PACS Data Reduction Guide

issue dev, Version 2, Document Number:
March 2010

PACS Data Reduction Guide

Table of Contents

1. A First Quick Look at your Data 1
   - 1.1. Introduction 1
   - 1.2. Structure of this guide 2
   - 1.3. A quick-look at your data 2
     - 1.3.1. first: the ObservationContext (a.k.a. your observation) 3
     - 1.3.2. Then: look the Level 2 products 6
     - 1.3.3. And finally: inspect the data with GUIs 10
2. Introduction to PACS Data 13
   - 2.1. A PACS observation 13
   - 2.2. The data structure (simple version) 14
   - 2.3. The spectrometer pipeline steps 17
   - 2.4. The photometer pipeline steps 18
   - 2.5. The Levels 19
3. In the Beginning is the Pipeline. *Spectroscopy* 21
   - 3.1. Introduction 21
   - 3.2. Retrieving your ObservationContext and setting up 22
   - 3.3. Level 0 to 0.5 23
     - 3.3.1. Pipeline steps 23
     - 3.3.2. Inspecting the results 26
     - 3.3.3. Masks 34
   - 3.4. Level 0.5 to 1 38
     - 3.4.1. Pipeline steps 38
     - 3.4.2. Special issues 39
     - 3.4.3. Inspecting the results 43
   - 3.5. Level 1 to 2 51
     - 3.5.1. Pipeline steps 51
     - 3.5.2. Inspecting the results 53
   - 3.6. The End Of The Pipeline 55
4. Further topics. *Spectroscopy* 56
   - 4.1. Introduction 56
5. In the Beginning is the Pipeline. *Photometry* 57
   - 5.1. Introduction 57
   - 5.2. Retrieving your ObservationContext and setting up 58
     - 5.2.1. Scan map AOT 58
     - 5.2.2. Point Source AOT 60
   - 5.3. Level 0 to Level 0.5 61
     - 5.3.1. photFlagBadPixels 62
     - 5.3.2. photFlagSaturation 62
     - 5.3.3. photConvDigit2Volts 63
     - 5.3.4. addUtc 63
     - 5.3.5. photCorrectCrosstalk 63
     - 5.3.6. photMMTDeglitching and photWTMMLDeglitching 64
     - 5.3.7. convertChopper2Angle 64
     - 5.3.8. photAssignRaDec 65
     - 5.3.9. cleanPlateauFrames 65
   - 5.4. The AOT dependent pipelines 66
   - 5.5. Point Source AOR 66
     - 5.5.1. Level 0.5 to Level 1 66
     - 5.5.2. Level 1 to Level 2 70
   - 5.6. Scan Map AOR 72
     - 5.6.1. Level 0.5 to Level 1 72
     - 5.6.2. Level 1 to Level 2 73

Chapter 1. A First Quick Look at your Data

1.1. Introduction

This guide is still under construction, it is being updated as the HIPE and the PACS pipeline tasks are updated. You should always read the most recent possible version of this guide, with the most recent version of HIPE.

...

Welcome to the PACS data reduction guide #. We hope you have got some good data from PACS and want to get stuck in to working with them. In this guide we will (i) show you how to have a first quick look at your pipeline reduced data, (ii) explain how data are gathered by PACS and hence how they are structured, and summarise the pipeline steps, (iii) show you how to go through the pipeline yourself, (iv) show you how to inspect the products you produce as you proceed through the pipeline, and (v) discuss issues that are of concern for particular AOTs (such as rastering) or targets (such as moving targets) or which are still under development.

This guide is aimed at those who are new to HIPE and new to PACS. It will take a while to get used to HIPE and to reducing PACS data, so allow yourself a lot of patience. Satellite sub-mm data are complex because the detectors and the observing requirements are. If the data reduction seems difficult to you it is not because we have made it so, but because it is so. Our aim with this guide is to teach by doing: we will take you through the pipeline as a tutorial, so you can learn what to do and how to inspect what you have done. Along the way we will explain the what and the why of the data reduction. We recommend you run though the pipeline once following this guide, and then if you want to change some things you can run through it again on your own. This guide is designed to be read from *beginning to end*, so read the whole thing before you claim something is not working or not understandable.

HIPE is the Herschel Interactive data Processing Environment. HIPE is not just for running the pipeline, it provides an environment in which you can also analyse data using tools provided or by writing your own scripts. In HIPE you can write scripts to do any type of manipulation, mathematics, reformatting, analysis, or fitting on your data. Because it uses jython (python, java) it may be unfamiliar to many astronomers, but python and java both are languages that are well worth learning. Note that while python and java can be used within HIPE, the actual language is (H)DP, that is the (Herschel) Data Processing language. So some jython-ese will not work and there are additional capabilities that have been programmed into HIPE that are unique. Scattered throughout this guide are "seed" scripts, which were written primarily to accompany the pipeline to allow easy plotting and inspection of the data, but you can also use them to start to learn the scripting language yourself. There is also much help on DP scripting available from the help page, however, unless you are a good programmer, it is probably a good idea to first work your way through the pipeline and this PACS data reduction guide before starting to script. Doing it the other way around will guarantee much frustration.

There are a number of other documents you should read before and along with this one. This may sound boring, but you **really should**. While this PACS data reduction guide is meant to be complete, it is *not* stand-alone: we do not describe the elements of HIPE, the GUIs, and scripting, rather we direct you to the relevant documents.

The guides that introduce you to HIPE and to the post-pipeline capabilities of HIPE, and which are available from the Help page, are:

1) A guide to HIPE itself is the *HIPE Owners Guide* (*HOG*), and you should also see the *Quick Start Guide* (*QSG*). These tell you how to start up and work in HIPE and the easiest way to extract data from the Herschel Science Archive (HSA) and take a preliminary look at them.

2) The *Data Analysis Guide (DAG)* tells you about the tools that are provided in HIPE for you to do your data analysis (everything you do after your pipeline data reduction) and inspection of your spectra, cubes or images.

3) The *Scripting and Data Mining* guide *(SaDM*) contains a lot of information about working in HIPE with arrays, the DP syntax and working therein, doing mathematics, fitting, photometry and so on This is recommended to be read after you have worked your way through the first chapters of this PACS guide: here we give specific examples of working in HIPE with your data and using the DP language, and there you can go for the more general instructions.

4) The HIPE help page also has a search capability, in which you can type in the names of tasks or acronyms that are unfamiliar to you.

More advanced documents are:

1) PACS Advanced User Manual (*PAUM*). This discusses the details of the pipeline tasks (all the parameters and the algorithms). It has been written by and for internal users, so for the 3.0 release of HIPE may still be a little difficult to follow. You can treat it as a pipeline reference document.

2) The Product Description Document (*PDD*), a Herschel generic document, which describes the layout and terminology of the instrument-specific products. It is somewhat, but not overly, technical.

3) The HCSS or PACS User's Reference Manual (the PACS URM is the HCSS URM + extra PACS bits): these contain information about many of the tasks you will use, but be warned that these have been written by and for internal PACS users and hence may be rather difficult to understand.

4) The Developer Reference Manual (a.k.a. API), which gives you information about the java classes that underlie the DP system. This will be very difficult to understand at first if you are not a (java or python) programmer, but hopefully some of the examples provided in this guide will help you to understand them a little better.

1.2. Structure of this guide

In this first chapter we explain how to get your observations from the HSA and look at your Level 2 product, that is data which has already been pipeline-processed. In Chapter 2 we summarise the data reduction steps from Level 0 (minimally processed) to Level 2 (science quality), and explain a little about how the data are structured. Chapter 3 takes you through the pipelines for the various spectrometer AOTs, with some detail about what you are doing at each stage and presenting you with inspection recipes. In Chapter 4 issues of concern for particular AOTs or types of targets are discussed, and more information about the structure of PACS data are given. Chapters 5 and 6 are the same as 3 and 4 but for the photometer.

Note that chapters 4 and 6 are not yet ready.

1.3. A quick-look at your data

Your observations have been performed, now you probably want to know what they look like. This section will show you how to grab the fully pipeline-processed data and look at them. If you then want to run the pipeline yourself you will read Chap. 3 and onwards; but it is a good idea to first have a quick look at your data, to at least see what it is you have be given!

For spectroscopy these fully processed products are cubes, that is data with two spatial axes and one spectral axis (the PACS spectrometer is an integral field spectrograph). If you are not familiar with looking at cubes we suggest you read up a little on integral field spectroscopy before you start working on your PACS data. For photometry the fully-processed data are a stack of frames (images/maps).

Start up HIPE. If you followed the installation instructions this should be a matter of simply typing "hipe" on your command line or clicking on an icon. We recommend that you run HIPE with at least 2GB of memory, more if you can. To increase the memory allocation you can either change it on the

HIPE command line—but the allocation will go back to default next time you start HIPE—or you can edit one of the "hcss properties files" before starting HIPE. For instructions, see the *HOG*. If HIPE runs low on memory (it has a tracking bar to show memory use) it will freeze and you may have to kill your session, so don't stint on allocating memory.

When you start up HIPE first go into the Work Bench or the Full Work Bench perspective, by clicking on the green or blue clapperboard icons on the top right of the HIPE GUI. The *HOG* tells you what the various tabs in the Work Bench are for and how to customise your view. It is in the Console section of your work bench that you type commands.

1.3.1. first: the ObservationContext (a.k.a. your observation)

1.3.1.1. Some definitions

To start working you of course need to get hold of your data. You will either have been give it by someone, probably in the form tarball, or you will extract it from the HSA. After you have the entire dataset you need to read it into HIPE and at the same time extract from it the "ObservationContext". This ObservationContext contains your "observation". It can be thought of as a container of data products that belong to a specific observation (and *ObservationContext* is the HIPE class of this container#more on classes later). It provides the associations between all the products you need to process that single observation and also includes the results of the automatic pipeline reductions done by the HSA. The products contained in an ObservationContext include not just the actual astronomical observation (raw and reduced), but also the data products that were used to process the data in the automatic pipeline, such as: spacecraft pointing, time synchronisation data, the satellite orbit, the parameters you entered in HSPOT when you submitted the proposal, and the pipeline calibration tables. The reduced data contained in the ObservationContext are spectra and spectral cubes or images (spectrometer and photometer respectively); also provided should be a quality assessment of the observation/reductions.

We should also here explain here what a "pool" is. The easy explanation is that a pool is a collection of data that belong together—your HSA-obtained data, maybe all your observations of the same object, or all your Level 1 processed products, or everything you worked on in a single day. The commonality between the products in a pool is yours to decide upon. On disc a pool is simply a directory (and by default, HIPE expects these directories to be located off of [HOME]/.hcss/lstore). The data are held in these directories individually as FITS files, but organised in a very specific way. In HIPE you can read in and write out most data as FITS files or via pools, and for the less complex products either is OK to do. But more complex products, such as ObservationContexts, which contain many separate but associated datasets and products, then it is recommended to use a pool to hold the data on disc. This is in particular because if you want to query the data, a pool is by far easier to inspect that a single FITS file. A pool allows for the associations between these products to be made by the tasks that read, write and inspect them. It is the need for these links that is the reason why Herschel data are held in pools, and is also the reason why Herschel products can sometimes take a while to be extracted from or into a pool. Because the data in a pool are linked to each other, it is necessary to use the tasks we provide to inspect, query, and access them. You cannot simply read a single FITS file from a pool into HIPE and necessarily expect that you can do something with it.

A pool can hold any type of Herschel data product, not only the ObservationContext that you will start with in your data reduction experience. You can export products that you produce in the course of your data reduction into pools (more of that later). If you wish to share pools, to send someone processed data for example, tar up the whole directory and send them that. The pool's directory name must *not* be changed or HIPE will not be able to find the data therein.

1.3.1.2. Get the ObservationContext

So, you first get hold of your data and then you extract the ObservationContext from it. How you do this depends on how you have gotten your observation. We will explain the most common methods here, but you should also see the *QSG* and the first chapter of the *DAG*.

These are the most common methods; read them all as general explanations are scattered throughout.

- *From the HSA load your observation directly into the HIPE memory* via "Send to External Application" in the HSA view. Doing this, what you get is already the ObservationContext. To avoid having to extract the data from the HSA again the next time you work on it, save it now to a pool:

  ```
  saveObservation(myobs,poolName="swimming")
  ```

  where "myobs" is here the name of the ObservationContext (which can be anything, and with the HSA import method the product will in fact have some standard name, which you will see appear in your Variables panel); and here "swimming" is the directory on disc in which you wish to place these data. As said above, by default the host of this directory is [HOME]/.hcss/lstore/. If the directory does not already exist, it will be created.

  If you want to place the data in some disc location other than [HOME]/.hcss/lstore, you can use the saveObservation parameters to do this:

  ```
  saveObservation(myobs [,verbose=<boolean>] [,poolLocation=<string>]
     [,poolName=<string>] [,saveCalTree=<boolean>]
  ```

  where the optional parameters (those in []) are: `poolName`, a string and the name of the pool if you have given it your own, unique name; `poolLocation`, a string and is given in case the pool directory is not in [HOME]/.hcss/lstore; `verbose` (True or False, default is False, don't use "" when specifying this) for a full reporting; and `saveCalTree`, which allows you to save the calibration tree you have been using along with your ObservationContext (something you are unlikely to want to do at this point in time, but would be useful if you want to try to reproduce the pipeline processing done by someone else, as the calibration tree you use is crucial to the results you get: more later). So, if your pool is located at /Users/me/pools/obsid1342111 then you need to specify "/Users/me/pools" as the `poolLocation` and "obsid1342111" as the `poolName`, but if your data are in /Users/me/.hcss/lstore/obsid1342111 then you only need to specify "obsid1342111" as the `poolName`.

- *If you got your data via ftp from the HSA* then you need to import them into HIPE using the "import Herschel data into HIPE" view (accessed from the HIPE Window menu: see the *DAG* chap. 1 for instructions). This will extract the ObservationContext from the directory that you untared the data into and put it directly into a pre-existing, user-chosen pool. The reason you have to do this is because the directory in which the data are (once you have untared them) is not in the correct format for the pipeline tasks to be able to deal with. The data first have to be extract out of the HSA directory and put into a new directory/pool which will have a different structure.

  Once you have done that you can get the ObservationContext from that pool into HIPE in the following way:

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

  where the optional parameters are the same as for saveObservation, with the addition of "obsid". This is the observation identifier which you should know already, but if not you can hunt for your observation in the HSA and the obsid will be there listed. You may need to spectify the number as "1342182002L", that is, with an "L" at the end. The task run only with the obsid should find your ObservationContext, although you may find you need to specify also the `od` (observing day and `poolName` also. When you have executed this command, "myobs" will appear in the Variables panel listing, this now being name of your ObservationContext.

- *If your ObservationContext is in a pool already* (e.g. someone sent you an entire pool that they had already looked at) you get the ObservationContext using the same command getObservation explained above.

- *If you got your data from the HSA via the "Retrieve" button*: You use the same method for importing data into HIPE for that when getting data via ftp. *Note*: if you used the "Retrieve" button but only

on the Level 2 product, you may not at present be able to use the this interface to get the data (this is being fixed).

- *If you have not already gotten your data from the HSA* and put them in to a pool, and if you know the obsid and don't want to use the GUIs to access the HSA, then with the following command you can get your data and import them into HIPE:

```
myobs=getObservation(1342182002L, useHsa=True)
```

For this to work you must have your HSA username and password written in your [HOME]/.hcss/user.props file with the following lines:

```
hcss.ia.pal.pool.hsa.haio.login_usr = your username
hcss.ia.pal.pool.hsa.haio.login_pwd = your password
```

If you are only now writing these in that file, then you should restart HIPE for it to take effect. Alternatively you can type directly into your current session the commands:

```
login_usr = -"hcss.ia.pal.pool.hsa.haio.login_usr"
login_pwd = -"hcss.ia.pal.pool.hsa.haio.login_pwd"
Configuration.setProperty(login_usr,"xxxxxx")
Configuration.setProperty(login_pwd,"xxxxxx")
```

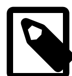
(Where xxx are your username and password.) We recommend you immediately then save myobs to a pool, because with getObservation you only load the ObservationContext into HIPE memory, not onto disc.

If you want to look at what observations are in a pool, use

```
allObs = LocalPool("swimming", -"/Users/me/.hcss/lstore").allObservations
```

and then double click on allObs in the Variables panel for a human-understandable listing.

These methods work for an ObservationContext, not for any other type of product. Importing and exporting other products will be explained later.



Note

You can give you variables#the things on the left of the = sign#any name you like. So instead of "myobs" used here, you could write "anobs" or "elmioobs".....

Be aware that when you enter od=number, that number must have no leading 0s. 0045 is not the same number as 45.

1.3.1.3. How can I work out what is in the ObservationContext?

One thing that will help you work out what your observation is of and what instrument configuration you had is to have a copy of the AOR (the Astronomer's Observation Request), which is where the commanding of the pointing and the instrument configuration would have been taken from. You can also look at the "meta data" of the ObservationContext. These are like FITS headers, a listing of various information about a product. Most products will have meta data. To see the meta data of your ObservationContext you can look at myobs with the Observation Viewer, getting this via a right-click menu on myobs in the Variables panel. This viewer opens in the Editor panel, and at the top are the meta data, as shown in the following screenshot:

ObservationContext for PACS data of observation 1342185578

Summary

Instrument:	PACS	**RA:**	283.39616
DEC:	33.02916666666667	**Operational Day:**	149
Observation ID:	1342185578	**Observation Mode:**	Scan map

Meta Data

name	value	unit	description
type	OBS		Product Type Identification
creator	SPG v1.2.0		Generator of this product
creationDate	2009-10-23T09:11:19Z		Creation date of this product
description	ObservationContext for PACS data of observation 1...		Name of this product
instrument	PACS		Instrument attached to this product
modelName	FLIGHT		Model name attached to this product
startDate	2009-10-10T07:15:48Z		Start Date
endDate	2009-10-10T07:42:12Z		End Date

Data

obs
auxiliary
calibration
level0

obs

Figure 1.1. Meta data

You can scroll down this list to see everything listed in there, which includes the parameters commanded in the AOR of your observation: pointing; repetition factors; observing mode; raster movements; band/wavelength.

There are other panels in the Observation viewer (see the *HOG* or the *DAG* chap. 2.3). The Data panel lists the products and datasets held in the ObservationContext (red means the product has been loaded into memory, red means it has not). To the right of that is an area where the default viewer of the products in this Data panel will open, if you select them (click on them). If you click on one of the products lssted in the Data panel (e.g. "+level2") the meta data now listed at the top are those associated with that particular product. There will be less meta data here than for the ObservationContext. To see the myobs meta data again, simply click on "myobs" at the top of the Data panel (which in the screenshot however is called "obs").

If you have multiple settings in your observation, for example rastering or dithering or cross scans, then unfortunately at present it is not easy to immediately work out what product is what part of your observation. This is something we are working hard to improve.

1.3.2. Then: look the Level 2 products

1.3.2.1. Spectroscopy

To look at what is in myobs, in your ObservationContext, again open the Observation Viewer on myobs:

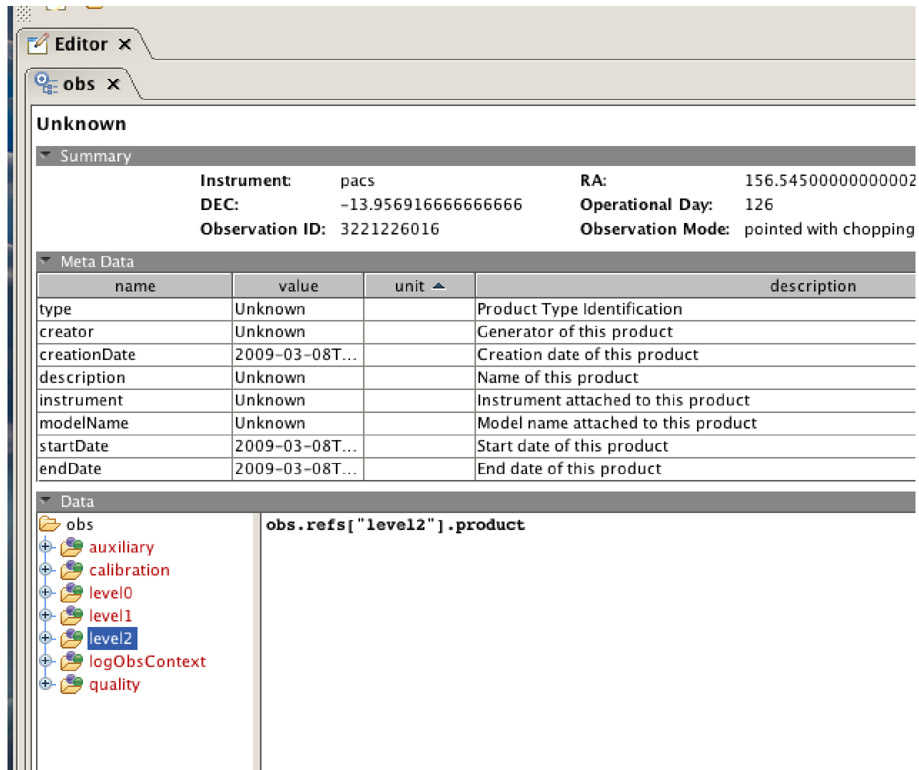


Figure 1.2. Your second glance at an ObservationContext with the Observation Viewer

The entries with + next to them can be thought of as directories of data. In each are products that correspond to the directory name, e.g. quality information are held in "quality". As here we are showing you how to look at a Level 2 (fully processed) product, you need to look at the "level2" entry. If there is no "level2" entry there, it means that your observation has not been processed through the automatic pipeline to that level, and hence there are no cubes (or maps) for you to look at. In that case you will need to reduce the data yourself through the pipeline. However, you should still read the rest of this chapter because it contains useful information that is not repeated in Chap. 3.

Click on the + next to it "level2" see what lies therein. You will see something like this:

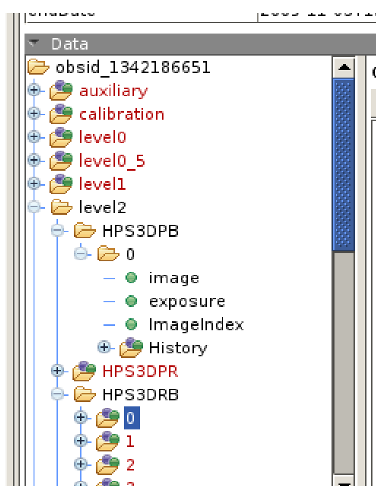


Figure 1.3. A further inspection of your ObservationContext

The screenshot shows you a listing of what is in the Level 2 of your ObservationContext. Listed there should be HPS3D[PB|PR|RB|RR] (or something similar: it changes faster than I can keep up!). The final "B" or "R" means "blue" or "red", and the "3D" indicates that it is a 3D (cube) product. The

difference between HPS3DPB and HPS3DRB is that they are Level 2 products produced by different pipeline tasks (more of that in Chap. 3).

If you move your mouse over the e.g. +HPS3DPB a banner will pop up indicating what type of product (what "class" of product) it is. It should say "ListContext", which means that this is a list of products (cubes), not a single product on its own. There could be anything from 1 to [a number > 1] products therein contained. If you click on the +HPS3DPB you will get a listing of all the products (the cubes) contained in this list, numbered 0..1..2 etc. In our screenshot the HPS3DPB has only one cube in it, the HPS3DRB has many. Hover over one of the numbers of the HPS3DPB and the banner should tell you that this is a *SpectralSimpleCube*; if you hover over the HPS3DRB you will be told that it is a *PacsRebinnedCube*.

Exactly what is in your Level 2 depends on what type of observation you requested. It is likely that you will have multiple cubes if your AOR included dithering/rastering/more than one spectral line.

You will need to read Chap. 3 to find out what the difference between the *SpectralSimpleCube* and *PacsRebinnedCube* is, but for now, suffice it to say that the SpectralSimpleCube is the final output of the pipeline, which takes the 5x5 simple cube as input and "projects" it into a cube of smaller-sized but more abundant spaxels (spatial pixels).. It also combines multiple rasters, if this was part of your AOR. You can chose to look at either or both of these cubes right now.

However, you should be aware these these cubes have been processed through a standard pipeline and you should consider them only to be of browse quality. For reasons that become clear as you read Chap. 3, it is strongly recommended that you reprocess the data yourself. The SpectrumSimpleCube especially is only of browse quality: the projection task that created this cube is still under development and several aspects of PACS data have yet to be accounted for in this projection.

In the screenshot above you can see that within the +0 "directory" are datasets called "image" etc. These are the datasets that make up the cube, these including the "image" (which contains the cube's flux values), "exposure" and "ImageIndex" (which contains the cube's wavelengths). You can also look at these (right-click on them to see what viewers will work) but that would be less than useful at this point in time.

