSpeed=highScanSpeed)
  frames = filterOnScanSpeed(frames, limit=limits)
  map, mi = photProject(frames, pixfrac=pixfrac, outputPixelsize=outpixsz,
 calTree=calTree[i], __list__=True)
  d = Display(map, title="HighpassFilter task result on " + str(obsids[i]) + " (with
 a source mask)")
  simpleFitsWriter(map, fileRoot + "_" + str(obsids[i]) + "_map_firstStep.fits")

# Remove variables that are not needed anymore
del i, frames, map, mi, d, med, index, signal_stdev, threshold, mask, outMask
```

Second processing pass: tight S/N masking based on combined maps

After creating our initial mask, and highpass filtered data we now combine all of our maps to get a better S/N for the sources to be masked out:

```
if(len(obsids) > 1):
  images = ArrayList()
  for i in range(len(obsids)):
    #ima = SimpleImage()
    #importImage(image=ima, filename=fileRoot + "_" + str(obsids[i]) +
 "_map_firstStep.fits")
    ima = simpleFitsReader(file=fileRoot + "_" + str(obsids[i]) +
 "_map_firstStep.fits")
    images.add(ima)
  mosaic = MosaicTask()(images=images, oversample=0)
  del images, i, ima
else:
  mosaic = simpleFitsReader(file=fileRoot + "_" + str(obsids[0]) +
 "_map_firstStep.fits")
d = Display(mosaic, title="Mosaic map")
if saveIntermediateSteps :
  simpleFitsWriter(mosaic, fileRoot + "_mosaic_firstStep.fits")
```

Then we create a new sourcemask using the combined map:

```
med = MEDIAN(mosaic.coverage[mosaic.coverage.where(mosaic.coverage > 0.)])
index = mosaic.image.where((mosaic.coverage > med) & (ABS(mosaic.image) < 1e-2))
signal_stdev = STDDEV(mosaic.image[index])
threshold = 2.0*signal_stdev
print "Threshold for the source mask = ", threshold
mosaicMask = mosaic.copy()
mosaicMask.image[mosaicMask.image.where(mosaic.image > threshold)] = 1.0
mosaicMask.image[mosaicMask.image.where(mosaic.image < threshold)] = 0.0
mosaicMask.image[mosaicMask.image.where(mosaic.coverage < 0.5*med)] = 0.0
d = Display(mosaicMask, title="Source mask obtained using the mosaic map")
if saveIntermediateSteps :
  simpleFitsWriter(mosaicMask, fileRoot + "_mosaicMask_firstStep.fits")

# Remove variables that are not needed anymore
```

```
del mosaic, d, med, index, signal_stdev, threshold
```

The mask is then applied during the HPF process of the original restored frames. and the new frames are deglitched according to the chosen method.

```
for i in range(len(obsids)):
  print "Start processing: ", obsids[i]
  restore(fileRoot + "_" + str(obsids[i]) + "_framesWithoutHPF.ser")
  frames, outMask = photReadMaskFromImage(frames, si=mosaicMask,
 maskname="HighpassMask", extendedMasking=True, calTree=calTree[i], __list__=True)
  frames = highpassFilter(frames, hpfradius[i], maskname="HighpassMask",
 interpolateMaskedValues=True)
  # Apply 2nd level deglitching after highpass filtering
  if(iindDeg[i]):
    s = Sigclip(10, 30)
    s.behavior = Sigclip.CLIP
    s.outliers = Sigclip.BOTH_OUTLIERS
    s.mode = Sigclip.MEDIAN
    # mdt = MapDeglitchTask() # hipe 10
    # mdt(frames, algo=s, deglitchvector="timeordered", calTree=calTree[i]) # hipe
 10
    mapDeglitch(frames, algo=s, deglitchvector="timeordered", calTree=calTree[i])
    del s
  else:
    frames = photMMTDeglitching(frames, incr_fact=2, mmt_mode="multiply", scales=3,
 nsigma=5)
  #frames = filterOnScanSpeed(frames, lowScanSpeed=lowScanSpeed,
 highScanSpeed=highScanSpeed)
  frames = filterOnScanSpeed(frames, limit=limits)
  map, mi = photProject(frames, pixfrac=pixfrac, outputPixelsize=outpixsz,
 calTree=calTree[i], __list__=True)
  d = Display(map, title="HighpassFilter task result on " + str(obsids[i]) + " (with
 the mosaic source mask)")
  simpleFitsWriter(map, fileRoot + "_" + str(obsids[i]) + "_map_secondStep.fits")
  # Save the glitch mask into the non highpass filtered frames and save them again
 for the next step
  glitchmaskMeta = frames.meta["Glitchmask"]
  glitchmask = frames.getMask(glitchmaskMeta.value)
  restore(fileRoot + "_" + str(obsids[i]) + "_framesWithoutHPF.ser")
  frames.setMask(glitchmaskMeta.value, glitchmask)
  save(fileRoot + "_" + str(obsids[i]) + "_framesWithoutHPF.ser", "frames")

# Remove variables that are not needed anymore
del i, frames, outMask, map, mi, d, glitchmaskMeta,  glitchmask
```

Finally the frames are projected onto a map for each obsid. Each individual maps are then combined together into a mosaic.

```
if(len(obsids) > 1):
  images = ArrayList()
  for i in range(len(obsids)):
    #ima = SimpleImage()
    #importImage(image=ima, filename=fileRoot + "_" + str(obsids[i]) +
 "_map_secondStep.fits")
    ima = simpleFitsReader(file=fileRoot + "_" + str(obsids[i]) +
 "_map_secondStep.fits")
    images.add(ima)
  mosaic = MosaicTask()(images=images, oversample=0)
  del images, i, ima
else:
  mosaic = simpleFitsReader(file=fileRoot + "_" + str(obsids[0]) +
 "_map_secondStep.fits")
d = Display(mosaic, title="Mosaic map")
if saveIntermediateSteps :
  simpleFitsWriter(mosaic, fileRoot + "_mosaic_secondStep.fits")
```

The mosaic map is used to create a new multiple source mask based on S/N. This multiple source mask will be combined during the third and ultimate pass to a circular patch centered on the main source.

```
med = MEDIAN(mosaic.coverage[mosaic.coverage.where(mosaic.coverage > 0.)])
```

```
index = mosaic.image.where((mosaic.coverage > med) & (ABS(mosaic.image) < 1e-2))
signal_stdev = STDDEV(mosaic.image[index])
threshold = 2.0*signal_stdev
print "Threshold for the source mask = ", threshold
mosaicMask = mosaic.copy()
mosaicMask.image[mosaicMask.image.where(mosaic.image > threshold)] = 1.0
mosaicMask.image[mosaicMask.image.where(mosaic.image < threshold)] = 0.0
mosaicMask.image[mosaicMask.image.where(mosaic.coverage < 0.5*med)] = 0.0
d = Display(mosaicMask, title="Source mask obtained using the mosaic map")
if saveIntermediateSteps :
  simpleFitsWriter(mosaicMask, fileRoot + "_mosaicMask_secondStep.fits")

# Remove variables that are not needed anymore
del d, med, index, signal_stdev, threshold
```

Combining multiple source mask and source circular patch: final map

If a file with the coordinate of the expected source(s) on the maps are provided then we can create mask using this list and a radius of 20 arcsec. But first we need to be sure that our coordinates are accurate so we fit a gaussian on each source and calculate the exact Ra and Dec. If the source is too faint for the fitter we just used the supplied coordinate. Of course if we are sure that our supplied coordinates are perfect we can skip the fitting part by setting the doSourceFit variable to False at the beginning of the script. If there is no target list given the script will use the source coordinates given in HSPOT as a center for the circular patch.

```
if(doSourceFit):
  if(fromAFile):
    tlist, ralist, declist = readTargetList(tfile)
    # Loop over the targets, fit a gaussian to them, and save the fit coordinates
    rasource = Double1d(len(tlist))
    decsource = Double1d(len(tlist))
    for i in range(len(tlist)):
      pixcoor = mosaic.wcs.getPixelCoordinates(ralist[i], declist[i])
      cropsize = 20.
      r1 = int(pixcoor[0]-cropsize/2.)
      r2 = int(pixcoor[0]+cropsize/2.)
      c1 = int(pixcoor[1]-cropsize/2.)
      c2 = int(pixcoor[1]+cropsize/2.)
      cmap = crop(image=mosaic, row1=int(pixcoor[0]-cropsize/2.),    \
                                row2=int(pixcoor[0]+cropsize/2.),    \
                                column1=int(pixcoor[1]-cropsize/2.), \
                                column2=int(pixcoor[1]+cropsize/2.))
      try:
        sfit = mapSourceFitter(cmap)
        rasource[i] = Double(sfit["Parameters"].data[1])
        decsource[i] = Double(sfit["Parameters"].data[2])
        del sfit
      except:
        print "mapSourceFitter couldn't fit the source"
        print "The coordinates in the target file will be used"
        rasource[i] = ralist[i]
        decsource[i] = declist[i]
    #
    del ralist, declist, i, pixcoor, cropsize, r1, r2, c1, c2, cmap
  else:
    sfit = mapSourceFitter(mosaic)
    tlist = String1d(1, object)
    rasource = Double1d(1, sfit["Parameters"].data[1])
    decsource = Double1d(1, sfit["Parameters"].data[2])
    del sfit
else:
  if(fromAFile):
    tlist, ralist, declist = readTargetList(tfile)
    rasource = Double1d(ralist)
    decsource = Double1d(declist)
    del ralist, declist
  else:
    # Get source Ra and Dec from the metadata
    tlist = String1d(1, obs[0].meta["object"].value)
    rasource  = Double1d(1, obs[0].meta["ra"].value)
    decsource = Double1d(1, obs[0].meta["dec"].value)

for i in range(len(rasource)):
  print "Source = " + tlist[i] + ", coordinates = ", rasource[i], decsource[i]
```

Then both the multiple sources mask and a circular patch mask centered on the main source(s) are combined and used during a final HP filtering on restored original frames.

```
radius = 20.
for i in range(len(obsids)):
  if(isBright[i]): radius = 30.
combinedMask = mosaicMask.copy()
#combinedMask.image = combinedMask.image*0. # uncomment if source list mask only
mfc = MaskFromCatalogueTask()
combinedMask = mfc(combinedMask, rasource, decsource, Double1d(len(rasource),
 radius), copy=1)
d = Display(combinedMask, title="Combined mask")
if saveIntermediateSteps :
  simpleFitsWriter(combinedMask, fileRoot + "_finalMosaicMask.fits")

# Remove variables that are not needed anymore
del mosaic, i, radius, mfc, mosaicMask, d
```

The final step is the actual map making from Level 1: the frames are:

- • Highpass filtered using the final combined mask

```
  print "Start processing: ", obsids[i]
  restore(fileRoot + "_" + str(obsids[i]) + "_framesWithoutHPF.ser")
  frames, outMask = photReadMaskFromImage(frames, si=combinedMask,
 maskname="HighpassMask", extendedMasking=True, calTree=calTree[i],
 __list__=True)
  frames = highpassFilter(frames, hpfradius[i], maskname="HighpassMask",
 interpolateMaskedValues=True)
```

- The turnovers are removed from the frames

```
frames = filterOnScanSpeed(frames, limit=limits)
```

- The frames are finally projected onto the final individual maps taking into account the chosen pixfrac and output pixel size. An extra set of map is created for comparison with a pixfrac = 1

```
  map, mi = photProject(frames, pixfrac=1, outputPixelsize=outpixsz,
 calTree=calTree[i], __list__=True)
  d = Display(map, title="Final map of " + str(obsids[i]) + " (pixfrac = 1)")
  simpleFitsWriter(map, fileRoot + "_" + str(obsids[i]) +
 "_finalMap_defaultPixfrac.fits")
  # Make a map with the user defined pixfrac
  map, mi = photProject(frames, pixfrac=pixfrac, outputPixelsize=outpixsz,
 calTree=calTree[i], __list__=True)
  d = Display(map, title="Final map of " + str(obsids[i]) + " (pixfrac = " +
 str(pixfrac) + ")")
  simpleFitsWriter(map, fileRoot + "_" + str(obsids[i]) + "_finalMap.fits")
```

- The final mosaics are combined for each given pixfrac.

```
if(len(obsids) > 1):
  images = ArrayList()
  for i in range(len(obsids)):
    #ima = SimpleImage()
    #importImage(image=ima, filename=fileRoot + "_" + str(obsids[i]) +
 "_finalMap_defaultPixfrac.fits")
    ima = simpleFitsReader(file=fileRoot + "_" + str(obsids[i]) +
 "_finalMap_defaultPixfrac.fits")
    images.add(ima)
  mosaicDefPixfrac = MosaicTask()(images=images, oversample=0)
  mosaicDefPixfrac.meta = images[0].meta
  mosaicDefPixfrac.history = images[0].history
  del images, i, ima
else:
  mosaicDefPixfrac = simpleFitsReader(file=fileRoot + "_" + str(obsids[0]) +
 "_finalMap_defaultPixfrac.fits")
d = Display(mosaicDefPixfrac, title="Final mosaic map (pixfrac = 1)")
simpleFitsWriter(mosaicDefPixfrac, fileRoot +
 "_finalMosaic_defaultPixfrac.fits")

if(len(obsids) > 1):
  images = ArrayList()
  for i in range(len(obsids)):
    #ima = SimpleImage()
    #importImage(image=ima, filename=fileRoot + "_" + str(obsids[i]) +
 "_finalMap.fits")
    ima = simpleFitsReader(file=fileRoot + "_" + str(obsids[i]) +
 "_finalMap.fits")
    images.add(ima)
  mosaic = MosaicTask()(images=images, oversample=0)
  mosaic.meta = images[0].meta
  mosaic.history = images[0].history
  del images, i, ima
else:
  mosaic = simpleFitsReader(file=fileRoot + "_" + str(obsids[0]) +
 "_finalMap.fits")
d = Display(mosaic, title="Final mosaic map (pixfrac = " + str(pixfrac) + ")")
simpleFitsWriter(mosaic, fileRoot + "_finalMosaic.fits")
```

Each intermediate steps are saved on disk

3.2.1.5. Photometry

If chosen, aperture photometry is performed on the source and several aperture on the sky around the source to determine the uncertainties and S/N of the source.

Results are given for the given pixfrac and each the apertures: centered on source and on the sky are displayed. The aperture on sky are chosen such that they overlap as little as possible with the source. The sky aperture are displayed: blue all, red selected for aperture photometry of the sky background.

The photometry is given: un-corrected and corrected for the used aperture. The on-source apertures can be modified according to the source size. The parameters controlling the apertures and the sky annulus are

```
  ap_blue  = [12,15,20]
  ap_green = [12,15,20]
  ap_red   = [8,15,20]
```

Her the first number in each array is the aperture radius the second and third numbers are the inner and outer radii of the sky annulus respectively.

By definition, HPF removes the sky contribution, i.e. the skybackground values of the mosaics should be distributed around a zero value. However, this is in theory, and the sky aperture photometry gives a good indication of the residual noise. The sky photometry is also corrected for the apertures used.

Each intermediate mosaics, masks and the original frames are saved to disk during the processing. If the user chose to not save the intermediate steps, then all but the final mosaic are removed from disk.

3.2.2. Extended sources: 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.

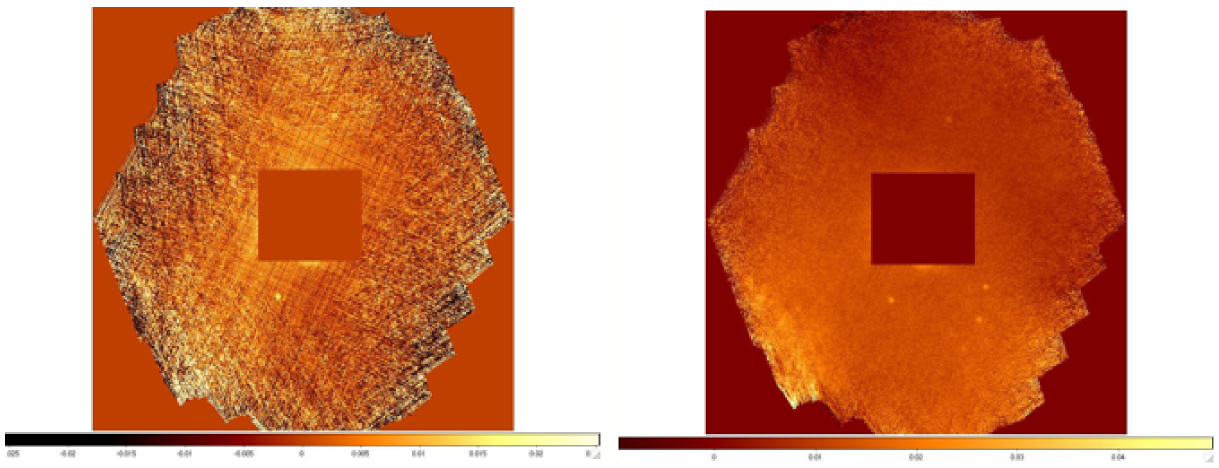


Figure 3.1. The function of MADmap is to remove the effect of 1/f noise. Left: image created without MADmap processing. Right: The same image after MADmap processing. The central object has been masked out.

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.

3.2.2.1. 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 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 three main 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.

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. In order to estimate the zero level, MADMap uses 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.setStatus("OnTarget", Bool1d(dim[2], 1))
frames = photOffsetCorr(frames)
```

Figure 3.2 shows the image that is the median of the signals in each pixel, which is what is done when you use photOffsetCorr.

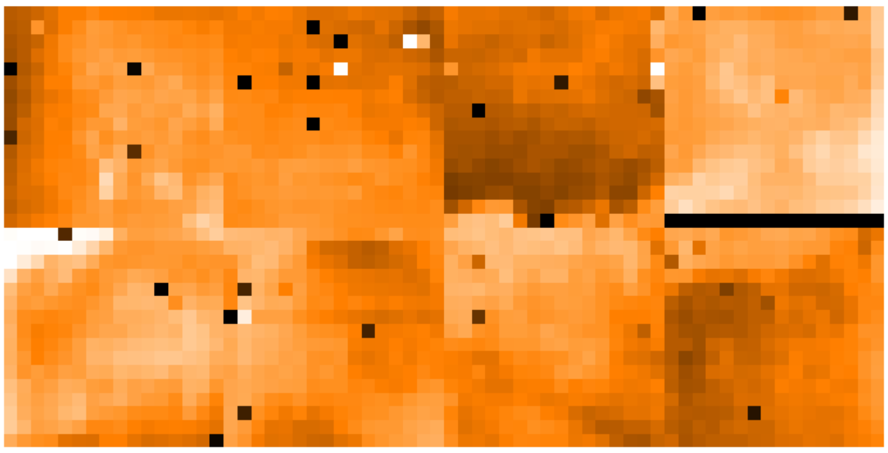


Figure 3.2. The map of the median of the signals in each pixel showing the pixel to pixel electronic offset.

