

# **PACS Data Reduction Guide**

**issue dev, Version 3, Document Number:  
May 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 .....	12
2.1. A PACS observation .....	12
2.2. The data structure (simple version) .....	13
2.3. The spectrometer pipeline steps .....	16
2.4. The photometer pipeline steps .....	17
2.5. The Levels .....	18
3. In the Beginning is the Pipeline. <i>Spectroscopy</i> .....	20
3.1. Introduction .....	20
3.2. The sliced pipeline .....	21
3.3. Setting up to run the pipeline .....	21
3.4. Level 0 to 0.5 .....	22
3.4.1. Pipeline steps .....	23
3.4.2. Inspecting the results .....	26
3.4.3. Masks .....	36
3.5. Level 0.5 to 1 .....	40
3.5.1. Pipeline steps .....	40
3.5.2. WorkArounds .....	42
3.5.3. Inspecting the results .....	43
3.6. Level 1 to 2 .....	51
3.6.1. Pipeline steps .....	51
3.6.2. Inspecting the results .....	53
3.7. The End Of The Pipeline .....	56
4. Further topics. <i>Spectroscopy</i> .....	57
4.1. Introduction .....	57
5. In the Beginning is the Pipeline. <i>Photometry</i> .....	58
5.1. Introduction .....	58
5.2. Retrieving your ObservationContext and setting up .....	59
5.2.1. Scan map AOT .....	59
5.2.2. Point Source AOT .....	61
5.3. Level 0 to Level 0.5 .....	62
5.3.1. photFlagBadPixels .....	63
5.3.2. photFlagSaturation .....	63
5.3.3. photConvDigit2Volts .....	64
5.3.4. addUtc .....	64
5.3.5. photCorrectCrosstalk .....	64
5.3.6. photMMTDeglitching and photWTMMLDeglitching .....	65
5.3.7. convertChopper2Angle .....	65
5.3.8. photAssignRaDec .....	66
5.3.9. cleanPlateauFrames .....	66
5.4. The AOT dependent pipelines .....	67
5.5. Point Source AOR .....	67
5.5.1. Level 0.5 to Level 1 .....	67
5.5.2. Level 1 to Level 2 .....	71
5.6. Scan Map AOR .....	73
5.6.1. Level 0.5 to Level 1 .....	73
5.6.2. Level 1 to Level 2 .....	74

---

# Chapter 1. A First Quick Look at your Data

## 1.1. Introduction

Welcome to the PACS data reduction guide (*PDRG*) #. 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, moving targets, point-vs-extended sources), which are still under development, or which require longer explanations.

(Comment for the pdf version of this document: the hyphens come out as # and the links have not been included.)

This guide is aimed at those who are new to HIPE and new to PACS, and it is the first PACS data reduction document you should read. A companion to this guide is the *PACS Advanced Pipeline Topics* (or something similar), available from the HIPE help pages. 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, and the pipeline has to deal with this. 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 through the pipeline once following this guide, and then if you want to change some things you can run through it again on your own.

Once you are comfortable with working with the pipeline you may want to reduce your data following only the "ipipe" scripts#the interactive pipeline scripts that you can access via the "pipeline" menu of HIPE. Basic explanations are incorporated in these scripts, but they are not as detailed as in this data reduction guide.

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.

There are a number of other documents you should read before and along with this one. This may sound boring, but it is necessary. While the *PDRG* 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)* <LINK>. 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)* <LINK> 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*) <LINK> 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 Pipeline Topics (PAPT)* <LINK> (or something similar). This discusses the details of the pipeline tasks (all the parameters and the algorithms) and includes more advanced topics that are too long to fit in the *PDRG*.

2) The *Product Description Document (PDD)* <LINK>, a Herschel reference document, which describes the layout and terminology of the instrument-specific products. It is somewhat, but not overly, technical, and note that it describes the layers of the PACS products that we will refer to in the *PDRG*.

3) The HCSS and the *PACS User's Reference Manual*: these contain information about the tasks and classes provided in HIPE, and while written in a technical way the aim is that they can also be read by non-experts. The *PACS URM* contains the tasks written by the PACS team, the *HCSS URM* contains all the general tasks.

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 spectral AOTs or types of targets are discussed, and longer explanations are given for some pipeline-relevant information. 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 been 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 2GB of memory assigned, but for pipeline-processing (especially for large datasets) you almost certainly will need more. 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* <LINK>. 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 <LINK>. 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. Definitions: ObservationContext and pool

To start working you of course need to get hold of your data. You will either have been given it by someone, possibly 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. This container provides the associations between all the products that you need to process that single observation. Within are the actual astronomical observation#raw and automatically pipeline-reduced#and the intermediate data products that were used to process the observation by 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.

When working with your observation in HIPE you can either hold it in memory the whole time, or you can periodically save the data to disc. The latter is recommended, because if HIPE crashes you will lose everything that was held in memory. To save the data to disc you can save them as single FITS files, or#the recommended method for the more complex products#save them to a "pool".

On disc a pool is simply a directory of data organised in a Herschel-unique way, and in HIPE a pool is a pointer to this directory. HIPE needs to have data organised in a very specific way so that it can work with them, for example so the associations between the individual products of an *ObservationContext* can be made. Hence the need for pools. By default HIPE expects the pool directories to be located off of [HOME]/.hcss/lstore. The data are held in these directories individually as FITS files, but, as already said, organised in a very specific way. 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. Note that you can have more than one observation and more than on type of Herschel product contained in a pool, but it is wise not to put too many different things in a single pool.

#### Note

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 (this will be changed in later versions of HIPE). So if you pool is on disc in /Users/me/.hcss/lstore/myFirstHerschelObs, and in there you have a directory (among many others) called "herchel.ia.obs.ObservationContext", then you need to tar up the entire "myFirstHerschelObs", not just "herchel.ia.obs.ObservationContext".

### 1.3.1.2. Get the ObservationContext (and save it to pool)

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 a 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]/.hcscs/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]/.hcscs/lstore, you can use 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 is the name of the pool if you want to give it your own, unique name; `poolLocation`, a string and is the directory path (the default path is something like "/Users/me/.hcscs/lstore"); `verbose` (True or False, default is False) 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 may be useful 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/.hcscs/lstore/obsid1342111 then you only need to specify "obsid1342111" as the `poolName`. Note that saveObservation can take a while to run, especially if you ask to save the calibration tree with it.

- *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 untarred the data into and put it directly into a user-chosen, but must be pre-existing, pool. The reason you have to do this is because the directory into which the HSA data were untarred is not in the correct format for the pipeline tasks to be able to deal with. The data first have to be extracted 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 specify 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`. When you have executed this command, "myobs" will appear in the Variables panel listing, this now being name of your ObservationContext. Note that getObservation can take a while to run.

- *If your ObservationContext is in a pool already* (e.g. someone sent you an entire pool that they had been working on) you get the ObservationContext using the 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 as 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 save/getObservation tasks 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 during the observing, is to have a copy of the AOR, 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 have meta data that are relevant to that specific product. 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			
<b>Instrument:</b>	PACS	<b>RA:</b>	283.396166
<b>DEC:</b>	33.02916666666667	<b>Operational Day:</b>	149
<b>Observation ID:</b>	1342185578	<b>Observation Mode:</b>	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			

**Figure 1.1. Meta data**

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

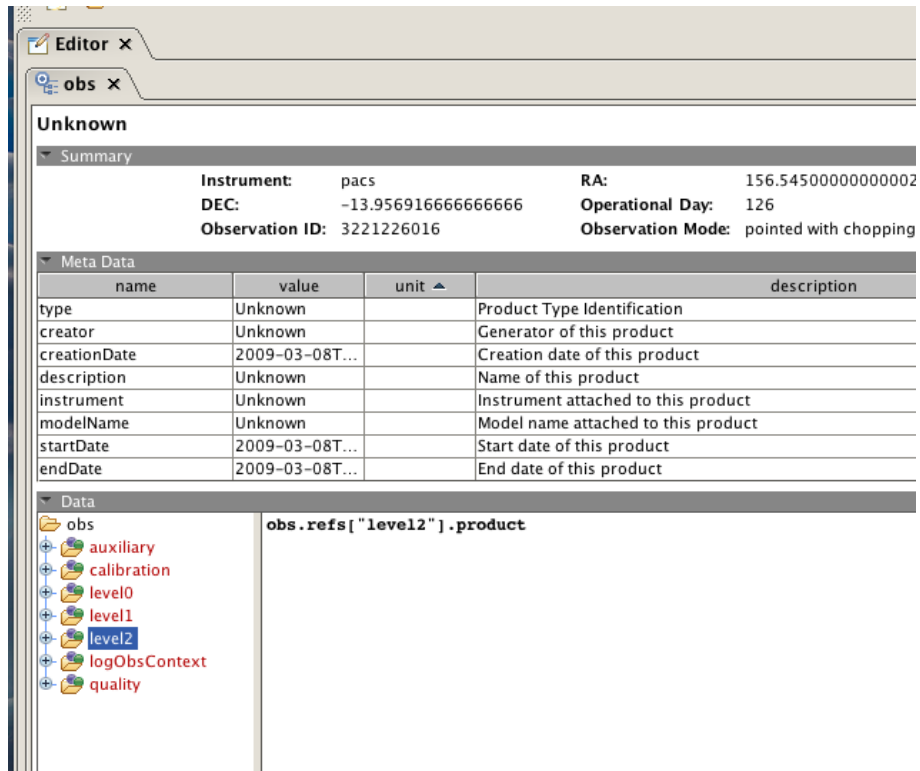
There are other panels in the Observation viewer (see the *HOG* or the *DAG* chap. 2.3). The Data panel lists the products and datasets held in the ObservationContext (black means the product has been loaded into memory, red means it has not). To the right of that is an area into which the default viewer of the currently selected product in the Data panel will open. If you click on one of the products listed 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, although this will improve during HIPE version 4.x. 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").

From later versions of HIPE 3.x onwards the automatic pipeline switched to being the so-called "slicing" pipeline. Where you have multiple settings in your observation, for example rastering or multiple spectral lines, then as the pipeline progresses these "slices" are put as separate products into lists (according to a level-specific and astronomical logic—see Chap. 3). During the middle stage of HIPE 4.x appropriate meta data were added to allow you to understand what the slices are of. These meta data are explained <LINK>.

## 1.3.2. Then: look the Level 2 products

### 1.3.2.1. Spectroscopy

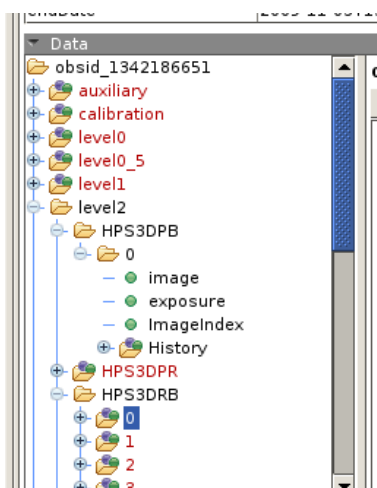
To look at what is in myobs, in your ObservationContext, open the Observation Viewer on myobs:



**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 to that level, and hence there are no cubes 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]. 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). *Note*: if the product is listed in red and you click on it to view it, you are at the same time loading it into memory. This can sometimes take a long time, *especially* if at the same time the default viewer for that product tries to open (the default viewer will load in the central panel of the Observation viewer).

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*. The HPS3DR product is the second (of three) cubest that is produced in the course of pipeline processing, HPS3DP is the final one, and the one you will be looking at here.

Exactly what is in your Level 2 depends on what type of observation you requested. You will have multiple HPS3DR cubes if your AOR included dithering/rastering/more than one spectral line, and you will have multiple HPS3DP cubes if you have more than one spectral line in your observation (dithered/rastered pointings have been combined). For each raster and spectral line you will have one cube of the *PacsRebinnedCube* class; each raster is in the final stage of the pipeline combined, per line, into the one cube of *SpectralSimpleCube* class.

*You should be aware these the cubes have been processed through a standard pipeline and untill you reduce the data yourself, you should consider them to be of browse quality only.*

In the screenshot above, showing you the datasets of the HPS3DP cube, 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 is not particularly 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. For ScanMap data this includes products with the names HPP[NIM]MAP[BIR], 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 ("M" means mad map and "N" means naive map). For PointSource AOT this includes products with the names HPP[DIP]MAP[BIR]

The HPPxxxx are, as before, *ListContexts* and the products therein are *SimpleImages*. 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.