1.3.2.2. Photometry

For photometry the same layout and similar syntax is found as for spectroscopy, and you should see something similar to the next screenshot. This includes products with the names HPP[N|M]MAP[B|R], where again a "B" or "R" as the final letter in the name stands for blue or red, and the difference between the "M" and "N" products is that a different mapping scheme was used. The HPPxxxx are, as before, *ListContext*s and the products therein are *SimpleImage*s. These HPPxxxx products contain multiple dataset within: the actual image, a noise map and a history reporting on what pipeline tasks and parameters were used during the processing.

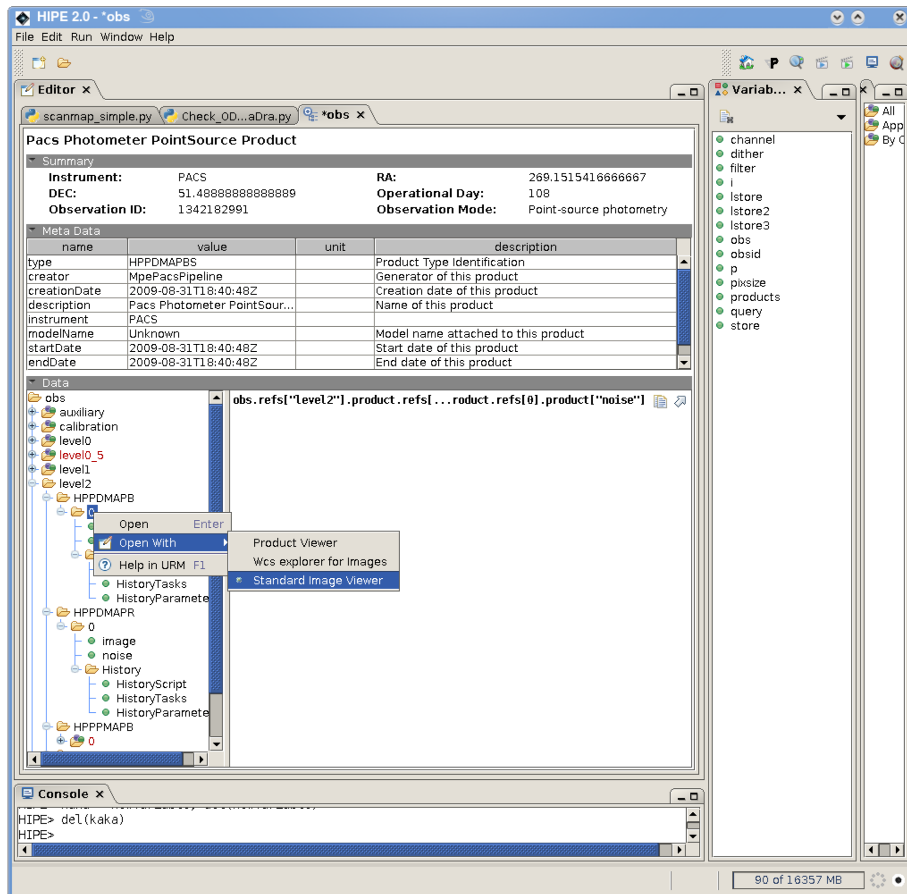


Figure 1.4. ObservationContext layout for photometry

In fact, whatever is (are) listed there in the Level 2 box is (are) what you want to look at, the differences between products there listed being the band and the type of map/cube that was made. More than one product will be listed, because more than one band and more than one type of map will be provided, and repetitions may be held separately and/or combined into one. In Chap. 2 and onwards we explain more about what these various products are.

1.3.2.3. Both

To now view your product(s) (the maps or cubes) you need to click on the +0 (or +1...) (*not* the datasets) next to the HPxxxx entry you are first interested in. This will give you access to the various viewers for your product. A double-click gives you the default viewer, a right-click gives you a viewer menu. The default viewer for spectroscopy is either the CubeAnalysisToolBox or the SpectrumExplorer, although these may only work on the SpectralSimpleCube, not the PacsRebinnedCube. The toolbox will open in the window to the right of the Data listing or in a new tab. For photometry the default viewer is the Standard Image Viewer (as shown in the previous screenshot).

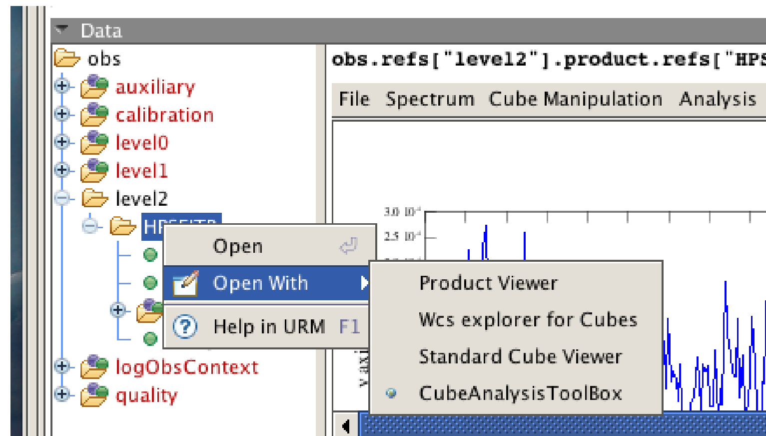
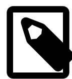


Figure 1.5. Viewing your Level 2 product

Note: as we are still working on the pipeline it is possible that the here-mentioned GUIs will not work on the data you have. Whatever viewers are offered for your product are the only viewers you can use.

For spectroscopy and photometry both you could also export the Level 2 product to FITS files and use a FITS viewer to look at them. To do this you need to extract the maps or cube out of the ObservationContext first. We postpone a full explanation of how to do this to Chap. 3/6 but in case you want to know right now: you can click-drag a +0 to the Variables panel, and that will selected out that particular cube/map product. When it appears in the Variable panel it will have a name like "newVariable". If you right-click on it there, you will be offered the opportunity to "Send to" FITS (remember to add the ".fits" to the name, and it is by default saved to the directory you started HIPE from). As you click-drag the product to the Variable panel you will see echoed to the Console the command that does this self-same thing (so you can do this yourself on the command line next time).

If you want to inspect separately the individual datasets, e.g. "image", then double click on them for their default viewer which will also open in a new tab (and which will not be the CubeAnalysisToolBox because these datasets are not cubes, they are the information that are held in the cube), or right click for a viewer listing. But at present viewing these datasets rather than the entire product will be less than useful for you.



Note

Data products are of different classes. The class types are indicated in this guide with *italics*, for example the Level 2 cube "mycube" should be a *SpectrumSimpleCube*. You can tell what class a product has either by hovering the mouse over it in the Variables panel to see the information banner; clicking on it in the Variables panel to see an information listing in the Outline panel,;or typing >print mycube.class in the Console panel. The class of a product defines what information are held in it and their organisation, and depends on what level of the pipeline the product has been taken to. Tasks, functions and GUIs are all written to work on specific classes of products, so if you cannot use a particular viewer, for example, it means the class of the product you are trying to use it on is incorrect.

1.3.3. And finally: inspect the data with GUIs

In this section we introduce you to the viewers that HIPE provides for you to look at your data. We assume that you want to only look at the data, and maybe have a play around with what is in them; the main emphasis of this Data Reduction Guide is the pipeline data reduction, which is the subject of all subsequent chapters.

The help page of HIPE—in particular the *DAG*—is the reference for data analysis tools.

For spectral cubes, what you will probably want to look at is the spatial distribution of your spectra, to find where your point source is or to make an emission line map. You will want to look at the spectra from individual spaxels, to access the quality of your data, and maybe add together spaxels to get a

spectrum of everything in your field of view (be it a point or an extended source). For photometry you will probably want to look at the maps of different scans, to see how well the map construction has been done, what the background looks like and whether the maps from different scans look the same.

1.3.3.1. Spectroscopy

Please note that the tools for doing spectral manipulation are still under development, and at the time of writing they do not all work directly on PACS cubes. Hence I warn you now that this part of working with PACS spectroscopy data will be rather frustrating. Some workarounds are provided in Chap. 4/6.

Here we will introduce you to the various GUIs that can be used to inspect your PACS Level 2 cubes. There are other ways you can inspect (and later manipulate) the data, but for a first quick-look we recommend you use the GUIs. These are called up with a right-click on the cube, be it within the Observation viewer in the Editor panel or in the listing in the Variables panel.

- To see scroll through wavelength slices of your cubes you can use the Standard Cube Viewer.
- # The SpectrumExplorer (SE). This is a spectral visualisation tool for sets of 1d spectra and the *SpectralSimpleCube*, but probably not right now on the *PacsRebinnedCube*. It allows for an inspection and comparison of spectra from individual spaxels. It is a very useful quick-look and quick-compare for the spectra from your cube, and highly recommended as a first look tool. The *DAG* provides a guide to the use of the SpectrumExplorer.
- # The CubeAnalysisToolBox (CAT). This allows you to inspect your cube spatially and spectrally at the same time. It also has analyses tasks#you can make line flux maps, position-velocity diagrams and maps, extract out spectral or spatial regions, and do some line fitting. The *DAG* includes a guide to this GUI. It currently works on the *SpectralSimpleCube*s and hopefully also the *PacsRebinnedCube.*
- # The SFTool and other mathematics tasks. The SpectrumFitterTool will allow you to fit and do mathematics on your spectra. To access the SFTool, click-highlight the *SpectrumSimpleCube* (the numbered entry, e.g. +0, not the "HPS3Dxxx" entry, see Fig. 1.3)\); go to the Task panel at the top-right of the Full Workbench; and double click on Applicable. All applicable tasks will be listed, this will include various mathematical tasks and the SFTool. The *DAG* explains the use of these tasks, which at present do not work on the *PacsRebinnedCube*.

If you want to look at the spectra in the *PacsRebinnedCube* you will have to use a command line method. (Hopefully the GUIs will at some point work on this product also.) You will first need to extract the cube it out of the ObservationContext, the easiest way being to drag the cube from the Data panel of the Observation viewer to drop it in the Variables panel. You need to drag the "+0" entry of your cube (see Fig. 1.3), not the "HPS3Dxxx" entry. When in the Variables panel you can (right-click) rename it. Here we call it "mycube". Then you can plot out spectra from the (5x5) spaxels with the command

```
#for SpectrumSimpleCube (included for completeness) to plot spaxel 12,11
PlotXY(mycube.getWave(),mycube.getFlux(12,11))
#for PacsRebinnedCube, to plot spaxel 2,2
PlotXY(mycube.wavegrid,mycube.flux[:,2,2])
```

In Chap. 3 we say more about using PlotXY with your PACS data.

1.3.3.2. Photometry

There are fewer separate GUIs for image viewing and analysis than there are for spectra, so there is less for you to learn about! There is one GUI which provides a first look and quick quality assessment of the data: the Standard Image Viewer (SIV). You call this either with a right-click on mymap in the Variables panel or the +0 entry of the ObservationContext, as explained before. If you want to do image analysis then HIPE provides many separate tasks you can run, to do contouring, overlaying, photometry, mathematics, etc. You access these tasks by click-highlighting mymap in the Variables panel, and then looking to see what "Applicable tasks" are listed in the Tasks panel of HIPE (one of

the "viewers" you can access from the main HIPE window menu). The instructions for using these tasks are in the *DAG*.

Chapter 2. Introduction to PACS Data

2.1. A PACS observation

If you are not familiar with how PACS observations work we recommend you read the Observer's Manual (<LINK>). PACS observations involve the synchronised movements of many parts of the instrument for the purpose of exploring the spatial and spectral space your AOR specified. During a PACS observation you can have: chopper movements between two mirror positions (to calculate the telescope background at each grating position, and as the grating moves along its range quite rapidly it gives rise to a need for rapid chopping); nodding of the telescope between two fields (to remove this calculated background); moving over many fields to make a bigger map; grating movements to sample the wavelength domain (for spectroscopy). All of these movements are tightly synchronised, so that at each field-of-view of each nod, the right (same) number of chops and right (same) wavelength range and sampling are included, and the nods are positioned and timed to fit in correctly with movements between consecutive (mapping) fields-of-view. The grating moves in discrete steps, usually down the wavelength range and back up again (and maybe more than once), during which the chopper will be chopping. Thus, moving along the time axis you are not just gathering more and more photons, but you will be looking at different sky positions, different wavelengths, and different focal plane positions. It is this instrument dance that the pipeline has to account for.

Spectroscopy:

The PACS spectrometer detectors are photo-conductors. When far-infrared photons fall onto the detector crystal, charge carriers are released that enable an electric current to flow through the detector. These currents are integrated over a capacitance. The more flux that falls onto the detector, the faster the voltage over this capacitance increases, and the larger the signal value will be. It is this voltage increase that is measured in the PACS detector electronics. The voltage over the capacitance is read out at 256Hz. Typically, the detector capacitance is discharged every 0.125 or 0.25 seconds and the voltage reset to a reference value; the detector is read out non-destructively (usually 32 or 64 times) before this discharge is performed. The non-destructive reading out is accumulative, that is the signal you read for readout at time T(2) is the value of the signal of readout at time T(1) plus the extra that is due to the light that fell on the detector since time T(1).

The raw PACS detector signals are ramps ("ramp"#"incline") of 32 or 64 increasing voltages. This information cannot be downlinked in its raw volume (which is huge), except for 1 pixel which is fully read out for data-checking purposes (by the PACS instrument team); therefore the instrument reduces the data on-board. For short ramps (32 samples) a slope fitting is done, and per pixel one number (the value of the slope) per integration ramp is downlinked and visible at Level 0. For long ramps (64 samples) the on-board software averages the voltages per 16 samples. In that case the Level 0 data consists of averaged ramps with four numbers per integration ramp.

The easiest way to check which of the two on-board reductions has been applied to your data is to check the Level 0 data (in the same way as explained in Chap. 1 for looking at what is in Level 2). If you see in the Level 0 listing product branches with the name HPSFITB or HPSFITR (Herschel-Pacs-Spectroscopy-FITted-Blue or Red) then on-board slope fitting was done, and you start the pipeline processing from these *Frames* "class" products. If you see products with the name HPSAVGB or HPSAVGR (Herschel-PacS-AVeraGed-Blue or Red detector) then the integration ramps were averaged on-board and you start the pipeline processing from these averaged *Ramps* class products. The dimensions of a HPSFITR product will be something like 18,25,980 (18x25 pixels, each with 980 readouts along the time dimension; later this time dimension is turned into the wavelength dimension). The dimensions of the equivalent HPSAVGR product will be 18,25,980,4 (each of the 980 individual ramps contain 4 averaged readout values).

The Level 2 products HPS3DRR and HPS3DPR stand for Herschel-Pacs-Spectroscopy-3Dimensional-Rebinned_cube -Red (which is of class *PacsRebinnedCube*), and Herschel-Pacs-Spectroscopy-3Dimensional-Simple_cube-Red (which is of class *SpectralSimpleCube*). At Level 1 we also have the HPS3D[B|R], these being of class *PacsCube*.

Your observation will contain data from your astronomical source, auxiliary data to allow the telescope pointing and timings to be calibrated, calibration data so the detector response and dark can be corrected, and more. In your astronomical dataset(s) there will be data not just from your target but1G also, probably in the beginning, a "calibration block", where the internal calibration sources are observed. Gradual changes to the response of instrument and degradations of the calibrators will be followed by the PACS team over the lifetime of Herschel, and will be included in the calibration data.

There is also a Status table, and later there will be a BlockTable, attached to your ramp and frame products, these contain information about the instrument status of the data and its organisation (in time). These are added to (and changed) as the pipeline proceeds. In Chap. 4 we explain the most useful entries of the Status and Block tables.

The PACS spectrometer detectors (one red and one blue) are of dimensions 18 along the Y and 25 along the X. Each of the 25 columns are a single spaxel (spatial pixel), and collectively these have an on-sky arrangement of 5x5. These columns are referred to as modules: a module is the physical entity to which the column corresponds to in the instrument. Each column contains 18 pixels (hence 18 rows), although the first and last hold no astronomical data (the first is an open channel, which has no associated detector unit, and the last is a dummy channel, being a resistor instead of a detector unit). The 16 active pixels between collect the spectral information for their spaxel, where each of the 16 pixels sees a wavelength range that is slightly shifted along compared to the previous. These 16 pixels are also known as "detectors"#confusing, yes, but the name comes from the fact that they are each little detectors of light.

Photometry:

The PACS photometer detectors are bolometer arrays. Each pixel of the array can be considered as a little cavity in which sits an absorbing grid. The incident infrared radiation is registered by each bolometer pixel by causing a tiny temperature difference, which is measured by a thermometer implanted on the grid. What we call "signal" is the voltage measured at this thermometer. The blue channel offers two filters, 60–85 µm and 85–130 µm and has a 32x64 pixel array. The red channel has a 130–210 µm filter has a 16x32 pixel array. Both channels cover a field-of-view of ~1.75'x3.5', with full beam-sampling in each band. The two short wavelength bands are selected by two filters via a filter wheel. The field-of-view is nearly filled by the square pixels, however the arrays are made of sub-arrays which have a gap of ~1 pixel in between. For the long wavelength end 2 matrices of 16x16 pixels are tiled together. For science observations the multiplexing readout samples each pixel at a rate of 40 Hz. Because of the large number of pixels, data compression is required and hence we do not see the raw data; they are binned to an effective 10 Hz sampling rate.

As with spectroscopy, the observations contain auxiliary data such as telescope pointing, time, and calibration information beside the target signal. Photometry observations also include nodding and chopping and calibration blocks.

2.2. The data structure (simple version)

The structure of PACS data are given in better detail in the *PAUM* and the *PDD*, but here we give a overview of everything you need to know for now.

Although the screenshots and the emphasis here is on spectroscopic data, the data structure is more or less the same for photometric data.

In Chap. 1 we included some screenshots showing listings of what is held in a PACS ObservationContext. A screenshot of the structure of your ObservationContext will look something like this:

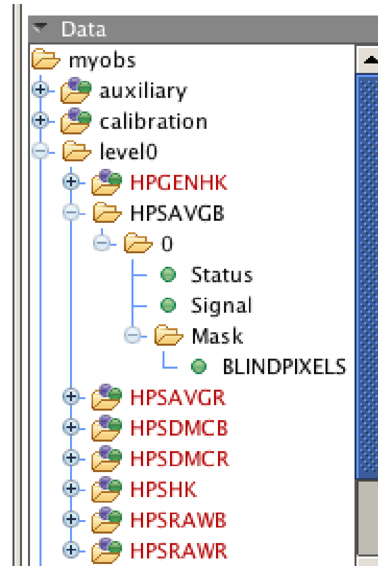


Figure 2.1. The contents of an ObservationContext for spectroscopy

This screenshot (and you could also look again at those of Chap. 1) shows that within an ObservationContext (called "myobs" here) you find layers of products with names such as level0, auxiliary, calibration...Within the level0/1/2 "directories" you can see products called HPSxxx (spectrometer) or HPPxxx (photometer): among these are the products that you will work on, as they contain the actual astronomical observations. The other directories (e.g. auxiliary and calibration) are extra information which are necessary for the data reduction but which you do not need to access directly yourself. The same click methods as previously mentioned can be used to inspect these products (i.e. double-click to view, right-click for viewing menu listing).

On the Console command line you can print-list these products, e.g.,

```
print myobs.calibration.spectrometer
print myobs
print myobs.level0
```

where *the first line* will produce a listing similar to the next screenshot, *the second line* produces a listing of what meta data are there plus the "directories" you can see in the screenshot above, and *the third line* shows what Level 0 products there are in your ObservationContext. Be warned, however, that this type of syntax will only take you so far: for example to "print" further something in Level 0 (e.g. HPSAVGB) you cannot type "print myobs.level0.HPSAVGB. We recommend, in any case, that you stick to the GUI listings rather than the command line.

In the HPSAVGB "directory" (for photometry this would be called HPPAVGB) in the screenshot above there is only 1 product (0), and in there are the datasets of Status, Signal, and a listing of Masks (in the beginning there will only be one mask listed). It may be that there is more than one HPSAVGB product present (referred to then as 0 1 2 3...), and if so you will later need to extract these out separately when you run through the pipeline. What has just been said applies equally to an HPSFITB/R "directory", which you will have if your data are the fit ramps instead of the averaged ramps products.

There may also HPSRAWB/R "directories", these products being the raw ramps that are downlinked for 1 pixel and used for calibration purposes (i.e. not by you). The organisation therein is different than the HPSFIT/AVG products and we do not explain them here.

All the other HPSxxxx entries in the screenshot above are additional products that contain data necessary for data reduction or data checking. Important for the pipeline are the products called HPSDMCR/B (or HPPDMCR/B), which are the DecMec data (more on this later). Not important for you are the HPS[HK|GENHK|ENG], which are "housekeeping" and engineering data, information

about the temperatures, instrument settings, status etc. of the satellite and of PACS. These information are for instrument scientists to interpret.

A calibration tree, containing all the information necessary to calibrate your observation, comes with your data and also with your HIPE installation (more on that later). If you click on "calibration" from the screenshot above you will see:

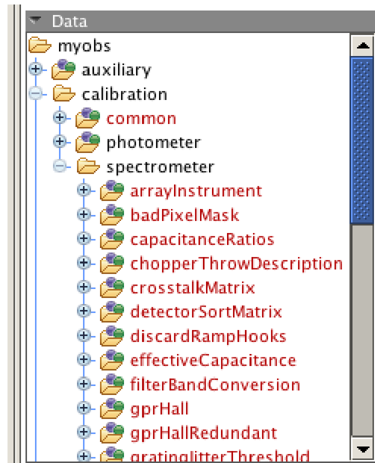


Figure 2.2. The contents of the calibration tree

These all are the calibration products that were used to produce the Level 0.5, 1 and 2 products that are all part of your ObservationContext.

The auxiliary tree, shown below, also contains products that are necessary for the reduction of your data, for example the obit ephemeris and pointing products. These are information that are mainly about the satellite.

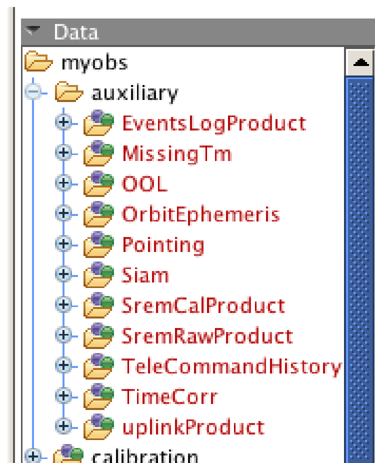


Figure 2.3. The contents of the auxiliary tree

The log and quality listings are: a log of the processing that produced that level's data (even for Level 0 there has been processing to convert the data from raw satellite format to an ObservationContext); and quality information.

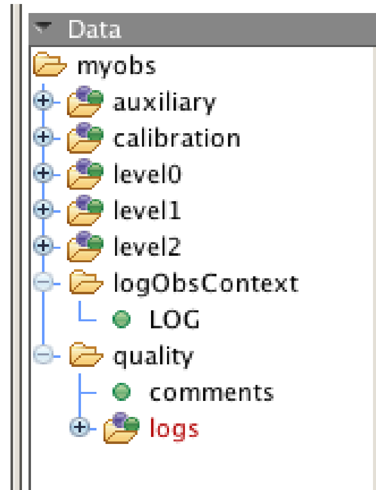


Figure 2.4. The contents of the log and quality trees

2.3. The spectrometer pipeline steps

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

1. If working on *Ramps* data, flag for saturation. Then fit the slopes to convert the data to a *Frames* product. If working on a *Frames* product skip to 2
2. Signal is converted from digits/readout_interval to Volts/s
3. Status entry for calibration blocks is added to; Status table is updated
4. Spacecraft time is converted to UTC
5. Spacecraft pointing is added to the Status table for the central pixel of the detector; chopper units are converted to sky angle; pointing is added to all pixels
6. Wavelengths for each pixel are calculated; Herschel's velocity is corrected for
7. Data "blocks" are recognised and the information organised in a table
8. Masking. Bad pixels will have already been masked. Masking for readouts taken during grating and chopper movements is performed, and for saturation if the data reduction began on a *Frames* product
9. Masking for glitches is performed
10. Signal non-linearities are corrected for
11. Signal is converted to a level that would be if the instrument had been set to the minimum capacitance (no change made if that was already the case)
12. The dark current and pixel responses (their individual sensitivities) are calculated using differential (internal) calibration source measurements to populate the absolute response arrays; a response drift is then calculated
13. Chop-nod AOT: the up- and down-chops are combined (i.e. a background+dark subtraction); the signal is divided by the relative spectral response function and then pixel responses (and their drift) are corrected for; such that each nod-cycle (not each nod) becomes one
14. Wavelength-switching AOT: *TBD*

15. Off-map AOT: *TBD*

16. Calibrated 5x5xlambda data cubes are generated

17. The cube's wavelength grid is created

18. Outliers are flagged (another glitch detection)

19. For chop-nod the nods A and B are combined

20. The data cube is spectrally resampled

21. The data cube is spatially rebinned, different pointings combined and resampled (mosaicked) or 3D drizzled (*not yet ready*)

The steps described here follow those in the "ipipe" pipeline scripts. Within the directory with your HIPE software, these are located in /scripts/pacs/toolboxes/spg/ipipe and can be accessed also from the HIPE toolbar. The name of the ipipe script corresponds to the AOT type. *Bear in mind that this data reduction guide is updated less frequently that the pipeline tasks, so if there are differences in the order of running tasks, use the order in the ipipe scripts.*

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

2.4. The photometer pipeline steps

We summarise here the basic steps of the PACS photometry data reduction. Level 0 to 0.5 is the same for all AOTs (steps 1 to 10).