Figure 3.3 shows a single readout slice (one single time point) of an input frame after the pixel-to-pixel offset correction using photOffsetCorr.

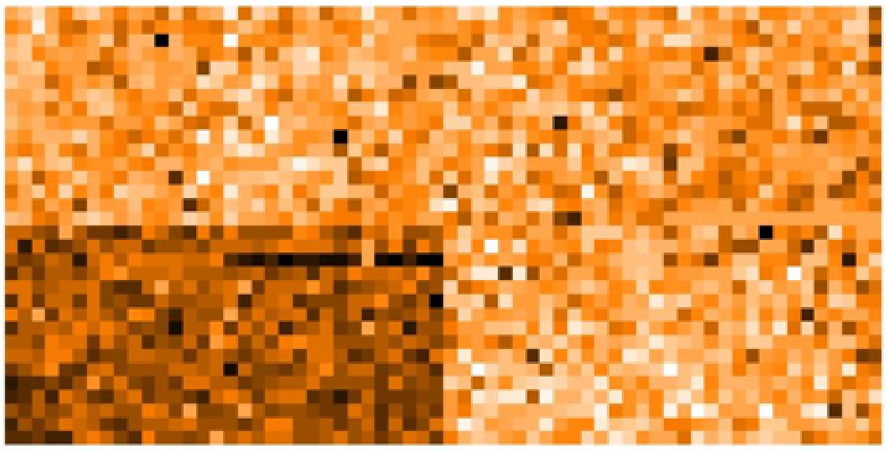


Figure 3.3. A single slice (one single time-point) of a raw signal Frame after the offset (pixel to pixel) image removal step (i.e. after subtracting Figure 3.2 from the original Frames). While the pixel to pixel variation is mitigated, the result shows two modules are systematically at a different signal 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))
```

Exponential signal drift removal

The procedure to remove this fast transient consists in the following steps: * * * * .

- guess drift
- remove drift and make an approximate map

- backproject and subtract this map from the timelines. What remains is drift and 1/f noise
- fit individual pixels with an exponential model

For the first-guess of the drift, the user has two options:

1. divide the median array in bins of N readouts (N typically of the order of 1000 readouts), take the minimum value in each N readout bins, and fit the resulting curve with a polynomial . This is the best option in the presence of bright sources and corresponds to setting the "useMinMedFirstGuess" keyword at the beginning of the script to "True". In this case, guess the drift from the minimum value of the median array of the binned timelines:

   ```
   x, driftGuess = photGlobalDrift(frames, binsize=binSize, doPlot=doPlot,\
                        saveToFile=saveImagesToDisk, closePlot=closePlot)
   ```

   To this, fit an exponential + polynomial function:

   ```
   driftFit = photGlobalDrift_fitExpPoly(x, driftGuess, showFit=doPlot,\
                   saveToFile=saveImagesToDisk, verboe=verbose,\
                   title=obsid + "_" + camera)
   ```

2. alternatively, fit the median values directly with a polynomial (set "useMinMedFirstGuess" to "False" (default). This solution works well for faint and normal sources, which do not produce a significant signal in a single image. In this case, guess the drift from the raw mean of the unbinned timelines:

   ```
   driftGuess = MadMapHelper.calculateMeanTimeline(frames)
   ```

   To this, fit an exponential + polynomial function:

   ```
   driftModel = photGlobalDrift_fitExpPoly(x, driftGuess, showFit=doPlot,\
                     saveToFile=saveImagesToDisk, title=obsid + "_" + camera,\
                     returnModel=True)
   ```

   Now remove the sources from the drift by removing the positive outliers. This is an iterative procedure, in which the sources are identified with respect to the previous estimate of the drift (i.e. driftModel):

   ```
   maxDiff = 10000
        unmaskedIndices = Selection(Int1d.range(dim[2]))
        for iter in range(globalExpFitIterations):
           #
           diff = driftGuess[unmaskedIndices] - driftModel(x[unmaskedIndices])
           orderedDiffIndex = SORT.BY_INDEX(diff)
           fivePerCent = int(0.05*len(diff))
           maxDiff = min(maxDiff, diff[orderedDiffIndex[-fivePerCent]])
           diff = driftGuess - driftModel(x)
           unmaskedIndices = diff.where(diff < maxDiff)
   ```

   At this stage, the processing is the same, independently from how the drift was initially estimated. The next step consists in removing the estimated average drift from the frames signal:

   ```
   frames = MadMapHelper.subtractDrift(frames, driftFit)
   ```

   Make a map of drift-subtracted frames:

   ```
   tod = makeTodArray(frames, calTree=calTree, scale=1)
   naivemap = runMadMap(tod, calTree, maxRelError, maxIterations, 1)
   ```

   Then backproject this map and subtract it from the original frames. This operation allows the subtraction of the sky signal from the frames. What is left is the remaining drift and uncorrelated 1/f noise (i.e. "clean timelines"):

   ```
   smi = tod.getMapIndex()
   ```

```
frames.setSignal(signalCopy)
frames.signalRef.subtract(image2SkyCube(naivemap, smi)["skycube"].data)
```

At this stage, perform a pixel-to-pixel fit to the "clean timelines" with an exponential + polynomial function:

```
result = fitScansExpPoly(frames)
```

Remove the exponential + Polynomial pixel-to-pixel drift from the original frames:

```
frames.setSignal(signalCopy)
frames.signalRef.subtract(result["perPixelFittedCube"].data)
```

All the steps described above are applied to each scan (i.e. obsid). So, before moving forward, the drift-corrected scan/X-scan frames are merged:

```
if firstObs:
     mergedFrames = frames
     firstObs = False
  else:
     mergedFrames.join(frames)
```

Iterative drift correction

The remaining drift is mitigated by iteratively fitting a baseline (i.e. a polynomial of order 1) to each scan. This part of the processing starts by making a map with the fast-transient corrected merged frames:

```
tod = makeTodArray(mergedFrames, calTree=calTree, scale=pixScale)
smi = tod.getMapIndex()
sourceMap = runMadMap(tod, calTree, maxRelError, maxIterations, 1)
```

Then, the iterative loop begins. At the start of each iteration, the current estimate of the map is subtracted from the timeline:

```
mergedFrames.signalRef.subtract(image2SkyCube(sourceMap, smi)["skycube"].data)
```

The remaining drift is fit per-scan and per-pixel:

```
result = photFitScanLines(mergedFrames, mode="perPixel")
```

And then it is subtracted from the original merged frames:

```
mergedFrames.signalRef.subtract(result["fittedCube"].data)
```

A new map is created:

```
makeTodArray.rewriteTod(tod, mergedFrames.signalRef)
sourceMap = runMadMap(tod, calTree, maxRelError, maxIterations, 1)
```

The procedure above is repeated for N iterations, where N is by default set to 5.

3.2.2.2. Optimising MADmap pre-processing

In this new version of the MADmap script, the processing requires very little user interaction. The only parameter that can/should be tuned - as discussed above - is "useMinMedFirstGuess". This should be set to "True" only for very bright sources. Otherwise it should be left to the default value ("False").

- perPixelExpPolyFit: fit an exponential model to the initial fit. Recommended value: True
- resetScanMedLevels: legacy from prototype code. It should be removed in the next release of the code. Recommended value: False.

- deglitch: deglitch again the timelines. If set to "True", it will ignopre the deglitching applied up to Level 1 processing and perform the deglitching again using the JScanam algorithm. Recommended value: True
- nSigmaDeglitch: definition of a glitch. Recommended value: 5
- globalExpFitIteraQons: number of iterations used to decouple signal from exponential drift. Recommended value: 5
- nIterations: number of iterations to use for the iterative drift removal. Recommended value: 5
- minDuration: this is the minimum size of the timeline on which MADmap can work. Recommended value: 150
- binSize: binSize to use with useMinMedFirstGuess. Recommended value: 1003

3.2.2.3. Running MADmap

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

Preparation

After pre#processing, in theory the only drift left in the signal should be due to the 1/f noise. Figure 3.4 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.

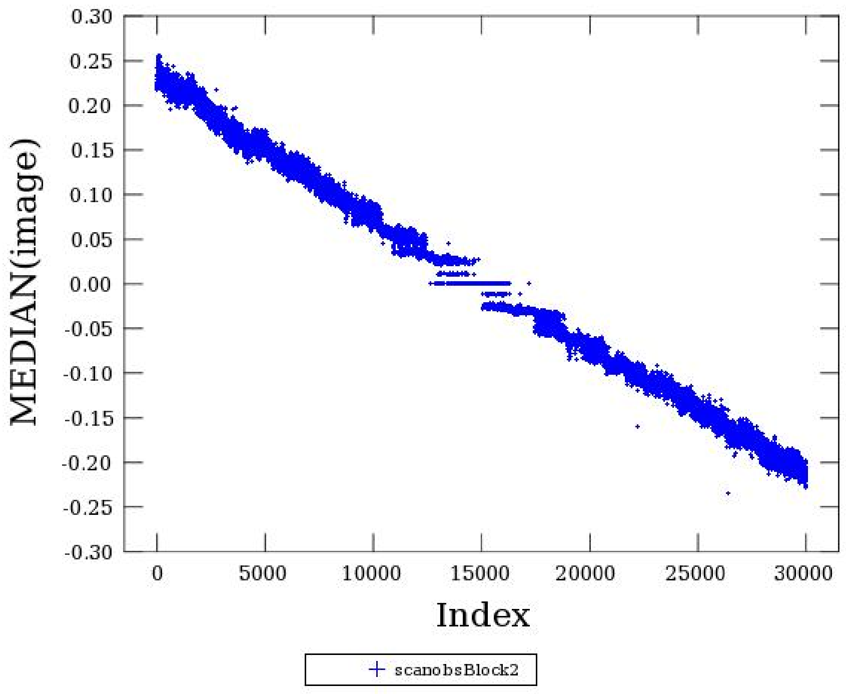


Figure 3.4. The power spectrum of the full data stream after the drift removals (averaged for all pixels). Some structure is expected due to the astrophysical sources and from the unremoved glitches (not the final figure, just a placeholder until I get the real one).

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.

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)
```

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].

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.

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).

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.

3.2.2.4. MADmap post-processing

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].

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 computationally 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 3.5, where the impact of the correlated noise can be seen in the form of stripes following the scan lines.

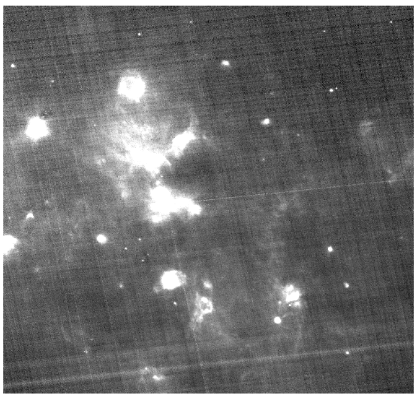


Figure 3.5. Impact of the correlated noise in the form of stripes following the scan lines.

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 3.6. 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.

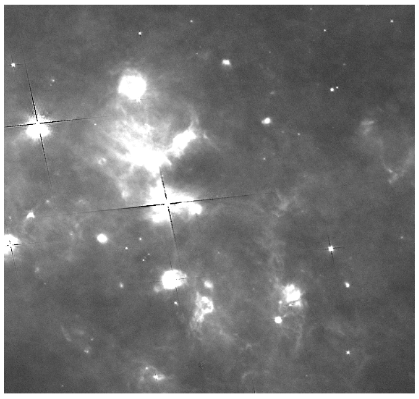


Figure 3.6. Point source artifact in a form of crosses places on bright point sources.

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.

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.

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 fFigure 3.7 and Figure 3.8. 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.

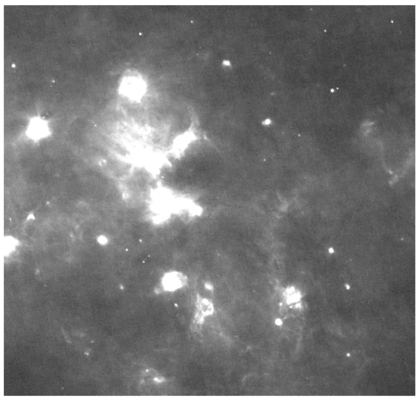


Figure 3.7. Post-processed image with the artifacts removed.

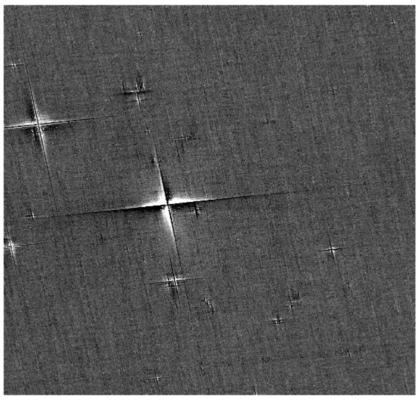


Figure 3.8. The point source artifacts that were removed.

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.

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.

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

3.2.2.5. Open issues and known limitations

The following items are known limitations of MADmap processing:

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.

3.2.2.6. 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.

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 3.9 illustrates this.

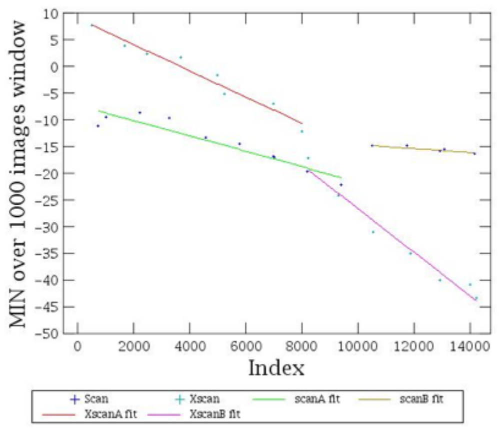


Figure 3.9. The minimum median ???) plotted versus readout index. There appears to be a change in the magnitude of the drift, likely caused by a cosmic ray or charged particle impact on the readout electronics. You can see this by the break in the lines that fit the data: the scan direction data are 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 3.10 shows the smoking gun that indicates a glitch caused the global drift to change.

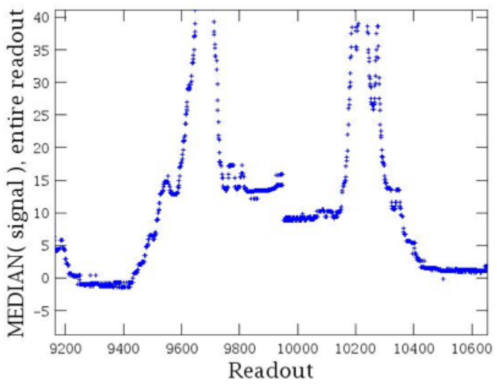


Figure 3.10. An expanded region of time-ordered data, near where the drift shows an abrupt change in magnitude in Figure 3.9. There is a clear break in the signal near readout value 9900

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.

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 3.11.

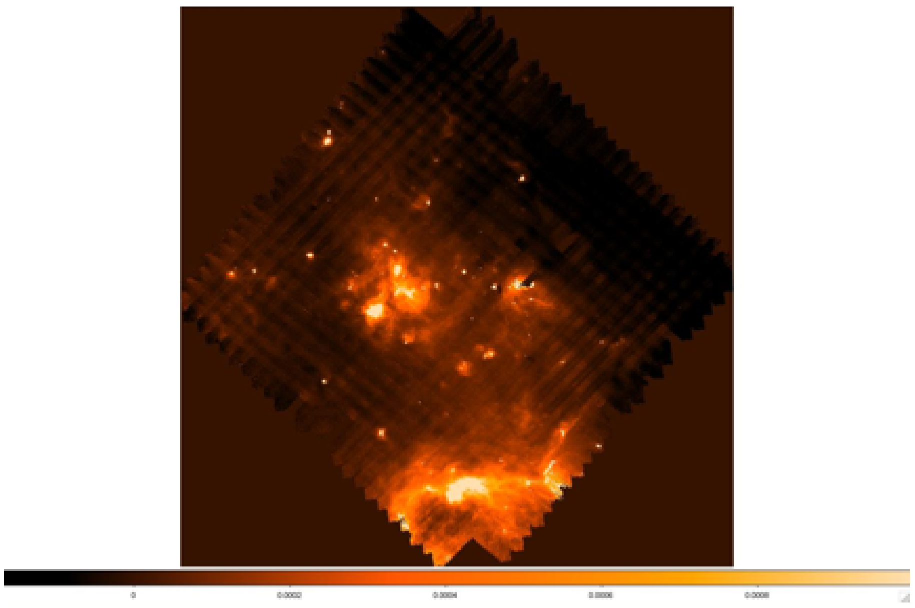


Figure 3.11. The final mosaic with a clearly visible "checkered" noise pattern super imposed on the sky. This artifact is due to improper correction for the module-to-module drift

Point source artifact

Figure 3.12 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.

