

PACS Data Reduction Guide: Spectroscopy

**Issue user Version 15.0
March 2017**

PACS Data Reduction Guide: Spectroscopy

Table of Contents

1. PACS Spectroscopy Launch Pad I	1
1.1. Introduction	1
1.1.1. Terminology	2
1.2. Getting and saving PACS observations	3
1.3. What type of observation do I have?	4
1.4. What are the science products in my observation?	5
1.4.1. The quality report	5
1.4.2. The science-quality cubes	5
1.4.3. The spectrum tables	7
1.4.4. The standalone browse products	7
1.5. I just want to look at my cubes!	8
1.5.1. Quick-look cube tools	9
1.5.2. Create quick-look images or spectra from the cubes	9
1.6. Useful scripts for working with PACS data	10
2. PACS Spectroscopy Launch Pad II	12
2.1. Introduction	12
2.2. The PACS pipelines	12
2.2.1. The SPG scripts: the automatic pipeline run by the HSC	12
2.2.2. The interactive pipeline scripts	13
2.3. For what observations <i>must</i> I re-pipeline the data?	14
2.4. For what observations is it <i>useful</i> to re-process the data?	14
2.5. Which are the crucial pipeline tasks?	15
2.6. Point and slightly-extended sources	16
2.7. Extended sources	17
2.8. Correcting for the uneven illumination: the forward modelling tool	18
2.9. Where can I learn about the errors in my spectra?	18
3. Setting up the pipeline	20
3.1. Terminology	20
3.2. Getting and saving your observation data	20
3.3. What type of observation do you have?	22
3.4. The pipeline menu	24
3.4.1. Where are the scripts?	24
3.4.2. What are the differences between the pipeline scripts?	25
3.5. Technical considerations for running the pipeline	28
3.6. Calibration files and the calibration tree	30
3.6.1. Installing and updating the calibration files	30
3.6.2. Checking what has been updated	30
3.6.3. The calibration tree	31
3.6.4. Comparing calibration file versions	32
3.6.5. Saving your <i>ObservationContext</i> and its calibration tree to pool	32
4. The pipeline: the first and the last steps	34
4.1. Introduction	34
4.1.1. The pipeline menu	34
4.2. The first part of all the pipelines: Level 0 to 0.5	34
4.2.1. Slicing	34
4.2.2. Pipeline setup	35
4.2.3. The first pipeline tasks	37
4.2.4. Flagging; adding the wavelengths	38
4.2.5. Slicing, more flagging	39
4.3. The final part of the pipeline: post-processing	40
4.3.1. Extended sources	40
4.3.2. Pointed observations of extended sources	43
4.3.3. Point sources	43
4.3.4. Slightly extended sources	46
5. ChopNod pipelines	47

5.1. Introduction	47
5.1.1. The pipeline menu	47
5.2. The 0.5 to 2 scripts for all pipelines	47
5.2.1. Masking for glitches; convert the capacitance	47
5.2.2. Compute the dark and the response: "Calibration source" scripts only	48
5.2.3. Compute the differential signal	48
5.2.4. Absolute flux calibration: "Calibration source" scripts only	49
5.2.5. Pointing offset correction: "Pointing offset correction" scripts only	49
5.2.6. Spectral flatfielding: all line+short range scan pipeline scripts	52
5.2.7. Spectral flatfielding: all long range scan and SED pipeline scripts	54
5.2.8. Wavelength grid and outlier flagging	56
5.2.9. Spectral rebinning	57
5.2.10. Combine the nods	58
5.2.11. Flux calibration for "Telescope normalisation" scripts only	59
5.2.12. Post-processing	60
5.3. "Telescope normalisation drizzle maps" pipeline	60
5.4. Pipeline steps for spectral lines in the light leak regions (mainly longer than 190 μ m)	61
5.5. The Split On-Off 'testing' script	61
5.6. Wrapper script 'Combine observations for a full SED'	63
5.6.1. Explanation	63
5.6.2. Running the script	63
6. Unchopped and Wavelength switching pipelines	68
6.1. Introduction	68
6.2. The 0.5 to 2 scripts for all pipelines	68
6.2.1. Masking for glitches	68
6.2.2. Compute the dark and the response; subtract the dark	69
6.2.3. Flux calibration	70
6.2.4. Correct for transients: original line scan pipeline script only	70
6.2.5. Correct for transients: new ("with transients") line scan pipeline script only	71
6.2.6. Correct for transients: new ("with transients") range scan pipeline script only	73
6.2.7. Spectral flatfielding: original line scan pipeline script only	74
6.2.8. Spectral flatfielding: original range scan pipeline script only	77
6.2.9. Select the on/off slices: for both the line scan pipeline scripts	79
6.2.10. Wavelength grid and outlier flagging	80
6.2.11. Spectral rebinning	81
6.2.12. Subtract the background: line scan only	83
6.2.13. Plot and save the results	83
6.3. Helper script 'Combine off-source with on-source' in unchopped range spectroscopy	84
6.3.1. Explanation	84
6.3.2. Alternative useful script	85
6.3.3. Running the helper script	85
6.4. Pipeline steps for spectral lines in the light leak regions (longer than 190 μ m)	87
7. More detail on pipeline tasks	89
7.1. Introduction	89
7.2. The pipeline task wavelengthGrid	89
7.2.1. As used in the pipeline	89
7.2.2. The oversample and upsample parameters	90
7.2.3. Noise properties	93
7.2.4. The dispersion	93
7.3. Cubes with an equidistant wavelength grid	93
7.3.1. Create your own equidistant cubes	94
7.3.2. Comparison of equidistant with standard cube spectra	95
7.4. The spectral flatfielding	97
7.4.1. Line scan flatfielding	98
7.4.2. rangeScan flatfielding	100

7.5. Advice on background subtraction for the unchopped modes	104
7.6. Glitches, outliers, and saturation	105
7.6.1. Saturation	105
7.6.2. Glitches	106
7.6.3. The interaction of saturation and glitch masks for targets with bright spectral lines	106
7.7. Spectrum errors in the final cubes	106
7.7.1. Inspecting the StdDev and RMS arrays in the <i>PacsRebinnedCubes</i>	108
7.7.2. Inspecting the errors array of the projected/drizzled/interpolated <i>SpectralSimpleCubes</i>	110
7.7.3. The errors in the extracted central spectrum of point sources	110
7.8. NaNs and flags in rebinned and mosaic cubes	110
7.9. Drizzle: spectral-spatial resampling and mosaicking of cubes	111
7.9.1. When can I use drizzle?	111
7.9.2. Dealing with the off- and on-cubes (unchopped) and nod A and B (chopNod)	111
7.9.3. Construction of the spatial grid	112
7.9.4. pixFrac: the shrinking factor	112
7.9.5. Outlier detection	112
7.9.6. Drizzle error array	113
7.9.7. Structure of the output, and how to extract the sub-products that drizzle creates	113
7.9.8. Using drizzle on longer wavelength ranges	114
7.10. SpecProject	114
7.11. SpecInterpolate	115
7.12. The Pointing offset correction tasks	116
7.13. The transient corrections of the unchopped pipelines	117
8. Dealing with point and slightly extended sources	120
8.1. Introduction	120
8.2. Combining the PACS and SPIRE full SED for point sources	120
8.3. Spectral line skew induced by off-centred sources	121
8.4. Extracting point source spectra that are not located in the central spaxel	121
8.5. Extracting point source spectra that are located in the central spaxel	122
8.5.1. What	123
8.5.2. How to run the task	123
8.6. Extracting the spectra of slightly extended sources	124
8.7. Correcting for the uneven illumination: the forward modelling tool	126
8.7.1. Description of the "forward modelling" process	126
9. The post-pipeline tasks for mapping observations and extended sources	129
9.1. Introduction	129
9.2. The native footprint of the PACS IFU and sampling of the beam	129
9.3. Tiling and pointed observations	130
9.4. Oversampled and Nyquist sampled mapping observation	132
9.5. Correcting for the uneven illumination: the forward modelling tool	132
9.5.1. Description of the "forward modelling" process	133
9.6. Fitting spectral cubes, and making maps from the results	134
9.7. Changing the spaxel size of SPG cubes; mosaicing unrelated observations	135
10. Plotting and inspecting PACS data	136
10.1. Introduction	136
10.2. Pipeline helper tasks explained	136
10.3. Interacting with sliced products	144
10.4. Plotting PACS spectral data yourself: how; and what are you looking at?	145
10.4.1. General considerations: what difference the Level makes	146
10.4.2. Basic plotting, and comparing the before- and after-task spectra	147
10.4.3. Plotting the on-source vs. off-source spectra for chopNod mode observations	149
10.4.4. Plotting the good and bad data	151
10.4.5. Plotting Status values and comparing to the signal and masks	152

10.4.6. Plotting the pointing	154
10.5. Using the Spectrum Explorer on PACS spectra	157
10.6. The PACS IFU footprint	160
10.7. The PACS Spectral Footprint Viewer	163
10.8. PACS product viewer (PPV)	164
10.9. Looking at the background spectrum for unchopped mode AOTs, directly from an observation	166
10.10. How to know the PACS spectral resolution at a given wavelength	167
10.11. Masks and masking	167
10.11.1. Editing masks for a few bad values	169
10.11.2. Manipulating masks on the command-line	171
10.11.3. The "flag" array in cubes	173
10.12. Using the Status and BlockTables to perform selections on data	174
10.12.1. The Status Table	174
10.12.2. Selecting discrete data-chunks using the Status	178
10.12.3. BlockTable and MasterBlockTable	178
10.12.4. Selecting discrete data-chunks using the block table	178

List of Figures

1.1. Some of the cubes of Level 2: the Rebinned (HPS3DR[B R]), with one cube per wavelength range (the outside 0 and 1) and per pointing (the inside 0 to 13); and the Projected (HP-S3DP[B R]), with one cube per wavelength range (0 and 1: the pointings have been combined into a single cube)	6
1.2. The standalone browse products (an example from a pointed observation from SPG 14.2)	8
3.1. Pipeline menu 1: chopNod line scan	24
3.2. Pipeline menu 2: chopNod rangeScan	24
3.3. Pipeline menu 3: unchopped line scan	24
3.4. Pipeline menu 4: unchopped rangeScan	24
3.5. Pipeline menu 5: wavelength switching	24
3.6. Updating the calibration files	31
5.1. Loading cubes from a <i>ListContext</i> into the Spectrum Explorer	62
7.1. upsample fixed, vary oversample	91
7.2. oversample fixed, vary upsample	91
7.3. vary upsample and oversample	92
7.4. Dispersion as a function of wavelength for the various spectroscopy filters for a wavelengthGrid created with upsample=1 and oversample=2.	93
7.5. Comparison of the spectrum with a scaled grid and the same data interpolated onto different equidistant grids (fracMinBinSize = 0.5: blue; 0.35: red; 0.25: magenta): a very short rangeScan	96
7.6. Comparison of the spectrum with a scaled grid and the same data interpolated onto different equidistant grids (fracMinBinSize = 0.5: blue; 0.35: red; 0.25: magenta): blue part of an SED	96
7.7. Comparison of the spectrum with a scaled grid and the same data interpolated onto different equidistant grids (fracMinBinSize = 0.5: blue; 0.35: red; 0.25: magenta): red part of an SED	97
7.8. Before and after flatfielding: blue curve and black dots are from before; yellow curve and red dots are from after	99
7.9. Before and after flatfielding: zoom	99
7.10. Before and after flatfielding (polynomial fitting): blue curve and black dots are from before; yellow curve and red dots are from after	101
7.11. Before and after flatfielding: zoom	101
7.12. An example of an "S-curve" (red spectrum) that is a signature of flatfielding with too-high an order for the fitting (in rangeScans)	102
9.1. Sky footprint of the <i>PacsRebinnedCube</i> (and <i>PacsCube</i>)	130
9.2. PSF Nyquist sampling	130
9.3. Some examples of cubes for a tiling observation. Top centre is the rebinned cube taken from the central position in the raster, with spaxels of 9.4" and the approximate N-E axes indicated. Left and right of that are the same cube but interpolated (right) and projected (left), with N-E indicated and spaxel sizes as you would get from an observation created by SPG 13. Bottom are the entire raster made into a mosaic with specInterpolate (right) and specProject (left), with N-E indicated and spaxel sizes as you would get from an observation created by SPG 13 ..	131
10.1. The plot from slicedSummaryPlot run at the very beginning of the pipeline: grey is the signal, black are the grating movements. Each grating scan is done twice, in opposite directions, creating a V shape, and the 2 sets of Vs in each nod (A and B) because there are two wavelength ranges in these data. The signal is noisy due to the presence of glitches. The block of data on the very left is from the calibration block. When run later in the pipeline there will be additional curves on the plot.	138
10.2. The plot from slicedPlotPointing. The grey curve shows the end of the slew on to the source, and the coloured crosses are labeled.	139
10.3. The plot from plotSignalBasic, where you can see the off (lower) and on (higher) line of datapoints	140
10.4. The plot from plotSignalBasic, after the off signals have been subtracted	140
10.5. The plot from plotPixels, showing the <i>PacsCube</i> data in dots and a rebinned version of those data as a curve.	141

10.6. plotCubesMasks bad data (blue squares) over plotted with plotCube good data (brown/ green circles)	141
10.7. An example of the spectral from three different pixels (1, 8, 16) of a single module	146
10.8. A zoom in on PlotXY of the grating and chopper movements for a Frames	153
10.9. Masked/unmasked data v.s. Status	154
10.10. Movement of PACS during an observation	155
10.11. The IFU footprint. Crosses are the module centres and circled ones are modules 0 (bot- tom-most),1,2 (top-most). Module 5 is immediately to the left of module 0.	156
10.12. The Spectrum Explorer on a <i>PacsCube</i> (screenshot from HIPE 12)	158
10.13. The Spectrum Explorer on a <i>Frames</i> (screenshot from HIPE 12)	158
10.14. The PACS spectroscopy detector layout	160
10.15. Sky footprint of the <i>PacsRebinnedCube</i> and <i>PacsCube</i>	161
10.16. The PACS spectral footprint viewer	164
10.17. The PACS Product Viewer	165
10.18. Editing masks with the PPV	170
10.19. The MasterBlockTable	179

List of Tables

10.1. Coordinates of the Frames to the PacsCube	161
10.2. Status information	175
10.3. BlockTable columns	179

Chapter 1. PACS Spectroscopy Launch Pad I

1.1. Introduction

Welcome to the PACS data reduction guide (*PDRG*). We hope you have gotten some good data from PACS and want to get stuck in to working with them. The PDRG explains the running of the data reduction pipelines for PACS spectroscopy in HIPE, and how to interact with the data. This first chapter—the *PACS Spectroscopy Launch Pad I*—is a "ReadMeFirst" to working with PACS spectroscopy.

1. How do I get and save a PACS spectroscopy observation?
2. How do I understand what type of observation I have?
3. What are the science products in my observation?
4. How can I plot my spectra and view my cubes?
5. What scripts can help me work with PACS spectroscopy?

The second chapter—[Chapter 2](#), the *PACS Spectroscopy Launch Pad II*—is about the pipeline, and should be read by everyone working with PACS spectroscopy data (even if they do not want to run one of the pipelines in HIPE themselves). In this chapter we also summarise what post-pipeline processing *can or should be done* to different types of observation, and for which you may need to use HIPE.

1. What are the pipelines available for reducing PACS data
2. Do I *need* to re-run the pipeline (and if so, which one?)
3. Is it anyway *useful* to re-run the pipeline to improve some of the results?
4. Which are the crucial pipeline tasks?
5. I have a point source: what do I do next?
6. I have an extended source: what do I do next?
7. Where do I go to learn about the errors in my spectra?

The rest of the PDRG is:

- [Chapter 3](#): gives more detail on the calibration files used in the pipeline processing, on getting and saving data, on where the pipeline scripts are, and an explanation of the differences between these interactive pipeline scripts. *An observation gotten from the HSA will only have been processed through one type of pipeline: this chapter is also useful for understanding whether you should consider running one of the other pipelines for your observation.*
- [Chapter 4](#): concerns the beginning (Level 0—0.5) and the end (post-processing) parts of the reduction of PACS data, which are almost the same for all the pipelines. [Chapter 5](#): explains the data reduction from Level 0.5 to 2 for chopNod mode observations. [Chapter 6](#): explains the data reduction from Level 0.5 to 2/2.5 for unchopped mode observations.
- [Chapter 7](#): explains some of the more crucial pipeline tasks: wavelength regridding; flatfielding; using the different cube mosaicking tasks (drizzle, interpolate, project); transient correction for the unchopped mode; background subtraction for the unchopped mode; and the unique tasks of the pointing offset pipeline. The errors in the final PACS spectroscopy products are also explained here.

Whether you run the pipeline or not, read this chapter to understand the effect these crucial tasks can have on the appearance of the spectra in the science products of the pipeline.

- [Chapter 8](#): explains the post-pipeline tasks that are provided for point sources and semi-extended sources, and which can be run after any pipeline script.
- [Chapter 9](#): explains the post-processing tasks to deal with mapping observations, creating mosaic cubes of various type, the differences between the various types of cubes, and what each one is best used for. *Whether you run the pipeline or not, read this chapter to understand why the different science-level cubes look the way they do.*
- [Chapter 10](#): includes scripts and general information about plotting PACS data for diagnostic reasons during the pipeline processing, describes the viewers for inspecting PACS data (spectrally and spatially); and explains how to work at a deeper level with some of the datasets of the cubes.

Additional reading when working in HIPE can be found on the HIPE help page, which you can access from the *Help#Help Contents* menu. This covers the topics of: HIPE itself, I/O, scripting in HIPE, and using the various data inspection and analysis tools provided in HIPE. In particular the [PPE](#) in *PACS Products Explained* should be consulted to learn more about PACS products: how they are constructed, for what purpose they are variously created, and accessing them within and outside of HIPE.

Documentation outside of HIPE can be found on the PACS calibration pages on the Herschel Science Centre site, where the Observer's Manual and calibration documentation and information are provided (currently at herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb). A more permanent repository of PACS documentation can be found with the HELL archive [PACS pages](#). The [Quick Start Guide](#) is recommended to begin with your PACS journey, and a set of flowcharts guiding you through the various PACS spectroscopy products can be found in the *Product Decision Tree* document posted on HELL. The *PACS Handbook* is also recommended, as it is the post-operations, and final, version of the Observer's Manual.

Text written *like this* refers to the class of a product (or to any product of that class). Different classes have different (java) methods that can be applied to them and different tasks will run (or not) on them. See the [Scripting Guide](#), to learn more about classes. Text written *like this* refers to the parameters of a task.

1.1.1. Terminology

The following definitions will be useful to know:

- **HIPE** Herschel Interactive Processing Environment
- **DAG**: the HIPE Data Analysis Guide (this explains the general HIPE data analysis tools)
- **SG**, the Scripting Guide (a guide to scripting in HIPE)
- **PACS URM** the User's Reference Manual, describes the PACS tasks, their parameters and their function
- **PPE** in *PACS Products Explained* the PACS Products Explained, which is about the products you get from the HSA or which you produce while pipeline processing PACS data
- **HSA, HSC** Herschel Science Archive and Herschel Science Centre
- **AOT** different PACS observing modes were programed with different Astronomical Observing Templates by the proposer
- **Level 0** products are raw and come straight from the satellite
- **Level 0.5** products have been partially reduced, spatially calibrated, and corrected for instrument effects by tasks for which no interaction is required by the user

- **Level 1** products have been more fully reduced, some pipeline tasks requiring inspection and maybe interaction on the part of the user
- **Level 2** products are fully reduced and flux calibrated, including tasks that require the highest level of inspection and interaction on the part of the user. Post-pipeline processing for point and extended sources is done on these data
- **Level 2.5** products can be found for unchopped range observations only. For this AOT, the on-source and off-source observations are taken separately: therefore each is reduced to Level 2, and then the off-source data are subtracted from the on-source data and the resulting products placed in Level 2.5 in the on-source observation. The post-pipeline processing can be done on these data
- **Level 3:** can be found for chopNod, pointed, full SED-coverage observations only. These are spectrum tables, which are a concatenation of data from the two or three observations that were taken to cover the entire PACS spectral range for a target
- **SPG:** standard product generation: the data reduced with a standard pipeline by the HSC. For each major version of HIPE a new SPG is run (this taking a few months of time). The SPG version of any observation is shown in the HSA results tab, in the "Summary" tab of the Observation Viewer, and also in the Meta datum "creator" of the *ObservationContext*. "SPG 13" is the SPG using the pipeline scripts of HIPE 13.
- **spaxel:** a spatial pixel, the spatial unit of an Integral Field Unit (IFU). Each spaxel contains the spectrum from one unique "pixel" on the sky. The native spaxel size of PACS is 9.4x9.4 arcsec; when you mosaic together cubes the resulting cube's spaxels are smaller, but they are still called "spaxels".

1.2. Getting and saving PACS observations

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

- They are identified with a unique number known as the Observation ID (**obsid**). You can find the obsid via the HSA.
- They can be downloaded directly into HIPE, or one at a time to disc, or many as a tarball.
- The data you get from the HSA is an **Observation Context**, which is a container for all the science data and all the auxiliary and calibration data that are associated with an observation, and includes the **SPG** products. The entire observations is stored on disk as individual FITS files organised in a layered directory structure. The *ObservationContext* you load into HIPE contains links to all these files, and GUIs are provided to navigate through the layers.

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

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

```
obsid = 134..... # enter your own obsid
# To load into HIPE:
myobs = getObservation(obsid, useHsa=True)

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

See the *DAG* [sec. 1.4.5](#) for more information on `getObservation` (for example, how to log on to the HSA before you can get the data, and more about "MyHSA"), and [Section 3.2](#). For full parameters of `getObservation`, see its [URM](#) entry.

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

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

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

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

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

In [Chapter 3](#) you can find a longer summary of getting and saving. The PACS pipelines use these command-line methods. To learn about the GUI methods, see chap. 1 of the [DAG](#).

1.3. What type of observation do I have?

PACS spectrometer observations were executed following a set of observing templates: the **AOTs**. Different AOTs may need to be reduced with different pipelines.

- **chopNod, unchopped, or wavelength switching AOTs:** the difference between these lies in the observing technique used to sample the telescope+astronomical background. ChopNod was the most common AOT. Unchopped was used for sources in crowded fields where chopping-nodding was not possible; for unchopped rangeScans the observers had to specify a separate observation for the off-source (background) position, for unchopped lines scans the on- and off-source were taken within one observation. Wavelength switching was also for crowded fields but this mode was discontinued a few months into the mission.
- **Line or Range spectroscopy:** with a wavelength range that encompasses one unresolved line only (Line); an observer-defined wavelength range (Range); or the full spectral range of PACS (SED, which is part of Range).
- **Pointed, undersampled mapping (tiling), Nyquist mapping, or oversampled mapping:** refers to the pointing mode: a single pointing; a large-step raster to cover a large field-of-view (tiling); a smaller-step raster to achieve a Nyquist spatial sampling of the beam; or very a fine-sampling raster to oversample the beam.

For the three mapping modes, whether undersampled or not dictates which pipeline task can be used to combine the rasters into a single, mosaic cube.

A more detailed summary for each mode is provided in [Section 3.3](#). Note that observations always contain data from the blue and the red camera (the only exception being when one camera failed).

Once you have downloaded an observation into HIPE you can find all relevant AOT information with:

```
obsSummary(myobs)
```

or look at the **pacObsSummary** in the `ObservationContext` (use the `Observation` viewer on your "obs"). In the summary text look for the section "AOT and instrument configuration", e.g.

```
AOT and instrument configuration:
AOT:                               PacsLineSpec
Mode:                               Mapping, Chop/Nod
Bands:                              B3A R1 (prime diffraction orders selected)
```

```

Is bright:      YES (shortened range mode)
Raster lines:  5
Raster columns: 5
Raster line step: 14.5 (arcseconds)
Raster point step: 16.0 (arcseconds)
Chopper:       large throw
Nod cycles:    1

```

and there you will find:

- *AOT* (line or range)
- *Mode* (mapping or pointed; and unchopped, wavelength switching, or chopNod)

For Mapping modes, note down the size of the steps (Raster line or point step) and number of steps (Raster lines or columns); this information will be useful later.

1.4. What are the science products in my observation?

1.4.1. The quality report

The quality report comes in the form of a **quality** and/or **qualitySummary** in the ObservationContext. They both contain the same information, but the qualitySummary (if present) also contains a report created after a manual check of the observation has been done at the HSC.

Click on the +quality/+qualitySummary from within the Data tab of the Observation viewer and the report viewer will open to the right of that tab. The most important things to check are the Quality flags and comments: noting that by SPG 14 most observations have no flags and if there are no comments to make, the comments part will be blank. See also the quality documentation on herschel.esa.int/twiki/bin/view/Public/DpKnownIssues.

1.4.2. The science-quality cubes

The cubes produced at the end of the SPG pipeline are found in **Level 2** for most observations, **Level 2.5** for the on-source observation of an unchopped range pair. Since there can be, within any observation, several pointings and multiple wavelength ranges, all the related cubes are gathered together in *contexts*: one *context* for the red camera and one *context* for the blue camera. In addition, PACS produces different types of cube depending on the pointing mode adopted, and there are always separate sets of red and blue *contexts* for each type of cube.

- Double-click on your observation in the *Variables* panel (or right-click and select the **Observation Viewer**). The viewer will open in the *Editor* pane of HIPE.
- In the directory-like listing on the left of the Observation viewer (under "Data"), click on the + next to the "level2" (or "level 2.5" if there is one)
- The names of the cube *contexts* start with **HPS3D** (Herschel-PACS 3-dimension) and then contain the letters indicating the specific type of cube held therein:
 - HPS3DP[R|B] are the red and blue ([R|B]) projected cube contexts
 - HPS3DR[R|B] are the red and blue rebinned cube contexts
 - HPS3DD[R|B] are the red and blue drizzled cubes contexts
 - HPS3DI[R|B] are the red and blue interpolated cubes contexts

Click on the + next to the HPS3DXX to see their individual, numbered, cubes:

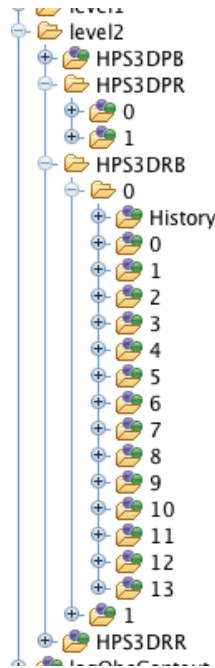


Figure 1.1. Some of the cubes of Level 2: the Rebinned (HPS3DR[B|R]), with one cube per wavelength range (the outside 0 and 1) and per pointing (the inside 0 to 13); and the Projected (HPS3DP[B|R]), with one cube per wavelength range (0 and 1: the pointings have been combined into a single cube)

A summary of the contents of all levels of an *ObservationContext* is given in the [PPE](#) in *PACS Products Explained*. To summarise:

- **HPS3DR[R|B]**. The *context* of **rebinned cubes** of class *PacsRebinnedCube*. There is one cube per wavelength range and per pointing specified in the observing proposal. These cubes have an irregular spatial grid and a non-equidistant wavelength grid. The spaxels are 9.4". For observations of point or slightly extended sources, the correctly-calibrated spectrum must be extracted from these rebinned cubes.
- **HPS3DP[R|B]**. The *context* of **projected cubes** of class *SpectralSimpleCube* are mosaic cubes created for all mapping observations. There is one cube per wavelength range requested in the observing proposal. These cubes have a regular spatial grid but also a non-equidistant wavelength grid. The spaxels are 0.5" for pointed observations, and up to 3" for mapping observations (3" if there is no drizzled cube provided, otherwise the same size as the spaxels of the related drizzled cube).
- **HPS3DD[R|B]**. The *context* of **drizzled cubes** of class *SpectralSimpleCube* and are mosaic cubes created for lineScan Nyquist and oversampled mapping observations, i.e. those with:
 - *Nyquist mapping*: in the blue, step sizes of up to 16" with a 3x3 raster; in the red step sizes of up to 24" with a 2x2 raster
 - *Oversampled mapping*: in the blue, step sizes of up to 3.0" with a 3x3 raster; in the red 4.5" step sizes with a 2x2 raster

There is one cube per wavelength range specified in the observing proposal. These cubes have a regular spatial grid but also a non-equidistant wavelength grid. These cubes have a spaxel size that depends on the spatial sampling of the raster and the wavelength.

Warning: you should not use the drizzled cubes from SPG 13 as the fluxes are incorrect. Use the projected cubes instead. Those of SPG 14 and 15 (i.e. the final products in the archive) have correct fluxes.

- **HPS3DI[R|B]**. The **interpolated cubes** of class *SpectralSimpleCube* and are mosaic cubes created for pointed and tiling observations. There is one cube per wavelength range specified in the

observing proposal. These cubes have a regular spatial grid but also a non-equidistant wavelength grid. They always have a spaxel size of 4.7" (SPG 13) or 3" (SPG 14). Undersampled mapping is any mapping mode for which the steps sizes are larger than the Nyquist values given above, or the number of steps less, for each camera independently.

- For cubes at Level 2.5, which is a level you will find in the on-source observations taken in the unchopped range scan mode, the same cube are provided but with a **BS** just before the camera letter. This stands for "background subtracted", and these are cubes for which the off-source Level 2 has been subtracted from the on-source Level 2: these are then the science products to use.

Go to the + next to the HPS3DXX. The list of numbers (+0, +1, +2...) are the individual cubes. A tooltip appears when you hover over a cube detailing its contents (wavelength range, position in raster, type of cube). Click on the numbers to see the associated cube open, either within the Observation Viewer, or to see it in a new window, right-click on the number and chose the **Spectrum Explorer** (SE: see the *DAG* [chap. 6](#)). Using the SE you can look at the spectra of your spaxels, perform mathematical operations, extract spectra or sub-cubes, make velocity and flux maps and fit your spectra. You can also drag-and-drop the cubes (to the *Variables* panel) to take them out of the *ObservationContext* and from there export as FITS.

See [Section 10.6](#) to learn more about the native footprint of the PACS integral field unit, and that entire chapter to learn about the various cubes provided. (To find the observing mode of your observation, see [Section 1.3.](#))

Note that if the observer did not request an off-source observation, then the final science products for unchopped rangeScans will be in the Level 2 of the observation, rather than Level 2.5.

1.4.3. The spectrum tables

From Track 13 we provide a table of the data of the rebinned cubes, which can be found in the *contexts* called **HPSTBR[R|B]**. Within each *context* there is one table for each requested wavelength range of the observation, and all the spaxels *and all pointings* (for mapping observations) are contained in each table. The columns of data include the spaxel coordinates, the raster coordinate, as well as the wavelengths, fluxes and stddev values. See the [PPE](#) in *PACS Products Explained* for a more detailed explanation of this table.

From Track 14 we provide a table of extracted spectra: **HPSSPEC[R|B]** at **Level 2**, **HPSSPECB-S[R|B]** at **Level 2.5**, and **HPSSPEC** at **Level 3**. The first you can find for all pointed observations, and the second only for the subset of pointed observations that are chopNod and were taken to cover the entire SED in two or three separate observations. For this second, the data from both cameras and all the SED obsids are included in a single table (i.e. there is no [R|B]). The data in these tables are: the spectrum taken directly from the central spaxel, the point source-calibrated spectra "c1", "c9" and "c129" from the task extractCentralSpectrum for the chopNod AOTS and "c1" and "c9" for the unchopped AOTs. The spectra are converted into columns in the tables, and column names and Meta data help you understand the tables. See the [PPE](#) in *PACS Products Explained* for a more detailed explanation.

1.4.4. The standalone browse products

The standalone browse products are created from the Level 2/2.5 cubes, and the Level 2/2.5/3 tables produced by the SPG, all of which are explained above. The standalone products are: the interpolated, drizzled, and/or projected cubes (depending on the AOT) but with an equidistant wavelength grid (i.e. each bin in the spectral grid is the same size); the data of the rebinned cubes as a table. All are provided as FITS files.

The reason for providing cubes with an equidistant wavelength grid is that these cubes then have a full WCS, with equidistant grids in the two spatial *and* the spectral direction. Cube viewers (e.g. ds9) can deal nicely with these cubes. A table of the data of the rebinned cubes is also provided: these cubes can never have a regular spatial grid, but by providing the spectral data as a table, the user can read them into most other software and deal with them perhaps more easily than in cube format.

These standalone products can either be downloaded directly from the HSA, or can be extracted from an *ObservationContext*: in the **browseProduct** level or in the Level they come from. They have the same name as the cubes discussed above but with an "EQ" added:

- **HPS3DEQP[R|B]**. The **projected cube** *context* in with an equidistant wavelength grid
- **HPS3DEQD[R|B]**. The **drizzled cube** *context* with an equidistant wavelength grid
- **HPS3DEQI[R|B]**. The **interpolated cube** *context* with an equidistant wavelength grid
- **HPSTBR[R|B]**. The **rebinned cube** *contexts*, with the data in a tabular format rather than a 3d product
- **HPSSPEC[R|B]**. The extracted spectrum table *contexts*, taken from Level 2 or 2.5, for all pointed observations
- **HPSSPEC**. The extracted spectrum table *context*, taken from Level 3, for all pointed, chopNod, full SED observations: concatenated table for red and blue camera and all obsids that contributed to the full SED coverage
- If you have the on-source observation of an unchopped range mode on/off pair, the same products from Level 2.5, but with **BS** in the name, are also provided.

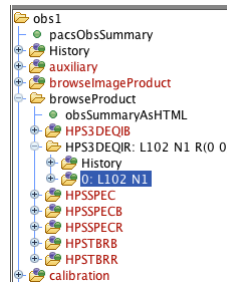


Figure 1.2. The standalone browse products (an example from a pointed observation from SPG 14.2)

Read the [PPE](#) in *PACS Products Explained* for more information on these products.

1.5. I just want to look at my cubes!

Which Level 2 cubes should you look at?

- If you have a Level 2.5 in your *ObservationContext*, then you are looking at the background-subtracted on-source observation of an unchopped rangeScan observation-set and this Level contains the final cubes. Otherwise go to Level 2
- If you have a pointed observation of a point or semi-extended source, look at the rebinned or interpolated cubes (HPS3DR[R|B], HPS3DI[R|B]): the second are easier to load into software outside of HIPE because they have a regular spatial grid (which has been spatially-resampled from the first by the *specInterpolate* task), while the first have the native footprint of the PACS IFU
- Tiling observations: the projected or interpolated cubes (HPS3DP[R|B], HPS3DI[R|B]), where the second are generally preferred
- Mapping observations: drizzled or projected cubes (in that preference order) (HPS3DD[R|B], HPS3DP[R|B])

Extract the cubes from their contexts (the HPSXXX) with a drag-and-drop, or on the command line:

```
# To get the first level 2 blue drizzled cube
cubes = obs.refs["level2"].product.refs["HPS3DDB"].product.refs[0].product
# To get the second level 2.5 red projected cube
```

```
cubes = obs.refs["level2_5"].product.refs["HPS3DPBSR"].product.refs[1].product
```

1.5.1. Quick-look cube tools

There are a number of GUIs that can be used to inspect PACS cubes. These are explained in the [DAG](#) chps 6 and 7. When you have a cube highlighted in the *Variables* pane of HIPE (or in the directory listing in the Data panel of the Observation viewer) you can call up these tasks via the right-click menu. Note that all of these GUIs work on the individual cubes, not on the *context* they are contained within (see [Section 1.4](#)), so you need to go down past the HPS3DXX level in the Level 2 layer of the *ObservationContext*, to the +0, +1...

- To scroll through 2D wavelength slices of your cubes you can use the **Standard Cube Viewer**.
- The **SpectrumExplorer** (see the *DAG* [chap. 6](#)). This is a spectral#spatial visualisation tool for spectra and cubes. It allows for an inspection and comparison of spectra from individual spaxels or from separate cubes, and it gives access to various mathematical tasks via its **Spectral Toolbox** menu (see [chap. 6.4](#)).
- The **Cube ToolBox**, which you also access via the SpectrumExplorer: see the *DAG* ([chap 6.4](#), [chap 6.5](#)). Together with the Spectrum Toolbox, this allows you to inspect a cube spatially and spectrally at the same time. It also has analyses tasks#you can make line flux maps, velocity maps, and extract out spectral and spatial regions.
- The **Spectrum Fitter GUI**, which you also access via the SpectrumExplorer: see the *DAG* [chap. 7](#). This GUI allows you to fit the spectra of your cubes with a variety of models. (For cubes it should be accessed via the Toolbox menu of the Spectrum Explorer, **not** directly from the HIPE *Tasks* panel.)

1.5.2. Create quick-look images or spectra from the cubes

A few quick inspection tasks so you can get a feel for your cube data:


- **Extract the spectrum of a single spaxel from a pointed observation/rebinned cube** with the task `extractSpaxelSpectrum`, which will only work a rebinned cube (HSP3DR[R|B]).