1. Identify the structure of the observation and identify the main blocks (calibration and science blocks)

2. Perform data cosmetics: flag bad/saturated pixels and flag/correct cross talk and glitches

3. Convert signal from digits to volts

4. Correct for crosstalk *Currently on hold*

5. Deglitching

6. Spacecraft time is converted to UTC *Not yet ready*

7. Covert chopper position from engineering units into angle

8. Satellite pointing information are added to frames (sky coordinates of reference pixel for each readout)

9. The dark current and pixel responses (their individual sensitivities) are calculated using differential (internal) calibration source measurements to populate the absolute response arrays

10. Flag data taken while the chopper was moving

11. Point Source AOT: check what dithering pattern was implemented and update Status table; average signals taken at each and every chopper position, if more than one in each; add the pointing information; subtract the nod positions (per nod cycle and dither position); average the differential nod A and B images; do the flatfielding and response correction; combine dithers; make a map

12. Scan Map AOT: add the pointing information; remove data taken during slews; run the highpass filter; make a map

13. Small Extended source AOT: check what dithering pattern was implemented and update Status table; average signals taken at each and every chopper position, if more than one in each; add the pointing information; subtract the nod positions (per nod cycle and dither position); average the differential nod A and B images; do the flatfielding and response correction; another adding of pointing information; remove data taken during slews; make a map

The steps described here follow those in the "ipipe" pipeline scripts. Within the directory with the HIPE software, these are located in /scripts/pacs/toolboxes/spg/ipipe and can be accessed also from the HIPE toolbar. The name of the ipipe script corresponds to the AOT type. *Bear in mind that this data reduction guide is updated less frequently that the pipeline tasks, so if there are differences in the order of running tasks, use the order in the ipipe directory.*

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

2.5. The Levels

There is a Herschel-wide convention on the processing levels of its instruments. The different levels reflect how much of the pipeline has been run to create the data and the amount of additional information that has been attached to them.

- *Level 0 data:*

  Level 0 is a complete set of minimally processed data. After Level 0 data generation (done by the HSC) there is no connection to the database from which the raw data were extracted (this database is not available to the general user). Therefore the Level 0 data contain all the information required.

  - Science Data

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

  - Auxiliary data

    Auxiliary data for the time-span covered by the Level 0 data, such as the spacecraft pointing (attitude history, which however is only available after Level 0.5), the time correlation, selected spacecraft housekeeping, etc. The information are partly held as status entries attached to the basic science classes (*Ramp* and *Frame*) and the rest are available as separate products (e.g. the "pointing product") which you can access.

  - Calibration data

    This is the data that is used to calibrate the observations. A calibration dataset is included at Level 0, however calibration data is also provided with your HIPE installation, and generally it is the HIPE calibration dataset you should use when you process your data through the pipeline.

  - Quality data

    Quality control information, including (or maybe only) messages produced by the processes that produced the Level 0 data, or messages from the pipeline processing that produces later levels.

- *Level 0.5 data:*

  Processing until Level 0.5 is AOT independent. These data are also present with what you got from the HSA. At this level additional information has been added to the *Frames* science products (masks for saturation and bad pixels, RA and Dec, the BlockTable,...) and basic unit conversions have been

applied (digital values to volts, chopper position to sky angle). For the spectrometer, during Level 0.5 production the *Ramps* are turned in to *Frames*.

- *Level 1 data:*

  Level 1 data generation is AOT dependent (although there will be much overlap between the AOTs). Level 1 data are also available for selection from your pool, having been processed automatically at the HSA. Data processing at this level is concerned with cleaning and calibrating, and as the end the data are converted to a basic spectrometer cube (the 16x25 useful pixels have been converted to 5x5 spaxels, each holding 16 individual spectra).

- *Level 2 data:*

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

- *Level 3 data:*

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

Chapter 3. In the Beginning is the Pipeline. *Spectroscopy*

3.1. Introduction

The purpose of this chapter is to tutor users in running the PACS spectroscopy pipeline. Previously we showed you how to extract and look at the Level 2 fully 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 the sections here are divided into (i) a listing of the task steps with brief explanations and (ii) demonstrations for viewing the data just processed: plotting, displaying etc. More information on inspecting data, on the pipeline, and on particular issues with PACS data are in Chap. 4. In addition, the full glory of the pipeline tasks are outlined in the *PAUM*. However, we recommend you read through this chapter first, to learn at least how to run the pipeline and how to check the intermediate products. In this guide we are not necessarily telling you *what* to check—that is up to you, as the astronomer, to decide—we only tell you how to check the intermediate products that are produced as the pipeline processing proceeds.

We provide short scripts to inspect and select the data as you go through the pipeline. Between all the script snippets you will find more than one way of doing such selections, and examples of different syntax for doing similar things. In this way you can learn not only how to handle PACS data but also a little about the scripting that can be done in HIPE.

The PACS pipeline can be run in one of two ways: (i) The scripts in the ipipe directory can be run in one go. You can find these in your HIPE installation in /scripts/pacs/toolboxes/spg/ipipe, and the one you want corresponds to (or at least is similar to) the AOT name, e.g. pacschopnodstarframesIA.py for a pipeline for chop-nod observations. Starting from a Level 0 *Frames* product, or pacschopnodstarrampsIA.py if starting with a *Ramps* product. To do this you can load it into the Editor panel and run it (see the note below). (ii) You can run the pipeline as a long series of individual tasks, one by one. If you want to inspect intermediate products we recommend this method, which is what is followed here. (If you follow the ipipe scripts, copy it to your home directory first and work on that copy.) The ipipe scripts can be accessed also from the HIPE toolbar.

We will first take you through the pipeline for a chop-nod observation, then other AOTs will the be discussed; so if you are working with data from one of these other AOTs we recommend you still read this entire chapter. *At present only chop-nod is discussed*.

A suggestion before you begin: the pipeline runs as a series of commands, and as you gain experience you may want to add in extra tasks, construct your own plotting mini-scripts, write if loops and note down what it is you did to the data. Rather than running the tasks on the command line of the Console (and having to retype them the next time you reduce your data), we suggest you write your commands in a jython text file and run your tasks via this script.

Note

How to create and run a script in HIPE. From the HIPE menu and while in the Full/Work Bench perspective select File#New#Jython script. This will open a blank page in the Editor. You can write commands in here (remember at some point to save it...if HIPE has to be killed you will lose everything you have not saved). As you are doing so you will see at the top of the HIPE GUI some green arrows (run, run all, line-by-line). Pressing these will cause lines of your script to run. Pressing the big green arrow will execute the current line (indicated with a small dark blue arrow on the left-side frame of the script). If you highlight a block of text the green arrow will cause all the highlighted lines to run. The double green arrow runs the entire file. The red square can be used to (eventually) stop commands running. If a command in your script causes an error, the error message is reported in the Console (and probably also spewed out in the xterm, if you started HIPE from an xterm) and the guilty line is highlighted in red in your script. A full history of commands is found in History, available underneath Console for the Full Work Bench perspective.

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

Note

Syntax: *Ramps* and *Frames* are the "class" of a data product. "Ramp" or "frame" are what we use in this guide to refer to any particular *Ramps* or *Frames* product. A Frame is an image, for the photometer it is an image corresponding to 1/40s of integration time, for the spectrometer it is and image made up of the slopes of all detectors over one "ramp" (over one reset interval—see Chap. 2).

Please read this whole chapter before doing your reductions. Explanations for what you are doing are included in the sections that detail the pipeline tasks *and* the sections that detail how to inspect your data.

3.2. Retrieving your ObservationContext and setting up

How to retrieve the Observation Context from your pool was explained in Chap. 1. Continuing from there: since you are re-reducing the data you will want to this time start from Level 0 (if you want to start instead from Level 0.5 or 1, you follow these same instructions but you will start your pipeline reductions from this later level). You can selected either (i) *Ramps* or (ii) *Frames* products to work on, depending on which you have; these will be called (i) HPSAVGR, HPSAVGB or (ii) HPSFITR, HPSFITB. To do this, on the command line type:

```
myramp = myobs.level["level0"].refs["HPSAVGB"].product.refs[0].product
# or
myframe = myobs.level["level0"].refs["HPSFITB"].product.refs[0].product
# and extract a product necessary for one pipeline task
myrawramp = myobs.level["level0"].refs["HPSRAWB"].product.refs[0].product
```

where myobs is the ObservationContext from Chap. 1. This extracts out from Level 0 the first of the averaged blue ramps or the blue fit ramps. In addition you can get out the rawramps, this is something you *only* need for one masking pipeline task and we do not explain this product further in this guide; if you do not have this product, just skip this step. If you want to start with the red ramps, you replace the final B with an R. If there is only one product of HPSXXXX then you still need to specify the ".refs[0]", and if there is more than one you can select out the subsequent with ".refs[1]", ".refs[2]",...... To find out how many HPSAVGBs are present at Level 0, have a look again at Fig. 3 from Chap. 1; if you click on the + next to HPSAVGB it will list all (starting from 0) that are present.

An alternative way to get your HPSAVGB..ref[x] product is use the Observation viewer on myobs (right click on it in the Variables panel), go to the Data tab, click on +level0, then on +HPSAVGB to see the entries 0, 1, 2... You should be able to drag and drop whichever entry you want to the Variables panel (i.e. the 0 or 1 or... is what you drag and drop). The command that is echoed to the Console when you do this will be very similar to the one shown above, only now the new product is called "newProduct" (which name you can change via a right click on it in the Variables panel).

PACS data processing proceeds through various stages: Level 0 data have had almost nothing done to them and is where we begin here. Level 0.5 data processing is AOT-independent, the ramps are fit to turn a *Ramps* product in to a *Frames* one, and information is added to the data (telescope pointing is translated into RA, Dec and added in, bad data masks are set, etc.). The AOT-dependent part then continues to Level 1, from which level scientific-grade data is found. At Level 1 the wavelengths will have been calibrated, response of the detector corrected, chopping and nodding accounted for, etc. At Level 2 the data are turned in to a 5x5 cube, spatially and spectrally rebinned, and that marks the end of the pipeline.

Before beginning you will need to set up the calibration tree. You can either chose that which came with your data or that which is attached to your version of HIPE. The calibration tree contains the

information HIPE needs 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. As long as your HIPE is recent then the caltree that comes with it will be the most recent, and thus most correct, calibration tree. If you wish to recreate the pipeline processed products as done at the HSC you will need to use the calibration tree there used, i.e. that which comes with the data (and which is shown in Fig. 2 of Chap. 1). We recommend you use the calibration tree that comes with HIPE. Structurally, the two are the same, but the information may be different (more, or less, up-to-date).

```
# from your data
mycaltree=myobs.calibration
# or from HIPE recommended
mycaltree=getCalTree("FM")
# where FM stands for flight model and is anyway the default
```

It is necessary to extract a few other products in order for the pipeline processing steps to be carried out. These are the dmcHead, the pointing product, and the orbit ephemeris. You can get these with

```
pp=myobs.auxiliary.pointing
dmcB=myobs.level["level0"].refs["HPSDMCB"].product.refs[0].product
dmcR=myobs.level["level0"].refs["HPSDMCR"].product.refs[0].product
orbitephem = myobs.auxiliary.orbitEphemeris
timeCorr=myobs.auxiliary.timeCorrelation
```

(There should only be one dmcHeader for you to extract, i.e. just a "0" to specify in the "refs[]", not a 1 or 2... However, in vary rare cases it may be that there is more than one. If in your ObservationContext that is so—look at it with the Observation viewer in the way described in Chap. 1 to find out—then see Chap. 4.) If for some reason of these product does not exist, if you get an error when you run one of these commands above, then raise this as an issue with the helpdesk but you can still continue with your reductions: most of these products are not crucial to the data reduction, although not being able to use them will make your resulting spectra more noisy. The pointing product (pp), however, is necessary.

The pointing product is used to calculate the pointing of PACS during your observation, the dmcB/R, or the products called HPSDMCB and HPSDMCR, contain the position and status of the PACS mechanisms and detectors sampled at high frequency. The orbit ephemeris is used to correct for the movements of Herschel during your observation, and the time correlation product is used by the time conversion tasks. If the time correlation and orbit ephemeris products are not present, don't worry, you can run the pipeline for now without them.

3.3. Level 0 to 0.5

First we list the pipeline steps, then we tell you how to inspect the products just created.

Tip

As you run tasks in HIPE you will see a small rotating circle at the bottom right of the HIPE GUI indicating that "processing is occurring". While this is running you cannot execute other commands.

HIPE task names, and most other things you will type in HIPE while reducing your data, are case sensitive.

If you want to stop a task running with the red stop button, you can only do that if you ran the task from a script in the Editor panel, not if you ran it from the Console command line.

3.3.1. Pipeline steps

If you start with the blue product called HPSAVGB you begin with:

```
myramp = specFlagSaturationRamps(myramp, calTree=mycaltree)
myramp=activateMasks(myramp,String1d([" -"]), exclusive=True)
myframe = fitRamps(myramp)
```

If you start with the HPSFITB product, you skip these two tasks (the flagging is done later). The task activateMasks we explain at the end of this section. The task specFlagSaturationRamps flags the data for saturation, creating a mask called SATURATION which subsequent tasks can take into account (or not). Later we show you how to inspect this mask. The task uses a calibration file held in the caltree to work out where saturation has occurred, this all being based on ground-based and Performance Verification observations. (The task masking for saturation on frames, used later, is based on the same data.) In this task you can include the "rawramp" product if you have extract that out of your ObservationContext (as explained above), but if you don't have this product you can just leave it out of the task call. See below for more information on this task. "myramp" and "myframe" are the names of the products you are creating and working on (you can, of course, give them any name you like). fitRamps is a task that fits the ramps with a 1st order polynomial (the details of which have been determined by the PACS team) and returns the slopes values in units of digits/readout_interval. It changes the dimensions of the data, so

```
print myramp.dimensions
print myframe.dimensions
```

will return something like: 18,25,980,4 and 18,25,980, respectively: 980 individual ramps, each of which has 4 readout values, have been converted to 980 new readouts, the value of each being that of the slope of the polynomial fit to the 4 original readouts.

fitRamps does not take into account any masks, rather it propagates them. So if in pixel 0,0, for the 545th ramp the 4th readout is saturated, the whole ramp, including the saturated readout, will be fit but for pixel 0,0 the 545th slope value in myframe will carry the saturation flag=True.

To take care of solar system objects you will begin with:

```
if (myobs.meta.containsKey("naifid"):
   if (obs.meta["naifid"].value == 0):
      isSso = False
   else:
      isSso = True
else:
   isSso = False
```

This "isSso" variable is called on later in the pipeline. The scripting syntax here will become clear to you as you work your way through this guide, note that spacing is important, so have no spaces at the ends of lines and keep to the indentation!

You now continue with the following:

```
# if you are beginning from a Frames product
myframe = specFlagSaturationFrames(myframe,rawRamp=myrawramp,
   calTree=mycaltree)
# and then
myframe = specConvDigit2VoltsPerSecFrames(myframe, calTree=mycaltree)
myframe = detectCalibrationBlock(myframe)
myframe = specExtendStatus(myframe, calTree=mycaltree)
myframe = addUtc(myframe, timeCorr)
myframe = specAddInstantPointing(myframe, pp, orbitEphem=orbitephem,
  isSso=isSso, calTree=mycaltree)
myframe = convertChopper2Angle(myframe, calTree=mycaltree)
myframe = specAssignRaDec(myframe, calTree=mycaltree)
myframe = waveCalc(myframe, calTree=mycaltree)
myframe = specCorrectHerschelVelocity(myframe, orbitephem, pp)
myframe = findBlocks(myframe, calTree=mycaltree)
myframe = specFlagBadPixelsFrames(myframe, calTree=mycaltree)
myframe = flagChopMoveFrames(myframe, dmcHead=dmcB, calTree=mycaltree)
myframe = flagGratMoveFrames(myframe, dmcHead=dmcB, calTree=mycaltree)
```

Note: the line wrap around for the first command is just there so this line fits on the pdf printed page. Typing this on the command line or in a script, you would not wrap-around.

And to explain this all:

- In the order listed these tasks do the following: flag for saturation if starting from a `Frames` product (and this task was explained previously for the `Ramps` product); convert the units to V/s; add to the Status table information about the calibration sources; update the Status table with chopper and scanning position information; if timeCorr is present, then convert from spacecraft time to UTC; add the pointing and position angle of the central detector pixel; convert the chopper positions to sky positions; calculate the pointing for every pixel (which is not just a set offset from the central pixel, but depends on the chopper position seen by each pixel); calculate the wavelengths; if the orbit ephemeris and pointing products are present, correct the wavelengths for Herschel's velocity; organise the data into blocks (per line observed, per raster position, per nod....); flag for bad pixels and if the dmcHeader product is present, flag for a moving chopper and grating.

- If a parameter is specified as "calTree=mycalTree" then it can be anywhere in the call, but if you specify only the parameter value (i.e. just "calTree") then it has to be in the right place in the call.

- specCorrectHerschelVelocity is a task you can elect not to run, since at this point in time its results are not used later in the pipeline. It corrects for the velocity of Herschel and adds the correction to the Status (VSC). It will be changed so that it can also be run to only calculate the correction and not apply it.

- The reason for flagging data taken while parts of the instrument were moving is that data is taken continuously, it does not stop for chopping, grating movements, nodding, or even rastering. To do so would be time-inefficient. The masking tasks use automatic criteria, mostly taken from the calibration tree, to determine if a readout needs to be flagged as potentially bad. For example, detector readouts taken while the grating is moving are flagged in flagGratMoveFrames. In Chap. 4 we discuss how to modify and add to these masks, and later here we explain how to look at them to check that the masking was done correctly. We do recommend that you accept the default masking. The masks that were created in this part of the pipeline are SATURATION, UNCLEANCHOP, GRATMOVE. Created by the task specFlagBadPixelFrames are the masks BADPIXELS and NOISYPIXELS, the former being truely bad pixels and the latter an indication that elevated noise was found from instrument tests done in-orbit. If you run fitRamps you will also then have a BADFITPIX mask, which is a quality indicator for pixels for which the averaged ramps are suspected to have not been well fitted during fitRamps. If you began work on fit ramps, however (i.e. HPSFIT[B|R]), this mask will not be present. The SATURATION mask is created where your AVG or FIT ramps exceed a certain signal limit. We should point out that because the AVG or FIT products that you are reducing are not the raw data, it is not 100% certain that the readouts that exceed the pre-determined saturation limit are saturated, it is only very likely. Later we show you how to look at these data to see if they probably really are saturated. If this task is called with as input the "rawramp" product, then it will also create a mask called RAWSATURATION, where this mask is based on saturation having been detected in raw data and thus it is certain they really are saturated. However, because the raw data are huge, we can only downlink from Herschel the raw data for a single pixel, and for that we have chosen the most responsive pixel of the detector (5,12 for the blue 10,12 for the red). When saturation is detected in this pixel then for all pixels of the detector for those same readouts (those same time-line datapoints), the mask is filled with 1, i.e. True. At the same time a Status column called RAWSAT will be filled. However, just because the most responsive pixel is saturated doesn't mean that all are (since the other pixels are less responsive and hence their signal level will be lower). Thus this mask should be considered a warning mask. Later we will show you how to look at the readouts for all pixels that have been flagged with this mask. Finally, the raw data may have a BLINDPIXEL mask from the beginning, this simply being a badpixel that is masked already at Level 0.

- The pipelines tasks that will be described in the next section will take into account the masks you have created here, each task having its own set of default masks it considers (those believed necessary).

- Note that the tasks that use the dmcHead as a parameter may well run without specifying the dmcHead, *but* the results will be wrong. Here, as we have elected to reduce the blue data, the dmcHead we are using in dmcB[lue], as was extracted earlier in this chapter. When working with the red data, naturally you should use the dmcR extracted product. Make sure you use the right one, as the task does not necessarily know if you give it the wrong one.

- The Status table contains information about the status of Herschel and PACS during the observation, and is added to as the data processing proceeds. You can look at these information (as a dataset or plot) by double clicking on myframe from the Variables panel, then in the Editor tab this appears in select the "Status" dataset. Selecting with a right click presents a menu for viewing, including as: a plot, which however does not show all the parameters; and a dataset, i.e. a table. We tell you more about this in Chap. 4.

- Ideally the tasks do not change the input frame unless you give the output frame the same name as the input frame; if you gave the output a different name to the input, the input should be preserved in the pre-task state, i.e., the syntax "myframe=waveCalc(myframe)" should add to myframe, whereas "myframe_wave=waveCalc(myframe)" should create a new product called myframe_wave and not change myframe. However, some tasks unfortunately do not do that. Therefore we recommend, for now, that if you want to preserve the pre-task state of a frame, you first copy it and then run the task. So,

  ```
  # first do this:
  myframe_prewave=myframe.copy()
  # then
  myframe = waveCalc(myframe, calTree=mycaltree)
  # and now myframe_prewave and myframe are distinct products
  ```

  It is really not necessary to make a new product for every pipeline task you run (certainly not for waveCalc), only perhaps for the tasks that actually alter the state of the data. Of the steps so far described, none really qualify as that. PS do not forget the .copy() part of the syntax!

Now, what is this activateMasks task? Pipeline tasks can produce masks and/or they can propagate masks. Tasks that create masks also by default activate them. Once activated, a mask remains so until deactivated. It turns out that some tasks run better if only certain masks of the input frame are "active" and others are "inactive". Hence it is necessary to specify which masks should be active and which should be inactive before running one of these sensitive tasks; that is what activateMasks does. Chap. 4 explains in more detail the activateMasks task.

This marks the end of Level 0.5, up to which the data reduction is AOT independent. Next we will tell you how to save and inspect the products you have just created.

3.3.2. Inspecting the results

What are you likely to what to check of your frame as you work through the pipeline to the end of Level 0.5? One obvious thing is to check the effect the reduction tasks have had on your spectra by looking at befores and afters. You could also look at the pointing, the masking, and the relationship between the movements of the chopper, grating, and nodding and how they modulate your signal. Looking at these details will allow you to understand better what goes into producing your final cubes. We will introduce you to the Status; tell you how to look at the chopper and grating; show you how to (over)plot the spectral signal; the pointing; and show you how to inspect masks. We leave it up to you to mix and match the different sets of instructions. As you work your way through the instructions you will learn how to extract particular datasets out of your *Frames* product, out of myframes. The same approach will be taken when checking the intermediate products of the later stages of the pipeline.

First, to know the dimensions of your data, use:

```
print myramp.dimensions
# giving us something like: array([18, 25, 672, 4], int)
print myframe.dimensions
# giving us something like: array([18, 25, 672], int)
```

The first 3 dimensions of myframe will be the same as those of myramp (the 1st and 2nd are spatial axes, the 3rd is the time-line and later spectral axis): the "4" in the 4th dimension of myramp are the 4 averaged readouts per ramp, and as these are fit when creating myframe, that dimension has disappeared.

There are two approaches to looking at what is in your ramps and frames: use one of the viewer applications, or plot out bits of the data.

Tip

Note that when you look at images of the PACS detector you will see that the Y length is 18 and the X length is 25. However, C, python and java expect references to (row, column) which here is (Y,X), and this is why the lengths are actually always listed or referred to as 18,25.