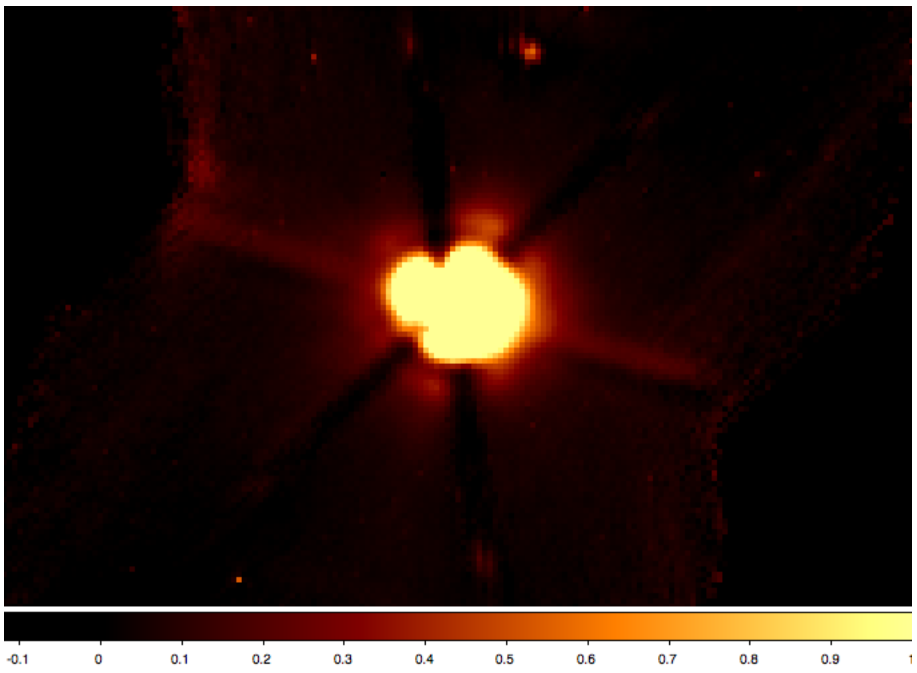


Figure 3.12. An example of the point source artifact around a very bright source. The MADmap reduction creates regions of negative (dark) stripes in the scan and cross-scan direction centred on the point source.

3.2.3. Extended sources: JScanam

This script (scanmap_Extended_emission_JScanam.py) is the HIPE implementation of the IDL map-maker called Scanmorphos. Please look at the following link for more information and a detailed description of this map making tool.

The script starts from level1 frames and creates a map from the data. It always combines two (and only two) obsids. If you have more than one pair use the scanmap_Extended_emission_JScanam_multiplePairs.py script which is basically the same script embedded into a for cycle and can combine any number of scan and cross-scan obsid pairs. The script is designed in a way that you can run it line by line or in one go. It is also possible to check all the intermediate products created during the processing.

First you need set some variables to control the behaviour of the script.

```
camera = "red" (or "blue")
obsids = [obsid-scan1,obsid cross-scan1] or [obsid-scan1,obsid cross-scan1,obsid-
scan2,obsid cross-scan2,...] in case of multiple pairs
solarSystemObject = True/False
galactic = True/False
calculateRaDec = True/False
showMapsAfterTasks = True/False
debug = False/True
deglitch = True/False
nSigmaDeglitch = 3
makeFinalMap = True/False
outputPixelSize = the size of your pixel in arcseconds
pixfrac =
```

- camera: sets which side of the PACS photometer you want to process. Be careful with memory allocation, blue requires about 4 times the memory used in red.
- obsids: sets the obsids for scan and cross-scan pairs
- solarSystemObject: set it to True if the source is a Solar System object (i.e. is a moving target).

Note

if there is extended emission in the map, the main JScanam assumptions will fail, because the scan and cross scan will be shifted to the object reference system.

- galactic: the galactic option should be set to True if the maps extended emission is not confined in a small area.
- calculateRaDec: Calculate the coordinates of every pixel with time. It is usually safe to set it to True.
- showMapAfterTasks: if True, produces a map after each step in the pipeline.
- deglitch: set it to True if you want to run the JScanam deglitch task
- nSigmaDeglitch: the new deglitch threshold to use in case the deglitch parameter is set to True. The default is 3 times over the stDev value of the signal in a given map pixel.
- debug: Debug mode. Use wisely, if set to True it will clutter your desktop with graphs and displays. It is not recommended unless you are really an expert user of Scanamorphos.
- makeFinalMap: generate a final map.
- outputPixelSize: Pixel size for the final map in arcseconds. If its value is -1 value, the pixel size will be the default one (3.4" in blue and 6.2 in red).
- pixfrac: this parameter is used for the final map. It fixes the ratio between the input detector pixel size and the map pixel size.

After setting your initial parameters you can start downloading the data. The best way to do it to directly download it from the Herschel Science Archive (HSA). For alternative methods see Chapter 1 of *DAG*.

```
scansObs = getObservation(obsids[0], useHsa=True, instrument="PACS")
level1 = PacsContext(scansObs.level1)
scans = level1.averaged.getCamera(camera).product.selectAll()
blueFilter1 = scansObs.meta["blue"].value

cscansObs = getObservation(obsids[1], useHsa=True, instrument="PACS")
level1 = PacsContext(cscansObs.level1)
cscans = level1.averaged.getCamera(camera).product.selectAll()
blueFilter2 = cscansObs.meta["blue"].value
```

this part of the scripts downloads your data from the HSA and extracts the level1 frames for the two given obsids. If you set the solarSystemObject parameter to True then it shifts the coordinates to the object reference system

```
if(solarSystemObject):
  print " Setting the scans and crossScans coordinates to the object reference
 system "
  pp = scansObs.auxiliary.pointing
  orbitEphem = scansObs.auxiliary.orbitEphemeris
  horizons = scansObs.auxiliary.horizonsIf you set
  cal = getCalTree(obs=scansObs)
  scans = photAddInstantPointing(scans, pp, calTree=cal, orbitEphem=orbitEphem,
 horizonsProduct=horizons)
  timeOffset = scans.getStatus("FINETIME")[0]
  scans = correctRaDec4Sso(scans, timeOffset=timeOffset, orbitEphem=orbitEphem,
 horizonsProduct=horizons, linear=0)
  #
  pp = cscansObs.auxiliary.pointing
  orbitEphem = cscansObs.auxiliary.orbitEphemeris
  horizons = cscansObs.auxiliary.horizons
  cal = getCalTree(obs=cscansObs)
  cscans = photAddInstantPointing(cscans, pp, calTree=cal, orbitEphem=orbitEphem,
 horizonsProduct=horizons)
```

```
  cscans = correctRaDec4Sso(cscans, timeOffset=timeOffset, orbitEphem=orbitEphem,
 horizonsProduct=horizons, linear=0)
  del pp, orbitEphem, horizons, cal, timeOffset
```

Then, if calculateRaDec was set to True, it calculates the Ra and Dec for each pixels at each frame

```
   scans = photAssignRaDec(scans)
   cscans = photAssignRaDec(cscans)
```

and removes the unnecessary frames that are taken during the turnaround of the spacecraft

```
scans = scanamorphosRemoveTurnarounds(scans, limit=5.0, debug=debug)
cscans = scanamorphosRemoveTurnarounds(cscans, limit=5.0, debug=debug)
```

Here you can control the amount of removed data through the limit parameter. It is expected in percent and gives a zone around the nominal scanspeed (in our case +- 5%) Task removes every frames that have a scan speed higher or lower than these limits.

After removing the turnaround frames we mask long the term glitches using scanamorphosMask-LongTermGlitches. This task produces a mask called Scanam_LongTermGlitchMask. You should check this mask and if the results are not optimal (some glitches are not detected or some sources are flagged as glitches), you can try to modify the parameter stepAfter in order to get better results.

```
scans = scanamorphosMaskLongTermGlitches(scans, stepAfter=20, galactic=galactic)
cscans = scanamorphosMaskLongTermGlitches(cscans, stepAfter=20, galactic=galactic)
```

This task produces a mask called Scanam_LongTermGlitchMask. You should check this mask and if the results are not optimal (some glitches are not detected or some sources are flagged as glitches), you can try to modify the parameter stepAfter in order to get better results. This parameters controls the length of the event tat is considered as glitch.

At this point we save the scan and cross-scan for later use in a temporal pool

```
from herschel.pacs.share.util import PacsProductSinkWrapper
  scansRef = PacsProductSinkWrapper.getInstance().saveAlways(scans)
  cscansRef = PacsProductSinkWrapper.getInstance().saveAlways(cscans)
```

Then we subtract the baseline of every scanleg in every pixel

```
scans = scanamorphosScanlegBaselineFitPerPixel(scans, nSigma=2, debug=debug)
cscans = scanamorphosScanlegBaselineFitPerPixel(cscans, nSigma=2, debug=debug)
```

scanamorphosScanlegBaselineFitPerPixel is a task that subtracts a linear baseline to the signal of every pixel and every scanleg in the frames. The objective of this task is to remove signal drifts with time scales larger than a scanleg, preserving real sources as much as possible, with the intention of creating a first order map that can be used for the source mask calculation. Extended emission is generally not preserved by this task, so use it carefully.

These are the main algorithm steps:

- Select the unmasked timeline data for each instrument pixel and every scanleg. If a source mask is provided via the sourceMask optional parameter, the points in this mask will not be used.
- If there is enough points, fit them with a linear baseline. If this is not the case, calculate their median value.
- The linear baseline fit is performed on a iterative way. The unmasked data is fitted first. Fit positive outliers are then removed, and the fit is performed again with the clean data. The same steps are repeated until the iteration converges. This happens when either there is no new outliers found, the number of iterations is larger than 50, or there is too few data points to continue with the fitting process. In this last case, the median of the unmasked data is calculated.

- Subtract the fit (or the median) to the whole scanleg signal, including the masked values.

The most important parameter of the task is nSigma that controls the threshold limit for the source removal. Points above nSigma times the standard deviation of the linear fit residuals will be classified as real sources, and will not be considered in the next fit iteration.

The next step is then to create the source mask. But before we do that we need to join the scan and cross-scan to increase the signal to noise for detecting the sources.

```
scans.join(cscans)
sourceImage, scans = scanamorphosCreateSourceMask(scans, nSigma=4.0,
 createMask=False, galactic=galactic, calTree=calTree, debug=debug)
```

Here nSigma controls also the threshold above which the emission should be masked out. Modify nSigma until you get an optimal mask. The mask should cover only a small fraction of the map (<~30%). It's not necessary that all the faint emission is masked, only the brightest regions.

Then we replace the scan and cross-scan with the saved ones, and to save some memory we delete the saved data.

```
del(scans)
scans = scansRef.product
cscans = cscansRef.product
del(scansRef, cscansRef)
System.gc()
```

Then we add the mask to the scans:

```
maskImage, scans = scanamorphosCreateSourceMask(scans, inputImage=sourceImage,
 createMask=True, calTree=calTree, debug=debug)
maskImage, cscans = scanamorphosCreateSourceMask(cscans, inputImage=sourceImage,
 createMask=True, calTree=calTree, debug=debug)
```

After adding the source mask to the data we can start the real processing by removing the general offset using a simple baseline removal.

```
scans = scanamorphosBaselineSubtraction(scans, galactic=True, debug=debug)
cscans = scanamorphosBaselineSubtraction(cscans, galactic=True, debug=debug)
```

Here we set the galactic option to through because e just want to remove and offset. The task calculates the signal median value in every pixel array and every scan (for galactic = True) or scanleg (galactic = False), and subtracts it from the frames. Only the unmasked data is considered in the median offset calculation.

Before we move further with the processing we need to identify and mask the signal drifts produced by the calibration block observation. The PACS photometer signal is sometimes affected after the calibration block observation by a strong signal drift with a typical time length of 1000 time indices. The task scanamorphosBaselinePreprocessing identifies those drifts and masks them. The derived mask is saved in the frames with the name Scanamorphos_CalibrationBlockDrift. This mask will be active after the execution of the task. If a mask with the same name is already present in the frames, it will be overwritten with the new mask values.

```
scans = scanamorphosBaselinePreprocessing(scans, galactic=galactic, debug=debug)
cscans = scanamorphosBaselinePreprocessing(cscans, galactic=galactic, debug=debug)
```

After we got rid of the effect of the calibration blocks we can remove the real baseline from the scans (not only a simple offset as we did before) by running the basline subtraction again:

```
scans = scanamorphosBaselineSubtraction(scans, galactic=galactic, debug=debug)
cscans = scanamorphosBaselineSubtraction(cscans, galactic=galactic, debug=debug)
```

Here, of course, we use the galactic option that is suitable to our data.

After the final baseline removal we can remove signal drifts with time scales longer than a scan-leg. This is called de-striping. We use the task scanamorphosDestriping.

```
scans, cscans = scanamorphosDestriping(scans, cscans, iterations=6, calTree=calTree,
 debug=debug)
```

This task removes signal drifts in the scans and the cross scans with time scales longer than a scan-leg. It assumes that the scans and the cross scans have perpendicular scan directions, and as a result the projected maps have different drift gradients. The drift removal technique is based on the following points:

- The drift power increases with the length of the considered time scale (1/f noise). For that reason the task starts removing the strongest drift component on a time scale equal to the scans length, decreasing this scale in the next step to 4 scalegs, and finally to 1 scanleg. In each step the remaining drifts are weaker.
- Back projecting a map obtained in a given scan direction, to the perpendicular direction, will transform the generally increasing or decreasing signal drifts into oscillatory drifts that cancel out on large time scales.

These are the main algorithm steps:

- Create a map of the cross scans and back project it to the scan direction of the scans. By default, the brightest sources will be removed using the mask created by scanamorphosCreateSourceMask (Scanamorphos_SourceMask). This can be changed with the task optional parameter sourceMask.
- Subtract the back projected timeline from the scans timeline. This will remove the remaining sources (common in the two timelines) a will leave the combination of the two drifts.
- Fit a line to the timeline difference. The fit will contain mostly the effect of the scans drift, because the contribution from the cross scans drift cancels due to its oscillatory nature.
- Subtract the fit (i.e. drift) to the scans timeline.
- Repeat the previous steps on the cross scans, using the back projection of the scans. Repeat all steps, decreasing the considered time scale from the scans length, to 4 scanlegs, and finally 1 scanleg.
- Iterate until convergence for timescales equal to one scanleg. Convergence is reached when the fit of the differences is considerably smaller than the fit residuals. The maximum number of iterations can be controled with the task optional parameter iterations.

At this point we can finish working with the scans and cross-scans separately and can join all the data together

```
mergedScans = scans
mergedScans.join(cscans)
del(scans, cscans)
System.gc()
```

and remove signal drifts with time scales shorter than a scan-leg using scanamorphosIndividualDrifts.

```
mergedScans = scanamorphosIndividualDrifts(mergedScans, debug=debug)
```

This is what the task do:

- Measure the scanspeed and calculate the size of a mappixel, that can hold 6 subsequent samples of one crossing of a detector pixel.
- Make a mapindex with the specified gridsize. Collect only pixfrac = 0.001 (only pixelcenters)
- Count the crossings of all pixels into every map pixel

- Calculate the average flux value for each crossing and its standard deviation: bi and si
- Now use the threshold noise. This is the blue line in Figure 3.1 called HF noise. Check, how many percent of the stddev of every mappixel are smaller than this threshold noise. -> < 80%; do not use that mappixel for the drift correction -> >= 80%: use the mappixel but don't use the values with stddev > HF noise
- Calculate the average flux fi in every mappixels. Use the white noise of the detectors as a weight.
- for each crossing calculate: average time ta, drift of the time ti as a difference of the average flux of the crossing and overall average di = bi - fi.
- build the drift timeline from the di at the times ti.
- interpolate the missing values of the real timeline (linear interpolation) and subtract the drift from the real timeline.
- new map for pointing corrections
- calc the rms (stddev) of the drift timeline sdi. If sdi < HFnoise, this iteration is converged if sdi > HFnoise, build the mapindex from the new corrected timelines and iterate.
- empirical improvement: average the drifts to time-bins of 27, 9, 3 and 1 (in subsequent iterations) and build the drift timeline by interpolating these much larger bins.

Then we deglitch the merged scans

```
scanamorphosDeglitch(mergedScans, nSigma=5, calTree=calTree, debug=debug)
```

and remove the individual drifts

```
mergedScans = scanamorphosIndividualDrifts(mergedScans, calTree=calTree,
 debug=debug)
```

If the deglitch parameter is set to True here comes a new deglitching with the user defined nSigma.

Finally we can create the final map using photProject using the desired pixel size and pixfrac:

```
finalMap = photProject(mergedScans, outputPixelsize=outputPixelSize,
 pixfrac=pixfrac)
```

of course we can save our map as a fits file:

```
outputMapFile = /your/Home/Dir/finalMap_"+camera+".fits"
simpleFitsWriter(finalMap, outputMapFile)
```

If we process multiple obsid pairs the end of the processing is a little bit different. First we need to save all of our pairs after the final deglitching using a product sink (we also examine here if the observation are done in parallel mode):

```
   if(camera == "blue"):
      if(blueFilter1 == "blue1"):

 blueScansReferences.append(PacsProductSinkWrapper.getInstance().saveAlways(mergedScans))
         if PhotHelper.isParallelObs(mergedScans):
            blueParallel = True
      else:

 greenScansReferences.append(PacsProductSinkWrapper.getInstance().saveAlways(mergedScans))
         if PhotHelper.isParallelObs(mergedScans):
            greenParallel = True
   else:
```

```
 redScansReferences.append(PacsProductSinkWrapper.getInstance().saveAlways(mergedScans))
      if PhotHelper.isParallelObs(mergedScans):
         redParallel = True

   del(mergedScans)
   System.gc()
```

then we merge all the observations by filter:

```
print " Merging all the pairs of scans and cross scans "
mergedScansBlue = None
mergedScansGreen = None
mergedScansRed = None

if(camera == "blue"):
   if len(blueScansReferences) > 0:
      mergedScansBlue = blueScansReferences[0].product
      for i in range(1, len(blueScansReferences)):
         mergedScansBlue.join(blueScansReferences[i].product)
   if len(greenScansReferences) > 0:
      mergedScansGreen = greenScansReferences[0].product
      for i in range(1, len(greenScansReferences)):
         mergedScansGreen.join(greenScansReferences[i].product)
else:
   mergedScansRed = redScansReferences[0].product
   for i in range(1, len(redScansReferences)):
      mergedScansRed.join(redScansReferences[i].product)

del(blueScansReferences, greenScansReferences, redScansReferences)
System.gc()
```

first we define three empty variables (one for each filter) for holding the merged scans then fill them up with the saved products. Finally we project the map using all of our obsid pairs:

```
if(outputPixelSize < 0):
   if(camera == "blue"):
      if(mergedScansBlue != None):
         if blueParallel:
            outputPixelSizeBlue = 3.2
         else:
            outputPixelSizeBlue = 1.6
      if(mergedScansGreen != None):
         if greenParallel:
            outputPixelSizeGreen = 3.2
         else:
            outputPixelSizeGreen = 1.6
   else:
      outputPixelSizeRed = 3.2
else:
   outputPixelSizeBlue = outputPixelSize
   outputPixelSizeGreen = outputPixelSize
   outputPixelSizeRed = outputPixelSize

if(makeFinalMap):
   print " Projecting the merged scans onto the final map "
   print " Projection with drizzle using: " + str(outputPixelSize) + " arcsec"
   if(mergedScansBlue != None):
      finalMapBlue, mi = photProject(mergedScansBlue,
 outputPixelsize=outputPixelSizeBlue, pixfrac=pixfrac, calTree=calTree,
 useMasterMaskForWcs=False)
      d = Display(finalMapBlue, title="Final map (blue)")
      # outputMapFile = "/your/Home/Dir/finalMap_"+camera+"_blue.jpg"
      # d.saveAsJPG(outputMapFile)
   if(mergedScansGreen != None):
      finalMapGreen, mi = photProject(mergedScansGreen,
 outputPixelsize=outputPixelSizeGreen, pixfrac=pixfrac, calTree=calTree,
 useMasterMaskForWcs=False)
      d = Display(finalMapGreen, title="Final map (green)")
      # outputMapFile = "/your/Home/Dir/finalMap_"+camera+"_green.jpg"
      # d.saveAsJPG(outputMapFile)
   if(mergedScansRed != None):
      finalMapRed, mi = photProject(mergedScansRed,
 outputPixelsize=outputPixelSizeRed, pixfrac=pixfrac, calTree=calTree,
 useMasterMaskForWcs=False)
      d = Display(finalMapRed, title="Final map (red)")
      # outputMapFile = "/your/Home/Dir/finalMap_"+camera+"_red.jpg"
      # d.saveAsJPG(outputMapFile)
```

3.2.4. Extended sources: Unimap

Unimap[1, 3] is a Generalized Least Squares (GLS) mapmaker developed in MATLAB and released by the DIET department of the University of Rome 'La Sapienza' (link). See the the Unimap Documetation web page for more information, and in particular for a detailed description of this mapmaker, read the Unimap User's Manual.