```
slicedRebinnedCube = obs.refs["level25"].product.refs["HPS3DRR"].product
spaxelX, spaxelY = 2,2
slice = 0
spectrum = extractSpaxelSpectrum(slicedFinalCubes, slice=slice,\
    spaxelX=spaxelX, spaxelY=spaxelY)
```

To know which slice you want (i.e which cube in the *context*), you can use the task `slicedSummary(slicedRebinnedCube)`. To know which spaxel you want to extract, open the cube with the Standard Cube Viewer, and the spaxelX and Y coordinates (in that order) of the spaxel under the mouse can be found at the bottom-left of the viewer. The "spectrum" output can be opened in the Spectrum Explorer.

- **Extract the spectrum of a single spaxel from a mapping observation/any cube** with the cube GUIs provided in HIPE. Extract the cube, e.g. for the second cube

```
cube = obs.refs["level2"].product.refs["HPS3DPR"].product.refs[1].product
```

Open the cube in the **Spectrum Explorer** (right-click menu on "cube" in the Variables panel of HIPE), and from there select the **Cube Toolbox** (the  icon at the top of the "SE"), which will open in the top part of the SE.

The Cube Toolbox tasks are located in the drop-down menu to the right of the plot panel: to extract a single spectrum use "extractRegionSpectrum", where you can select out a single spaxel with a

click on the cube image. See the *DAG* ([chap 6.7.3](#)) to learn more about using the SE and the Cube Toolbox.

- **Extract the summed or average spectrum of a region:** using the Cube Toolbox mentioned above, you can also select a rectangular or circular region using the task `extractRegionSpectrum`: see the *DAG* ([chap 6.7.5](#)). This task will work on the Level 2/2.5 cubes: `HSP3D[R|D|I|P][R|B]`.
- **Extract an image** of a single wavelength point is done the most rapidly with the Standard Cube Viewer, on any mosaic cube, right-click menu "Extract current layer".
- **Extract a wavelength-integrated image:** following the steps for "Extract the spectrum of a single spaxel" above to select the correct cube and open the Cube Toolbox, from there select the task "IntegrateSpectralMap". See *DAG* ([chap 6.7.10.1](#)) to learn how this works.
- **Plot the spectrum of a spaxel together with the RMS estimate:** using the pipeline helper task "plotCubesStddev" on any rebinned cube (`HPS3DR[R|B]`). See [Section 7.7.1](#) for a longer explanation of this task.

1.6. Useful scripts for working with PACS data

There are useful scripts provided for working with PACS spectroscopy that can be obtained via the HIPE Scripts menu. Those of most interest to the non-pipeline processing astronomer are:

- **Fitting PACS cubes:** three scripts are provided for fitting a spectral line in PACS cubes and making images from the fitting results, e.g. integrated flux and velocity. These scripts are:
 1. *Spectroscopy: Fitting mapping observations (mosaic cubes)*. For the mosaic cubes of mapping observations, and starting from the interpolated, drizzled, or projected cubes (`HPS3D[I,D,P][R|B]`).
 2. *Spectroscopy: Fitting mapping observations (pre-mosaic cubes)*. Also for mapping observations but this time starting from the rebinned cubes (`HPS3DR[R|B]`). The difference with the previous script is that the fitting is done on these individual cubes of the raster and then the mosaicking is done on the fitting result images, i.e. creating 2d mosaics (images) rather than 3d mosaics (cubes).
 3. *Spectroscopy: Fitting single pointing cubes*. For pointed observations, creating fitting images is more qualitative than quantitative, but nonetheless is useful for visualising the results for extended sources observed as a single pointing. The script starts with the interpolated cubes (`HP-S3DI[R|B]`).
- *Changing the spaxel size:* for those who do not wish or need to reprocess the data but do wish to create cubes with a different spaxel size (e.g. to compare cubes that the SPG created with slightly different spaxel sizes), this can be done with a useful script called *Spectroscopy: create mosaic cubes with any spaxel size*.
- *Mosaicking multiple observations:* how to combine observations that were taken as separate obsids but which cross-over spatially and spectrally, is explained in the useful script called *Spectroscopy: Mosaic multiple observations*
- **Point sources:** for point sources it is necessary to use the tasks provided to produce a correctly-calibrated spectrum of the point source from the rebinned cubes (those in `HPS3DR[R|B]` in the observation). The two scripts are: *Spectroscopy: Point source loss correction (central spaxel)* and *Spectroscopy: Point source loss correction (any spaxel)*.
- *Spectroscopy: Combine PACS and SPIRE spectra:* is a script that is aimed at observations of point sources, where you wish to combine the spectrum of these two instruments into a single spectrum. Note: this is not a mathematical combination, the spectra are simply stored in a single product, for ease of viewing and transporting.

- *Spectroscopy: Convolution for spectral images*: this script shows you how to take two spectral images (e.g. as created in the fitting scripts) and convolve the shorter wavelength image to the beam of the longer wavelength image: the images can then be directly compared to each other.

Chapter 2. PACS Spectroscopy Launch Pad II

2.1. Introduction

This chapter should be read by those wondering whether it is necessary to run the pipeline on their data themselves. *For many observations, the SPG will produce good results and reprocessing will produce little change. However, note that the pipeline processing results do depend to a degree on the observations themselves (on the "observing conditions", if you like, and the brightness and complexity of the source).* Hence, a re-processing for the more complex sources may improve the results. A detailed inspection of the spectra of the *PacsRebinnedCubes* (HPS3DR[R|B]) of Level 2/2.5 is strongly recommended (for *all* AOTs), before deciding whether to reprocess or not.

This chapter should answer the following questions:

1. Where and what are the PACS spectroscopy pipelines?
2. For what observations do I *need* to re-pipeline the data?
3. For what observations will it be *useful* to re-process the data, to try to achieve a better result?
4. Which are the *crucial pipeline tasks that have the greatest effect* on the resulting spectra?
5. I have a point source: what do I do next?
6. I have an extended source: what do I do next?
7. Where do I go to learn about the errors in my spectra?

2.2. The PACS pipelines

2.2.1. The SPG scripts: the automatic pipeline run by the HSC

The SPG scripts are those run at the HSC in automatic mode. The SPG treats the following AOTs separately: unchopped line and wavelength switching, unchopped range, chopNod line, chopNod range. The SPG processing takes a few months to run and hence there is a lag between a release of HIPE and the availability of those SPG products in the HSA. The very final processing of PACS data is that run in HIPE 14.2, for which this PDRG is written.

To know which SPG your downloaded observation was processed with, look at the Summary tab for the observation in the Observation viewer, or, for an observation called "obs",

```
print obs.meta.["creator"]
```

The SPG scripts used within any track of HIPE are available from the HIPE Pipeline menu, within each of the submenus described below. They are not intended for interactive use#the interactive scripts should be used#rather they are provided for completeness. In HIPE 14 the SPG scripts are based on the **Telescope normalisation** pipeline for the chopNod observations, and the **Calibration source and RSRF** pipeline for the unchopped observations. There are no differences in the range of tasks run in the SPG pipeline and the equivalent interactive ones.

2.2.2. The interactive pipeline scripts

The pipeline menu in HIPE is split into five: chopNod line, chopNod range, unchopped line, unchopped range, wavelength switching (an old mode, which uses the unchopped line pipeline script and which we do not discuss separately further). Inside each of these are a choice of pipeline scripts.

For the **chopNod modes**:

1. Using the telescope background spectrum for the flux calibration (**Telescope Normalisation**; this is the default script), and that which the SPG script is based on
2. Producing drizzled cubes for lineScan mapping observations (**Telescope normalisation drizzled maps**), and which is also incorporated in the SPG script
3. Using the calibration block and RSRF to flux calibrate the data, including producing drizzled cubes (**Calibration source and RSRF**)
4. Pointing offset corrections for bright point sources, reducing with the telescope normalisation method (**Pointing offset correction (point sources)**)
5. Comparing on-source spectra to off-source spectra; a "helper" script, not a full pipeline (**Split On-Off**)
6. **Combine observations for a full SED**, also a "helper" script, runs the pipeline on multiple obsids and combines the results at the end (rangeScan only)

For the **unchopped modes**:

1. Using the calibration block to flux calibrate the data (**Calibration source and RSRF**). Includes a "transient correction" task for the lineScan AOTs only, and is the script on which the SPG script is based
2. Using the calibration block to flux calibrate the data and apply a more effective transient correction, for line and rangeScan (**...with transient correction**)
3. Pipeline process the on-source and off-source observations, and then subtract them (**Combing off-source with on-source**: rangeScan only). There is also a PACS useful script offered via the HIPE *Scripts#PACS Useful scripts#Spectroscopy: Off-subtraction and post-processing in unchopped range spectroscopy*

How many wavelength ranges are included in your observation, and whether you have a single pointing or a mapping observation does not matter: all pipelines handle all these cases.

To learn more about these pipeline scripts and their differences: see [Section 3.4.2](#).

To access the scripts: go to the HIPE menu *Pipelines#PACS#Spectrometer*. The scripts assume

- You know the "obsid" of your observation
- You have the calibration files on disk; normally you will use the latest update (updates are searched for automatically when you start HIPE)
- You do the red and the blue camera separately

To run the scripts,

- Read the instructions at the top, and at least skim-read the entire script before running it
- It is *highly* recommended you run line by line (at least the first time)
- To be able to edit and save the script, save the script to a new, personalised location: otherwise you are changing the script that comes with your HIPE installation. *However*, as these scripts evolve

with time, do not blindly continue to use that pipeline script for future processing: always check against the latest release of HIPE

As you run the scripts,

- Plotting and printing tasks are included, with which you can inspect the data layout or the spectra themselves
- You will be offered various ways to save the intermediate data products to a pool on disk, but saving cubes or spectra as FITS files is only possible towards the end of the pipeline, when single FITS-able products are created

In addition, the SPG scripts are provided in each of the pipeline sub-menus.

2.3. For what observations *must* I re-pipeline the data?

There are only a few cases where it may be necessary to re-pipeline the data.

1. **Spectral lines at wavelengths redder than 190 μ m**, it will be necessary to run the "calibration sources and RSRF" scripts for these observations, and moreover using a very particular version of the RSRF (relative spectral response function). See [Section 5.4](#) (chopNod) or [Section 6.4](#) (unchopped) for more explanation. You can re-reduce your data from Level 0.5 (if working from an SPG ≥ 13 observation).

The observations that have this "red leak" will be processed separately (and correctly) and will be provided via the HSA as Highly-processed data products some time in 2017. Hence you should look first for these HPDPs before reprocessing the data yourself.

2. **Updates to the calibrations** (especially if your previous reduction was several HIPE version ago). Any changes to the calibration between the time the data you have were processed and the current status will require running some or all of the pipeline. You can consult the "What's New" pages to find out what is new in each track (e.g. herschel.esac.esa.int/twiki/bin/view/Public/HipeWhatsNew14x for Track 14), and to learn how to install calibration updates go to [Section 3.6](#). If you do have data with an old calibration, you should either reprocess the data yourself, or, since the final product in the archive are those with the final calibration, get your observations again from the HSA and use the SPG products.

Note: in SPG 13 and earlier, the rangeScan observations were not flatfielded by the SPG. In SPG 14 this is now done: ranges of less than about 5 microns are flatfielded with the lineScan task and those of longer with a new version of the rangeScan flatfielding task. See the pipeline chapters to learn more.

2.4. For what observations is it *useful* to re-process the data?

For certain types of observations it is useful to try out a different pipeline script to that run by the SPG: to see if you can get a better result, to check for contamination; or to try to improve the results of some of the crucial pipeline tasks.

1. **Bright point sources**: with continuum flux levels of order 10s Jy and more. A special pipeline script for these brighter point sources where the source is located with the central 3x3 spaxels—and ideally close to the central one—is provided for pointed chopNod mode observations. This end result of this script is a calibrated spectrum of the point source, and this spectrum should be cleaner than the standard, point-source calibrated spectrum that you can obtain yourself from the cubes created by the SPG or from the spectrum tables provided in the *ObservationContext*. The

script is called **Pointing offset correction (point sources)**. You can start the pipeline from Level 0.5 if working from an SPG ≥ 13 observation. Do compared the results of this script to that of the standard script. The pipeline is explained in [Section 5.2](#).

2. **Check for contamination in the off-source pointings** for chopNod mode observations using the script **Split on-off** scripts. If the continuum level in the off-cubes is higher than in the on cubes, or spectral lines are visible, then you probably have contamination.

Another way to check for off-source contamination for chopNod mode AOTs is to run the **Telescope Normalisation** and the **Calibration source and RSRF** pipeline scripts and compare their results (using the rebinned cubes for the comparison). If there is line emission in the off-source spectra, then the spectral lines will look very different in the two sets of resulting cubes. In this case, you should favour the result from the calibration block pipeline, *while noting that this will not "get rid" of the contamination, it will at least not exaggerate its effect.*

For unchopped rangeScan observations, checking the off-source data is straightforward: compare the Level 2 rebinned cubes (HPS3DR[R|B]) from the off-source observation to the same cubes for the on-source observation. For unchopped lineScans, doing this comparison is also simple but requires a few additional lines of code as the appropriate products are not immediately available in the *ObservationContext*, but need to be created. This is explained in [Section 10.9](#).

3. **Transients correction for unchopped AOTs**: unchopped rangeScan AOTs have no transient correction in the SPG scripts, and that in the lineScan script is a first version of the correction method. A better transient correction can be found in the **...with transient correction** pipeline scripts, for line and rangeScan observations. Transients are short or mid-term effects that change the response of the detector (e.g. following cosmic ray hits), and so affect the flux levels of the spectra over short and intermediate time-scales. *Please note that new transient correction tasks are interactive and will require more than a blind running of the script.*

While there is no transient correction in the SPG script for unchopped observations, it turns out that the flatfielding (which is done in both scripts) does a good job of fixing transients anyway.

4. **Complex spectra**: with many emission lines, with absorption lines, faint spectra, or those with broad-band features. It is worth checking the results of the flatfielding task for these spectra (for this you need to re-run the pipeline: see below), possibly adjusting the flatfielding to deal better with the presence of these features.

2.5. Which are the crucial pipeline tasks?

1. **Spectral flatfielding**. The flatfielding can be a crucial task for improving the SNR of the final spectra. The flatfielding operates in the spectral domain. For each spaxel of a cube there are several discrete spectra that need to be averaged to create the final single spectrum. If some of the discrete spectra are discrepant in signal level, the flatfielding will correct this, and so improve the SNR of the subsequently-averaged spectrum, and should also smooth the continuum shape.

For very faint targets (continuum of only a few Jy and/or with faint lines) it could be worth checking the results of the flatfielding by running this task yourself, with the "verbose" pipeline script parameter to set True. For faint spectra, the correction is very difficult to compute when the continuum is near 0, and hence it is also worth checking that the results are reasonable—if the continuum of the flatfielded spectra have the same SNR as those before, it is even not worth doing a flatfielding at all.

For targets with crowded spectral lines or with absorption lines, the flatfielding may not adequately exclude the lines when fitting the continuum (a necessary step in the process). It is worth checking the results by performing the flatfielding yourself, with the "verbose" pipeline script parameter to set True. The flatfielding masks out lines as part of its process, but for spectra with multiple lines or absorption lines it may help to specify the lines to mask out via an input line list.

Flatfielding for rangeScans. The longer wavelength range of the SEDs can make it difficult to flatfield well if there are changes in slope (and especially if they are steep) in the spectra. For

some targets this left a "residual" curvature in the flatfielded spectra, especially before HIPE 14.2 when the default fitting was a polynomial. In HIPE 14.2 a spline became the default fitting and this performed better: very few observations are expected to be affected by these flatfield residuals. However, if you do see curves in the spectra of the cubes you get from the HSA, this could be due to the flatfielding.

To run the flatfielding, you need to run the pipeline script from Level 0.5. The flatfielding steps in the pipeline scripts are explained in [Section 5.2.6](#) and [Section 5.2.7](#) (chopNod line, range), [Section 6.2.7](#) and [Section 6.2.8](#) (unchopped line, range). How to compare different flatfielding attempts, and some more things to pay attention to is explained in [Section 7.4](#).

2. **Wavelength grid.** The wavelength grid used in the SPG pipeline (which is also the default in the pipeline scripts) has been chosen to give the best-looking spectra for most observations. The wavelength grid of the final cubes is created from the individual grids that are present in each of the 16 pixels that feed each spaxel, each of which is slightly offset from the others. The pipeline regularises this collection of grids into a single one, common to all spaxels and all cubes of the same wavelength setting. This regularised grid is created by the task "wavelengthGrid" with the aid of two parameters—`oversample` and `upsample`—which determine the final spectral sampling and which data-points are sampled for each bin. *It is important to note that if upsample is #1 (which it is in the SPG), the signal in the bins then become correlated as some of the same data-points feed neighbouring bins.* If this is a problem you will need to re-run the pipeline from Level 0.5 until the end, and chose the parameters of the wavelength grid differently. The interplay and effect of choosing different values of `oversample` and `upsample` is explained in more detail in [Section 7.2](#).
3. **For the unchopped modes** you may want to redo the background subtraction. For unchopped range scans the SPG uses only one obsid as the off-source, and if your observation has more than one you will need to reprocess the data yourself. For unchopped line scans (and possibly range scans), it is worth changing the `algorithm` parameter in the task "specSubtractOffPosition". For both cases you need to re-run the pipeline from Level 1: the unchopped pipelines are explained in [Chapter 6](#), and some examples of background subtraction are included in [Section 7.5](#). *However, the accuracy of the continuum level was never a priority for this AOT, and hence it is assumed that most observations with this mode were focussed on emission lines.*

2.6. Point and slightly-extended sources

For point sources, stop the pipeline at the creation of the final rebinned cubes (called "slicedFinalCubes" or "slicedDiffCubes", or HPS3DR[BS][R][B] in Level 2/2.5 of the *ObservationContext*). Then you *have to* extract your point source spectrum using one of the tasks provided to do this, since you have to remove the extended source correction factor and apply the point source flux correction(s). These tasks are explained in [Section 8.4](#) (not-centred point sources) and [Section 8.5](#) (centred point sources) and their use in the pipeline can be found in [Section 4.3](#).

The post-processing tasks for point sources are part of the pipeline scripts, and hence are documented using the terminology and product names of the pipeline ([Section 4.3](#)). However, if you want to try some of these tasks on your SPG products, as just gotten from the HSA, we have created two useful scripts that show how to run these tasks: on a pipeline product or on any observation gotten from the HSA: `Scripts#PACS useful scripts#Spectroscopy: Point source loss correction (any spaxel)` and `Spectroscopy: Point source loss correction (central spaxel)`.

Noting that all pointed observations have a spectrum table as part of the Level 2/2.5/3 and standalone browse products, in which you can find the spectrum of the central spaxel and the point-source corrected output of the pipeline task `extractCentralSpectrum`. Hence, if you have a point source in a pointed observation, *and that source is located within the central spaxel*, this spectrum table is the science-grade product you need to use. More detail on this spectrum table can be found in the [Quick Start Guide](#) and in the [PPE](#) in *PACS Products Explained*, and in the flowcharts in the *Product Decision Tree* which you can find in the [HELL](#) archive.

A few tips:

1. For *bright point sources* with continuum levels exceeding 10 Jy, we also offer the **Pointing offset correction (point sources)** interactive script for chopNod observations. The resulting spectrum should be cleaner than that gotten from the SPG. The output of this script is the "c9" or "c129" from `extractCentralSpectrum`. This task is explained in [Section 4.3](#).
2. For *very faint point sources*, where there is more noise than signal in the spaxels surrounding the central one, use the output "c1" from the task `extractCentralSpectrum` ([Section 8.5](#)). Use "c129" only if it has more flux and the same SNR as "c1".
3. For *point sources with an uncertain continuum level*: any unchopped mode observation or for spectra longwards of 190 μm : use only the result "c1" (faint sources) or "c9" (most sources) produced by `extractCentralSpectrum` (see [Section 8.5](#)).
4. For *point sources where you have SPIRE data, or several PACS bands*, and you want to push the extracted point source spectra together into a single product, you can use a useful script `Scripts#PACS useful scripts#Spectroscopy: combine PACS and SPIRE spectra`. This script is explained in [Section 8.2](#).

For **semi-extended sources**, i.e. those that are still entirely contained within the central 3x3 spaxels of the IFU (a practical limit being a diameter of about 15") and for which you have a good idea of the source morphology, there is a task to extract and calibrate its spectrum. See [Section 8.6](#).

2.7. Extended sources

For extended sources there is a choice of several different types of cubes that can be created. This depends on the AOT of the observation: whether pointed or mapping, what type of mapping observation it was, and whether it is a `lineScan` or a `rangeScan`. See [Section 1.3](#) and [Section 3.3](#) for advice on how to know the AOT. For mapping observations, the end result of the pipeline is usually a "mosaic" cube, of which we offer three: *drizzled*, *projected*, and *interpolated*. For pointed observations, the end result is either the same type of cube as used for point source work (a *rebinned* cube) or a spatially-resampled version of that (*interpolated* cube).

If you want to create a cube that is not present in the `ObservationContext` gotten from the HSA (since only the two of the three possible cubes are provided), or re-create cubes with a different spaxel size, but do not wish to run a full pipeline script, you can follow two PACS Useful scripts: `Scripts#PACS useful scripts#Spectroscopy: Post-processing for extended sources`, and `Spectroscopy: Re-create the standalone browse products`.

A few tips:

1. `Drizzle` is optimised for spectral regions of 1 or 2 microns, and it may take a long time to run. `Drizzle` results for HIPE 13, from the HSA or those created using the pipeline scripts had incorrect fluxes. However, in HIPE 14 this was corrected, and all the products now in the HSA are correct.
2. `SpecInterpolate` is not recommended for oversampled maps, and while it is not optimised for Nyquist-sampled maps, the results can be used (compare to the `specProject` results first).
3. For single pointings, bear in mind always that the beam is *very much* undersampled: the spaxels are not small enough to fit at least 2 or 3 of them within the FWHM of the beam. This means that there are gaps in the spatial coverage, which will result in a loss of information—flux—such that the true source morphology can never be fully reconstructed and flux not recovered. How important this is depends on the size and morphology of your source. It is recommended that you compare your results to those obtained from photometry where-ever possible.

Once you know what cube you want to work with, and where to get it from—Level 2.5 for unchopped range observations, Level 2 for all others—you can then work with your cubes with various tools in HIPE. Some useful scripts have been written to introduce you to the tools that HIPE provides for inspecting and fitting cubes: see [Section 1.6](#).

2.8. Correcting for the uneven illumination: the forward modelling tool

The spaxels of the PACS IFU are not evenly illuminated, with the result that there is effectively some flux loss between the spaxels over most of the PACS wavelength range. To correct for this an extended source correction was created, and this is applied to *all* observations by the pipeline. This fully corrects the integrated fluxes for all fully-extended sources (those with a flat or gradual flux gradient of no more than 20% within a single 47" square FoV). For point sources sources, this correction is taken out before the point source corrections are applied, and the fluxes will still be correct. This also applies to semi-extended sources if using the extended-to-point correction in HIPE. However, the integrated flux derived in an aperture in spectral cubes that contain:

- crowded fields
- off-centred point sources
- semi-extended sources not processed through the extended-to-point correction in HIPE
- sources which are extended and with flux gradients

will not be correct. The inaccuracy in the flux will depend on the morphology of the source at the wavelength of your observation and its coupling to the detector's beam efficiencies on every pointing (mapping) pattern. Hence, if you do aperture photometry on your source (from small sources to the entire field), you could end up under- *or* over-estimating the flux in the aperture.

To estimate the inaccuracy in the measured flux, if you know (or can model/estimate) the surface brightness distribution of the source (i.e. its morphology at the wavelength of your observation), you can apply a "forward projection" in HIPE: this takes in your input surface brightness model/image, and working with the pointing/mapping pattern of your observation, it produces a result which folds in the uneven illumination. You can then compare the modelled result to your observed result. See [Section 9.5](#) (extended sources) and [Section 8.7](#) (point and small sources) for more information. Scripts and explanation for this tool will be provided on the PACS documentation pages of the Herschel web-site, currently at herchel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb, and the Herschel Explanatory Legacy Library pages. You can also raise a HSC helpdesk ticket to request the "FMT" package.

2.9. Where can I learn about the errors in my spectra?

An explanation of how error estimates are provided for PACS spectroscopy is given in [Section 7.7](#). You are encouraged to read this section carefully.

All the cubes at Level 2/2.5 have an error array, but they are created differently.

- Rebinned cubes (HPS3DR[R|B]), aka *PacsRebinnedCubes*. These cubes are created from *PacsCubes*. In each spaxel of a *PacsCube* there is a collection of (at least 16x2) spectra, all of the same wavelength range but gathered by 16 different instrument detectors and during subsequent runs on the grating. These individual spectra are combined into a single spectrum, for each spaxel, by averaging along an input regular wavelength grid, with bin-sizes slightly larger than the separation of the data-points and optimised to the wavelength. Each bin in the wavelength grid of the output rebinned cube, therefore, has a contribution from several data-points from the input *PacsCube*. The scatter in these data-points is computed and becomes the "stddev" array of the rebinned cube. Since PACS spectral sampling is highly redundant over most of the spectral range, this can be considered a measure of the noise in the input spectrum due to a combination of stochastic (photon) noise and instrumental noise.

The task that creates the wavelength grid that is used for the rebinned cube has the parameters "upsample" and "oversample" that determine (i) the bin sizes as a fraction of the spectral resolution and (ii) the stepping forward along the input *PacsCube* wavelength array as the data-values of each bin are computed. These two parameters affect the spectral sampling and also the degree of dependence of the bins on their neighbours (i.e. how many data-points are shared between bins). Different values of upsample and oversample will result in a slightly different appearance of the resulting spectra—the line shapes and the apparent noise. The values chosen by the SPG pipeline are determined by the density (degree of redundancy) of the spectra: there is one set of values for all line scans and range scans performed with High density, and another set of values for range scans performed with Nyquist density, and all SEDs. This is explained in more detail in [Section 7.2](#).

- Drizzled cubes (HPS3DD[R|B]) are created by the task drizzle, which works on *PacsCubes*. It also takes the scatter in the *PacsCubes* data-points along the same wavelength grid used to create the rebinned cubes, and propagates them using the standard error propagation rules, slightly modified to work with the drizzle method. For more information on how drizzle handles errors, see [Section 7.9](#). The result is placed in an array called "error".
- Projected cubes (HPS3DP[R|B]) are created by the task specProject. This task propagates the stddev array of the rebinned cubes using standard error propagation, i.e. for each projected spaxel (which are smaller than the spaxels of the rebinned cubes), the error is $\text{SQRT}([\text{stddev of rebinned spaxel}] \times (\text{weight of the projection of that rebinned spaxel onto the projected output pixel})^2)$. The result is placed in an array called "error".
- Interpolated cubes (HPS3DI[R|B]) have an error array created by an interpolation of the stddev array of the rebinned cube, using the same algorithm that interpolates the flux array. The interpolation error (specInterpolate *estimates* the fluxes between irregularly gridded datapoints) is not included.

For point sources extracted from rebinned cubes using the point-source tasks, you will find the following:

- extractCentralSpectrum: works on the rebinned cubes. The stddev array(s) of these cubes is(are) propagated by the standard rules, e.g. the errors are combined for the output spectra which are a combination of several spaxels (c9 and c129). The output spectra all have a weights array, which is the propagated stddev of the rebinned cube but inverted: $1/\text{stddev}^{**2}$
- extractSpaxelSpectrum: works on the rebinned cubes, and does propagate the stddev to a weights array. The associated task pointSourceLossCorrection, used when the extracted spaxel spectrum is to be point-source calibrated, also propagate errors.

Calibration uncertainties are *not* included in these errors. The calibration uncertainties are given in the [Quick Start Guide](#), and information about additional corrections necessary for point, semi-extended, and extended sources can also be found there.

Chapter 3. Setting up the pipeline

3.1. Terminology

Level 0 products are raw and come straight from the satellite. **Level 0.5** products have been partially reduced and corrected for instrument effects generally by tasks for which no interaction is required by the user. **Level 1** products have been more fully reduced, some pipeline tasks requiring inspection and maybe interaction on the part of the user. **Level 2** products are fully reduced, including tasks that require the highest level of inspection and interaction on the part of the user. **Level 2.5** products can be found for unchopped range observations, where the separate off-source observations have been subtracted from their associated on-source observations. **Level 3** products are provided for chopNod pointed SED observations, and are a concatenated table of spectra from all the observations taken to cover the entire SED (usually 2 or 3 obsids).

The *ObservationContext* is the product class of the entity that contains your entire observation: raw data, SPG-reduced products (SPG: the automatic pipeline run at the HSC), calibration products the SPG used, auxiliary products such as telescope pointing, and etc. You can think of it as a basket of data, and you can inspect it with the **Observation Viewer**. This viewer is explained in the [HOG](#) chap. 15, and what you are looking at when you inspect a PACS *ObservationContext* is explained in the [PPE](#) in *PACS Products Explained*.

Frames is the class that contains Level 0, 0.5 and 1 astronomical data; the pipeline will start on the Level 0 *Frames* taken out of your *ObservationContext*. Also in Level 1 and in Level 2 are various cubes: *PacsCube*, *PacsRebinnedCube*, and three types of *SpectralSimpleCube*: projected, interpolated, and drizzled cubes, and each can have an "equidistant" variations (see the [PPE](#) in *PACS Products Explained*). Moreover, in the pipeline the data are held in a "sliced" form: at the end of Level 0.5 the single *Frames* product split into several separate *Frames*, of different wavelength, nod, pointing etc., and these are then taken through the rest of the pipeline. This makes handling by the pipeline tasks more efficient. The slices are held in products of class *SlicedFrames*, *SlicedPacsCube*, *SlicedPacsRebinnedCube* or *ListContext*.

For the spectrometer detector we have *pixels*, *modules* and *spaxels*. The "pixels" are the basic unit of the detector, there being 18 pixels arranged in 25 columns, each column being a "module". The "pixels" are also known as the "spectral pixels", since between the central 16 of them the entire spectrum asked for in the observation is detected (pixel 0 and 17 are not used for science). Each module of the detector is converted into a single "spaxel" on the sky. The "spaxels" are also known as spatial pixels, these are the basic unit of an Integral Field Unit such as PACS. Each spaxel is 9.4"x9.4" on the sky and contains an entire spectrum.

The following (Help) documentation acronyms are used here: [DAG](#): the Data Analysis Guide, a guide to the data analysis tools in HIPE for all types of Herschel data; [PDRG](#): this PACS Data Reduction Guide; [HOG](#): HIPE Owner's Guide; [PPE](#) in *PACS Products Explained*: the PACS Products Explained, which shows what products are contained in a PACS observation and what the various layers, meta data, and etc of these products are.

Text written *like this* refers to the class of a product (or to any product of that class). Different classes have different (java) methods that can be applied to them and different tasks will run (or not) on them. See the [SG](#) to learn more about classes. Text written *like this* refers to the parameter of a task.

3.2. Getting and saving your observation data

The fastest ways to get an observation into HIPE were explained in [Section 1.2](#). Here we add a longer guide to getting and saving Herschel data.

- **Get an observation directly from the HSA into HIPE** on the command line

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

```
# to load into HIPE:
myobs = getObservation(obsid, useHsa=True)

# to load into HIPE and at the same time to save to
# the "MyHsa" directory (HOME/.hcss/MyHsa):
myobs = getObservation(obsid, useHsa=True, save=True)
```

See the *DAG* [sec. 1.4.5](#) for more information on `getObservation` (for example, how to log on to the HSA before you can get the data, and more about "MyHSA").

Use the parameter `save=True` in `getObservation` to save the data after they are gotten. They are saved to a **pool** on disk, located off of your "MyHSA" directory (HOME/.hcss/MyHsa). You do not need to specify anything else: the data go to this disk location with a naming and organisation that are predetermined.

This method is useful for single observations and brings the data directly into HIPE in the format that the PACS pipeline requires (i.e. an *ObservationContext*).

- **To save your observation to disk** if you loaded your data into HIPE using `getObservation` *without* `save=True`, use `saveObservation`. Your data is placed either in a pool in your **local store** (HOME/.hcss/lstore) or in a specified location.

```
# "myobs" is the name of the ObservationContext
# To save to your "local store" disk location:
saveObservation(myobs)
# To save to another disk location
pool1 = "/Volumes/BigDisk/"
pooln = "NGC3333"
saveObservation(myobs, poolLocation=pool1, poolName=pooln)
```

To save the observation to the local store with a default (sub)directory name that is the observation identification number (the 1342 number), use the first command given above. Otherwise use the second. For both, the directory will be created if it does not already exist.

Note: if using `saveObservation`, the *ObservationContext* is not saved by default with the calibration tree it was reduced with by the SPG. If you wish to do that, set the parameter `saveCalTree=True` (see [Section 3.6.5](#)). It is not *necessary* to do this, since you can instead just note down the version of the calibration tree used, which you can load from disk any time.

- **Download a tar file from the HSA.** See the *DAG* [sec. 1.4.7](#) for more information. This is the method to use if you have several observations, or a very large one. Once you have the tarfile on disk, you will need to import the data into HIPE. This is explained in the *DAG* [sec. 1.5](#). To summarise:

```
# Get the data from the HSA as a tarball
# On disk, untar the tarball, e.g.
cd /Users/me/fromHSA
tar xvf me1342.tar

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

# obsid is necessary only if more than one observation
# is in that directory
```

You can chose to keep the data on disk as they are, in the directory they were untarred into. The format of the data on disk is different to the pool format that `getObservation(save=True)` or `saveObservation` create, but as long as you always remember where you put your data and what the obsid is, you can get it back into HIPE using `saveObservation`, no matter how you saved it to disk.

- **Import directly into HIPE while on the HSA.** This is useful for single observations and can bring the data directly into HIPE as an *ObservationContext*. This is covered in the *DAG* [sec. 1.4.5](#) ("[using the GUI: HUI](#)"). Briefly, you need to find your observation (e.g. by obsid), then in the query results

tab of the HUI, click the download icon to access a "Send to HIPE" menu, from which select to download "All".

- **To get the data back from disk into HIPE:** If you data were saved to disk with `saveObservation` or with `save=True` in `getObservation`, then to access them again you can use `getObservation`.

If you saved the data to disk with the default name and location (which will be MyHSA using `getObservation`, or `[HOME]/.hcss/lstore` using `saveObservation`), then you need only specify the `obsid`:

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

If you used `saveObservation` with a `poolName` and/or `poolLocation` specified (i.e. you did not save to not the default location) then instead you type:

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

The *DAG* explains these parameters of `getObservation`: see its [sec. 1.7](#) to know how to access the data if they were saved elsewhere than your local store, or to learn about the GUI methods.

The reason for these several ways to get and save data, and the different formats that data are stored on disk as, is partly historical and partly to provide a range of useful methods for users.

The full set of parameters for `saveObservation` and `getObservation` can be found in the HCSS URM, for example [here](#).

3.3. What type of observation do you have?

There were five observing modes for PACS spectroscopy, differing in the wavelength range covered and in the way the background was measured. How you process the data will depend on the observing mode. All modes had a pointed version (one single position) and a raster version (tiling/mapping).

- **chopNod line scan** (a.k.a "chopped"). A scan of a narrow wavelength range with deep spectral sampling, centred around a central defined wavelength (often specified as a particular spectral line taken from a library of lines). The sky background was measured through a fast modulation ("chopping") between the source and a near-by background (off-source) using the internal PACS chopper mirror. This was done at two telescope pointings (two nod positions); hence the name "chopNod".
- **chopNod rangeScan**. The same observing method as for chopNod line scan, but with a longer wavelength range covered, with a choice of a deep or a Nyquist spectral sampling. This mode also allowed one to cover the entire SED (this was usually done with two observations).
- **Unchopped line scan**. A scan of a narrow wavelength range with deep spectral sampling, centred around a central defined wavelength (often specified as a particular spectral line taken from a library of lines). The sky background was measured by repeating the same scan at a clean sky position (off-source) just after the source had been observed. This mode was used for sources without any near-by source-free emission.
- **Unchopped rangeScan**. The observing method as for unchopped line scan, but with a longer wavelength range covered, with a choice of a deep or a Nyquist spectral sampling. This mode also allowed one to cover the entire SED. However, in this mode the off-source observation was a separate obsid, rather than being the end of the same observation.
- **Wavelength switching**: this mode was not offered for very long into the mission, and it is reduced in the same way as the unchopped line scan data are. This mode was also for targets with no near-by clean background, and the switching was done in wavelength space rather than between the sky and the target.

A convenient way to see the observing mode, and other relevant information about the observation you are looking at, is with `obsSummary`:

```
obsid = 1342196875
myObs = getObservation(obsid, useHsa=True)
obsSummary(myObs)
```

Or look at the "pacsObsSummary" in the *ObservationContext*. In the 'AOT and instrument configuration' section of the output you will find information about the observing mode: LineSpec or RangeSpec, Mapping or Pointed, Chop/Nod or Unchopped:

```
AOT and instrument configuration:
AOT:                PacsLineSpec
Mode:               Mapping, Chop/Nod
Bands:              B3A R1 (prime diffraction orders selected)
Is bright:          YES (shortened range mode)
Raster lines:       5
Raster columns:     5
Raster line step:   14.5 (arcseconds)
Raster point step: 16.0 (arcseconds)
Chopper:            large throw
Nod cycles:         1
```

The different types of pointing "Mode" are:

- **Oversampled mapping:** in the blue, steps of up to 3.0" with a 3x3 raster; in the red steps of up to 4.5" with a 2x2 raster. In this mode the PSF of PACS is over-sampled, and it is possible to reconstruct the appearance of a source by combining (via a mapping) the individual pointings into a larger cube. The SPG pipeline uses the tasks `drizzle` (for line scans) or `specProject` (for rangeScans) for these cubes, and these tasks are also recommended in the interactive pipeline scripts. The name of the SPG products in an *ObservationContext*, in Level 2 or 2.5, are **HPS3DDR** (Herschel PACS spectroscopy 3D drizzled red [or blue]) and **HPS3DPR** (projected red [or blue]).
- **Nyquist mapping:** in the blue, step sizes of up to 16"x14.5" with a 3x3 raster; in the red step sizes of up to 22"x24" with a 2x2 raster. In this mode the beam (which is a convolution of the PSF with the spaxels' footprints) is Nyquist sampled—a worse spectral sampling than for the oversampled mode but good enough to still use `drizzle` or `specProject` to map the rasters into a single cube.
- **Undersampled mapping:** any step size larger than Nyquist (in either direction), or where the number of raster steps is less than required for Nyquist or oversampled. For these cubes the beam is not fully sampled and hence while it is possible to create a "mosaic" cube using interpolation (rather than mapping), the appearance of the target cannot be properly reconstructed—too much information (flux) is missing where there are gaps in the sampling of the beam pattern. In the SPG pipeline (and repeated in the interactive pipeline scripts) we provide the interpolated and projected mosaic cubes for this mode; the name of the interpolated cubes in an *ObservationContext*, in Level 2 or 2.5, is **HPS3DIR** (Herschel PACS spectroscopy 3D interpolated red [or blue]).

Note that it is possible to have one type of mapping in the blue and another in the red: for example, if the observer requested a 22"x24" raster as their line of interest was in the red, then the SPG will consider the red data to be Nyquist sampled but the blue data to be undersampled. Therefore the SPG will provide projected and drizzled red cubes but projected and interpolated blue cubes. You can easily create any types of cube yourself via the pipeline scripts or a useful scripts: see [Section 2.7](#).

- **Single pointing:** for these observations the science should be done on the rebinned cubes (**HP-S3DR[R|B]**: Herschel PACS spectroscopy 3D rebinned cubes red [or blue]). However, the rebinned cubes have an irregular footprint and can be difficult to load into cube viewers outside HIPE. Therefore we also provide interpolated cubes (**HPS3DI[R|B]**) for this mode.

It is possible for an observation to be oversampled in one direction but Nyquist or undersampled in the orthogonal direction. In these cases the mapping is done according to the lowest spatial sampling.

For more information on these modes and the tasks that create their cubes, see [Chapter 9](#).

3.4. The pipeline menu

3.4.1. Where are the scripts?

In the following chapters we describe how to run each of the spectroscopy pipelines. A summary of the various pipeline scripts can be found in the Spectroscopy Launch Pad ([Section 2.2.2](#)). The pipelines take you from Level 0 (raw) to Level 2 (fully-processed) or Level 2.5 (for background subtracted for unchopped rangeScan AOTs). The pipelines can be found in the HIPE *Pipeline* menu, as these figures show:

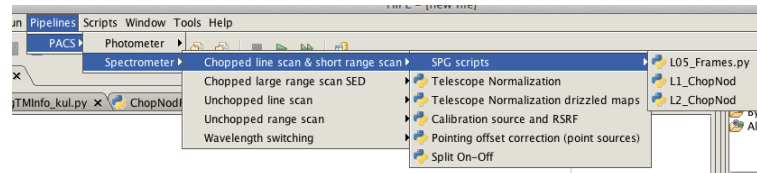


Figure 3.1. Pipeline menu 1: chopNod line scan

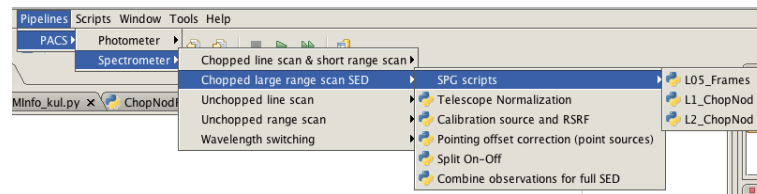


Figure 3.2. Pipeline menu 2: chopNod rangeScan

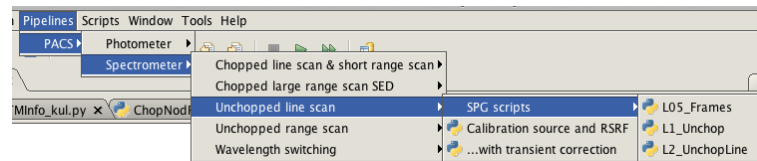


Figure 3.3. Pipeline menu 3: unchopped line scan

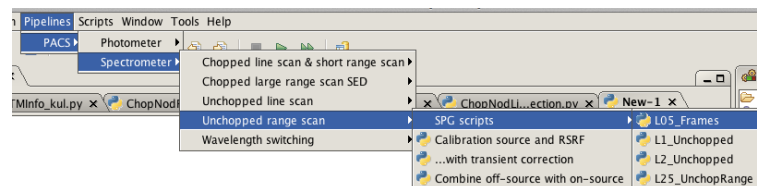


Figure 3.4. Pipeline menu 4: unchopped rangeScan

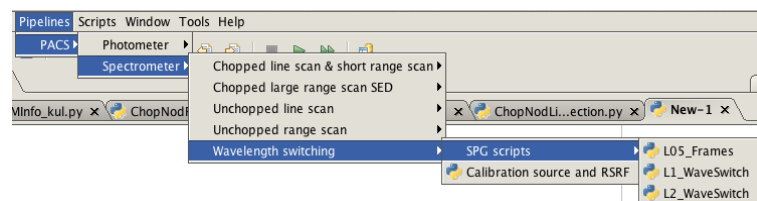


Figure 3.5. Pipeline menu 5: wavelength switching

Select the pipeline script you want and it will open in the *Editor* pane (noting that the SPG scripts are not intended for interactive use, they are provided only for completeness). From there you can edit it and run it, but if you want to save your edits you should save it to a different location on disk—otherwise you are changing the HIPE copy of that pipeline script.

The scripts start with an explanation and some set-up. They can be run all in one go or line-by-line, to be used variously depending on your level of experience. The scripts contain all the pipeline tasks,

pipeline helper plotting tasks, and basic descriptions of what the tasks are doing; the individual pipeline tasks are more fully described in the PACS [URM](#).

The "SPG Scripts" sub-menu is the PACS standard product generation [SPG] pipeline which is run, in automatic mode, by the HSC: those in HIPE 14 will produce "SPG 14" products. These pipelines scripts do not differ much from the interactive ones which are described in this PDRG.

We remind you here that you should consult PACS documentation web-page, hosted on the Herschel site ([here](#)), for information about the calibration of PACS data such as spectral leakages, sensitivity, uncertainties, ghosts, saturation limits, and PSFs.

3.4.2. What are the differences between the pipeline scripts?

As shown in the figures in the previous section,

- In the "*Chopped line scan and short rangeScan*" menu are several interactive pipeline scripts: "Telescope normalisation", "Calibration sources and RSRF", "Telescope normalisation drizzled maps" and "Pointing offset correction (point sources)". This menu is for chopNod observations of line scan and short rangeScan (<5 microns) mode. The additional script "Split On-Off" is a helper script, which can be used to produce cubes of on-source and off-source data to check for off-source contamination.
- In the "*Chopped large rangeScan SED*" menu are: "Telescope normalisation", "Calibration sources and RSRF", "Single obs", "Background Normalisation", "Pointing offset correction (point sources)". The additional helper scripts are: "Combine observations for a full SED", to run the chosen pipeline script on two or more observations and combine the final cubes and any point-source calibrated spectra into a single product, and "Split On-Off", which can be used to produce cubes of on-source and off-source data to check for off-source contamination. This menu is for chopNod rangeScan AOTs (from 5 microns up to the full SED).
- In the "*Unchopped line*" menu: "Calibration sources and RSRF" and "...with transient correction". This menu is for line scan unchopped observations. The first script includes the original transients correction task written for this mode, which corrects the transient that can occur at the beginning of an observation. The second script is new to HIPE 13, and includes a more complete transients correction task.

This new pipeline with the new transient correction tasks should not be used blindly: test the results carefully.

- In the "*Unchopped range*" menus: "Calibration sources and RSRF", "...with transient correction", and "Combine on-off". This menu is for all rangeScan unchopped observations, no matter how long or short the range. The first and second scripts reduce any single observation, and the third is to subtract the off-source observation results (the background) from the on-source observation results. There is also a PACS script do this subtraction: *Scripts#PACS useful scripts#Spectroscopy: off-subtraction and post-processing in unchopped range spectroscopy*. The first script does not include any transients correction. The second script was new to HIPE 13, and applies a new and comprehensive transients correction.

This new pipeline with the new transient correction tasks should not be used blindly: test the results carefully.

3.4.2.1. The SPG scripts vs the interactive scripts

The SPG scripts are provided in the pipeline menus for each AOT, however they are not intended for interactive use. They were provided in previous versions of HIPE for users to check what had been done on their *ObservationContext* by the SPG of that version of HIPE, as in previous versions of HIPE there were some tasks not run by the SPG. From HIPE 14 onwards, all tasks of the standard pipeline are those of the SPG pipeline.

3.4.2.2. When to use line and when to use range

For chopNod AOTs, the "line scan" menu scripts are to be used with Line spectroscopy AOTs and Range spectroscopy AOTs with a range of less than about 5 microns. The flatfielding from the line scan script performs better for these short ranges than the flatfield of the rangeScan script. This is what is also done in the SPG scripts.

For unchopped AOTs, you always have to use scripts from the "line scan" menu for Line spectroscopy AOTs and scripts from the "rangeScan" pipeline for Range spectroscopy AOTs, even for short range scans. Later we demonstrate how to use the flatfielding task of the line scan pipeline script for the shorter range scans. This is what is also done in the SPG scripts.

More on the differences in flatfielding for line and rangeScan AOTs can be found in [Section 7.4](#).

3.4.2.3. The pipeline scripts "Telescope normalisation" and "Calibration sources and RSRF"

This "Telescope normalisation" pipeline script has been the default and the SPG script since HIPE 13. Before that it was the "Calibration sources and RSRF" script. These two methods differ in the way the telescope background is handled and the data are flux calibrated.

Both scripts are valid, they are just different. This earlier script uses the data from the calibration block, taken in the beginning of all observations at key wavelengths, together with the relative spectral response function (RSRF), to flux calibrate the science spectra. The RSRF was computed before the launch of Herschel; the RSRF and the spectra of the calibration blocks were monitored throughout the mission with observations of calibration sources.

However, this pipeline script was never the preferred script for sources with low continuum flux levels (less than about 10 Jy) or of long duration. To deal better with these observations, the "Telescope normalisation" script was developed, and this one is now preferred for *all* chopNod AOTs. Instead of using the RSRF and the calibration block to flux calibrate the spectra, it uses the off-source spectra to background subtract *and* to also relatively spectrally response-correct the on-source spectra, resulting in a spectrum corrected for the background but in units of "telescopes". These units are then turned into Jy using a calibration file that contains the spectrum of the telescope in Jy. Since the off-source datapoints and the on-source datapoints are taken right next to each other, any detector response drifts or jumps will be better 'corrected out' than with the calibration block method. It produces results that are cleaner (especially for broad-band features), and better for the long-duration and faint observations. For all observations, the band-matching is also better.

Note that the absolute flux levels of the two methods have been calibrated to return the same results. The spectra produced by the calibration block and the background normalisation scripts may show slight differences in the continuum shape, but the spectral line (integrated) fluxes should not differ strongly. If they do, it could be that you have contamination in the background (off-source) pointings. In this case, you should not only check for the contamination but also prefer the calibration block result: this script cannot remove the contamination, but it will not also be incorrected in the flux calibration it applies.

To measure the fluxes of spectral lines longer than 190 μm , you *must* use the "Calibration sources and RSRF" script: see [Section 5.4](#) (chopNod) or [Section 6.4](#) (unchopped).

3.4.2.4. Telescope normalisation drizzle maps

New to HIPE 14 and higher: because of an oversight that lead to an incorrect calibration of drizzle cubes created by HIPE version 13 (see the HSC pages which are currently at herschel.esac.esa.int/twiki/bin/view/Public/DpKnownIssues and herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb) when using the Telescope normalisation method, it is now necessary to calibrate the data when creating drizzle cubes slightly differently to when calibrating any other type of cube. The drizzle cubes require a calibration applied to the Level 1 *PacsCubes*, for which one par-

ticular calibration file has to be used. All other cubes (rebinned, projected, and interpolated) require a calibration applied to the final *PacsRebinnedCubes* of Level 2, for which a slightly different calibration file is necessary. The two calibrations differ only very slightly: the noise level is very slightly higher for the drizzle cubes, but the overall flux levels, line fluxes, and continuum slopes, will be the same. However, this means that for mapping line scan chopNod observations for which drizzle cubes are required, the "Telescope normalisation drizzle maps" pipeline script needs to be used rather than the "Telescope normalisation" script.

If using the "Calibration sources and RSRF" pipeline script, no change from previous HIPE versions for drizzled cubes is necessary, and these can be created from within the pipeline script itself.

3.4.2.5. The pipeline scripts "Pointing offset correction (point sources)"

This script is for chopNod pointed observations of bright point sources (continuum #10 Jy) which are located close to the central spaxel (certainly well with the central 3 spaxels). It uses the Telescope normalisation method to do the flux calibration, but in addition applies a wavelength-dependent correction to the spectral fluxes to correct for the effect of pointing jitter; the spectra of the central spaxels (where the point source is expected to be located) are corrected. The correction is done by comparing the fluxes of the point source in the central spaxels of the cubes to the corresponding fluxes of the PACS beams, and for this reason it works best on brighter sources located closer to the centre of the FoV. The science end-product of this pipeline is an extracted, point source calibrated spectrum. Its purpose is to produce spectra that are cleaner, and hence if you are looking for broad features it is worth trying this pipeline.

We recommend you experiment with the pointing corrections of this pipeline, since they can have an important effect on the result. It is also recommended to compare the pipeline end-result spectrum with that from the "Telescope normalisation" script, e.g. using the Spectrum Explorer (see the [DAG chap. 6](#)). They should not differ too markedly.

3.4.2.6. The unchopped pipeline scripts with the old and the new transient correction

A "transient" is any temporary impact on the detector while an observation was being performed, and which affected the response detector for that of that time-line data-stream. An example would be a cosmic ray, or observing something faint immediately after something very bright. Since a transient changes the response of the detector, the flux calibration will be incorrect for the transient-affected datapoints. The usual appearance of a transient event, in the time-line spectrum, is a spike (up or down) followed by a decay back to the normal count levels. The decay can have a long or a short timescale. Transients often occurred at the beginning of an observation, immediately after the (bright) calibration block was "observed". For the chopNod AOTs, the signal we deal with is differential: subsequent data-points are subtracted from each other by the pipeline before the flux calibration is done. As the datapoints are taken *very* close in time to each other, transients generally have a negligible effect and are dealt with by the glitch detection pipeline tasks. But for the unchopped AOTs, this is not the case and the transients need to be dealt with in another manner.

Before HIPE 13, a pipeline task to correct for the transients that occurred in the beginning of the observation (after the calibration block) was done only for line scan AOTs. In HIPE 13 we provided two new pipeline scripts, which both apply a new, more comprehensive set of tasks for correcting transients occurring at any point in an observation, for line *and* rangeScan AOTs. These new tasks are presented in the menu as "...with transient correction". The transient corrections tasks should not be run blindly: the plots produced should be checked and the results should be compared to the SPG scripts: they should not differ too much.

While no transient correction is done in the SPG scripts for unchopped observations, a flatfielding is done. It turns out that the correction performed by the flatfielding task is similar to that performed by some of the new transient correction tasks. Hence, a level of transient correction is, effectively, done for the unchopped observations.

3.4.2.7. The pipeline script "Combine off-source with on-source" (unchopped range)

This pipeline script is for rangeScan unchopped AOTs. For this observing mode, the off-pointings are a separate observation to the on-source pointings. Hence, to fully reduce such data it is necessary to reduce first the on-source and then the off-source observations, and then subtract the off-source (background) from the on-source. The script starts with some set up, including setting the obsids of the on- and off-source observations, and runs the main pipeline script to reduce the data, and then does the subtraction.

There is also a PACS script *Scripts#PACS useful scripts#Spectroscopy: off-subtraction and post-processing in unchopped range spectroscopy*. The same tasks are run in the pipeline and the useful script, but the first does it in a hands-off loop while the second is run line-by-line.

3.4.2.8. "Split On-Off" testing script

This is a script offered for all chopNod AOTs. Its aim is to produce separate cubes for the on-source pointings and the off-source pointings, which can be compared to see if the off-source spectra have significant spectral features (lines) that will affect their use in subtracting the telescope background from the on-source spectra. After checking that all is well, you then need to run one of the actual pipeline scripts.

Since this script is just to check the on and off-source spectra to each other, you could run it almost blindly, i.e. just set the obsid and the camera you want to reduce, and run the rest all in one go.

3.4.2.9. The bulk-processing script "Combine observations for full SED"

For chopNod rangeScans (but aimed at full SEDs) we offer a "Combine observations for full SED" script that can be used for point sources, where at the end of the pipeline the point source spectrum is extracted, and multiple spectra arising from separate observations and from the red and blue camera are concatenated into a single spectral product.

In this script you will run a pipeline script for each observation and camera—whichever pipeline script you want to run: you should copy to a directory and edit it as necessary (setting the parameters of tasks, making the appropriate saving and plotting, etc.), since the script will then be run in one go. At the end of each running of the single observation script, it will save the various cubes and then run the point source extraction task `extractCentralSpectrum` on them, and save the final extracted spectra concatenated into a single *Spectrum1d*.

3.5. Technical considerations for running the pipeline

Some technical considerations when running the pipeline are:

- If you run the pipeline helper plotting tasks (by setting `verbose=1` in the pipeline script), most the plots will open in new windows, the ones near the end open in the Editor panel.
- The amount memory to assign to HIPE to run the pipeline depends on how much data you have, but ≥ 4 GB for sure is recommended. Consider the size of your observation, in pool, on disk, compared to the memory you have assigned to HIPE.
- To optimise memory use you can use "temporary pools" into which the sliced spectroscopy pipeline products will be read into and out of during each pipeline task, rather than being held in HIPE memory the whole time. If you know before starting HIPE that you want to use this option then you should set, in `[HOME]/.hcss/user.props`, the following:

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

If, as you run the pipeline, you run low on memory (#25% free) a pop-up will ask you if you wish to switch to using a temporary pool. If you accept then this property ("usesink") will be enforced in your current HIPE session after the ongoing task has finished. If you want to switch to using the temporary pool at any time while you are pipeline processing your data, type the following into your HIPE *Console*:

```
from herschel.pacs.share.util import PacsProductSinkWrapper
PacsProductSinkWrapper.getInstance().setUsed(1)
```

and then continue with the pipeline but you will now be using the temporary pool method.

The temporary pool will be placed in [HOME]/hcss/lstore/standard, *a directory which every now and then you will want to clear out*. The downside of this technique this is that the pipeline processing is slower, and this can be a problem when reducing multiple-repetition SEDs#however, processing such large SEDs can also be demanding on memory. If you are working on a machine with many 10s of GB of free memory, then you should be able to run without this temporary pool.



Tip

If you want to change the directory off of which the "standard" directory is placed (i.e. "lstore"), the easiest way is to change where the local store ("lstore") is held. This you can change in the HIPE preferences/properties panel, typing in the path where you want /lstore to go in *Data Access#Local Store*. Note, however, that by default *all* saving and restoring of products to pool go to this "local store" directory, so if you change this property you change where everything you save when following the pipeline will go (from the *ObservationContext* through to the *SlicedFrames* and thence on to the final cubes). If you are going to change this definition, it will be much easier to do so before you have any interaction with pools, i.e. before you begin pipeline processing.



Tip

If you need to clear up memory, you can try to delete all products you don't need

```
del(slicedFramesold, notwantedproduct)
System.gc()
#or delete all
clear(all=True)
```

where the first deletes two products and the second is a general memory clean-up command, deleting everything.

You can also temporarily clear up some memory by saving "slicedFrames" (or "sliced-Cubes") to pool, because when you do this the pointer to that product moves from HIPE memory to the pool, hence clearing out some HIPE memory:

```
name="OBSID_"+str(obsid)+"_"+camera+"a description"
saveSlicedCopy(slicedFrames,name)
```

This is only a temporary measure, because as soon as you run the next pipeline task the pointer will then move back to the HIPE memory location of "slicedFrames". But it may be useful to clear up as much memory as possible, implement the temporary pool method, and then begin with the pipeline again.

Note that you do not need to reload the slicedFrames from disk, you are saving to disk only to move the pointer "outside" of HIPE.

The syntax used here will make more sense once you have run the pipeline.

- If you wish to fiddle with your data (other than using the plotting tasks provided in the pipeline) you could consider doing that in a separate instance of HIPE. Using GUIs and creating busy plots can use quite a bit of memory.

- If you close HIPE without saving products created, they will be lost. Consider saving the pipeline results at judicious places in the pipeline script.
- The cube mosaicking task `drizzle` requires more memory and processing time than the sister tasks `specProject` and `specInterpolate`. The difference in requirements is not dramatic if you have a single line and a relatively small raster (9 pointings or so). But we do recommend that you run with ≥ 10 GB assigned to HIPE and for larger raster observations, have patience. Output to the terminal you started HIPE from can help you to assess its progress.



Warning

Note: stopping a task using the red STOP button will not always produce immediate results. A task can take a while to stop—if it is in the middle of a loop, for example. Hence it is not a good idea to allow the memory use to stay in the red: if HIPE freezes you will have to kill it and you will lose everything you have done (and not saved).

Usually, if you stop a pipeline task then the product the task was being run on will be unchanged: the product is only changed at the very end of the task's processing.

3.6. Calibration files and the calibration tree

The PACS calibration tree is updated (online) whenever an update is ready, and loaded onto your disk via the updater. It can then be loaded into HIPE via a command (you will see this as you run the pipeline). You also get a calibration tree with the *ObservationContext* that you get from the HSA. This is the calibration tree that the SPG used when it created your *ObservationContext*, i.e. it is dated from that time.

When you save your *ObservationContext* to disk using the `saveObservation` task, to save it together with its calibration tree requires the use of the parameter `saveCalTree`, set to `True`. Saving with the calibration tree takes up more space on disk and more time to work, and it is not necessary if you instead remember to note down which calibration tree that *ObservationContext*—whether you just got it from the HSA or reduced it yourself—was created with. To save a calibration tree along with the *ObservationContext*, it must be attached to the observation: see [Section 3.6.5](#).

3.6.1. Installing and updating the calibration files

First, you should consult the calibration documentation e.g. **Observer's Manual and Performance and Calibration documents** on the PACS public wiki: [here](#). Information about spectral leakages, sensitivity, uncertainties, saturation limits, ghosts and PSFs can be found there.

The calibration files are not provided with the HIPE build; if, after opening HIPE, you get a pop-up telling you to install the calibration products, it means that the calibration file set has been updated by the PACS team and you are being offered the chance to get that update. Click on "Install" and the new calibration products will be downloaded and installed. They are placed in `[HOME]/.hcss/data/pcal-community` (or `pcal-icc`, but only for the PACS team). If this is the very first time you are using HIPE and you have never installed any calibration files before, then you should select "Install", otherwise you will have no calibration files at all.

3.6.2. Checking what has been updated

The updater GUI tells you which calibration files have been changed. To see the relevant information about the release, in the calibration updater pop-up click on "Show details...". In the new panel that appears, look at the "Release Notes" tab for a summary of the new set version. In there will be listed the calibration files (the FITS files) that have been included in the update and information about the changes made.

You can also look at the individual "Files" tab to see what (if anything) has changed in the individual files that are being updated. Some files will have no information in them—most of the information is

in the Release Notes tab and in the Files tab in the files called "PCalBase_TimeDependency_FM_v#", which also contain a summary of the release.

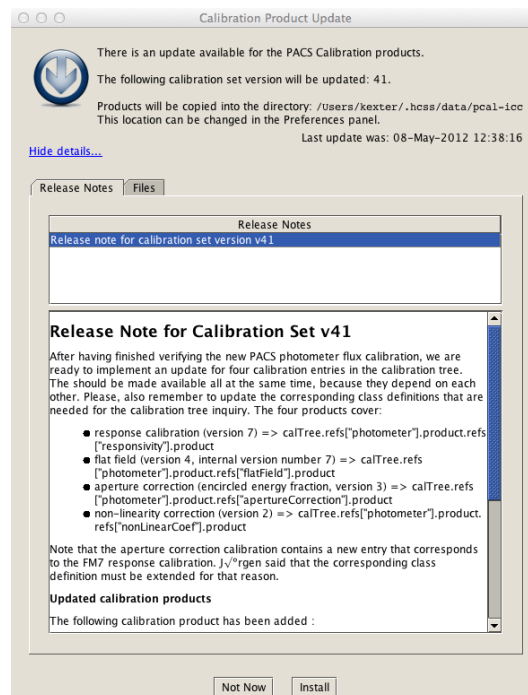


Figure 3.6. Updating the calibration files

To check which pipeline tasks this will affect, check the comments in the pipeline script which say which calibration files are used by various tasks.

The calibration files take up ≥ 1 gigabyte. To install them in a directory other than the default `[HOME]/.hcss/data/pcal-community`: in the updater pop-up click "Not Now"; go to the HIPE Preferences panel (from the Edit menu); click on *Data Access#Pacs Calibration*; in the "Updater" tab that is now in the main panel change the name of the directory in the space provided. Do not click on anything else—you do want to use the "Community Server" as these are the products that have been tested, the "ICC" ones are still in the process of being validated. Click to Apply and Close. Then go to the Tools menu of HIPE, and select *pac-cal#run Updater*. Voilà.

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

3.6.3. The calibration tree

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

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


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

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

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

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

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

```
calTree=myobs.calibration
```

Data just gotten from the HSA will always have a calibration tree attached to it. Data someone else reprocessed may not, especially if they did not save it to pool with its calibration tree attached (see [Section 3.6.5](#)).

3.6.4. Comparing calibration file versions

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

```
# The caltree that comes with you data
print myobs.calibration
print myobs.calibration.spectrometer

# The caltree you have on disk
calTree=getCalTree(obs=myobs)
print caltree
print caltree.spectrometer
```

To print out the information on the calibration tree gotten from "myobs" (the first command in the script above), it is necessary that the calibration tree is present in "myobs". This will be the case for SPG pipeline-reduced data just freshly gotten from the HSA. But if you (or anyone else) used `saveObservation` to save it to disk, then whether the `calTree` is included or not depends on whether the parameter `saveCalTree` was set to `True` or `False` when the observation was saved with `saveObservation` (see below).

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

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

To check what version of the pipeline your SPG data were reduced with, look at the Summary of the *ObservationContext* (using the Observation viewer, via a right-click on your *ObservationContext* in the *Variables pane* of HIPE): it will say something like "SPG v9.1.0" which means that HIPE user-release 9.1.0 and the "SPG" pipeline script found in there were used to create your SPG Levels. To see that script, you will need to get that version of HIPE and look in the pipeline menu.

3.6.5. Saving your *ObservationContext* and its calibration tree to pool

To save the calibration tree with your *ObservationContext*, you need to first attach the calibration tree to it, and then set the parameter `saveCalTree` to `True`:

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

```
# Example I: you loaded data into HIPE directly from the
# HSA and now wish to save it to disk, to the default location,
# along with its calibration tree:
saveObservation(myobs,saveCalTree=True)
# and to retrieve these both later:
myobs=getObservation(obsid)
myCalTree=obs.calibration

# Example II: you loaded data into HIPE directly from the
# HSA and now wish to save it to disk, to the default location,
# but with the latest calibration tree that you have on disk
myCalTree=getCalTree(obs=myobs)
myobs.calibration=myCalTree
saveObservation(myobs,saveCalTree=True)
# and to retrieve these both later:
myobs=getObservation(obsid)
myCalTree=obs.calibration
```

Note that it is not really necessary to save the calibration tree: the calibration files themselves are held on disk, all you need to know is the calibration tree version that was used (or which you want to use) to reduce the data.

Chapter 4. The pipeline: the first and the last steps

4.1. Introduction

The very first part of all pipelines—level 0 to 0.5—and the very final, post-processing parts are almost the same for all the pipeline scripts. Therefore we explain these both in this chapter.

A change from the pipeline scripts in HIPE 12: some of the pipeline "helper" tasks, to print or plot information at various stages of the pipeline, have been taken out. However, they still exist in HIPE, and are still explained in [Section 10.2](#).

4.1.1. The pipeline menu

Before reading this chapter, you should read [Chapter 1](#), [Chapter 2](#) and [Chapter 3](#), which explain how to get data, how to have a quick look at them, how to find the interactive pipeline scripts, what the differences between the pipeline scripts are, whether you need to reprocess your data, about the calibration updater, what to pay attention to for different types of sources, and how much memory you should run HIPE with—this is all information which we do not repeat here.

The pipeline scripts are found in the HIPE Pipeline menu. Read [Chapter 3](#) to decide which script you want to use. The script will open in the Editor pane, and if you want to edit and save it, you will need to save it to a new location so as to not overwrite the version that comes with the HIPE build.

The following (Help) documentation acronyms are used here: [PACS URM](#): User's Reference Manual, also the [HCSS URM](#); [PDRG](#): this PACS Data Reduction Guide; [SG](#) the Scripting Guide.

4.2. The first part of all the pipelines: Level 0 to 0.5

The first part of the pipeline is where the instrument signature is removed and the data are sliced according to an astronomical logic. No user interaction is required, and this stage of processing is almost the same for all the pipeline scripts. If you are working with "SPG 14.x" data from the HSA, there is in fact no need to run this part of the pipeline, and you can go straight to [Chapter 5](#) for chopNod observations and [Chapter 6](#) for unchopped.

4.2.1. Slicing

The spectroscopy data that are input into the pipeline are held not as a single uniform whole, but rather as a *ListContext* of individual parts of the observation—these are called "slices". This slicing of the observation data follows an astronomical logic: mainly on wavelength and pointing (e.g. on- and off-source, the position in a raster,...). The class name of the sliced product is the class name of the products it contains but with a "Sliced" at the front: the *SlicedFrames* holds *Frames* products, for example.

Slicing was introduced because it makes handling the data more efficient on memory use. It also allows the users to break their observation data into logical units to process them separately, if the entirety is too large to reduce in one go. However, note that the HIPE visualisation tools will only work on single slices at a time. Consequently, when you want to visualise the data in a product, or see its contents listed, you will need to take the product out of the SlicedProduct: the *Frames* out of its *SlicedFrames*. You will be shown how to do this.



Tip

The task `selectSlices` was written to allow subsets of data, sets of slices, to be extract from a product and placed into a new one. This is no longer really necessary in the pipeline, but

may be useful for the few very heavy observations or when working on a memory-poor computer.

To select slices for other factors fill the necessary parameters with an appropriate value: for example, set `lineId` to the number that is the line-identification for the spectral line you want to work on. The task `slicedSummary` can help fill in the parameters with the appropriate values (e.g. the line identification number), e.g.

```
HIFE> slicedSummary(slicedCubes)
noSlices: 4
noCalSlices: 0
noScienceSlices: 4
slice isScience nodPosition nodCycle rasterId lineId band
dimensions wavelengths
0 true ["B"] 1 1 1 1 [2] ["B3A"] [18,25,1631] 63.288 -
63.489
1 true ["A"] 1 1 1 1 [2] ["B3A"] [18,25,1631] 63.288 -
63.489
2 true ["B"] 1 1 1 1 [3] ["B3A"] [18,25,1631] 57.362 -
57.947
3 true ["A"] 1 1 1 1 [3] ["B3A"] [18,25,1631] 57.362 -
57.947
```

To select the short wavelength range set `wavelength` to any number between 57.362 and 57.947, or set `lineId` to [3].

Consult the [URM](#) entry for the `selectSlices` to learn more about these parameters.

4.2.2. Pipeline setup

```
# >>>>> 1 Setting Up

import os,sys
verbose = True
# For new transients correction line scan pipeline script:
from javax.swing import JOptionPane
interactive = False

if ((not locals().has_key('multiObs')) or (not multiObs)):
    obsid = 1342250905

useHsa = 1
obs = getObservation(obsid, verbose=True, useHsa=useHsa, poolLocation=None, \
    poolName=None)
#if useHsa: saveObservation(obs, poolLocation=None, poolName=None)

if verbose: obsSummary(obs)
```

1. To see plots of intermediate processing stages, set `verbose` to `True`.
2. For the new transients correction pipeline script for unchopped line scan, you can chose to inspect a plot of the transient correction task and decide whether to apply the correction, or not. By default the choice is `interactive=False`.
3. Input the `obsid` of your observation; the example observation can be used to test the pipeline.
4. **getObservation**: then you get the *ObservationContext* of that observation, either from the HSA (`useHsa=1` optionally followed by saving that observation to disk) or from a pool on disk (`useHsa=0` and possibly also defining `poolLocation` and `poolName`). The use of `get/saveObservation` is explained in [Section 1.2](#).

The "if ((not locals()..." is because this script can also be run as part of another script that combines observations; this syntax means that "if this parameter has not already been set, then set it here". When running this script on a single observation, you can execute the if statements after changing the `obsid` to your value.

5. **obsSummary**: print a summary of the observation details: see [Section 10.2](#) for more information on this task.

```
# >>>>> 2 Extract the Level 0

pacsPropagateMetaKeywords(obs, '0', obs.level0)
level0 = PacsContext(obs.level0)
```

- **pacsPropagateMetaKeywords** propagates the Meta data of the to-level of the *ObservationContext* to the Level 0, so it can be carried forward to the reduced products later.
- There is a commented-out set of commands related to the pointing products: this is no longer necessary if working with observations created by SPG 13 and higher, since the pointing products that come with these observations are the improved version that the pipeline task `calcAttitude` calculates.

Next, calibration tree and getting the data.

```
# >>>>> 3 Get the calibration tree, camera, extract the
#         sliced products

calTree = getCalTree(obs=obs)
if verbose:
    print calTree
    print calTree.common
    print calTree.spectrometer

if ((not locals().has_key('multiObs')) or (not multiObs)):
    camera = 'blue'

slicedFrames = SlicedFrames(level0.fitted.getCamera(camera).product)
slicedRawRamp = level0.raw.getCamera(camera).product
slicedDmcHead = level0.dmc.getCamera(camera).product

if verbose: slicedSummary(slicedFrames)
```

- **getCalTree** gets the calibration tree (see [Section 3.6](#) to learn more). We always recommend that you use the latest calibration tree rather than the one that comes with the data.
- Set the camera ("red" or "blue").
- Extract the first sliced products to begin the pipeline processing on: the science data (*SlicedFrames*); the raw science data for a single detector pixel (used in the saturation masking task); and the DecMec data (used in some of the masking tasks).
- **slicedSummary**: prints a summary of each slice in "slicedFrames", including the wavelength information (line, band, and whether wavelengths have been calibrated or not) and pointing (raster, nod, nodCycle). See [Section 10.2](#) for a description of the output of this task.

Before beginning the pipeline, you can set up some variables that will be used when saving data.

```
# >>>>> 4 Setup for saving products

calSet = str(calTree.version)
try:
    target = str(obs.meta["object"].value).\
        replace(" ", "").replace(".", "").replace("+", "plus")
    od = "OD"+str(int(obs.meta["odNumber"].value)).zfill(4)
    hipeVersion = (str(Configuration.getProjectInfo().track) \
        + '_' + str(Configuration.getProjectInfo().build)).replace(".", "_")
except:
    target, od, hipeVersion = "", "", ""

saveOutput = True

outputDir = str(Configuration.getWorkDir())+"/pacsSpecOut/"
if (not os.path.exists(outputDir)): os.mkdir(outputDir)
if verbose and saveOutput:
```

```
print "The products of this pipeline will be saved in ",outputDir

nameBasis = str(obsid)+"_"+target+"_"+od+"_Hipe_"+hipeVersion+"\
    "_calSet_"+calSet+"_"+camera+"_rsrf"
```

- To save products at various stages of the pipeline, either SlicedProducts or single FITS files (depending on which stage of the pipeline you are at), set the basis of the filenames here: the target name, observing day, number of the calibration tree and the version of HIPE you are using, and an indication of what pipeline you are running ("rsrf", or "telescope", ..)
- Set saveOutput to True to save products. *Note that it is not possible to overwrite previous Sliced-Products saved to a pool:* if you run this script twice with saveOutput set to True, then during the second running it will stop when it is asked to write out a SlicedProduct to a pool that already exists. Either delete the directory on your disk, or change the name of the pool to save to, e.g. add a counter to it

```
counter = 1
nameBasis = str(obsid)+"_"+target+"_"+od+"_Hipe_"+hipeVersion+"\
    "_calSet_"+calSet+"_"+camera+"_rsrf"+"_"+str(counter)
```

4.2.3. The first pipeline tasks

```
# >>>>> 5 Instrumental and satellite corrections

slicedFrames = specFlagSaturationFrames(slicedFrames, rawRamp = slicedRawRamp,\
    calTree=calTree, copy=1)
slicedFrames = specConvDigit2VoltsPerSecFrames(slicedFrames, calTree=calTree)
slicedFrames = detectCalibrationBlock(slicedFrames)
slicedFrames = addUtc(slicedFrames, obs.auxiliary.timeCorrelation)
slicedFrames = specAddInstantPointing(slicedFrames, obs.auxiliary.pointing,\
    calTree = calTree, orbitEphem = obs.auxiliary.orbitEphemeris, \
    horizonsProduct = obs.auxiliary.horizons)
if (isSolarSystemObject(obs)):
    slicedFrames = correctRaDec4Sso (slicedFrames, timeOffset=0, \
    orbitEphem=obs.auxiliary.orbitEphemeris, horizonsProduct=obs.auxiliary.horizons, \
    linear=0)
slicedFrames = specExtendStatus(slicedFrames, calTree=calTree)
slicedFrames = convertChopper2Angle(slicedFrames, calTree=calTree)
slicedFrames = specAssignRaDec(slicedFrames, calTree=calTree)
```

These are the first pipeline tasks. For none is interaction required from the user.

- **specFlagSaturationFrames:** flag for saturation, creating the masks SATURATION and RAWSATURATION.

The SATURATION mask: datapoints which exceed a certain signal limit will have the flag=1 (bad) set in the relevant mask. The limit has been determined from in-flight tests on raw-ramps data and is held in a calibration file.

The RAWSATURATION mask: indicates where saturation has occurred in the most responsive pixel of the central module (spaxel). Why? The *Frames* Level 0 data that you work on are not actually the raw data, as the data underwent some processing before they were down-linked from Herschel. For these data it is not possible to be 100% certain of saturation (although to be honest, the task does do a very good job). So, for one pixel of the central spaxel (that with the strongest response and where the astronomical source is most likely to be located) we do get the raw data. specFlagSaturationFrames looks at these data, and if the signal is saturated, then for that time-point, all the pixels of all the spaxels have their rawsaturation mask filled with a 1.

However, just because the most responsive pixel is saturated doesn't necessarily mean that all are, since the other pixels are less responsive and may not be sitting on the brightest part of the source. In the pipeline scripts this mask is "activated" by default, along with most of the other masks, and you may want to check that this is not flagging out data unnecessarily. This checking and deactivating can be done at the Level 1—2 part of the pipeline.

**Note**

If you want to *include* RAWSATURATION datapoints, it is particularly important to *not* activate it when activateMasks is called before and after the flatfielding.

**Tip**

How to inspect the data for a particular mask: the fastest way is to use the PACS Product Viewer (PPV: [Section 10.8](#)), which plots your signals vs. time for each pixel of the detector and for which masks can be over-plotted. You can chose the RAWSATURATION mask from the drop-down menu, and check whether any of these data look to be saturated, and if so, whether other pixels appear to be saturated at that same point in time (at the same x-position). The sign of a saturated line is a flattening to the peak.

However, since the PPV plots data along the time-line, it can be difficult to see spectral details. You can also wait until the first cube of the pipeline has been created—the *PacsCubes*, called slicedPacsCubes—and inspect those with the Spectrum Explorer (see [Section 10.5](#)). The data masked as bad within all the masks can be plotted with this tool.

The parameter `copy=1` is necessary to decouple the *slicedFrames* produced by this task from the *slicedFrames* put into the task, which is the same *slicedFrames* in level0 (both objects called "slicedFrames" are only links to the actual data file). If you do not do this, then the *slicedFrames* in level0 is also changed by the task.

- **specConvDigit2VoltsPerSecFrames**: convert the units to V/s.
- **detectCalibrationBlock**: add to the Status table the location of the calibration block in the data time-line (see [Section 10.12](#) for information on the Status table).
- **addUtc**: convert from spacecraft time to UTC. Adds "Utc" to the Status table.
- **specAddInstantPointing**: add the pointing and position angle of the central detector pixel to the Status table. Adds RaArray, DecArray and a number of other columns to the Status table.
- **correctRaDec4Sso**: for Solar System Objects, moving the target to a fixed position in the sky.
- **specExtendStatus**: update the Status table with chopper and grating position information, and whether the datapoints are from on-source or off-source.
- **convertChopper2Angle**: convert the chopper positions to sky positions.
- **specAssignRaDec**: 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). This adds the datasets "ra" and "dec" to the *Frames* slices (see the [PPE](#) in *PACS Products Explained* for information on the datasets of the *Frames*).