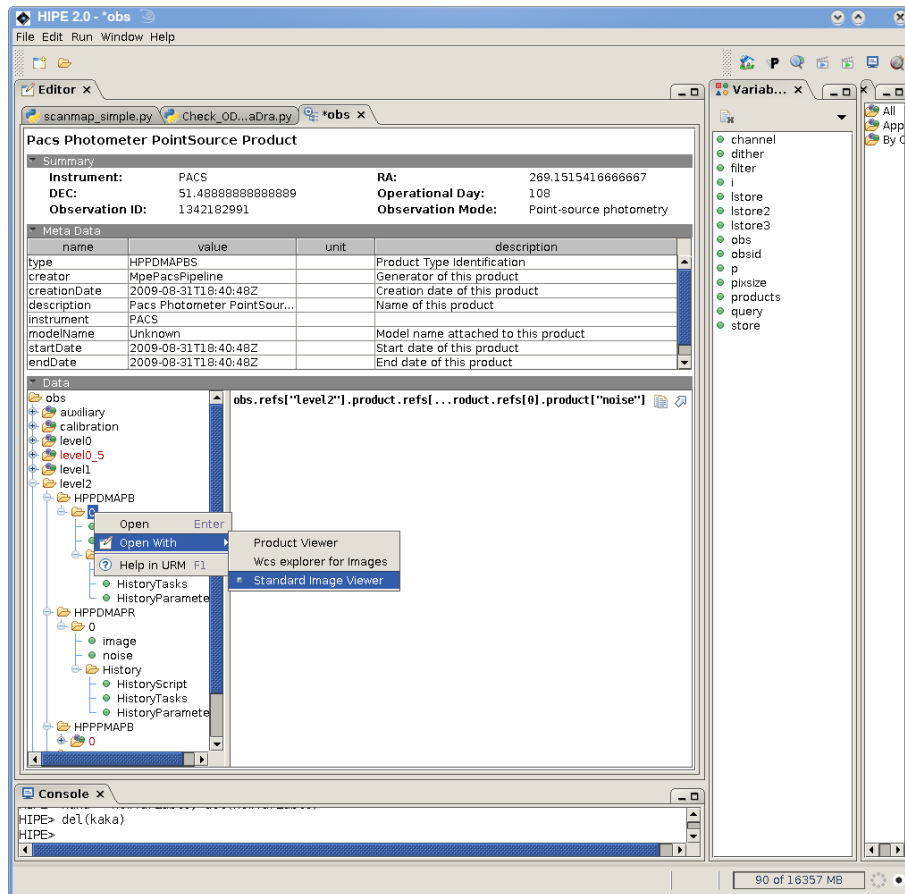
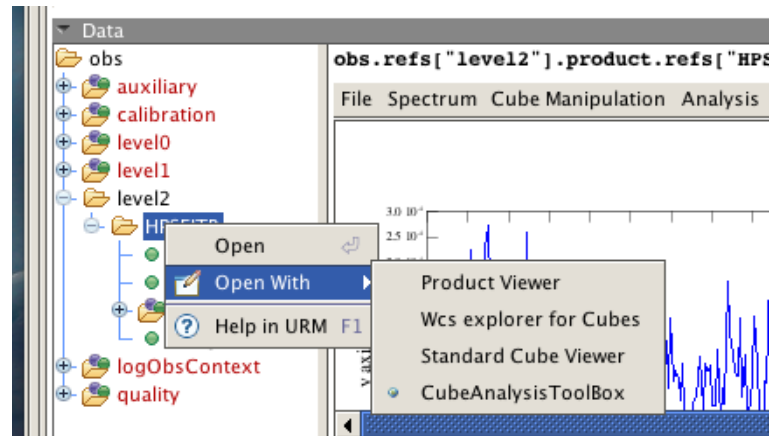


Figure 1.4. ObservationContext layout for photometry

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 individual 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. 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 cubes out of the ObservationContext first. Briefly: 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.

#### 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 the *PDRG* 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 may 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* and 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 works on the *SpectralSimpleCubes* and 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; 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.

If you want to look at the spectra in any other way, we refer to you Chap. 3. If you have a version of HIPE for which the GUIs do not work with the *PacsRebinnedCube* you will have to use a command line method. 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 PacsRebinnedCube, to plot spaxel 2,2
PlotXY(mycube.wavegrid,mycube.flux[:,2,2])
#for SpectrumSimpleCube (included for completeness) to plot spaxel 12,11
PlotXY(mycube.getWave(),mycube.getFlux(12,11))
```

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 works 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 (for spectroscopy this is very rapid to allow the telescope background to be sampled at every grating position the observation scans over); nodding of the telescope between two fields (to remove this calculated background); moving over many fields/scanning directions to make a bigger map; grating movements to sample the wavelength domain; looking at the calibration block (for flux calibration and stability monitoring). 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/scanning) 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, meanwhile 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 Her-

schel-Pacs-Spectroscopy-3Dimensional-Simple\_cube-Red (which is of class *SpectralSimpleCube*). At Level 1 we also have the HPS3D[BIR], 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 but 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 files.

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 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  $\mu\text{m}$  and 85–130  $\mu\text{m}$  and has a 32x64 pixel array. The red channel has a 130–210  $\mu\text{m}$  filter has a 16x32 pixel array. Both channels cover a field-of-view of  $\sim 1.75' \times 3.5'$ , 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  $\sim 1$  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.

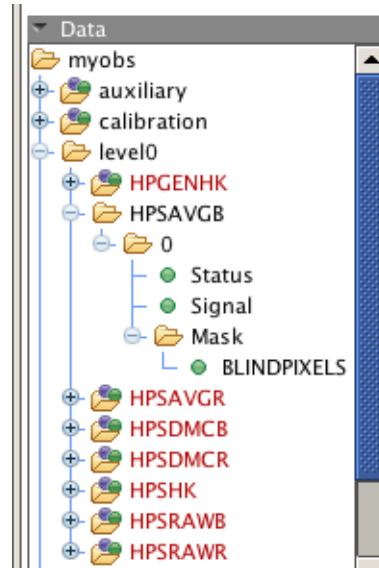
## 2.2. The data structure (simple version)

The structure of PACS data are given in better detail in the *PAPT* 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 Observation-Context. 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
```

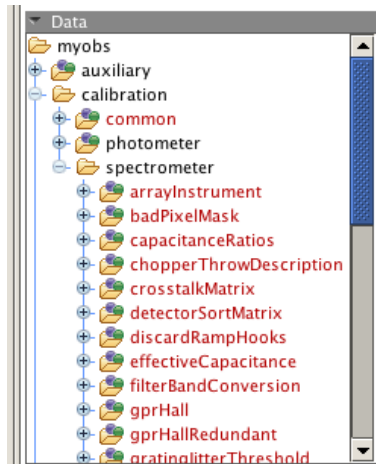
where *the first line* will produce a listing similar to the next screenshot, *the second line* produces a listing of what meta data are there plus the "directories" you can see in the screenshot above, and *the third line* shows what Level 0 products there are in your ObservationContext. Note that this 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".

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...). These are all dealt with appropriately in 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 explain more in Chap. 4.

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[HKIGENHKIENG], 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.

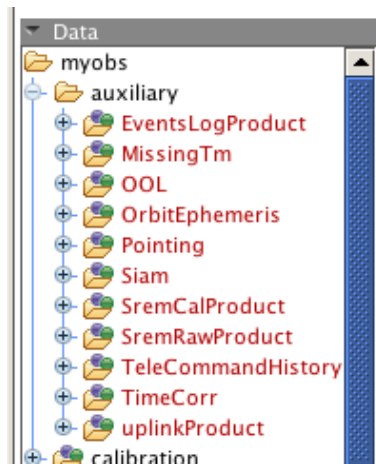
A calibration tree, containing all the information necessary to calibrate your observation, comes with your data and also with your HIPE installation (more on that later). If you click on "calibration" from the screenshot above you will see:



**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 orbit 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.

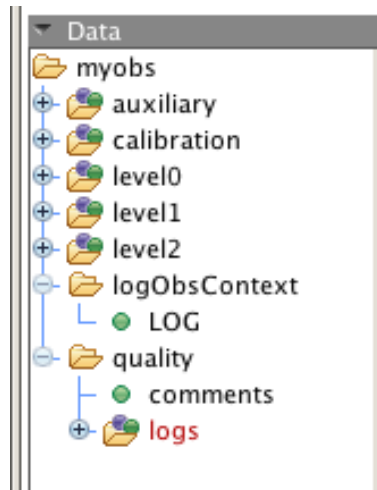


Figure 2.4. The contents of the log and quality trees

## 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; the nod A and B are combined ((older method) or they are separated into independent products (newer method) *This is under development*
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. (newer method) For chop-nod the nods A and B are combined *This is under development*
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. These can be accessed from the HIPE toolbar. The name of the ipipe script corresponds to the AOT type. The reduction for all AOTs will begin with the scripts with L05 in the name (indicating an ending at Level 0.5) and "frames" if your Level 0 product is a *Frames* or "ramps" if the Level 0 product is a *Ramps*, and then will proceed to the AOT-dependent L1 and L2 scripts. Some comments are embedded in these scripts.

During the HIPE 3.x release track the automatic pipeline switched to the so-called "slicing pipeline": the products are 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).

## 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. These can be accessed from the HIPE toolbar. The name of the ipipe script corresponds to the AOT type. Some comments are embedded in these scripts.

For very large datasets the data will probably have been sliced, that is organised in distinct and separate, but linked parts using an "astronomical" logic.

## 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 Level 2 automatically pipeline-processed data; if you are now reading this chapter we assume you wish to reprocess the data and check the intermediate stages. To this end the sections here are divided into (i) a listing of the task steps with brief explanations, followed by (ii) demonstrations for viewing the data just processed: plotting, displaying etc. The sections are separated by Level. Further information on the reduction of PACS data are given in Chap. 4, and the fullest details of the pipeline tasks are given in *PAPT*.

Throughout this chapter are included short scripts to inspect and select on the data as you go through the pipeline, although as various product-inspection GUIs are completed it will become unnecessary to do much of this scripting yourself. Nonetheless, by following the scripts you can learn not only how to handle PACS data but also a little about the scripting that can be done in HIPE.

As well as this *PDRG* you should look at the ipipe scripts#the interactive pipeline scripts. These are updated more frequently than this guide, and hence should be given greater weight where differences are found. They also form a guide to your memory if you already know how to run the pipeline and don't want to follow this more detailed *PDRG*. They can be accessed via the "pipeline" menu of the HIPE tool bar. The reduction for all AOTs will begin with the scripts with L05 in the name (indicating an ending at Level 0.5) and "frames" if your Level 0 product is a *Frames* or "ramps" if the Level 0 product is a *Ramps*, and then will proceed to the AOT-dependent L1 and L2 scripts.

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

We will first take you through the pipeline for a chop-nod observation, then other AOTs will 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.*

### Note

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

### Note

Syntax: *Ramps* and *Frames* are how the *PDRG* indicates the "class" of a data product. "Ramp" or "frame" are what we use 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 an 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. The sliced pipeline

As of HIPE continuous integration build 3.0.1307 (equivalent to the later versions of 3.x User release) the recommended pipeline is the so-called "sliced pipeline". This was designed to make it easier to reduce datasets in which more than one raster or spectral line was observed. It is also more friendly in its memory use than the previous pipeline. It uses the same "non-sliced pipeline" tasks and they have the same name with a "sliced" concated to the beginning. The data are sliced according to Level logic, which is to say by raster at Level 0.5 and by line id (spectral line id, not spectral band) and nod at Level 1. To improve the memory use still further, the tasks can run such that within the tasks, the products are read in and out of a temporary pool, rather than holding everything, always, in HIPE memory. The individual "slices" are kept together at all times#the products (the frames, ramps and cubes) are held in *ListContexts* (lists of products). Hence you can still inspect the individual slices by simply extracting them out of the list. It is the sliced pipeline that we describe here.

(The ObservationContext may itself be sliced, this follows a logic to do with filesize rather than anything astronomical.)

## 3.3. Setting up to run the pipeline

To set up the running of the pipeline, first decide if you wish to reduce the red or blue data and set the variable "camera" accordingly. Then you need to propagate the meta data of the ObservationContext to the Level 0 product (more below) and extract the Level 0 product from the ObservationContext. Next you need to gather some necessary auxiliary products. To do all of this you need to have in HIPE your ObservationContext#as explained in Chap. 1#which here is called "myobs".

These steps are:

```
camera = "blue"
pacsPropagateMetaKeywords(myobs, '0', myobs.level0)
level0 = PacsContext(myobs.level0)
timeCorr = myobs.auxiliary.timeCorrelation
pp = myobs.auxiliary.pointing
orbitEphem = myobs.auxiliary.orbitEphemeris
```

The need to propagate the meta data keywords is mainly for your own convenience. The full meta data of the ObservationContext is not repeated at each Level; if you use the Product viewer on a Level 0/0.5/1/2 product you will not see listed all the information about your observation because by default only the information relevant to that direct level is included. Since it is useful for you to be able to see, for example, the AOR parameters you programmed into the observation no matter which level product you are looking at, we recommend you run this pacsPropagateMetaKeywords task.

"level0" is the Level 0 product containing your astronomical and calibration-block data. The "pp", pointing product, is used to calculate the pointing of PACS during your observation; 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.

Next you should select from the "level0" product the *Ramps* or *Frames* product and the so-called DecMec header. Whether you begin with *Frames* or *Ramps* depends on your data—when you look at myobs (for example with the Observation viewer accessed via a right-click on "myobs" in the Variables panel) you will only have the HPSFIT (*Frames*) or HPSAVG (*Ramps*) products, not both. The DecMec product contains the position and status of the PACS mechanisms and detectors sampled at high frequency. You will also select out a raw *Ramps* product, this containing only the fully-raw data for a single pixel of the PACS detector, to be used for the saturation detection task of the pipeline.