3.3.2.1. The Status - what was PACS doing during your observation?

The Status is a attached to your *Frames* or *Ramps* product and holds information about the instrument status, where the different parts of PACS were pointing and what it was doing, all listed in time order. To view Status information for your observation you can click-to-send your frame or a ramp to the Editor panel, locate the Status therein, and right-click on it to view using the Dataset Viewer, TablePlotter, or OverPlotter (screenshot below). The entries in the Status table are of mixed type#integer, double, boolean, and string. The ones that are plain numbers can be viewed with the Table/OverPlotter, the others you need to look at with the Dataset Viewer. You cannot currently overplot easily entries that have very different ranges. In Chap. 4 we explain more about the Status and what parts of the Status table you are likely to want to look at and how you can plot the entries that are not numbers.

For a *Frames* product the Status column entries are single values per time stamp (per reset index) and for a *Ramps* product some entries will be an array of values. The Status is added to as the pipeline processing proceeds. The Status table of myframe contains the same as, and more columns than, myramp and is more useful to look at (the frame has had more tasks run on it). Of particular interest to you at this point in time will be the chopper movements (CPR) and the grating movements (GPR), and maybe also how the signal modulates with these. To remind you what the chopper and grating do: (i) The chopper moves between a position that is pointing at your target and a position that is pointing at blank sky. The blank-sky data will be subtracted from the on-target data in order to remove effect of the rapidly varying telescope background and also remove the dark current. This chopping happens with a very high frequency. You may want to check that the signal really is lower in the blank-sky position than the on-target position (although bear in mind that with the short integration times that PACS operates at, the difference in signal between the chopper positions will not be huge). (ii) The grating moves with a certain speed and step size in order to sample the wavelength range at the dispersion you have requested, and does this usually at least twice (once down in wavelength and once up in wavelength). You may also want to look at how the signal changes with grating position.

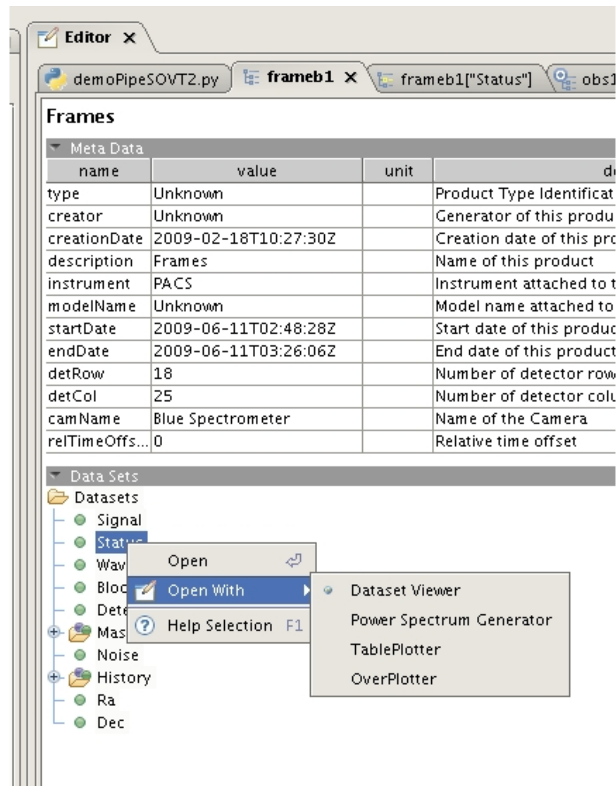


Figure 3.1. Viewers for the Status

You cannot inspect the "Signal" product of frame/ramp with the Table/OverPlotter, nor can you overplot to see two Status entries whose axes ranges do not overlap. For these cases we can recommend PlotXY.

3.3.2.2. Plotting the spectrum to understand what you have: 1

We cannot predict everything you will want to look at for your data so we provide examples of the most likely possibilities, and you can bootstrap from those to plot other things. Here we show you how to plot the signal (v.s. time or wavelength).

If you just want to plot the signal of frame, in the (time#array) order it is held:

```
p=PlotXY(myframe.getSignal(8,12), titleText="your title", line=0)
p.xaxis.title.text="Readouts"
p.yaxis.title.text="Signal [Volt/s]"
```

(The titles are not necessary.) To do the same for myramp you need to add RESHAPE() to the command:

```
PlotXY(RESHAPE(myramp.getSignal(8,12)))
```

Why the RESHAPE? The dimensions of a *Frames* product is 18,25,z where z is the number of slopes present. When you plot pixel 8,12,[all z] you are plotting a 1D array. For our averaged *Ramps* product, however, the dimensions are 18,25,z,4, and selecting out pixel 8,12 will give you a 2D array to plot; PlotXY does not like this, so you need to reshape the data.

It is not necessary to specify >p=PlotXY(), you could just type >PlotXY(), but with the first you can add more things to the plot (more data, annotations...).

To plot a spectrum, that is signal versus wavelength, after you have run the waveCalc task,

```
p = PlotXY(myframe.getWave(8,12),myframe.getSignal(8,12),
```

```
   titleText="title",line=0)
p.xaxis.title.text="Wavelength [$\mu$m]"
p.yaxis.title.text="Signal [Jy]"
```

Now, depending on what type of observation you are looking at (e.g. SED vs. line scan) and at what pipeline stage you are looking at your plotted spectrum, it is possible that you will see something that does not look quite like right. When you plot using the command above, you are plotting everything that is in your dataset. This can include: data from the calibration sources (take at the key wavelengths only); multiple spectra if your observation includes more than one field-of-view (for rastered/dithered observations); data taken while the telescope was slewing; data from the two chop positions and from the two nod positions (chops and nods are not combined until the next stage of the pipeline). In addition, if you have several grating runs (if you sampled the wavelength domain more than once), then each spectrum will be multiple and it is possible that the spectra from multiple grating runs will not be exactly at the same "counts" levels. So, if you have a line scan and you see this:

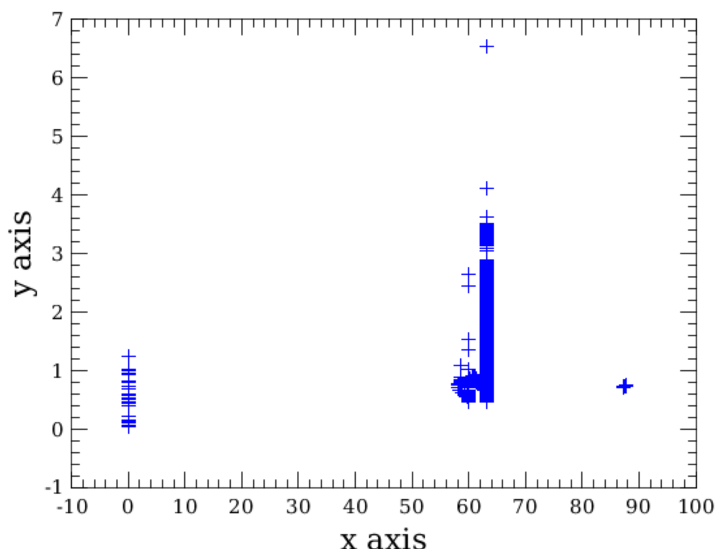


Figure 3.2. Level 0.5 line scan spectrum: entire dataset

try to zoom in on the wavelength you requested in your AOR, when you should see this:

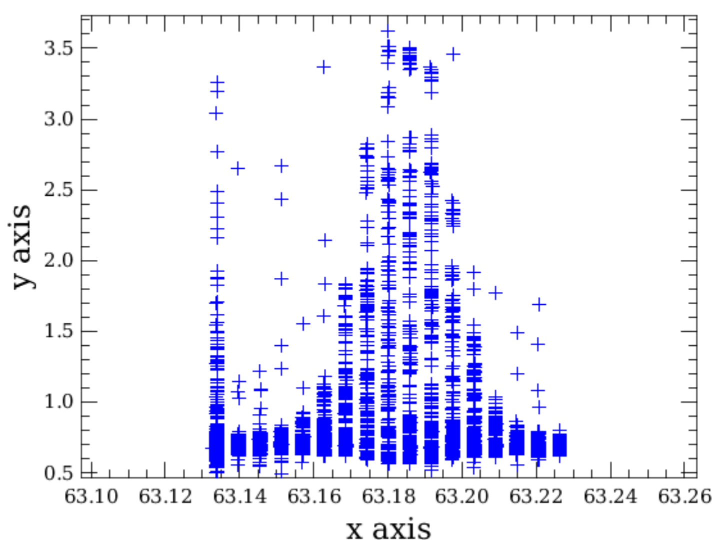


Figure 3.3. Level 0.5 line scan spectrum: zoom

In this spectrum (of a single pixel) the spectral line is "filled in", which is not what one would expect. However, bear in mind that these data have not yet been corrected for the nodding and chopping (and may include multiple rasters). Hence this spectrum is that of at least 5 different pointings. In the next section we will show you what this spectrum becomes when further corrected.

We have already pointed out that each of the 16 active pixels that feed into each spaxel (spatial pixel, a.k.a. module) sample a wavelength range that is slightly shifted with respect to the next pixel. Hence if you overplot several pixels of a single module (e.g. 1 8 and 16 of module 12) in different colours you will see this:

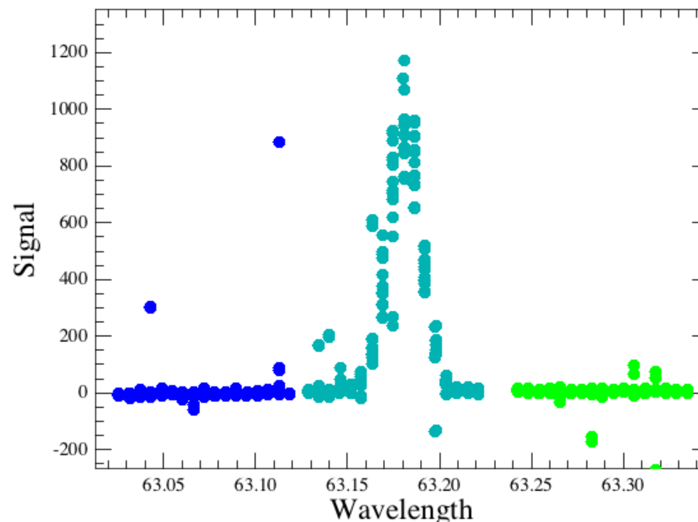


Figure 3.4. 3 pixels of a single module

where the dark blue is pixel 1,12; light blue is 8,12; green is 16;12. Hence, if you just plotted pixel (16,12) and saw no spectral line, this may be the reason why. If you want to the spectra from all pixels of a module at once (in one colour), then in the PlotXY command you will need to use the RESHAPE command (because otherwise you are asking PlotXY to plot out a 3d product). The following will plot all the active pixels (1 to 16 inclusive) of module 12 (the central spaxel of the field of view):

```
p = PlotXY(RESHAPE(myframe.wave[1:17,12.:]),RESHAPE(myframe.signal[1:17,12,:]),
   titleText="title",line=0)
p.xaxis.title.text="Wavelength [$\mu$m]"
p.yaxis.title.text="Signal [Jy]"
```

Consider also that the dispersion is also important in determining what you see when you plot a single pixel. If your dispersion is low, e.g. you have a fast SED AOT, then it is possible that a spectral line as viewed in a single pixel will "fall" a bit between the gaps in the dispersion; you will need to plot all the pixels of the module to see the fully sampled spectrum.

3.3.2.3. Plotting the spectrum to understand what you have: 2

You can next check the movement of the instrument during your observation, and maybe look to see how the signal varies with these movements. You will also use what is explained here to overplot the signal with the masks.

The following is an example of how to plot, with full annotation, the Status parameter CPR (chopper position), GPR (grating position) and signal together for a *Frames* product:

```
# first create the plot as a variable (p), so it can next be added to
p = PlotXY(titleText="a title")
```

```
# (you will see P appear in the Variables panel)
# add the first layer, that of the status CPR
l1 = LayerXY(myframe.getStatus("CPR"), line=1)
l1.setName("Chopper position")
l1.setYrange([MIN(myframe.getStatus("CPR")), MAX(myframe.getStatus("CPR"))])
l1.setYtitle("Chopper position")
p.addLayer(l1)
# now add a new layer
l2 = LayerXY(myframe.getStatus("GPR"), line=0)
l2.setName("Grating position")
l2.setYrange([MIN(myframe.getStatus("GPR")), MAX(myframe.getStatus("GPR"))])
l2.setYtitle("Grating position")
p.addLayer(l2)
# and now the signal for pixel 8,12 and all (:) time-line points
l3 = LayerXY(myframe.getSignal(8,12), line=2)
l3.setName("Signal")
l3.setYrange([MIN(myframe.getSignal(8,12)), MAX(myframe.getSignal(8,12))])
l3.setYtitle("Signal")
p.addLayer(l3)
# x-title and legend
p.xaxis.title.text="Readouts"
p.getLegend().setVisible(True)
```

The Y-range is by default the max to the min, so you would not need to specify those if you wanted to plot from max to min. The "l1." commands create new layers which are then added to the plot with p.addLayer().

As before, if you want to plot for myramp rather than myframe, then around every myramp.getStatus() or myramp.getSignal() command you will need to write RESHAPE(), for example:

```
sey = RESHAPE(myramp.getSignal(8,12))
l3 = LayerXY(sey, line=2)
l3.setYrange([MIN(sey), MAX(sey)])
```

(This also shows you an alternative way of specifying things to plot.)

If you fiddle with the plot Properties and/or zoom in tightly the plot you just made should look something like this:

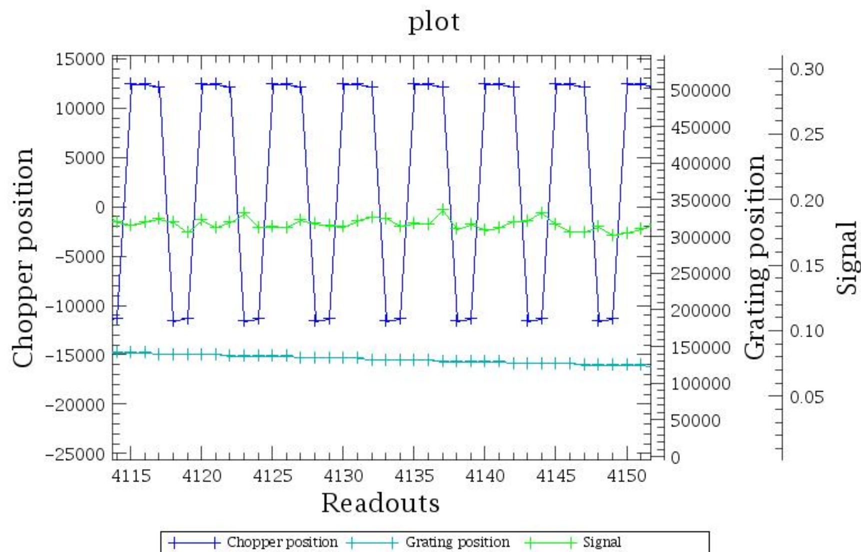


Figure 3.5. A zoom in on PlotXY of the grating and chopper movements for frame

What the figure above shows is that the chopper is moving up and down, with 2 readings taken at chopper minus and 3 readings taken at chopper plus. The grating is moving gradually, and its moves

take place so that 2 minus and 3 plus chopper readings are all made at each grating position (there are 5 grating points for 5 chopper points). The signal can be seen modulating with chopper position, as it should be because one chopper position is on target and the other on blank sky.

Using the same set of instructions you can chose yourself what to plot against what, from the Status table and with the signal. You may, for example, wish to check whether the readouts (signals) taken during a moving chopper have been flagged correctly in the UNCLEANCHOP mask. This is covered below.

3.3.2.4. Plotting the pointing

Since you have run the tasks to calculate the pointing, you can plot the RA Dec movements of the central pixel (i.e. where was Herschel/PACS pointing?):

```
p = PlotXY(myframe.getStatus("RaArray"),myframe.getStatus("DecArray"),
           line=0,titleText="text")
p.xaxis.title.text="RA [degrees]"
p.yaxis.title.text="Dec [degrees]"
```

where you will get something that shows the entire track of PACS while your calibration and astronomical data were being taken:

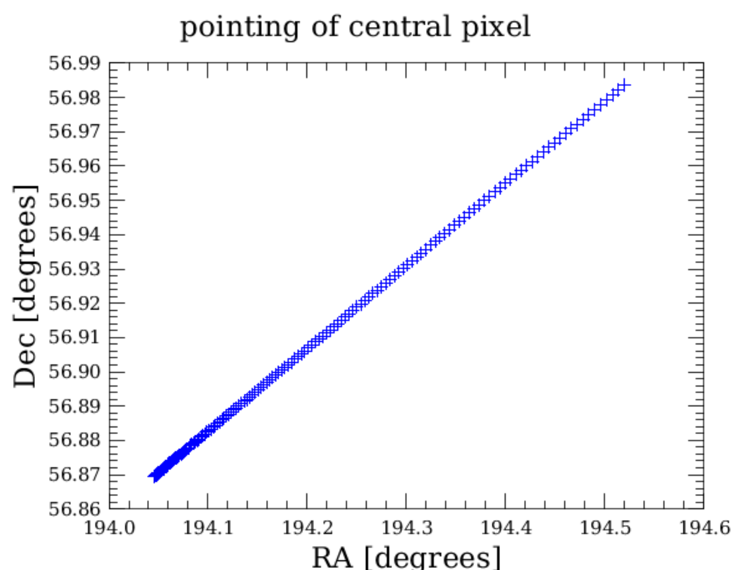


Figure 3.6. Movement of PACS during an observation

To plot all the spaxels' sky positions together with the source position for the last datapoint of myframe:

```
pixRa=RESHAPE(myframe.ra[:,:,-1])
pixDec=RESHAPE(myframe.dec[:,:,-1])
plotsky=PlotXY(pixRa, pixDec, line=0)
plotsky[0].setName("spaxels")
srcRa=myobs.meta["ra"].value
srcDec=myobs.meta["dec"].value
plotsky.addLayer(LayerXY(Double1d([srcRa]),Double1d([srcDec]), line=0,
  symbol=Style.FSQUARE))
plotsky[1].setName("Source")
plotsky.xaxis.title.text="RA"
plotsky.yaxis.title.text="Dec"
plotsky.getLegend().setVisible(True)
```

giving you something like this:

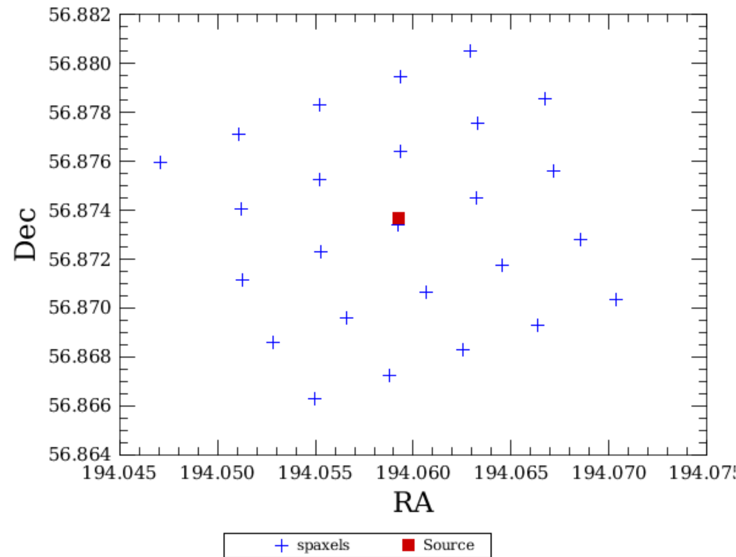


Figure 3.7. Pointing of the IFU and the source position. This shows you what the on-sky lauyout of the integral field unit of PACS is

Explanation:

- RESHAPE() is necessary for ra and dec because they have dimensions X,Y and Z, and so extracting out only the last entry of the third dimension, which is what the -1 syntax does, gives you a 2D array.
- myframe.ra/dec are the ra/dec datasets, which is not the same as the RaArray in the Status. "ra" and "dec" have dimensions X,Y,Z and were produced by the task specAssignRaDec, whereas Ra/DecArray are 1D (they are just for the central pixel), and were produced by the task specAddInstantPointing.
- The "-1" means you are asking to plot the ra for the final readout of the timeline (the last element in an array is specified with a -1); you can of course ask to plot all but that will make a very busy, and *very* slow, plot.
- srcRa and secDec are taken from the Meta data of the ObservationContext, these being the source positions that were programmed in the observation. Here we plot them as Double1d arrays, because PlotXY cannot a single value (which is what they are), so we "fake" them each into an array (in fact we are converting them from *Double* to *Double1d*).
- The different syntax here to previous examples shows you how flexible (or annoying) scripting in our DP environment can be. p[0].setName("spaxels") does the same as the l1.setName("signal") in a previous example. The first layer (layer 0) is always the one created with the "PlotXY=" command, subsequent layers can be added with the "plotsky.addLayer(LayerXY())" command.

If your observation includes several raster pointings you may want to plot the pointings for each. To to this you need to select out the part of myframe that corresponds to each pointing, and then you can use the same commands as written above to plot the pointings of each subframe. In Sec. 3.4.2 we explain how to split the frame on raster pointing. We also explain there how to plot the pointing for the two nods (A and B) and the chops (Sec. 3.4.3.2).

3.3.2.5. Display

It is also possible to look at your frame in 2D, using a display tool <LINK>. This is launched with:

```
Display(myframe.signal[:,:,100:150],depthAxis=2)
```

and when you zoom in you will see a 2D image: we are looking at the signal part of frame. Here we plot all X and Y ranges but only 50 wavelength/time-line layers (to plot all uses a lot of memory).

Plotting "spectra" is not possible with Display; you can, however, scroll through the signal time-line using the scroll bar at the bottom right of the image. depthAxis=2 tells Display to show the whole detector on the (2D) image and scroll along the time-line axis. depthAxis=0 and 1 are not useful to view with Display, showing you 1D "spectra" that are a single slice looking down successively along each of the detector axes.

Unfortunately, the 100:150 you specify above are the array positions, not the wavelength positions. If you want to Display (or otherwise look at) specific wavelengths of your frame you need to figure out what array positions are what wavelengths. To do this you can extract out the wavelength array, and by printing or plotting it, you can identify what array positions correspond to which wavelengths. So,

```
wave=myframe.getWave(8,12)
PlotXY(wave)
```

will plot a line of points, array position on the X axis and wavelengths on the Y axis.

3.3.3. Masks

First we explain the MaskViewer GUI, and then how to plot single spectra with and without masked data. Both are ways to inspect your masks, the GUI gives you a good overview and the PlotXY allows for a more detailed inspection. What you should be checking is that the readouts have been correctly masked. For example, were the readouts masked for UNCLEANCHOP readouts that were taken while the chopper was actually moving; or are the readouts msked for GLITCH really deviant in signal (note that glitches are masked only in the next part of the pipeline, so you would inspect these later)? With the MaskViewer you can check easily on the GLITCH mask since the MaskViewer shows you, for each pixel, the "spectrum" of signal vs. readout/time, with the masked data plotted in red and the unmasked in black. You can also look at the pixels flagged as bad or noisy or at readouts flagged as saturated, since the "spectra" from these should look worse (noisier/flattened due to saturation) than those from clean pixels. To check on the other masks, that is UNCLEANCHOP and GRATMOVE, you will use PlotXY as you need to overplot the signal with values taken from the Status, which you cannot do with the MaskViewer.

3.3.3.1. MaskViewer GUI

With the MaskViewer you can see which data were flagged in the masking tasks *and* at the same time you also get to see the data themselves. You can also see which masks are active, although this can also be done on the command line:

```
print myframes.mask.activeMaskTypes
```

The MaskViewer works on a frame and a ramp but not a cube. Using the MaskViewer you can also create and modify masks yourself, although if you wanted to do a mass-flagging of data it is easier to do that with a python script (see Chap. 4 for some help).

The capabilities of the MaskViewer are explained in <LINK>. To call it up type

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

At the top of the mask viewer (see the screenshot below) you see the PACS detector displayed in 5 bits, in which the individual pixels are selectable for viewing as a plot shown at the bottom of the GUI. In between these is a menu in which you can select which mask to look at—pixels (in the image) and data-points (in the plot) subject to this mask are plotted in red, all others in black. What you see in the plot below is the actual detector signal time-line for the selected pixel#"signal" vs. "sample index" (never wavelength)##and for the example averaged ramps data shown here there are 4 lines of datapoints following a zig-zag pattern. In the beginning the detector was looking at the internal calibration sources (the narrow messy block of data at the very left). Then it moved to observe a source, moving back and forth in wavelength leading to an up-down pattern in the dataline (as the

signal varies strongly with wavelength, if only due to the spectral response of PACS): 3 repeats of wavelength switching performed in nod position A (the first 3 "triangles"), B (second 3), B (third 3), and then again A (final 3). (Bear in mind that if your observations are in a flat part of the relative spectral response function and of a flat-spectrum, or very faint line, source, then you may not see well-defined triangle shapes because the flux from your source does not vary much with wavelength.)