```
# >>>>> 6 Pipeline helper tasks
if verbose: ppoint = slicedPlotPointing(slicedFrames)
```

A pipeline helper task that plots the full pointing track of PACS during your observation. For an explanation of this task, see [Section 10.2](#).

4.2.4. Flagging; adding the wavelengths

```
# >>>>> 7 Flagging and wavelengths
slicedFrames = waveCalc(slicedFrames, calTree=calTree)
slicedFrames = specCorrectHerschelVelocity(slicedFrames,
  obs.auxiliary.orbitEphemeris,\
  obs.auxiliary.pointing, obs.auxiliary.timeCorrelation, obs.auxiliary.horizons)
slicedFrames = findBlocks(slicedFrames, calTree = calTree)
```

```
slicedFrames = specFlagBadPixelsFrames(slicedFrames, calTree=calTree)

if verbose: slicedSummary(slicedFrames)
```

More tasks that the user need not interact with. At the end you have masks for bad pixels and for data taken while the chopper and grating are moving, and you can inspect these masks with the PPV: [Section 10.8](#).

- **waveCalc**: calculate the wavelengths, adding a "wave" array to the *Frames* slices. It also adds the mask OUTFBAND that flags as 1 (bad) any data that fall outside of the filter bands: this may occur if the observation was defined on a e.g. blue wavelength such that the red wavelength falls out of its filter band. This mask will be activated when the Level 2 cubes are created.
- **specCorrectHerschelVelocity**: correct the wavelengths for Herschel's velocity, adding the correction to the Status. Velocities are in the LSR frame.
- **findBlocks**: figure out the organisation of the data blocks (per line/band observed, per raster position, per nod...). Add this information to a BlockTable for each *Frames* and a MasterBlockTable for the *slicedFrames*: see [Section 10.12](#) for an explanation of BlockTables.
- **specFlagBadPixelsFrames**: flag for bad and noisy pixels. The masks created are BADPIXELS and NOISYPIXELS and they contain the flag values of 1 for bad data (is flagged) and 0 for good data.



Tip

You can inspect masked data using the PACS Product Viewer (PPV: [Section 10.8](#)), which plots your signals vs time for each pixel of the detector and for which masks can be over-plotted.

4.2.5. Slicing, more flagging

```
# >>>>> 8 Slicing

slicedFrames, additionalOutContexts = \
    pacsSliceContext(slicedFrames, [slicedDmcHead], removeUndefined=True, \
        removeMasked=True)
slicedDmcHead = additionalOutContexts[0]

if verbose: slicedSummary(slicedFrames)

slicedFrames = flagChopMoveFrames(slicedFrames, dmcHead=slicedDmcHead, \
    calTree=calTree)
slicedFrames = flagGratMoveFrames(slicedFrames, dmcHead=slicedDmcHead, \
    calTree=calTree)

# For the Unchopped or the Split on-off pipelines:
# On-Off pipeline
slicedFrames = specAddGratingCycleStatus(slicedFrames)
```

The final tasks at this level of the pipeline. Again, no interaction on the part of the user is required.

- **pacsSliceContext**: slices the data according to a defined astronomical—observational logic: wavelength range (LineId, Band), pointing (RasterLine|ColumnNum, NoddingPosition and IsOut-OffField) and repetition (NodCycleNum). To read more about the slicing rules, look up the task in the [URM](#). The parameters `removedUndefined` and `removeMasked` do a cleaning up.

There are two outputs from the task: `slicedFrames` and `additionalOutContexts`. The second output parameters is the "DMC header" which is extracted with the subsequent command, this is used later in the pipeline.

- **flagGratMoveFrames, flagChopMoveFrames**: flag for a moving chopper and grating. PACS data was taken continuously, even while parts of the instrument were moving, but using such data is undesirable. The masks created are GRATMOVE and UNCLEANCHOP and they contain the flag values of 1 for bad data (is flagged) and 0 for good data.

- **specAddGratingCycleStatus**: is a task you will run only if you are using the pipeline on unchopped data or the Split On-Off pipeline. It adds information to the Status table that will allow one to distinguish on- from off-chop data: in the chopNod pipeline this information is added in a later pipeline task. The [URM](#) entry for this task gives more information.

This is the end of the Level 0 to 0.5 part of the pipeline. Now you go to [Chapter 5](#) if working with chopNod observations, and [Chapter 6](#) if working with unchopped or wavelength switching observations. After that, you will return to this chapter, to the next section.

4.3. The final part of the pipeline: post-processing

There are a number of steps that can or must be performed when you get to the end of the pipeline: Level 2 (chopNod, unchopped line) or Level 2.5 (unchopped range). These post-processing steps are included in the pipeline scripts, but not all steps are appropriate for all pipelines. Here we explain all of the possible post-pipeline tasks, and you can find out from the pipeline script itself whether a task is appropriate for your observation.

The particular post-pipeline tasks provided are for: mapping observations (using specProject, specInterpolate, or drizzle), single pointing observations, point sources, and slightly-extended sources. The final cubes of the pipeline are called slicedFinalCubes (chopNod and unchopped line pipeline scripts) or slicedDiffCubes (unchopped range scripts, after the off-source has been subtracted from the on-source) and most of these post-processing tasks start from these.

For more detail on these tasks we refer you to [Chapter 8](#) for point and slightly extended sources, and [Chapter 9](#) for extended sources and mapping observations. Here we show you only how to run the tasks.

4.3.1. Extended sources

For all observations of extended sources, whatever type of mapping pattern (or indeed whether pointed or mapping), the reader should be aware of the fact that the illumination of the PACS IFU was not even, effectively one can say that there are gaps between the spaxels. This affects the accuracy of the reconstruction of the flux levels of an observed source in any type of PACS cube. See [Section 8.7](#) (point and small sources) and [Section 9.5](#) (extended sources) for more information.

4.3.1.1. Extended sources: Nyquist or oversampled mapping observations

For Nyquist or spatially oversampled range scan and line scan observations, i.e. raster observations with very small step sizes so that the PACS beam is fully sampled, the pipeline does not end with the creation of the slicedFinalCubes: you can perform a choice of final steps to combine the separate pointings of the raster into a single cube. At the same time you can change the spaxel size of your new cube to a smaller value.

The spatial sampling of PACS observations is important because the native spaxel size (9.4 arcsec) is almost as large as the FWHM of the PACS beam, and so to fully sample the beam a mapping pattern when observing must have been adopted. This is explained in the PACS Observer's Manual and the AOT release notes, both to be found on the PACS documentation page hosted on the Herschel site: currently at [here](#). How to know whether your observation is spatially oversampled, Nyquist sampled, or undersampled is explained in [Section 3.3](#).

The mapping tasks that will combine the pointings of a Nyquist or oversampled observation into a mosaic are **specProject** and **drizzle**.

- The task **drizzle** will work on line scans and very short range scans. To create drizzle cubes using the "Telescope normalisation" method (the recommended method for chop-nod AOTs), it is necessary to use a drizzle-specific pipeline ("Telescope normalisation drizzle maps"), see the advice in

[Section 3.4.2.4](#) to learn more. This pipeline script is explained in [Section 5.3](#). For the "Calibration source and RSRF" pipeline (for unchopped AOTs, and an alternative pipeline for chop-nod AOTs), the drizzled cubes are created within that same script. The actual process of creating the drizzle cubes, i.e. running the pipeline tasks, is the same in both pipeline scripts and is explained here.

The description of how drizzle works—the algorithms for doing the spectral and spatial regridding—can be found in [Section 7.9](#), and there it is also explained why it is used for short wavelength ranges only. You can also consult its [PACS URM](#) entry.

- The task **specProject** works on *all* wavelength ranges (but for the shorter ranges drizzle is the recommended task). SpecProject is offered in all the pipeline scripts. SpecProject does a "spectral projection": the spectrum of each output spaxel is a weighted combination of the spectra of the input spaxels, and the weighting takes care of (i) the smaller sizes of the output spaxels and (ii) any overlaps in spaxels from different pointings. In doing so it creates a new, regular spatial grid that covers the field of view of the raster sequence, and combines all the input cubes into a single one. The combining is done along the spectral dimension, as well as the spatial. To learn more about how it works you can consult its [PACS URM](#) entry and [Section 7.10](#).

Running specProject

This task runs with a single command, after which you can inspect the results in the Spectrum Explorer. As with all other tasks of the pipeline, this task will work no matter how many cubes, of how many spectral or spatial ranges, are contained within the input slicedFinalCubes.

To run the task:

```
slicedProjectedCubes = specProject(slicedFinalCubes,outputPixelSize=3.0)

if verbose:
    slice = 0
    oneProjectedSlice = slicedProjectedCubes.refs[slice].product
    openVariable("oneProjectedSlice", "Spectrum Explorer")
```

The spaxel size is set with the parameter `outputPixelSize` (the default value is 3 arcsec). The task works by projecting the input spaxel positions on to the output spaxel positions and averaging together, per wavelength point, when more than one cube's spaxel falls on the same output spaxel position. This task propagates the stddev dataset of the input *PacsRebinnedCubes*. The output of the task is a *ListContext* of *SpectralSimpleCubes*, one per wavelength range found in the input, each cube being a mosaic of the input cubes.

To compare the results of `specProject` and `drizzle`, run `drizzle` first and use it as an input to `specProject` so that you obtain a drizzled and a projected cube with the same WCS (the cubes can then be compared spaxel-by-spaxel). Setting the appropriate parameter in the task `specProject` will do this.

```
slicedProjectedCubes =
    specProject(slicedFinalCubes,cubeWithOutputGrid=slicedDrizzledCubes)
```

Running drizzle

Drizzle requires a set of tasks to run. It starts with the Level 1 *slicedPacsCubes*, rather than *sliced-PacsRebinnedCubes* of Level 2. The task resamples the data in the *PacsCubes* spectrally and spatially following a spectral and a spatial grid you give it, these grid both having been created with prior tasks.

The drizzle code-block:

```
# for the unchopped line scan pipeline script, begin with:
slicedDiffCubes = specSubtractOffPosition(sCubesOn, slicedRebinnedCubesOff,\
    algorithm="AVERAGE")

# for other line scan scripts, continue with
oversampleWave = 2
upsampleWave   = 4
# 3 is mentioned in the pipeline scripts, but 4 is used by the SPG and for creating
all other cubes
```

```

waveGrid      = wavelengthGrid(slicedCubes, oversample=oversampleWave, \
    upsample=upsampleWave, calTree = calTree)
oversampleSpace = 3
upsampleSpace  = 2
pixFrac        = 0.3 # for oversampled mapping observations, 0.6 for the Nyquist
spaceGrid      = spatialGrid(slicedCubes, wavelengthGrid=waveGrid, \
    oversample=oversampleSpace, \
    upsample=upsampleSpace, pixfrac=pixFrac, calTree=calTree)

slicedDrizzledCubes = drizzle(slicedCubes, wavelengthGrid=waveGrid, \
    spatialGrid=spaceGrid)[0]

if saveOutput:
    name = nameBasis+"_slicedDrizzledCubes"
    saveSlicedCopy(slicedDrizzledCubes, name, poolLocation=outputDir)

```

1. *ChopNod and unchopped scripts*

- **wavelengthGrid** is a task also run earlier in the pipeline (Level 1 to 2). It creates a set of wavelength grids, one per wavelength range in the slicedCubes. The grid is based on the wavelength coverage of the data, with a sampling that is set by `upsample` and `oversample`. These parameters are explained in the [PACS URM](#) entry for `wavelengthGrid`, and the effect of different choices of parameters on the resulting spectra is explained in [Section 7.2](#).
- **spatialGrid** creates the spatial grid to map on to, via the spatial `oversample` and `upsample` parameters (these carry the same meaning as those parameters in the `wavelengthGrid` task). The overall size of the field-of-view is taken from the data themselves.

The `pixFrac` parameter is explained in the [PACS URM](#) entry for `spatialGrid` and in [Section 7.9](#). The SPG scripts use a value of 0.6 for oversampled rasters and 0.3 for Nyquist sampled rasters (see [Section 3.3](#) and [Section 1.3](#)).

- Then you run the `drizzle` task itself.

2. *Unchopped script only*

There is one precursor task to run before continuing with the tasks explained above

- **specSubtractOffPosition** subtract the off-source data from the on-source data (i.e. to remove the telescope+sky background). Unlike its earlier use in the pipeline, where the input are the on- and off-source rebinned cubes (called "slicedRebinnedCubesAll" in the pipeline script), here the on-source cubes of class *PacsCube* (called "sCubesOn") and the rebinned off-source cubes (class *PacsRebinnedCube*, called "slicedRebinnedCubesOff") are used. This is necessary because the subsequent input to `drizzle` (and hence output from `specSubtractOffPosition`) must be of class *PacsCube*.



Tip

If your off-source pointings are significantly less deep than the on-source pointings, you may prefer to subtract a smoothed version of the off-source data. For this, recreate those cubes but with a smaller value of `oversample` in the `wavelengthGrid` task, e.g.

```

oversample = 1
upsample   = 2
waveGrid=wavelengthGrid(sCubesOff, oversample=oversample,
    upsample=upsample, \
    calTree = calTree)
slicedRebinnedCubesOff = specWaveRebin(sCubesOff, waveGrid)

```

and then carry on with `specSubtractOffPosition`.

There are a number of algorithms in `specSubtractOffPosition` for the subtraction of the off-source spectra: those closest in time to each on-source pointing, the average of all off-source pointings, or a time-weighted interpolation. This is important for observations with more than one

off-source pointing taken between a long on-source observation. The algorithms are explained in their [PACS URM](#) entry, but see also advice given in [Section 7.5](#).

3. Drizzle can also do a deglitching, using its `detectOutlier` and `clipLevel` parameters—see its [PACS URM](#) entry to learn more. By default the sigma clipping *is* done.

As mentioned previously, the task is recommended only for short wavelength ranges, a single line scan or a `rangeScan` of less than a few microns. This is why the task is not included in the `rangeScan` pipeline scripts.

4.3.1.2. Extended sources: spatially undersampled mapping observations

For tiling observations, with large steps between the pointings in the raster, the task `specInterpolate` is recommended for creating the mosaic. This task is explained in its [PACS URM](#) entry and also in [Section 7.10](#). The task is flux conserving, and works by first creating a spatial grid via Delaunay triangulation from the input grid(s), and then interpolating the fluxes from the old grid(s) to the new one.

```
outputPixelSize = 3.0
slicedInterpolatedCubes = specInterpolate(slicedFinalCubes, \
    outputPixelSize=outputPixelSize, conserveFlux=True)

if saveOutput:
    name = nameBasis+"_slicedInterpolatedCubes"
    saveSlicedCopy(slicedInterpolatedCubes, name, poolLocation=outputDir)
```

The default spaxel size is 3.0 arcsec for this task in HIPE 14 (it was 4.7 arcsec in HIPE 13). You can try other values: spaxels too small will create artificial radiating shapes in the resulting source morphology (a typical effect of over-interpolating), and spaxels too large will result in rather small fields-of-view (because `specInterpolate` does not extrapolate beyond the edge of the original footprint, and large spaxels may get "cut off").

Note that the resulting cube will be more of an approximation of the true source morphology than spatially oversampled observations can give. Large raster step sizes between pointings means that the PACS beam is not fully sampled. Since the observed flux is a convolution of the beam with the source, by missing parts of the beam you are also missing parts of the source. Hence the results should be considered more indicative than real.

4.3.2. Pointed observations of extended sources

For single pointing observations of extended sources we offer the same advice as for spatially undersampled mapping observations (see above). The task `specInterpolate` can be used on these data to create a cube with a regular spatial grid, which is easier to visualise than the rebinned cubes are, although the science measurements are better made on the rebinned cubes. An, although less recommended, alternative to using `specInterpolate` is to run `specProject` with very small (0.5 arcsec) spaxel sizes, in this way creating a cube with a regular spatial grid but with spaxels small enough that the original footprint (the 9.4 arcsec spaxels and their slightly irregular distribution) can be seen.

4.3.3. Point sources

For point sources it is necessary to work from the final `PacsRebinnedCubes` (the `slicedFinalCubes` or `slicedDiffCubes`). To extract the spectrum of the point source you must run one of two tasks: "extract-CentralSpaxel"; or "extractSpaxelSpectrum" followed by "pointSourceLossCorrection" (and preceded by "undoExtendedSourceCorrection"). The first uses the point source loss correction calibration file to correct the flux of the central spaxel for the point source losses, and can also apply a secondary correction for pointing jitter, this being based on the source data. This task also removes the extended source correction factor which is applied to all PACS data by the pipeline. The second set of tasks remove the extended source correction factor, extract the spectrum from any spaxel, and apply the

basic point source loss correction to it. These are to be used for point sources not located in the central spaxel. It is important to note that a point source correction is *necessary* to produce a science-grade result for point sources. Summing up the flux in the field of view is not sufficient.

Information about how these tasks work, and advice about when to apply these tasks and under what conditions they will *not* return a scientifically-valid result, can be found in [Section 8.5](#) (centred point sources) and [Section 8.4](#) (off-centred point sources).

4.3.3.1. Point source loss correction for sources located in the central spaxel: `extractCentralSpectrum`

This task is for point sources that are located in the central spaxel. There is a difference between line and rangeScans since for the rangeScans one can apply a wavelength-dependent jitter correction to the flux, for line scans a mean correction is used instead.

```
# Apply the point source calibration and extraction for line scan AOTs
for slice in range(len(slicedFinalCubes.refs)):
    if verbose: print
        c1,c9,c129 = extractCentralSpectrum(slicedFinalCubes.get(slice), \
            isLineScan=-1, calTree=calTree, verbose=verbose)
    if saveOutput:
        name = nameBasis +
        "_centralSpaxel_PointSourceCorrected_Corrected3x3NO_slice_"
        simpleFitsWriter(product=c1,file = outputDir + name +
            str(slice).zfill(2)+".fits")
        name = nameBasis + "_central9Spaxels_PointSourceCorrected_slice_"
        simpleFitsWriter(product=c9,file = outputDir + name +
            str(slice).zfill(2)+".fits")
        name = nameBasis +
        "_centralSpaxel_PointSourceCorrected_Corrected3x3YES_slice_"
        simpleFitsWriter(product=c129,file = outputDir + name +
            str(slice).zfill(2)+".fits")

# Apply the point source calibration and extraction for rangeScan and SED AOTs
smoothing = 'wavelet'
nLowFreq = 4
# or
smoothing = 'filter'
gaussianFilterWidth = 50
medianFilterWidth = 15

for slice in range(len(slicedFinalCubes.refs)):
    c1,c9,c129 = extractCentralSpectrum(slicedFinalCubes.get(slice), \
        smoothing=smoothing, width=gaussianFilterWidth,
        preFilterWidth=medianFilterWidth,\
        nLowFreq=nLowFreq,calTree=calTree, verbose=verbose)
    if saveOutput:
        name = nameBasis +
        "_centralSpaxel_PointSourceCorrected_Corrected3x3NO_slice_"
        simpleFitsWriter(product=c1,file = outputDir + name +
            str(slice).zfill(2)+".fits")
        name = nameBasis + "_central9Spaxels_PointSourceCorrected_slice_"
        simpleFitsWriter(product=c9,file = outputDir + name +
            str(slice).zfill(2)+".fits")
        name = nameBasis +
        "_centralSpaxel_PointSourceCorrected_Corrected3x3YES_slice_"
        simpleFitsWriter(product=c129,file = outputDir + name +
            str(slice).zfill(2)+".fits")
```

ExtractCentralSpectrum. This task works on *PacsRebinnedCubes*, hence you are working on the cubes in "slicedFinalCubes" or "slicedDiffCubes", *not* anything produced by specProject, specInterpolate, or drizzle. This task first removes the extended source correction, then extracts the spectrum from the cube, and applies one of 3 possible corrections, returning hence three spectra:

1. c1: The central spaxel spectrum with the c1-to-total point-source correction applied; this is for cases where the source spectrum is almost completely confined to the central spaxel, i.e. faint sources.

This spectrum should never be used if you have processed the data through the "point source background normalisation" pipeline script.

2. c9: The summed spectrum in the 3x3 central spaxel "box" with the c9-to-total point-source correction applied. This product should only be used if the source is bright enough to have noticeably more signal in the 3x3 spaxel box than in the central one. It is more robust against slight pointing offsets and jitter than is c1.
3. c129: A combination of the two: the flux level of c9 but the spectrum of c1. This is suitable for bright sources if the SNR of c1 is better than that of c9 but there is still noticeable flux in c9. **It should never be used for unchopped mode observations or if you have used the special RSRF to calibrate line fluxes longwards of 190 μ m** (e.g. [Section 5.4](#)). This is because the c129 spectrum is scaled by the c9 flux level, and this scaling is based on the continuum level in c9, which cannot be guaranteed to be correct for these two cases. (In addition, for these two cases you should only trust the line fluxes, *not* also the continuum flux levels.)
4. *For line and short rangeScans observations*, we recommend a wavelength-independent scaling, i.e. `smoothing = 'median'`.

For long rangeScan observations you can apply a wavelength-dependent scaling, but try it without first. For this case you will apply two filters to the correction "spectrum", and you should play with the filter types and their widths—`width`, `preFilterWidth` and `nLowFreq`—to find the smoothest result that still reflects the global shape of the correction spectrum. Alternatively ask for the wavelet filtering method instead of smoothing. The filtering/smoothing is done to the correction, not to the actual extracted spectrum. This is a second order correction: it will not improve the flux accuracy any further, it will only (slightly) correct the spectral shape of the continuum.

More detail about how these three spectra are computed is given in [Section 8.5](#), and the parameters of the task are explained the [PACS URM](#) entry: they control the smoothing and whether the scaling used to create c129 is per wavelength or one average value. The scaling curve/median is multiplied into your spectrum.

This task will only produce a scientifically-correct result if the entire spectrum in the central spaxel/central 3x3 is from the point source, and includes *no contamination*.

The task will return plots (if `verbose=1`) and some text.

1. The "Central9/Central" plot shows a blue and a green curve overplotted on the data they are created from
 - The blue is the ratio of the spectrum from the central spaxel ("s1") to the sum of the central 3x3 spaxel box ("s9"), compared to the ratio expected for a perfectly-pointed. In grey is the spectrum the blue curve is computed from. This comparison gives you an idea of how well pointed and/or how point-like your source is. The mean value of this curve is printed to screen "(median)correction", together with the relative error of the data the correction was calculated from (RMS/sqrt(n), from the grey spectrum with n being the number of datapoints in the spectrum), and the RMS in those data. **If the printed correction is greater than 20-30%, this probably means that your source is slightly extended, too faint for a good correction calculation, or not well-enough centred in the central spaxel.** If the RMS value are high, the correction is also likely to be uncertain.
 - The green is the smoothed version of the thick black curve, which is the ratio of "s9" but now point source corrected (i.e. "c9"), to the spectrum from the central spaxel ("s1").
2. A plot of all three spectra, c1, c9, and c129.

You can open the Spectrum Explorer (see the *DAG* [chap. 6](#)) on your final spectra, and you can also save them to disk as a FITS file (right-click on the product in the Variables panel to access that option, or use the `simpleFitsWriter`).

4.3.3.2. Point source loss correction for sources not located in the central spaxel

If your source is not located in the central spaxel then you can use an alternative task to extract the point source spectrum. The task can be found in the Scripts menu of HIPE (*PACS useful scripts#Spectroscopy point source loss correction*) and it is described there, where you can test it out on a public observation. The two important tasks of the script are:

- **undoExtendedSourceCorrection:** the first step is to remove the "extended source correction" (this is done automatically in `extractCentralSpectrum`, but not in `extractSpaxelSpectrum`).
- **extractSpaxelSpectrum:** a simple task that extracts the spectrum from any spaxel and returns a spectrum in the *SimpleSpectrum* format.
- **pointSourceLossCorrection:** this task takes the extracted spectrum and applies the point source correction. This correction is based on a PSF model (including the wings), and has been verified at the key wavelengths (see its *PACS URM* entry). This correction is contained in the calibration tree (`pointSourceLoss`), and is the same correction as used in `extractCentralSpectrum` for "c1".

You can open the Spectrum Explorer (see the *DAG chap. 6*) on the spectrum, and you can also save it to disk as a FITS file: right-click on the product in the Variables panel to access that option, or use the `simpleFitsWriter`.

Note that the point-source correction factor of the task `pointSourceLossCorrection` is based on the beam of the central spaxel. However, it is known that the beams of other spaxels are not exactly the same. Hence the correction will never be as good as that applied to centrally-located point sources, although it is better than applying no correction at all.

Two alternatives exist, both of which require using HIPE.

- Use the "Forward modelling tool", which will take a synthesised rebinned cube with a point-source (located in whichever spaxel) as the input model (an input spectrum is also required) and produce an output result from which a correction to the SPG product can be estimated. [Section 8.7](#) for more information on this tool.
- For offsets of less than 10", you can use the POC pipeline script: [Section 3.4.2.5](#). This pipeline script can, in fact, be used even for centred point sources: since it performs a more direct correction for pointing offsets, it can produce a cleaner spectrum than the SPG does.

4.3.4. Slightly extended sources

For sources that are slightly extended—larger than a point but fitting within a diameter of about 15 arcsec and within the central 3x3 spaxels of the FoV of a single pointing—it is recommended to use a HIPE task to extract the correct integrated spectrum. First a point source spectrum must be extracted. The recommendation is to use `extractCentralSpectrum`: using the central spaxel result ("c1") but only for the faint sources, or the more recommended 3x3 ("c129" or "c9") for all other sources and especially for those not centred in the central spaxel (but still within the central 3x3). If the source do not cross into the central spaxel at all, the point source tasks for non-central sources can be used instead: this is all explained in more detail in [Section 8.6](#). Then, a point-to-extended correction is applied, taking as input a surface brightness distribution model of the source. The task—**specExtendedToPointCorrection**—essentially modifies the point source correction for the specified source morphology. It uses an ellipse as its default model, but the user can define other morphologies. As the point source extraction tasks both take out the extended source correction that is applied to all product by the pipeline, the fluxes will be correctly calibrated (assuming your model is correct).

Chapter 5. ChopNod pipelines

5.1. Introduction

5.1.1. The pipeline menu

The pipeline menus were introduced in [Chapter 4](#). After having selected the pipeline script you want to run, and carried out the Level 0 to 0.5 parts of the pipeline ([Chapter 4](#)) you can now read this chapter: the Level 0.5 to Level 2 parts of the pipeline. The post-pipeline processing (the very end of Level 2) is then explained in [Section 4.3](#).

The differences between the pipeline scripts is not large, and we take you though all together. Your prime source of material for the pipeline should be the scripts (from the HIPE build) themselves, since these may change slightly from what is presented here.

More information on testing out some of the pipeline tasks is given in [Chapter 7](#) and information on the post-pipeline tasks is given in [Chapter 8](#) and [Chapter 9](#). How to plot and inspect your pipeline products can be found in [Chapter 10](#).

5.2. The 0.5 to 2 scripts for all pipelines

The Level 0.5 to Level 2 part of the pipeline is almost the same for all of the chopNod scripts. Here we take you through the scripts, indicating where code is for one or another pipeline script only. You should in any case have the pipeline script already open in HIPE before carrying on.

If you want to begin from Level 0.5, i.e. not run that part of the pipeline yourself (especially as this is no longer necessary), the first command is

```
slicedFrames = obs.level0_5.blue.fitted.product # for the blue camera
slicedFrames = obs.level0_5.red.fitted.product # for the red camera
```

and it is still necessary to set up the general pipeline parameters: verbose, saveOutput, calTree and if you do want to saveOutput, then also nameBasis (see [Chapter 4](#)).

5.2.1. Masking for glitches; convert the capacitance

```
# >>>>> 1 Masking for glitches; convert the capacitance

slicedFrames = activateMasks(slicedFrames, StringId([" "]), exclusive = True)
slicedFrames = specFlagGlitchFramesQTest(slicedFrames,copy=1) # for
slicedFrames = activateMasks(slicedFrames, slicedFrames.get(0).getMaskTypes())
slicedFrames = convertSignal2StandardCap(slicedFrames, calTree=calTree)
```

- **activateMasks:** this task activates (or deactivates) the indicated masks, so that the following task(s) can (or not) take them into account. The parameter `exclusive` is say what to do with the not-named masks. The first call to this task activates *no* masks, i.e. all are deactivated, and the second activates all masks that are present in the slicedFrames. Consult the [URM](#) entry of activateMasks to learn more.
- **specFlagGlitchFramesQTest:** flag the data for glitches (e.g. cosmic rays) using the Q statistical test, creating a mask called GLITCH. The deglitching task works on the time-line, and it identifies and flags out the bad data (1=is bad, 0=is not bad), it does not change them. The GLITCH mask is automatically activated when it is created. There are parameters of this task that you could play with (see its [URM](#) entry), but note that these have been much tested and the default settings are

good for practically all cases. There is in any case a later outlier detection task, which will clean up any left-over glitches.

The parameter `copy=1` is necessary to decouple the `slicedFrames` produced by this task from the `slicedFrames` put into the task, since otherwise the input is changed along with the output.

- **convertSignal2StandardCap**: converts the signal to a value that would be if the observation had been done at the lowest detector capacitance setting, if that were not the case. This is necessary because the calibrations are for data taken at the lowest capacitance.

5.2.2. Compute the dark and the response: "Calibration source" scripts only

```
# >>>>> 2 Compute the dark and response

calBlock = selectSlices(slicedFrames,scical="cal").get(0)
csResponseAndDark = specDiffCs(calBlock, calTree = calTree)
```

- **selectSlices**: to select the calibration slice out from `slicedFrames`. See `selectSlices` in the [URM](#) to learn more.
- **specDiffCs**: calculates the response and dark current. Briefly, `specDiffCs` uses a combination of standard star observations (stars, asteroids, Neptune and Uranus) contained in the calibration file `ObservedResponse`, and the signal from the calibration block observed during the observation compared to the calibration block absolute fluxes contained in the calibration file `calSourceFlux`. From these data the response of the instrument and the dark current during the observation is calculated, and this is placed in the product `"csResponseAndDark"`. This dark current is not used since it is subtracted in the subsequent task `specDiffChop`.

5.2.3. Compute the differential signal

```
# >>>>> 3 For Calibration source pipelines: subtract the off-chop
# >>>>>    from the on-chop data,
# >>>>>    For Telescope normalisation pipelines: compute their ratio

# For the "Calibration source" scripts:
slicedFrames = specDiffChop(slicedFrames, scical = "sci", keepall = False, \
    normalize=False)
# For the "Telescope normalisation" scripts:
slicedFrames = specDiffChop(slicedFrames, scical = "sci", keepall = False, \
    normalize=True)

if verbose:
    slicedSummary(slicedFrames)
    pbasic = plotSignalBasic(slicedFrames, slice=0, titleText="plotSignalBasic")
```

- **specDiffChop for the scripts using the Calibration source method**: computes the pairwise difference, $A-B$, being (on-source chops) - (off-source chops), to remove the telescope+sky background and the dark current. Masks are propagated—the bad data are still subtracted but the result is flagged as bad. This procedure will change the number of readouts in each `Frames` slice and it will remove the calibration slice from the `slicedFrames` input. See the [URM](#) entry to learn more.
- **specDiffChop for the Telescope normalisation scripts**: computes the pairwise difference-ratio, $2*(A-B)/(A+B)$, because `normalize` has been set to `True`. After this, the flux can be considered to be in units of "telescope background". The mask `INVALID` is added, where data of value 0 in the denominator (i.e. $on+off=0.0$) are flagged as bad, so they do not later cause divide-by-0 problems.
- **slicedSummary and plotSignalBasic**: see [Section 10.2](#) for an explanation of these tasks. `slicedSummary` prints a summary of the slices held in `slicedFrames`, and `plotSignalBasic` is a basic plot of the signal of the central pixel in the central module of the instrument. To plot a different pixel or module, specify the pixel and module to plot.

**Tip**

The PACS spectrometer detector array has a size of 18,25 (pixels,modules), with science data being contained in pixels 1 to and including 16 only, for the first dimension.

After running `specDiffChop` you can plot the signal with `plotSignalBasic`. To compare the before and after result, you can do any of the following:

1. Run `plotSignalBasic` on the `slicedFrames` before running `specDiffChop` and then again on the `slicedFrames` after the task: but note that the first slice in the `slicedFrames` before running this task has now been removed, so slice 0 from before is slice 1 from after.
2. Use `PlotXY` to plot these data together on the same plot: information to do this is provided in [Section 10.4.2](#).
3. Run the helper script `Split On-Off` ([Section 5.5](#)). That script runs the pipeline to this point, and then rather than subtracting the off- from the on-chop data, it separates the two and produces a cube for each at the end of the pipeline: you can then compare the source+telescope background+dark+sky cubes to the telescope background+dark+sky cubes. You will then need to run one of the full pipeline scripts to reduce your data.

5.2.4. Absolute flux calibration: "Calibration source" scripts only

```
# >>>>> 4 Absolute flux calibration

slicedFrames = rsrfCal(slicedFrames, calTree=calTree)
slicedFrames = specRespCal(slicedFrames, csResponseAndDark = csResponseAndDark,
                           calTree=calTree)
```

- **rsrfCal**: apply the relative spectral response function. The RSRF is taken from the calibration tree. See the [URM](#) entry for this task to learn more.

For spectral lines at wavelengths longer than 190 μ m, see [Section 5.4](#). For these cases you need to use a particular version of the RSRF for band R1. Note that only the Calibration source pipeline scripts will calibrate these ranges correctly.

- **specRespCal**: apply the absolute response correction, using the response that was calculated from the calibration block. The flux unit is now Jy.

You can save to disk before the next tasks are run:

```
if saveOutput:
    name=nameBasis+"_slicedFrames_B4FF"
    try:
        saveSlicedCopy(slicedFrames,name, poolLocation=outputDir)
    except:
        print "Exception raised: ",sys.exc_info()
        print "You may have to remove: ", outputDir+'/'+name
```

5.2.5. Pointing offset correction: "Pointing offset correction" scripts only

The calculation of the pointing correction for centred, bright point sources, is done here. It is a multi-step procedure, and we recommend you read the instructions before doing any processing. *We also recommend that you if experiment with the parameters of this task, use "saveSlicedCopy" to copy "slicedFrames" to disk before beginning to experiment, so you always have a "clean" copy to start again from (using readSliced).*

The pointing offset tasks are explained in more detail in [Section 7.12](#). Here we explain how to use the tasks in the scripts.

```
# >>>>> 4 Pointing offset correction

slicedFrames, background = specRespCalToTelescope(slicedFrames, obs.auxiliary.hk, \
    calTree = calTree, reduceNoise = 1)

if verbose:
    pBack = PlotXY(background.refs[0].product.wavelengths, \
        background.refs[0].product.fluxJy, xtitle="Wavelength [μm]", \
        ytitle="Flux [Jy]", titleText="Telescope Background")
    for slice in range(1,len(background.refs)):
        pBack.addLayer(LayerXY(background.refs[slice].product.wavelengths, \
            background.refs[slice].product.fluxJy))

slicedFrames = activateMasks(slicedFrames, \
    StringId(["SATURATION", "RAWSATURATION", "NOISYPIXELS", "BADPIXELS", "UNCLEANCHOP", \
        "GRATMOVE", "GLITCH"]), exclusive = True)

usePointingProduct = 1

spaxels = IntId([6,7,8,11,12,13,16,17,18])
rules = [SlicingRule("ScanDir",1)]
slicedFrames = pacsSliceContext(slicedFrames, slicingRules = rules, \
    removeUndefined=1)[0]
if verbose: slicedSummary(slicedFrames)
if saveOutput: saveSlicedCopy(slicedFrames, nameBasis+"_slicedFrames_prePOC", \
    poolLocation=outputDir)
```

- **specRespCalToTelescope**: this task computes the telescope background flux from the off-source data in the observation, and normalises the on-source data by this. It takes the telescope background that has been created from numerous calibration observations, scales it by the temperature during your observation (taken from the housekeeping, "hk", in the *ObservationContext*), and then uses that to perform the flux unit conversion, from "telescope backgrounds" to Jy. The task uses the "offRatio" calibration product. The stddev dataset is propagated by this task. See the [URM](#) entry for this task to learn more about how it works.
- A plot showing the background spectrum is created. The Masks are then activated, in preparation for the next task, with **activateMasks**.
- Masks are activated. As discussed in [Section 4.2.3](#) you may want to not include the RAWSATURATION mask in this call.
- There are two choices for computing the pointing offset correct, and which you use is determined by the value of **usePointingProduct**.
 1. **usePointingProduct = 1** (the default): the POC has two components, and this option requests that both are used: the absolute offset and the pointing jitter, as computed from the pointing products in the observation.
 2. **usePointingProduct = 0**: compute only the first component.
- Some set-up: isolate the central 3x3 spaxel box where the source must be located (otherwise this script will not work). Then the data are sliced, to reduce the memory-requirements of the subsequent tasks. Next you are offered the choice to save the product before any corrections are applied.

```
# >>>>> 4 Pointing offset correction

oversampleBeam = 11 # for line scan, 5 for range scan
smoothFactor1 = 4 # for line scan, 10 for range scan
chiSqr = specDetermineChiSquare(slicedFrames, calTree = calTree, \
    spaxels = spaxels, oversample = oversampleBeam, smoothFactor = smoothFactor1, \
    perNod = usePointingProduct)

if verbose:
```

```

d = Display(chiSqr.refs[0].product["chi_square"].data[:, :, 0])
d.setColortable("Real", "Log", "logarithmic")

pointingOffset = specDeterminePointingOffset(slicedFrames, chiSqr, \
  spaxels=spaxels, chiThreshold = 1.2, smoothWidth = 0, sigmaClip = 2.5, \
  searchMax=1, useMedianPointingInLeak=1)

if usePointingProduct:
  smoothFactor2 = 3
  pointingOffset = specDeterminePreCalculatedPointing(slicedFrames, \
    pointOffset=pointingOffset, oversample = oversampleBeam,
    smoothFactor=smoothFactor2)

if usePointingProduct:
  pointingOffset =
  specDeterminePointingOffsetFromPreCalculatedPointing(slicedFrames, \
    pointingOffset, calTree=calTree, spaxels=spaxels)

if verbose: pjitter = plotPointingOffset(pointingOffset, calTree=calTree)

```

- **specDetermineChiSquare:** the first step in the calculation of the POC: comparing the source flux distribution to that of the PACS beam, producing an array of Chi-squared values as output.

See the [URM](#) entry for this task to learn more, for this and all the tasks.

- **specDeterminePointingOffset:** uses the chiSqr to determine the pointing offsets that caused the measured flux offsets. The two parameters you can play with are the threshold of the chiSqr to work with (chiThreshold), and the width of the median box in which to smooth the signal along the timeline according to the bin sizes of the chiSquare product (smoothWidth).
- **specDeterminePreCalculatedPointing:** is executed if usePointingProduct is 1. The "gyro-propagated" pointing products are used to compute jitter pointing-offset values. The previous two steps produce an absolute POC, and this step the jitter POC.
- **specDeterminePointingOffsetFromPreCalculatedPointing:** is executed if usePointingProduct is 1. It again calculates the flux corrections arising from the pointing offsets determined in the previous steps, and adds the flux correction factors to the pointingOffset product.
- **plotPointingOffset:** plots the pointing offsets (of the measured centre of the point source with respect to the centre of the central spaxel) computed for your observation.