These steps are:

```
# EITHER
slicedRamp = level0.averaged.getCamera(camera).product
```



```
# OR
slicedFrames = SlicedFrames(level0.fitted.getCamera(camera).product)
# AND
slicedRawRamp = level0.raw.getCamera(camera).product
slicedDmcHead = level0.dmc.getCamera(camera).product
```

(You can, of course, give your products any name you please: mysliceR instead of slicedRamp, for example.)

Finally, you should define the calibration tree to use with your reductions. 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. You have the choice of (i) the calibration tree that comes with the ObservationContext or (ii) that from your installation of HIPE. If your ObservationContext came from the HSA, the "caltree" is that which was used by the automatic pipeline, whereas if someone else reduced the data, the caltree will be that which they used *if* they added their caltree to the ObservationContext themselves (we show you how below). 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, and is the one we recommend you use.

The steps are:

```
# EITHER (recommended)
# get the calTree from HIPE and add it to the ObservationContext
myobs.calibration = getCalTree()
calTree = myobs.calibration
# OR
# get the calTree directly from the ObservationContext
calTree=myobs.calibration
```

You are not actually putting the caltree physically (so to speak) into myobs, rather you are placing there the pointers to the calibration files that you will use in your subsequent pipeline processing. The old versions of these calibration files are never thrown away—you will get all versions with your installation of HIPE. Hence the pointers will be valid no matter how old they are; if the pointers are newer than your HIPE installation, you will need to update HIPE to access them.

## 3.4. 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. You can usually stop a task running with the red STOP button, though it does not have an immediate effect.

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

Names of variables, such as "slicedFrames" or "myframe", are not objects themselves, rather they are pointers to the objects which are held in memory. Hence, if you copy `>myframe_save=myframe` and then change something in "myframe" (e.g. run it through a pipeline task), the same change will be made to "myframe\_save". So if you want to save a copy of a product before you run a task on it, the safest way to do this is with the syntax `>myframe_save=myframe.copy()`. The ".copy()" part of the syntax copies the data which the variable name (myframe) is pointing to, i.e. the actual data that is the product "myframe". This same .copy() method will work on slicedFrames/Cube/Ramps products, although not on any other type of *ListContext* product (slicedFrames/etc are special *ListContexts*).

This stage of the pipeline corresponds to the ipipe script with a name containing "frames" or "ramps" and "L05".

### 3.4.1. Pipeline steps

You need to import before beginning your processing:

```
from herschel.pacs.spg.pipeline import *
from herschel.pacs.signal.context import *
from herschel.pacs.cal import GetPacsCalDataTask
```

If you are working on a *slicedRamps* you will begin with:

```
slicedRamps = slicedSpecFlagSaturationRamps(slicedRamps, rawRamp = slicedRawRamp,
      calTree=calTree)
slicedRamps = slicedActivateMasks(slicedRamps, StringId([" "]), exclusive = True)
slicedFrames = slicedFitRamps(slicedRamps)
```

If you start with a *slicedFrames* you begin instead with:

```
slicedFrames = slicedSpecFlagSaturationFrames(slicedFrames, rawRamp =
      slicedRawRamp,
      calTree=calTree)
```

The task `slicedActivateMasks` we explain at the end of this section. The task `slicedSpecFlagSaturationRamps/Frames` 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. In this task you can include the "rawRamp" product if you have extracted that out of your `ObservationContext` (as explained above), but if you don't have this product you can just leave it out. `slicedFitRamps` 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`. `slicedFitRamps` 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.

The ramp fitting task changes the dimensions of the data, which you can check in the following ways:

```
print slicedRamps.refs[0].product.dimensions
print slicedFrames.refs[0].product.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. At this point in the pipeline there is likely to be only one slice in your `slicedFrames`, but later there will be more.

The syntax above is the way you extract a single frame or ramp out of the *ListContext* it is contained within (out of `slicedRamps/Frames`). The ".refs[0]" refers to the first slice in the list.

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

```
if (myobs.meta.containsKey("naifid"):
    if (myobs.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:

```
slicedFrames = slicedSpecConvDigit2VoltsPerSecFrames(slicedFrames, calTree=calTree)
slicedFrames = slicedDetectCalibrationBlock(slicedFrames)
slicedFrames = slicedSpecExtendStatus(slicedFrames, calTree=calTree)
slicedFrames = slicedAddUtc(slicedFrames, timeCorr)
slicedFrames = slicedSpecAddInstantPointing(slicedFrames, pp, calTree = calTree,
    orbitEphem = orbitEphem, horizons = None)
slicedFrames = slicedConvertChopper2Angle(slicedFrames, calTree=calTree)
slicedFrames = slicedSpecAssignRaDec(slicedFrames, calTree=calTree)
slicedFrames = slicedWaveCalc(slicedFrames, calTree=calTree)
slicedFrames = slicedSpecCorrectHerschelVelocity(slicedFrames, orbitEphem, pp)
slicedFrames = findBlocks(slicedFrames, calTree = calTree)
slicedFrames = slicedSpecFlagBadPixelsFrames(slicedFrames, calTree=calTree)
```

*Note: the line wrap-around for the first command is just there so this line fits on the pdf printed page. On the command line or in a script you'd write it all on one line.*

And to explain this all:

- In the order listed these tasks do the following: 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.
- 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 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. At present it corrects the wavelengths computed by waveCalc.
- The task slicedSpecFlagBadPixelsFrames creates the masks BADPIXELS and NOISYPIXELS, the former being truly bad pixels and the latter an indication that elevated noise was found from instrument tests done in-orbit. If you ran slicedFitRamps 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. The SATURATION mask is created where your AVG (*Ramps*) or FIT (*Frames*) 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 necessarily 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 Status table contains information about the status of Herschel and PACS during the observation, and is added to as the data processing proceeds (here by the task slicedSpecExtendStatus). You can look at these information as a dataset or plot. Use the Observation viewer on slicedFrames to see

its contents; click on any of the individual products therein to see its dataset listing; and then on the Status dataset, right-click select a viewer: a plot, which however does not show all the parameters; and a dataset, i.e. a table. The BlockTable (the task findBlocks) is an organisation table showing the data "blocks", that is the observation organised by instrument status (raster, nod, grating scan direction...). You can look at this also via the Dataset viewer. We tell you more about the Status and BlockTable in Chap. 4.

The next step is to slice the data according to the Level 0.5 logic, which is by raster. If there is only one raster pointing in your observation, there will be only one science slice created. In addition, the calibration block(s) (usually taken at the beginning of the observation) is sliced out and made into a slice(s) in the *ListContext*. After slicing the astronomical data you will do the same to the dmcHead product, telling the task to take the same slicing organisation as slicedFrames has. Finally you run two masking tasks.

```
slicedFrames = pacsSliceContext(slicedFrames, level='0.5')
slicedDmcHead = pacsSliceContextByReference(slicedDmcHead, slicedFrames)
slicedFrames = slicedFlagChopMoveFrames(slicedFrames,
    dmcHead = slicedDmcHead, calTree=calTree)
# in older versions of the pipeline this first flagging task
# was called slicedCleanPlateauFrames
slicedFrames = slicedFlagGratMoveFrames(slicedFrames,
    dmcHead = slicedDmcHead, calTree=calTree)
```

The masking tasks here flag data taken while the chopper and grating were moving. As data is taken continuously#it does not stop for chopping, grating movements, nodding, or even rastering#data taken during movements should be masked out. These masking tasks use automatic criteria, mostly taken from the calibration tree, to determine if a readout needs to be flagged as potentially bad. 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 UNCLEANCHOP and GRATMOVE.

Now, what is this slicedActivateMasks 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. 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 slicedActivateMasks does.

This marks the end of Level 0.5, up to which the data reduction is AOT independent. Before continuing with the rest of the pipeline you may wish to save the slicedFrames back to the *ObservationContext* you started from (myobs here), and save that to a pool (to disc). This you would do so that when you work on myobs later, you can see the full product—the meta data, each Level, all the auxiliary data, etc. When saving the new Level 0.5 product you have just created back to myobs, you will be overwriting the Level 0.5 product that was there before, i.e. that which the HSA produced via the automatic pipeline. If you wish to compare the HSA result with yours, you can always copy myobs first:

```
myobs_hsa = myobs.copy()
```

To save the product you do the following: first

```
if myobs.level0_5 == None:
    pc0_5 = PacsContext()
else:
    pc0_5 = PacsContext(obs.level0_5)
```

If there is already a Level 0.5 in myobs then it is extracted out (in the next stage to be added to) and given the variable name pc0\_5 and class *PacsContext*. If there is nothing presently in Level 0.5 then a new, empty variable, of class *PacsContext* and called pc0\_5 is created. If your data were already pipeline-processed then a Level 0.5 will already exist; there will be two of these, one for the red camera and one for the blue. If you add new blue and red camera Level 0.5 products then you overwrite what was there before. If you want to be sure to clear the level first you can use the command.

```
obs.level0_5 = None
```

In the next command you will put your newly-created slicedFrames (for the red or blue camera, whichever you have been working on) into the 0.5 level of myobs, and if for either camera the level is already filled, then what is there already will be overwritten:

```
# add the pipeline processed result to the PacsContext; this is camera-specific
pc0_5.spectrometer.fitted.getCamera(camera).product = slicedFrames
pc0_5.spectrometer.dmc.getCamera(camera).product = slicedDmcHead
```

where the variable "camera" was defined in the beginning of this chapter ("red" or "blue").

Finally you can copy some of the meta data and slicing logic information to the Level 0.5 (so it is there for you if you later extract out that level to look at), and then place pc0\_5 in to myobs:

```
pacPropagateMetaKeywords(myobs, '0.5', pc0_5)
addSliceMetaData(pc0_5, '0.5')
myobs.level0_5 = pc0_5
```

To save myobs to pool you can use the task saveObservation, explained in Chp. 1, although note that if you save the *ObservationContext* to the same pool you took it from, when you get it from pool next time you will get this new version, not the old one.

You can also save just the slicedFrames or the PacsContext pc0\_5 to pool: this is explained in Chap. 4.

Next we will tell you how to save and inspect the products you have just created.

### 3.4.2. 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 before 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. (These latter would be mainly to find out what happens during an observation, they are not really a diagnostic of the pipeline.) 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.

To work out what a particular slice in a slicedFrames/Ramps corresponds to you need to look at the BlockTable—each individual *Frames* product in slicedFrames has a BlockTable and there is also a MasterBlockTable for slicedFrames. As the slicing proceeds, more *Frames* slices are added to slicedFrames. The first slice will usually be the calibration block, which is separate from the others and as, in our current AOT implementation, only one calibration block is done at the beginning of each observation, in this case only one calibration slice will be created. You can check on this with the command

```
print slicedFrames.getNumberOfCalFrames()
print slicedFrames.getNumberOfFrames()
print slicedFrames.getNumberOfScienceFrames()
```

to see the number of calibration, all, and science frames in slicedFrames. To extract out a particular slice you can use

```
myframe=slicedFrames.getScience(i)
```

where *i* is the *i*th science frame (not, note, the *i*th frame) you want to extract out (remembering that 0, not 1, is the 1st). This method is an alternative to the

```
myframe=slicedFrames.refs[i].product
```

introduced before and which simply extracts sliced number *i* from *slicedFrames*. (There are many methods you can use on your *slicedFrames* product to interact with it—these are described in the *DRM* entry of *slicedFrames* <LINK>.) To understand what each slice in your frame corresponds to a method will be implimented to give you a listing, but at present you need to inspect the *MasterBlockTable*. See Chap. 4.

The instructions here expect you to be working on a single frame/slice of a *slicedFrames*.