Unimap performs an accurate pre-processing to clean the dataset of systematic effects (offset, drift, glitches), and it uses the GLS algorithm to remove the correlated 1/f noise while preserving the sky signal over large spatial scales.

The GLS estimator provided by Unimap can introduce distortions at the positions of bright sky emission, especially if point-like. The distortions, which generally appear as cross-like artifacts, are due to the approximations of the signal model and to an imperfect compensation of disturbances at the pre-processing stage.

In the first Unimap releases (up to version 6.3) the distortions were estimated and removed by means of a post-processing (PGLS, WGLS). However, starting from the version 6.3 a new method was developed: this takes into account an additional disturbance within the GLS algorithm, that is the pixel noise.

In the post-processing, a high-pass filtering approach is implemented within Unimap to remove these distortions within a specified spatial scale, by generating the Post-GLS map (PGLS). The application of the high-pass filter method has the drawback that it injects back correlated noise over the same spatial scale used for the removal of the distortions. To minimise this noise injection, the filtering can be applied only over the bright emission regions - these are selected using a threshold approach, having at the latest stage the Weighted-GLS map (WGLS).

The distortions that the post-processing removes by means of an empirical approach are mainly due to the pixel noise, that is the noise that is introduced by digitally sampling (with pixels) a continuum signal (sky emission). The variation in the readouts, due to the random displacement of the sampling points with respect to the centre of the pixel (where readouts are assumed in the ‘hard-pointing’ approximation [5]), introduces an extra source of noise, especially on the sky regions where the signal variation are steep, i.e. where the displacements are statistically larger.

Compensating for the pixel noise within the GLS algorithm is the most convincing way of preventing distortions, rather than removing them by means of the post-processing. However, accounting for the pixel noise makes the GLS heavier from a computational point of view, since in each iteration of the GLS an internal iterative algorithm is run, and this can increase the processing time up to a factor 40. For this reason, for large datasets it may be necessary to run the GLS *without* the pixel noise compensation and apply the post-processing to remove the distortions instead.

The values of parameters that contribute to the generation of the GLS, PGLS and WGLS maps depend on the characteristics of the sky emission and on the level of accuracy to be achieved by the different estimators. The best choice for the parameters values is automatically set by Unimap (from the track 6 and beyond) by performing a statistical analysis of the sky emission. These parameters are described below and they can always be fine-tuned by the users according their purposes.

Finally, Unimap versions 6.3 or later can combine the pixel noise and the electronic noise to produce an estimate of the noise affecting the final map and to generate an error map associated directly with the final map.

3.2.4.1. Unimap installation and configuration

Before running the interactive script, install UNIMAP by downloading the TAR file from the web page (link). Unimap is developed in MATLAB and it is spawned from the interactive HIPE script by using the runUnimap task. If you do not have MATLAB on your computer, install the MATLAB Compiler Runtime (MCR) by following the instructions provided on the UNIMAP download web page.

You need to set the following path variables within the script:

```
unimapDir = '/your/Unimap/directory'
mcrDir = '/your/MATLAB/Compiler/Runtime/directory'
dataDir = '/your/data/directory'
```

- `unimapDir`: sets the path where the UNIMAP files (run_unimap.sh, unimap/unimap.app, unimap_par.tx) are, as provided by the TAR file.
- `mcrDir`: sets the path where the MCR bin folder is located.
- `dataDir`: sets the root path for storage on your computer. The Unimap output files are saved in a subdirectory (dataDir/firstObsid_lastObsid/camera) if the cleanDataDir parameter is set equal False (see below).

3.2.4.2. The Unimap interactive script

The script starts from Level1 frames and invokes the runUnimap task to generate the final maps.

First you set some variables to control the behaviour of the script (the path variables described above).

Unimap makes use of a large number of parameters that are defined and managed within the unimap_par.txt file. In the interactive script a limited number of parameters (9) are included and propagated. The default values adopted in the script allow to the pipeline to calculate their optimal values, but these parameters can be always fine-tuned by the user. The other Unimap parameters are taken from the unimap_par.txt file. In Section 3.2.4.4 we explain how to modify all of the Unimap parameters.

In the following, the parameters that the user can edit within the ipipe script are described. These are grouped into 3 sections: "Unimap parameters" includes the parameters that contribute to the generation of the GLS map, while the parameters for the post-processing and for an advanced usage of the script are gathered together into "Unimap post-processing" and "Advanced-use parameters" (the latter also in Section 3.2.4.4).

```
camera = "red" (or "blue")
obsids = [firstObsid, secondObsid, …., lastObsid]
solarSystemObject = True (or False)

#Unimap parameters
numProc = number of logical processor to use (default 1)
sync = compensate the time shifting along the scan leg (default -1)
pixelNoise = apply the correction for the Pixel Noise (default -1)
outputPixelSize = the size of your pixel in arcseconds (default 0)
minimumChange = the minimum change (in dB) to keep iterating (default 1)
startImage = starting image for the GLS (default 3)

# Unimap post-processing
filterSizeInArcsec = PGLS high pass filter size in arcsec (default 0)
wglsDThreshold = WGLS threshold to detect an artifact (default 0)
wglsGThreshold = WGLS threshold to grow an artifact (default 0)

# Advanced-use parameters
maskToUse = the mask in the frames that are adopted by Unimap
rewriteFramesFitsFiles = True (or False)
rewriteParametersFile = True (or False)
cleanDataDir = True (or False)
```

- `camera`: sets which channel of the PACS photometer you want to process. Note that the blue requires about 4 times the memory than the red.

- `obsids`: sets the obsids to combine. The script is designed for combining any number of obsids, being limited only by the computer memory and computational time. Remember that standard Level 2.5 products are the combination of 2 concatenated obsids - scan and cross-scan. If a large number of obsids are to be combined, it is recommended to use the memory_optimized script available in the Pipelines menu.

- `solarSystemObject`: is your source a solar system object? In such a case the reduction will be performed in the coordinate system of the moving object.

- `numProc`: how many logical processors do you want to use? Unimap 6.3 or later is designed to parallelise the GLS computation (the most time-consuming module) over a number of processors. If a negative number is given, Unimap detects the number of available processors. Remember that for managing a parallelised run, an overhead in term of RAM is required with respect to the usage of a single processor.

- `sync`: in Unimap 6.3 or later, the component of the Relative Pointing Error due to the time shift along the scan leg can be compensated. If sync=2, the synchronisation is applied, while it is skipped if sync=0. Since the synchronisation is estimated by comparing the source positions observed along different scan directions, Unimap evaluates whether the image is signal-rich enought to apply a reliable sync compensation, if a negative number is given.

- `pixelNoise`: the gain to apply to the estimated pixel noise into the GLS pixel noise compensation (available in Unimap 6.3 or later). 0 means no pixel noise compensation, 0.5 means half of the estimated pixel noise is compensated, 1 all of it, and so on. If a negative number is given, Unimap decides if the field is suited for pixel-noise compensation, according the dimension of the dataset and the amount of signal.

- `outputPixelSize`: pixel size for the final map in arcsec. If 0, the pixel size is 3.2", except for non-parallel blue observations, where it is 1.6"

- `minimumChange`: expressed in dB, it is the minimum change in the GLS map to keep iterating. If it is set to a positive number, the stop level is automatically selected by taking into account the morphology of the map. The rational is that a higher precision is requested for images which are background dominated and for intermediate cases with bright and extended emission over a flat background (e.g. nearby galaxies, nebulae).

- `startImage`: is the initial guess adopted by the GLS for starting the iterations. It can be: 0 for a zero (flat) image, 1 for a naïve map, 2 for a mixture map and 4 for a Alternating Least Squares (ALS) map (for Unimap 6.4.2 or later). The mixture map is composed of portions of a flat map at the positions recognised as background regions, while for the signal regions, the mixture map is formed by portions of the naïve map. Background and signal regions are identified by adopting a threshold approach over the computation of the noise level of the observation. If the ALS map is selected, the same method used for the drift removal is used to obtain a destriper starting image, by fitting the timelines with a piecewise function. The number of iterations required by the GLS to converge depends strongly on the adopted starting image, thus the selection of proper initial guess can affect the quality of the final maps and it can significantly reduce the process running time. Unimap performs a statistical analysis on the Naïve map for identifying the optimal starting image according the signal properties of the field. This starting guess is applied if the startImage parameter is 3.

- `filterSizeInArcsec`: sets the dimension (in arcseconds) of the high-pass filter adopted by the PGLS method for removing possible distortions introduced by the GLS. If it is 0, Unimap calculates the best dimension of the PGLS filter by using an iterative approach, where the convergence is controlled in the similar way as it is for the GLS algorithm. If the automatic setting doesn't provide a satisfactory result, a fine-tuning of this parameter can enhance the quality of PGLS map (see Section 3.2.4.5).

- `wglsDThreshold`: threshold for the detection of distortions introduced by the GLS algorithm, for the building of the WGLS mask (mask of artifacts). The greater the value, the smaller the number of detected artifacts. The optimal value (wglsDThreshold = 0) is computed by varying the value of the threshold and by performing a statistical analysis on the distortion map.
- `wglsGThreshold`: threshold for the building of the WGLS mask (mask of artifacts), starting from the anchor points provided by the wglsDThreshold parameter. The greater the value, smaller the extent of the masked artifacts. The optimal value (wglsDThreshold = 0) is computed by varying the value of the threshold and by performing a statistical analysis on the distortion map.
- `maskToUse`: the masks that should be considered by Unimap. The Scanamorphos_CalibrationBlockDrift mask is added to the default masks because the JScanam module scanamorphosBaselinePreprocessing is used within the Unimap preprocessing to remove the drift due to calibration blocks.
- `rewriteFramesFitsFiles`: set to True if you want to generate the FITS files used by Unimap as the input dataset and to save them in the data directory (see Section 3.2.4.4).
- `rewriteParametersFile`: set to True if you want to rewrite the parameter file in the data directory (see Section 3.2.4.4).
- `cleanDataDir`: set to True if you want to remove input and output files from the data directory (see Section 3.2.4.4).

After setting the initial parameters, the first part of the script creates a list with the Level1 frames (frameList) of the observations that you want to reduce. This list is filled using a ‘for’ statement, within which:

- data are downloaded and Level1 frames extracted. The best way is to download the data from the Herschel Science Archive (HSA). For alternative methods see Sec 1.4 of the DAG.

```
obs = getObservation(obsid, useHsa=True, instrument="PACS")
level1 = PacsContext(obs.level1)
frames = level1.averaged.getCamera(camera).product.selectAll()
```

- if a Solar System Object, the coordinates in the frames are set to the object reference system;
- pixel coordinates are assigned at the dataset and the optical distortion correction is applied

```
frames = photAssignRaDec(frames, calTree=calTree)
frames = convertToFixedPixelSize(frames, calTree=calTree)[0]
```

Once the frameList is complete, it can be passed to the runUnimap task, together with the parameter previously defined.

```
     unimapMap = runUnimap(framesList = framesList, filterSize =
filterSizeInArcsec,
     startImage = startImage, outputPixelSize = outputPixelSize,
     wglsDThreshold = wglsDThreshold, wglsGThreshold = wglsGThreshold,
     minimumChange = minimumChange, masksToUse = masksToUse, unimapDir =
unimapDir,
     mcrDir = mcrDir, dataDir = dataDir, dataDirName = dataDirName,
     rewriteFramesFitsFiles = rewriteFramesFitsFiles,
     rewriteParametersFile = rewriteParametersFile, cleanDataDir = cleanDataDir,
     sync = sync, pixNoise = pixelNoise,numProc = numProc)
```

3.2.4.3. Run Unimap

RunUnimap works in two steps: the first converts the Level1 frames into FITS files suitable for Unimap and writes the Unimap parameter file, the second spawns Unimap from HIPE and creates the final maps, together with intermediate evaluation products. All files generated by the task are saved in the directory dataDir/firstObsid_lastObsid/camera, and this directory is removed if cleanDataDir=True.

RunUnimap creates a Frames FITS file for every obsids by adopting the naming convention unimap_obsid_camera_obsid.fits, and writes the Unimap parameter file unimap_par.txt. Then it launches the Unimap pipeline. While the pipeline is running, the log is displayed into the HIPE Log and a unimap_log.txt file is generated at the end of the process. The Unimap log is also saved within the output context (unimapMap) into the unimapLog TableDataset (see below).

For a detailed description of the pipeline, read the Unimap User's Manual, while here the main steps of the pipeline are summarised:

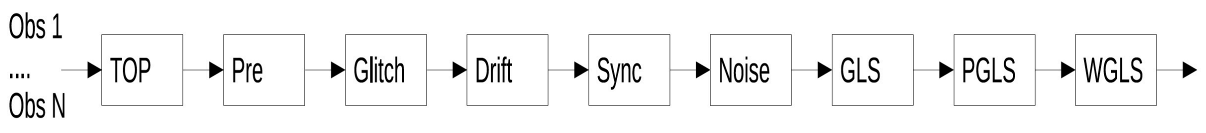


Figure 3.13. The Unimap pipeline.

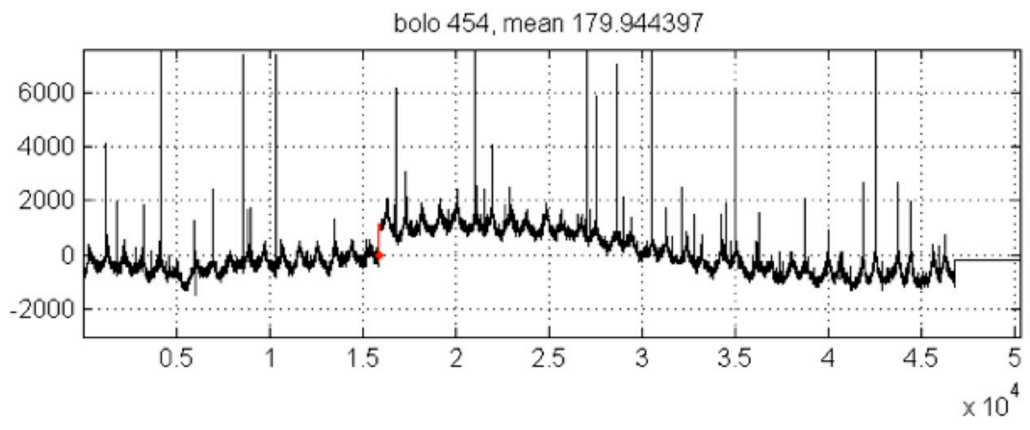


Figure 3.14. Detection of a jump (in red) on the detector timeline.

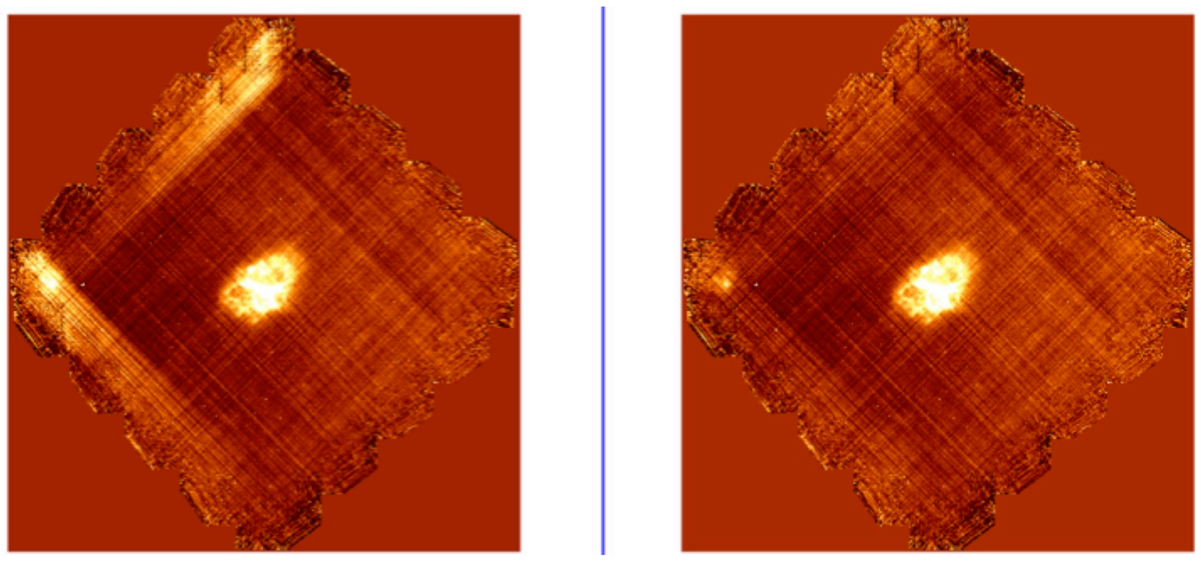


Figure 3.15. The Crab field before and after onset correction.