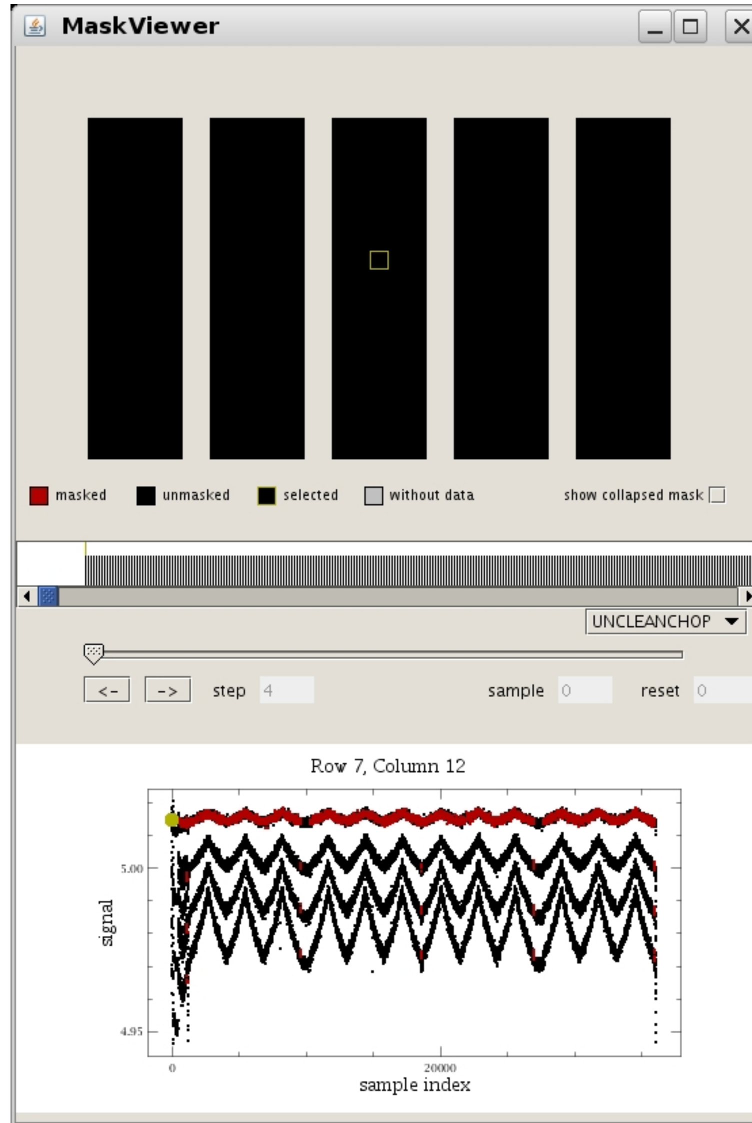


Figure 3.8. The MaskViewer window

The plot is based on PlotXY, and so the functionalities of PlotXY are available to you <LINK>. If you zoom in very tightly (right click inside the plot), and change the properties so that lines are joining the datapoints, you will see that the data of these 4 lines are joined, from the top to bottom. Each line of 4 descending dots is a single ramp of your *Ramps* product.

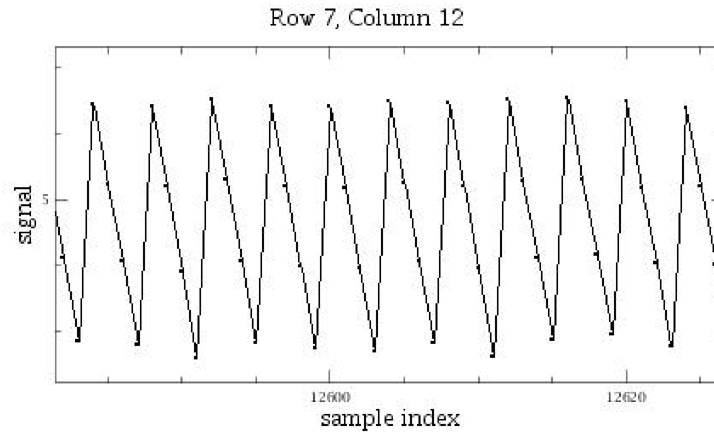


Figure 3.9. Zoom on a "spectrum" of a pixel of ramp in the MaskViewer

The slope of the line joining each 4 is essentially what is produced by the task fitRamps; this slope is a measure of the photocurrent in the detector and related to the infalling FIR flux. If you look at your frames data in this same way you will only see 1 line of data, the values thereof being the fit slopes.

If you zoom in tightly on the left of your timeline you should see the data of the calibration block (taken at the key wavelengths only). The two calibration sources have a different temperature and we chop between them, so you should see either 2 lines of datapoints of different signal level (for myframe) or 2 lines of sets_of_4 datapoints of different mean level (for myramp).

(Data that have been flagged are plotted in a different "layer" to the rest and by default as small red dots. Thus if you change the plot properties to have a line+points, the flagged data will be joined to themselves and not to the rest, leading to a plot that looks a little different to this. But this is just a facet of the plotting, not of the data themselves.)

When looking at the GLITCH masked data, glitches stand out as bright and narrow spikes in the signal timeline (more information is in Chap. 4). It is possible that a handful of readouts that do not look deviant will also have been masked as GLITCH. Extensive testing has shows us that many of these readouts would have been taken just after a cosmic ray (glitch) hit the detector, but that it hit between readouts being taken and hence is not obvious in the readout before. It is up to you whether you wish to accept these glitch-masked dataoints-as stated before, you could write your own glitch-detection algorithm and use that instead. A later "outliers" pipeline masking task is also peformed.

3.3.3.2. Plotting masked data

You can plot and overplot masked and unmasked data-points for single pixels using PlotXY. This is useful for checking in particular the UNCLEANCHOP and GRATMOVE masks. It is a more cumbersome way of looking at your data, but it is necessary at present if you want to overplot different parts of your frame (in this case, the signal and the masks). Here is an example of plotting all datapoints for pixel 8,12 and then overplotting the unmasked ones:

```
flx=myframe.getSignal(8,12)
wve=myframe.getWave(8,12)
p=PlotXY(wve,flx,line=0)
index_cln=myframe.getUnmaskedIndices(String1d(["UNCLEANCHOP"]),8,12)
p.addLayer(LayerXY
   (wve[Selection(index_cln)],flx[Selection(index_cln)],line=0))
```

Now to explain this. Have a little patience please, because this involves telling you something about the DP scripting language.

- The first two lines are how you extract out of your frame the fluxes and wavelengths and put in each them a new variable (which is of class *Double1d*, as you will see if you type > print flx.class). The

syntax "myframe.getSignal()" means you are calling on a "method" that is available for a *Frames* class object. A method is a set of commands that you can call upon for an object (myframe) of a class (*Frames*), these commands will do something to the object you specify—in this case it extracts out the signal or wavelengths from the frame. Methods can have any number of parameters you need to specify, in this case it is just the pixel number—8,12.

- The third command opens up a PlotXY and puts it in the variable "p", which you need to do if next want to add new layers to the plot. Line=0 tells it to plot as dots rather than a line (the default).
- The next command places into a (*Int1d*) variable called index_cln the X-axis (wavelength) array indices of the frame where the data have not been flagged for the specified mask. The parameters are the mask name/names (listed in a *String1d*) and the pixel coordinates (8,12). (You can use getMaskedIndices to select out the indices of the masked data points.)
- Finally you add a new layer to "p", in which you plot the unmasked data points, and these appear in a new colour. The syntax wve[Selection(index_cln)] will select out of wve those array indices that correspond to the index numbers in index_cln. You need to use the "Selection()" syntax because you are doing a selection on an array.

In the DP scripting language there is more than one way to do anything, and you may well be shown scripts that do the same thing but using different syntax. Don't panic, that is OK. But, do pay attention to the syntax—using a (instead of a [can cause a command to fail (or do the wrong thing).

To now look at your masked-unmasked signal vs the mask you mix-and-match the instruction here with those given previously for plotting masks (and noting that it is easier to do if you plot the signal against readout/time/array position rather than wavelength). To check, for example, if the signals masked as UNCLEACHOP were taken while the chopper was moving, you will plot the masked and unmasked signal and the Status CHOPPOS together, zoom in on the plot, and look to see if the masked readouts correspond to where the chopper was moving. For example, you will product a plot such as this:

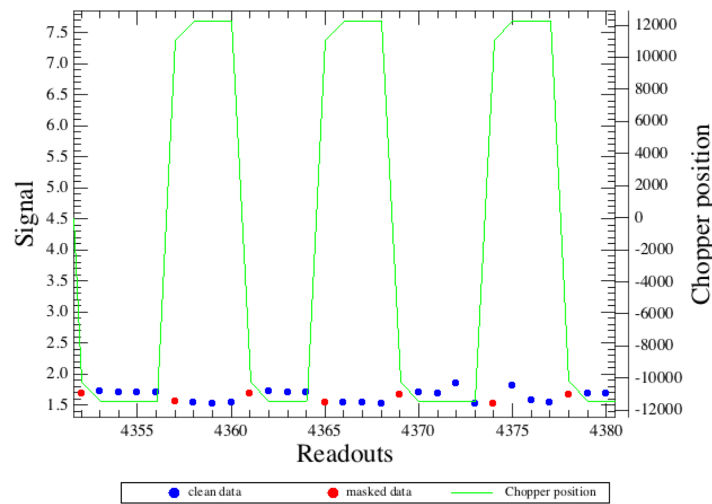


Figure 3.10. Masked/unmasked data v.s. Status

which was produced with the following script:

```
signal=myframe.getSignal(8,12)
x=myframe.getStatus("RESETINDEX")
# need to plot vs x so that the selected data ("Selection" below) are
# of the same dimensions as the unselected data
p=PlotXY()
l1=LayerXY(x,signal,line=0,symbol=Style.FCIRCLE,symbolSize=5)
l1.setName("clean data")
index_ncln=myframe.getMaskedIndices(String1d(["UNCLEANCHOP"]),8,12)
```

```
l2=LayerXY(x[Selection(index_ncln)],signal[Selection(index_ncln)],
   line=0,color=java.awt.Color.red,symbol=Style.FCIRCLE,symbolSize=5)
l2.setName("masked data")
l3 = LayerXY(myframe.getStatus("CPR"), line=1)
l3.setName("Chopper position")
l3.setYrange([MIN(myframe.getStatus("CPR")), MAX(myframe.getStatus("CPR"))])
l3.setYtitle("Chopper position")
p.addLayer(l1)
p.addLayer(l2)
p.addLayer(l3)
p.xaxis.title.text="Readouts"
p.yaxis.title.text="Signal"
p.getLegend().setVisible(True)
```

You can see that the red datapoints (masked) were taken at the same time that the chopper was not yet stable in position, i.e. it was still moving in to a stable value. For these data there are 4 readouts taken at a negative chopper position (-12000), then 4 at the positive (+12000), then back to the negative and so on. The first readout at each chopper + or - setting is still moving into position, which you can tell because the absolute value of the chopper position here is lower than those of the following 3 readouts. It is exactly these readouts that need to be flagged for UNCLEANCHOP. Changes in chopper or grating at the beginning of a "plateau" (i.e. where the values of the positions for these Status entries are stable) may be very slight. (It is also likely that all the masking will have been done correctly; in the standard AOT only one readout per "plateau" gets the UNCLEACHOP flag.)

See <LINK> to learn more about using PlotXY, and see the *SaDM* to learn more about scripting. In Chap. 4 we explain a little about creating your own masks.

3.4. Level 0.5 to 1

3.4.1. Pipeline steps

Before running this part of the pipeline read the "Special Issues" section below.

The next set of tasks to take you to Level 1 are

```
myframe = activateMasks(myframe, String1d([" -"]), exclusive=True)
myframe = specFlagGlitchFramesQTest(myframe)
myframe = activateMasks(myframe, String1d(["UNCLEANCHOP","GRATMOVE",
  -"GLITCH"]), exclusive = 1)
myframe = specEstimateNoise(myframe)
myframe = convertSignal2StandardCap(myframe, calTree=mycaltree)
myframe = activateMasks(myframe, String1d(["UNCLEANCHOP", -"GLITCH",
  -"BADFITPIX"]), exclusive=True)
csRespAndDark = specDiffCs(myframe, calTree=mycaltree)
myframe = specDiffChop(myframe)
myframe = rsrfCal(myframe, calTree=mycalTree)
myframe = specRespCal(myframe, calTree=mycaltree)
# before doing these last bits, see the Special Issues section
frameA=myframe.select(myframe.getStatus("IsAPosition") == True)
frameB=myframe.select(myframe.getStatus("IsBPosition") == True)
cubeA = specFrames2PacsCube(frameA)
cubeB = specFrames2PacsCube(frameB)
```

- These tasks do the following: Flag the data for glitches (cosmic rays) using the Q statistical test, creating a mask called GLITCH. Prior to this task you need to run activateMasks to dectivate all masks (i.e. it is as if there were no masks at all); the statistical test runs a lot better this way); Estimate the noise for each pixel and fill the Noise dataset (prior to this task you need to run activateMasks). This task is not necessary for the pipeline to be run (its results are not used elsewhere at present), it provides you with some noise information (essentially a standard deviation value calculated from the data median-normalised for variations due to the 2 chopper positions). At present the error calculations and propagation are still being worked on. Convert the signal to a value that would be if the observation had been done at the lowest detector capacitance setting (if this was the case anyway, no change is made. This task is necessary because the subsequent calibration tasks have

been designed to be used on data taken at the lowest capacitance); Calculate the dark current and pixel responses (prior to this task you need to run activateMasks), *although note that at present we do not use the result of this task so there is not much point you running it*; Subtract the off chops from the on chops to remove the rapidly varying telescope background. This will change the number of readouts and also subtracts the dark current; apply the relative spectral response function; Correct for the pixels' response; separate the nods (see below); turn the frames into cubes with dimensions of 5x5 spaxels (created from the 25 modules) and Z wavelength points, with 16*x individual spectra held in each spaxel. These 16*x spectra are from the 16 pixels that feed into each spaxel (pixels 1—16) each being of a slightly shifted wavelength range than the previous, and the x runs on the grating (ups and downs). In Chap. 4 we show you how to locate the 16*x separate spectra to inspect them.

- **Nod A and B.** Why do you have to separate the nods? PACS observations use the chop-nod method to remove the background; rapid chopping to remove the telescope background and less rapid nodding to compensate for the mirror temperature gradient (as when you chop you look at different parts of the mirror and so need to correct for that). There are 4 chop-nod combinations possible (chop- nodA, chop+ nodB, chop+ nodA, chop- nodB), and ideally the first 2 are both pointing at a blank sky position and the latter two are on source. One would subtract the + and - chops from each other, and then add together the remaining nod A and B data. (As the chops are subtracted from each other, the dark current is also removed.) Unfortunately we have the situation where the pointing on source for the chop+ nodA and chop- nodB are not pointing at exactly the same position. The offsets are not large, in Feb 2010 the offsets for the central spaxel are <1" for the small chopper throw and at worse are stil <2.5". For a point source confined to the central spaxel these offsets may matter (that is for you to decide). At present when running the pipeline you need to separate the *Frames* you are working with into 2 bits: nod A and nod B, as we have explained above. You then feed each frame into the cube-creating task. If in the ipipe scripts this task procedure is not included, and if you are reading this guide early in 2010, then this is one occasion where you should believe the PACS data reduction guide, not the ipipe scripts. If you overplot the spectra of a pixel of the nod A and B frame you should see 2 very similar spectra but with an offset in the continuum level.

- The glitch detection task works well in identifying glitches. By default it works on chopped data, and as this section is for a chop-nod AOT then you want the default case. (If not then you need to add the parameter "splitChopPos=False".) It has been tested on chopped and non-chopped data. After running this task if you look at your GLITCH mask it may seem to you that rather a lot of non-glitched datapoints have been masked, but in fact our tests show that a significant fraction of these are actually also glitch-affected. You could of course also write your own glitch detection algorithm, if you wanted. In Chap. 4 we tell you more about glitches.

- The task specRespCal corrects for the differential pixel responses, their the response drift (that occurs during your observation) and subtracts the dark current for staring AOTs—for chop-nod AOTs that is done when the chops are subtracted from each other. At this point in time (Feb. 2010) the dark current and nominal response used are the nominal values and we are still in the process of looking at this in more detail. The flatfielding—that is correcting for the pixel responses—is being improved upon. See below for more information on this.

- The tasks here that change the state of the data (and for which you may want to make a copy of myframe before running, as recommended in Sec. 3.3.1) are specDiffChop, rsrfCal, and specRespCal.

- The task specFrames2PacsCube simply rearranges the data from the frame, of 18x25 pixels, into a cube of 5x5 spaxels. It does nothing else. In particular it does not exclude any masked data. If you want to look at spectra from spaxels free of masked redouts you will need to select and plot in the same way as has been described previously, that is select only unmasked indices to plot.

3.4.2. Special issues

There are some particular considerations that will require to you step out of the pipeline, issues we are still dealing with. These are: (i) flatfielding (ii) rastered/dithered AOTs (iii) multi-line AOTs. For the multi-line and multi-raster datasets you will need to split the frame on line and raster position, as well as nod, so that later these data can be turned into separate cubes. Rastered cubes can then be combined.

3.4.2.1. Flatfielding

At this point in time the flatfielding is not perfect. This is something we are still working on, and it will improve over the coming weeks. If you overplot the spectra of the different pixels of a module (Sec. 3.3.2.2) you may notice that the continua do not line up. This is because the gain has not yet been correctly adjusted. If you check your data and decide that for your data it is significant enough to need to correct for it, you will need to do the correction manually. How you do that is up to you. Essentially you need to work out what the offset is for each pixel for each module separately, and then correct for that offset at the signal level of your *Frames* product (i.e. myframe). You may want to do this separately for the nodA and B data, mainly because if the pointings are slightly different then the spectrum of each pixel will be "smeared" out by combining nod A and B in one plot. You could, however, average the offset values you find for each pixel from the nods, if they are the same. The same reasoning should be applied to data that is from multiple lines, SEDs, or rasters. The spectra will be different so even if the offsets are the same, working out the offsets will require separating the data. Once you have worked out what your correction is for each pixel, then to adjust the signal level of a frame for that pixel you simply do the following:

```
sig=myframe.getSignal(8,12) -/ offset
myframe.setSignal(8,12,sig)
```

myframe.setSignal is a method you can use to replace the signal in a Frames. In the example here you have extracted out of myframe the signal for a single pixel, which creates a Double1d array, and divided it by a variable called offset; this "offset" must also be a Double1d. You then replace the signal in myframes for that pixel with the altered "sig". You can also call on this method simply as: myframe.setSigna(sig), where in this case "sig" is a Double3d array with dimensions the same as those of: myframe.dimensions, ie 18,12,z where z is the time dimension. You would do this if you have worked out the offsets to apply to all pixels at once, and of course then offset will have to be also a Double3d array. (See <LINK> for information about arrays).

Now while the offset is due to a gain misadjustment, testing has shown that the spectra line up better if you apply an additive offset rather than a multiplicative one. We are working hard to try to understand this, and you should consult either the most recent data reduction guide (in the most recent HIPE) or contact the helpdesk to ask them to pass on your query to the PACS team.

3.4.2.2. Rastered/Dithered AOTs

If you have multiple pointings in your dataset you need to do another slicing, because at present the later cube rebinning task does not honour these pointings but combined them as if all the same. (This may not be included in the ipipe scripts, since it is not part of the standard pipeline.) To look at how many pointing you have you can look at in Status:

```
print UNIQ(myframe.getStatus("RasterColumnNum"))
print UNIQ(myframe.getStatus("RasterLineNum"))
```

Where the first tells you how many column pointings were made (UNIQ will print out all the uniq values in the frame.getStats()) and the second, the line pointings. If your AOR requested dithering, that is observing with small (2" or so) jumps between successive pointings, you will probably have a small number of different column pointings, and if you have a full raster, you could have several line and column pointings. Changing column means PACS was moving along the PACS slit direction, changing line means PACS was moving perpendicular to the slit.

With multiple pointings in your data it is easier to handle (at a human level) your A and B products if you place them in a *ListContext*, which is a container of products. So, including the splitting by nod commands, you would do the following (after having run the pipeline up to specRespCal):

```
ListcubesA=ListContext()
ListcubesB=ListContext()
# To account for the fast that the raster counter sometimes
# can include data that are not onsource. This is included
# here because it can cause problems with the raster selection
# part of the script
```

```
frame=myframe.select(myframe.getStatus("IsOutOfField") == False)
# first slice on nod
frameA=frame.select(frame.getStatus("IsAPosition") == True)
frameB=frame.select(frame.getStatus("IsBPosition") == True)
# now for each of these, slice on raster. You will do this twice, one
# for each nod (frameA and frameB)
for rasterLine in UNIQ(frameA.getStatus("RasterLineNum")):
   for rasterColumn in UNIQ(frameA.getStatus("RasterColumnNum")):
      print -"doing pointing",rasterLine,rasterColumn
      frame_temp=frameA.select((frameA.getStatus("RasterLineNum")
       == rasterLine -) & (frameA.getStatus("RasterColumnNum") == rasterColumn))
      cube = specFrames2PacsCube(frame_temp)
      ListcubesA.refs.add(ProductRef(cube)) # add the cube to the list
```

This is quite complex scripting, and so we need a good explanation (but read also the *SaDM* guide for more scripting advice). First you select out the parts of the data that were for sure pointing on your source. This may not always be necessary, note, but it should not harm. If necessary and not done it can cause problems with the raster selection part of the script.

Then you make a frame for nod A and one for nod B. Then you look at the Status for the entry that indicates raster/dither position, which are RasterLineNum (a counter, 1 2 3 ...) and RasterColumnNum (also a counter). Look at the Status table and at these columns, you will see the column entries move from 1 to 2 to 3 ... as you scroll down the time direction (assuming you have multiple rasters in your dataset, that is). You now need to isolate the unique line and column values, and then slice iteratively on these. For each slice you turn it into a *PacsCube* and place that cube in a *ListContext* called ListcubesA|B. In the list there are as many cubes as there were unique line/column raster pointings. (You may want to add a meta data keyword that is the column and line number, so it stands out if you look at the cube with the Observation viewer.)

```
print len(ListcubesA.refs)
```

3.4.2.3. Multiple spectral line AOTs (with and without rasters)

If in your dataset you have several spectral lines that are not linked to each other by any continuum (i.e. you have separate and distinct spectral line scans, not a range scan or SED AOT) you will want to separate them from each other before you make the cubes. This is because otherwise the wavelength range will not be continuous and your cube will be more awkward to inspect. This is also something that is not included in the ipipe scripts, as it is not part of the standard pipeline. To deal with multiple lines we recommend a further separation of your data. There are several approaches you could take to how to split the data on line and on nod, one way to do this is the following:

```
# look in the BlockTable for the line identification numbers
# these will be integers: 1,2,3...
# (these will be the same for nod A and B)
lineIds = myframe["BlockTable"]["LineId"].data
print UNIQ(lineIds) # how many are there?

choice=["IsAPosition","IsBPosition"]
ListFrames=ListContext() # list to hold the sliced frames (optional)
ListCubes=ListContext() # list to hold the sliced cubes
for i in UNIQ(lineIds):
   # select the line to work on -- this creates a Selection, not an integer array
   lineIdsSel = lineIds.where(lineIds == i)
   # select the data corresponding to this line (index)
   startIdx = myframe["BlockTable"]["StartIdx"].data[lineIdsSel]
   endIdx = myframe["BlockTable"]["EndIdx"].data[lineIdsSel]
   # set up a Selection to hold our selected indices, fill with False
   selectionFrames=Bool1d(myframe.getSignal().dimensions[2],False)
   # fill the Selection with True where should be true
   for j in range(len(startIdx)): selectionFrames[startIdx[j]:endIdx[j]] = True
   frameL=myframe.select(selectionFrames)
   # now split into the nods, first A and then B (see -"choice" above)
   for j in range(2):
      frameN=frameL.select(frameL.getStatus(choice[j]) == True)
      ListFrames.refs.add(ProductRef(frameN)) # optional
      # create the cube and add that to a ListContext
```

```
      cube = specFrames2PacsCube(frameN)
      ListCubes.refs.add(ProductRef(cube))
```

This script will create a single *ListContext* in which you will place your line and nod segments of the original myframe. Say you have 3 lines, the order of the cubes (or frames) in the list will be: line 1 (nod A, nod B); line 2 (nod A, nod B)l line 3 (nodA, nod B).