To know the dimensions of your frame (or ramps), use:

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

#### Tip

You should use the *.copy()* method (first introduced in towards the end of Sec. 3.4.1) to make a secure copy of a product, and that you can do this for single *Frames* products and *slicedFrames* products. You do this because *>slicedFrames\_save=slicedFrames* does not actually copy *slicedFrames*, it just creates a new pointer to it, and so anything you change in *slicedFrames* will also change in *slicedFrames\_save*. We add here that the same holds to the product placed in *slicedFrames*, i.e. the individual *Frames* slices. For example, doing the following on a single frame slice:

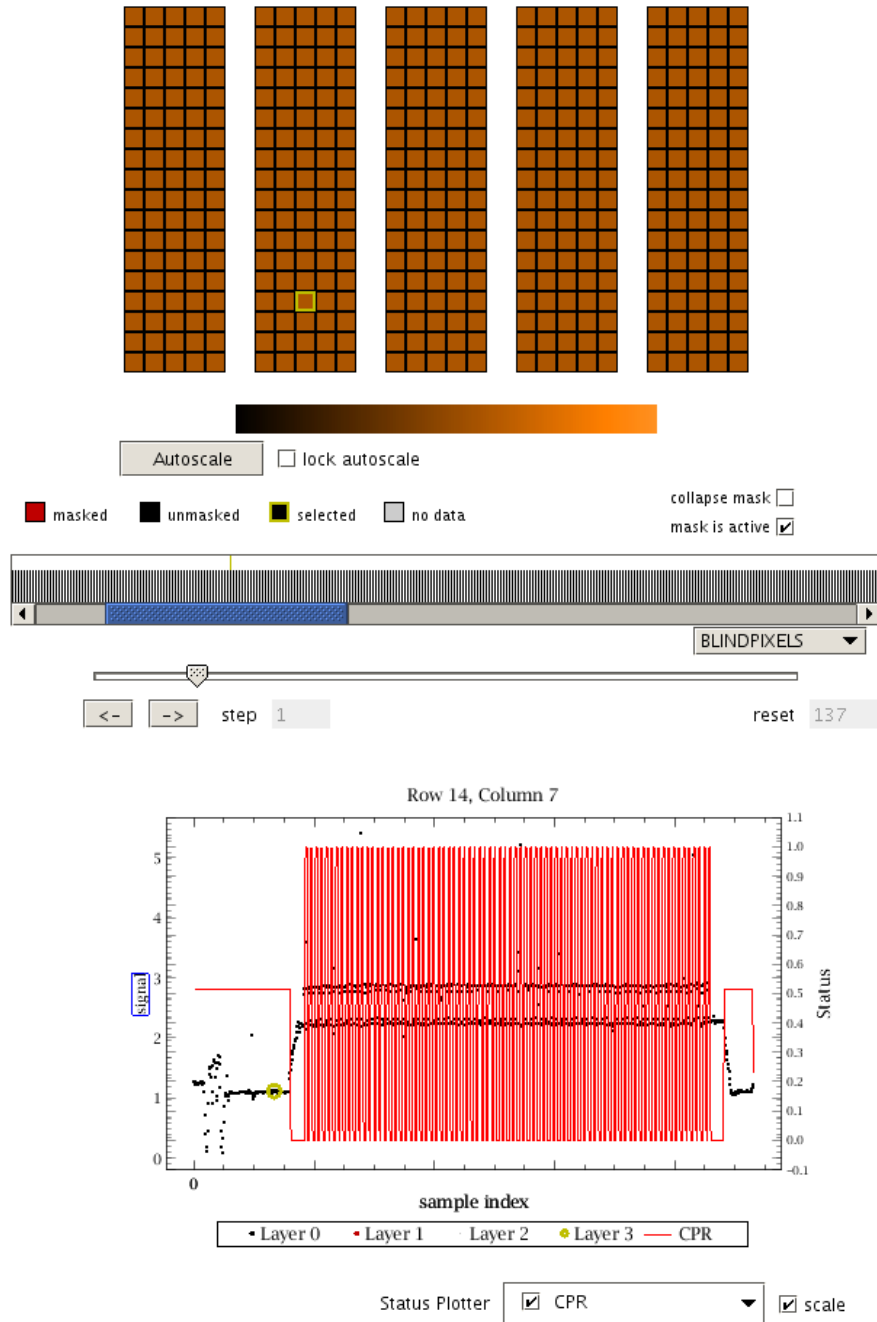
```
signal=frameSlice1.getSignal(8,12)
signal=2*signal
frameSlice1.setSignal(8,12,signal)
```

will not only change *frameSlice1*, but you will find that the first slice in *slicedFrames* has also been changed. So you need to do the follow to the product (a single *Frames* or a *slicedFrames*) before working on it:

```
frameSlice1=framesSlice.copy()
slicedFrames1=slicedFrames.copy()
```

### 3.4.2.1. GUIs

There is a nice PACS product viewer (right click on a single *Frames* product to access it) which shows the detector layout and the spectra in the pixels. In this way you can look at the spectra, in the form of signal vs readout (#time), for each pixel, one by one. It allows you to plot flagged data and overplot Status values on the signal datapoints. It is the same as the *MaskViewer*, which is explained here in 3.4.3.1. A screenshot of the PACS product viewer:



**Figure 3.1. The PACS Product Viewer**

The top shows the PACS detector, from which pixels are selectable for viewing. The 25 columns are the 25 spaxels, and in each are 18 pixels containing the spectral data, although pixel row 0 and 17 contain no astronomical data. The bottom is the plot of the data from the selected pixel, and in the middle (black stripy block) is an X-axis scroll bar. A yellow circle on the left of in the plot indicates where you are in the scroll bar. Masked data are plotted as red dots, otherwise the datapoints are black dots. You can select which masks to plots via the drop-down box on the right of the central scroll bar ("BLINDPIXELS" in the screenshot). You can also chose various Status values to overplot on datapoints, via a drop-down menu be the bottom of the plot (here we have plotted CPR in red over the datapoints in black). You can zoom in on the plot by selecting a window within the plot panel, and you can obtain a menu of actions via a right-click inside the plot.

The Status table is introduced in 3.4.2.2 and detailed further in Chap. 4, so read there to know more about what Status values you may want to look at. In 3.4.3 more is explained about the masks, so read that also.

Note that the PACS product viewer will not plot wavelength on the X-axis.

As of HIPE continuous build version 4.0.461 (User release 4.x) the SpectrumExplorer GUI (first mentioned in Chap. 1) can be used to inspect *Frames* products (the frames contained in your *slicedFrames*) in a friendly way. This will allow you to overplot spectra from different pixels and different time-frames of your data, and in this way you can inspect the results of the pipeline processing. To see if the SE will work on your data you need to extract individual *Frames* slices out of *slicedFrames* and run it on those. Use of the SE is described in <LINK>. If it is not offered for your frames then you will need to inspect your spectra in a more manual way; how do to this is described below. In these next sections we will also explain a little about what you will be looking for when you inspect your *Frames*, and we will also explain how to plot the pointing; hence it is worth reading the next sections even if you use the SE to inspect your *slicedFrames*.

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

The Status is attached to your *Frames* or *Ramps* product (not the *slicedFrames* product, but the individual *Frames* "slices" therein) 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 single 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 telescope background and also remove the dark current. This chopping happens with a very high frequency (as it much sample at each grating step). 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.



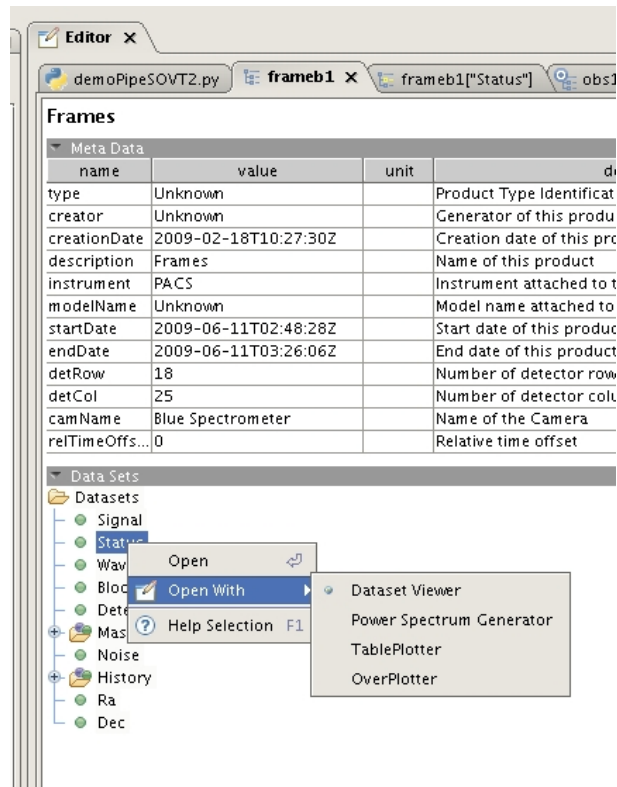


Figure 3.2. Viewers for the Status

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

### 3.4.2.3. 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 vs 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 afterwards (more data, annotations...).

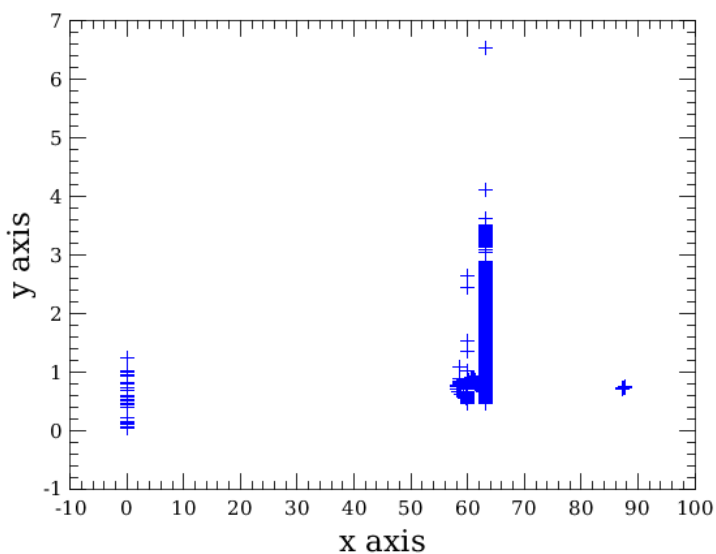
To plot a spectrum, that is signal versus wavelength, for pixel 8,12 after you have run the slicedWave-  
Calc task,

```
p = PlotXY(myframe.getWave(8,12),myframe.getSignal(8,12),
          titleText="title",line=0)
p.xaxis.title.text="Wavelength [ $\mu\text{m}$ ]"
p.yaxis.title.text="Signal [Volt/s]"
```

Here you are using the "methods" `.getWave(8,12)` and `.getSignal(8,12)` to extract out the wavelength and signal arrays for pixel 8,12 (over a range of 18,25, where 0,: and 17,: do not contain useful data, and where the Y and X dimensions are flipped in the call). You can find more information about the methods for the *Frames* (or *Ramps*) product by looking in the *DRM* <LINK>.

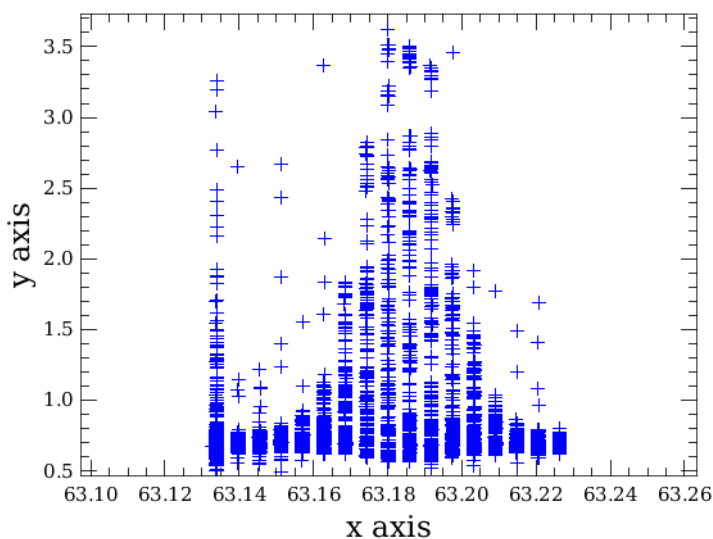
A method is a set of commands that you can call upon for an object (`myframe`) of a class (*Frames*), these commands will do something to the object you specify—in this case it extracts out the signal or wavelengths from the frame. Methods can have any number of parameters you need to specify, in this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

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 (taken 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 a later 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" level. So, if you have a line scan and you see this:



**Figure 3.3. Level 0.5 line scan spectrum: entire dataset**

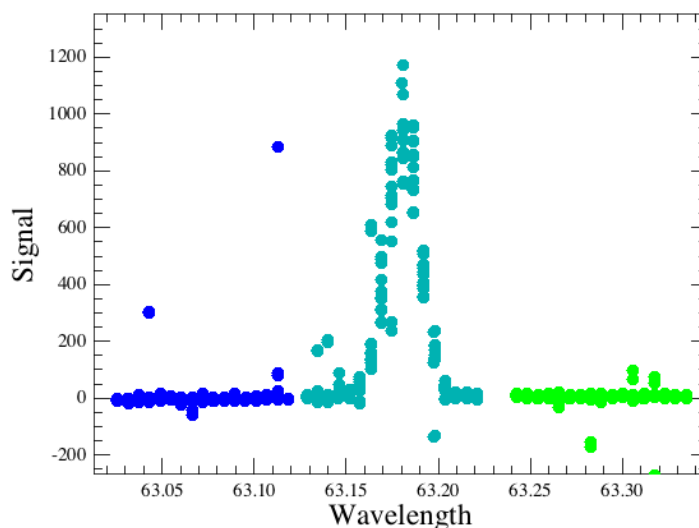
then zoom in on the wavelength you requested in your AOR, and you should see this:



**Figure 3.4. Level 0.5 line scan spectrum: zoom**

In this spectrum 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 pipeline-processed.