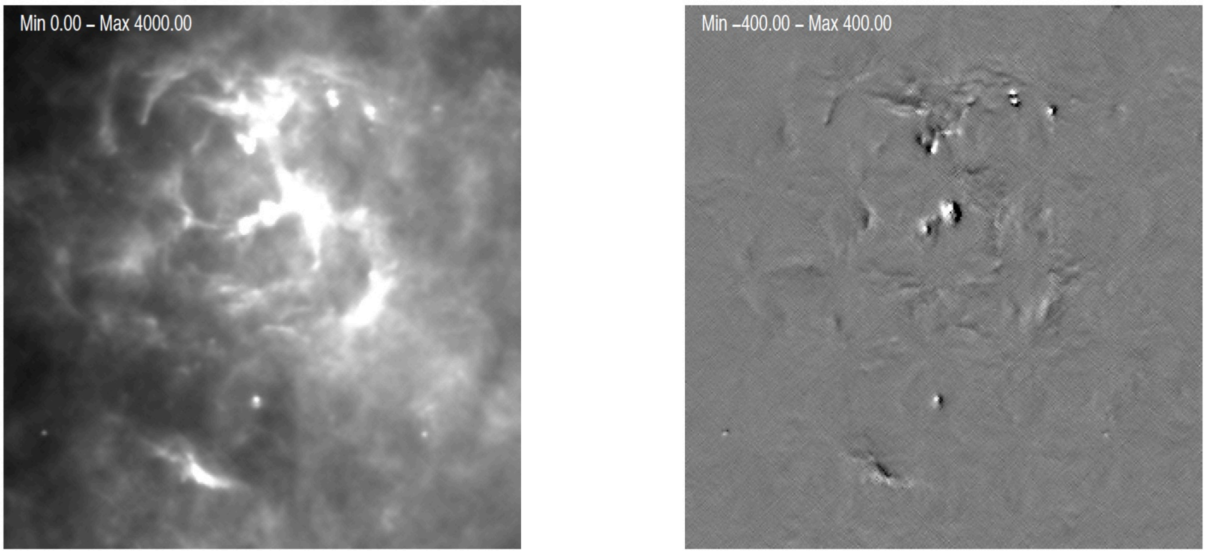


Figure 3.16. Effect of the synchronisation on a Galactic field (left image). On the right image is the difference image between a map with and another without the time-shift compensation. The impact of the compensation is appreciated in particular on maps acquired at fast scan speed (60 arcsec/sec).

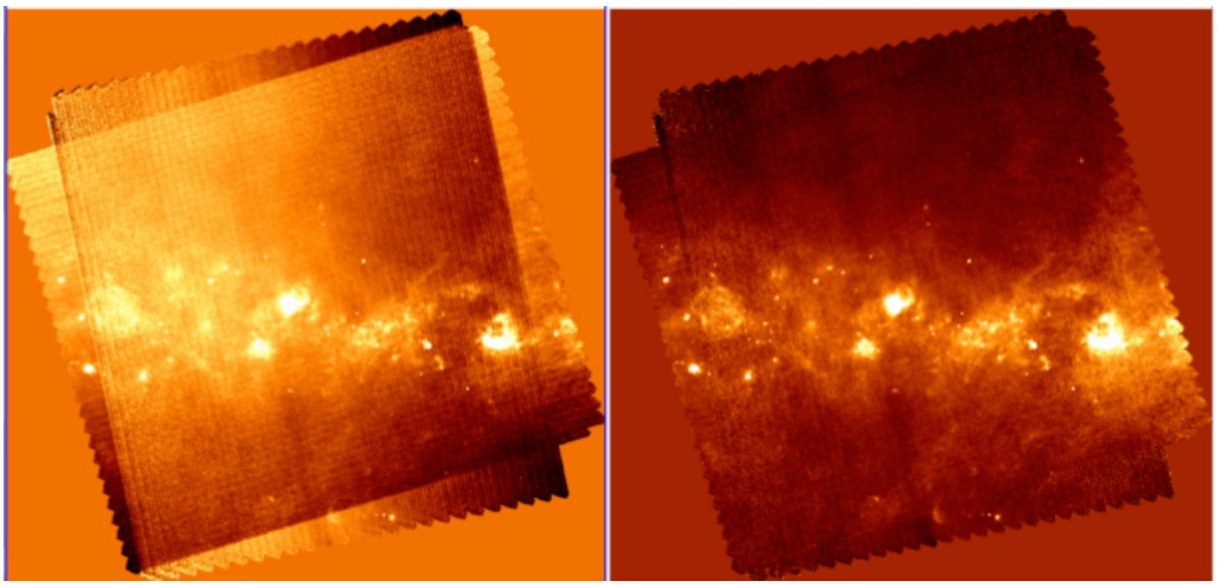


Figure 3.17. A Galactic Plane field before and after drift correction.

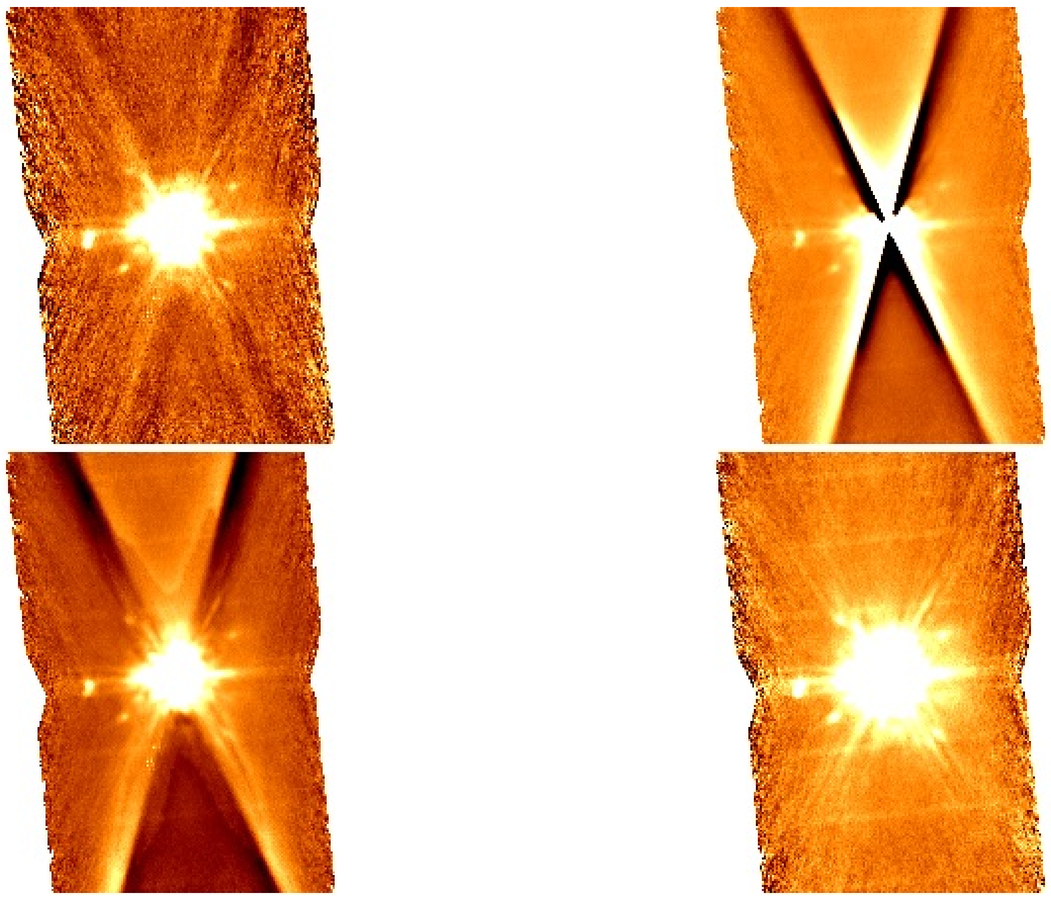


Figure 3.18. 70 micron maps of the bright Ceres: Clockwise: Naïve, GLS without pixel-noise compensation, GLS with pixel-noise compensation, WGLS. The strong distortions introduced by the GLS (without pixel noise) are not efficiently recovered by the WGLS, while they are completely compensated by taking into account the pixel noise within the GLS algorithm

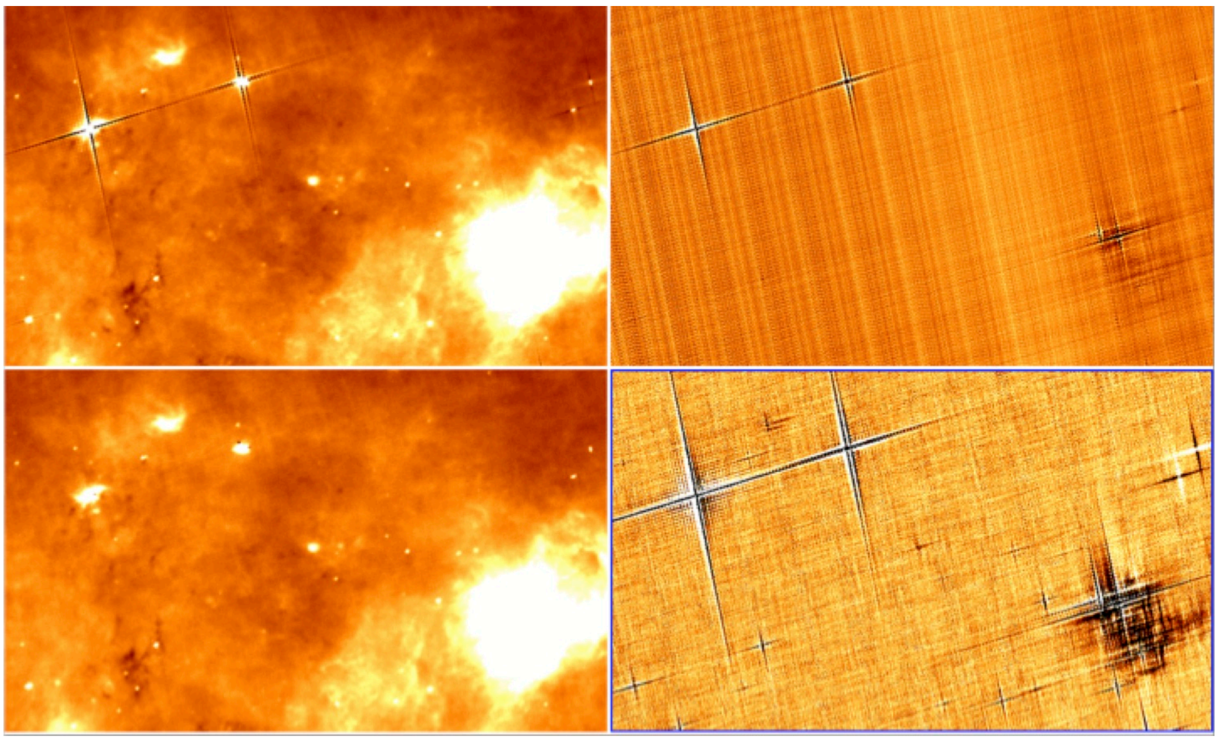


Figure 3.19. Clockwise: GLS, GLS-minus-Rebin, PGLS, GLS-minus-PGLS maps. The distortions introduced by the GLS algorithm are highlighted on the difference maps and they are no longer present in the PGLS map.

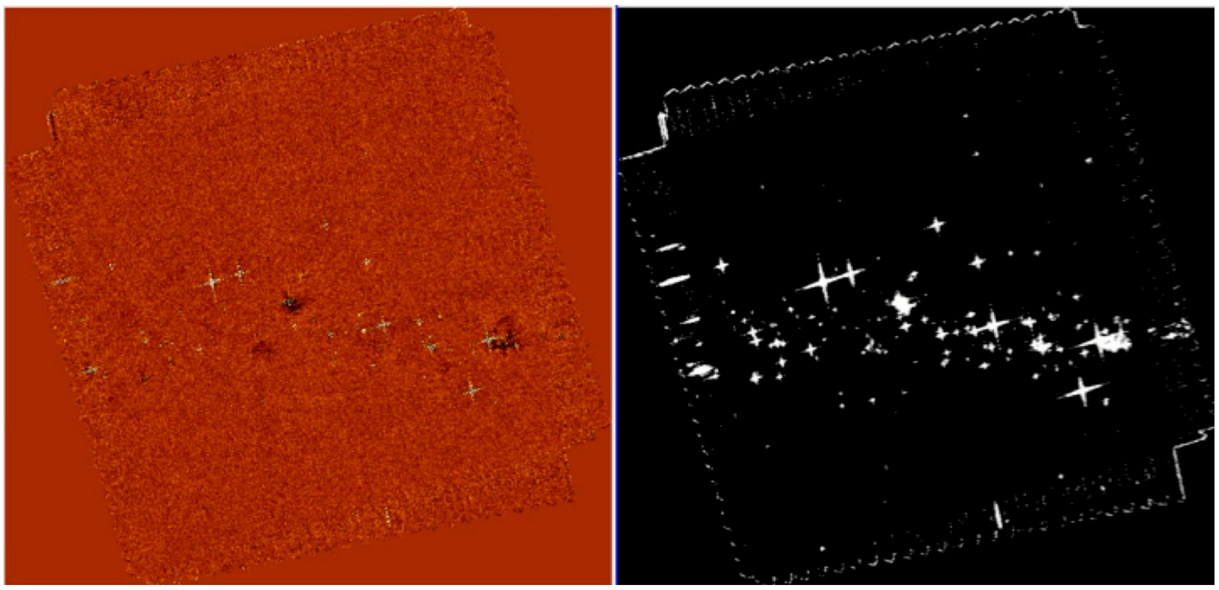


Figure 3.20. The distortions present in the PGLS-minus-GLS map (left image) are properly masked (right image) by the WGLS algorithm.

- `TOP`: Time Ordered Pixels. In the first step the sky grid is built and every readout is assigned to a sky pixel according the specified coordinate system (equatorial or galactic). The pointing matrix is built with assumption that each readout is simply associated to the centre of the sky pixel where the readout falls (hard-pointing [5]). With this assumption, the pointing matrix is diagonal and simple to handle (at least with respect to a full matrix), but it also introduces an additional source of noise, that is the pixel noise. In addition, the module removes the offsets by subtracting the median value from each timeline, and it generates a coverage map (cove_full.fits).

- `Pre`: This module works on the detector timelines. Signal jumps are identified and flagged, and onsets are corrected by subtracting an exponential fit.

- `Glitch`: this module detects and flags glitches by adopting a sigma-clipping approach that exploits the spatial redundancy. To simplify the glitch detection, a high pass filter is first applied over the timelines, to remove long-scale drift and 1/f noise. A glitch mask (flag_glitch.fits) is created by this module.

- `Sync`: this module corrects the shift along the timelines that may be due to errors in the timing system or to delay in the processing chain. The applied method is described in [4].

- `Drift`: this module removes the drift affecting the timelines by performing a polynomial fitting. The fit procedure is based on an iterative approach that uses the Subspace Least Square (SLS) drift estimation [2, 5]

- `Noise`: this module performs four important steps in the reduction chain.

  - It estimates the noise spectrum and constructs the noise filters to be used by the GLS algorithm. The GLS implementation used by Unimap exploits time-domain filtering.

  - It uses the flag-removed TOP for making a simple projection and generate a naive map (*img_re-bin.fits*).

  - It estimates the pixel noise, that is a source of noise due to the random displacement of the sampling point with respect to the centre of the sky pixel. The pixel noise is estimated by adopting a destriper map, obtained by fitting the timelines with a piecewise function within an ALS approach [5]. The length of the constant part of the piecewise function can be explicitly controlled by the user through the noise_pix_win parameter (from Unimap 6.5.1).

  - It generates an error map (img_noise.fits, stored in the ‘error’ product within the map context, see below) by combining the contributions of the pixel noise and the electronic noise, as it is described in [5]. The error map is the estimated standard deviation of the noise affecting the final map and is created using an approach similar to the Sanepic mapper [6]. At the time of Unimap release 6.4.3, the reliability of the error estimate has only just been checked by the PACS team and it is certain only to a factor of +/-2.

- `GLS`: this is the main module and the most time-consuming part of the whole pipeline. It computes the GLS estimator via an iterative approach. GLS is used for removing 1/f noise. For an accurate result, all other sources of noise (glitch, drift, offset) must be removed within the previous tasks. The main parameters used by the GLS module are the initial guess for starting the iteration and the level of precision to achieve between iterations. An additional iterative algorithm can be nested inside the GLS method to control the pixel noise removal. This compensation avoids the distortions in the GLS map at the positions of bright sky emission and strong signal gradients, with the drawback of increasing the computational time. If the increase of running time is not acceptable, the pixel noise compensation can be disable and the distortions introduced by the GLS algorithm can be estimated and removed by means of the post-processing modules.

- `PGLS`: this module estimates the GLS distortions using a non-linear, median high-pass filtering, and subtracts them from the GLS image to produce a clean Post-GLS map [1]. The critical parameter is the dimension of the high-pass filter, which should be enough large to include the distortions but not so large as to amplify the background noise of the image. Generally, the increase of the noise level introduced by the PGLS is negligible for images with high signal-to-noise ratio, which are also the images for which distortions are generally present. Nevertheless, it is not always obvious that the PGLS image is better than the GLS one, and that is why the WGLS module was introduced. The PGLS module also saves the difference map between GLS and PGLS (*delta_gls_pgls.fits*), which is a useful image to evaluate the distortion estimate.

- `WGLS`: this module applies the PGLS method only in the map sky regions where the distortions are relevant (Weighted-GLS). Distortions are detected and masked using the wglsDThreshold and wglsGThreshold parameters. In the WGLS map, the PGLS method is applied only where it is necessary, by minimising the injection of background noise generated by the PGLS. The mask of distortion is saved by the WGLS module for evaluation purposes (*flag_wgls.fits*).