```

# >>>>> 4 Pointing offset correction

slicedFrames = specApplyPointingCorrection(slicedFrames, pointingOffset, \
  spaxels = spaxels)

if calTree.version < 61 or calTree.version > 64:
  slicedFrames, telBackCor = specCorrectTelescopeBackground(slicedFrames, \
    calTree = calTree)

slicedFrames = pacsSliceContext(slicedFrames)[0]
point, chi = reSlicePointingProducts(slicedFrames, pointingOffset)
del(chiSqr, chi)

if verbose: slicedSummary(slicedFrames)

```

- **specApplyPointingCorrection:** applies the flux correction computed from the pointing offset corrections. *This correction is applied only to the central 9 spaxels*, as the outer spaxels are no longer useful.
- **specCorrectTelescopeBackground:** applies a time- and wavelength-dependent flux correction to the telescope background, based on calibrations determined by the PACS team. This changes the signal in the slicedFrames by a few percent.
- Finally, the data are re-sliced back to what they were before this set of tasks was run, in preparation for the rest of the pipeline.

5.2.6. Spectral flatfielding: all line+short range scan pipeline scripts

For long range scan (#about 5 microns) and SED pipelines, go to the next section.

Spectral lines existing on top of medium or high-flux continua should benefit from refining the spectral flatfielding, and any other type of spectra should also be at least slightly improved. *It is recommended to compare spectra obtained with and without spectral flatfielding, and to also check on the results of the flatfielding as you do it.* This particularly so for spectra with weak continua.

Before doing the flatfielding, you can chose to save the slicedFrames to pool. Then the data are converted to the first of the cubes produced in the pipeline.

```
if saveOutput:
    name=nameBasis+"_slicedFrames_B4FF"
    try:
        saveSlicedCopy(slicedFrames,name, poolLocation=outputDir)
    except:
        print "Exception raised: ",sys.exc_info()
        print "You may have to remove the directory: ", outputDir+'/'+name

slicedCubes = specFrames2PacsCube(slicedFrames)
if verbose: slicedSummary(slicedCubes)
```

- **specFrames2PacsCube:** turn the individual *Frames* in the *slicedFrames* into *PacsCubes* held in a *SlicedPacsCubes* product. This really is only a rearrangement of the data. These cubes have a spatial arrangement of 5x5 spaxels (created from the 25 modules), and along the wavelength dimension are all the spectra from the 16 pixels of each module, all packed together one after the other.

The flatfielding is a multi-step process for these short wavelength range data. Spectral flatfielding is to correct for the differences in the response of the 16 pixels of each of the 25 modules/spaxels, with respect to their 25 mean values. This should improve the SNR in the continuum of the subsequently combined spectrum in each spaxel (combining is the next stage in the pipeline), and will correct for a "spiking" effect in the final spectra that can result if one scan in a pixel is discrepant. The flatfielding is performed in a few steps: (i) outliers are masked out, (ii) spectral lines are identified (so they can be ignored), (iii) the mean continuum level of each pixel is then determined, and each is normalised to the overall mean of the spaxel they belong to, and (iv) then masks and intermediate results are cleaned up.

```
# >>>>> 5 Spectral Flat Fielding

# 1. Flag outliers and rebin
upsample = 4
# 3 is mentioned in the pipeline scripts, but 4 is used in the SPG and elsewhere
waveGrid=wavelengthGrid(slicedCubes, oversample=2, upsample=upsample,
    calTree=calTree)
slicedCubes = activateMasks(slicedCubes, StringId(["GLITCH","UNCLEANCHOP",\
    "NOISYPIXELS","RAWSATURATION","SATURATION","GRATMOVE", "BADPIXELS"]), \
    exclusive = True)
slicedCubes = specFlagOutliers(slicedCubes, waveGrid, nSigma=5, nIter=1)
slicedCubes = activateMasks(slicedCubes, StringId(["GLITCH","UNCLEANCHOP",\
    "NOISYPIXELS","RAWSATURATION","SATURATION","GRATMOVE", "OUTLIERS", \
    "BADPIXELS"]), exclusive = True)
slicedRebinnedCubes = specWaveRebin(slicedCubes, waveGrid)

# 2. Mask the spectral lines
widthDetect = 2.5 # default value
threshold = 10. # default value
widthMask = 2.5 # default value
lineList=[]
slicedCubesMask = slicedMaskLines(slicedCubes,slicedRebinnedCubes, \
    lineList=[],widthDetect=widthDetect, widthMask=widthMask, threshold=threshold,\
    copy=1, verbose=verbose, maskType="INLINE", calTree=calTree)

# 3. Actual spectral flatfielding
slopeInContinuum = 1
slicedCubes = specFlatFieldLine(slicedCubesMask, scaling=1, copy=1, \
```

```

maxrange=[55.,230.], slopeInContinuum=slopeInContinuum, maxScaling=2., \
maskType="OUTLIERS_FF", offset=0, calTree=calTree, verbose=verbose) # see text for
maxRange advice

# 4. Rename mask OUTLIERS to OUTLIERS_B4FF (specFlagOutliers will refuse
# to overwrite OUTLIERS) & deactivate mask INLINE
slicedCubes.renameMask("OUTLIERS", "OUTLIERS_B4FF")
slicedCubes = deactivateMasks(slicedCubes, StringId(["INLINE", "OUTLIERS_B4FF"]))
if verbose: maskSummary(slicedCubes, slice=0)

# 5. Remove intermediate results
del waveGrid, slicedRebinnedCubes, slicedCubesMask

```

- Masks are activated. As discussed in [Section 4.2.3](#) you may want to not include the RAWSATURATION mask in this call.
- **(1) Flag outliers and rebin.** This: (A) masks outliers so they are not included when the flatfielding task calculates its correction, and then (B) automatically identifies spectral lines so they are not included when the flatfielding is computed. For these it is necessary to run a few tasks that you will also later encounter in the pipeline (and they will be more fully explained at this later point).

(A) **wavelengthGrid** creates a wavelength grid that is common to all spaxels. **specFlagOutliers** runs with that wavelength grid, identifying outliers within the new wavelength grid bins, creating a mask called OUTLIERS.

(B) **specWaveRebin** spectrally re-grids the *PacsCubes* with the wavelength grid, to make *PacsRebinnedCubes*, and where the task **activateMasks** ensures that the identified bad data are not included. As discussed in [Section 4.2.3](#) you may want to not include the RAWSATURATION mask in this call (although at this point it does not matter). The rebinned cubes created by this task are not used for anything except line identification by the subsequent task **slicedMaskLines**; they are deleted at the end of the flatfielding process. You can also create a list of spectral lines for **slicedMaskLines** to use, mainly useful if the automatic identification does not find all the lines, or the absorption lines, in the spectra. In this case you do not need to do the first step (#1 in the script snippet).

- **(2) Mask the spectral lines** with **slicedMaskLines**, adding them to the new mask INLINE. See its [URM](#) entry (which is called "maskLines") for a full parameter list, but briefly:
 - You can either specify a line list (a list of central wavelengths) or opt for an automatic detection of a spectral lines. *If you have multiple, blended or absorption lines you want to flag out for the continuum fitting part of the flatfielding, you should specify their wavelengths in a line list.*
 - If you specify a line list then do not include "slicedRebinnedCubes" in the call to the task, instead fill in the parameter `lineList`. (If you specify a line list *and* ask for automatic identification, the line list will take precedent.) The line list is specified with the parameter `lineList` and is a *PyList* of wavelengths: e.g. `linelist=[52.4, 78.9, 124.4]`
 - The automatic line detection is done on the *PacsRebinnedCubes* created previously. The task looks in the central spaxel and identifies emission lines as local maxima in the rebinned cube ($\text{flux} > \text{calculated_local_rms} * \text{threshold}$). This wavelength region is then excluded, for all pixels, in the subsequent continuum work. `widthDetect` sets the width of the box within which the "local maxima" are checked for: the width is a multiple `widthDetect` of the FWHM (where the FWHM is taken from a calibration file). *If you have blended or wide lines but still use the auto-identification, you may want to increase the width factors.*
 - Note: **slicedMaskLines** will not work if the parameter `widthDetect` is set too small.

The line centres that are found, or given, are then extended such that the masked-out region width is defined by a given multiple `widthMask`, of the FWHM around the line-peak.

Set `verbose=1` for plots showing the position of the lines automatically identified. A dot will appear above the lines (the plot taken from the central spaxel, for each slice) and the line wavelength is printed to the *Console*.

- (3) **specFlatFieldLine** does the flatfielding.

The working of this task and its parameters is explained in [Section 7.4](#). It first creates a reference spectrum from the data of the entire spaxel, for each spaxel, and then compares the spectrum of different populations to the reference, for each spaxel. Offsets are removed by correcting the population-spectra by the ratio to the reference spectrum. The mean spectrum *then* created will have a better SNR than the mean from before.

If `verbose=1`, this task will produce a plot showing, for the central spaxel of each slice, the before and after spectrum taken from the *PacsCubes* (plotted as dots) and a rebinned version of each (plotted as a line and much easier to see).

The `slopeInContinuum=1` is for spectra with lines on a continuum with some slope, it effectively "weights" the ratios taken by the task to account for the slope. While this should allow for a better result for each pixel, note that if the slope is slightly different from pixel to pixel then the flatfielding can become slightly pixel-dependent.

`maxScaling` sets a limit to the scaling factors that the task calculates: limiting the scaling factor protects against abnormally large scaling factors that can result when working with spectra with flux levels close to 0. The mask called `OUTLIERS_FF` marks the datapoints that are outliers before the flatfielding is applied: it is an information mask and is not activated later in the pipeline.

The parameter `maxRange` is used to mask out spectral ranges that fall in the light-leak regions. If using a "Telescope normalisation" pipeline script it is necessary to exclude the region 55 to 190 microns, because outside of these regions the flux calibration is, and always will be, invalid. If using a "Calibration source" pipeline script, the spectral range redwards of 190 microns is, in SPG/HIPE 14.2 and 15, correctly calibrated (when using the latest calibration tree, with version 5 of file `RsrFR1`), however you have to be careful of your analysis: this is explained in [Section 5.4](#). In the pipeline scripts the `maxRange` blue limit is 50 microns, but in fact the value of 55 microns should be used: due to a combination of light leak and band edge, the flux calibration is very uncertain between 50 and 55 microns. For more information on the band edges and light leak, see herschel.esa.int/twiki/bin/view/Public/PacsCalibrationWeb?template=viewprint).

The parameter `maxRange` can also be used to focus the flatfielding effort on a particular range (e.g. a single line) in your data. The flatfielding will be done only in that range: no out of range data is cut out, so be aware that the "out" regions will probably look odd.

- (4) and (5) are cleaning up: renaming the masks and removing redundant necessary products, and deactivating the masks that are not wanted later in the pipeline.

You could save `slicedCubes` before running the flatfielding task, to compare the before and after (using the examples in [Section 10.4.2](#)).



Tip

It can be helpful to plot the data before and then after the flatfielding, to see the effect that it has had on your data, using the task `plotPixel`. See [Section 10.2](#) to learn more about this task.

You can also open any one of the slices in `slicedCubes` in the **Spectrum Explorer**, which is a tool for inspecting spectra from any number of Spectrum products in HIPE. You can inspect masked data and different grating scans in the SE. For a general description of the SE, see *DAG* [chap. 6](#), but to know how to use in on the *PacsCubes*, see [Section 10.5](#).

5.2.7. Spectral flatfielding: all long range scan and SED pipeline scripts

Spectral lines existing on top of medium or high-flux continua should benefit from refining the spectral flat fielding, and any other type of spectra should also be at least slightly improved. *It is recommended to compare spectra obtained with and without spectral flat fielding, and to also check on the results*

of the flatfielding as you do it. This particularly so for spectra with weak continua. The range scan flatfielding tasks should be used on all ranges longer than about 5 microns. For shorter ranges, the data should be reduced with the line scan pipelines.

```
# >>>>> 5 Spectral Flat Fielding for range scans

useSplinesModel = True
excludeLeaks = True
slicedFrames = specFlatFieldRange(slicedFrames, polyOrder=3, verbose=verbose, \
    excludeLeaks=excludeLeaks, selectedRange=None, useSplinesModel=useSplinesMode)

slicedCubes = specFrames2PacsCube(slicedFrames)
```

- **specFlatFieldRange.** The flat fielding task that normalises the response of all pixels within each module to the same reference level.

The working of this task is explained in [Section 7.4](#). First a reference spectrum—the mean of the entire spaxel (excluding bad data)—is created. The spectra from each spaxel are then split into populations and the spectra from each population are compared to the reference spectrum, so computing a scaling. Once the scaling is applied, this brings spectra with excessively high values down and those with excessively low values up, while maintaining the mean. The mean spectrum *then* created will have a better SNR than the mean from before.

SpecFlatFieldRange fits either a polynomial function over the available wavelength range as it computes the scaling factors, or—new to HIPE 14—a spline. The spline is recommended for longer ranges and where the spectra are "bendy". For straight and shorter spectra, the splines and polynomial give similar results. The shorter the wavelength range, the lower you should set the order of the polynomial, otherwise you can introduce artificial S-curves in the final spectra. The spline is the recommended function to use in HIPE 14.

SpecFlatFieldRange will work on any range, but, especially in case of SEDs, it will deliver even better results if you exclude the leak regions from your data beforehand (`excludeLeaks=True`). This is particularly true if using the polynomial function, which cannot handle large and fast variations of the continuum flux. The spline function works better, but it still sometimes struggles in the red light-leak region.

Before SPG/HIPE 14.2 (and 15) it was also the case that the red light leak spectral regions (redwards of 190 microns) were badly calibrated, but the RSRF has been updated (version 5 of the calibration file `Rsrfr1`) and this is no longer the case. The "Telescope normalisation" pipeline does not use the RSRF, and the leak regions will *always* be incorrectly calibrated. To recover spectra redwards of 190 microns, it is necessary to use the "Calibration source" pipeline script, but you still need to be careful about your analysis. More advice is given in [Section 5.4](#). See the Observer's Manual to learn about these leak regions (you can find it here: herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb?template=viewprint).

The parameter `selectedRange` can be used to specifically select a range of the data to flat-field, useful if you are interested only in a certain spectral region. The format of the parameter is `[waveMin, waveMax]`.

To change the box size for the sigma-clipping, set `maxbinsize` to a value in microns (2 is the default). To see plots of the fits set `doPlot=1`. These plots will allow you to assess the quality of the fitting.

- **specFrames2PacsCube:** turn the individual *Frames* in the *slicedFrames* into *PacsCubes* held in a *SlicedPacsCubes* product. This really is only a rearrangement of the data. These cubes have a spatial arrangement of 5x5 spaxels (created from the 25 modules), and along the wavelength dimension are the spectra from the 16 pixels all packed together one after the other. The spectra from these 16 spaxels may themselves be multiples, if the AOR asked for repeats on the wavelength range. At a minimum each pixel holds a spectrum from a grating scan up and one from a grating scan down (i.e. one spectral range sampled twice).

You could save `slicedFrames` before running the flatfielding task, to compare the before and after (using the examples in [Section 10.4.2](#) or the example in the pipeline script).



Tip

It can be helpful to plot the data before and then after the flatfielding, to see the effect that it has had on your data, using the task `plotPixel`. See [Section 10.2](#) to learn more about this task (note: it does also work on `slicedFrames`).

You can also open any one of the slices in `slicedCubes` in the **Spectrum Explorer**, which is a tool for inspecting spectra from any number of Spectrum products in HIPE. You can inspect masked data and different grating scans in the SE. For a general description of the SE, see *DAG* [chap. 6](#), but to know how to use in on the `Frames` of PACS Level 0, 0.5 and 1, see [Section 10.5](#).

5.2.8. Wavelength grid and outlier flagging

The spectra in each spaxel of each *PacsCube* are the spectra of the 16 pixels of each module of the input *Frames*, where each pixel had two or more spectra that covered very nearly, but not exactly, the same wavelength range and wavelength sampling. This throng of wavelength datasets need to be regularised, one wavelength grid for all spaxels of the cube. It is on this new grid that the spectra will be later spectrally resampled to create a single spectrum per spaxel per cube. The bins of the new grid are large enough to include several input data-points (and hence to improve the SNR in the result), but small enough to still allow for at least a Nyquist sampling of the spectral resolution at every wavelength.

```
# >>>>> 6 Wavelength grid and outlier flagging

if ((not locals().has_key('multiObs')) or (not multiObs)):
    oversample = 2
    upsample   = 4
waveGrid=wavelengthGrid(slicedCubes, oversample=oversample, upsample=upsample, \
    calTree = calTree)

# in line scan pipeline and also if you used the line scan flatfielding for
# range scans
slicedCubes = activateMasks(slicedCubes, \
    StringId([str(i) for i in slicedCubes.get(0).maskTypes if i not in \
    ["INLINE", "OUTLIERS_B4FF"]]), exclusive = True)
# in range scan pipeline
slicedCubes = activateMasks(slicedCubes, slicedCubes.get(0).maskTypes, \
    exclusive = True)

slicedCubes = specFlagOutliers(slicedCubes, waveGrid, nSigma=5, nIter=1)
```

If you want to test out several grids, it is useful to save "slicedCubes" to disk first, to always have a clean copy to work with (using `saveSlicedCopy` and `readSliced` to save and reload). See [Section 10.4.2](#) for some examples using `PlotXY` to plot before- and after-task spectra. Note that the best product to check the effect of the `wavelengthGrid` is that produced by the subsequent task `specWaveRebin` (next section).

- **wavelengthGrid**: creates the wavelength grids. This task creates a mask called `OUTOFBAND`. This identifies spectral regions that have fallen out of the range of the PACS filters, something that can happen if you request a range in one camera that results in an "illegal" range in the other camera. This is provided to avoid that the user works on data that are invalid.

The `upsample` and `oversample` parameters are explained in the [PACS URM](#) entry for `wavelengthGrid`, and the effect of different choices of parameters is explained further in [Section 7.2](#).

- **specFlagOutliers**: flag for outliers, i.e. a second level glitch-detection task, creating a mask called `OUTLIERS`. It will ignore all data that are masked as bad in all active masks. This task works by first rebinning the data according to the specified wavelength grid, and then looking for all outliers by searching the datapoints that fall in the new grid, going bin by bin. It only uses the `waveGrid` to

find the outliers, it does not also change the wavelength grid of the input cubes. `nIter` controls the number of iterations you do (1 means two runs of detection are done), and `nSigma` controls the sigma level you clip at. The parameter `nSigma` has the largest effect of the two on the performance of this task.

To look at the OUTLIERS in the *PacsCubes* use the task `plotCubes` ([Section 10.2](#)), the examples given in [Section 10.4.4](#), or use the Spectrum Explorer ([Section 10.5](#)).

Note that the previous deglitching task, `specFlagGlitchFramesQTest`, works on the time-line, whereas this task works with the wavelengths. Neither task tries to fix the glitches, they only mask them.

As discussed in [Section 4.2.3](#) you may want to not activate the RAWSATURATION mask in the task `activateMasks` here. To do that, include this mask together with `"["INLINE", "OUTLIER-S_B4FF"]"` from the call in the line scan pipeline, and copy over that expression to the call in the range scan pipeline.

5.2.9. Spectral rebinning

```
# >>>>> 7 Spectrally rebinning the cubes

# in line scan pipelines
slicedCubes = activateMasks(slicedCubes, \
    StringId([str(i) for i in slicedCubes.get(0).maskTypes if i not in \
    ["INLINE", "OUTLIERS_B4FF"]]), exclusive = True)
# in range scan pipelines
slicedCubes = activateMasks(slicedCubes, slicedCubes.get(0).maskTypes, \
    exclusive = True)

slicedRebinnedCubes = specWaveRebin(slicedCubes, waveGrid)

if verbose:
    slicedSummary(slicedRebinnedCubes)
    overlay = None
    p9 = plotCubesRaDec(slicedRebinnedCubes, overlay = overlay)
```

Here you do the spectral rebinning that combines the spectra held in each spaxel of the *PacsCubes* into one spectrum (per spaxel), improving the SNR and regularising the wavelength grid. This process will only include data *not* flagged as bad in the masked activated before the task is run. *Hence, if you have saturated data* then the saturated data-points may *not* be found in the resulting cubes: the saturated regions may become blank. More specifically, any wavelength bin that is masked as saturated in the input cube will have, in the output cube, either have a NaN value if *all* the datapoints that fed that bin are saturated, or will have an actual value if only *some* of the datapoints that fed that bin are saturated.

Some examples of checking on the masked data are given in [Section 10.4.4](#). See [Section 7.6](#) for more information about the effect of the saturation and glitch masks on the rebinned of the *PacsCubes* by `specWaveRebin`.



Note

The wavelength grid increases with resolution, i.e. with wavelength: the wavelength range of the PACS spectrograph is so long that to achieve at least Nyquist sampling at all wavelengths, it is necessary that the bin sizes scale with wavelength. This wavelength grid is not held in a World Coordinate System (WCS, specifically axis 3), but rather in an "ImageIndex" dataset of the cubes. For more information, see the PPE chps [3](#) in *PACS Products Explained* and [5](#) in *PACS Products Explained*.

- **specWaveRebin:** takes the input wavelength grid and rebins the spectra (via an averaging) from the input cubes and places them in the output cubes. It does this separately for each cube held in the input "slicedCubes", so the differences between the cubes—raster pointing, wavelength range, nod (A and B), nod cycle—are maintained. What *are* combined, per spaxel, are the spectra from the 16 pixels for all the repetitions on the grating scan—and what *are not* combined are the repetitions on nod cycle, as different nods are held as separate slices. The output is a set of *PacsRebinnedCubes* held in a *SlicedPacsRebinnedCube*.

It is important to activate all masks containing bad data you do not want included in your final cubes: including the not flatfielded data, the bad chops and grating movement data, glitches, outliers, saturation, noisy and bad pixels, out of band data. The pipeline activates all masks by default (in activate masks run before this task). As discussed in [Section 4.2.3](#) you may want to not activate the RAWSATURATION mask in the task activateMasks here. To do that, include this mask together with ["INLINE", "OUTLIERS_B4FF"] from the call in the line scan pipeline, and copy over that expression to the call in the range scan pipeline.



Note

For data obtained in Nyquist sampling mode before OD 305, it is normal to find empty bins (NaNs in the rebinned cube) even when rebinned with oversample=2. This is intrinsic to these data, and there is no way to correct for this.

Noise/Errors: This task creates a standard deviation dataset ("stddev"), which is explained in [Section 7.7](#), and you can also check the [URM](#) entry for this task to learn more about this. You will be given a chance to plot the error curve in the next code block, and this plot is also explained in [Section 7.7](#).

- **plotCubesRaDec** Plots the Ra and Dec coverage of the cube(s), i.e. the sky footprint of the observation, with the option of over-plotting this footprint on an image. See [Section 10.2](#) to learn more about this task.

5.2.10. Combine the nod

```
# >>>>> 8 Combine nod A and B

slicedFinalCubes = specAddNodCubes(slicedRebinnedCubes)

# if working with the Calibration source and RSRF script, continue, otherwise
# see the next section
if verbose:
    x,y = 2,2
    pfinal = plotCubes(slicedFinalCubes, x=x, y=y,stroke=1,title="plotCubes - \
        "+str(obsid)+" "+camera, \
        subtitle="Final Rebinning Spectrum. Spaxel ["+str(x)+","+str(y)+\
        "].\n No point source correction applied")
    pstd = plotCubesStddev(slicedFinalCubes, plotLowExp=1, plotDev=0, nsigma=3,
        islineScan=-1, \
        spaxelX=x, spaxelY=y, verbose=verbose, calTree=calTree, wranges=None)
    pstd.titleText, pstd.subtitleText="plotCubesStddev - "+str(obsid)+" "+camera, \
        "Final Rebinning Spectrum & uncertainties. Spaxel ["+str(x)+",\
        "+str(y)+"].\n No point source correction applied"
    slice = 0
    p55 = plotCube5x5(slicedFinalCubes.get(slice), frameTitle="plotCube5x5 - "\
        +str(obsid)+" "+camera+" slice "+str(slice))

if saveOutput:
    name = nameBasis+"_slicedCubes"
    saveSlicedCopy(slicedCubes, name, poolLocation=outputDir)
    name = nameBasis+"_slicedRebinnedCubes"
    saveSlicedCopy(slicedRebinnedCubes, name, poolLocation=outputDir)
    name = nameBasis+"_slicedFinalCubes"
    saveSlicedCopy(slicedFinalCubes, name, poolLocation=outputDir)
```

- The *PacsRebinnedCubes* are now combined on nod. All the nod A and B are average-combined, by **specAddNodCubes**, so slicedSummary run on slicedRebinnedCube and slicedFinalCubes will show different numbers of slices. Note that differences in raster pointing and wavelength are honoured by this task.

The stddev arrays are propagated by this task. This can then be taken to be the errors for the rebinned cubes. For more information, see [Section 7.7](#).

- **plotCubes**, **plotCube5x5**, and **plotCubesStddev**. See [Section 10.2](#) for details. These plot the spectra in the input cubes in different ways: of a single spaxel, of the 5x5 spaxels in the rebinned cube, or of a single spaxel with the standard deviation and continuum RMS datasets overplotted. For the task **plotCubesStddev**, the parameter `islineScan` determines whether the task is used in a way appropriate for line scans (1), range scans (0), or either (the task determines this itself by looking at the Meta data to find out what whether the observation is line or range).
- Spectrum Explorer: you can open the `slicedCubes` and the `slicedRebinnedCubes` in the SE, which is a tool for inspecting single or cube spectra. For a general description of the SE, see *DAG* [chap. 6](#).
- If you chose to save output to disk, these sliced cubes of Level 2 are saved with **saveSlicedCopy**, to a pool on disk in the directory called "outputDir" and with a pool name (sub-directory name) "name". To recover the sliced cubes, use the same `poolLocation` and `poolName` in the task "readSliced". See their [PACS URM](#) entry to learn more about saving and loading.

5.2.11. Flux calibration for "Telescope normalisation" scripts only

The final flux calibration for the telescope normalisation pipeline scripts, taking the spectra from "telescopes" to Jy, is done after the nods have been combined. Then the same plotting as in the section above can be done.

```
# For the Telescope normalisation pipeline script only
# (but not that for point sources)
slicedFinalCubes, background = specRespCalToTelescope(slicedFinalCubes, \
  obs.auxiliary.hk, calTree = calTree)

if verbose:
  x,y = 2,2
  pfinal = plotCubes(slicedFinalCubes, x=x, y=y,stroke=1,title="plotCubes - \
    "+str(obsid)+" "+camera, \
    subtitle="Final Rebinned Spectrum. Spaxel ["+str(x)+","+str(y)+\
    "].\n No point source correction applied")
  pstd = plotCubesStddev(slicedFinalCubes, plotLowExp=1, plotDev=0, nsigma=3,
    islineScan=-1, \
    spaxelX=x, spaxelY=y, verbose=verbose, calTree=calTree, wranges=None)
  pstd.titleText, pstd.subtitleText="plotCubesStddev - "+str(obsid)+" "+camera, \
    "Final Rebinned Spectrum & uncertainties. Spaxel ["+str(x)+",\
    "+str(y)+"].\n No point source correction applied"
  slice = 0
  p55 = plotCube5x5(slicedFinalCubes.get(slice), frameTitle="plotCube5x5 - "\
    +str(obsid)+" "+camera+" slice "+str(slice))

if saveOutput:
  name = nameBasis+"_slicedCubes"
  saveSlicedCopy(slicedCubes, name, poolLocation=outputDir)
  name = nameBasis+"_slicedRebinnedCubes"
  saveSlicedCopy(slicedRebinnedCubes, name, poolLocation=outputDir)
  name = nameBasis+"_slicedFinalCubes"
  saveSlicedCopy(slicedFinalCubes, name, poolLocation=outputDir)
```

- **specRespCalToTelescope** is the task that computes the telescope background spectrum in the background normalisation pipeline script. Its details are explained in its [URM](#) entry, but briefly: it takes the telescope background that has been created from numerous calibration observations, scales it by the temperature during the observation (taken from the housekeeping from the *ObservationContext*), and then uses that to performs the "flux" unit conversion for the `slicedFinalCubes`, from "telescope backgrounds" to Jy.

The stddev dataset is propagated by this task.

- **plotCubes**, **plotCube5x5**, and **plotCubesStddev**. See [Section 10.2](#) for details. These plot the spectra in the input cubes in different ways: of a single spaxel, of the 5x5 spaxels in the rebinned cube, or of a single spaxel with the standard deviation and continuum RMS datasets overplotted. For the

task `plotCubesStddev`, the parameter `isLineScan` determines whether the task is used in a way appropriate for line scans (1), range scans (0), or either (the task determines this itself by looking at the Meta data to find out what whether the observation is line or range).

- Spectrum Explorer: you can open the `slicedCubes` and the `slicedRebinnedCubes` in the SE, which is a tool for inspecting single or cube spectra. For a general description of the SE, see *DAG* [chap. 6](#).
- If you chose to save output to disk, these sliced cubes of Level 2 are saved with `saveSlicedCopy`, to a pool on disk in the directory called "outputDir" and with a pool name (sub-directory name) "name". To recover the sliced cubes, use the same `poolLocation` and `poolName` in the task "readSliced". See their [PACS URM](#) entry to learn more about saving and loading.

5.2.12. Post-processing

The steps to perform now are very similar for all pipeline scripts and are explained in [Section 4.3](#).

The science end-product of the pipeline scripts depends on the observation. For point sources, and in particular for the output of the Telescope normalisation point source script, the end result is an extracted, point source flux-loss corrected, spectrum. This means you *must* run one of the tasks provided to extract and calibration the point source spectrum, as is explained in [Section 4.3](#). For mapping observations you can mosaic together the cubes in the raster to create a single cube, as is also explained in [Section 4.3](#). For single pointing observations of extended sources, the rebinned cubes are those the science measurements should be made on, but see [Section 4.3](#) for some extras.

5.3. "Telescope normalisation drizzle maps" pipeline

New to HIPE 14 and higher: because of an oversight that lead to incorrecion calibration of drizzle cubes created by HIPE version 13 (see herschel.esac.esa.int/twiki/bin/view/Public/DpKnownIssues and herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb), **when using the "Telescope normalisation" pipeline script it is now necessary to calibrate the data when creating drizzle cubes slightly differently to when calibrating any other type of cube.** The drizzle cubes require a calibration applied to the Level 1 *PacsCubes*, for which one particular calibration file has to be used. All other cubes (rebinned, projected, and interpolated) require a calibration applied to the final *PacsRebinnedCubes* of Level 2, for which a slightly different calibration file is necessary. The two calibrations differ only very slightly: the noise level is very slightly higher for the "drizzle calibraiton", but the overall flux levels, line fluxes, and continuum slopes, are the same. However, this means that *line scan chopNod* observations reduced with the "Telescope normalisation" pipeline script, and for which mosaic drizzle cubes are desired (i.e. *Nyquist and oversampled mapping AOTs*), need to be reduced with their own pipeline script.

However, the actual running of the drizzle tasks has not changed at all, and the use of the drizzle tasks themselves are still explained (for all pipeline scripts) in [Section](#), and this is not repeated here. The running of the rest of the pipeline, from Level 0 to the end, is, with the exception of one task, the same as in the "Telescope normalisation" pipeline script for line scans. These parts of the pipeline are therefore not repeated here: the previous sections explains this.

The one different task is `specRespCalToTelescope`:

```
slicedFrames, background = specRespCalToTelescope(slicedFrames, obs.auxiliary.hk, \
  calTree = calTree, reduceNoise = 1)
```

which in the "Telescope normalisation" pipeline is done at the end of Level 2 (after `specAddNodCubes`), *but in this "drizzle pipeline" it is done after the task `specDiffChop`, just before "slicedPacsCubes" is created from "slicedFrames" with the `specFrames2PacsCube` task.*

At the end of the pipeline, you are shown how to create projected and interpolated cubes with the same WCS as the drizzled cubes have, which allow for a direct comparison of the different cubes.

(Although we note that creating interpolated cubes with the very small spaxel sizes of drizzled cubes is not recommended.)

5.4. Pipeline steps for spectral lines in the light leak regions (mainly longer than 190 μm)

An update to the calibration tree for SPG/HIPE 14.2 and 15 means that the RSRF in the red region (band R1: calibration file RsrFR1) now calibrates the spectral lines reasonably well. The continuum will be wrong, but the spectral line fluxes can be recovered. The pipeline scripts "Calibration source and RSRF" (line and range scans) must be used to work in this spectral range, because the "Telescope normalisation" scripts do not use the RSRF and will *always* be incorrectly calibrated in the light leak regions. This being the case, the SPG data (that which you obtain from the HSA) will be incorrectly calibrated below 55 microns and above 190 microns: hence these ranges are masked out and will not be available in the *ObservationContexts*.

To recover the red-leak spectral region (190 microns and upwards), you can reduce the data in HIPE yourself using one of the "Calibration source" scripts. The only change with respect to the script in HIPE and as described in this chapter is at the flatfielding stage:

- Range scans: set `excludeLeaks=False`. In addition, and especially if dealing with SEDs (long ranges), it will improve results if you limit the region included in the flatfielding to around your line. The parameter `selectedRange` can be used. Alternatively you can use the line scan flatfielding task and set the parameter `maxRange` to include your line only. See [Section 5.2.7](#) to learn about using the line scan flatfielding task while running the range scan pipeline script.
- Line and short range scans: for line scans it is probably not necessary to make any changes as the spectral range is so short anyway, but for ranges of a few microns (and especially if this includes some spectral slope) you can use the parameter `maxRange` to limit the flatfielding to around your red line(s).

A consequence of limiting the range that is flatfielded is that the not-flatfielded parts of the spectrum could look very strange. At the end of the pipeline these regions are anyway cut out, since they have the mask NOTFFED and hence are excluded when the Level 2 rebinned cubes are created. But between this part of the pipeline and the end, inspecting the cubes may be difficult due to strong curvature induced by the partial flatfielding. To make subsequent inspection of such cubes easier, you could use the following task to cut the unwanted regions out:

```
newSlicedCubes = pacsExtractSpectralRange(slicedCubes,waveRanges=[[200,205]])
```

This task creates a "newSlicedCubes" including only the data within the wavelength range specified. The parameter `waveRanges` can take a list of more than one `[min, max]`—`[[200,203],[203,204]]` for example—but if you want to select only one then a double `[[]]` is still necessary. The task will work for *SlicedPacsCube* and *SlicedPacsRebinnedCube* or any one *PacsCube* or *PacsRebinnedCube*.

It is important to note that since the continuum level for the red-leak calibrated spectral regions will always be wrong, when using the task `extractCentralSpectrum` on **point sources**, only the result "c1" or "c9" can be used, not also "c129" (since this latter spectrum computes a correction that is based on the continuum level).

See the Observer's Manual and the PACS spectrometer calibration document to learn more (you can obtain these from the HSC PACS web-page: herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb?template=viewprint).

5.5. The Split On-Off 'testing' script

Level 0.5 to 2 of the chopNod Split On-Off script is not described in a much detail as the previous scripts: the steps are exactly the same.

The aim of this helper script is to produce a set of on-source cubes and off-source cubes, which can then be easily compared to each other (for example with the Spectrum Explorer) to see if there is obvious contamination in the off-source pointings. This is not a pipeline script: you cannot do science on the end-results. In fact, after setting the obsid and camera, you could run the script blindly, since playing with the parameters will produce little difference to your comparison of the on-cube and off-cubes.



Tip

An easy way to compare the on- and off-spectra for the spaxels of your cubes is to load the individual on- and off-cubes into the Spectrum Explorer and "link" them (*DAG* [chap. 6.5](#); and use the "linking" capability), to see together the on- and off-source spectra for any spaxel.

Note that the Spectrum Explorer works with cubes, not the `meanRebinnedCubesOn|Off` (which is a *ListContext*) that is the end product of the Split On-Off pipeline. You need to work on the individual cubes in the *ListContext*. This can be done by clicking on "meanRebinnedCubesOn (and Off)" in the *Variables* pane, and then from the *Outline* pane you will see a cube listing: from there you can double-click to open the first cube with the Spectrum Explorer, and then drag-and-drop the remaining cubes into the GUI.

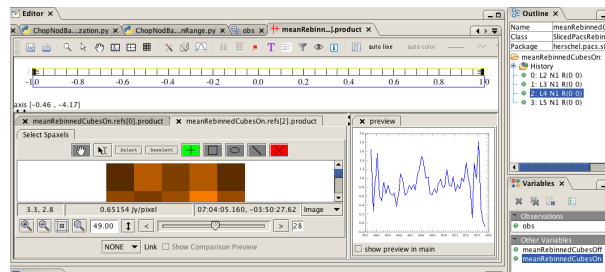


Figure 5.1. Loading cubes from a *ListContext* into the Spectrum Explorer

The only tasks we do describe here are those unique to this pipeline.