This scripting calls upon the BlockTable to do your selection, rather than the Status as has been the case otherwise so far. Because of this, the script above will only work if the BlockTable has this information in it, which mainly depends on what stage of your pipeline processing you are at. But since you should be doing this only after the specRespCal task the BlockTable should be complete. You can look at your BlockTable with the Dataset viewer by selecting "myframe" in the Variables panel to sent it to the Editor panel (Product viewer) or to bring it up in the listing in the Outline panel. You can then right-mouse menu select the Dataset viewer for the BlockTable dataset, and you will see a tabular listing of what is held therein. In Chap. 4 we explain more about the BlockTable (and the Status).

What this script does is to look at the BlockTable to see what line identification (numbers) there are, and then to select for each the index (a.k.a. array position) for each line identification, and split myframe on those indices. It then puts the new frames into a *ListContext* called ListFrames, and the then-created cubes into ListCubes. These frames are added in the order they were organised in the BlockTable, which may not be in wavelength order (i.e. the frame with a line at 63 microns may be be before the frame with a line at 58 microns).

If you have rastered or dithered observations you will then need to split on pointing as well as line. Again, it is up to you how to organise your splitting. If you have many lines and many rasters it could become difficult to remember what everthing is if you place it all in one *ListContext*, so here we take the approach of keeping the lines in separate *ListContexts*, with each including for each rasters the nod A and B, and then placing each line list in a super *ListContext*.

```
# to account for the fast that the raster counter sometimes
# can include data that are not onsource. This is included
# here because it can cause problems with the raster selection
# part of the script
frame=myframe.select(myframe.getStatus("IsOutOfField") == False)
# look in the BlockTable for the line identification numbers
# these will be integers: 1,2,3...
# (these should be the same for each raster and nod)
lineIds = frame["BlockTable"]["LineId"].data
lines= UNIQ(lineIds)
# what rasters are there (these should be the same for each line and nod)
rasterLine=UNIQ(frame.getStatus("RasterLineNum"))
rasterColumn=UNIQ(frame.getStatus("RasterColumnNum"))
choice=["IsAPosition","IsBPosition"]

SuperListCubes=ListContext() # list to hold the sliced cubes
for i in lines:
   ListCubes=ListContext() # list to hold individual lines
   # select the line -- this creates a Selection, not an integer array
   lineIdsSel = lineIds.where(lineIds == i)
   # select the data corresponding to this line (index)
   startIdx = frame["BlockTable"]["StartIdx"].data[lineIdsSel]
   endIdx = frame["BlockTable"]["EndIdx"].data[lineIdsSel]
   # set up a Selection to hold our selected indices, fill with False
   selectionFrames=Bool1d(frame.getSignal().dimensions[2],False)
   # fill the Selection with True where should be true
   for j in range(len(startIdx)): selectionFrames[startIdx[j]:endIdx[j]] = True
   frameL=frame.select(selectionFrames)
   # now split into raster
   for line in rasterLine:
      for column in rasterColumn:
         print -"Doing raster",line,column
         frameR=frameL.select((frameL.getStatus("RasterLineNum") == line -)
           & (frameL.getStatus("RasterColumnNum") == column))
         # now split into the nods, first A and then B (see -"choice" above)
         for j in range(2):
            frameN=frameR.select(frameR.getStatus(choice[j]) == True)
            cube = specFrames2PacsCube(frameN)
```

```
            # add cube for eg nod A, raster 1, line 1 to the list
            ListCubes.refs.add(ProductRef(cube))
            print -"cube length",len(ListCubes.refs)
   SuperListCubes.refs.add(ProductRef(ListCubes))
```

The list called ListCubes contains the frames in the following order: for the first line, for each raster you have nod A and B, e.g. raster 1 (nod A, nod B), raster 2 (nod A, nod B), raster 3 (nod A, nod B). Each time you go through the outside loop this *ListContext* is cleared, ready to take the next set of raster+nods for the next line. At the end of the outside loop you place the filled ListCubes in a list-of-lists, called SuperListCubes,. This last step is not necessary, but is useful if you have many lines and raster. To extract a single ListContext containing the raster+nods for a single line, say the 3rd line, you will use the syntax: >List1=SuperListCubes.refs[2].product. To extract then out of List1 the 4th raster+nod combination (which will be raster 2 nod B - see above) you use the syntax: >List2 = List1.refs[3].product.

It is recommended that you run a sanity check by plotting the ra and dec of your frames/cubes split on nod and raster!

If you wish have an SED or range scan and have therefore several BANDS (see the Status) in your data, you may want to compare the spectra of the bands separately. This is easily done, and will be explained later. There is no need, however, to split the frames on band as you do on spectral line.

3.4.3. Inspecting the results

For the Level 0.5 frame you are likely to want to check the spectra, masked data, and see how the spectra vary with chop and nod, that is the spectra after the tasks specFlagGlitchFrames, specDiffChop and specAddNod. You will probably also want to compare the spectra before and after the rsrfCal and specRespCal tasks, since these move the flux units from V/s to Jy and removed the spectral response of the instrument. For the cube you will want to compare the cubes of nodA and B and maybe also look at the spectra of the 16 pixels that feed each spaxel. At present there are no GUIs that work on *Frames* and *PacsCubes* product, so you have to use PlotXY.

3.4.3.1. For the *Frames*

Before and after tasks

To compare the spectra before and after a task has been run, e.g. the rsrfCal and specRespCal tasks, is simple and uses variations on the same recipies we have already given for the previous Level's data reduction. Remember to copy the frame before you run it through tasks if you want to compare a before and after frame:

```
frame_b4 = myframe.copy()
# then run the pipeline tasks
myframe = rsrfCal(myframe, calTree=calTree)
myframe = specRespCal(myframe, csResponseAndDark=RespandDark,
   calTree=mycalTree)
# then plot
sig=myframe.getSignal(8,12)
wve=myframe.getWave(8,12)
p = PlotXY(wve,sig,titleText="your title",line=0)
p[0].setName("after rsrf")
p[0].setYrange([MIN(sig), MAX(sig)])
p[0].setYtitle("Jy")
sig=myframe_b4.getSignal(8,12)
wve=myframe_b4.getWave(8,12)
p.addLayer(LayerXY(wve,sig),line=0)
p[1].setName("before rsrf")
p[0].setYrange([MIN(sig), MAX(sig)])
p[0].setYtitle("V/s")
p.xaxis.title.text="Wavelength [$\mu$m]"
p.getLegend().setVisible(True)
```

where the labelling of the Y axis, as well as its range, will here be different for the two spectra, allowing you to compare them.

With the same set of commands you can overplot the spectra before and after the specDiffChop task, and also after you have split on the nods. Bear in mind that when you look at the spectrum if a single pixel, signal versus wavelength, before specDiffChop has been run (or before you have split for nod), you will see what looks like many spectra plotted on top of each other: at least one for each chop position and one for each nod position (and probably one for each run on the grating, as there should be at least two runs on the grating per observation). If you plot the spectra versus array position (i.e. simply do not specify the X axis), this is the same as plotting versus time, and there you will see the spectra changing with time because with time also the instrument configuration (grating, chopper, nodding) changes. So, if you plot a pixel before having subtracted the chops or split on nod, the spectrum (signal versus wavelength) will consist of several separate spectra, all sitting on top of each other. This was shown in Fig. 3.3. After having run these tasks your spectrum will be cleaner, as shown here:

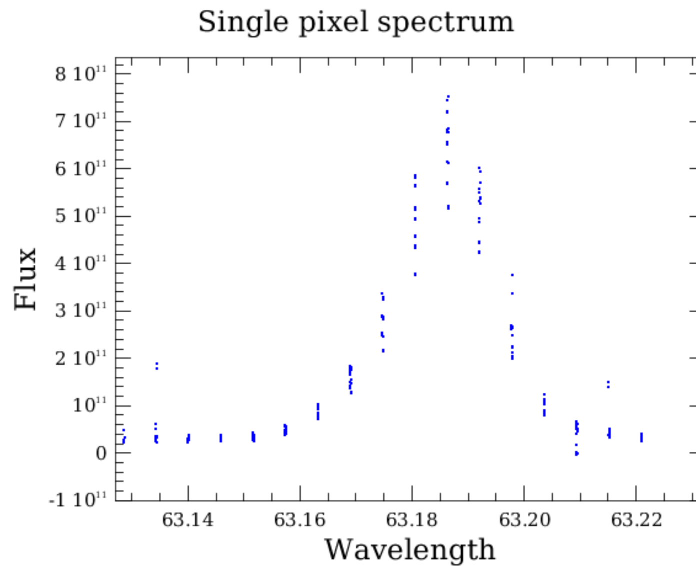


Figure 3.11. Spectrum of a single pixel. The spectrum is plotted versus wavelength and there are in fact 2 spectra plotted here, one for each run on the grating

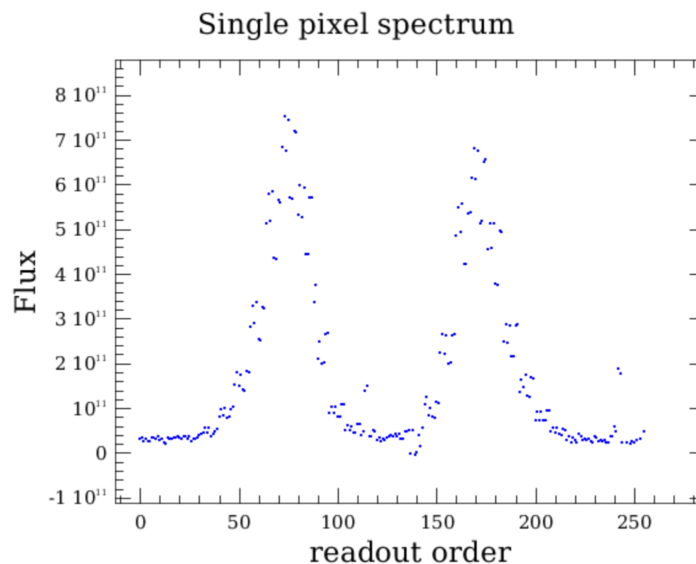


Figure 3.12. Spectrum of a single pixel. The spectrum is plotted versus readout order and the separate spectra that in the figure above lie on top of each other are now distinguishable

There are 2 separate spectra here because, even afer the chops and nods have been combined, there are still 2 runs (scans) on the grating remaining. When this this frame is turned into a cube by the pipeline task specFrames2PacsCube, 16 pixels, each with 2 spectra, are put into each spaxel (remember that there are 16 pixels in the frame that feed each spaxel of the cube). Later pipeline tasks will take these 16*2 spectra and merge them into one.

Later in this section we explain how to compare the spectra from a single pixel of a frame that are at different combinations of chop and nod, i.e. on- and off-source pointings, for the version of myframe you have *before* you run specDiffchop.

Plotting grating scans

How do select out the different grating scan repeats? Your observations may include more than one run on the grating, that is, more than one coverage of the wavelength range. Normally we run along the grating range in one direction and then back again, and you may have asked to repeate this pattern. To select out the different scan directions, that is forward or back, you can select on your frame on the Status parameter called SCANDIR, which takes on the values of 0 or 1.

If you have multiple sets of grating scan directions, multiple sets of 0 and 1, then of course when you select out the 0 directions, you will get all of them, they will not be separated into the distinct time segments. If you really want to overplot those, then you can look at the timestamps that correspond to the distinct segments, and identify what array position in the signal dataset corresponds to those segments. So:

```
#plot the scandir status vs array index
PlotXY(myframe.getStatus("SCANDIR"))
#Look at the plot to work out what array positions (X-axis values)
#you want to select out and plot the spectrum of.
#Say you want to plot 0:100 and 200:300 for pixel 8,12
p=PlotXY(myframe.wave[8,12,0:100],myframe.signal[8,12,0:100], line=0)
p.addLayer(LayerXY(myframe.wave[8,12,200:300],myframe.signal[8,12,200:300],line=0))
```

Note that you can plot all 16 pixels for a single module by simply specifying the appropriate range in the signal dataset of the frame: RESHAPE(myframe.signal[1:16,12,0:100]).

Alternatively you can select instead on the BlockTable, using syntax first introduced in the "Special Issues" of slicing on line and raster. However, here also if you select on "ScanDir" you will get all the data of scan direction 0 or 1, and if you wish to select particular 0s or 1s only, you will need to look at the start and end index that corresponds to the particular 0s and 1s you wish to look at. This script is longer but does not require any manual inspection as the previous one does. (This script is not the only way you can do this, and not even necessarily the most efficient, but it follows the syntax used before and hence is easier to explain.)

```
scanIds = myframe["BlockTable"]["ScanDir"].data
print scanIds

# let's say this gives a listing: 0 1 0 1 0 1
# (assuming they haven't changed the numbering recently)
# and you want to plot all 0 vs all 1
# find the starting and ending indices for the 0 and 1
scanIdsSel = scanIds.where(scanIds == 0)
startIdx0 = myframe["BlockTable"]["StartIdx"].data[scandsSel]
endIdx0 = myframe["BlockTable"]["EndIdx"].data[scanIdsSel]
scanIdsSel = scanIds.where(scanIds == 1)
startIdx1 = myframe["BlockTable"]["StartIdx"].data[scanIdsSel]
endIdx1 = myframe["BlockTable"]["EndIdx"].data[scanIdsSel]
# create smaller frames containing only these data
selectionFrames=Bool1d(myframe.getSignal().dimensions[2])
selectionFrames[:] = False
for j in range(len(startIdx)): selectionFrames[startIdx0[j]:endIdx0[j]] = True
frame0=myframe.select(selectionFrames)
selectionFrames[:] = False
for j in range(len(startIdx)): selectionFrames[startIdx1[j]:endIdx1[j]] = True
frame1=myframe.select(selectionFrames)
```

```
# now plot
p=PlotXY(frame0.getWave(8,12),frame0.getSignal(8,12),line=0,color=java.awt.Color.blue)
p.addLayer(LayerXY(frame0.getWave(8,12),frame0.getSignal(8,12),line=0,color=java.awt.Color.red)

# now you want to plot the first 0 vs the first last 0
scanIdsSel = scanIds.where(scanIds == 0)
startIdx0 = myframe["BlockTable"]["StartIdx"].data[scanIdsSel]
endIdx0 = myframe["BlockTable"]["EndIdx"].data[scanIdsSel]
```

If you want to plot these for the cube rather than the frame, see below.

Plotting different bands

If you want to compare the bands in your observation (and this makes sense mostly after you have run RSRF and specRespCal) then you need to extract on the Status. The entry to look for is BANDS, and your script could look something like this:

```
print UNIQ(myframe.getStatus("BAND"))
# giving a reply such as ["B2A", -"B2B"]
# BAND has only the timeline dimension
band1 = (myframe.getStatus("BAND") == -"B2B")
if (ANY(band1)):
    w = band1.where(band1)
    signal=myframe.getSignal(pix,mod)[w])
    wave=myframe.getWave(pix,mod)[w])
```

First you locate all the readouts (the timeline datapoints) that correspond to observations at band "B2B", then you create a *Selection* product called w which is True for the readouts where the band was so set and False otherwise. Then you plot the spectrum for those selected readouts only.

3.4.3.2. Overplotting spectra of chop and nod combinations

Here we show you how to select out from your frame (not cube), the parts of the data of chop+ nod A and chop- nod B, which should be pointing both at the target, and chop- nod A and chop+ nod B which should both be off-target. You can then yourself, using previous instructions, plot these spectra and pointings, any indeed other combination of chop and nod you wish.

(I) First you need to select out the parts of the frame that correspond to the 2 nods, as shown before (Sec. 3.4.1). You can then use the previously given scriptettes to overplot the spectrum from the same pixel for frameA and frameB, and to plot out the pointing for the final datapoint (or the first, or the middle, or any random datapoint) in frameA and frameB.

(II) If you want to select out the chop throw + and - you do the frame selection thus:

```
frameP=myframe.select(myframe.getStatus("CHOPPOS") == -"+large")
frameM=myframe.select(myframe.getStatus("CHOPPOS") == -"-large")
```

This works if your programmed chopper throw was large. You will need to look at the CHOPPOS Status entry to see if your throw range is large, medium or small (it will not change during an observation). You can use >print UNIQ(myframe.getStatus("CHOPPOS")) to find out what CHOPPOSitions were for your observations. There will be several position keywords returned, but you are only interested in whether you have "large", "medium" or "small".

(III) To select on nod and chop together you combine the selections like this:

```
framePA=myframe.select( (myframe.getStatus("CHOPPOS") == -"+large") &
 (myframe.getStatus("IsAPosition") == True) -)
frameMB=myframe.select( (myframe.getStatus("CHOPPOS") == -"-large") &
 (myframe.getStatus("IsBPosition") == True) -)
```

and the plot you can make from these selections will look like this:

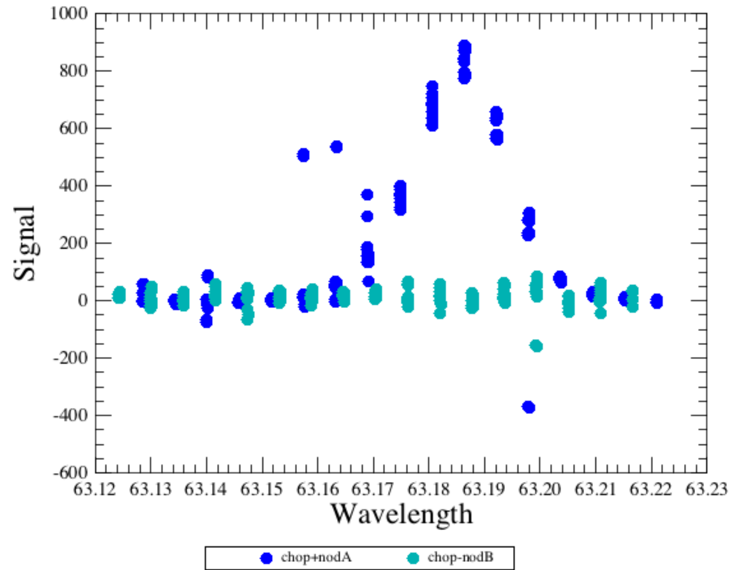


Figure 3.13. Spectrum of chop- nod A and chop+ nod B

It is clear that the spectrum that is off-source (chop- nodB) has no spectral line, which is only seen for the on-source (chop+ nodA) data. The pointing that corresponds to these positions will look something like this:

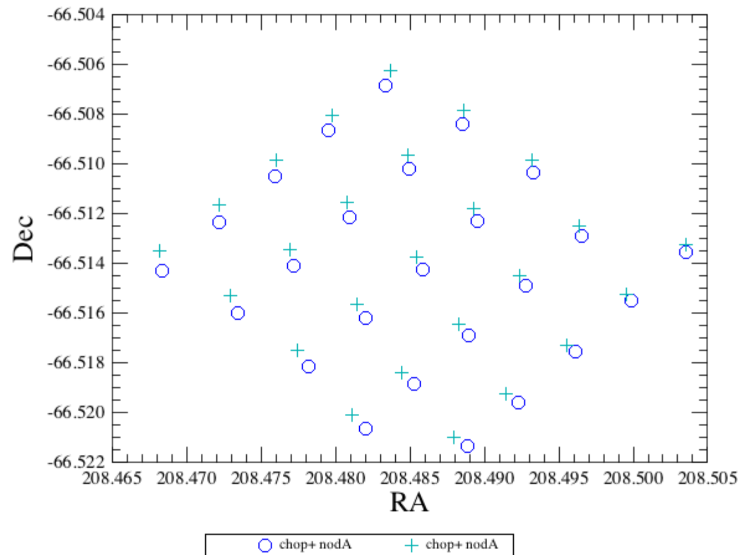


Figure 3.14. Pointing for chop- nod A and chop+ nod B

These data are from PV phase observations, hence the slight offset in the positions for the spaxels (yours will overlay more closely). Don't be alarmed by the slightly higgldey-piggledy nature of the spaxel positions, this is what the integral field unit looks like on-sky.

You will now be able to add in any other selections you wish to compare, e.g. for raster position (Status entries RasterLineNum and RasterColumnNum, and look at the Status table yourself to see what range of values these take on). (You may also want to plot only the unmasked readouts.)

Note One thing to bear in mind is that depending on when in your data reduction you do the plotting, it is possible that you will be plotting data/pointings that belong to the calibration block or to slewing

periods as well as your astronomical data. These parts of the data should generally be cleaned away by the time the frame is ready to be turned into a cube, or you can use other Status entries to eliminate them. We refer you to Chap. 4 where the Status is further explained.

So far we have shown you how to do a selection on the frame, creating a smaller frame where the Status corresponds to some desired limits. If you want to combine selections, e.g. select only the unmasked data and those from nod A, then you could use the following syntax:

```
pix=8
mod=12
notGlitched = (myframe.getMask("GLITCH")[pix,mod,:] == False)
nodA = (myframe.getStatus("IsAPosition")[:] == TRUE)
2plot = notGlitched & nodA
if (ANY(2plot)):
    w = 2plot.where(2plot)
    signal=myframe.getSignal(pix,mod)[w])
    wave=myframe.getWave(pix,mod)[w])
```

Here, for pixel (8,12), you select the data that are not glitched and are from nod A. "notGlitched" and "nodA" are all *Bool1d*, that is arrays of length equal to the timeline-length of myframe, and which are of class *Bool1d* and hence contain the values True or False where the condition (e.g. GLITCH=False) has or has not been met (so a bad datapoint has a mask value True). These are then merged into one *Bool1d* called 2plot. Then, if there are any TRUE in 2plot then (i) move 2plot into an array and (ii) extract out of myframe the signal and wave but selecting only those array positions that correspond to the "w" array that was made from 2plot. You can then plot these *Double1d* arrays or do anything else you wish to them.

3.4.3.3. For the *PacsCube*

The *PacsCube* product is undergoing a change, after which it should contain a Status table in the same way that a *Frames* product does. When this has been implemented, selecting out parts of a *PacsCube* will follow the same logic as selecting out parts of a *Frames* product, and most of what was said above will apply here also. An addition to the Status will be pixel, so you will be able to select for each spaxel the individual pixels (of which there are 16), to inspect or manipulate as you wish.

At present the *PacsCube* holds the relevant information in separate datasets, as you can see in the screenshot below. It also has a BlockTable.

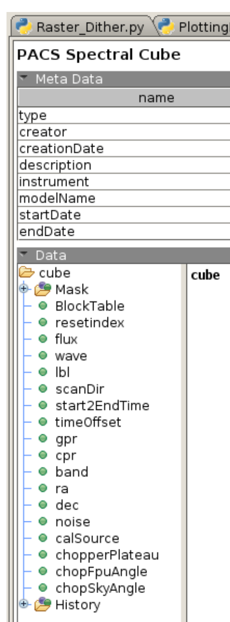


Figure 3.15. PacsCube listing

Certain methods previously introduced for *Frames* work on *PacsCube* also. For example, to plot the spectrum of a single spaxel and overplot the spectrum of the unmasked data points:

```
flx=mycube.flux[:,2,2]
wve=mycube.wave[:,2,2]
p=PlotXY(wve,flx,line=0)
index_cln=mycube.getUnmaskedIndices(String1d(["GLITCH"]),2,2)
p.addLayer(LayerXY(wve[Selection(index_cln)],flx[Selection(index_cln)],line=0))
```

Remembering that to extract out the cube from a ListContext you type something like: >cube=MyListContex.refs.[1].product.

Currently none of the visualisation GUIs that were listed in Chap. 1 works on a *PacsCube*. Instead you have to use PlotXY. You can plot the spectrum of a single spaxel in mycube with

```
p=PlotXY(cube.wave[:,2,2],cube.flux[:,2,2],titleText="your title")
p.xaxis.title.text="Wavelength [$\mu$m]"
p.yaxis.title.text="Signal [Jy]"
```

Note that the wavelength dimension is the first, not the last as with a frame, and the spatial dimensions are 5,5.

As mycube contains in each of its 5x5 spaxels simply all of the spectra that belong to that point of the sky, if you plot versus wavelength you will see a mess of a spectrum, at least 16 spectra overlayed, and probably more (one spectrum per pixel and one also per grating run). If you plot versus array order (which is the same as time) then you will see these separated out. This is clear from the next 2 figures.