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 3.5. 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 plot 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 2D 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 [ $\mu\text{m}$ ]"
p.yaxis.title.text="Signal [Volt/s]"
```

Here we have changed the way to extract the wavelength and signal arrays because the method use before does not allow you to specify a range of pixels to plot. The `.wave` and `.signal` syntax calls upon the "wave" and "signal" datasets of `myframe`.

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.

### 3.4.2.4. Plotting the spectrum to understand what you have: 2

You can next check the movement of the instrument during your observation, and 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)
```

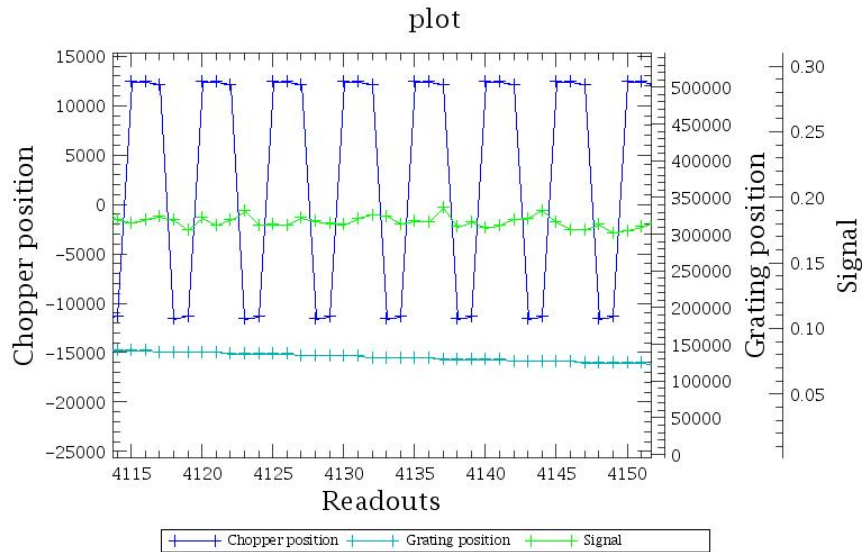
The Y-range is by default the max to the min, so you would not need to specify those if you wanted to plot from max to min. The "l1." commands create new layers which are then added to the plot with `p.addLayer()`.

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 3.6.** 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. Your data should show a similar arrangement of movements (but with different samplings).

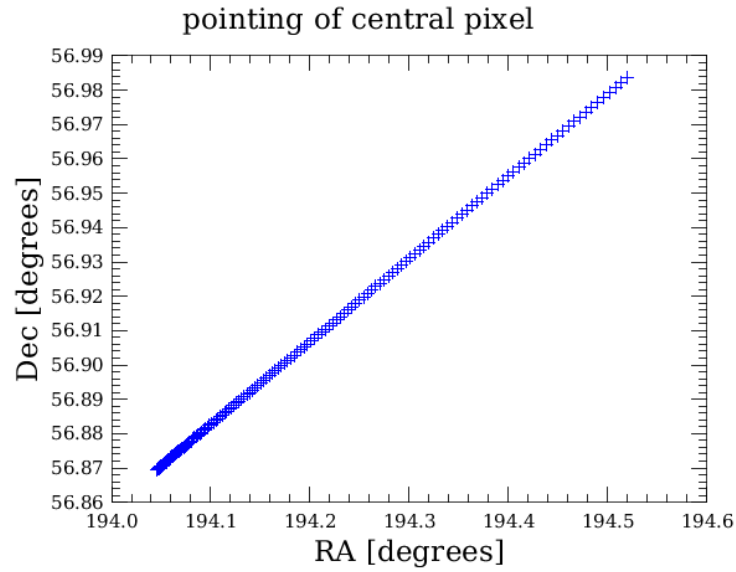
Using the same set of instructions you can chose yourself what to plot against what from the Status table and with the signal. You may, for example, wish to check whether the readouts (signals) taken during a moving chopper have been flagged correctly in the UNCLEANCHOP mask. This is covered below.

### 3.4.2.5. 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 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:

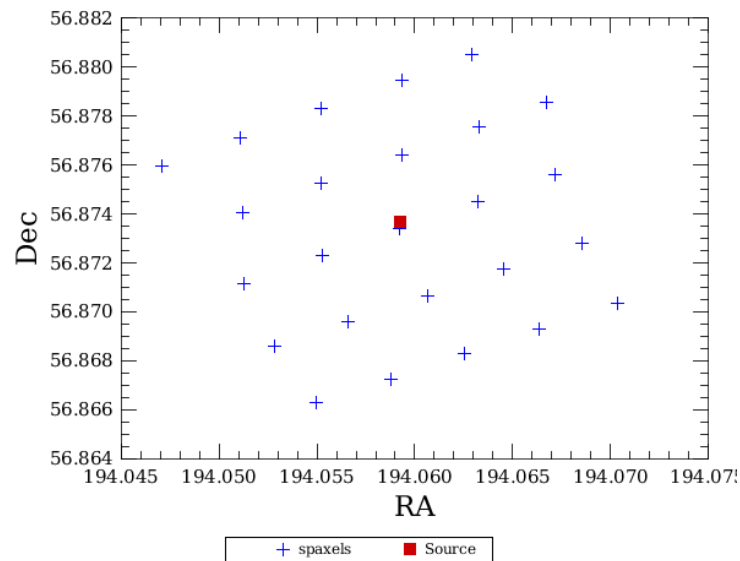


**Figure 3.7. Movement of PACS during an observation**

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=slicedFrames.meta["ra"].value
srcDec=slicedFrames.meta["dec"].value
# note, "ra" is in "myobs" as well as "slicedFrames"
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 3.8. Pointing of the IFU and the source position. This shows you what the on-sky layout 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 or the slicedFrames (if it is in there), 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 do this you need to select out the relevant slices from slicedFrames that corresponds to each pointing, and then you can use the same commands as written above to plot the pointings of each subframe. You can do the same if you wish to compare the pointing for different spectral line slices. See Sec. 3.5.3.1.5.

More information about the spatial arrangement of the spaxels is given in Chap. 4.

### 3.4.2.6. 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])
```

and when you zoom in you will see a 2D image: we are looking at the signal part of frame. Here we display all Y and X 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.

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.

### 3.4.3. Masks

First we introduce the MaskViewer GUI, and then explain 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 would 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? The MaskViewer plots, 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.

### 3.4.3.1. MaskViewer GUI

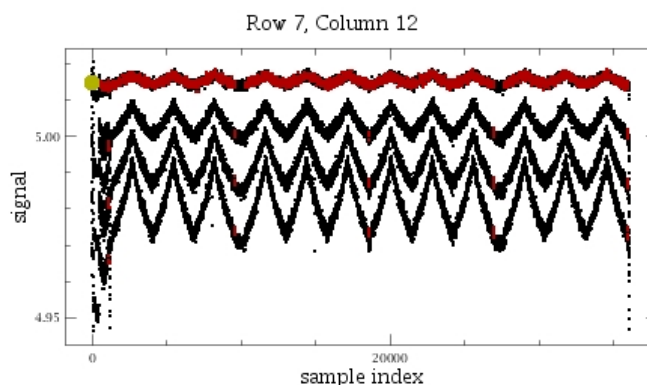
The MaskViewer the same as the PACS product viewer and it was briefly introduced in Sec. 3.4.2.1. With the MaskViewer you can see which data were flagged in the masking tasks *and* at the same time you also get to see the data themselves. You can also see which masks are active, although this can also 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). To call it up, here on a *Ramps* product, type:

```
# only necessary once per HIPE session:
from herschel.pacs.signal import MaskViewer
# and then
MaskViewer(myramp)
```

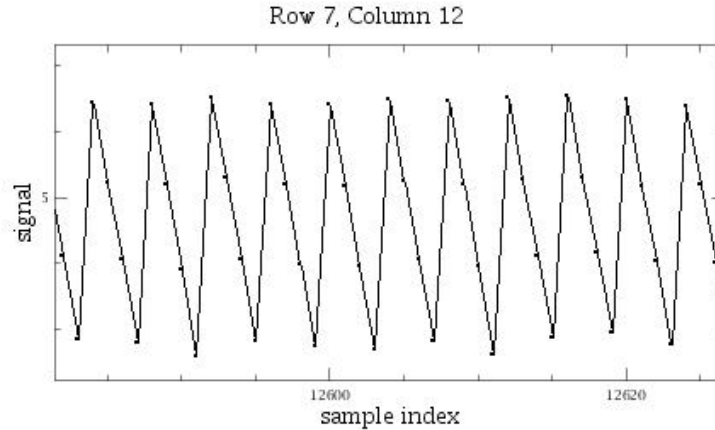
A screenshot of the MaskViewer was shown in Sec. 3.4.2.1. In the MaskViewer the plots you see, when you look at any pixel, are the actual detector signal time-line for the selected pixel#"signal" vs "sample index" (never wavelength). If you use it on a *Ramps* product, as in the example above, then you will see in the plot 4 lines of datapoints following a zig-zag pattern, as shown below. 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 an astronomical source, moving back and forth in wavelength leading to an up-down pattern in the data-line (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 3.9.** The MaskViewer plot

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 3.10.** 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.)

### 3.4.3.2. 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 datasets 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))
```

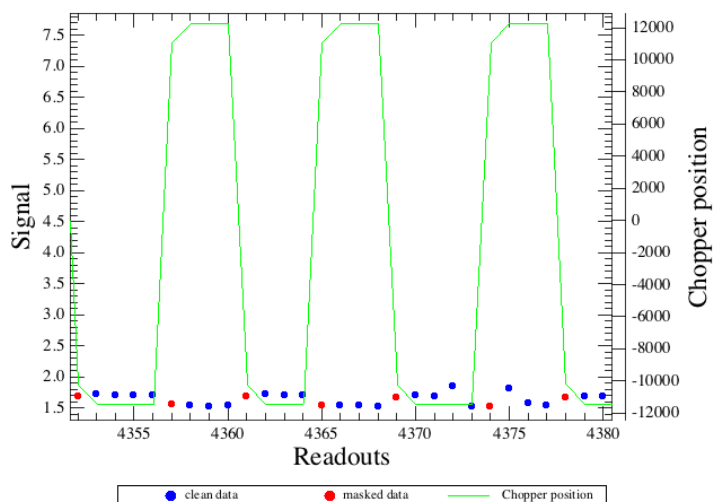
What does this do?

- 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 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.

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- or time-array position rather than wavelength). To check, for example, if the signals masked as UNCLEANCHOP were taken while the chopper was moving, you will plot the masked and unmasked signal and the Status CPR 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 produce a plot such as this:



**Figure 3.11. 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 the data shown here 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.

## 3.5. Level 0.5 to 1

In this part of the pipeline you will do a further slicing and do the flux calibration, ending with the first of the three cubes created by the pipeline. One issue that you need to consider after you have run this pipeline is to do with the nodding part of the chop-nod AOT. As discussed in the *Observer's Manual* and here in Chap. 2, the chop-nod AOT sees PACS nodding between an A and a B pointing, and off- and on-chop positions within each nod pointing. After you have subtracted the chop-off from the chop-on, to remove the telescope background, you will want to combine the nods. Unfortunately, after launch we discovered that the nod A and B do not align perfectly, they have very slightly different FoVs. The pipeline tasks to deal with this are still evolving, mainly in the Level 1 to 2 part of the pipeline. In this, Level 0.5 to 1, part of the pipeline, what you may want to look at is the spectra from the same spaxels of nod A and B, and their sky relative pointings (Sec. 3.5.3.1.4). The smaller the offset in FoV between the nods, the more similar will be the spectra from the same spaxels and the easier will be combining the nods. The offset is smallest for the smallest chop throw (<LINK>). Whether this is an issue at all also depends on your science programme; generally for point sources it should not matter much. See Chap. 4 for more information on these nod offsets, and for a discussion of what the ways of tackling it are. For this part of the pipeline, the nod issue does not affect your running the pipeline, it will only add one extra thing you would want to check after having run the pipeline.

### 3.5.1. Pipeline steps

This stage of the pipeline corresponds to the ipipe script with the words "L1" and "chop nod" in the name. If you are beginning from this level then follow the next 3 script boxes. Otherwise skip to the "You need to import...".

Extract out of the ObservationContext, myobs, the level0\_5 product:

```
level0_5 = PacsContext(myobs.level0_5)
```

and, as before, you should also define the variables camera and calTree. You can either use the calTree that is in myobs, for example if you reduced the Level 0.5 yourself and wish to continue from there, or you can get it from HIPE:

```
# get the calTree from myobs
calTree=myobs.calibration
# get the calTree from HIPE
calTree=getCalTree()
# put that HIPE calTree into myobs, if you have not already done so
myobs.calibration=getCalTree()
```

Next you extract the slicedFrames from level0\_5:

```
slicedFrames = level0_5.fitted.getCamera(camera).product
```

You need to import before running this part of the pipeline:

```
from herschel.pacs.spg.pipeline import *
from herschel.pacs.signal.context import *
from herschel.pacs.cal import GetPacsCalDataTask
from herschel.pacs.spg import SlicingRule
```

If you are continuing from the Level 0 to 0.5 reductions from before then you begin from this next task, where you slice slicedFrames further, this time according to line id logic. If you have more than

one spectral line defined in your AOT, extra slices are here created. Spectral ranges (for range or SED AOTs) do not get sliced. It is also necessary to slice by nod. These two slicings are done in one command: you end up with a slice per line and nod, plus the (usually) first slice being the calibration block. Remember that any repeats on the grating scan will be contained within each single slice; the slice numbers are listed in the MasterBlockTable in the column "FramesNo".

```
rules = [SlicingRule("NoddingPosition",1)]
slicedFrames = pacsSliceContext(slicedFrames, slicingRules = rules, level='1')
```

The slicing by line id is done because you have specified that the "level" to slice is 1, where the slicing logic is line id. The extra slicing by nod is commanded via the slicingRules parameter. The "rules" have been set by specifying the column name of the MasterBlockTable that you want to slice on (see Chap. 4 for more on BlockTables) and the "1" means you want to have 1 nod value to enter each new slice. You can tell what sequence you have after slicing by looking at the MasterBlockTable, either via the Product viewer, or on the command line this would be:

```
print slicedFrames["MasterBlockTable"]["NoddingPosition"]
# in the output, 1 = A and 2 = B
```