```
slicedFrames = specSubtractDark(slicedFrames, calTree=calTree)
slicedFrames = pacsSliceContext(slicedFrames, splitOnOff=1)[0]

lineId      = []
wavelength  = []
rasterLine  = []
rasterCol   = []
nodPosition = ""
nodCycle    = []
scical      = ""
band        = ""
sliceNumber = []

onOff       = "ON"
sCubesOn   = selectSlices(slicedCubes, lineId=lineId, wavelength=wavelength, \
    rasterLine=rasterLine, rasterCol=rasterCol, nodPosition=nodPosition, \
    nodCycle=nodCycle, scical=scical, \
    band=band, sliceNumber=sliceNumber, onOff=onOff, verbose=verbose)
onOff      = "OFF"
sCubesOff  = selectSlices(slicedCubes, lineId=lineId, wavelength=wavelength, \
    rasterLine=rasterLine, rasterCol=rasterCol, nodPosition=nodPosition, \
    nodCycle=nodCycle, scical=scical, \
    band=band, sliceNumber=sliceNumber, onOff=onOff, verbose=verbose)
```

- **specSubtractDark**: subtracts the dark signal.
- **pacsSliceContext**: is used to split the on-source from the off-source data.
- **selectSlices**: is then used to create an on-source and an off-source product.

5.6. Wrapper script 'Combine observations for a full SED'

5.6.1. Explanation

This helper script is for the long range and SED AOTs. It will run any chosen single-observation pipeline script on your obsids, and then extract the point source spectra and combine them. The intention is that this script is used on observations of SEDs, where you want to be able to see the entire stretch of the spectrum within one product. Note that we do not *merge* the extracted point source spectra, they are simply stored in a single product.

This script could also be used as the basis for your own bulk-processing script, for the more advanced users of the PACS pipeline. With a few tweaks it would allow you to reduce any number of obsids and to save the results on disk.

It is necessary for you to (i) have reduced at least some data through the single-observation pipeline script that you want to use on these data, so you know what you are doing and understand the instructions here, and (ii) make some edits to the single-observation pipeline script that you will run. Some parameters are set in this combined-observation script (e.g. directory names) but others (e.g. verbose) have to be set in the pipeline script itself.

The script will reduce each obsid and save to pool the Level 1 *PacsCubes*, the Level 2 *PacsRebinnedCubes* before being combined on nod, and the Level 2 *PacsRebinnedCubes* after being combined on nod. You can choose to save none, any, or all of these by commenting in or out the appropriate lines in the script. The output of the pipeline at all levels is a *ListContext* of the cubes: a *SlicedPacsCubes* containing the Level 1 *PacsCubes*, and a *SlicedPacsRebinnedCubes* contain the Level 2 *PacsRebinnedCubes*. The number of cubes contained in each list depends on how many wavelength ranges/bands, nod repeats, and pointings you have in your observations. The script will then run the point source extraction task on all the cubes and join the resulting spectra into a *SpectrumId*.

5.6.2. Running the script

5.6.2.1. Setup

First decide which pipeline script to run. If you want to change *any* of the parameters in the script you, first copy it to a new name/location.

The parameters that you definitely have to set in your copied pipeline script are:

- *verbose*: to produce plots showing the results of pipeline tasks, or not; *verbose*=1 or 0
- *useHsa*; for the task *getObservation*; get the data directly from the HSA or from pool on disk); *useHsa*=1 or 0
- *poolName and/or poolLocation*: for the task *getObservation/saveObservation*; either where the data are on disk, or to where to save them

Next, set up some parameters:

```
import os, time

# 1
multiObs = 1
saveIndividualObsids = 1

# 2
outputDir = Configuration.getProperty("user.dir")
# either
```



```

HCSS_DIR = Configuration.getProperty("var.hcss.dir")
scriptsDir = HCSS_DIR + "/scripts/pacs/scripts/ipipe/spec/"
script = scriptsDir + "ChopNodrangeScan.py"
# or, if you have your own script to run:
script = "/Users/me/myscript.py"
# or if the script is located in the directory you started HIPE from
script = outputDir + "myscript.py"
# or
scriptDir = "/Users/me/scripts"
script = scriptsDir + "ChopNodrangeScan.py"

# 3
# Chose the obsids
obsids = {}
obsids[1342229701] = "SEDA"
obsids[1342229702] = "SEDB"

# 4
# Rebinning Parameters
oversample = 2
upsample = 2
strovup = "ov"+str(oversample)+"_up"+str(upsample)

# 5
# For saving the results
outputDir = None
prefix=""
if outputDir: prefix = outputDir+"/"
buildNumber = str(Configuration.getProjectInfo().track) + '.' + \
    str(Configuration.getProjectInfo().build)
trackfilename = prefix+"ChopNodSEDMultiObsTracking.txt"
trackfile = open(trackfilename, 'w')
trackfile.write("START \n")
trackfile.close()

# 6
finalCubeList = []
starttime = time.time()

```

These do the following:

- **(1):** In the pipeline scripts that reduced the individual observation ,you will see commands such as

```

if ((not locals().has_key('multiObs')) or (not multiObs)):
    obsid = 1342..... # enter your obsid here

```

Setting **multiObs** to True/1 means that the indicated parameter (obsid in the example above) will be taken from the Combine Observation script rather than in the actual pipeline script itself.

- **(1): saveIndividualObsids** set to True/1 will tell the script to save all the final cubes and extracted global spectrum.
- **(2):** Setting of the directory "HCSS_DIR" and then "scriptsDir", which is where the interactive pipeline scripts can be found: HCSS_DIR is the location of the HIPE software, and scriptsDir is the standard location for the PACS scripts in the HIPE build. "ChopNodBackgroundNormalization-Range.py" is the name of the Telescope normalisation pipeline script; you will see the names of the scripts when you load them into HIPE via the Pipeline menu (the tab name is the file name). If you want to change anything in the pipeline script, you need to save to a new location/name; we have added a few lines of code that show you how to use this one instead of the default. "outputDir" is where you started HIPE from or you can set it to anything else, using the same syntax as used for scriptsDir. The script that you will use to reduce each single observation is set as "script".
- **(3):** Then you define your obsids. Either you specify this in a file on disk and then read it in, or specify here.
- **(4):** Set the values for the wavelength grid upsample and oversample (hence over-ruling whatever is in the script you later run).

- (5): Information to create unique filename in the saved products.
- (6): Then you create the *PyList* that is a list that will contain your pipeline-produced products, and note the starting time (not necessary, just nice to know).

5.6.2.2. Run the loop

Next follows a loop over all the obsids and cameras and the pipeline script is run on each, starting from getting the *ObservationContext* into HIPE.

After the loop has run the *slicedFinalCubes*, in each loop, are pushed into "finalCubeList" and then save that to disk. After the loop, these are then concatenated into a single product, which can be saved to disk as a pool:

```
allFinalCubes = concatenateSliced(finalCubeList)
objectName = "NoObjectName"
try:
    objectName = obs.meta["object"].value.replace(" ", "").\
        replace("+", "plus").replace(".", "_")
    objectName += "_" + str(obs.meta["odNumber"].value)
except:
    pass
name = objectName + "_ChopNodMultiObs_allFinalCubes"
saveSlicedCopy(allFinalCubes , name, poolLocation=outputDir)
```

- **concatenateSliced** will take the individual cubes of *finalCubeList*, i.e. each cube in each *SlicedPacsRebinnedCubes* in the *finalCubeList PyList*, and create a new *ListContext*. So, if you pushed two *SlicedPacsRebinnedCubes* into *finalCubeList*, and each *SlicedPacsRebinnedCubes* had two *PacsRebinnedCubes* in it, then *allFinalCubes* will have four *PacsRebinnedCubes* in it.
- The try-except will try to take the object name out of the header of the *ObservationContext* called *obs* that you have in your HIPE session (the same "obs" that you extracted in your final loop of pipeline processing above), but if it cannot succeed it will do nothing. Then your product is saved to a pool on disk with **saveSlicedCopy**.

Finally, have a look at the spectra in your cubes:

```
if verbose:
    slicedSummary(allFinalCubes)
    x,y = 2,2
    pfinal = plotCubes(allFinalCubes, [], x=x, y=y)
```

Plot the spectra of the chosen spaxel using the pipeline helper task **plotCubes**. See [Section 10.2](#) for more detail.

5.6.2.3. Extract the point-source spectra

In the "Combine observations" script the we include the task that you must run if you are dealing with point sources located in the centre of a pointed observation. See [Chapter 8](#) to learn more about the actual task, and [Section 4.3](#), to learn more about running the task in the pipeline. The extra bits in this script are there to push the output point source spectra into a single *SpectrumId*, so you can view the entire SED in one go, e.g. in the Spectrum Explorer.

```
# Setup
fullSpec = SpectrumId()
segments = IntId()
fullSpec9 = SpectrumId()
segments9 = IntId()
fullSpecCorr3x3 = SpectrumId()
segmentsCorr3x3 = IntId()

target = slicedFinalCubes.meta["object"].value.replace(" ", "_")

# either these two
```

```

smoothing = 'wavelet'
nLowFreq   = 4
# or these three - the ones chosen in the example below
smoothing = 'filter'
gaussianFilterWidth = 50
medianFilterWidth   = 15

for slice in range(len(slicedFinalCubes.refs)):
    central11,central19,central129 = \
        extractCentralSpectrum(slicedFinalCubes.get(slice), \
            smoothing=smoothing, width=gaussianFilterWidth, \
            preFilterWidth=medianFilterWidth, \
            nLowFreq=nLowFreq, calTree=calTree,verbose=verbose)
    if saveIndividualObsids:
        name = "OBSID_"+str(obsid)+"_"+target+"_"+camera+\
            "_centralSpaxel_PointSourceCorrected_Corrected3x3NO_slice_"
        simpleFitsWriter(product=central11,\
            file = name+str(slice).zfill(2)+".fits")
        name = "OBSID_"+str(obsid)+"_"+target+"_"+camera+\
            "_central9Spaxels_PointSourceCorrected_slice_"
        simpleFitsWriter(product=central9,\
            file = name+str(slice).zfill(2)+".fits")
        name = "OBSID_"+str(obsid)+"_"+target+"_"+camera+\
            "_centralSpaxel_PointSourceCorrected_Corrected3x3YES_slice_"
        simpleFitsWriter(product=central129,\
            file = name+str(slice).zfill(2)+".fits")
    if verbose:
        centralSpec = central129.spectrumId
        try:
            openVariable("centralSpec", "Spectrum Explorer")
        except:
            print "Spectrum Explorer only works within HIPE"
        # Create the SpectrumId
        spec = central11.spectrumId
        fullSpec.concatenate(spec)
        segments.append(IntId(spec.flux.length(), slice))
        spec9 = central9.spectrumId
        fullSpec9.concatenate(spec9)
        segments9.append(IntId(spec9.flux.length(), slice))
        specCorr3x3 = central129.spectrumId
        fullSpecCorr3x3.concatenate(specCorr3x3)
        segmentsCorr3x3.append(IntId(specCorr3x3.flux.length(), slice))

```

- The point-source calibrated spectra from all the *PacsRebinnedCube* you loop on are saved into a *SpectrumId* product called "spec", "spec9", "fullSpecCorr3x3". The slice numbering is saved in "segments", a separate value for each slice (i.e. for each cube you run the loop on).

See first [Section 4.3.3.1](#), where a brief text on this task is given. Then move to [Section 8.5](#) to learn more about this task.

```

# Include the column segment in the final SpectrumId & save to fits
fullSpec.setSegment(segments)
fullSpec9.setSegment(segments9)
fullSpecCorr3x3.setSegment(segmentsCorr3x3)
name = objectName + \
    "_ChopNodSEDMultiObs_1d_centralSpaxel_correct3x3_NO_PointSourceCorrected.fits"
simpleFitsWriter(SimpleSpectrum(fullSpec),prefix+name)
name = objectName + \
    "_ChopNodSEDMultiObs_1d_central9Spaxels_PointSourceCorrected.fits"
simpleFitsWriter(SimpleSpectrum(fullSpec9),prefix+name)
name = objectName + \
    "_ChopNodSEDMultiObs_1d_centralSpaxel_correct3x3_YES_PointSourceCorrected.fits"
simpleFitsWriter(SimpleSpectrum(fullSpecCorr3x3),prefix+name)

if verbose:
    # Display in spectrum explorer
    try:
        openVariable("fullSpec", "Spectrum Explorer")
        openVariable("fullSpec9", "Spectrum Explorer")
        openVariable("fullSpecCorr3x3", "Spectrum Explorer")
    except:

```

```
print "Info: openVariable could not open Spectrum Explorer"

trackfile = open(trackfilename, 'a')
trackfile.write("END Total Duration: "+ str("%7.1f\n" % \
    (time.time() - starttime)) + "\n")
trackfile.close()
```

- After you have extracted the spectra and placed them into the *SpectrumId*, you add the segment number array to them (the line "fullSpec.setSegment(segments)"). This allows you to identify which part of the *SpectrumId* comes from which slice: because its segment number will be its slice number. To learn more about *SpectrumId* you should read the [SG](#) chap. 3. The usefulness of saving these spectra to a *SpectrumId* is that you can then open it with the SpectrumExplorer (see the *DAG* [chap. 6](#)) and look at each and all the segments, and you can also save the file to FITS.
- Finally you save the file, open it with the SpectrumExplorer, and write to and close the trackfile again.

Chapter 6. Unchopped and Wavelength switching pipelines

6.1. Introduction

The **unchopped pipeline** menus were introduced in [Chapter 4](#). After having selected the pipeline script you want to run, and carried out the Level 0 to 0.5 parts of the pipeline ([Chapter 4](#)) you can now read this chapter.

The differences between the pipeline scripts is not large, so this chapter goes through each stage of all scripts together. Your prime source of material for the pipeline should be the scripts (from the HIPE build) themselves, since these may change slightly from what is presented here.

The difference between the line scan and range scan unchopped menus is that for the *line scan* mode, the on-source and off-source data are contained in the same observation, and for the *range scan* mode, the on-source and off-source are separate observations which need to be reduced independently and then subtracted. In the line scan and range scan menus there is one script that is the same as the SPG script, and one that includes a new set of transient correction tasks that can be tested.

More information on testing out some of the pipeline tasks is given in [Chapter 7](#) and information on the post-pipeline tasks is given in [Chapter 8](#) and [Chapter 9](#). How to plot and inspect your pipeline products can be found in [Chapter 10](#). In this chapter we also include the wavelength switching pipeline: this is an old mode that was little used, and the script is that for unchopped line scan, so read that section.

6.2. The 0.5 to 2 scripts for all pipelines

The Level 0.5 to Level 2 part of the pipeline is almost the same for the line scan and range scan unchopped scripts. Here we take you through the scripts, indicating where code is for one or another pipeline script only. You should in any case have the pipeline script already open in HIPE before carrying on.

If you want to begin from Level 0.5, i.e. not run that part of the pipeline yourself (especially as this is no longer necessary), the first command is

```
slicedFrames = obs.level0_5.blue.fitted.product # for the blue camera
slicedFrames = obs.level0_5.red.fitted.product # for the red camera
```

and it is still necessary to set up the general pipeline parameters: `verbose`, `saveOutput`, `calTree` and if you do want to `saveOutput`, then also `nameBasis` (see [Chapter 4](#)).

6.2.1. Masking for glitches

```
# >>>>> 1 Masking for glitches; convert the capacitance

slicedFrames = activateMasks(slicedFrames, StringId([" "]), \
    exclusive = True)

# for the original pipeline script
slicedFrames = specFlagGlitchFramesQTest(slicedFrames, copy=1)
# for the new pipeline ("with transients") script
slicedFrames = specFlagGlitchFramesMAD(slicedFrames, copy=1)

slicedFrames = activateMasks(slicedFrames, slicedFrames.get(0).getMaskTypes())
slicedFrames = convertSignal2StandardCap(slicedFrames, calTree=calTree)
```

There is one pipeline task here—flagging for glitches—which is preceded by a task that de/activates the masks,

- **activateMasks**: this task activates (or deactivates) the indicated masks, so that the following task(s) can (or not) take them into account. The parameter `exclusive` is say what to do with the not-named masks. The first call to this task activates *no* masks, i.e. all are deactivated, and the second activates all masks that are present in the `slicedFrames`. Consult the [URM](#) entry of `activateMasks` to learn more about the parameters.
- **specFlagGlitchFramesQTest/MAD**: these two do the same thing but with a different algorithm. They flag the data for glitches (e.g. cosmic rays) using the (i) Q statistical test or (ii) the MAD (median absolute deviation), creating a mask called GLITCH. The MAD algorithm is used in the transients correction pipeline as it is less aggressive, and so less likely to also catch transients, which need to be dealt with in a different way. The deglitching task works on the time-line as the X-axis, and it identifies and flags out the bad data (1=is bad, 0=is not bad), it does not change them. The GLITCH mask is automatically activated when it is created. There are parameters of this task that you could play with (see their [URM](#) entries), but note that these have been much tested and the default parameter settings are good for practically all cases. There is in any case a later outlier detection task run, which will clean up any left-over glitches.

The parameter `copy=1` is necessary to decouple the `slicedFrames` produced by this task from the `slicedFrames` put into the task, as was done previously in the Level 0—0.5 task `specFlagSaturationFrames`.

- **convertSignal2StandardCap**: converts 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.

6.2.2. Compute the dark and the response; subtract the dark

```
# >>>>> 2 Compute the dark and response

calBlock = selectSlices(slicedFrames,scical="cal").get(0)
csResponseAndDark = specDiffCs(calBlock, calTree = calTree)
slicedFrames = specSubtractDark(slicedFrames, calTree=calTree)

# For the original line scan pipeline only
gaps = specSignalGap (slicedFrames)
```

- **selectSlices**: to select the calibration slice out from `slicedFrames`. See `selectSlices` in the [URM](#) to learn more.
- **specDiffCs**: calculates the response and dark current. Briefly, `specDiffCs` uses a combination of standard star observations (stars, asteroids, Neptune and Uranus) contained in the calibration file `ObservedResponse`, and the signal from the calibration block observed during the observation compared to the calibration block absolute fluxes contained in the calibration file `calSourceFlux`. From these data the response of the instrument and the dark current during your observation is calculated, and this is placed in the product `"csResponseAndDark"`. However, we still subtract the dark current calculated from ground-tests than that this result from `specDiffCs`, since it still provides a more reliable result.
- **specSubtractDark**: this task subtract the dark current as a single value held in the calibration file `"spectrometer.darkCurrent"`.
- **specSignalGap**: is used only in the original line scan pipeline. Later in the pipeline the data will be correction for transients. Before this, and also before applying the RSRF, it is necessary to do some housework on the data: identify gaps between consecutive slices in time order. See the [URM](#) entry for this task to learn more.

6.2.3. Flux calibration

```
# >>>>> 3 Flux calibration

slicedFrames = rsrfCal(slicedFrames, calTree=calTree)
slicedFrames = specRespCal(slicedFrames, csResponseAndDark = csResponseAndDark,
    calTree=calTree)

# for the range scan original pipeline script, and for the range
# and line scan new ("with transients") pipeline scripts
slicedFrames = selectSlices(slicedFrames, scical="sci")
```

The pipeline helper tasks `slicedSummaryPlot` and `plotSignalBasic` (here running on the first slice) have been explained before, and they are also explained in [Section 10.2](#).

- **rsrfCal**: apply the relative spectral response function. The RSRF is taken from the calibration tree. See the [URM](#) entry for this task to learn more.

For spectral lines at wavelengths longer than 190 μ m, see [Section 6.4](#). For these cases you need to use a particular version of the RSRF for band R1.

- **specRespCal**: apply the absolute response correction, using the response that was calculated from the calibration block. The flux unit is now Jy.
- **selectSlices**: has been used before. Here you effectively clip out the calibration block data. This is in all pipeline scripts except the original line scan script, where it is run later.

After this task you could again save `slicedFrames`, so you can compare the after product to the before product. You can also follow the examples given in [Section 10.4.2](#) to compare using PlotXY.

```
if saveOutput:
    name=nameBasis+"_slicedFrames_B4TransientCorrection"
    try:
        saveSlicedCopy(slicedFrames,name, poolLocation=outputDir)
    except:
        print "Exception raised: ",sys.exc_info()
        print "You may have to remove: ", outputDir+'/'+name
```

For line scan AOTs, the first slice (after the calblock slice) should be on, and the second off, and so on. The task `slicedSummary` lists the *Frames* in "slicedFrames" in the correct order.

6.2.4. Correct for transients: original line scan pipeline script only

```
# >>>>> 4a Correct for transients

if verbose:
    slicedSummary(slicedFrames)
    slice = 1
    module = 12
    ptrans = plotTransient(slicedFrames, slice=slice, module=module, \
        color=java.awt.Color.black, title="Transient Correction - Slice "\
        +str(slice)+" Module = "+str(module))
    ptrans.getLayer(ptrans.getLayerCount()-1).setName("Before [black]")
    ptrans.getLegend().setVisible(1)

slicedFrames = specLongTermTransient(slicedFrames, gaps=gaps)

if verbose:
    ptrans = plotTransient(slicedFrames, p=ptrans, slice=slice, \
        module=module,color=java.awt.Color.red)
    ptrans.getLayer(ptrans.getLayerCount()-1).setName("After [red]")

slicedFrames = selectSlices(slicedFrames, scical="sci")
```

```

if verbose: slicedSummary(slicedFrames)

if saveOutput:
    name=nameBasis+"_slicedFrames_B4FF"
    saveSlicedCopy(slicedFrames,name, poolLocation=outputDir)

```

For information on what transients are and how they are corrected by the pipeline tasks, see [Section 7.13](#). Here we only explain the use of the tasks.

- **specLongTermTransient**: will remove the effect of the long-term transient that occurs the beginning of an observation for the unchopped line mode. It detects and models these transients, separately for each spaxel/module. This task can take a long time to run.
- **plotTransient**: plots the data (black points) from the first science *Frames* (slice = 1) for the central module of the IFU (module = 12) normalised to the signal level in the final grating scan data-chunk. The various grating scans cover the same wavelength range as each other (and there are at least two grating scans in any observation): since the effect of a transient is to increase the signal levels, a transient will make the first spectral scan stand out from those of later in the observation. The second call to this task plots the corrected data as red points.
- Clip out the calibration slice with **selectSlices**.

6.2.5. Correct for transients: new ("with transients") line scan pipeline script only

In the new line scan pipeline script there is a new set of tasks for detecting and correcting long-term and short-term transients. The end result of this is a signal stream that has been corrected for transients, or masked out where correction was not possible, and the signal levels of the different populations of data have been normalised to the mean, in this way bringing down the scatter in the data and improving the SNR. This latter part is essentially the same as performed in the original line scan pipeline script by the flatfielding task, *which is why in this new script there is no subsequent flatfielding step*.

For information on what transients are and how they are corrected by the pipeline tasks, see [Section 7.13](#). Here we only explain the use of the tasks.

The first part is to plot the data in such a way as to see the long-term transients therein. Set "interactive" to True (default is False) at the beginning of the pipeline to see the plots before applying the correction.

```

# >>>>> 4b Correct for long-term transients

if interactive == True:
    pta = plotLongTermTransientAll(slicedFrames,obs,step=0,module=12,\
        calTree=calTree)
    myAnswer = JOptionPane.showConfirmDialog(None, \
        "Compute the whole observation long-term transient correction ?")
    if myAnswer == 0:
        slicedFrames, fitResult = specLongTermTransientAll(slicedFrames,\
            calTree=calTree,obs=obs)
        pta=plotLongTermTransientAll(slicedFrames,obs,pta=pta,fitResult=fitResult,\
            step=1,module=12,calTree=calTree)
        myAnswer2 = JOptionPane.showConfirmDialog(None, \
            "Apply the whole observation long-term transient correction ?")
        if myAnswer2 == 0:
            slicedFrames=specApplyLongTermTransient(slicedFrames,\
                fitResult=fitResult,obs=obs)
    elif myAnswer == 1:
        print "Correction skipped"
    else:
        print "Correction canceled"
else:
    if verbose == True:
        pta = plotLongTermTransientAll(slicedFrames,obs,step=0,module=12,\
            calTree=calTree)
        slicedFrames, fitResult = specLongTermTransientAll(slicedFrames,\
            calTree=calTree,obs=obs)

```



```

if verbose == True:
    pta=plotLongTermTransientAll(slicedFrames,obs,pta=pta,fitResult=fitResult,\
                               step=1,module=12,calTree=calTree)
    slicedFrames=specApplyLongTermTransient(slicedFrames,fitResult=fitResult,\
                                           obs=obs)

```

- **plotLongTermTransientAll:** This task plots the data taken at the beginning of the observation normalised to the expected background telescope flux. The on- and off-source data are plotted as blue and green points. The second call to `plotLongTermTransientAll` plots the subsequently-detected long-term transient fit as a red line.

You can chose which module (0—24: these correspond to the spaxels: [Section 10.6](#)) to plot.

- **specLongTermTransientAll:** This task is used to detect the long-term transient in an observation.
- **specApplyLongTermTransient:** This task applies the correction computed by the task `specLongTermTransientAll`.

The next step is to correct for transients that occur between slices.

```

pbasic = plotSignalBasic(slicedFrames, slice=0, titleText="Start")
if verbose:
    slicedSummary(slicedFrames)
    slice = 0
    module = 12
    ptrans = plotTransient(slicedFrames, slice=slice,module=module, \
                          color=java.awt.Color.black, title="Transient Correction - Slice "\
                          +str(slice)+" Module = "+str(module))
    ptrans.getLayer(ptrans.getLayerCount()-1).setName("Normalized flux [black]")
    ptrans.getLegend().setVisible(1)

slicedFrames = specLongTermTransCorr(slicedFrames,calTree=calTree,\
                                     verbose=verbose, applyCorrection=1)
if verbose:
    pars = slicedFrames.refs[slice].product['MODELPARS'].data[module,:]
    n = slicedFrames.refs[slice].product.signal[0,0,:].length()
    t = slicedFrames.refs[slice].product.status["FINETIME"].data/1.e6
    t -= t[0]
    x = Double1d(range(n))
    model = pars[0]+pars[1]*EXP(pars[2]*x)+pars[3]*EXP(pars[4]*x)+\
            pars[5]*EXP(pars[6]*x)
    ptrans.addLayer(LayerXY(t,model,color=java.awt.Color.red,stroke=1.5))
    ptrans.getLayer(ptrans.getLayerCount()-1).setName("Model [red]")

pbasic = plotSignalBasic(slicedFrames, slice=0, titleText="Post LTT")

```

- **plotTransient:** plots the data (black points) from the first science *Frames* (`slice = 1`) for the central module of the IFU (`module = 12`) normalised to the signal level in the final grating scan data-chunk. The various grating scans cover the same wavelength range as each other (and there are at least two grating scans in any observation): since the effect of a transient is to increase the signal levels, a transient will make the first spectral scan stand out from those of later in the observation. The second call to this task plots the corrected data as red points.
- **specLongTermTransCorr:** Sudden differences in flux between two adjacent slices cause long-term transients in the detector response. This task fits a transient model to the signal normalised to the last wavelength scan, and applies the correction to the data.
- **plotSignalBasic:** is a pipeline helper plot and is explained in [Section 10.2](#). It produces a basic plot of the signal of the central pixel in the central module of the instrument. To plot a different pixel or module, specify the pixel and module to plot.

The final step is to clean up the short-term transients.

```

slicedFrames = specMedianSpectrum(slicedFrames,calTree=calTree)
slicedFrames = specTransCorr(slicedFrames,calTree=calTree)

```

```

pbasic = plotSignalBasic(slicedFrames, slice=0, titleText="Post TC")

if saveOutput:
    name=nameBasis+"_slicedFrames_B4TransientCorrection"
    try:
        saveSlicedCopy(slicedFrames,name, poolLocation=outputDir)
    except:
        print "Exception raised: ",sys.exc_info()
        print "You may have to remove: ", outputDir+'/'+name
        # To restore the data:
        # slicedFrames = readSliced(name, poolLocation=poolLocation)

```

- **specMedianSpectrum:** This task computes a first guess of the spectrum for each spatial module of the *Frames* ("modules" later become "spaxels") using the median value of the flux (per wavelength) from the 16 pixels that feed each module. The median spectrum is used to normalise the signal by the next task, `specTransCorr`, to compute the transient correction.
- **specTransCorr:** This task identifies the discontinuities in the signal caused by cosmic ray hits, fits the subsequent transients, and then corrects the signal for these. Any signal which is too damaged to be corrected is instead flagged in a new mask: `UNCORRECTED`. To see these data, you can open the *Frames* in `slicedFrames` in the Spectrum Explorer ([Section 10.5](#)).
- **plotSignalBasic:** the same task as used previously (see explanation above).
- **saveSlicedCopy:** an optional save of the products just created.

6.2.6. Correct for transients: new ("with transients") range scan pipeline script only

In the new range scan pipeline script there is a set of tasks for detecting and correcting the long- and short-term transients. The end result of this is a signal stream that has been corrected for transients, or masked out where correction was not possible, and the signal levels of the different populations of data have been normalised to the mean, in this way bringing down the scatter in the data and improving the SNR. This latter part is essentially the same as performed in the original line scan pipeline script by the flatfielding task, *which is why in this new script there is no subsequent flatfielding step*.

The first part is to plot a spectrum, then correct the long-term transients, and then over-plot the corrected spectrum.

```

# >>>>> 4c Correct for transients

pbasic = plotSignalBasic(slicedFrames, slice=0, titleText="Start")
if verbose:
    slicedSummary(slicedFrames)
    slice = 0
    module = 12
    ptrans = plotTransient(slicedFrames, slice=slice, updown=1, module=module,\
        color=java.awt.Color.black, title="Transient Correction - Slice "\
        +str(slice)+" Module = "+str(module))
    ptrans.getLayer(ptrans.getLayerCount()-1).setName("Normalized flux [black]")
    ptrans.getLegend().setVisible(1)

slicedFrames = specLongTermTransCorr(slicedFrames, calTree=calTree,\
    verbose=verbose, applyCorrection=1)

if verbose:
    pars = slicedFrames.refs[slice].product['MODELPARS'].data[module,:]
    n = slicedFrames.refs[slice].product.signal[0,0,:].length()
    t = slicedFrames.refs[slice].product.status["FINETIME"].data/1.e6
    t -= t[0]
    x = Double1d(range(n))
    model = pars[0]+pars[1]*EXP(pars[2]*x)+pars[3]*EXP(pars[4]*x)+pars[5]\
        *EXP(pars[6]*x)
    ptrans.addLayer(LayerXY(t,model, color=java.awt.Color.red,stroke=1.5))
    ptrans.getLayer(ptrans.getLayerCount()-1).setName("Model [red]")

```

- **plotSignalBasic**: is a pipeline helper plot and is explained in [Section 10.2](#). It produces a basic plot of the signal of the central pixel in the central module of the instrument. To plot a different pixel or module, specify the pixel and module to plot.



Tip

The PACS spectrometer detector array has a size of 18,25 (pixels,modules), with science data being contained in pixels 1 to and including 16 only, for the first dimension.

- **plotTransient**: plots the data (black points) from the first science *Frames* (slice = 1) for the central module of the IFU (module = 12) normalised to the signal level in the final grating scan data-chunk. The various grating scans cover the same wavelength range as each other (and there are at least two grating scans in any observation): since the effect of a transient is to increase the signal levels, a transient will make the first spectral scan stand out from those of later in the observation. The second call to this task plots the corrected data as red points.
- **specLongTermTransCorr**: Sudden differences in flux between two adjacent slices cause long-term transients in the detector response. This task fits a transient model to the signal normalised to the last wavelength scan, and applies the correction to the data.

The next step is to tackle the short-term transients.

```
pbasic = plotSignalBasic(slicedFrames, slice=0, titleText="Post LTT")
slicedFrames = specUpDownTransient(slicedFrames, calTree=calTree, verbose=0)
pbasic = plotSignalBasic(slicedFrames, slice=0, titleText="Post UpDown")

slicedFrames = specMedianSpectrum(slicedFrames, calTree=calTree)
slicedFrames = specTransCorr(slicedFrames, calTree=calTree)
pbasic = plotSignalBasic(slicedFrames, slice=0, titleText="Post TC")
```

- **plotSignalBasic**: see above.
- **specUpDownTransient**: This task corrects transients while treating the up-scans and down-scans separately, median-correcting all the scans to the global mean. The correction applied is at the halfway point between each up- and down-scan.
- **specMedianSpectrum**: This task computes a first guess of the spectrum for each spatial module of the *Frames* using the median value of the flux (per wavelength) from the 16 pixels that feed each module.
- **specTransCorr**: This task identifies the discontinuities in the signal caused by cosmic ray hits, fits the subsequent transients, and then corrects the signal for these. Any signal which is too damaged to be corrected is instead flagged in a new mask: UNCORRECTED. To see these data, you can open the *Frames* in slicedFrames in the Spectrum Explorer ([Section 10.5](#)).

6.2.7. Spectral flatfielding: original line scan pipeline script only

Spectral lines existing on top of medium or high-flux continua should benefit from refining the spectral flat fielding, and any other type of spectra should also be at least slightly improved. *It is recommended to compare spectra obtained with and without spectral flat fielding, and to also check on the results of the flatfielding as you do it.* This particularly so lines on weak continua.

The flatfielding tasks are not done in the new ("with transients") pipeline script.

Before doing the flatfielding, you can chose to save the slicedFrames to pool. Then the data are converted to the first of the cubes produced in the pipeline.

```
if saveOutput:
    name=nameBasis+"_slicedFrames_B4FF"
    try:
        saveSlicedCopy(slicedFrames,name, poolLocation=outputDir)
    except:
```

```

print "Exception raised: ",sys.exc_info()
print "You may have to remove the directory: ", outputDir+'/'+name

slicedCubes = specFrames2PacsCube(slicedFrames)
if verbose: slicedSummary(slicedCubes)

```

- **specFrames2PacsCube**: turn the individual *Frames* in the *slicedFrames* into *PacsCubes* held in a *SlicedPacsCubes* product. This really is only a rearrangement of the data. These cubes have a spatial arrangement of 5x5 spaxels (created from the 25 modules), and along the wavelength dimension you will find the spectra from the 16 pixels all packed together one after the other. The spectra from these 16 spaxels may themselves be multiples, if the observer asked for repeats on the wavelength range. At a minimum each pixel holds a spectrum from a grating scan up and one from a grating scan down (i.e. one spectral range sampled twice).

The flatfielding is a multi-step process for these short wavelength range data. Spectral flatfielding is to correct for the differences in the response of the 16 pixels of each of the 25 modules/spaxels, with respect to their 25 mean values. This should improve the SNR in the continuum of the subsequently combined spectrum in each spaxel (combining is the next stage in the pipeline), and will correct for a "spiking" effect in the final spectra that can result if one scan in a pixel is discrepant. The flatfielding is performed in a few steps: (i) outliers are masked out, (ii) spectral lines are identified (so they can be ignored), (iii) the mean continuum level of each pixel is then determined, and each is normalised to the overall mean of the spaxel/module they belong to, and (iv) then masks and intermediate results are cleaned up.

```

# >>>>> 5 Spectral Flat Fielding

# 1. Flag outliers and rebin
upsample = 4
# 3 is mentioned in the pipeline scripts, but 4 is used in the SPG and elsewhere
waveGrid=waveLengthGrid(slicedCubes, oversample=2, upsample=upsample,
    calTree=calTree)
slicedCubes = activateMasks(slicedCubes, StringId(["GLITCH","UNCLEANCHOP",\
    "NOISYPIXELS","RAWSATURATION","SATURATION","GRATMOVE", "BADPIXELS"]), \
    exclusive = True)
slicedCubes = specFlagOutliers(slicedCubes, waveGrid, nSigma=5, nIter=1)
slicedCubes = activateMasks(slicedCubes, StringId(["GLITCH","UNCLEANCHOP",\
    "NOISYPIXELS","RAWSATURATION","SATURATION","GRATMOVE", "OUTLIERS", \
    "BADPIXELS"]), exclusive = True)
slicedRebinnedCubes = specWaveRebin(slicedCubes, waveGrid)

# 2. Mask the spectral lines
widthDetect = 2.5 # default value
threshold = 10. # default value
widthMask = 2.5 # default value
lineList=[]
slicedCubesMask = slicedMaskLines(slicedCubes,slicedRebinnedCubes, \
    lineList=[],widthDetect=widthDetect, widthMask=widthMask, threshold=threshold,\
    copy=1, verbose=verbose, maskType="INLINE", calTree=calTree)

# 3. Actual spectral flatfielding
slopeInContinuum = 1
slicedCubes = specFlatFieldLine(slicedCubesMask, scaling=1, copy=1, \
    maxrange=[55.,230.], slopeInContinuum=slopeInContinuum, maxScaling=2., \
    maskType="OUTLIERS_FF", offset=0, calTree=calTree,verbose=verbose) # see text for
maxRange advice

# 4. Rename mask OUTLIERS to OUTLIERS_B4FF (specFlagOutliers will refuse
# to overwrite OUTLIERS) & deactivate mask INLINE
slicedCubes.renameMask("OUTLIERS", "OUTLIERS_B4FF")
slicedCubes = deactivateMasks(slicedCubes, StringId(["INLINE", "OUTLIERS_B4FF"]))
if verbose: maskSummary(slicedCubes, slice=0)

# 5. Remove intermediate results
del waveGrid, slicedRebinnedCubes, slicedCubesMask

```

- Masks are activated. As discussed in [Section 4.2.3](#) you may want to not include the RAWSATURATION mask in this call (in all instances where it is called).

- **(1) Flag outliers and rebin.** This: (A) masks outliers so they are not included when the flatfielding task calculates its correction, and then (B) automatically identify spectral lines so they are not included when the flatfielding is computed. For these it is necessary to run a few tasks that you will also later encounter in the pipeline.

(A) **wavelengthGrid** creates a wavelength grid that is common to all spaxels. **specFlagOutliers** runs with that wavelength grid, identifying outliers within the new wavelength grid bins, creating a mask called OUTLIERS.

(B) **specWaveRebin** spectrally re-grids the *PacsCubes* with the wavelength grid, to make *PacsRebinnedCubes*, and where the task **activateMasks** ensures that the identified bad data are not included. The rebinned cubes created by this task are not used for anything except line identification by the subsequent task **slicedMaskLines**; they are deleted at the end of the flatfielding process. You can also create a list of spectral lines for **slicedMaskLines** to use, mainly useful if the automatic identification does not find all the lines, or the absorption lines, in the spectra. In this case you do not need to do the first step (#1 in the script snippet).

- **(2) Mask the spectral lines** with **slicedMaskLines**, adding the spectral-line wavelengths to the new mask **INLINE**. See its [URM](#) entry (which is called "maskLines") for a full parameter list, but briefly:

- You can either specify a line list (a list of central wavelengths) or opt for an automatic detection of a spectral lines. *If you have multiple, blended or absorption lines you want to flag out for the continuum fitting part of the flatfielding, you should specify their wavelengths in a line list.*
- If you specify a line list then do not include "slicedRebinnedCubes" in the call to the task, instead fill in the parameter `lineList`. (If you specify a line list *and* ask for automatic identification, the line list will take precedent.) The line list is specified with the parameter `lineList` and is a *PyList* of wavelengths: e.g. `lineList=[52.4, 78.9, 124.4]`
- The automatic line detection is done on the *PacsRebinnedCubes* created previously. The task looks in the central spaxel and identifies emission lines as local maxima in the rebinned cube ($\text{flux} > \text{calculated_local_rms} * \text{threshold}$). This wavelength region is then excluded, for all pixels, in the subsequent continuum work. `widthDetect` sets the width of the box within which the "local maxima" are checked for: the width is a multiple `widthDetect` of the FWHM (where the FWHM is taken from a calibration file). *If you have blended or wide lines but still use the auto-identification, you may want to increase the width factors above the default.*
- Note: **slicedMaskLines** will not work if the parameter `widthDetect` is set too small.

The line centres that are found, or given, are then extended such that the masked-out region width is defined by a given multiple `widthMask`, of the FWHM around the line-peak.

Set `verbose=1` for plots showing the position of the lines automatically identified. A dot will appear above the lines (the plot taken from the central spaxel, for each slice) and the line wavelength is printed to the *Console*.

- **(3) specFlatFieldLine** does the flatfielding.

The working of this task is explained in [Section 7.4](#). First a reference spectrum—the mean of the entire spaxel (excluding bad data)—is created. The spectra from each spaxel are then split into populations and the spectra from each population are compared to the reference spectrum, so-computing a scaling factor. Once the scaling is applied, this brings spectra with excessively high values down and those with excessively low values up, while maintaining the mean. The mean spectrum *then* created will have a better SNR than the mean from before.



Note

What is a "population"? PACS data were taken continuously, while the grating was moving and while the instrument was moving on the sky (e.g. for rasters). Hence there is a population of data that were taken during movements and which will have been masked as bad, and a population that are good data. There are additional populations of

datapoints that come from different parts of the instrument. The PACS signal-detector was constructed from 25 rows of 16 pixels: each pixel was in fact a separate detector with its own response, and each row of 25 corresponds to a single spaxel and hence to a single position on the sky. The signal stream from each pixel for each module is also a separate population. Finally, the response of each pixel differs if you are moving along the grating towards increasing or towards decreasing wavelength, and this makes additional populations.

The scaling applied is a multiplicative correction if you set `scaling` to 1 (default). Setting to 0 you would apply an additive correction. If `verbose=1`, this task will produce a plot showing, for the central spaxel of each slice, the before and after spectrum taken from the *PacsCubes* (plotted as dots) and a rebinned version of each (plotted as a line and much easier to see).

The `slopeInContinuum=1` is for spectra with lines on a continuum with some slope, it effectively "weights" the ratios taken by the task to account for the slope. While this should allow for a better result for each pixel, note that if the slope is slightly different from pixel to pixel then the flatfielding can become slightly pixel-dependent.

`maxScaling` sets a limit to the scaling factors that the task calculates: limiting the scaling factor protects against abnormally large scaling factors that can result when working with spectra with flux levels close to 0. The mask called `OUTLIERS_FF` marks the datapoints that are outliers before the flatfielding is applied: it is an information mask and is not activated later in the pipeline.

The parameter `maxRange` is used to mask out spectral ranges that fall in the light-leak regions. The red light leak comes in after 190 microns, and while the RSRF used in SPG/HIPE 14.2 and 15 is valid in this spectral range, you have to be careful about what you do with the data: this is explained in [Section 5.4](#). In the pipeline scripts the `maxRange` blue limit is 50 microns, but in fact the value of 55 microns should be used: due to a combination of light leak and band edge, the flux calibration is very uncertain between 50 and 55 microns. For more information on the band edges and light leak, currently at herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb?template=viewprint.

The parameter `maxRange` can also be used to focus the flatfielding effort on a particular range (e.g. a single line) in your data. The flatfielding will be done only in that range: no out of range data is cut out, so be aware that the "out" regions will probably look odd.

- (4) and (5) are cleaning up: renaming the masks and removing redundant necessary products, and deactivating the masks that are not wanted later in the pipeline.

You could save `slicedCubes` before running the flatfielding task, to compare the before and after (using the examples in [Section 10.4.2](#)).



Tip

It can be helpful to plot the data before and then after the flatfielding, to see the effect that it has had on your data, using the task `plotPixel`. See [Section 10.2](#) to learn more about this task.

You can also open any one of the slices in `slicedCubes` in the **Spectrum Explorer**, which is a tool for inspecting spectra from any number of Spectrum products in HIPE. You can inspect masked data and different grating scans in the SE. For a general description of the SE, see *DAG* [chap. 6](#), but to know how to use in on the *PacsCubes*, see [Section 10.5](#).

6.2.8. Spectral flatfielding: original range scan pipeline script only

Spectral lines existing on top of medium or high-flux continua should benefit from refining the spectral flat fielding, and any other type of spectra should also be at least slightly improved. *It is strongly recommended to compare spectra obtained with and without spectral flat fielding, and to also check on the results of the flatfielding as you do it.* This particularly so lines on weak continua.

The flatfielding tasks are not done in the new ("with transients") pipeline script.

For short spectral ranges (5 microns or less), and especially if the continuum is fairly flat, it is recommended to use the flatfielding task from the line scan pipeline. How to do this is explained below.