The output of `runUnimap` task is a single SimpleImage, called unimapMap, which contains multiple Tabledatasets. The products that are in the unimapMap SimpleImage (HPF) are listed below. For every product the name of the file generated by Unimap is specified, and these are also saved into the working directory if *clearDataDir* = False:

- `image` is the WGLS map (*img_wgls.fits*) or the GLS map (*img_gls.fits*);
- `pgls/als` is the Post-GLS map or the ALS map (*img_pgls.fits*);
- `gls` is the GLS map (*img_gls.fits*);
- `naive` is the Naïve map (*img_rebin.fits*);
- `error` is the error map (*img_noise.fits*).
- `coverage` is the coverage map obtained with no flagged data (*cove_full.fits*).
- `unimapLog` is the unimap log file (*unimap_log.txt*).

The image layer is assumed as the optimal map to provide; if the pixel noise compensation is applied, the image layer is the GLS map, since the post-processing was excluded, and the pggs product is replaced by the ALS map. Differently, if the pixel noise is skipped (e.g. because it was not necessary according to the signal characteristic of the field or because the dataset is too large) the image layer is the WGLS map. The error map is always the same for both cases.

3.2.4.4. Advanced use of the unimap script

The output of `runUnimap` task (unimapMap) contains all the relevant maps generated by Unimap and the other files are not saved into the working directory if the *clearDataDir* is set equal True (default value).

However, if a user wants to inspect the Unimap files for evaluation purposes, *clearDataDir* must be set to False. The expert user can then exploit the *rewriteFramesFitsFiles* and *rewriteParametersFile* variables for modifying all the unimap parameters.

The Unimap parameter file (*unimap_par.txt*) is in the installation directory and it is copied to the working directory for every Unimap run. It contains several parameters, for the 9 modules of the Unimap pipeline previously described, and only 9 parameters are set in and propagated by the interactive script.

You can run the script several times, modifying the default values of Unimap parameters by editing the *unimap_par.txt* in *dataDir/firstObsid_lastObsid* (NOT the one in the installation directory) and by launching the interactive script with *rewriteFramesFitsFiles* = False and *rewriteParametersFile* = False (for this, a first run must have already been done). In this way, the formatting part of the pipeline will be skipped and it just invokes unimap (so it will not create Frames FITS files and the unimap_par.txt will not be overwritten). This all saved a significant amount of running time. In a similar way, if the you want to modify the Unimap parameters only via the ipipe script, set *rewriteFramesFitsFiles* = False and *rewriteParametersFile* = True and you will save on the running neceesary for generating again the Frames FITS file, while the parameters file will be updated with your new configuration.

Finally, an additional ipipe script (memory_optimized) is provided within the Pipelines menu in HIPE, to allow one to run Unimap while using the minimum amount of memory. In this script, the complete `runUnimap` task (it takes level1 frames and provides Unimap maps), is split in 2 subtasks in a way designed to minimise the amount of data loaded in the RAM. It is particularly useful for processing large observations or/and if several obsIDs are processed all together in a single run.

3.2.4.5. Quality inspection of final maps

With pixel noise compensation

In very few cases the estimated pixel noise is not applied in the optimal way and a different gain value with respect the default value (1) must be applied so you can avoid residual distortions or to reduce the noise level of the final map.

In general, if the gain is high, the electronic noise becomes negligible and the GLS map tends to the Naïve one; on the contrary, if the gain is the very low, the pixel noise becomes negligible, and you have a GLS map without pixel noise compensation (and the the post-processing is necessary because of possible distortions on the GLS map).

The goal is to have a gain enough high to match all the bright emission, for which the Naïve map is the best approximation, while the GLS can introduce distortions. To do that you can inspect the morphology of the pixel noise map (noise_pix.fits), checking that the pixel noise emission is concentrated around bright sources and sky emission and, if necessary, modify the gain value to obtain a better result.

With post-processing

The WGLS map is the main and most reliable product if the pixel noise compensation is not applied. It removes artifacts by injecting a small amount of noise in delimited regions of the map.

The automatic setting of the post-processing parameters guarantees, in most cases, that the PGLS and WGLS method are applied when and where they are necessary, but in some cases a fine-tuning of the PGLS/WGLS parameters can generate better results.

To evaluat the quality of the post-processed Unimap maps one can inspect the Delta Images, which are the differences between Unimap maps. Delta Images are generated and stored in the working directory if *clearDataDir* = False.

- `delta_gls_pgls.fits`: delta map between GLS and PGLS maps. It highlights the distortions introduced by the GLS method. The artifacts shown in the delta_gls_pgls.fits image depend on the dimensions of the high-pass filter used by the PGLS method (see Figure 3.17 and Figure 3.18).
- `delta_gls_rebin.fits`: delta map between GLS and Naïve maps. It provides an unbiased representation of the artifacts because it is not influenced by the distortion evaluation performed by the PGLS method. Nevertheless, it can be very noisy, due to the presence 1/f noise of the naïve map, and it may not be reliable for the evaluation of artifacts of low flux levels (Figure 3.17).
- `delta_pgls_rebin.fits (delta_wgls_rebin.fits)`: delta map between PGLS (or WGLS) and Naïve maps. It provides a representation of the goodness of the artifact removal performed by the PGLS and WGLS methods. Only the pattern of the correlated noise (i.e. stripes following the scan-line directions) should be visible in the delta_pgls_rebin.fits and delta_wgls_rebin.fits images if the algorithms have correctly removed the artifacts.

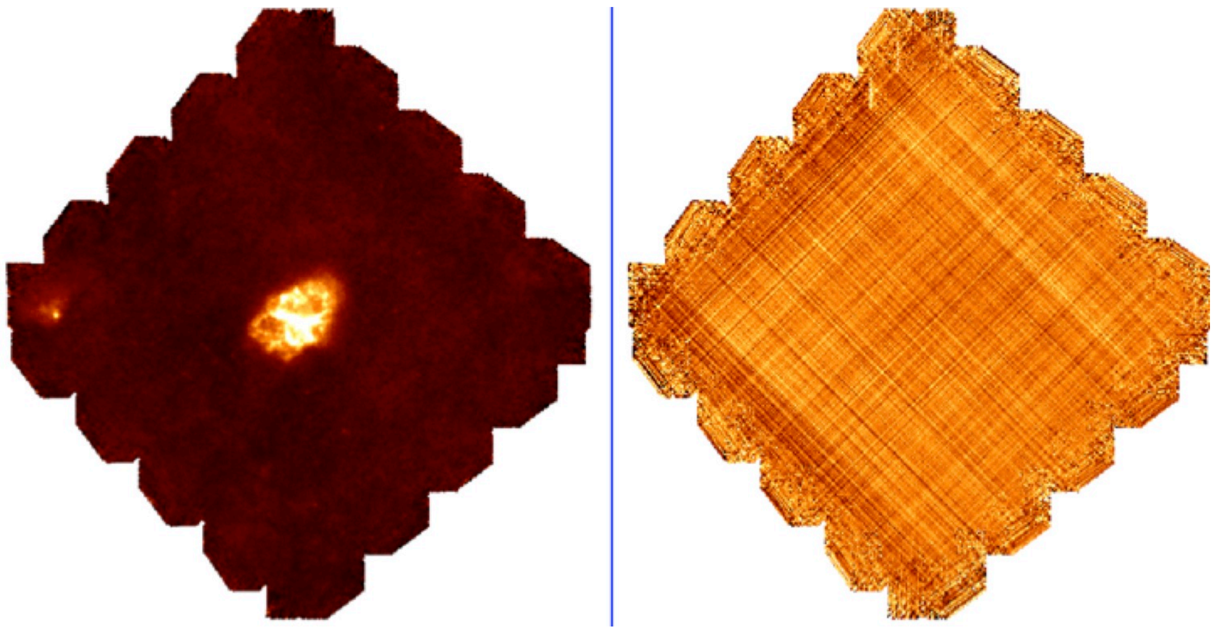


Figure 3.21. . Is the WGLS map (left image) reliable? Artifacts visible in the PGLS-minus-GLS map are not present into the WGLS-minus-Rebin map (right image), where only the pattern of the correlated noise can be seen.

By inspecting the Delta images, you can learn the following:

- Post-processing is not necessary (even if the pixel noise compensation is not applied) if residuals are not present in the *delta_gls_pgls.fits* or *delta_gls_rebin.fits* maps. In this case, the PGLS and WGLS modules don't increase the quality with respect to the GLS map.
- If the *delta_pgls_rebin.fits* map shows residuals, the length of the PGLS high-pass filter (filterSize) was too short to remove all the artifacts.
- If the *delta_wgls_rebin.fits* map shows residuals, then in addition to tuning the length of the high-pass filter, the WGLS thresholds used for building the mask (*wglsDThreshold* and *wglsGThreshold*) should be fine-tuned.

3.2.4.6. Unimap Referencess

[1] L. Piazzo et al., ”Artifact Removal for GLS Map Makers by means of Post-Processing”, IEEE Trans. on Image Processing, Vol. 21, Issue 8, pp. 3687-3696, 2012.

[2] L. Piazzo et al., "Drift removal by means of alternating least squares with application to Herschel data", Signal Processing, vol. 108, pp. 430-439, 2015.

[3] L. Piazzo et al.: 'Unimap: a Generalised Least Squares Map Maker for Herschel Data', MNRAS, 447, pp. 1471-1483, 2015.

[4] L. Piazzo et al.: 'Least Squares Time-series Synchronization in Image Acquisition Systems', IEEE Trans. on Image Processing, 2016 , submitted.

[5] L. Piazzo et al.: 'Generalised Least Squares Image Estimation in the Presence of 1/f and Pixel Noise', manuscript in preparation, 2016.

[6] Patanchon et al.: 'SANEPIC: A Mapmaking Method for Time Stream Data from Large Arrays', ApJ, Vol. 681, pp. 708-725, 2008.

3.3. Chopped point source pipeline

As mentioned above, there is no particular interactive pipeline script for observations executed via the chopped point source (PS) AOT. The reason for this is that this observing mode was only rarely used during the Science Demonstration Phase (SDP), and the PACS ICC had recommended to use the superior mini scan map mode instead. Nevertheless, there is still a large amount of data obtained for calibration purposes that may also be exploited scientifically. Therefore, we here describe a processing script that can be used as a template. It contains the same processing steps the SPG performs, but provides a better readability. It also allows the user to include passages to inspect the intermediate data products. Note that there are not many alternatives how to process chopped PACS photometer data.

3.3.1. Differences in processing between chopped and scanned observations

There is a number of differences in the data structures of chopped and scanned observations that are also reflected in the various processing steps and the tasks that are applied. In short, chopped data do not need highpass filtering nor deglitching. The latter is done automatically when coadding the individual frames of a chopper position by applying a sigma clipping algorithm. In addition, the final map is created with a shift-and-add procedure that combines the four images of two chopper and the two nod positions to a single combined signal. A drizzling algorithm as provided for scan map data is not implemented.

Note

The flux calibration is tuned to the scan map observing mode. Point source fluxes derived from the chopped PS AOT were known to be too low by a few percent. A new task *photFluxCalPsToScan* was introduced to correct for this scaling difference.

3.3.2. A typical processing script for chopped PS AOT data

Let's first set up the session with a few pre-definitions.

```
from java.util import ArrayList

# define the OBSID of the observation (chopped PS AOT)
obsid = 1342244899

# select the blue or red channel
camera = 'blue'

# choose an putput pixel size of the final map
pixsize = 1.0
```

This section sets the options and definitions that are needed to run the script. Set the OBSID of the corresponding observation to process. Each observation comes with a short and a long wave data set. Select `blue` for the short band (70/100 µm) and `red` for long band (160 µm). Finally, select a pixel size in arcseconds of the final map. Good values to start with are 1.0 for the blue and 2.0 for the red band.

Now, we retrieve the necessary detector and auxiliary data.

```
# retrieve the ObservationContext from the archive
obs = getObservation(obsid=long(obsid),useHsa=True)

# get the calibration applicable to this observation
calTree=getCalTree(obs=obs)

# retrieve the auxiliary data
pp=obs.auxiliary.pointing
```

```
oep = obs.auxiliary.orbitEphemeris
timeCorr = obs.auxiliary.timeCorrelation
photHK=obs.level0.refs["HPPHK"].product.refs[0].product["HPPHKS"]

# Is it a solar system object?
isSso = isSolarSystemObject(obs)

if (isSso):
    horizons = getHorizonsProduct(obs)
else:
    horizons = None
if (horizons == None):
    hp = None
else:
    hp = obs.auxiliary.horizons
```

The task *getObservation* downloads the entire *ObservationContext* from the Herschel Science Archive (HSA). Be sure to have access to the HSA and you are already logged in via the HIPE GUI. The task *getCalTree* extracts the applicable calibration files from the database that comes with your HPIE/HCSS installation. Then, a number of certain auxiliary data are extracted from the *ObservationContext*. Especially, the variable photHK contains valuable information about the instrument status, the so called Housekeeping Data.

```
# extract level 0 data and apply the slicing
pacsPropagateMetaKeywords(obs,'0',obs.level0)
level0 = PacsContext(obs.level0)
frames = SlicedFrames(level0.getCamera(camera).averaged.product)
del(level0)
```

First we make sure that all the necessary metadata are correctly forwarded from the top layer of the *ObservationContext* to the Level 0 data from which we start the processing. As already mentioned, the processing of chopped PS AOT observations are done using the slicing scheme that is also employed for the spectroscopy data processing, but was discarded for the scan maps. The default slicing rule structures the data according to nod cycles. This is done in the background, so the user does not even notice the difference (except for the final step, when combining the individual maps). Each slice appears as an individual index number in the frames. The tasks internally loop over these items to perform the processing steps. The processing is done on the so called (sliced) frames.

```
# apply coordinates and convert to instrument boresight
frames =
 photAddInstantPointing(frames,pp,calTree=calTree,orbitEphem=oep,horizons=horizons,isSso=isSso,copy
del(pp)
frames = findBlocks(frames, calTree=calTree)
if (not frames.meta.containsKey("repFactor")) :
    frames.meta["repFactor"] = LongParameter(1)
frames = pacsSliceContext(frames, level='0.5')[0]
```

In addition, we join the pointing information with the data by applying the instrument boresight information (*photAddInstantPointing*). To save memory, the pointing product is deleted at this stage. The task *findBlocks* determines the data structure and prepares a table that contains the corresponding information. Finally, some old Level 0 data are missing the *repFactor* keyword in the metadata. In order to be able to run the pipeline on those data as well, we define a default number of 1 for them.

```
# The actual processing begins here:
# mask bad and saturated pixels + those causing electric crosstalk
frames = photFlagBadPixels(frames,calTree=calTree,scical="sci",keepall=0,copy=1)
frames = photFlagSaturation(frames,calTree=calTree,hkdata=photHK,check='full')
frames = photMaskCrosstalk(frames)
```

In the beginning, the script flags the pixel outliers, such as bad and saturated pixels. We also mask those pixels that cause electronic crosstalk. This is done by masking the first column of each detector sub-matrix. Now, we apply the flux calibration (see note above). The first task converts the digital detector readouts into Volts, followed by the flatfielding and the response calibration in Jy.

```
# convert digital detector values into volts
```

```
frames = photConvDigit2Volts(frames,calTree=calTree)

# apply flatfield and response calibration (V -> Jy)
frames = photRespFlatfieldCorrection(frames,calTree=calTree)
```

Now, we add some timing information.

```
# add a column in the status table that lists UTC time
frames = addUtc(frames, timeCorr)
del(timeCorr)

# convert digital chopper position readouts into real angels
frames = convertChopper2Angle(frames,calTree=calTree)

# discard data with still moving chopper
frames = cleanPlateauFrames(frames)
```

This paragraph applies the calibration of the chopping mirror. First, the digital position sensor readout is converted into chopping angles, while *cleanPlateauFrames* flags data that were taken during the transition between two chopper plateaus, i.e. the chopper was neither on the off nor the on position. Then we have to deal with the chop/nod pattern.

```
# identifies a dither pattern, if applicable
frames = photMakeDithPos(frames)

# identifies raster positions of the chop/nod pattern
# (needs photAddInstantPointing)
frames = photMakeRasPosCount(frames)
```

In HSPOT, the observer can specify, if the chopper should perform a dithering pattern. This displaces the on and off positions by a few fractions of a pixel for each chopping cycle. The task *photMakeDithPos* identifies such a pattern and corrects the pointing information accordingly. Then, the global chop/nod pointing pattern is identified by *photMakeRasPosCount* as well.

```
# average all data per chopper plateau (first readout is dropped)
frames = photAvgPlateau(frames,skipFirst=True)
```

There are four readouts per chopper on and off position (plateau). The corresponding flux values are averaged, so that *photAvgPlateau* produces a single value for each. The option `skipFirst=True` discards the first of each of the four readouts, because it was found that it is still affected by the chopper movement and contaminates the measurement. Therefore, we recommend to always use this option. Now, we apply the pointing information to each detector pixel.

```
# only applicable to solar system object observations:
# recenter individual images on moving target instead of referencing to
# celestial coordinates
if ((isSso == True) and (horizons != None)):
    frames =
 correctRaDec4Sso(frames,orbitEphem=oep,linear=False,horizons=horizons,horizonsProduct=hp)

# apply coordinates to individual pixels
frames = photAssignRaDec(frames,calTree=calTree)

# extract pointing information for chop/nod pattern
frames = photAddPointings4PointSource(frames)
```

The first part only applies to solar system objects. They move in time and therefore do not possess a fixed celestial coordinate. In order to avoid trails of the moving target in the final map, the individual frames must be re-referenced to the target position. This is done by *correctRaDec4Sso*. Then, *photAssignRaDec* applies the pointing information (astrometry) to each detector pixel, while *photAddPointings4PointSource* corrects the coordinates for the displacements produced by the chop/nod pointing pattern. It relies on the information extracted previously by *photMakeRasPosCount*.

```
# subtract on and off chooper positions
frames = photDiffChop(frames)
```

This task produces the differential signal of the chopping cycles per dither position. It subtracts a constant signal component that is dominated by the telescope mirror emission. At this stage, we apply a few flux corrections that are based on a thorough investigation of detector response effects. Note that the order of applying these corrections matters. The individual corrections are:

1. Detector non-linearity: For very high source fluxes, the bolometers exhibit a non-linear relationship between the incident signal and the detector response. Applying the task *photNonLinearityCorrection* corrects for this effect.