Now you can run the pipeline:

```
slicedFrames = slicedActivateMasks(slicedFrames, StringId([" "]), exclusive = True)
slicedFrames = slicedSpecFlagGlitchFramesQTest(slicedFrames)
# optional...
slicedFrames = slicedActivateMasks(slicedFrames, StringId(["UNCLEANCHOP",
"GRATMOVE", "GLITCH"]), exclusive = True)
slicedFrames = slicedSpecEstimateNoise(slicedFrames)
# ... back to required
slicedFrames = slicedSpecCorrectSignalNonLinearities(slicedFrames, calTree=calTree)
slicedFrames = slicedConvertSignal2StandardCap(slicedFrames, calTree=calTree)
slicedFrames = slicedSpecDiffChop(slicedFrames, scical = "sci", keepall = False)
slicedFrames = slicedRsrCal(slicedFrames, calTree=calTree)
slicedFrames = slicedSpecRespCal(slicedFrames, calTree = calTree)
slicedCubes = slicedSpecFrames2PacsCube(slicedFrames)
```

- 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 deactivate all masks (i.e. it is as if there were no masks at all); the statistical test runs a lot better this way); Optional task to estimate noise; Correct the signal slope (V/s) for non-linearities; 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); Subtract the off chops from the on chops to remove the sky 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. Turn the sliced frame into a sliced cube 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).
- The task to estimate the noise is an information task only, and hence not necessary for the pipeline. Its workings are given in the *PAPT*: basically it determines the signal error at Level 1, and all errors will be propagated to the end of the pipeline, including those errors that are also given in the calibration files.
- The ipipe scripts may have an extra set of tasks listed, including "specDiffCs". At present the product of this task is not used (slicedSpecFitSignalDrift), hence we do not describe it here.
- The task slicedSpecFrames2PacsCube simply rearranges the data of the input frame, from 18x25 pixels to 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 readouts you will need to select and plot in the same way as has been described previously, that is select only unmasked indices to plot. The individual slices of slicedCubes are *PacsCubes*.

- 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 datapoints—as stated before, you could write your own glitch-detection algorithm and use that instead. A later "outliers" pipeline masking task is also performed.
- The task `slicedSpecRespCal` 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 (May 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 the "Special issues" section below.
- The tasks here that change the state of the data are `slicedSpecCorrectSignalNonLinearities` (except for the blue dataset if you began the pipeline on a *Frames*), `slicedSpecDiffChop` (see Sec. 3.5.3.1.4), `slicedRsrCal`, and `slicedSpecRespCal` (see Sec. 3.5.2.1).

The glitch detection task by default it works on chopped data, and as this section is for a chop-nod AOT then you want the default case. (If you observed with a staring AOT then you need to add the parameter "splitChopPos=False".) It has been tested on chopped and non-chopped data. 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: but extensive testing has shown 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. (A later "outliers" pipeline masking task is also performed.)

Finally you can save your `slicedCubes` back into the *ObservationContext* you began with, as first explained in 3.4.1, and we repeat the commands here:

```
if myobs.level1 == None:
    pc1 = PacsContext()
else:
    pc1 = PacsContext(myobs.level1)

pc1.spectrometer.fitted.getCamera(camera).product = slicedFrames
pc1.spectrometer.cube.getCamera(camera).product = slicedCubes

pacsPropagateMetaKeywords(myobs, '1', pc1)
addSliceMetaData(pc1, '1')
myobs.level1 = pc1
```

## 3.5.2. WorkArounds

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.

### 3.5.2.1. Flatfielding

Flatfielding in the context of PACS spectroscopy means making it so that the gain of each pixel has been corrected for. To tell if your spectra have been flatfielded properly, you can overplot the spectra of the 16 individual data-containing pixels of each module (i.e. pixels 1:17 of each of the 25 modules): the continua of the spectra within a single module should sit on top of each other. If they do not it means that the flatfielding, and specifically the task `specRespCal`, has not done a perfect job. This task is still being worked on, and in the version of HIPE you are working with it may still not be correct. See <LINK> for updates.

To correct for this a function/tool has been written and placed in <LINK>. Contact the Herschel help desk to know how to import it (the location in the build had not been decided on by the time of writing

this guide). This file contains a number of functions which were written for the Dec. 2009 Madrid data reduction workshop. To run the flatfielding function, on a single *Frames* product ("myframe"), you type:

```
# set the wavelength of your spectral line
line=63.1
# set up
useMasks = String1d(["SATURATION", "GLITCH", "BADPIXELS"])
doNodShift = 1
# run the flatfielding task
myframe = toolsMedianOffsetCorrection(myframe,line,doNodShift,useMasks)
```

This task loops over all pixels of each module, identifies the edge continuum regions of your spectra based on the line wavelength you give it (hence assuming lines that are not as broad as the wavelength range and that no features are at the edge of the spectra), computes the continuum median and subtracts it, and then (for `doNodShift = 1`) "adds back" the continuum level by adding to each pixel the median of the (subtracted) medians. It deals with the data of each nod separately (although as you are running the slicing pipeline, each frame you read into the task will only contain data from one nod). To identify what the wavelength is of your slice, you can either plot the spectrum or look at the "IWave" entry in the meta data of the frame (`>print frame.meta["IWave"]`). Remember that this entry refers to the wavelength that was entered in HSpot, and so is dependent on what the proposer wrote. *NOTE*: this task has been written only for use on spectral line AOTs. If you have SED or range AOTs you should contact the Herschel help desk to ask for help from the PACS team.

### 3.5.3. Inspecting the results

In the Level 1 pipeline the *Frames* or *slicedFrames* (a list of *Frames*) is turned into a *PacsCube* or *slicedCube* (a list of *PacsCubes*). The *PacsCube* contains, in each spaxel, all the spectra lined up in time order; they have not yet been mathematically combined. This is done in the next stage of the pipeline, and before that a further glitch flagging task is run.

To inspect a *PacsCube* or *Frames* slice you need to first extract it out of *slicedFrames/Cubes*. The method for doing this for the *slicedFrames* product was explained before.

```
# frame
myframe=slicedFrames.getScience(i)
# cube
mycube=slicedCubes.getSlice(i)
```

The difference in the syntax is because *slicedFrames* will contain at least one calibration frame along with the science frames, but the *slicedCube* does not. To get a listing of what each slice is, you need to inspect the *MasterBlockTable* for *slicedFrames* (or use the method to get a listing, when that method is implemented). For the *slicedCubes* there is currently no *MasterBlockTable* but the order of the cubes in the *slicedCubes* is the same as the order of the frames in the *slicedFrames*. In Chap. 4 we explain the *BlockTable* entries.

It is possible that for each frame or cube you will want to look at these individual spectra in each spaxel, to check their consistency (remember, there are 16 spectra, from 16 pixels, per cube spaxel). Previously we introduced two GUIs that can be used to inspect *Frames* products (and explained the alternative manual methods if you have an older version of HIPE in which these GUIs do not work on *Frames*). For the cube created in this part of the pipeline, which has class *PacsCube*, the *SpectrumExplorer* will work, the PACS product viewer may not. Use of the SE is described in <LINK>. If it is not offered for your frames or cubes then you will need to inspect your spectra in a more manual way. In these next sections we will also explain a little about what you will be looking for when you inspect your frames and cubes, and we will also explain how to plot the pointing; hence it is worth reading the next sections even if you use the SE to inspect your pipeline processing.

For the Level 0.5 frame, before you send it to a cube, you may want to check the spectra after the task *slicedSpecFlagGlitchFrames* and those that change the state of the data (see the explanation in 3.5.1), you may want to compare the spectra from the same pixels of nod A and B (to see by how much they

differ), and you may want to check on how well the flatfielding has been done. Since the final task of this level of the pipeline only rearranges the spectra of the input *slicedFrames* to turn them into a cube, inspecting the *slicedFrames* or the *slicedCubes* will show much the same thing.

If you want to learn how to mask datapoints yourself, see Chap. 4.

### 3.5.3.1. For the *Frames*

Here we show you various ways to select out parts of your *Frames* or *slicedFrames* products to look at. Many of these can be done now with the Spectrum Explorer (which works on single *Frames* products); the more manual methods are described here in case you have a version of HIPE for which the SE does not yet work on PACS products. Bear in mind, when looking at your spectra, that unless you specify—i.e. select for—to show only the unmasked data, your spectra will look noisy. We do not include here how to select unmasked datapoints only, as that is described in Sec. 3.4.3.2.

#### Before and after tasks

To compare the spectra before and after a task has been run, e.g. the *slicedRsrfsCal* and *slicedSpecRespCal* tasks, is simple and uses variations on the same recipes we have already given for the previous level's data reduction. Remember to copy the data before you run it through tasks if you want to compare a before and after result. You can either copy the whole *slicedFrames*, or an individual element thereof:

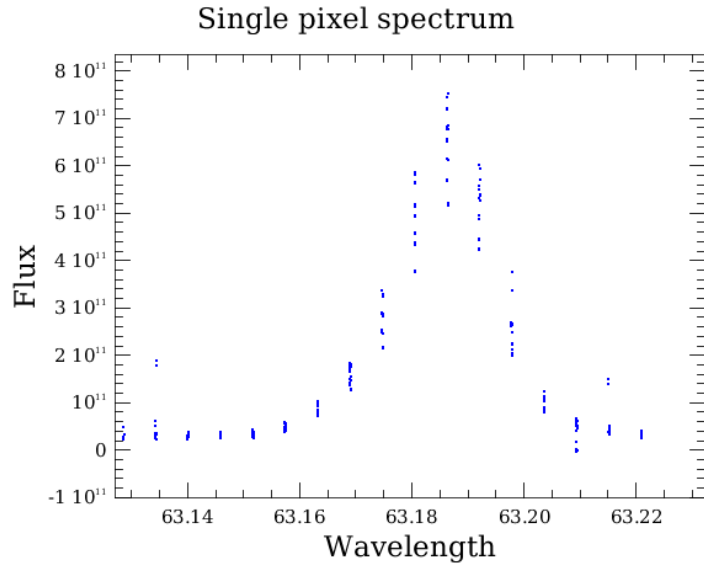
```
# copy a slicedFrames
slicedFrames_b4 = slicedFrames.copy()
# copy a single slice
slice1_b4 = slicedFrames.refs[1].product.copy()
# or
slice1 = slicedFrames.getScience(0).copy
```

As before, to plot the spectrum from a pixel of a slice you need to extract the slice from the *slicedFrames*, and then you follow the same plotting commands that were introduced in 3.4. So, e.g. if you extracted "slice1\_b4" before you ran *slicedRsrfsCal* and "slice1\_aft" after running the task, then:

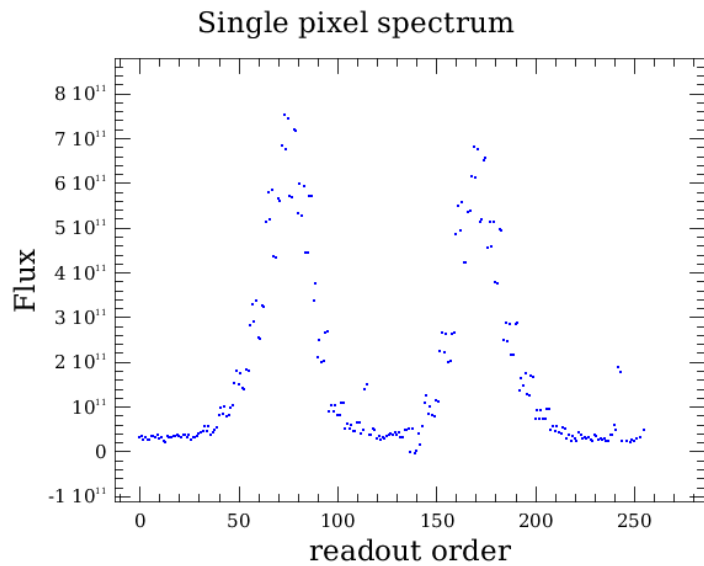
```
sig=slice1_b4.getSignal(8,12)
wve=slice1_b4.getWave(8,12)
p = PlotXY(wve,sig,titleText="your title",line=0)
p[0].setName("before rsrf")
p[0].setYrange([MIN(sig), MAX(sig)])
p[0].setYtitle("V/s")
sig=slice1_aft.getSignal(8,12)
wve=slice1_aft.getWave(8,12)
p.addLayer(LayerXY(wve,sig),line=0)
p[1].setName("after rsrf")
p[1].setYrange([MIN(sig), MAX(sig)])
p[1].setYtitle("Jy")
p.xaxis.title.text="Wavelength [Å]"
p.getLegend().setVisible(True)
```

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

Bear in mind that when you look at the spectrum if a single pixel, signal versus wavelength, before *slicedSpecDiffChop* has been run, you will see what looks like several spectra plotted on top of each other, one for each chop position and one for each grating scan, and these will all be slightly different. 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) changes. The chops are merged in the pipeline task *slicedSpecDiffChop*, the grating scans, and later nods, will be combined in the next part of the pipeline. So, if you plot a pixel, signal versus wavelength, before having subtracted the chops, the plot will contain several spectra all sitting on top of each other, but if you plot the spectra as signal versus time, they will be separated in time. An example of this is shown in the next two plots:



**Figure 3.12. Spectrum of a single pixel. The spectrum is plotted versus wavelength and there are in fact two spectra plotted here, one for each run on the grating**



**Figure 3.13. 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**

In these plots there are two separate spectra because there are two grating scans per pixel remaining. When this this frame is turned into a cube by the pipeline task `specFrames2PacsCube`, 16 pixels, each (in this case) 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  $16 \times 2$  spectra and merge them into one.