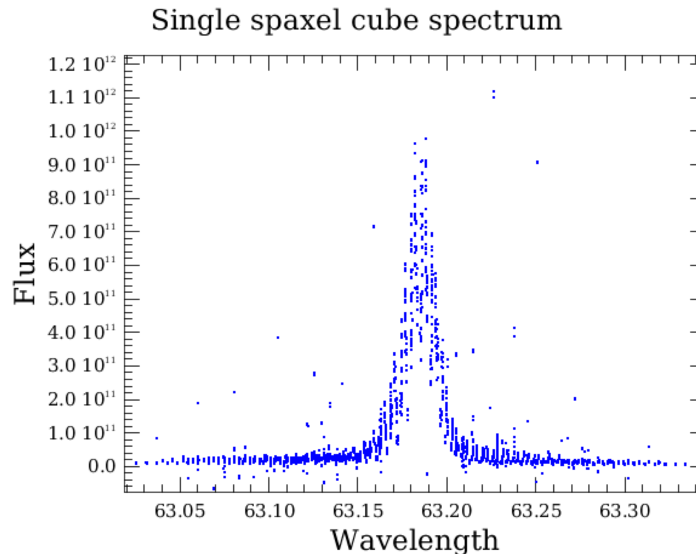


Figure 3.16. Spectrum of a single spaxel in the pacsCube. The spectrum is plotted versus wavelength and the separate spectra all lie on top of each other: 32 in total: 2 grating runs from each of the 16 pixels that each spaxel is fed by

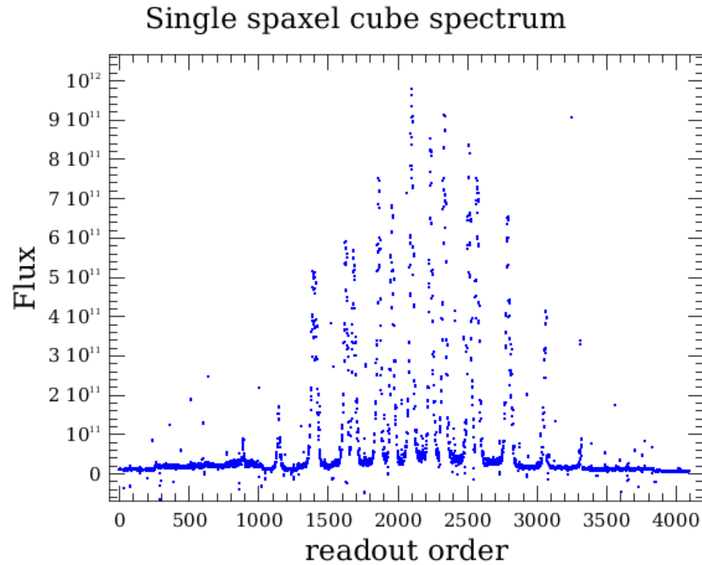


Figure 3.17. Spectrum of a single spaxel in the pacsCube. The spectrum is plotted versus readout order and the separate spectra that in the figure above lie on top of each other are now distinguishable

If you want to see where in the cube your source is located, so you know which spaxel to plot, you will need to use Display to plot the 2D image:

```
Display(mycube.flux[1000:1010,:,:],depthAxis=0)
```

Which will display a 2d image that is 10 (wavelengths) thick, and the brightest spaxel will be your source. Note that the 1000:1010 specifies the array positions of flux, you do not type in the wavelenghts here. depthAxis=0 here because it is being run on a cube, not a frame. (Top-left is position 0,0).

Plotting the spectra of the different pixels in each spaxel

How do you go about selecting out the various pixels in each spaxel of a *PacsCube* to plot and compare? In fact, the only thing that will be separatable in the cubes' spaxels will be pixel and grating run, because the chops and nods have already been combined (or sliced out). To compare the spectra of different pixels in a spaxel, until the *PacsCube* contains a Status table it is far easier for you to inspect the *Frames* product (myframe) rather than the cube. We have already shown you how to overplot the pixels of each module (each module is a single spaxel), e.g. pixel (1,12) to (15,12) are the first to last pixel of module 12, and module 12 is the central spaxel (2,2).

3.4.3.4. Special issues

If you split your data and placed them in a ListContext, then plotting the spectra is simply a matter of following the preceeding recipies after having extracted the single cubes or frames from the ListContext. The syntax for that is

```
cube=ListCubesA.refs[0].product
frame=ListFrames.refs[1].product
# how many frames are there in total?
print len(LineFrames.refs)
```

3.5. Level 1 to 2

3.5.1. Pipeline steps

3.5.1.1. Basic procedure

The final tasks take the cube from Level 1 to Level 2. First we need work out the rebinning details for the cube to give it a uniform wavelength grid, based on what wavelengths are currently in the cube. We then do another glitch detection, creating a mask called OUTLIERS. Then the nod A and B can be combined. The 16*x spectra held in each cube are then resampled and merged according to the wavelength grid, creating a new cube (*PacsRebinnedCube*), a process for which you can specify which masked data to exclude. Finally, that cube is spatially resampled, projected onto a regular RA and Dec grid on the sky, and rasters are combined, to create the final cube (*SpectrumSimpleCube*). The final cube should be considered at present to only be of browse quality, as there are still issues to do with the spatial—spectral rebinning and the mosaicing together of rasters that need addressing.

The tasks that take you to just before the *PacsRebinnedCube* are:

```
waveGrid = wavelengthGrid(mycube, calTree=mycaltree, oversample=2,
      upsample=3)
mycube = activateMasks(mycube, String1d(["GLITCH","UNCLEANCHOP",
     -"SATURATION","GRATMOVE", -"BADFITPIX"]), exclusive=True)
mycube = specFlagOutliers(mycube, waveGrid, nIter=2, nSigma=5)
mycube = activateMasks(mycube, String1d(["GLITCH","UNCLEANCHOP",
     -"SATURATION","GRATMOVE", -"BADFITPIX", -"OUTLIERS"]), exclusive=True)
rebinnedCube = specWaveRebin(mycube, waveGrid)
```

The first task, wavelengthGrid, creates a single and uniform wavelength grid according to the wavelengths that are in the mycube (a *PacsCube*). The parameters listed here are all optional, the values given are our recommendation. `oversample` is by how much you want to oversample the wavelength bins from what they are at present and `upsample` is by how much you move forward along the original wavelength array as you calculate the new resampled wavelength array. It is possible that waveGrid you create will depend on the type of observation you have, so try various grids and compare the resulting spectra. Bins too large will smooth the data, bins too small will make the spectra too "bitty". specFlagOutliers does a type of sigma-clipping, and by activating the masks before running it you are telling it not to mask these data points which have already been masked. The parameters we specify are our recommendations, and they are optional (there are good default values hardwired into the task). `nIter` is the number of iterations and `nSigma` the sigma value to flag at. The penultimate task selects all the masks for which you do not want the readouts included when the data are combined, and then specWaveRebin resamples the flux domain based on the wavelength grid (waveGrid) and combines the spectra that have been stuffed into each spaxel of the *PacsCube*, using this wavelength grid to do the rebinning. This creates a cube of dimensions [wavelength,5,5] where now in each spaxel holds a single spectrum. The final cube is of class *PacsRebinnedCube*.

3.5.1.2. Combining nods

As at a minimum you have split your observation on nod, you will want to combine the data after they have been turned into cubes. If the offsets are very small, if your source is confined to the central spaxels and the specra of the nods are very similar, then you can combine the cubes simply by adding them together (we show you how to do that below). If, however, you have a very faint source, or a spatially very inhomogenous source and you think the spatial offsets are too large, if the spectra from the same spaxel of the 2 nods are different by a lot in the continuum level or also in the spectral line profile, then combining the cubes will require a sophisticated "nod adding, spatial resampling and mosaicking" routine. The tasks to do this at at sophisticated level are not yet ready, so at present your final product will not be perfect.

Why is this so? The reason for chopping is to allow one to remove the telescope background for each grating position. However, there is a temperature gradient across the mirror, and the chops point at

slightly different parts of the mirror, as well as the sky. The differential spectra you get after the task specDiffChop is:

nodA=(SourceA+SkyA+Tel_cold) - (Tel_warm + SkyAoff)nodB=(SourceB+SkyB+Tel_warm) - (Tel_cold + SkyBoff)

where Tel_cold and Tel_warm simply indicate 2 (very slightly) different telescope temperatures. In the chop-nod scheme you then remove the telescope background by adding:

(SourceA+SourceB) + (SkyA-SkyAoff) + (SkyB-SkyBoff) + (Tel_Warm-Tel_Warm) + (Tel_Cold-Tel_Cold)

If the telescope points exactly at the same position in nod A and nod B (which currently is almost the case for chopper throw = small), then SourceA=SourceB and SkyA=SkyB=Sky. If the sky is well behaved (if it is linear between all sky positions) then the midpoint between SkyAoff and SkyBoff equals Sky and the sky contributions cancel. In this situation the average of the nods returns the source: Source=(nodA+nodB)/2.

Nevertheless, the equations above also mean that the nods, taken separately, contain different contributions from the telescope background, hence they cannot be directly compared (i.e. they are not exactly the same). This is especially for faint sources, e.g. you may find that the continuum of the spectrum on nod A is negative, and especiallly so at the large chopper throw. However, it *is* necessary to combine the nods to remove the telescope background from the source's own continuum. You cannot simply treat them as separate pointings and combine them as a mosaic (i.e. treat them as 2 raster pointings), you *have* to add them in order to remove the telescope background (look at the equations above).

The conclusion at this point in time is that combing nod A and nod B is the only thing you can do within the pipeline. If the object varies spatially on smaller scales than our spatial resolution, it will be 'smoothed', but for now you will have to live with it. We *are* working on creating pipeline tasks to deal with this.

Continuing with the pipeline: the masking task specFlagOutliers is run on the nod A and B cubes separately, otherwise the continuum offsets of the two will cause more data to be flagged as outliers than is necessary. If you have looked at your rebinnedCube spectra of nod A and B and decided that you do want to combine the nods, then at present you do that "manually", i.e. you add and divide by 2 (a pipeline task to do this is currently being written). This should be done after the cubes have been rebinned, because (i) the rebinning excludes data that have been masked. If a readout has been masked in nod A but not B and you add and divide by 2 that readout, you will either get a very bad datapoint (because nod A was masked and hence bad) or if you exclude nod A you will be (incorrectly) halving the value of the nod B datapoint. (ii) the rebinning moves the spectra to the same wavelength grid, which is necessary for a correct combining of data if the combining is a simple "add and divide by 2". Hence, you should create and use the **same wavelengthGrid for each nod**.

To combine the nods you do the following:

```
sigA=rebinnedCubeA.getFlux()
sigB=rebinnedCubeB.getFlux()
# this gives you 2 Double3d arrays
sig=(sigA+sigB)/2.
rebinnedCubeAB=rebinnedCubeA.copy()
rebinnedCubeAB.setFlux(sig)
```

This will make a copy of rebinnedCubeA but replaces the signal with the average of A and B. Since rebinnedCubeAB is a copy of rebinnedCubeA it will contain the associated datasets of nod A, in particular the RA and Dec datasets. If you wanted to average those also, you use the same syntax but with the method ".getRa()" and ".setRa()' (or Dec instead of Ra) instead.

If you want to see the projected cube (which is recommended if only because most of the data visualisation GUIs work on the *SpectrumSimpleCube* and not, at present, the *PacsRebinnedCube*) you do the following

```
projectedCube=specProject(rebinnedCubeAB)
```

3.5.1.3. Combining nods, rasters and lines

Read 5.1.2 for information about combining nods. The addition of rasters is not a big complication (certainly not compared to the complication of adding nods!). You will extract the cubes out of the *ListContexts* into which you have placed your rasters and nods, and run the same commands listed in 5.1.1 for each nod separately, and then run the 5.1.2 commands. The only difference is that rather that feeding a single cube into specProject you will feed in a *ListContext* containing your individual rebinnedCubeABs, one for each raster pointing: > projectedCube=specProject(ListContext).

Exactly the same should be followed when you have multiple lines in your dataset, although of course you will not want to combine the different lines into one final cube!

3.5.2. Inspecting the results

To summarise: the mycube is a final Level 0.5 product, rebinnedCube is Level 1 and projectedCube is a Level 2 product. The mycube is a *PacsCube* class product, the rebinnedCube is a *PacsRebinnedCube*, and the projectedCube is a *SpectrumSimpleCube*. For the latter cube you can use a number of GUIs (outlined in Chap. 1). At some point during the 4.0 track of HIPE these, or other, GUIs will also work on the *PacsCube* and the *PacsRebinnedCube*.

To inspect a cube you need to take it out of the *ListContext* we have had you place them in. The syntax for doing this is:

```
mycube=ListCubesA.refs[0].product
```

and etc. So you need to do this before following any of the advice next given.

3.5.2.1. Plotting and visualising cubes

Other than using the GUIs, you can use PlotXY to inspect spectra. Plotting a single spectrum from rebinnedCube and projectedCube are done differently, an inconsistency that you will have to bear with for now.

```
# REBINNED CUBE
# first check the dimensions to know how many spaxels can be plotted:
print rebinnedCube.dimensions
# plot a single spaxel's spectrum as thus
PlotXY(rebinnedCube.wavegrid,rebinnedCube.flux[:,2,2])
# PROJECTED/SPECTRUMSIMPLE CUBE
# get the dimensions
print projectedCube.dimensions
PlotXY(projectedCube.getWave(),projectedCube.getFlux(10,10))
```

This will give you plots looking similar to this:

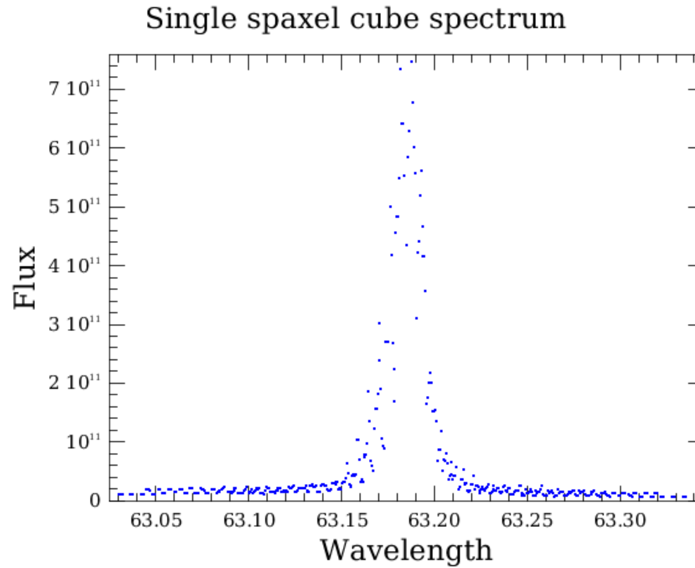


Figure 3.18. Spectrum of a single spaxel in the rebinnedCube. The spectrum is cleaner than the example from the PacsCube because the spectra have been combined and rebinned

Remembering that the rebinnedCube contains only one spectrum per spaxel, created from the throng of spectra that occupied each spaxel of the previous stage *PacsCube*, there is not so much to check here (although a lot to admire). If you sliced your data on raster then you may want to check the spectra from spaxels overlapping between the pointings, although consider that the different pointings are overlapping, not overlying. How can you tell which spaxel in which raster are overlapping? Probably the conceptually easiest way is the plot the pointing of the rebinnedCubes you wish to compare. You can do this thus:

```
#select the final ra and dec of a PacsRebinnedCube
ra=RESHAPE(rebinnedCube1.ra[-1,:,:])
dec=RESHAPE(rebinnedCube1.dec[-1,:,:])
plotsky=PlotXY(ra, dec, line=0)
plotsky[0].setName("cube1")
#select the final ra and dec of a second PacsRebinnedCube
ra=RESHAPE(rebinnedCube2.ra[-1,:,:])
dec=RESHAPE(rebinnedCube2.dec[-1,:,:])
plotsky.addLayer(LayerXY(ra, dec, line=0))
plotsky[1].setName("cube2")
plotsky.xaxis.title.text="RA"
plotsky.yaxis.title.text="Dec"
plotsky.getLegend().setVisible(True)
```

Which will plot the RA and Dec of 2 raster pointings. If you want to select from that 2 spaxels to compare the spectra of, you need to relate position on the skyplot to position in the rebinnedCube. This plot may help for the projectedCube:

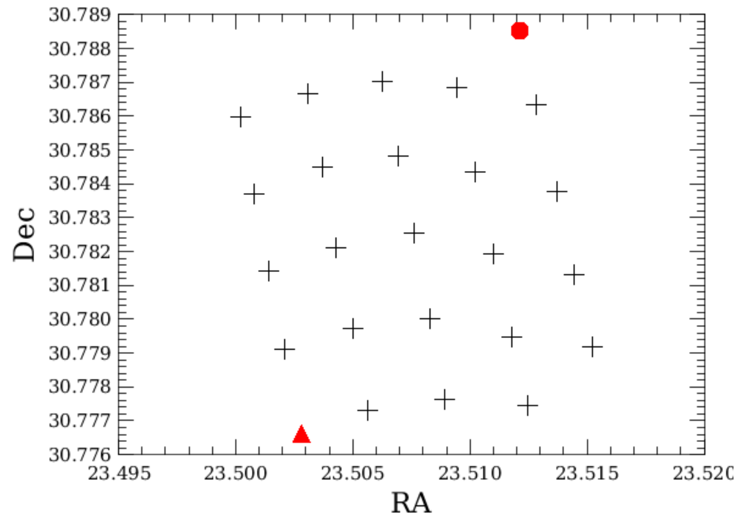


Figure 3.19. The red circle is from the projectedCube[:,0,0] and the red triangle from projectedCube[:,4,4]. I am not 100% certain, but spaxel 0,0 is probably, on the Frames product, module 0, and 4,4 is probably module 24

For both cubes you can also use Display, with the same syntax as with the pacsCube. Below is a screenshot of Display on the projectedCube (data from an observation of more-or-less nothing). From it you can see that the spaxels are smaller but cover the same total area:

Figure 3.20. Display on a projectedCube

3.6. The End Of The Pipeline

Congratuations! Now you know how to run the pipeline and how to inspect the intermediate products. The hard work is only just beginning.........

In Chap. 4 (not yet provided) are discussed issues that are of importance for certain types of AOTs and also issues that have arisen in the course of your pipeline processing. We also tell you more about the Status and the BlockTable, as these are two very important datasets that allow you to make selections on your observation for inspection purposes. Also included is: how to save and restore the products that you create while pipeline processing your data; how to identify glitches; working with masks; noise and errors within the pipeline; the PSF. If you want to know more about the parameters and altgorithms of the pipeline tasks you should read the PACS Advanced User Manual, available on the HIPE help page.

Chapter 4. Further topics. *Spectroscopy*

4.1. Introduction

This chapter is still being written. Chapter 4 will be reinserted in April.

Chapter 5. In the Beginning is the Pipeline. *Photometry*

5.1. Introduction

The main purpose of this chapter is to tutor users in running the PACS photometry pipeline. Previously we showed you how to extract and look at the Level 2 fully pipeline-processed data; if you are now reading this chapter we assume you wish to reprocess the data and check the intermediate stages. Later chapters of this guide will explain in more detail the individual tasks and how you can "intervene" to change the pipeline defaults; but first you need to become comfortable with working with the data reduction tasks.

The PACS pipeline runs as a long series of individual tasks, rather than as a single application. We will take you through the pipeline tasks one by one through all the levels. Up to Level 0.5 the data reduction is level independent.

A suggestion before you begin: the pipeline runs as a series of commands, and as you gain experience you may want to add in extra tasks, construct your own plotting mini-scripts, write if loops and remember what it is you did to the data. Rather than running the tasks on the command line of the Console (and having to retype them the next time you reduce your data), we suggest you write your commands in a python text file and run your tasks via this script.

The pipeline steps we outline here are also available in the ipipe scripts (one per AOT). These can be found in the directory where you installed the HIPE software, hopefully in /scripts/pacs/toolboxes/spg/ipipe. We suggest you copy the relevant file and open it in HIPE. You can then follow this manual and that ipipe script at the same time, editing as you go along (and please excuse any differences between the ipipe script and this guide, but they will not always be updated at the same time: generally the ipipe scripts should be updated first).

This chapter has been taken from the more advanced data reduction guide and so it is more complex than you will need. Throughout PV and SD phase it will be improved upon, at present you will just have to accept that it is not quite ready. You will need to read Sec. 1 and 2 of Chap3. before beginning here. Also, what there is called mycalTree, here is called calTree.

How to write in a script text file in HIPE:

From the HIPE menu and while in the Full/Work Bench perspective select File → New → Jython script. This will open a blank page in the Editor. You can write commands in here (remember at some point to save it... if HIPE has to be killed you will lose everything you have not saved). As you are doing so you will see at the top of the HIPE GUI some green arrows (run, run all, line-by-line). Pressing these will cause lines of your script to run. Pressing the big green arrow will execute the current line (indicated with a small dark blue arrow on the left-side frame of the script). If you highlight a block of text the green arrow will cause all the highlighted lines to run. The double green arrow runs the entire file. The red square can be used to (try to) stop commands running. If a command in your script causes an error, the error message is reported in the Console (and probably also spewed out in the xterm, if you started HIPE from an xterm) and the guilty line is highlighted in red in your script. A full history of commands is found in History, available underneath Console for the Full Work Bench perspective.

5.2. Retrieving your ObservationContext and setting up

Before beginning you will need to set up the calibration tree. You can either chose that which came with your data or that which is attached to your version of HIPE. The calibration tree contains the information HIPE needs 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. As long as your HIPE is recent then the caltree that comes with it will be the most recent, and thus most correct, calibration tree. If you wish to recreate the pipeline processed products as done at the Herschel Science Centre you will need to use the calibration tree there used, i.e. that which comes with the data (and which is shown in Fig. 2 of Chap. 1). We recommend you use the calibration tree that comes with HIPE. Structurally, the two are the same, but the information may be different (more, or less, up-to-date).

```
# from your data
caltree=myobs.calibration
# or from HIPE recommended
caltree=getCalTree("FM","BASE")
# where FM stands for flight model and is anyway the default
obs.calibration=calTree
```

It is necessary to extract a few other products in order for the pipeline processing steps to be carried out. These are the dmcHead, the pointing product, and the orbit ephemeris. You can get these, to be used later, with

```
pp=myobs.auxiliary.pointing
oem = obs.auxiliary,refs["OrbitEphemeris"].product
hkdata = myobs.level0.refs["HPPHK"].product.refs[0].product["HPPHKS"]
```

The orbit ephemeris (oem) is used to correct for the movements of Herschel during your observation, the pointing product is used to calculate the pointing of PACS during your observation.

You also need to get the time correlation product to correct the time in the meta data

```
timeCorr = obs.auxiliary.timeCorrelation
```

Then you need to retrieves the Observation Context from your pool as it was explained in Chap. 1. Continuing from there: since you are re-reducing the data you will want to start from Level 0 in case of scanmap and level_0.5 sliced frames in case of the point source AOT.

5.2.1. Scan map AOT

For scan map mode you access you data in the following way

```
myframe = myobs.level["level0"].refs["HPPAVGB"].product.refs[0].product (in case
of the blue array)
```

```
or
myframe = myobs.level["level0"].refs["HPPAVGR"].product.refs[0].product (in case
of the red array)
```

where myobs is the ObservationContext from Chap. 1. This extracts out from Level 0 the first of the averaged blue (or red) ramps. If there is only one you still need to specify refs[0], if there is more than one you select the subsequent with refs[1], refs[2],...... To find out how many HPPAVGBs are present at Level 0, have a look again at Fig. 2 from Chap. 1; if you click on the + next to HPPAVGB it will list all (starting from 0) that are present.

An alternative way to get your HPPAVGB..ref[x] product is to click on myobs in the Variables panel to send it to the Editor panel, click on +level0, then on +HPPAVGB to see the entries 0, 1, 2... You can then drag and drop whichever entry you want to start working on first to the Variables panel. The command that is echoed to the Console when you do this will be very similar to the one you typed above, only now the new product is called "newVariable" (which name you can change via a right click on it in the Variables panel).

In case you want to retrieve a parallel mode observation getObservation does not work, and the following script shall be used:

```
archive = HsaReadPool()
store = ProductStorage()
store.register(archive)
query=MetaQuery(ObservationContext,"p","p.instrument=='PACS' and
p.meta['obsid'].value==%il" %(obsid))
result=store.select(query)
obs=result[0].product
```

Since you start with the level0 product you need to identify the blocks in the observations. In the current observation design strategy a calibration block is executed at the beginning of any observation. It is possible that in the future the current design will be changed to include more than one calibration block to be executed at different times during the observation. In order to take into account this possible change, the pipeline includes as a very first step a pre-processing of the calibration block(s) that is planned to work under any possible calibration block(s) configuration. The calibration block pre-processing is done in three steps: a) the calibration block(s) is identified and extracted from the frames class, b) it is reduced by using appropriate and pre-existing pipeline steps, c) the result of the cal block data reduction is attached to the frames class to be used in the further steps of the data reduction.