2. Evaporator temperature: The temperature of the evaporator of the ^{3}He cooler has a small but measurable effect on the detector response. Applying the task *photTevCorrection* provides a correction for this effect at a level of a few percent. For details see: Balog et al. (2013)

3. Scan map based flux calibration scheme: The flux calibration of the PACS photometer is based on scan map measurements, which introduces a time dependent flux scaling difference for the chopped observations. The correction is done by applying the task *photFluxCalPsToScan*. For details see: Nielbock et al. (2013)

```
# apply non-linearity correction
frames = photNonLinearityCorrection(frames,calTree=calTree,scical="sci")

# apply the photometer evaporator temperature correction
frames = photTevCorrection(frames, calTree=calTree, hk=photHK)

# correct the flux calibration for PACS chopped photometry observations
frames = photFluxCalPsToScan(frames)
```

The three dithering positions are averaged. At the same time, a sigma clipping algorithm is applied to account for and mask glitches. Note that we do not perform MMT nor 2nd order deglitching on the chopped/nodded data.

```
# combine images of the three dither positions done by the chopper and
# deglitch data via a sigma clipping algorithm
frames = photAvgDith(frames,sigclip=3.)

# subtract on and off nodding positions
frames = photDiffNod(frames)

# combine all nod cycles
frames = photCombineNod(frames)
```

Now that we have the full set of differential signals of the chopper cycles, we also subtract the nod positions from each other. In this way, small scale differences in the telescope background due to the varying line of sight caused by the chopper positions cancel out. Finally, we combine all data to a coherent signal timeline. If the object is bright enough, one should be able to see the characteristic 2x2 chop/nod pattern on each of the individual frames. In order to improve the S/N, we combine the flux of the four object images by a shift-and-add algorithm.

```
# produce a map for each slice
mapcube =
 photProjectPointSource(frames,allInOne=1,calTree=calTree,outputPixelsize=pixsize,calibration=True)
```

This is done per slice, so it produces one map each. In this way, only the centre of the map contains the valid co-added image of the source. The eight images around the central one are to be disregarded, as they are just the byproduct of the shift-and-add algorithm, and do not contain any additional information. In particular, they must not be used to try to improve the S/N of the target. The central image is already the final result.

This image cube is finally merged to a single map product by applying the *mosaic* task.

```
# combine the individual maps of the slices to a single map
images = ArrayList()
for si in mapcube.refs :
  images.add(si.product)
pass

map = images[0]
histo = map.history
meta  = map.meta
if (images.size() > 1):
    map = MosaicTask()(images=images, oversample=0)
    h = map.history
    histo.addTask(h.getAllTasks()[0])
    map.history = histo
    map.meta    = meta
    del(h)
del(images,si,mapcube,histo,meta)
```

The result is a single Level 2 map that contains both the final image and a coverage map. Finally, we produce a noise map from the coverage map and add it to the Level 2 product.

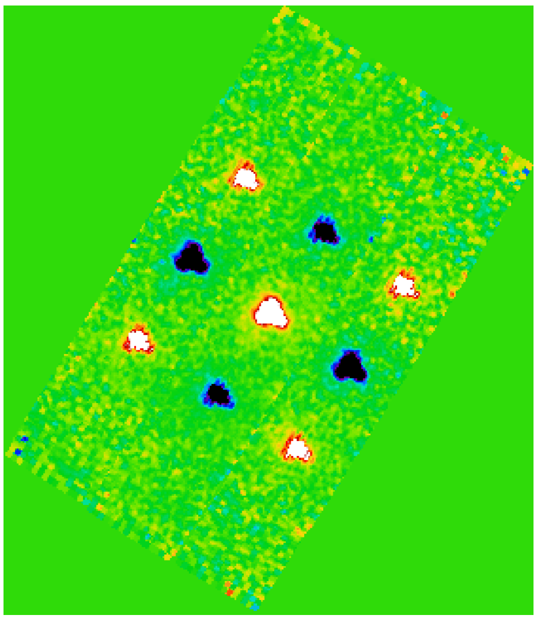


Figure 3.22. Final map of a chopped PS AOT observation. Only the central image of the object contains the correct properties. The surrounding eight images are a byproduct of the shift-and-add algorithm and must not be used.

```
# add a noise map
map = photCoverage2NoisePointSource(map)
```

The task *photCoverage2NoisePointSource* that uses the *photCoverage2Noise* task originally defined for scan maps to produce a sensible error map from the coverage map. This version adds some scaling factors that were derived empirically by analysing a large set of calibration observations. The resulting error map should - at this time - only be taken as a reasonable estimate of the noise in the final map. However, we recommend to check these values during the photometry again. This concludes the processing of data obtained with the chopped PS AOT.

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 also contain information about how to branch off after performing the basic calibration and about how to perform aperture photometry on your PACS images.

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 ChapChapter 3).

This chapter contains:

- The second level deglitching
- Reducing mini-maps
- Solar System objects
- Branching off after level1
- Aperture correction

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:

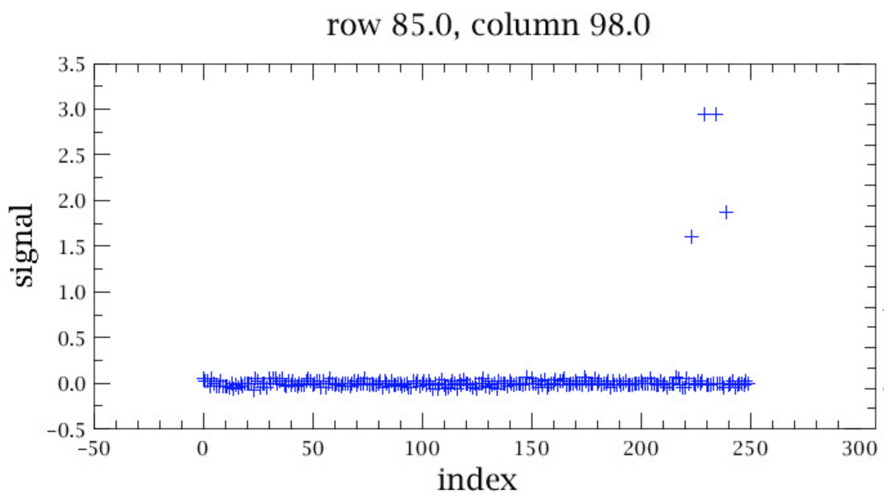


Figure 4.1. Vector of roughly 250 signal contributions to map pixel (85, 98)

4.3.1. Pre-requisites and what is second 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 secondLevelDeglitchingTask.

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 secondLevelDeglitchTask
```

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

```
mapIndex = MapIndexTask()
secondLevelDeglitch = secondLevelDeglitchTask()
```

Note

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). Usually these aliases are already created so you don't need to redefine them.

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 = secondLevelDeglitch(mi, frames)
```

Now we will describe these tasks and their parameters.

4.3.3. The most important syntax options

The second step—map = secondLevelDeglitch(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 = secondLevelDeglitch(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. 4.3.7) 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 = secondLevelDeglitch(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*:

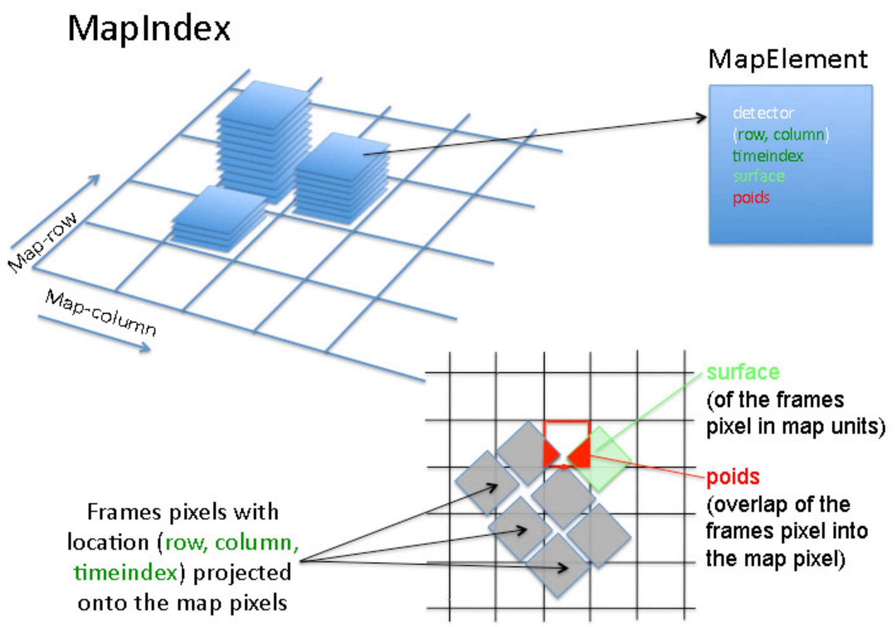

MapIndex
MapElement
detector
(row, column)
timeindex
surface
poids
Map-row
Map-column
surface
(of the frames pixel in map units)
poids
(overlap of the frames pixel into the map pixel)
Frames pixels with location (row, column, timeindex) projected onto the map pixels

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:

```
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. 4.3.4.1. 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 slimMapIndex 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 quickly. 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:

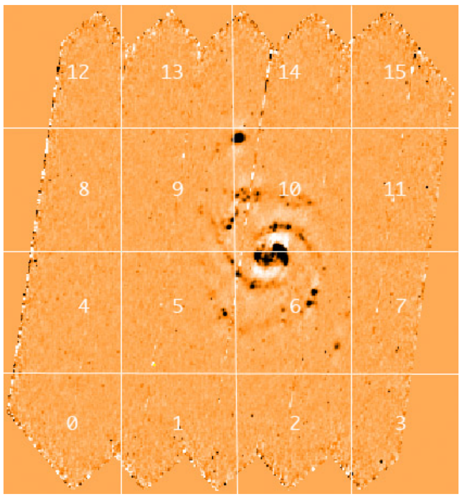


Figure 4.3. The numbering scheme of the tiles for iterative deglitching. This map is sliced into 4 rows and 4 columns. This results in 16 tiles.

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 secondLevelDeglitchTask. The value for this parameter is always None (at first). In the first loop, None tells the secondLevelDeglitchTask 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:

```
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 = secondLevelDeglitch(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.

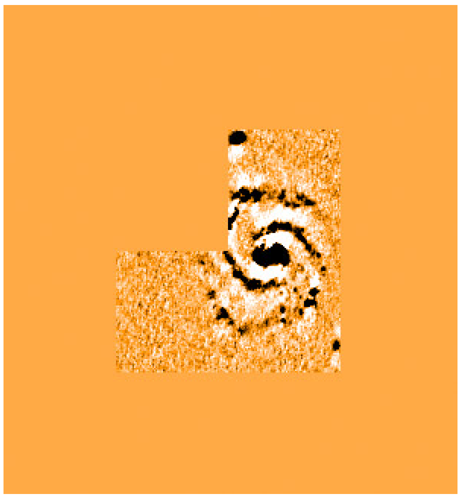


Figure 4.4. Deglitching slices 5,6 and 10

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:

```
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
    secondLevelDeglitch(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 secondLevelDeglitch task

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

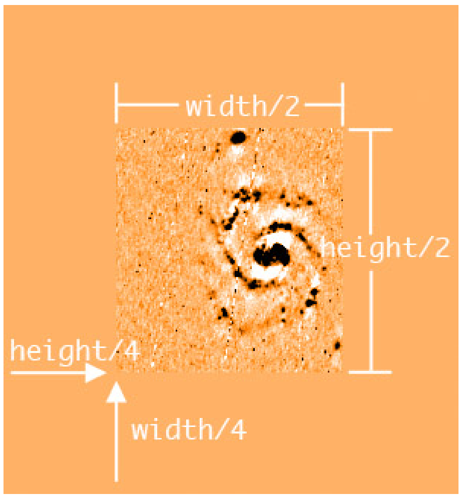


Figure 4.5. Deglitching performed with the submap option

- `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 = secondLevelDeglitch.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. 4.3.5.1.
- `algo`: a customized Sigclip algorithm. See Sec. 4.3.2 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).

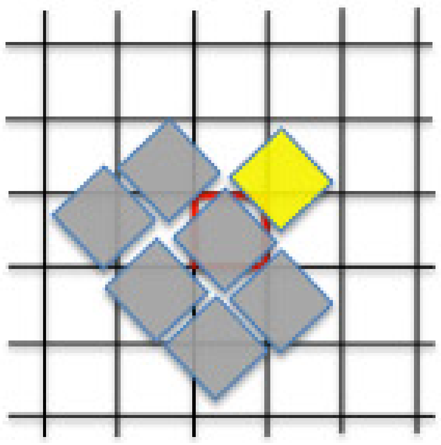


Figure 4.6. Mapping a small but strong source

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 secondLevelDeglitchingTask 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 (mapDeglitch)

One of our latest achievements is the MapDeglitchTask (mapDeglitch). It virtually does the same as the secondLevelDeglitchTask, 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 secondLevelDeglitch (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 secondLevelDeglitchTask with the Mapindex.

```
s = Sigclip(10, 4)
s.mode = Sigclip.MEDIAN
s.behavior = Sigclip.CLIP

mapDeglitch(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

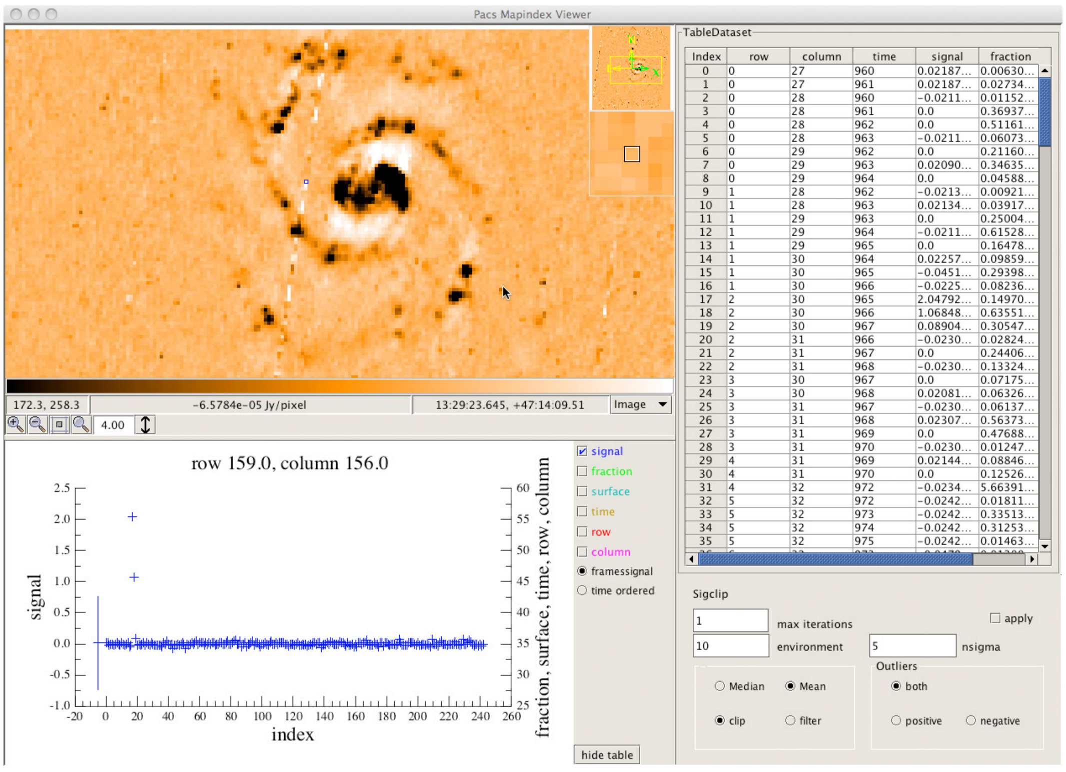
A MapIndex contains a lot of interesting data that is very useful to be analysed. The tool for doing this is the MapIndexViewer. It is called like this:

```
MapIndexViewer(mi, frames)
```

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

```
MapIndexViewer(mi, frames, img)
```

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



TableDataset

Index	row	column	time	signal	fraction
0	0	27	960	0.02187...	0.00630...
1	0	27	961	0.02187...	0.02734...
2	0	28	960	-0.0211...	0.01152...
3	0	28	961	0.0	0.36937...
4	0	28	962	0.0	0.51161...
5	0	28	963	-0.0211...	0.06073...
6	0	29	962	0.0	0.21160...
7	0	29	963	0.02090...	0.34635...
8	0	29	964	0.0	0.04588...
9	1	28	962	-0.0213...	0.00921...
10	1	28	963	0.02134...	0.03917...
11	1	29	963	0.0	0.25004...
12	1	29	964	-0.0211...	0.61528...
13	1	29	965	0.0	0.16478...
14	1	30	964	0.02257...	0.09859...
15	1	30	965	-0.0451...	0.29398...
16	1	30	966	-0.0225...	0.08236...
17	2	30	965	2.04792...	0.14970...
18	2	30	966	1.06848...	0.63551...
19	2	30	967	0.08904...	0.30547...
20	2	31	966	-0.0230...	0.02824...
21	2	31	967	0.0	0.24406...
22	2	31	968	-0.0230...	0.13324...
23	3	30	967	0.0	0.07175...
24	3	30	968	0.02081...	0.06326...
25	3	31	967	-0.0230...	0.06137...
26	3	31	968	0.02307...	0.56373...
27	3	31	969	0.0	0.47688...
28	3	31	970	-0.0230...	0.01247...
29	4	31	969	0.02144...	0.08846...
30	4	31	970	0.0	0.12526...
31	4	32	972	-0.0234...	5.66391...
32	5	32	972	-0.0242...	0.01811...
33	5	32	973	-0.0242...	0.33513...
34	5	32	974	-0.0242...	0.31253...
35	5	32	975	-0.0242...	0.01463...

Sigclip

max iterations: 1
environment: 10
nsigma: 5
apply [ ]
Median [✓ Mean]
[✓ clip] filter
Outliers: [✓ both] positive negative

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.