If you ran the task `slicedSpecEstimateNoise`, to inspect the noise spectrum is the same as inspecting the astronomical spectrum; the dataset you want to look at in your frame slice is "noise", and you can use the method `.getNoise()` in the same way as `.getSignal()`.

## Plotting grating scans

*Note: this is much easier to do using the Spectrum Explorer!*



How do select out the different grating scans? 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 repeat 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.

Again, to follow the script snippets here you need to have first selected your slice out of your sliced-Frames. Here we call that "myframe".

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 want to separate these distinct time segments while plotting, 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[1:17,12,0:100]).

Alternatively you can select instead on the BlockTable. The BlockTable is a dataset of your frame, in which blocks identifications have been given to the readouts that correspond to distinct instrument settings (see Chap. 4 for a longer explanation). You will select on the data from the BlockTable entry "ScanDir", which separates all the 0 and 1 scan direction readouts. (0 is for moving along the grating in one direction, 1 for moving back again, and there will only be entries 0 or 1, no matter how many times the grating went back and forth.) Selecting on the BlockTable, however, will get you all the data of scan direction 0 or 1; if you wish to select particular 0s or 1s only (e.g. the first up- and down-run on the grating 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.)

First, take out the slice from slicedFrames; here we call that slice "myframe". Then:

```
# to make an array of BlockTable ScanDir entries:
scanIds = myframe["BlockTable"]["ScanDir"].data
print scanIds

# let's say this gives a listing: 0 1 0 1 0 1
# 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[scanIdsSel]
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=Boo1ld(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]
```

To read the entire BlockTable for a *Frames* you can use the Dataset viewer on the BlockTable dataset of your myframe. You can also inspect the MasterBlockTable of slicedFrames using the same viewer. The MasterBlockTable is all the individual BlockTables of the slices in slicedFrames. There is no MasterBlockTable for slicedCubes, but the individual cube slices should have BlockTables.

**Comment:** Although here and in the sections that follow we describe actions performed on single frame slices of your *slicedFrames* product, some things can be done on the entire *slicedFrames*. For example, it is possible to select parts of the frames to plot using the MasterBlockTable and the .select() method that is provided for *slicedFrames* products. More detail on the MasterBlockTable is given in Chap. 4 and the .select() method is described in the *DRM* entry for slicedFrames <LINK>. In this way you could select out for plotting any part of the slicedFrames that is separated in MasterBlockTable blocks. This is how you select out a (new) frame that contains data only from MasterBlockTable entries 4 and 5:

```
newframe = slicedFrames.select(Selection([4,5]))
```

The number to select (here 4 and 5) is the "Index" value you will see on the very left when you use the Dataset viewer on the MasterBlockTable. A new frame—newframe—is created which contains only those parts of the sliced observation that corresponds to the chosen MasterBlockTable blocks. This same method can be used as an alternative selecting/plotting that is discussed in the following sections.

## Plotting different bands

First, take out the slice from slicedFrames; here we call that slice "myframe".

If you want to compare the bands in your observation (and this makes sense mostly after you have run slicedRsrCal and slicedSpecRespCal) 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.

## Plotting nod A and nod B, chop- and chop+

As discussed in the Observer's Manual and in Chap. 2, the chop-nod AOT sees PACS nodding between an A and a B pointing, and a chopping between blank sky and the source. The pipeline task slicedSpecDiffChop subtracts the chop- (blank sky) from chop+, and you are left then with nod A and B. There are 4 chop-nod combinations possible (chop- nodA, chop+ nodB, chop+ nodA, chop- nodB). Before running the task slicedSpecDiffChop you may want to look at the chop- spectra, to see if they are clear of any spectral signature (any non-blank sky signal) and to compare them to the chop+ spectra. After running slicedSpecDiffChop you may want to compare the nod A and B spectra. As mentioned in the beginning of Sec. 3.5, the nod A and B spectra should be the same when you compare those from the same (pixel, module) combination from a *Frames* product or the same spaxel (module) from a *PacsCube* product. However, because the pointing is not exactly aligned between the nods, they may not be. This difference is dealt with in the next pipeline level, but for now you may want to look at the spectra themselves and the pointings of the two nods, to see what differences there are.

The chop- nodA, chop+ nodB should be pointing at blank sky (but different pointings), and chop+ nodA, chop- nodB should be on source. Before you run the pipeline task `slicedSpecDiffChop` you will have, for each separate nod, the chop+ and chop- present within a single frame. To compare the spectra from chop+ and chop- for a single frame from your `slicedFrames` you could do something like this:

```
frame = slicedFrames.getScience(1) #
stat = frame.getStatus("CHOPPOS") # the chopper position Status
on = (stat == "+large") # a boolean, true where +large
on = on.where(on) # a Selection array
off = (stat == "-large")
off = off.where(off)
p=PlotXY(frame.getWave(8,12)[on],frame.getSignal(8,12)[on])
p.addLayer(LayerXY(frame.getWave(8,12)[off],frame.getSignal(8,12)[off])
```

(Bear in mind that for nod A, on source is chop+ and off source is chop-, while for nod B the opposite is the case.) You only need to know whether your chopper throw was "large", "medium" or "small", and this you can get by looking at the CHOPPOS entries of the Status. To identify which nod this science slice belongs to, you can look at the `MasterBlockTable` of `slicedFrames`, in the column "NoddingPosition".

After running the `slicedSpecDiffChop` task you will no longer have these chop+ and chop- datasets separate, as they have been subtracted from each other. You will only have chop+ (for the nod A) or chop- (for the nod B). For this `slicedFrames` you could then compare the nod A and nod B spectra, but for the same line and raster pointing. You could select out the appropriate science frames by looking at the "NoddingPosition" `MasterBlockTable`, and there may be now a convenient method to see a `slicedFrames` listing (see the *DRM* for a listing of methods), or you can do a selection on the command line:

```
nod=slicedFrames.getMasterBlockTable()["NoddingPosition"]
line=slicedFrames.getMasterBlockTable()["LineId"]
rasterc=slicedFrames.getMasterBlockTable()["RasterColumnNum"]
rasterl=slicedFrames.getMasterBlockTable()["RasterLineNum"]
print nod, line, rasterc, rasterl
```

This gives you a set of `Int1d` arrays containing, in order, the nod, line and raster column/line values from the `MasterBlockTable`. If you want to select the rasterc/l value 0, the line value 1 and nod value 2 (A), you do it this way:

```
select=nod.where((nod == 2) & (line == 1) & (rasterc == 0) & (rasterl == 0))
```

where `select` now contains the index values in the `MasterBlockTable` that corresponds to this particular instrument configuration. You then select those frames out of `slicedFrames` and put them into a new, single `Frames` product using:

```
newframe=slicedFrames.select(select)
```

You can now either use `PlotXY` to plot the spectra of any particular pixels of the new `Frames` product, or use the `Spectrum Explorer` on it.

## Plotting the pointing of raster slices

To compare the pointing for different rasters you can follow the script snippets of above to select out the `Frames` you want to look at, and then plot the RA and Dec using the instructions provided in Sec. 3.4.2.5.

See Chap. 4. for more information on the placement of the spaxels of PACS.

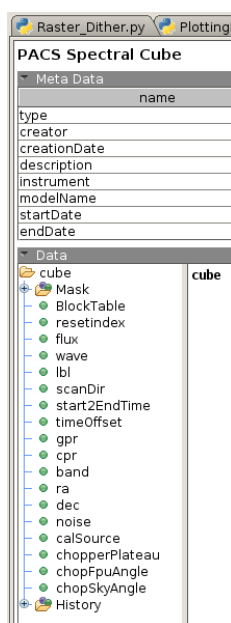
### 3.5.3.2. For the *PacsCube*

By far the easiest way to inspect your `PacsCubes`, contained within `slicedCubes`, is to use the `Spectrum Explorer` on it. However, if that is not available in your version of HIPE, here are some more manual methods.

The description here is for the *PacsCube* product, that is a single slice of a *slicedCubes* product. Here we describe how to inspect such a cube, called "mycube". You can search for the organisation of the slices, and extract them out of the *ListContext* in the same way as has already been explained for the *slicedFrames*.

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 3.14. 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:

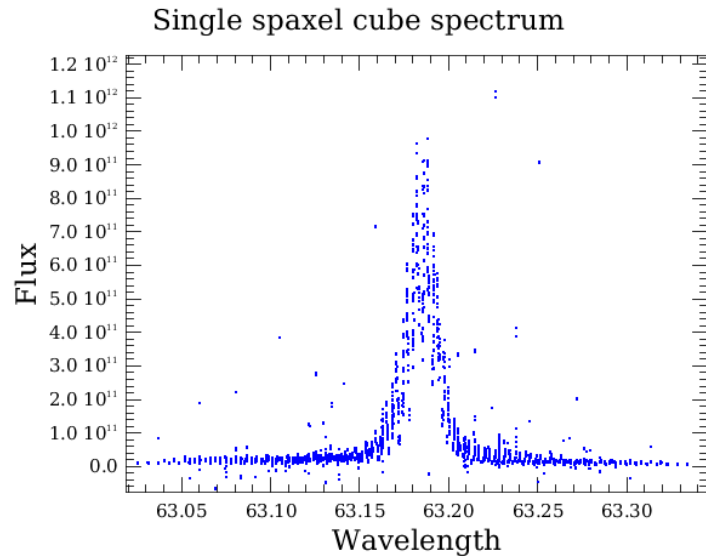
```
flx=mycube.flux[:,2,2]
wve=mycube.wave[:,2,2]
p=PlotXY(wve,flx,line=0)
index_cln=mycube.getUnmaskedIndices(StringId(["GLITCH"]),2,2)
p.addLayer(LayerXY(wve[Selection(index_cln)],flx[Selection(index_cln)],line=0))
```

You can plot the spectrum of a single spaxel in mycube with:

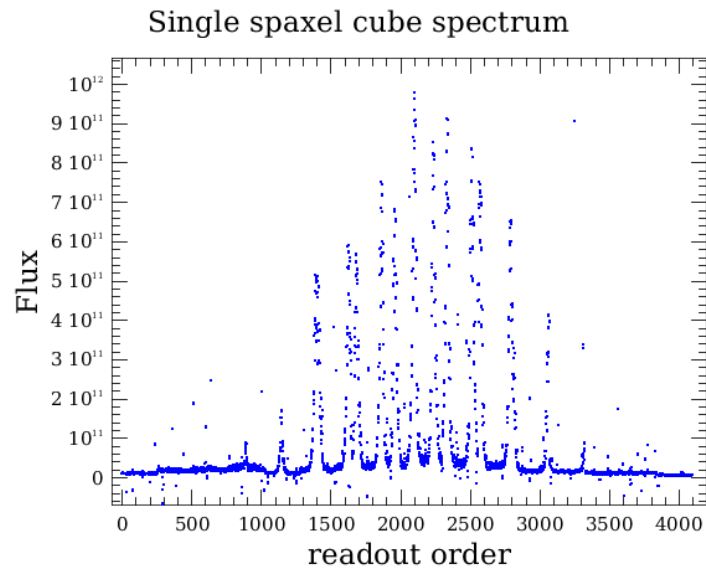
```
p=PlotXY(cube.wave[:,2,2],cube.flux[:,2,2],titleText="your title")
p.xaxis.title.text="Wavelength [Å]"
p.yaxis.title.text="Signal [Jy]"
```

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 overlaid, 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 two figures.



**Figure 3.15.** 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 3.16.** 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 wavelengths 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 separable in the cubes' spaxels via the BlockTable will be

the grating scan direction, because the chops and nods have already been combined (or sliced out) and pixels are not offered as a `BlockTable` or `Status` parameter. To compare the spectra of different pixels in a spaxel it is far easier for you to inspect the equivalent `Frames` product 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) up to and including (15,12) are the first to last pixel of module 12, and module 12 is the central spaxel (2,2).

## 3.6. Level 1 to 2

In this part of the pipeline you will rebin the cubes along the wavelength array, and then combine them spatially. It is here that you need to consider how you want to combine the nod A and B pointings. In Chap. 4 is more information on this issue, including the options for combining the nods, and our recommendations. (If Chap. 4 is incomplete, then instead contact the Herschel Helpdesk.) As said before, look at the spectra and pointings of the same module/spaxel for nod A and B to see how similar they are (or are not). Here we will break down the pipeline into sections on combining the data in the final part of the pipeline, when the projected cube is created, or before the projected cube is created.