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

and remove the calibration block to keep only the science frames:

```
myframes = removeCalBlocks(frames)
```

Unfortunately removeCalBlocks still leaves sometimes a few frames of the calibration block hence the following is recommended until further notice to remove the initial calibration block

```
skip=430 (or some other observation dependent number)
frames = frames.select(Int1d.range(frames.signal.dimensions[2]-1-skip)+skip)
```

These are organisational tasks. Their purpose will be discussed in later chapters. You also need to add the pointing information using:

```
myframe = photAddInstantPointing(myframe, pp,calTree=calTree, orbitEphem=oep,
horizons=horizons, isSso=isSso)
```

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

This first part of the astrometric calibration deals with two elements: the satellite pointing product and the SIAM product. Both are auxiliary products of the observation and are contained in the Observation context delivered to the user. The satellite pointing product gives info about the Herschel pointing. The SIAM product contains a matrix which provides the position of the PACS bolometer virtual aperture with respect to the spacecraft pointing. The time is used to merge the pointing information to the individual frames.

5.2.2. Point Source AOT

For point source mode you start with the level0.5 sliced frames so you don't need to worry about the calibration blocks and initial pointing.

```
camera= -'blue'
or
camera = -'red'
level0_5 = PacsContext(obs.level0_5)
slicedFrames=level0_5.averaged.getCamera(camera).product
pacsPropagateMetaKeywords(obs,'0',slicedFrames)
```

The last line is needed to make sure that all the keywords which are available for the level0 products are assigned to the slicedFrames as well. You can also check what is the size of your data cube and the number of slices you have:

```
print -"Data cube dimension: -"+str(slicedFrames.refs[1].product.signal.dimensions)
noofsciframes=slicedFrames.meta["repFactor"].long/2
print noofsciframes
```

For the old (pre OD150) L0 processed data the filter information is not correct so you need to execute the following piece of code to make it right. Later it is going to be an independent pipeline task but for the time being we need to live with this temporary solution.

```
if camera == -'blue':
 # calibration block slice
 wpr=slicedFrames.refs[0].product.getStatus("WPR")
 band=slicedFrames.refs[0].product.getStatus("BAND")
 if wpr.where(wpr == 0).length() > 0:
   if band[wpr.where(wpr == 0)][0]=='BS':
     print -'WARNING for blue filter: WPR=0 was erroneously assigned BS, now reset
to BL'
     band[wpr.where(wpr == 0)] = String('BL')
```

```
 if wpr.where(wpr == 1).length() > 0:
   if band[wpr.where(wpr == 1)][0]=='BL':
     print -'WARNING for blue filter: WPR=1 was erroneously assigned BL, now reset
to BS'
     band[wpr.where(wpr == 1)] = String('BS')
 slicedFrames.refs[0].product.setStatus("BAND",band)
 # science block slice
 wpr=slicedFrames.refs[1].product.getStatus("WPR")
 band=slicedFrames.refs[1].product.getStatus("BAND")
 if wpr.where(wpr == 0).length() > 0:
   if band[wpr.where(wpr == 0)][0]=='BS':
     print -'WARNING for blue filter: WPR=0 was erroneously assigned BS, now reset
to BL'
     band[wpr.where(wpr == 0)] = String('BL')
 if wpr.where(wpr == 1).length() > 0:
   if band[wpr.where(wpr == 1)][0]=='BL':
     print -'WARNING for blue filter: WPR=1 was erroneously assigned BL, now reset
to BS'
     band[wpr.where(wpr == 1)] = String('BS')
 slicedFrames.refs[1].product.setStatus("BAND",band)
```

You also need to correct one of the keyword in case of a red channel

```
# for red channel only key word missing
if camera == -'red':
  slicedFrames.meta["repFactor"] = LongParameter(noofreps)
```

You can make several check on your data before beginning to process. E.g. check the size of the cube

```
print frames.signal.dimensions
```

which might be interesting to know if you deal with large amount of data. Or the repetition factor

```
print obs.meta["repFactor"].value
```

which helps you determine how many slices you will need (see later).

5.3. Level 0 to Level 0.5

The PACS Photometer pipeline is composed of tasks written in java and jython. In this section we explain the individual steps of the pipeline up to Level 0.5. Up to this product level the data reduction is mostly AOT independent. The only AOT dependent task executed in this part of the data reduction is the CleanPlateauFrames task, which is executed only for chopped observations.

Next the pipeline tasks are introduced in the order they should be run.

Having the sliced frames, you execute the following for each nod-cycle. For the scanmap mode you have to skip the "Extract one slice" part and start directly with the processing since that there are no slicing in scanmap mode.

```
for i in range(noofsciframes):
```

```
 # ++++++++++++++++++++++++++++++++++++++++++++++++++
 # Extract one slice
 # ++++++++++++++++++++++++++++++++++++++++++++++++++
 framesnod = slicedFrames.getCal(0).copy()    # This stands, index is always zero
 framesScience = slicedFrames.getScience(i).copy()  # this goes from 0 to the
number of ABBA nods.
 framesnod.join(framesScience)
 #
 # ++++++++++++++++++++++++++++++++++++++++++++++++++
 # Processing
 # ++++++++++++++++++++++++++++++++++++++++++++++++++
 framesnod = photFlagBadPixels(framesnod)
 framesnod = photFlagSaturation(framesnod)
 framesnod = photConvDigit2Volts(framesnod)
 #framesnod = photCorrectCrosstalk(framesnod)
 # ground-based correction is overcorrecting, hence switched off for the time being.
 if (timeCorr -!= None) -:
    frames = addUtc(frames, timeCorr)
 framesnod = convertChopper2Angle(framesnod)
 framesnod = photAssignRaDec(framesnod)
```

5.3.1. photFlagBadPixels

The purpose of this task is to flag the bad or noisy pixels in the BADPIXEL mask. The task should do a twofold job: a) reading the existing bad pixel mask provided by a calibration file ("PCalPhotometer_BadPixelMask_FM_v1.fits" in the current release), b) identifying additional bad pixels during the observation. In the current version of the pipeline only the first functionality is activated. The algorithm for the identification of additional bad pixels is not in place. So the task is just reading the bad pixel calibration file and transforming the 2D mask contained in it in the 3D BADPIXEL mask. The task is doing the same for the BLINDPIXEL mask. This is an uplink mask, which currently is completely set to false. The purpose is to use it to indicate the pixels which should not be read at all and for which data should not be downloaded.

```
myframe = photFlagBadPixels(myframe,calTree=calTree)
```

A note on syntax: myframe is the input frame (which in Chap 1. and 3. we have called myframe) and myframe in the output frame. It is up to you whether you give myframe the same name as myframe; it is certainly possible for you to do so, and for tasks that only flag data it is recommended, otherwise you will clutter up the HIPE memory with many products. Also note that we use myframe as frame in the individual task description to be consistent with the previous chapters but the ipipe script uses different variable name (e.g. framesnod as in the above partial script).

5.3.2. photFlagSaturation

This tasks identifies the saturated pixels on the basis of saturation limits contained in a calibration file for the two types of saturation possible: readout circuit and the Analogue to Digital Converter (ADC) Before doing that, the task identifies the reading mode led by the warm electronic BOLC (Direct or DDCS mode) and the gain (low or high) used during the observation. These information are provided for each sample of the science frames by the BOLST entry in the status table. The task compares the pixel signal at any time index to the dynamic range corresponding to the identified combination of reading mode and gain. Readout values above the saturation limit are flagged in the 3D SATURATION mask.

```
myframe = photFlagSaturation(myframe -,calTree=calTree)
```

5.3.3. photConvDigit2Volts

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

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

```
myframe = photConvDigit2Volts(myframe -,calTree=calTree)
```

5.3.4. addUtc

The task provides correction of time difference between the on board time and ground UTC using the time correlation file. A new status column Utc is added.

```
myframe = addUtc(myframe -,timeCorr)
```

5.3.5. photCorrectCrosstalk

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

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

where '*a*' is the percentage of signal of the correlated pixel to be removed from the signal of the crosstalking pixel. The task is still under investigation, in the sense that invariability of 'a' is still an assumption to be tested in further tests. Currently it is not used in the pipeline because ground-based correction is over correcting.

5.3.6. photMMTDeglitching and photWTMMLDeglitching

These tasks detect, mask and remove the effects of cosmic rays on the bolometer. Two tasks are implemented for the same purpose: photMMTDeglitching is based on the multi resolution median transforms (MMT) proposed by Starck et al (1996), WTMMLDeglitching is based on the Wavelet Transform Modulus Maxima Lines Analysis (WTMML). The latter task is still under investigation and debugging phase, so only the multi-resolution median transform is supported. At this stage of the data reduction the astrometric calibration has still to be performed. Thus, the two tasks can not be based on redundancy. Both tasks have to overcome the following problems:

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

A full explanation of what these tasks do, how they work and results of testing them, is left to the Appendix. To run them, use

```
myframe = photMMTDeglitching(myframe, incr_fact=2,mmt_mode='multiply', scales=3,
nsigma=5)
myframe = photWTMMLDeglitching(myframe)
```

However, these task are not part of the standard PS pipeline, so do not use them when reducing PS data. We mention them here because later they might become part of the pipeline. The photMMTDeglitching is part of the scanmap pipeline.

5.3.7. convertChopper2Angle

This task converts the Chopper position expressed in technical units to angles. This is done by reading the CPR entry in the Status table and express it in two ways:

- as angle with respect to the FPU (CHOPFPUANGLE entry in the Status table)

- as angle in the sky (CHOPSKYANGLE).

Both angles are in arc seconds. In particular, the CHOPFPUANGLE is a mandatory input for the PhotAssignaRaDec task, to be executed after Level 0.5 for the final step of the astrometric calibration. Thus, the convChopper2Angle task must be executed even if the chopper is not used at all as in the scan map (chopper maintained at the optical zero). CHOPFPUANGLE corresponds to the chopper throw in arc seconds in HSpot.

```
myframe = convertChopper2Angle(myframe, calTree=calTree)
```

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

5.3.8. photAssignRaDec

The purpose of this task is to convert the image into World Coordinate System by assigning RA and DEC coordinates to each pixels using the Array Instrument calibration file with spatial distortions.

```
myframe = photAssignRaDec(myframe)
```

5.3.9. cleanPlateauFrames

This task is executed before Level 0.5 only for chopped observations.

```
myframe = cleanPlateauFrames(myframe)
```

The module flags the readouts at the beginning of a chopper plateau, if they correspond to the transition between two chopper positions. In the chopper transition phase, the chopper is still moving towards to proper position and the signal of this readouts does not correspond to the on or off position. Usually the chopper is moving so fast that only one readout needs to be masked out. The module just adds the 3D UNCLEANCHOP mask to the input frame. The task identifies the chopper plateaus on the basis of the CHOPPERPLATEAU (for the science data) and CALSOURCE (for the calibration block) entries in the status table. For each chopper plateau the readouts with a chopper position deviating from the mean position (threshold provided by the calibration file ChopJitterThreshold) are flagged in the UNCLEANCHOP mask. However, this task is superfluous in its current implementation, hence it is not used.

5.4. The AOT dependent pipelines

After level 0.5, the pipeline is AOT dependent. In the following sections we will describe separately the different AOT pipelines, point source, small source, chopped raster, scan map AOTs, up to Level 2. There is two observing modes available using the PACS Photometer. The point source mode and the scanmap mode.

5.5. Point Source AOR

5.5.1. Level 0.5 to Level 1

```
framesnod = photMakeDithPos(framesnod)
framesnod = photMakeRasPosCount(framesnod)
framesnod = photAvgPlateau(framesnod,skipFirst=True,copy=1)
framesnod = photAddPointings4PointSource(framesnod)
framesnod = photDiffChop(framesnod)
framesnod = photAvgDith(framesnod,sigclip=3.)
framesnod = photDiffNod(framesnod)
framesnod = photCombineNod(framesnod)
framesnod = photRespFlatfieldCorrection(framesnod)
#frames = photDriftCorrection(frames)
```

5.5.1.1. photMakeDithPos

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

```
myframe = photAvgPlateau(myframe)
```

5.5.1.2. photMakeRasPosCount

The task adds raster position counter to status table.

```
myframe = photMakeRasPosCount(myframe)
```

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

5.5.1.3. photAvgPlateau

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

```
myframe = photAvgPlateau(myframe,skipFirst=True,copy=1)
```

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

The signal of the bad pixels, identified by the BADPIXEL mask, is reduced by the task as the unmasked pixel. The pixels flagged in the other available masks (SATURATION, GLITCH, UNCLEANCHOP) are discarded in the average. If the chopper plateau contains no valid data (all pixels masked out) the signal is set to *NaN* (Not a Number). The noise is calculated for each pixel (*x,y*) and each plateau (*p*) as:

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

where *nn* is the number of valid readouts in the chopper plateau. This number is then stored as addition entry (`NrChopperPlateau`) in the status table. The noise is stored in the `Noisemap`. The skipFirst=True option gets rid of the first frame of each plateau. It is needed since the first group of 4 averaged readout after the chopper motion will have a different value from the one following it as the signal takes a few 40 Hz readouts to adjust to the new level

5.5.1.4. photAddPointing4PointSource

The task extracts pointing information for further photometer PointSource processing. Stores the averaged ra,dec of the virtual aperture for both nod positions, dither positions and chop positions and adds the PhotPointSource Dataset to the Frames class. It contains per nod position, dither position and chopper position the first value of : RaArray, DecArray, PaArray, CPR, DithPos, NodCycleNum, ChopperPlateau, isAPosition. This information is later used on PhotProjectPointSource to map the Frames.

```
myframe = photAddPointing4PointSource(framesnod)
```

5.5.1.5. photDiffChop

Subtract every off-source signal from every consecutive on-source signal. The result is a Frames class with one image per one chopper cycle. Note that in PS mode the 'off-source' image also contains the source but on a different position.

```
myframe = photDiffChop(myframe)
```

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

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

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

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

5.5.1.6. photAvgDith

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

```
myframe = photAvgDith(myframe, sigclip=3.)
```

This task uses several entries in the status table to identify the on-off differential images (output of `photDiffChop`) belonging to the A and B Nod position of a given Nod cycle and dithered position (*DithPos*, *NodcycleNum*, *IsAPosition*, *IsBPosition*, see output of `photAddIstantPointing`). Since only the average of the identified images is performed, the noise is propagated as follows:

For "c" chopper cycles (c=k), we average the n/2 differences: noise [x,y] = SQRT(MEAN(noise[x,y,:]**2)) / SQRT(n)

The sigclip=3. takes care of the deglitching.

5.5.1.7. photDiffNod

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

```
myframe = photDiffNod(myframe)
```

The noise is propagated as follows:

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

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

5.5.1.8. photCombineNod

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

```
myframe = photCombineNod(myframe)
```

The noise is propagated as follows:

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

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

5.5.1.9. photRespFlatFieldCorrection

The task applies flat field corrections and converts signal to a flux density:

```
myframe = photRespFlatFieldCorrection(myframe,calTree=calTree)
```

The formula managing the flat-field, the flux calibration and the photometric adjustment is the following:

$$\mathcal{S}f(t) = \mathcal{E}s(t) * \frac{(\delta\mathcal{C}_0)}{(\delta\mathcal{C})} * \frac{1}{(J\Phi)}$$

Figure 5.1.

where *f(t)* is the flux in Jy, *s(t)* is the signal in Volt, *DC0* is the difference of the calibration sources got during a calibration campaign, *DCs* is the difference of the calibration sources computed by the cal-block pre-processing, *J* is a flux calibration factor which contains the responsivity and the conversion factor to Jansky, *Phi* is the normalized flatfield. The ratio *1/J*Phi* converts the signal *s(t)* in Volt to *f(t)* in Jansky. This task applies the ratio *1/J*Phi* to flat-field and flux calibrate the data.

The noise is calculate in the following way:

```
noise = SQRT( s_out^2 * [ (sigmas2/s2) + (sigmaC0^2/C0^2) + (sigmaCs^2/Cs^2) -] -)
```

where *s* is the input signal in Volt, *sigmas* is the input noise, *C0* is our reference, *sigmaC0* is the noise of the reference, *DCs* is the differential image of the cal-block and *signaDCs* is the noise associated to that.

Addendum: the first *DC0* has been determined with data collected during ILT test campaign. The following biases have been used: 2.6 V for both the blue and green channel, 2.0 V for the red one.

5.5.1.10. photDriftCorrection

The task applies the drift correction of the flat field and controls the photometric stability:

```
myframe = photDriftCorrection(myframe)
```

The `PhotDriftCorrection` task has the goal to multiply the signal *s(t)* by the ratio *DC0/DCs*, where *DC0* is the differential image of the two internal calibration sources (calculated from the same data of the flat-field), *DCs* is the differential image of *CS1-CS2* obtained from the calibration block of the observation (output of the cal-block pre-processing). This factor corrects possible drift of the flat field. This drift can be due either to an alteration of the internal calibration sources or to an evolution of the detector pixels. The drift is compared with photometric stability threshold parameters (stored in the calibration files). If the ratio overtakes these thresholds, a *DriftAlert* keyword is added to the metadata. Note, that the task is currently not part of the standard pipeline

5.5.2. Level 1 to Level 2

5.5.2.1. photProject and photProjectPointSource

The `photProject` task provides one of the two methods adopted for the map creation from a given set of images (in the PACS case, a frame class). The task performs a simple coaddition of

images, by using a simplified version of the drizzle method (*Fruchter and Hook, 2002, PASP, 114, 144*). It can be applied to raster and scan map observations without particular restrictions. The only requirement is that the input frame class must be astrometric calibrated, which means, in the PACS case, that it must include the cubes of sky coordinates of the pixel centers. Thus, `photAddInstantPointing` and `photAssignRaDec` should be executed before `PhotProject`. There is not any particular treatment of the signal in terms of noise removal. The *1/f* noise is supposed to be removed before the execution of this task, e.g. by the previous steps of the pipeline in the case of chooped-nodded observations and by the `photHighPassFilter` or similar tasks in the scan map case. The tasks projects all images onto a map with a pixel size defined using the "outputPixelsize" option. Note, that the option "calibration=True" must be set in order to properly conserve fluxes of image that are not using native pixel sizes (3.2 in the blue and 6.4 in the red). The photProjectPointSource() is specific version of photProject for the chopped/nodded point source AOT style observations.If the allInOne=1 is set then the task create a final map by combining both chop and nod positions (4 images altogether) and rotate the image so that North is up and east is left. World Coordinate System data are produced for a later FITS file generation of the final product.

```
map1 = photProject(framesnod,outputPixelsize=3.2,calTree=calTree,calibration=True)
map2 = photProjectPointSource(myframe,
allInOne=1,outputPixelsize=3.2,calTree=calTree, calibration=True)
Display(map1)
Display(map2)
product = simpleFitsWriter(map1,"filename"+str(i)".fits")
product = simpleFitsWriter(map2,"filename"+str(i)".fits")
```

Since there are three additional copies made of the final dithering corrected product, the final map contains additional images of the source, but only the one in the centre is considered to be the relevant result. Besides the final image, the task creates additional products: i) error map: distribution of errors propagated throughout the data reduction; these errors do not reflect the statistical error of the final image, but also includes systematic uncertainties. As a result, the values usually overestimate the photometric error in the final image. ii) coverage map: gives the number of detector pixels that have seen a certain logical, rebinned pixel in the final image iii) exposure map: similar to coverage map, but this time it gives the total observing time spent on each logical, rebinned pixel in the final image

You can check the result of the projection by looking at the data using the 'Display' task. Don't forget that in most cases you will have more than one slices so name your files in a way that you can retrieve them easily. (See in the example)

The difference between the two task can be seen in the two different map created in the above example

map1 = photProject() gives a de-rotated map (equatorial, N up, E left) that contains all individual frames co-added to one, showing the characteristic four point chop nod pattern. Advantage: more homogeneous coverage of the sky background for determining the background noise. Disadvantage: S/N ratio of one individual image of the target is a factor of two lower than the map2 product.

map2 = photProjectPointSource() applies a simple shift-and-add algorithm to combine all images of the target into only one in order to provide to optimised S/N ratio. The relevant results will be in the centre of the final map; the other eight copies are just an artefact of the reconstruction and should not be used. Disadvantage: The area of homogeneous coverage is relatively narrow and closely confined around the source.

5.5.2.2. Combining the final image

If you have more then one slice you need to combine them in order to get the final image.

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

# making an empty list in which we are going to store the images
images1=ArrayList()
images2=ArrayList()

for i in range(slicedFrames.numberOfScienceFrames):
ima1 = simpleFitsReader("filename"+str(i)+'.fits')
ima2 = simpleFitsReader("filename"+str(i)+".fits")
ima1.exposure = ima.coverage   # this swap is peformed because the current
exposure map is incorrect.
ima2.exposure = ima2.coverage
images1.add(ima1)
images2.add(ima2)

# mosaicking
mosaic1=MosaicTask()(images=images1,oversample=0)
mosaic2=MosaicTask()(images=images2,oversample=0)
```

With the simple fits reader you need to read all the fits files created using photProjectPointSource and/or photProject project and mosaic them using the "MosaicTask" that simply combine all the images in the "images" array

5.6. Scan Map AOR

5.6.1. Level 0.5 to Level 1

See the detailed description of the same steps in the Point-source pipeline section:

- `photMMTDeglitching` we advise to use :

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

  it gives rather good results on scan maps at medium scan speeds, if the target it not too bright (above a few hundreds of mJy), otherwise the PSF core is also deglitched. Check the exposure map to see if this is not the case. You can reduce the scales to scales=2 to overcome this problem. At high speed (60'/s) deglitching is challenging, even with scales=1, the brightest sources in the galacctic plane can be affected.

- `photRespFlatFieldCorrection`

- `photAssignRaDec`

5.6.2. Level 1 to Level 2

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

5.6.2.1. High pass filter and simple projection on the sky

photHighPassfilter

The purpose is to remove the 1/f noise. Several methods are still under investigation. At the moment the task is using a Median Filter by removing a running median from each readout. The filter box size can be set by the user (*filterbox* parameter in the scheme below). The high-pass filter is well suited for deep fields with faint point-sources, but not for fields with extended emission as all structes at scales above 2*filterbox+1 will be filtered out in this process.

```
myframe = photHighPassfilter(myframe,20)
```

The width of the high pass filter, here 20, depends on the scan speed and PSF width, The smaller the better the 1/f is filtered out, but flux of the source and PSF will be affected for too small values. For bright sources a previous photMaskFromImageHighpass has to be applied. At medium speed (20 arcsec/s) a width of 20 is a good compromise in the blue and green channel , and 30 in the red channel, which corresponds to 40/60 arcsec on the sky respectively. At high speed (60"/s) a width of 10 can be used, corresponding to a length in the sky of 1 arcmin. For deep fields, the current best values for the widths are 15 readouts in the green and 26 in red channel.

At this stage you may want to remove the turnover loops between scan legs, this can be done with the following command :

```
myframe=myframe.select(myframe.getStatus("BBID") == 215131301l
```

photProject

```
map = photProject(frames, calTree=calTree,calibration=True,outputPixelsize=2.0)
Display(map)
simpleFitsWriter(map,'filename'.fits')
```

See the detailed description in the Point-source pipeline section.

5.6.2.2. The MadMap case

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

First you need to reset the on-target flag to True, as it is unreliable in the pointing product so far

```
myframe.setStatus("OnTarget",Bool1d(myframe.dimensions[2],True))
```

and correct for the offset differences between matrices.

```
myframe = photOffsetCorr(myframe)
```

makeTodArray

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

```
tod = makeTodArray(myframe,1,0.0,optimizeOrientation=True)
```

The TOD binary data file is built with format given above and the tod product includes and the astrometry of output map using meta data keywords.

The weights are set to 0 for bad data as flagged in the mask. Dead/bad detectors (detectors which are always, or usually, bad), are not included in TOD calculations. The skypix indices are derived from the projection of each input pixel onto the output sky grid. The skypix indices are increasing integers representing the location in the sky map with good data. The skypixel indices of the output map must have some data with non-zero weights,must be continuous, must start with 0, and must be sorted with 0 first and the largest index last.

The first argument of the task is the input frames in untis of mJy/pixel, the second is the output pixel scale in relation to the PACS detector pixel size, so for scale=1 the map has square pixels with size of the PACS nominal pixel size. The third argument is the crota2 keyword (default is 0.0). If the optimizeOrientation-true is set the task will try to compute a rotation for the final map so that both image pixel coordinates (x,y) and the WCS (ra,dec) are intelligently aligned. The idea is to save on the "empty" space in the final map. It only works if the rotation is larger than 15 degrees.

runMadMap

The module `runMadMap` is the wrapper that runs the JAVA MadMap module and creates the final image.

```
map = runMadMap(tod,calTree=calTree)
Display(madmap)
```