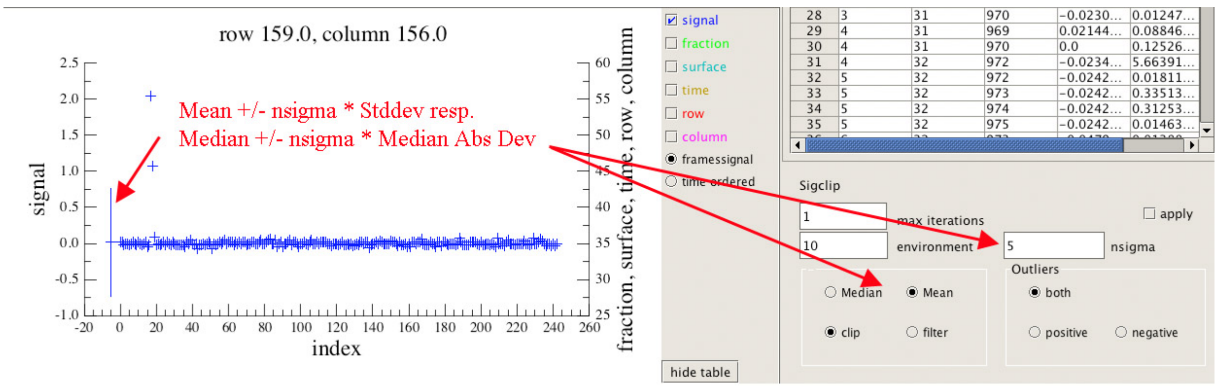
4.3.7.1. Optimising the Sigclip algorithm with the MapIndexViewer for best deglitching results

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.



28	3	31	970	-0.0230...	0.01247...
29	4	31	969	0.02144...	0.08846...
30	4	31	970	0.0	0.12526...
31	4	32	972	-0.0234...	5.66391...
32	5	32	972	-0.0242...	0.01811...
33	5	32	973	-0.0242...	0.33513...
34	5	32	974	-0.0242...	0.31253...
35	5	32	975	-0.0242...	0.01463...

Sigclip

1 max iterations

10 environment

5 nsigma

☐ apply

Median [✓ Mean]

[✓ clip] filter

Outliers: [✓ both] positive negative

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 secondLevelDeglitchingTask. 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 = secondLevelDeglitch(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.

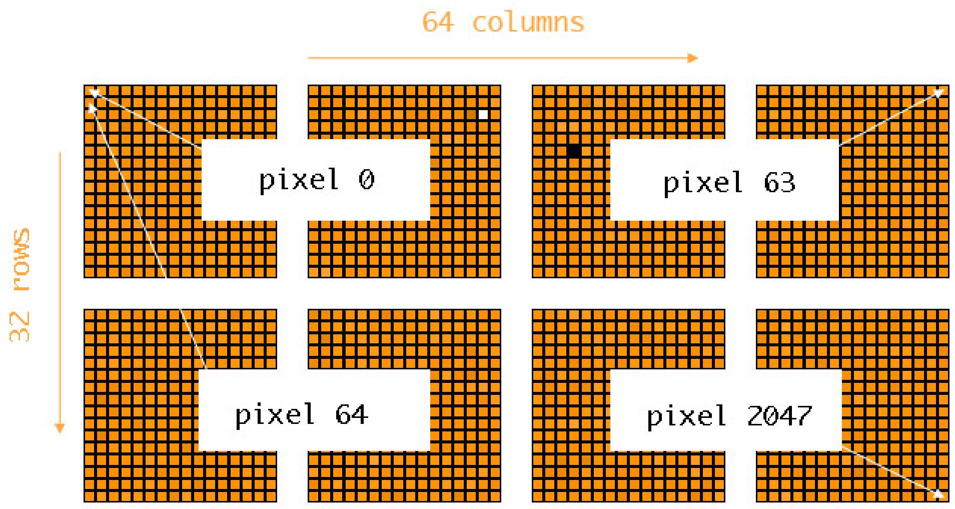


Figure 4.9. Detector numbering scheme for the blue photometer array:

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:

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

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

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

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

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

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

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

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.Gaussian-Noise 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.

As it was mentioned before

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 secondLevelDeglitchTask, 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 secondLevelDeglitchingTask. 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 secondLevelDeglitchingTask. 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.

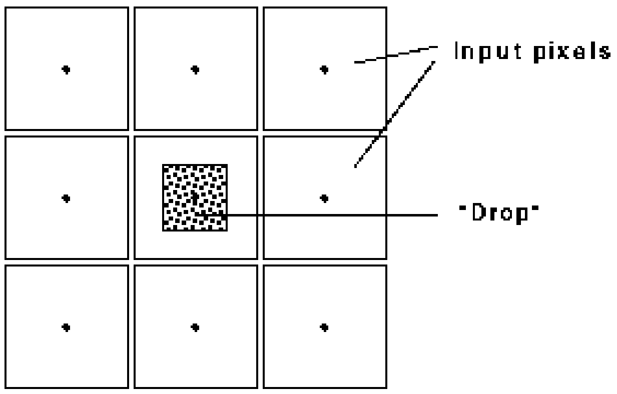


Figure 4.10. Drop size

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

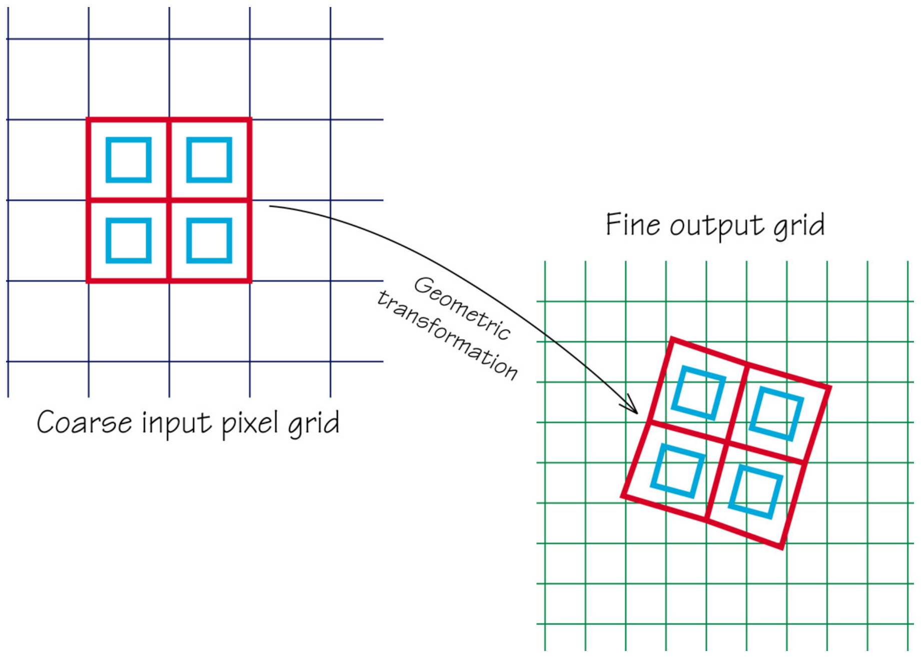


Figure 4.11. Drizzle

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:

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

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

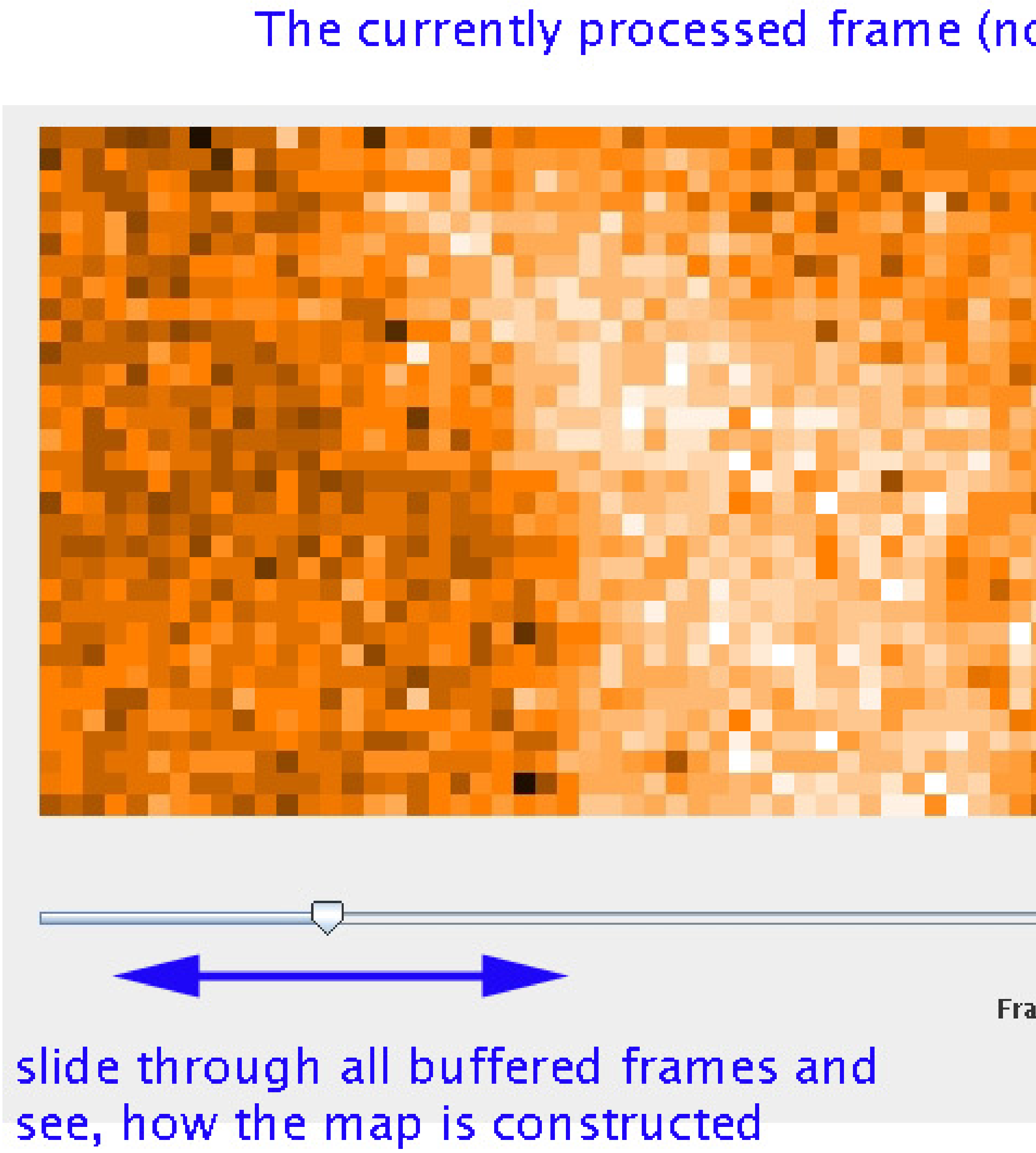
where I(x'y') is the flux of the output pixel (x',y'), a(xy) is the geometrical weight of the input pixel (x,y), w(xy) is the initial weight of the input pixel, i(xy) is the flux of the input pixel and W(x'y') is the weight of the output pixel (x'y'). The geometrical weight a(xy) is given by the fraction 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 currently processed frame (no
Fra
slide through all buffered frames and
see, how the map is constructed

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

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

4.10. Errors in PACS maps

4.10.1. High Pass Filtering

Two different components can be used to describe the noise in a PACS maps processed with HPF and photProject:

- noise per pixel
- cross-correlated noise

The former is given by the error map and it should scale as the inverse of the square root of the observing time or the coverage, until the confusion noise is reached. The latter depends on the mapping itself and on other sources of cross-correlation such as the 1/f noise left after running the HighPass filter task. In principle the noise per pixel should be estimated via error propagation. However, the high-pass filter task completely changes the noise power spectrum of the PACS timeline, thus making it impossible to propagate the errors. In opractice it means that the error maps generated by photProject contain values that has no real meaning. To overcome this problem we use two different methods to estimate the error map depending on the data redundancy.

- **high redundancy case** (few AOR of the same field or high repetition number within the same AOR): the error maps is computed from weighted mean error
- **low redundancy case** (only scan and cross scan without repetitions): the error map is derived from coverage map via calibration obtained from high redundancy case

For the detailed description of the methods and the analysis see Popesso et al. 2012 (A&A submitted)

4.10.1.1. PhotCoverage2Noise

Since it is the most frequent observational case in PACS maps, the low redundancy case has been implemented as a HIPE task PhotCoverage2Noise. The error map is produced via a calibration file using the coverage map. The calibration file parameterized as a function of hp width, output pixel size and pixfrac was produced on data with very high redundancy in medium and fast speed, with a cross-correlation noise correction factor parameterized as well in the same parameter space. The error map is generated applying a best polynomial fit to the coverage map as function of the highpass filter width, output pixel size and drop size. an example of running the task is given below:

```
map=photCoverage2Noise(map,hp=hp_width,pixfrac=pixfrac)
```

The parameters of the task are the following:

- **imageIn**: simple image containing image, coverage, error
- **imageOut**: image containing the new errormap. Usually it should be the same as the original image (that is the default if imageOut is not supplied)

- **hp**: highpass filter width with which the input map was processed
- **pixfrac**: drop size
- **camera**: "blue" or "red"
- **parallel**: parallel mode True or False
- **copy**: copy image or not
- **imageOut**: output image name
- **scical**: "sci", "cal", "all" to select whether you would like to process only the science part (sci) or the calibration part (cal) of your date or both (all
- **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

From the above parameters the input image, the highpass filter width, the pixel size and the drop size is required any any case.

4.10.2. JScanam

During the JScanam processing a standard deviation is calculated from the flux of the detector pixels which contribute to any given map pixel. This standard deviations are stored in a simple image attached to the maps.

4.10.3. Unimap

During the Unimap processing several tasks are used to estimate the contribution of different kinds of noises and prepare the data to the final GLS map maker. The main contributors are pixel noise, and electronic noise. The software combines the pixel noise and the electronic noise to produce an estimate of the noise (i.e. the error) affecting the final map. This is saved as an array dataset attached to the final Unimap product.

4.11. 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.

```
      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:

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

(See Chapter 1 for more instruction.) 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[i] = cleanPlateauFrames(frames[i], calTree=calTree)
      frames[i] = addUtc(frames[i], timeCorr)
      frames[i] = photRespFlatfieldCorrection(frames[i], calTree = calTree)
      frames[i] = photOffsetCorr(frames[i])
      frames[i] = photNonLinearityCorrection(frames[i])
```

The non-linearity of the PACS Photometer flux calibration at very high fluxes is taken into account by applying a correction factor after performing the flat-fielding and responsivity calibration from Volts to Jy by the photRespFlatfieldCorrection task. The correction, applied trhough the task photOffsetCorr and photNonLinearityCorrection, affects only pixels with fluxes above approximatively above 100 Jy.

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)
```

4.12. 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.

4.12.1. 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.

4.13. Branching off after level1

There are several mapmakers available for creating the final image from the basic calibrated data (level1 products). Some of these mapmakers require the level1 product in a specific format. Currently one converter is available in HIPE that converts the level1 data to a format digestible to Scanamorphos, one of the most popular mapmaker.

4.13.1. ConvertL1ToScanam

This code converts level1 frames obtained within the context of the classical HIPE pipeline into level1 frames that can be saved to fits and ingested in Scanamorphos. The frames object is saved in the current directory under the name: [obsid]_[filter]_Scanam_level1Frames.fits the suffix Scanam_level1Frames can be replaced by a user input, and the directory where the file is written can be changed. The usage is very simple, first you need propagate your data up to level1 stage when all the basic calibrations are performed then feed the final frame product (called frames in our example) to the task:

```
succes = convertL1ToScanam(frames, [obsid], [cancelGlitch], [assignRaDec],
[suffix=suffix], [outDir=outDir])
```

Basically the task only needs a level1 frames but optionally there are some parameters that one would want to change.

- **obsid:** obsid is an optional parameter that you can use to provide the obsid if it is not in the metadata.
- **cancelGlitch:** a boolean keyword that you can use to avoid propagating the glitch mask to the scanamorphos file (default is false)
- **assignRaDec:** a boolean parameter to force the computation of the pixels coordinate that scanamorphos needs (default is false). The code checks that the coordinates are present, if they are not there and assignRaDec is false, the code exits.
- **suffix:** a string to use to add to the name of the output fits files. Default is Scanam_level1Frames. **Do not include the fits extension to the string.**
- **outDir:** a string to specify the output directory in which the converted frames object will be stored. **Do not add the final / to the path**.

4.14. Photometry on PACS images - aperture correction

The final product of the PACS photometer is a simple image. The source extraction and aperture photometry of objects on images is described in Sec 4.19 and Sec 4.21 of the *DAG*. However, the aperture photometry measures the flux within a finite, relatively small aperture. The total flux however is distributed in a much larger area well outside the aperture. To account for this missing flux you need to apply a correction factor to the flux values. Such correction factors are determined through careful signal-to-noise measurements of bright celestial standards and are available as calibration files in HIPE. A there is a dedicated task to perform the aperture correction in HIPE.

4.14.1. photApertureCorrectionPointSource

Run the aperture correction task as in the following example:

```
result_apcor = photApertureCorrectionPointSource(apphot=myResult, band="blue",
 paralellObservation="False", scanSpeed=20\
calTree=calTree, responsivityVersion=7)
```

The `myResult` variable is the output of the aperture photometry task. The `band` parameter determines the filter. Because the aperture correction changes when there is an update in responsivity calibration, you must specify which responsivity calibration was used when your image was processed. You can check the responsivity calibration version by printing out your calibration tree (see Sec. Section 2.5.4) and looking for the entry *responsivity*. Currently there are two possible values for the version 5, 6 or 7. The aperture correction is slightly different for different scanspeeds and also might depend on if the observations carried out in parallel or prime mode. The user can choose the appropriate aperture correction by providing two optional parameters paralellObservation (True/False) and scanSpeed (10,20 or 60)

Note

Currently the task works only using the output of the annularSkyAperturePhotometry task

Note

The task overwrites the values in the myResult file so if one needs the fluxes without aperture correction it is advisable to save the before performing the task

Note

To those who want to do aperture photometry outside HIPE, the aperture correction values are available in the following documents: http://herschel.esac.esa.int/twiki/pub/Public/PacsCalibrationWeb/bolopsf_20.pdf and http://arxiv.org/abs/1309.6099ß

You can examine the task output in the same way as the output of the aperture photometry tasks (see e.g. Sec 4.21.2 of the *DAG*).