### 3.6.1. Pipeline steps

#### 3.6.1.1. Basic procedure

The final tasks take the cube from Level 1 to Level 2. As explained in 3.5.1, if you are not simply continuing from Level 1 (when the `slicedFrames` are converted to a `slicedCubes`) there are a few steps you do first. These steps we repeat here:

```
level1 = PacsContext(myobs.level1)
# get the calTree from myobs
calTree=myobs.calibration
# or get the calTree from HIPE
calTree=getCalTree()
# put that HIPE calTree into myobs, if you have not already done so
myobs.calibration=getCalTree()
camera="blue" # or "red"
```

Next, or to begin with if you didn't need to do the above:

```
from herschel.pacs.spg.pipeline import *
from herschel.pacs.signal.context import *
from herschel.pacs.cal import GetPacsCalDataTask
```

Next you extract the sliced frames and cubes:

```
slicedFrames = level1.fitted.getCamera(camera).product
slicedCubes = level1.cube.getCamera(camera).product
```

And then get a list of the (spectral) line identification, as this stage of the pipeline does a slicing on this:

```
lineIdCol = slicedFrames.getMasterBlockTable().getLineId()
print lineIdCol
```

This will give a list of numbers, a unique one for each line (but the numbers will be repeated if there are multiple pointings per line and for the nod As and Bs). To know what each line number refers to you should be able to look at the `LineDescription` column of the `MasterBlockTable`. (The calibration block, if in the beginning of the observation, will be the first `lineId` numbers, as the `lineIds` are assigned according to grating range and in the order the grating ranges are encountered in the `BlockTable`.)

As you will not want to create the final cubes of the pipeline with mixed spectral lines therein, the next stages of the pipeline will be iterated over the line ids present in your dataset. If there is only one line, you will run these tasks only once. You also have the choice of combining the nods when creating the projected cube, combining them before, or not at all.

Now you run part of the pipeline. The script here will iterate over each `lineId` to create rebinned cubes, which are the penultimate cubes created by the pipeline. In each `slicedRebinnedCubes` (one per line) there will be one rebinned cube per nod and per raster pointing:

```
# iterating over all lineIds
rebinnedCubesList=ListContext()
for lineId in UNIQ(lineIdCol):
    print "Doing lineId",lineId
    # copy slicedCube for use in loop, as will be changed in #3
    # here the .copy() method is not necessary and will not work anyway
    slicedCubestemp=slicedCubes
    # 1
    blocksel = lineIdCol.where(lineIdCol==lineId)
    rasterslices=slicedFrames.getMatchingFrames(blocksel)
    # 2
    cube = slicedCubestemp.refs[rasterslices[0]].product
    waveGrid=wavelengthGrid(cube, oversample=2, upsample = 1, calTree = calTree)
    # 3
    slicedCubestemp = slicedActivateMasks(slicedCubestemp,
        StringId(["GLITCH","UNCLEANCHOP","SATURATION","GRATMOVE", "BADFITPIX"]),
        exclusive = True, sliceSelection = rasterslices, keepall=False)
    slicedCubestemp = slicedSpecFlagOutliers(slicedCubestemp, waveGrid)
    # 4
    slicedCubestemp = slicedActivateMasks(slicedCubestemp,
        StringId(["GLITCH","UNCLEANCHOP","SATURATION","GRATMOVE", "BADFITPIX",
            "OUTLIERS"]), exclusive = True)
    slicedRebinnedCubes = slicedSpecWaveRebin(slicedCubestemp, waveGrid)
    rebinnedCubesList.refs.add(ProductRef(slicedRebinnedCubes))
```

The for-loop runs over `UNIQ(lineIdCol)` rather than `lineIdCol` because the `lineIdCol` is a list of all the `lineIds` you have and the entries in here will be repeated (repeated for raster, nod and repetition factor); you want to include in each `slicedRebinnedCubes` *all* the individual cubes (all nodes and rasters) that belong to each `lineId`. You could of course chose to run these tasks `lineId` by `lineId`, rather than in a loop. The "`rebinnedCubesList=ListContext()`" in the beginning and "`rebinnedCubesList.refs.add(ProductRef(slicedRebinnedCube))`" at the end allow you to put each `slicedRebinnedCubes`—one per `lineId`—in a *ListContext*, storing them for use in the final part of the pipeline. If you only have one `lineId` you don't need to do this, and another alternative to storing the `slicedRebinnedCubes` in a list is to give each one a separate name. That is up to you, but in the next script we assume you have followed what is written here.

And now an explanation of each step:

1. For the `lineId`, identify the parts of the `BlockTable` corresponding to that value. Then feed that into an array (`rasterslices`) that contains the selection numbers of the raster slices with that `lineId`.
2. Select the first cube, for that set of raster slices, of the `slicedCubes` and create a wavelength grid from it. This wavelength grid will be applied to all the other rasters with that `lineId` so they can later be combined. The task creates a single and uniform wavelength grid according to the wavelengths that are in the input cube. 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".
3. Activate masks and run an outlier flagging task (a second glitch detection task). `slicedSpecFlagOutliers` 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.
4. Rebin the raster slices to the wavelength grid previously defined. This will merge all the individual spectra held in each `spaxel`, so there will now only be one spectrum in each. Before running

this task you should select all the masks for which you do not want the to do readouts included when the data are combined. The task creates a cube of dimensions [wavelength,5,5] where now in each spaxel holds a single spectrum. The individual slices of the slicedRebinnedCubes are of class *PacsRebinnedCube*. There should be one cube per nod and raster pointing held in each slice-dRebinnedCubes.

To see how many slicedRebinnedCubes have been placed in the rebinnedCubesList, and to see the number of slices contained in any particular slicedRebinnedCubes:

```
print len(rebinnedCubesList.refs)
print len(rebinnedCubesList.refs[i].product.refs)
```

(At present the convenient methods for looking at the *slicedFrames* product, such as `.getNumberOfFrames`, are not available for the sliced cubes products, so this longer syntax is necessary.)

Now you will create, for each rebinned cube, a projected cube. This is done with a task that takes the RA and Dec coverage of the rebinned cube and "projects" it onto a uniform grid. By default this grid has a spaxel size of 3 arcsec (compared to the 9.4 arcsec that is the original). Remember that the PACS FoV coverage is not a uniform as it looks when you display your cubes (e.g. in the Cube of Spectrum explorer GUIs or with Display): see the plots in 3.4.2.5 and below. More on this task is discussed in Chap. 4 and the *PAPT*. At the same time the task combines raster pointings

### Option A: combine the nods before specProject

This is the recommended method. Here you add the nods together before feeding the result into `specProject`, the task that does the projection. The reason this method is recommended is because adding the nods, A and B for each individual spaxel, is the way of removing the difference in telescope background (the telescope background—i.e. the mirror—will be different for the nods); adding nod A to B gives you the correct spectrum for each spaxel. Nod A or B spectra on their own do not have the correct continuum level or shape.

The task to add the nods can accept a multi-raster pointing `slicedRebinnedCubes` as input:

```
slicedRebinnedCubes = specAddNod(slicedRebinnedCubes)
```

and then you can run `specProject`:

```
projectedCube=specProject(slicedRebinnedCubes_out)
```

### Option B: combine the nods in specProject

In this option you simply treat the nods as separate pointings. The reason that this method is less recommended is because by treating the nods as separate pointings, and at the same time oversampling the spatial plane (the output spaxel sizes are 1/3 that of the input), then for any particular new spaxel you could be adding together a combination of nod A and B from different original spaxels. This will have the effect of not correctly removing the telescope background difference between nod A and B. However, we should add that this final part of the pipeline is still being worked on by the PACS team, and if you have any questions you should contact the Herschel helpdesk.

It could be worth trying this method, if only to see the difference from Option A. Here, for each `slicedRebinnedCube`, i.e. for each spectral line, you do:

```
projectedCube=specProject(slicedRebinnedCubes)
```

## 3.6.2. Inspecting the results

For the `projectedCube` you can use a number of GUIs, outlined in Chap. 1, to inspect it. At some point during the 4.0 track of HIPE these, or other, GUIs will also work on the *PacsCube* and the *PacsRebinnedCube*, that is the output of the pipeline tasks `slicedSpecFrames2PacsCube` and `slicedSpecWa-`



veRebin. To know which GUIs will work on the respective cubes, you can right-click menu select on the cube in the Variables panel, or highlight it in the variables panel and look to see what Applicable Tasks will work on it (Tasks panel).

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
```

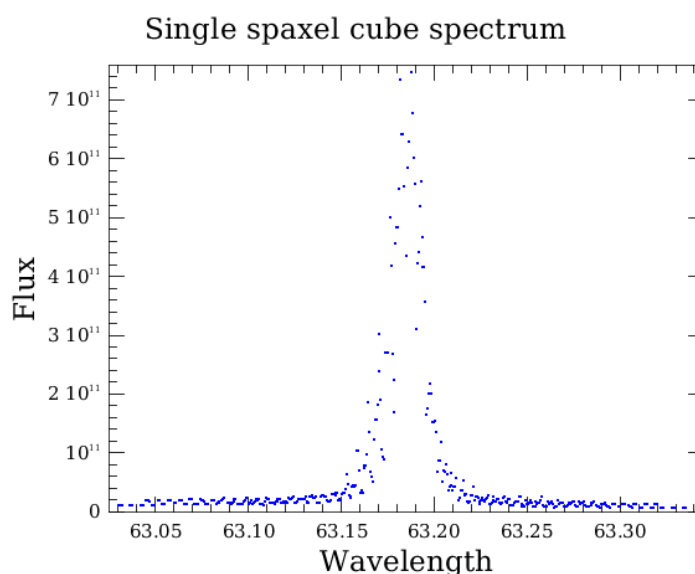
and etc. So you need to do this before following any of the advice next given.

### 3.6.2.1. Plotting and visualising cubes "manually"

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 3.17. 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. 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:

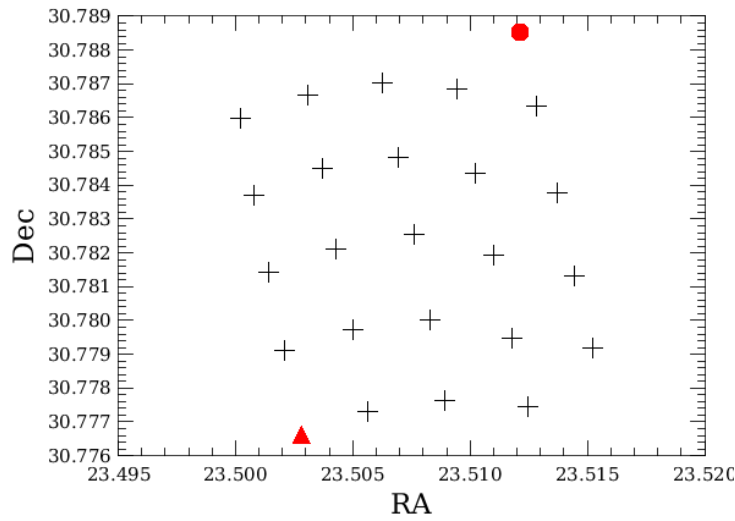
```
#select the final ra and dec of a PacsRebinnedCube
```

```

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 3.18.** 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 `pacCube`. 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 3.19.** `Display` on a `projectedCube`

## 3.7. The End Of The Pipeline

Congratulations! 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 complete) 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. In the *PAPT* the pipeline tasks are explained in more detail, in particular the parameters are presented. More advanced topics, too long to fit in the *PDRG*, are also given there.

---

# **Chapter 4. Further topics.**

## ***Spectroscopy***

### **4.1. Introduction**

This chapter is still being written. It is currently incomplete. It will be completed in the track 5.x version of HIPE.

---

# 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==%i" % (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 task 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:

$$\text{signal(volts)} = (\text{signal(ADU)} - \text{offset}) * \text{gain}$$

```
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:

$$\text{Signal\_correct(crosstalking pixel)} = \text{Signal(crosstalking pixel)} - a * \text{Signal(correlated pixel)}$$

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 PhotAssignRaDec 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 given 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 pixel 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 its proper position and the signal of this readout 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  $\sigma_{clip} > 0$ , 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:

$$\text{noise}[x,y,p] = \text{STDDEV}(\text{signal}[x,y,\text{validSelection}[p]]) / \text{SQRT}(nn)$$

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 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:

$$\text{noise}[x,y,k] = \text{SQRT}(\text{noise}[x,y,pON]**2 + \text{noise}[x,y,pOFF]**2)$$

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,  $\text{noise}[x,y,pON]$  and  $\text{noise}[x,y,pOFF]$  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 `photAddInstantPointing`). 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:  $\text{noise}[x,y] = \text{SQRT}(\text{MEAN}(\text{noise}[x,y,:]**2)) / \text{SQRT}(n)$

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:

$$\text{noise}[x,y,k] = \text{SQRT}(\text{noise}[x,y,A]**2 + \text{noise}[x,y,B]**2)$$

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:

$$\text{noise}[x,y,d] = \text{STDDEV}(\text{signal}[x,y,nd]) / \text{SQRT}(nd)$$

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:

$$\mathcal{F}(t) = \mathcal{E}(t) * \frac{(\mathcal{C}_0)}{(\mathcal{C})} * \frac{1}{(J\Phi)}$$

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,  $\sigma_s$  is the input noise,  $C0$  is our reference,  $\sigma_{C0}$  is the noise of the reference,  $DCs$  is the differential image of the cal-block and  $\sigma_{DCs}$  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 chopped-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 than 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 performed 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 is 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 galactic 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 structures at scales above  $2*\text{filterbox}+1$  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") == 2151313011)
```

#### `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 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 units 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)
```