```
# >>>>> 5 Spectral Flat Fielding for range scans

useSplinesModel = True
excludeLeaks = True
slicedFrames = specFlatFieldRange(slicedFrames, polyOrder=3, verbose=verbose, \
    excludeLeaks=excludeLeaks, selectedRange=None, useSplinesModel=useSplinesModel)

slicedCubes = specFrames2PacsCube(slicedFrames)
```

- **specFlatFieldRange.** The flat fielding task that normalises the response of all pixels within each module to the same reference level.

The working of this task is explained in [Section 7.4](#). First a reference spectrum—the mean of the entire spaxel (excluding bad data)—is created. The spectra from each spaxel are then split into populations and the spectra from each population are compared to the reference spectrum, so computing a scaling. Once the scaling is applied, this brings spectra with excessively high values down and those with excessively low values up, while maintaining the mean. The mean spectrum *then* created will have a better SNR than the mean from before.

SpecFlatFieldRange fits either a polynomial function over the available wavelength range as it computes the scaling factors, or—new to HIPE 14—a spline. The spline is recommended for longer ranges and where the spectra are "bendy". For straight and shorter spectra, the splines and polynomial give similar results. The shorter the wavelength range, the lower you should set the order of the polynomial, otherwise you can introduce artificial S-curves in the final spectra. The spline is the recommended function to use in HIPE 14.

SpecFlatFieldRange will work on any range, but, especially in case of SEDs, it will deliver better results if you exclude the leak regions from your data beforehand (`excludeLeaks=True`). This is particularly true if using the polynomial function, which cannot handle large and fast variations of the continuum flux. The spline function works better, but it still sometimes struggles in the red light-leak region. Before SPG/HIPE 14.2 (and 15) it was also the case that the red light-leak spectral regions (redwards of 190 microns) were badly calibrated, but the RSRF has since been updated (version 5 of the calibration file `RsrR1`) and this is no longer the case. However, you do need to pay careful attention to what you do if using data in this red range: see [Section 6.4](#) for more information. See the Observer's Manual to learn about these leak regions (you can currently find it here: herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb?template=viewprint).

The parameter `selectedRange` can be used to specifically select a range of the data to flat-field, useful if you are interested only in a certain spectral region. The format of the parameter is `[waveMin, waveMax]`.

To change the box size for the sigma-clipping, set `maxbinsize` to a value in microns (2 is the default). To see plots of the fits set `doPlot=1`. These plots will allow you to assess the quality of the fitting.

- **specFrames2PacsCube:** turn the individual *Frames* in the *slicedFrames* into *PacsCubes* held in a *SlicedPacsCubes* product. This really is only a rearrangement of the data. These cubes have a spatial arrangement of 5x5 spaxels (created from the 25 modules), and along the wavelength dimension are the spectra from the 16 pixels all packed together one after the other. The spectra from these 16 spaxels may themselves be multiples, if the AOR asked for repeats on the wavelength range. At a minimum each pixel holds a spectrum from a grating scan up and one from a grating scan down (i.e. one spectral range sampled twice).

You could save `slicedFrames` before running the flatfielding task, to compare the before and after (using the examples in [Section 10.4.2](#) or the example in the pipeline script).

**Tip**

It can be helpful to plot the data before and then after the flatfielding, to see the effect that it has had on your data, using the task plotPixel. See [Section 10.2](#) to learn more about this task (note: it does also work on *slicedFrames*).

You can also open any one of the slices in slicedCubes in the **Spectrum Explorer**, which is a tool for inspecting spectra from any number of Spectrum products in HIPE. You can inspect masked data and different grating scans in the SE. For a general description of the SE, see *DAG chap. 6*, but to know how to use in on the *Frames* of PACS Level 0, 0.5 and 1, see [Section 10.5](#).

6.2.8.1. Using the line scan flatfielding task on short ranges

To use the flatfielding task from the line scan pipeline script is not particularly difficult. It is recommended for ranges of 5 microns or less.

Begin with,

```
slicedCubes = specFrames2PacsCube(slicedFrames)
```

because line scan flatfielding requires cubes rather than *Frames* as input. Follow the subsequent flatfielding steps in the line scan pipeline script, which are explained in the previous section, starting with

```
waveGrid=wavelengthGrid(slicedCubes, oversample=2, upsample=3, calTree=calTree)
```

and ending with

```
del waveGrid, slicedRebinnedCubes, slicedCubesMask
```

This then takes you to the place in the range scan pipeline script where Level 1 ends:

```
# -----
#           Processing           Level 1 -> Level 2
# -----
```

6.2.9. Select the on/off slices: for both the line scan pipeline scripts

Now we can continue with the pipeline.

```
# >>>>> 6 Select slices for line scan AOTS
if verbose: slicedSummary(slicedCubes)
lineId      = []
wavelength  = []
rasterLine  = []
rasterCol   = []
nodPosition = ""
nodCycle    = []
band        = ""
scical      = ""
sliceNumber = []

onOff = "ON"
sCubesOn = selectSlices(slicedCubes, lineId=lineId, \
    wavelength=wavelength, \
    rasterLine=rasterLine, \rasterCol=rasterCol, \
    nodPosition=nodPosition, \
    nodCycle=nodCycle, band=band, scical=scical, \
    sliceNumber=sliceNumber, \
    onOff=onOff, verbose=verbose)
onOff = "OFF"
sCubesOff = selectSlices(slicedCubes, lineId=lineId, \
    wavelength=wavelength, \
```



```

rasterLine=rasterLine,\rasterCol=rasterCol, \
nodPosition=nodPosition, \
nodCycle=nodCycle, band=band, scical=scical,\
sliceNumber=sliceNumber, \
onOff=onOff, verbose=verbose)

if verbose:
    slicedSummary(slicedCubesOn)
    slicedSummary(slicedCubesOff)

```

The difference between the line and range scan AOT here is that the line scan has "on" and "off" data, so two "slicedCubes" are made, while the range scan is only either "on" or "off", and so no new "slicedCubes" need to be made.

- **selectSlices** selects a subset of the data to work on, e.g. only one spectral line, the on-source and off-source slices, only one band, For very large observations the memory usage of the pipeline can be reduced by selecting to process only some slices at a time. This is normally not necessary, which is why in the pipeline script the only filled selection parameter is `onOff`. This *is* necessary for the line scan unchopped observations.

Toward the end of the pipeline any selected-out data need to be concatenated: the task that does this is included in this pipeline script.

- **slicedSummary** has been described before.

6.2.10. Wavelength grid and outlier flagging

The spectra in each spaxel of each *PacsCube* are the spectra of the 16 pixels of each module of the input *Frames*, where each pixel had two or more spectra that covered very nearly, but not exactly, the same wavelength range and wavelength sampling. This throng of wavelength datasets need to be regularised, to a single grid for each spaxel. It is on this new grid that the spectra will be resampled to create a single spectrum per spaxel per cube. The bins of the new grid are large enough to include several input data-points (and hence to improve the SNR in the result), but small enough to still allow for at least a Nyquist sampling of the spectral resolution at every wavelength.

```

# >>>>> 7 Wavelength grid and outlier flagging

# for line scan
if ((not locals().has_key('multiObs')) or (not multiObs)):
    oversample = 2
    upsample   = 4

waveGrid=wavelengthGrid(sCubesOn, oversample=oversample, \
    upsample=upsample, calTree = calTree)

sCubesOn = specFlagOutliers(sCubesOn, waveGrid, nSigma=5, nIter=1)
sCubesOff = specFlagOutliers(sCubesOff, waveGrid, nSigma=5, nIter=1)

# for range scan
if ((not locals().has_key('multiObs')) or (not multiObs)\
    or (not locals().has_key('waveGrid'))):
    oversample = 2
    upsample   = 2
    waveGrid=wavelengthGrid(slicedCubes, oversample=oversample, \
        upsample=upsample, calTree = calTree)

# Do this unless you used the line scan flatfielding
slicedCubes = activateMasks(slicedCubes, slicedCubes.get(0).maskTypes, exclusive =
    True)

# If you used the line scan flatfielding, do this instead
slicedCubes = activateMasks(slicedCubes, \
    StringId([str(i) for i in slicedCubes.get(0).maskTypes if i \
        not in ["INLINE", "OUTLIERS_B4FF"]]), exclusive = True)

slicedCubes = specFlagOutliers(slicedCubes, waveGrid, nSigma=5, nIter=1)

```

If you want to test out several grids, it is useful to save "slicedCubes" to disk first, to always have a clean copy to work with (using `saveSlicedCopy` and `readSliced` to save and reload). See [Section 10.4.2](#) for some examples using `PlotXY` to plot before- and after-task spectra. Note that the best product to check the effect of the `wavelengthGrid` is that produced by the subsequent task `specWaveRebin` (next section).

- Masks are activated. As discussed in [Section 4.2.3](#) you may want to not include the `RAWSATURATION` mask in this call. To do this you can add the `RAWSATURATION` to the `"["INLINE", "OUTLIERS_B4FF"]"` syntax in the code snippet above at "If you used the line scan flatfielding", and copy over that same syntax to the call for the range scan flatfielding pipeline.
- **wavelengthGrid**: creates the wavelength grids. This task creates a mask called `OUTOFBAND`. This identifies spectral regions that have fallen out of the range of the PACS filters, something that can happen if you request a range in one camera that results in an "illegal" range in the other camera. This is provided to avoid that the user works on data that are invalid.

The `upsample` and `oversample` parameters are explained in the [PACS URM](#) entry for `wavelengthGrid`, and the effect of different choices of parameters is explained further in [Section 7.2](#).

- **activateMasks** for the range scan pipeline, if you used the flatfielding of the line scan pipeline, the `slicedCubes` will have two masks you do not want to activate: `INLINE` and `OUTLIERS_B4FF`. Hence the difference in the call to `activateMasks` for the case of using line scan flatfielding or the one in the range scan pipeline script.

As discussed in [Section 4.2.3](#) you may want to not include the `RAWSATURATION` mask in this call.

- **specFlagOutliers**: flag for outliers, i.e. a second level glitch-detection task, creating a mask called `OUTLIERS`. It will ignore all data that are masked as bad in all active masks. This task works by first rebinning the data according to the specified wavelength grid, and then looking for all outliers by searching the datapoints that fall in the new grid, going bin by bin. It only uses the `waveGrid` to find the outliers, it does not also change the wavelength grid of the input cubes. `nIter` controls the number of iterations you do (1 means two runs of detection are done), and `nSigma` controls the sigma level you clip at. The parameter `nSigma` has the largest effect of the two on the performance of this task.

To look at the `OUTLIERS` in the `PacsCubes` use the task `plotCubes` ([Section 10.2](#)), the examples given in [Section 10.4.4](#), or use the Spectrum Explorer ([Section 10.5](#)).

Note that the previous deglitching task, `specFlagGlitchFramesQTest`, works on the time-line, whereas this task works with the wavelengths. Neither task tries to fix the glitches, they only mask them.

6.2.11. Spectral rebinning

```
# >>>>> 8 Spectral rebinning and averaging the cubes

# for line scans
slicedRebinnedCubesOn = specWaveRebin(sCubesOn, waveGrid)
slicedRebinnedCubesOff = specWaveRebin(sCubesOff, waveGrid)

if verbose:
    slicedSummary(slicedRebinnedCubesOn)
    slicedSummary(slicedRebinnedCubesOff)
    overlay = None
    pradec = plotCubesRaDec(slicedRebinnedCubesOn, subtitleText=\
        "on-source positions", overlay = overlay)
    pradec = plotCubesRaDec(slicedRebinnedCubesOff, pradec, \
        subtitleText="on- and off-source positions", overlay = overlay)

slicedRebinnedCubesOn = specAverageCubes(slicedRebinnedCubesOn)
slicedRebinnedCubesOff = specAverageCubes(slicedRebinnedCubesOff)
```

```

# for range scans
# Do this unless you used the line scan flatfielding
slicedCubes = activateMasks(slicedCubes, slicedCubes.get(0).maskTypes, \
    exclusive = True)

# If you used the line scan flatfielding, do this instead
slicedCubes = activateMasks(slicedCubes, \
    StringId([str(i) for i in slicedCubes.get(0).maskTypes if i \
    not in ["INLINE", "OUTLIERS_B4FF"]]), exclusive = True)

# and then, in both cases
slicedRebinnedCubes = specWaveRebin(slicedCubes, waveGrid)

if verbose:
    slicedSummary(slicedRebinnedCubes)
    overlay = None
    p9 = plotCubesRaDec(slicedRebinnedCubes, overlay = overlay)

# Average all the cubes, per raster position
slicedFinalCubes = specAverageCubes(slicedRebinnedCubes)

```

Here you do the spectral rebinning that combines the spectra held in each spaxel of the *PacsCubes* into one spectrum (per spaxel), improving the SNR and regularising the wavelength grid. This process will only include data *not* flagged as bad in the masked activated before the task is run. *Hence, if you have saturated data* then the saturated data-points may *not* be found in the resulting cubes: the saturated regions may become blank. More specifically, any wavelength bin that is masked as saturated in the input cube will have, in the output cube, either have a NaN value if *all* the datapoints that fed that bin are saturated, or will have an actual value if only *some* of the datapoints that fed that bin are saturated.

Some examples of checking on the masked data are given in [Section 10.4.4](#). See [Section 7.6](#) for more information about the effect of the saturation and glitch masks on the rebinned of the *PacsCubes* by `specWaveRebin`.



Note

The wavelength grid increases with resolution, i.e. with wavelength: the wavelength range of the PACS spectrograph is so long that to achieve at least Nyquist sampling at all wavelengths, it is necessary that the bin sizes scale with wavelength. This wavelength grid is not held in a World Coordinate System (WCS, specifically axis 3), but rather in an "ImageIndex" dataset of the cubes. For more information, see the PPE chps [3](#) in *PACS Products Explained* and [5](#) in *PACS Products Explained*.

- **activateMasks** for the range scan pipeline, if you used the flatfielding of the line scan pipeline, the `slicedCubes` will have two masks you do not want to activate: `INLINE` and `OUTLIERS_B4FF`. Hence the difference in the call to `activateMasks` for the case of using line scan flatfielding or the one in the range scan pipeline script.

As discussed in [Section 4.2.3](#) you may want to not include the `RAWSATURATION` mask in this call. To do this you can add the `RAWSATURATION` to the `"["INLINE", "OUTLIERS_B4FF"]"` syntax in the code snippet above at "If you used the line scan flatfielding", and copy over that same syntax to the call for the range scan flatfielding pipeline.

- **specWaveRebin**: takes the input wavelength grid and rebins the spectra (via an averaging) from the input cubes and places them in the output cubes. It does this separately for each cube held in the input `slicedCubes`, so the differences between the cubes—raster pointing, wavelength range, nod (A and B), nod cycle—are maintained. What *are* combined, per spaxel, are the spectra from the 16 pixels for all the repetitions on the grating scan—and what *are not* combined are the repetitions on nod cycle, as different nods are held as separate slices. The output is a set of *PacsRebinnedCubes* held in a *SlicedPacsRebinnedCube*.

It is important to activate all masks containing bad data you do not want included in your final cubes: including the not flatfielded data, the bad chops and grating movement data, glitches, outliers, saturation, noisy and bad pixels, out of band data. The pipeline activates all masks by default (in `activate masks`).

**Note**

For data obtained in Nyquist sampling mode before OD 305, it is normal to find empty bins (NaNs in the rebinned cube) even when rebinned with oversample=2. This is intrinsic to these data, and there is no way to correct for this.

Noise/Errors: This task creates a standard deviation dataset ("stddev"), which is explained in [Section 7.7](#), and you can also check the [URM](#) entry for this task to learn more about this. You will be given a chance to plot the error curve in the next code block, and this plot is also explained in [Section 7.7](#).

- **plotCubesRaDec** Plots the Ra and Dec coverage of the cube(s), i.e. the sky footprint of the observation, with the option of over-plotting this footprint on an image. See [Section 10.2](#) to learn more about this task.
- **specAverageCubes** combines all the separate cubes of the same pointing and wavelength range.

6.2.12. Subtract the background: line scan only

```
# >>>>> 9 For line scan: subtract the off-cubes from the on-cubes

if verbose:
    x,y = 2,2
    activeMasks = sCubesOn.refs[0].product.getActiveMaskNames()
    ponoff = plotCubes(sCubesOn,[],x=x,y=y,masks=activeMasks)
    ponoff = plotCubes(sCubesOff,ponoff,x=x,y=y,masks=activeMasks)
    ponoff = plotCubes(slicedRebinnedCubesOn, ponoff,x=x,y=y, stroke=1)
    ponoff = plotCubes(slicedRebinnedCubesOff,ponoff,x=x,y=y, stroke=1)
    ponoff.titleText,ponoff.subtitleText=str(obsid)+" "\
        +camera,"Data cubes and averaged rebinned cubes (on- and off-source). Spaxel ["\
        +str(x)+","+str(y)+"]."

slicedRebinnedCubesAll = concatenateSliced([slicedRebinnedCubesOn, \
    slicedRebinnedCubesOff])
if verbose: slicedSummary(slicedRebinnedCubesAll)
slicedFinalCubes = specSubtractOffPosition(slicedRebinnedCubesAll)
```

- **plotCubes** plots of the spectra in a spaxel of your cube. The plots here compare the on-source and off-source spectra. See [Section 10.2](#) to learn more about this task.
- **concatenateSliced** to combine the indicated cubes (here the off-source and on-source) into a single *ListContext* ("slicedRebinnedCubesAll"), so that they can be considered together in the subsequent task. Meta data are used to indicate which cubes are on-source and which are off-source, so the order they are combined in does not matter.
- **specSubtractOffPosition** subtracts the off-cubes from the on-cubes. The stddev array is propagated.

There are a number of algorithms you can use in `specSubtractOffPosition`, and for observations with more than one off-source pointing (especially if taken between a long on-source observation) it is important to test these out. The algorithms are explained in their [PACS URM](#) entry: chose the closest in time, the average of all, or the time-weighted interpolate, off-source position(s) in the subtraction. See also the advice given in [Section 7.5](#).

(For the range scan AOTs the subtraction is done in the Combine On Off pipeline script: [Section 6.3](#).)

6.2.13. Plot and save the results

```
# >>>>> 10 Plotting and saving

if verbose:
```

```

x,y = 2,2
pFinal = plotCubes(slicedFinalCubes, x=x, y=y,stroke=1,\
    title="plotCubes - "+str(obsid)+" "+camera,subtitle="Final Rebinned Spectrum.
Spaxel ["\
    +str(x)+","+str(y)+"].\n No point source correction applied")
pstd = plotCubesStddev(slicedFinalCubes, plotLowExp=1, plotDev=0, nsigma=3,
islineScan=1,\
    spaxelX=x, spaxelY=y, verbose=verbose, calTree=calTree, wranges=None)
pstd.titleText,pstd.subtitleText="plotCubesStddev - "+str(obsid)+" "\
    +camera,"Final Rebinned Spectrum & uncertainties. Spaxel ["+str(x)+",\
    "+str(y)+"].\n No point source correction applied"
slice = 0
p55 = plotCube5x5(slicedFinalCubes.get(slice), frameTitle="plotCube5x5 - "\
    +str(obsid)+" "+camera+" slice "+str(slice))

# for range scans
if saveOutput:
    name = nameBasis+"_slicedCubes"
    saveSlicedCopy(slicedCubes, name, poolLocation=outputDir)
    name = nameBasis+"_slicedRebinnedCubes"
    saveSlicedCopy(slicedRebinnedCubes, name, poolLocation=outputDir)
    name = nameBasis+"_slicedFinalCubes"
    saveSlicedCopy(slicedFinalCubes, name, poolLocation=outputDir)

# for line scans
if saveOutput:
    name = nameBasis+"_sCubesOn"
    saveSlicedCopy(sCubesOn, name, poolLocation=outputDir)
    name = nameBasis+"_sCubesOff"
    saveSlicedCopy(sCubesOff, name, poolLocation=outputDir)
    name = nameBasis+"_slicedRebinnedCubesAll"
    saveSlicedCopy(slicedRebinnedCubesAll, name, poolLocation=outputDir)
    name = nameBasis+"_slicedFinalCubes"
    saveSlicedCopy(slicedFinalCubes, name, poolLocation=outputDir)

```

- **plotCubesStddev** plots the spectrum of a single spaxel with the standard deviation and continuum RMS datasets overplotted. See [Section 10.2](#) to learn more about this task and the RMS dataset it plots. The parameter `islineScan` determines whether the task is used in a way appropriate for line scans (1), range scans (0), or either (the task determines this itself by looking at the Meta data to find out what whether the observation is line or range).
- **plotCubes** plot the spectrum in a spaxel of your cube. **plotCubes5x5** produces a 5x plots of the 25 spaxels in the cube. See [Section 10.2](#) to learn more about this task.
- If you chose to save output to disk, the cubes of Level 2 are saved with **saveSlicedCopy**, to a pool on disk in the directory called "outputDir" and with a pool name (sub-directory name) "name". To recover these various sliced cubes, use the same `poolLocation` and `poolName` in the task "readSliced". See their [PACS URM](#) entry to learn more about saving and loading.

Now turn to [Section 4.3](#) to learn about what to do next for extended and point sources for the line scan AOTs. For range scan and SED AOTs it is necessary to pipeline process both the on-source and off-source observations before doing the background subtraction. This is explained in the next section.

6.3. Helper script 'Combine off-source with on-source' in unchopped range spectroscopy

6.3.1. Explanation

In the unchopped AOT for range Spectroscopy the on-source and off-source observations are two separate obsids. Hence you first reduce each observation and then do the background subtraction. The script "Combine on-off obs" will do this in a semi-automatic way: you set some parameters, set up the pipeline script to run on each obsid, and then one pair of on-source and off-source observations are reduced in a loop. After this, the background subtraction task is run and the data are saved to disk.

Since the pipeline processing is done by calling on a pipeline script in a loop, it is a good idea to be familiar with the pipeline. If you need to change some of the parameters of the tasks in the script, then it is also necessary that you copy the script to a unique location on disk and run that copy, rather than the default one in the HIPE build. Some necessary parameters are set in the "Combine off-source with on-source" script (e.g. directory names) but others (e.g. verbose) have to be set in the pipeline script itself.

After running this you can turn to [Section 4.3](#) to read about the post-processing tasks for extended and point sources. Some of these tasks are included in the script here.

Note that "Combine ..." will reduce only one on-source and one off-source observation. If you have more than one off-source obsid, you will need to use the alternative script instead (or modify the helper script yourself).

6.3.2. Alternative useful script

If you have reduced the on-source and off-source observations already, and saved to disk, you can do the subtraction with the script in the HIPE menu *Scripts#PACS useful scripts#Spectroscopy: off-subtraction and post-processing in unchopped range spectroscopy*. This does the same as the script in the pipeline menu, but in a manual fashion and you can run on any number of on-source and off-source observations. After running this you can turn to [Section 4.3](#) to read about the post-processing tasks for extended and point sources. Some of these tasks are included in this useful script.

6.3.3. Running the helper script

6.3.3.1. Setup

It is likely that you will make at least *some* edits to the pipeline script that the "Combine ..." script calls on: we recommend you edit and copy the pipeline script to a new name/location. Settings to note are:

- *useHsa*: used in `getObservation`, to get data directly from the HSA or get it from pool on disk; set `useHsa=1` or `0` in the pipeline script
- *poolName and/or poolLocation*: used in `get/saveObservation`, sets either where the data are on disk, or to where to save them, the are set in the pipeline script
- *saveOutput*: in the pipeline script is ignored in the "Combine ..." script: output is always saved as coded in the combining script.
- *scriptsDir*: to run the pipeline from your edited copy of the default script, change this from

```
scriptsDir = HCSS_DIR + "/scripts/pacs/scripts/ipipe/spec/"
```

to your own directory name.

The first stage:

```
# 1
multiObs = 1
verbose = 1

# 2
# Chose the obsids
# Input your values
obsidOn = 1342.....
obsidOff = 1342.....
# or entered in a file. An example of obsids.py file:
# --- start of file ---
obsidOn = 1342.....
obsidOff = 1342.....
# --- end of file ---
# and to then read in that file
#obsidsDir = Configuration.getProperty("user.dir")
```

```
#execfile(obsidsDir + "obsids.py")

# 3
HCSS_DIR = Configuration.getProperty("var.hcss.dir")
scriptsDir = HCSS_DIR + "/scripts/pacs/scripts/ipipe/spec/"
script = scriptsDir + "UnchoppedRange.py"
# Or, e.g.
script = "/Users/me/Scripts/myUnchoppedrangeScan.py"

outputDir = None
buildNumber = str(Configuration.getProjectInfo().track) + '.' \
    + str(Configuration.getProjectInfo().build)
buildNumber = buildNumber.replace('.', '_')
```

- (1): In most pipeline scripts is the following line of code

```
if ((not locals().has_key('multiObs')) or (not multiObs)):
    obsid = 1342229704
```

By setting **multiObs** to True/1, you are telling the pipeline script that the indicated parameter (obsid in the example above) will be taken from what is set in the "Combine ..." script rather than in the pipeline script.

- (1): **verbose** to produce (1) or not (0) the pipeline helper task plots
- (3): "HCSS_DIR" and "scriptsDir" together are where the interactive pipeline scripts can be found: HCSS_DIR is the location of the HIPE software, and scriptsDir is the standard location for the PACS scripts in the HIPE build. "UnchoppedRange.py" is the name of the range scan pipeline script; the names of the scripts can be found as the name of the tab when you load them into HIPE via the Pipeline menu. To instead run your own version of the pipeline script, use the second example for "script". "outputDir" is the poolLocation to where data are saved in following loop.
- (3): The build number is there because it is interesting to know.

6.3.3.2. Run the loop

A loop over all the obsids and cameras, running the specified pipeline script on each *ObservationContext*, starting from getting the *ObservationContext* into HIPE.

```
for camera in ["blue","red"]:
    # (1)
    if locals().has_key("waveGrid"): del waveGrid
    # (2)
    obsid = obsidOn
    execfile(script)
    observingMode = obs.obsMode
    slicedCubesOn = specSetOnOffSource(slicedCubes,1)
    slicedRebinnedCubesOn = specSetOnOffSource(slicedRebinnedCubes,1)
    slicedAveragedCubesOn = specSetOnOffSource(slicedFinalCubes,1)
    del slicedCubes, slicedRebinnedCubes, slicedFinalCubes
    # (3)
    obsid = obsidOff
    execfile(script)
    slicedCubesOff = specSetOnOffSource(slicedCubes,2)
    slicedRebinnedCubesOff = specSetOnOffSource(slicedRebinnedCubes,2)
    slicedAveragedCubesOff = specSetOnOffSource(slicedFinalCubes,2)
    del slicedCubes, slicedRebinnedCubes, slicedFinalCubes
    # (4)
    allAveragedCubes = \
        concatenateSliced([slicedAveragedCubesOn,slicedAveragedCubesOff])
    nameBase = "OBSID_"+str(obsidOn)+"_"+str(obsidOff)+"_"+camera+\
        "_"+buildNumber
    name=nameBase+"_slicedAveragedCubes_All"
    saveSlicedCopy(allAveragedCubes, name, poolLocation=outputDir)
    # (5)
    slicedDiffCubes = specSubtractOffPosition(allAveragedCubes)
    name=nameBase+"_slicedDiffCubes"
```

```
saveSlicedCopy(slicedDiffCubes, name, poolLocation=outputDir)
#
# Post-processing: see Section 4.3
```

For each camera, the on-source and off-source observations are reduced, then they are pushed into the task `specSubtractOffPosition` to do the background (off) subtraction.

- **(1):** if the variable `waveGrid` is defined, delete it, otherwise it will prevent the pipeline from running correctly.
- **(2) and (3):** Processing the on(off)-source observation. The pipeline itself is executed in with the command `execfile(script)`. This will execute whatever commands are in "script". Then find out what the observing mode is, as this is used later in the loop. Next mark the data as being on(off)-source with `specSetOnOffSource`. Rename the pipeline products by adding the "on" or "off" appendage.
- **(4): `concatenateSliced`** combine "slicedAveragedCubesOn" and "...Off" into a single `ListContext`, to be used in the next task. This is followed by more saving.
- **(5): `specSubtractOffPosition`:** subtracts the off from the on. The stddev array is propagated.

There are a number of `algorithms` you can use in `specSubtractOffPosition`, and for observations with more than one off-source pointing (especially if taken between a long on-source observation) it is important to test these out. The algorithms are explained in their [PACS URM](#) entry: chose the closest in time, the average of all, or the time-weighted interpolate, off-source position(s) in the subtraction. See also the advice given in [Section 7.5](#).

- See [Section 4.3](#) to learn about the post-processing tasks that can now be applied.

6.4. Pipeline steps for spectral lines in the light leak regions (longer than 190 μ m)

An update to the calibration tree for SPG/HIPE 14.2 and 15 means that the RSRF in the red region (band R1: calibration file `RsrR1`) now calibrates the spectral lines reasonably well. The continuum will be wrong, but the spectral line fluxes can be recovered. Note, however, that the spectral ranges below 55 microns and above 190 microns are not included in the `ObservationContexts` gotten from the HSA (even for v14.2). This is in particular because the red spectral region is difficult to flatfield well within the SPG script.

To recover the red-leak spectral region (190 microns and upwards), you can reduce the data in HIPE yourself using any of the unchopped pipeline scripts. The only change with respect to the script in HIPE and as described in this chapter is at the flatfielding stage:

- Range scans: set `excludeLeaks=False`. In addition, and especially if dealing with SEDs (long ranges), it will improve results if you limit the region included in the flatfielding to around your line. The parameter `selectedRange` can be used. Alternatively you can use the line scan flatfielding task and set the parameter `maxRange` to include your line only. See [Section 6.2.8](#) to learn about using the line scan flatfielding task while running the range scan pipeline script.
- Line and short range scans: for line scans it is probably not necessary to make any changes as the spectral range is so short anyway, but for ranges of a few microns (and especially if this includes some spectral slope) you can use the parameter `maxRange` to limit the flatfielding to around your red line(s).

A consequence of limiting the range that is flatfielded is that the not-flatfielded parts of the spectrum could look very strange. At the end of the pipeline these regions are anyway cut out, since they have the mask `NOTFFED` and hence are excluded when the Level 2 rebinned cubes are created. But between this part of the pipeline and the end, inspecting the cubes may be difficult due to strong curvature induced by the partial flatfielding. To make subsequent inspection of such cubes easier, you could use the following task to cut the unwanted regions out:


```
newSlicedCubes = pacsExtractSpectralRange(slicedCubes, waveRanges=[[200,205]])
```

This task creates a "newSlicedCubes" including only the data within the wavelength range specified. The parameter `waveRanges` can take a list of more than one [min, max]—[[200,203],[203,204]] for example—but if you want to select only one then a double [[]] is still necessary. The task will work for *SlicedPacsCube* and *SlicedPacsRebinnedCube* or any one *PacsCube* or *PacsRebinnedCube*.

It is important to note that since the continuum level for the red-leak calibrated spectral regions will always be wrong, when using the task `extractCentralSpectrum` on **point sources**, only the result "c1" or "c9" can be used, not also "c129" (since this latter spectrum computes a correction that is based on the continuum level). This advice is anyway also true for any unchopped data, at whatever wavelength range.

See the Observer's Manual and the PACS spectrometer calibration document to learn more (you can obtain these from the HSC PACS web-page: currently at herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb?template=viewprint).

Chapter 7. More detail on pipeline tasks

7.1. Introduction

In this chapter you will find more detail on the following pipeline tasks:

- Creating the wavelength grid for the rebinned cubes: [Section 7.2](#).
- Creating an equidistant wavelength grid: [Section 7.3](#). This is not done in the pipeline, but the standalone browse products offered via the HSA and in the downloaded *ObservationContexts* include so-called equidistant cubes. The standalone browse products are discussed in the [PPE](#) in *PACS Products Explained*, where you can also find the information to make these cubes yourself.
- The spectral flatfielding, and how to test different flatfield parameters: [Section 7.4](#).
- Some advice on background subtraction for unchopped modes: [Section 7.5](#).
- The glitch/outlier detection, and saturation: [Section 7.6](#).
- The data errors (not the calibration errors) of the Level 2 cubes: [Section 7.7](#).
- A few general comments on rebinned and mosaic cubes and the propagation of NaNs and flags: [Section 7.8](#).
- The task `drizzle`, which does the spectral projection and mosaicking of the Level 2 cubes. This is the preferred task for short ranges (1-2 microns), mapping observations: [Section 7.9](#).
- The task `specProject`, which also does the spectral projection and mosaicking of the Level 2 cubes. This is the preferred task for longer ranges, mapping observations: [Section 7.10](#).
- The task `specInterpolate`, which also created mosaic cubes at Level 2, via an interpolation over a Delaunay grid. This is the preferred task for tiling observations: [Section 7.11](#).
- The tasks of the "Pointing offset correction" pipeline script, for `chopNod`, pointed observations of centred point sources: [Section 7.12](#).
- The tasks of the transient correction pipelines (unchopped observations): [Section 7.13](#).

7.2. The pipeline task `wavelengthGrid`

7.2.1. As used in the pipeline

The Level 1 to 2 pipeline task `wavelengthGrid` is used to create a wavelength grid that the *PacsCubes* are resampled on to create the subsequent *PacsRebinnedCubes*. This is necessary because the wavelength grid in the *PacsCubes* is a combination of several, slightly offset wavelength grids (each from a discrete spectrum), with the end result that the total grid is not regularly sampled and is not exactly the same for the different spaxels and the different cubes in an observation. For the pipeline to proceed, it is necessary that the wavelength grid between spaxels and cubes is harmonised and regularised. The spectra of the *PacsCubes* are then combined and resampled on this grid, resulting in the smoother spectra of the *PacsRebinnedCubes*.

The important parameters of the task `wavelengthGrid` are `upsample` and `oversample`, which determine the sampling of the wavelength grid.

- The parameter `oversample` sets by how much you wish to oversample the spectral resolution, i.e. how many bins you want the grid to have per resolution element. So, if the instrument resolution is 1 micron and you set `oversample` to 2, each bin will be 1/2 micron wide.

Since the resolution varies with wavelength, so do the bin sizes, but always with the same `oversample` factor. This means that while the final wavelength grid is regular, it is not equidistant. Plots of bin size vs. wavelength can be found below ([Section 7.2.4](#)).

- The parameter `upsample` sets by how many (new) bins you want to shift forward when calculating the new grid. So, if you set `upsample` to 1 then each (new) bin will begin where the previous (new) bin ended. If you set `upsample` to 2, then the second (new) bin will begin half-way through the previous (new) bin, and so on.

Setting `upsample=1` and `oversample=2` corresponds to the native resolution of the instrument (in the mode your data were taken in). The pipeline scripts and the SPG scripts set `oversample=2` and `upsample=4` for line scans, and `oversample=2` and `upsample=2` for range scans. A value of `oversampling > 1` is possible for line scans and deep range scans, where the "native" spectral sampling is more than enough to provide sufficient datapoints per wavelength to "oversample" (i.e. there is a sufficient redundancy factor). This is also the case for observations which requested a Nyquist-spectral sampling if the repetition factor was 2 or higher.

7.2.2. The oversample and upsample parameters

We have performed tests varying these parameter values and inspecting the resulting rebinned spectra. *The width and peak of the emissions lines change as the parameters do, but the integrated line fluxes are not significantly affected. It is more the measured noise statistics and how easily lines can be seen that is affected.* For older obsids taken in Nyquist sampling mode, the number of NaNs you get with different parameter settings may also be a factor (see note at the end of the section).

Note that the *actual* noise in your spectra is fixed, no matter what grid you set: it is the *apparent, measured*, noise in the rebinned spectra, the scatter in the data, that is affected. Also, bear in mind that the wavelength grid you ask for will depend on (i) the depth of your data (how much redundancy there is), and (ii) somewhat on your scientific goals (e.g. smooth continuum vs. seeing every wart and dimple).

- Consider that: the pixel (bin) sizes of the grid = spectral resolution/`oversample`. Because the scatter in the spectrum rebinned along this grid depends on the number of datapoints included in the averaging, the scatter decreases with decreasing `oversample` (increasing pixel size).
- Consider that: the pixel size in the rebinned spectra is set by the separation between two consecutive points in the `wavelengthGrid`. This scale depends on `upsample` *and* `oversample`: pixel size = spectral resolution/(`oversample`*`upsample`).
- *Case*: Fix `upsample` to 1, which means that each datapoint of the bins of the new grid is independent of the datapoints of the neighbouring bins. As `oversample` decreases the pixel size used for the averaging increases and the noise decreases. At the same time, the line width increases because we are using bigger pixels to sample the line, and the line peak decreases. If we measure the scatter in the continuum, it will decrease with the square root of the pixel size in `wavelengthGrid` (as expected) because in this case it depends only on `oversample` (we have fixed `upsample=1`):

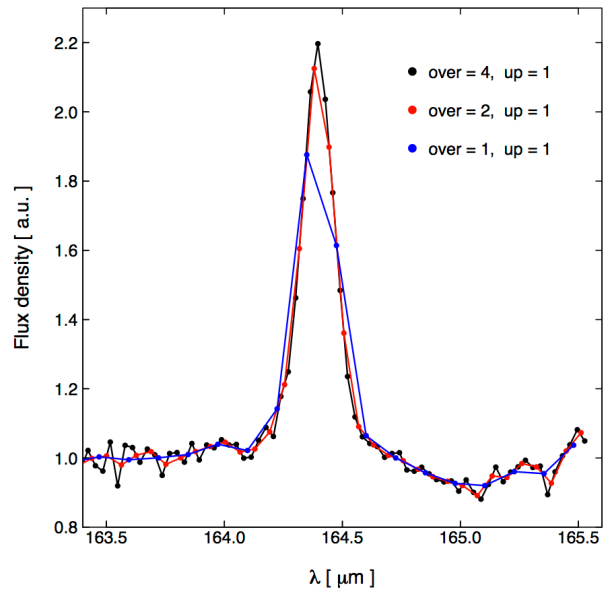


Figure 7.1. upsample fixed, vary oversample

- *Case:* Fix `oversample` to 1, which means that the bin sizes equal to the resolution at every wavelength. The noise should be very similar to that of the input spectra. As `upsample` increases, the number of points in the spectrum (`wavelengthGrid`) increases and datapoints become less and less independent (they share more and more data with the consecutive pixels). The line shape looks smoother, and the noise looks better, since we are smoothing the spectrum. The line width decreases with increasing `upsample`, and the peak increases (the flux is therefore conserved). This is due to the fact that `oversample` remains constant.

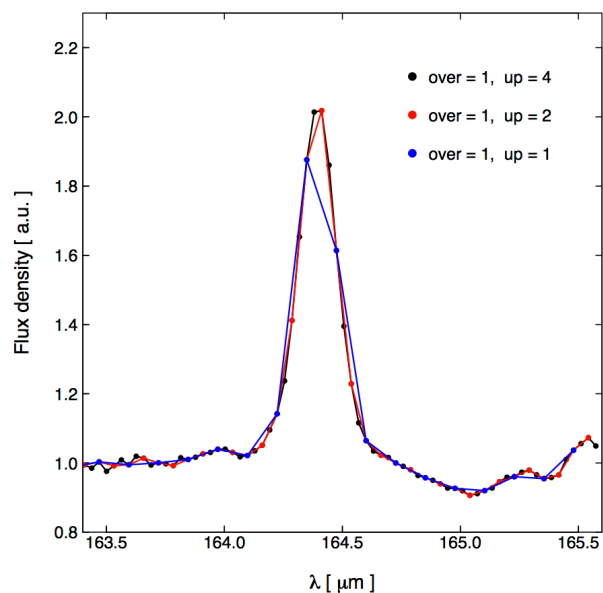


Figure 7.2. oversample fixed, vary upsample

- *Case:* Change `upsample` and `oversample` together, in a way that the spectral pixel separation in `wavelengthGrid` remains constant (`upsample*oversample = constant`). The result is a combination of the two previous effects.

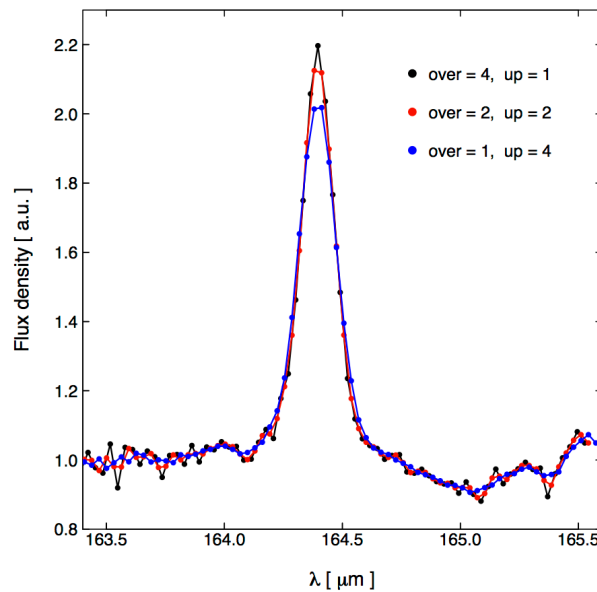


Figure 7.3. vary upsample and oversample

If you use `upsample = 1` you are changing the data the least. If you want to estimate the intrinsic line width, you could use a high value of `oversample` (#4) if you can: the line will be noisier but the pixel size will not affect the fit.

- Note that the total flux in the line, i.e. the integration of the line profile with respect to the continuum, should not change with wavelength grid details.

For Nyquist-sampled data (e.g. SEDs), the `oversample` parameter is set to 2 by the pipeline to avoid low spectral sampling in the middle of the spectral range, which happens when the bins are too small (i.e. `oversample` is too large). While low sampling-density bins is normal at the edges of a spectral range, it is a problem if they occur in the middle of the range. Hence, and in particular for ranges obtained with "Nyquist" sampling, one should use `oversample=2` to be able to properly sample the spectral lines. A smaller value will smooth the spectrum, so can only be used to analyse broader features. A larger value will result in an increased number of empty spectral bins (NaNs). With the task `plotCubesStddev` (see [Section 10.2](#)) with the `plotLowExp` parameter set to 1 (the default), you can see the low sampling-density bins and check that they occur only at the edges of the range.

If you set `upsample` high, you will ensure that you include many datapoints in the new bins, and the spectra will be somewhat smoother, but since many of the same datapoints will have made their way into neighbouring bins, the datapoints of the spectra of your new cube will not be independent, and this will affect the noise statistics.

For SED-mode spectra, after rebinning the cubes it is possible that **spikes** will occur at the edges of the spectra. These spikes typically occur when a very small number of contributors are present in rebinned spectra that use the default bin size defined by an `oversample = 2`, i.e. there is not enough data in the new wavelength bins at the very ends of the spectral ranges (the ends of the spectral ranges are *always* less well covered than the middle.) When the number of contributors is small the outlier rejection algorithm (`specFlagOutliers`) may not work very well, and so bad points there can propagate through to the final rebinned spectrum. Simply ignore the spikes and only consider the central part of the spectrum where the sampling is sufficient to produce good statistics



Note

For data obtained in Nyquist sampling mode before OD 305, it is normal to find empty bins (NaNs in the rebinned cube) even when rebinned with `oversample=2`. This is intrinsic to these data, and there is no way in the pipeline to correct for this.

7.2.3. Noise properties

With `upsample=1` and `oversample=2`, which Nyquist samples the native spectral resolution, there is no correlation between datapoints, since each new bin has a unique set of input datapoints (i.e. each X bins in the *PacsCubes* feed into only one bin in the *PacsRebinnedCubes*). *But if upsample is larger than 1, then neighboring bins will share datapoints, and this changes the noise properties that you measure from the spectra.* As stated above, it is not the actual noise that changes when you change `upsample` and `oversample`, but the measured noise.

If you want to measure the "native" noise in your spectra of any Level 2 cube, you will need to rerun the pipeline from "wavelengthGrid" onwards., setting the parameters appropriately. Especially if your spectra are faint, or you want to be sure of the shape of your lines, you should test the result of different wavelength grids, by creating various grids with the task `wavelengthGrid`, and run the steps of the pipeline script from there to "specWaveRebin" where the rebinned cubes are created. Then compare the spectra of the resulting cubes (e.g. with the Spectrum Explorer).

7.2.4. The dispersion

Here we present plots of the bin size of the scaled wavelength grids for the different filters, computed with the parameter `upsample=1` and `oversample=2` in the task "wavelengthGrid"—the bin sizes depend on these two parameters (see [Chapter 5](#)) but the same shape as plotted here will always be present.

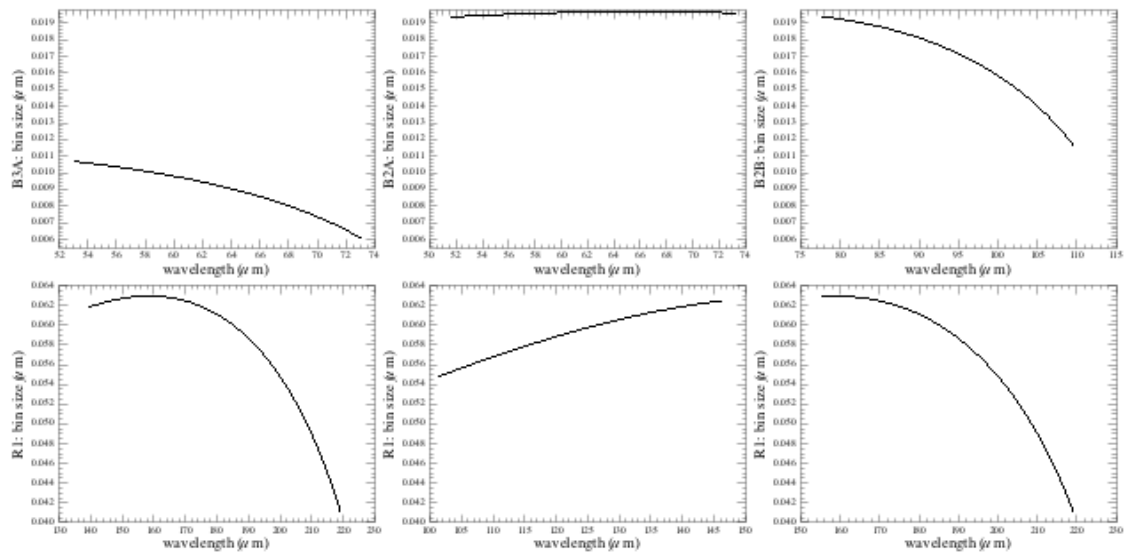


Figure 7.4. Dispersion as a function of wavelength for the various spectroscopy filters for a `wavelengthGrid` created with `upsample=1` and `oversample=2`.

Note: to create a similar plot to the one above, this is the key phrase:

```
wave = cube.getWave()
PlotXY(wave, wave[:-1]-wave[1:])
```

7.3. Cubes with an equidistant wavelength grid

Equidistant cubes are created with a task that spectrally resamples the flux array of an input cube (and its other data arrays) onto a regular and equidistant wavelength grid. Why do we do this? The wavelength grid created by the pipeline task "wavelengthGrid" and carried from the rebinned cubes to the mosaic

cubes (drizzled, projected, and interpolated) is regular but it is not equidistant: the bin sizes scale with the resolution, which in turn scales with the wavelength. This was done so that the spectral sampling was at least Nyquist at all wavelengths in an observation: this is especially important for the long wavelength range observations, although less necessary for short ranges and line scans. Because the mosaic cubes created by the pipeline do not have an equidistant wavelength grid, the third dimension of the WCS is not defined with a set of single values (cdelt3, crpix3, crval3, and ctype3) but are left undefined. Some software has trouble handling this and therefore cannot read PACS cubes directly. For this reason PACS created the "equidistant cubes", which are the three mosaic cubes interpolated on to an equidistant wavelength grid, and these are provided as standalone browse products (see the [PPE](#) in *PACS Products Explained*). These cubes do have the third dimension of the WCS filled in. The equidistant wavelength grid used is based on that of the input cube, but it has smaller bin sizes. In this section we show you how to create equidistant cubes and show plots comparing the equidistant spectra to the standard spectra for a few examples.

7.3.1. Create your own equidistant cubes

Here we show how to create equidistant cubes and compare them to the original cubes. This example follows that given in a HIPE scripts in *Scripts#PACS useful scripts#Spectroscopy: Re-create the standalone browse products*. We will call the wavelength grid from the first the "equidistant grid" and that from the second the "scaled grid".

Starting from an observation (or the example obsid can be used), extract the Level 2 *PacsCubes* (HP-S3DB), *PacsRebinnedCubes* (HPS3DRB), and then the cube you want to make equidistant, i.e. the projected, drizzled, or interpolated cube (HPS3D[P|D|I]B); noting that for this example observation there are no interpolated cubes):

```
obsid=1342187206
obs=getObservation(obsid,useHsa=1)

slicedCubes = obs.level2.blue.cube.product
slicedRebinnedCubes = obs.level2.blue.rcube.product

slicedProjectedCubes = obs.level2.blue.pcube.product
# Or, to get the drizzled or interpolated cube context
slicedDrizzledCubes = obs.refs["level2"].product.refs["HPS3DDB"].product
slicedInterpolatedCubes = obs.refs["level2"].product.refs["HPS3DIB"].product
```

To create the equidistant cubes it is necessary to create the wavelength grid using the same task that the original (non-equidistant) wavelength grid of the cubes was created with, some parameters being exactly the same and others being different. This is necessary so that the spectra of the equidistant cube are as much as possible the same as those of the input cube. The parameters for the task `wavelengthGrid` that need to be the same when creating the equidistant grid are: `upsample`, `oversample`, and `standardGrid`. The former two we get directly from "slicedRebinnedCubes" as shown below, the latter parameter is anyway always True when used in the pipeline (and it is the default value). To learn what these parameters mean, we refer you to the [URM](#) entry for "wavelengthGrid" and [Section 7.2](#).

```
# Query on the meta data to find the values of oversample
# and upsample used in creation of their the wavelength grid
slice = 0 # any slice will do
oversample = slicedRebinnedCubes.refs[slice].product.meta["oversample"].double
upsample = slicedRebinnedCubes.refs[slice].product.meta["upsample"].double

# Set the size of the new bins
frac = 0.5 # let's say you want bins of 50% of the smallest in the input cubes

# Get the calibration tree
calTree = getCalTree()
```

Create the wavelength grid, and then resample the projected (or drizzled, or interpolated) cube on that grid, e.g.

```
waveGrid = wavelengthGrid(slicedCubes, oversample=oversample, upsample=upsample,\
```

```
calTree = calTree, regularGrid = True, fracMinBinSize = frac)
slicedProjectedCubes_eq = specRegridWavelength(slicedProjectedCubes, waveGrid)
```

The important parameters in the task `wavelengthGrid` to be used when creating the equidistant grid are the following:

- **fracMinBinSize**: this is the fraction by which the smallest bin size of the scaled grid is multiplied to set the bin size of the equidistant grid.
- **fixedBinSizes**: to specify fixed bin sizes, enter these with this parameter; since there may be more than one cube in the input sliced cube product, you must specify a line id for each slice using a dictionary, e.g.

```
fixedBinSizes={2: 0.02, 3: 0.03}
```

sets a bin size of 0.02 micron for `lineId=2` and 0.03 micron for `lineId=3`. The `lineId` values can be obtained by running the task: `slicedSummary(sliced cube product)`.

- **oversample, upsample**: as stated above, their values should be the same as that with which the wavelength grid of the projected (drizzled, interpolated) cubes were created with.
- **regularGrid**: should be set to `True` to have a grid with equidistant bins. If `regularGrid` is `True` but neither `fracMinBinSize` nor `fixedBinSizes` is specified, then the bin size of the equidistant grid is the minimum bin size of the scaled grid (i.e. `fracMinBinSize = 1`).
- *standardGrid*: set to `True` to use a scaled grid with standardised starting and ending wavelength points; this parameter is not important for creating a regular grid but the name is similar—do not confuse this with "regularGrid"

Do not be tempted to use the equidistant grid in the pipeline when creating the rebinned cubes, rather than the scaled grid that is the default. Compared to re-sampling cubes at the end of the pipeline, the result is noticeably inferior for all except the narrowest wavelength range observations.

Note that it is not a good idea to apply the task `specRegridWavelength` to projected cubes created for single pointing observation: these cubes have very many very small spaxels (0.5"), and HIPE can run out of "Java heap space".

7.3.2. Comparison of equidistant with standard cube spectra

Here we show a comparison between a spectrum taken from a projected cube and the spectra from its equidistant counterpart cubes for different values of `fracMinBinSize`. In the black curve is the spectrum of the central spaxel from the projected cube with a standard wavelength grid (`oversample=2`, `upsample=4`), and in coloured dots are the datapoints from the equidistant version of that same cube with a wavelength grid created with `fracMinBinSize` of 0.5, 0.35, and 0.25. The top spectrum is an entire range and that below a zoom in on the line. It is clear that all of the equidistant spectra match the shape of the standard spectrum very well.

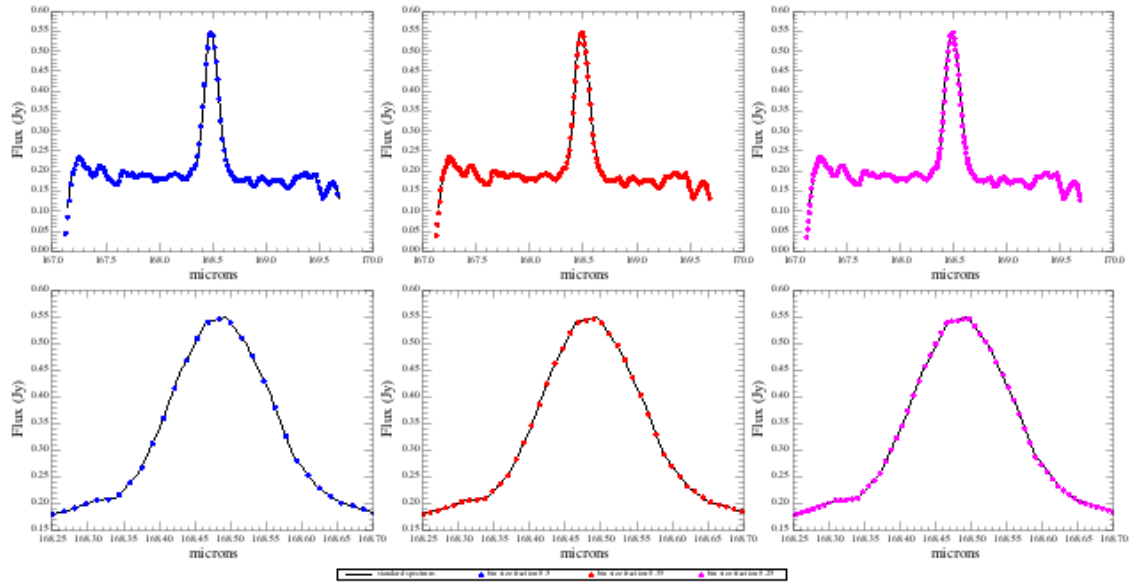


Figure 7.5. Comparison of the spectrum with a scaled grid and the same data interpolated onto different equidistant grids ($\text{fracMinBinSize} = 0.5$: blue; 0.35 : red; 0.25 : magenta): a very short rangeScan

Next we show the same plots but for a full SED, using the standard cube (created with $\text{oversample}=2$, $\text{upsample}=2$) and the equidistant cubes with a range of fracMinBinSize values. The plots show parts of the blue, central, and red ranges of the blue camera and red camera data. The largest bin size for the scaled grid for this observation is in the very blue for the blue camera (B2B) and also for the red camera (R1).

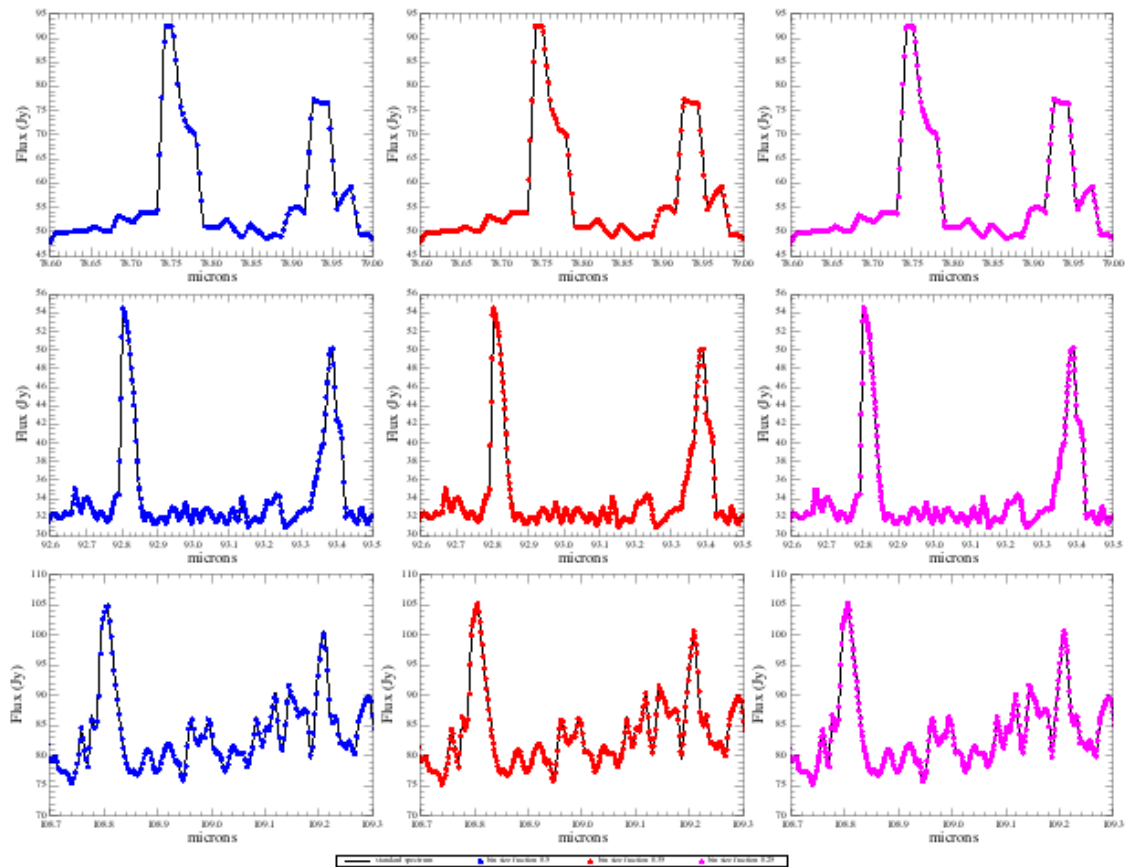


Figure 7.6. Comparison of the spectrum with a scaled grid and the same data interpolated onto different equidistant grids ($\text{fracMinBinSize} = 0.5$: blue; 0.35 : red; 0.25 : magenta): blue part of an SED

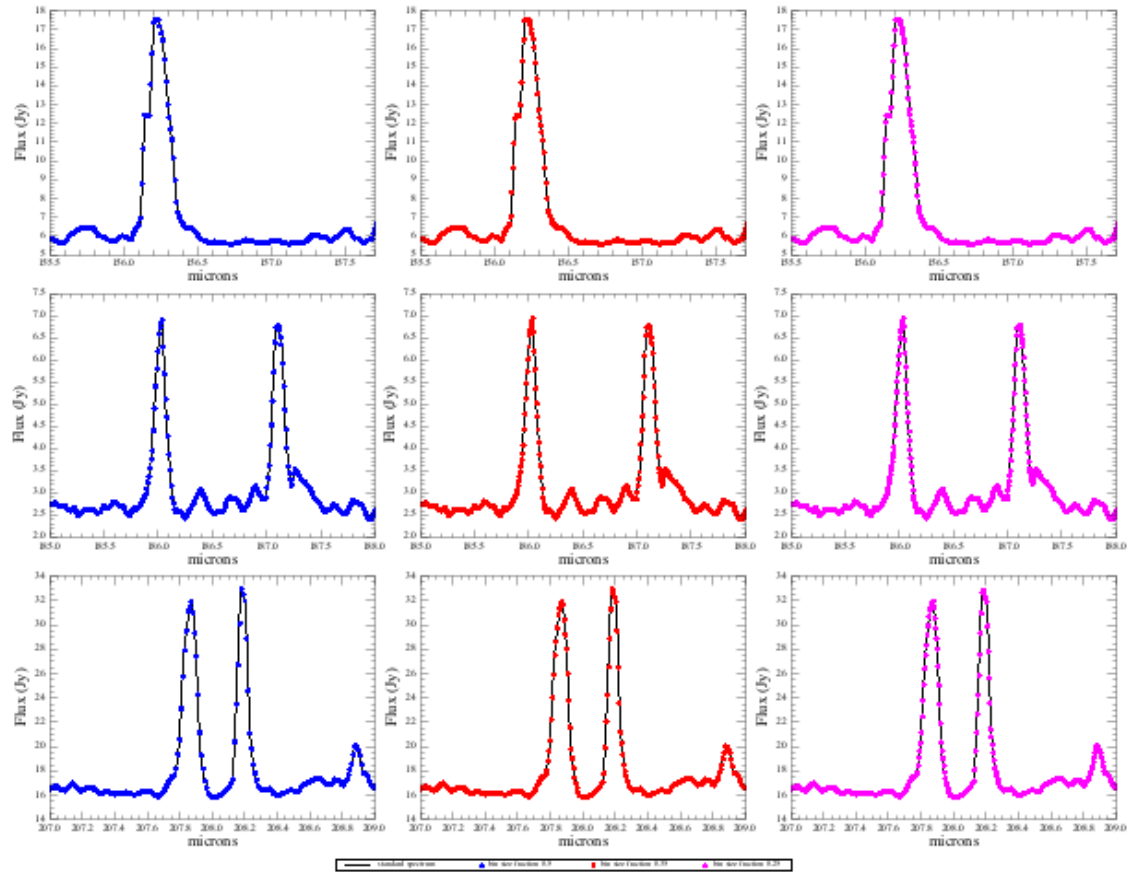


Figure 7.7. Comparison of the spectrum with a scaled grid and the same data interpolated onto different equidistant grids (fracMinBinSize = 0.5: blue; 0.35: red; 0.25: magenta): red part of an SED

It is clear from these plots that even for this SED, the equidistant grid follows the standard grid data very well.

A note of warning, however, about having many, small bins: this can make the data appear to have a resolution greater than it actually has. It must be remembered that these equidistant-grid data are an *interpolation* of the original data and have a sampling that implies a greater resolution than the data actually have. It is for this reason that the Meta datum "fracMinBinSize" is included with the cubes, so you can work out by how much this is.

7.4. The spectral flatfielding

The spectral flatfielding stage of the pipeline is provided to improve the SNR and the shape of the spectra. The tasks to do this for line scans and longer rangeScans are slightly different, but the general process is the same.

Spectral flatfielding is done to the *Frames* (rangeScan) or *PacsCubes* (line scan) before they are spectrally resampled to create the Level 2 rebinned cubes. The reason this is necessary is because the spectral response of each of the 16 pixels that feed each module of the detector (the 25 modules = the 25 spaxels) is slightly different; there are differences also in the natural level of spectra obtained when moving up the grating compared to those obtained when moving down the grating. In addition, repeats on the spectral range may also have slightly different absolute levels, sometimes because of the effect of transients. If these variations in the spectral level are not corrected before the spectra are averaged together along the wavelength grid created by wavelengthGrid (see [Section 7.2](#)) to create the rebinned cubes, then the SNR will suffer. The flatfielding tasks "tidy up" these variations while maintaining the mean spectrum of each module: by moving spectra too high down to the mean level, and moving those too low up to the mean level.

This tidying up is done by first creating a mean spectrum (excluding bad-masked data) and then splitting the data in each module into populations, and comparing the population-spectra to the mean. The populations chosen are those expected to differ in signal level: each grating scan direction (up and down) is a separate population, each of the 16 pixels is a different population, the signals taken at the sets of different grating and chopper positions encountered during an observation are different populations. So, for example, signals take while the chopper is moving from one position to the next (but which anyway are flagged as bad) are one population, signals take while the chopper is stable and in the off position are another, signals taken while the chopper is stable and in the on-position are yet a third, and so on.

7.4.1. Line scan flatfielding

The flatfielding is a multi-step process for line scan AOTs, and this task is also recommended for the shorter rangeScan AOTs, of up to ~5 microns. The flatfielding is performed in a few steps: (i) outliers are masked out, (ii) spectral lines are identified, so they can be ignored when (iii) the mean continuum level of each pixel is determined, and (iv) each population is normalised to the overall mean of the spaxel they belong to (using all the good data to compute the mean), and finally (iv) then masks and intermediate results are cleaned up.

The task that does the actual flatfielding, after the spectral lines have been identified and masked out, is called `specFlatFieldLine`, and its [URM](#) entry can be consulted. This task works on each *PacsCubes* of the input *SlicedPacsCubes* product. For each cube it works spaxel-by-spaxel, extracting out the individual spectra and, ignoring the masked-out data and spectral lines, it computes a median reference spectrum. The spectra from each spaxel are then split into populations and the spectra from each population are compared to the reference spectrum. Then each population spectrum is shifted by the ration between its median value and the median value of the reference spectrum. This is a multiplicative correction if you set `scaling` to 1, as is the default, and recommended, value. (Setting to 0 you would apply an additive correction.) If you `verbose=1`, then this task will produce a plot showing, for the central spaxel of each slice, the before and after spectrum taken from the *PacsCubes* (plotted as dots) and a rebinned version of each (plotted as a line and so it is much easier to see the spectral shape and lines). See the figure below. As variations in task you can ask to fit the continuum with a 1st order polynomial, limit the wavelength range to be flatfielded (e.g. if you want to focus the flatfielding on a certain range or you want to exclude unwanted ranges from the process), and set a few other parameters (as detailed in its [URM](#) entry).



Note

What is a "population"? PACS data were taken continuously, while the grating and chopper were moving and while the instrument was moving on the sky (e.g. for rasters). Hence there is a population of data that were taken during movements and which will have been masked as bad, and a population that are good data. There are additional populations of datapoints that come from different parts of the instrument. The PACS signal-detector was constructed from 25 rows of 16 pixels: each pixel was in fact a separate detector with its own response, and each row of 25 corresponds to a single spaxel and hence to a single position on the sky. The signal stream from each pixel for each module is also a separate population. Finally, the response of each pixel differs if you are moving along the grating towards increasing or towards decreasing wavelength, and this makes additional populations.

The parameter `slopeInContinuum=1` is for spectra with lines on a continuum with a slope. The continuum of the reference spectrum is fit with a first order polynomial. The individual (population) spectra are then compared to this fit. For every spectrum, the scaling is a single value which is the ratio between the reference spectrum interpolated to the average wavelength covered by that pixel, and the median level of the spectrum in that pixel. The correction is therefore computed at a slightly different wavelength for each pixel, since the wavelength coverage is slightly different for each pixel. While this should allow for a better result, note that if the slope is slightly different for each pixel, then the flatfielding can become slightly pixel-dependent.

The scaling applied is a multiplicative correction if you set `scaling` to 1 (default). Setting to 0 you would apply an additive correction.

For sources with very faint lines and those with 0 continuum levels, pay attention to the result of the flatfielding, since with insignificant levels of continuum flux will make the results of the flatfielding task uncertain. If there is no difference in the SNR before and after flatfielding, you can even skip this pipeline step.

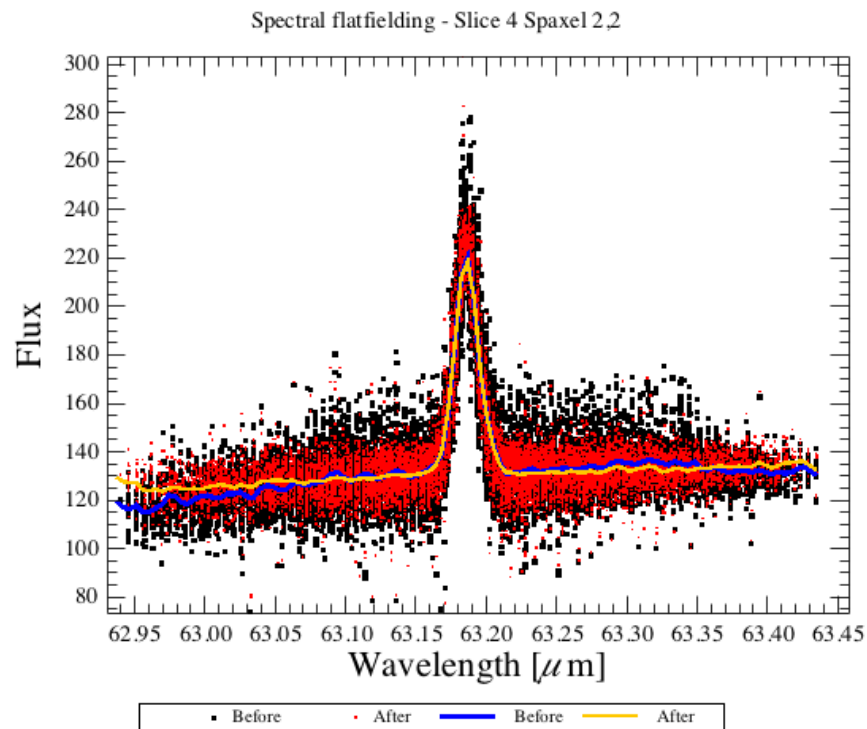


Figure 7.8. Before and after flatfielding: blue curve and black dots are from before; yellow curve and red dots are from after

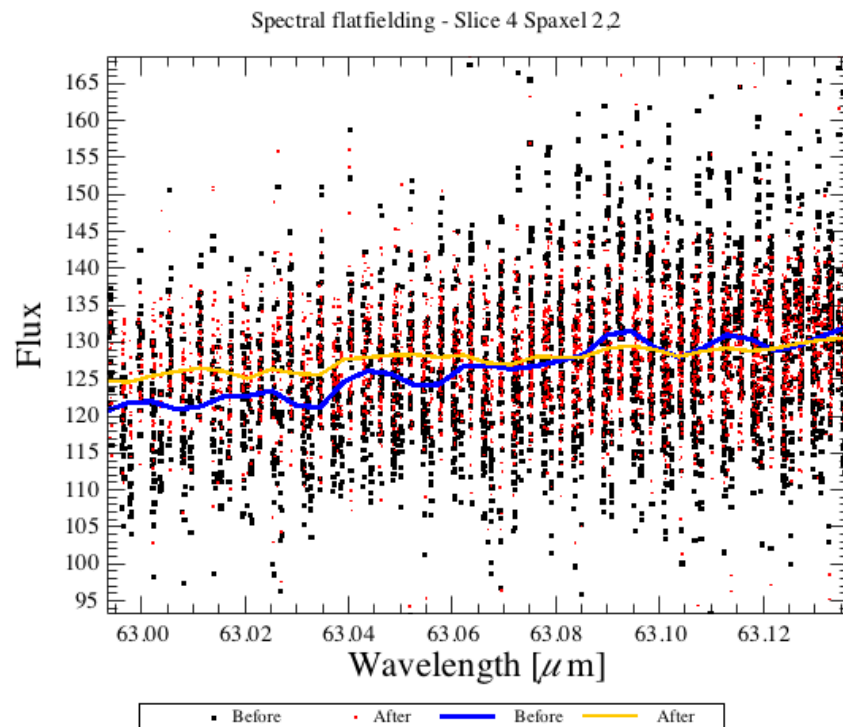


Figure 7.9. Before and after flatfielding: zoom

7.4.2. rangeScan flatfielding

Flatfielding for longer rangeScans can have a noticeably positive effect on the SNR of the spectra. The figures at the end of this section demonstrate what sort of improvement you can expect, and also why it is a good idea to experiment with the parameters of the task.

The flatfielding process for rangeScan AOTs works on the *Frames* products. It computes a reference spectrum created from all the non-masked datapoints for each spaxel; then computes the continuum fit to this reference spectrum and a continuum fit to the spectrum for each individual population in each module; and finally shifts the individual fits to that to the reference spectrum, to scale the populations together.

- For each module independently, the task takes the spectrum from each population of each of the 16 pixels and does a spline fitting or a polynomial fitting to the continuum. The task does a running filter sigma-clipping before fitting or calculating the median. This clipping is quite severe so it removes not only noise but also spectral features (but the clipping is not propagated to the output product).

The possibility to fit splines instead of a polynomial was added in HIPE 14, and it is the preferred option (although for shorter ranges, polynomial fitting and spline fitting produce very similar results). With the introduction of this option, flatfielding for longer rangeScans (and especially SEDs) in the SPG became possible. Splines deal better with the strong curves and slopes that these longer ranges can have in them. The details of the spline fitting can be found in the [URM](#) entry for "specFlatFieldRange".

- For each module independently, and using the data from all its pixels and all its populations, it also makes a continuum fit the median spectrum.
- For each module, it divides the fit to each population of each pixel by the module fit. It then uses these ratios to shift the individual populations to the mean fit. The effect of this is to reduce the scatter in the entirety of the data in each module, and then reduces the noise in the spaxel when the module data are turned into rebinned cube spaxel data.
- If the task cannot do a flatfielding, which usually only happens if there is not enough data to do so (normally because almost all of the spectral range turns out to be masked as OUTFBAND), then the task will not flatfield the problem slices: needless to say (and probably whether flatfielded or not) such slices contain only useless data.

As you can see from the figure below, running the flatfielding improves the appearance of "ringing/fringing" that appears in non-flatfielded spectra.

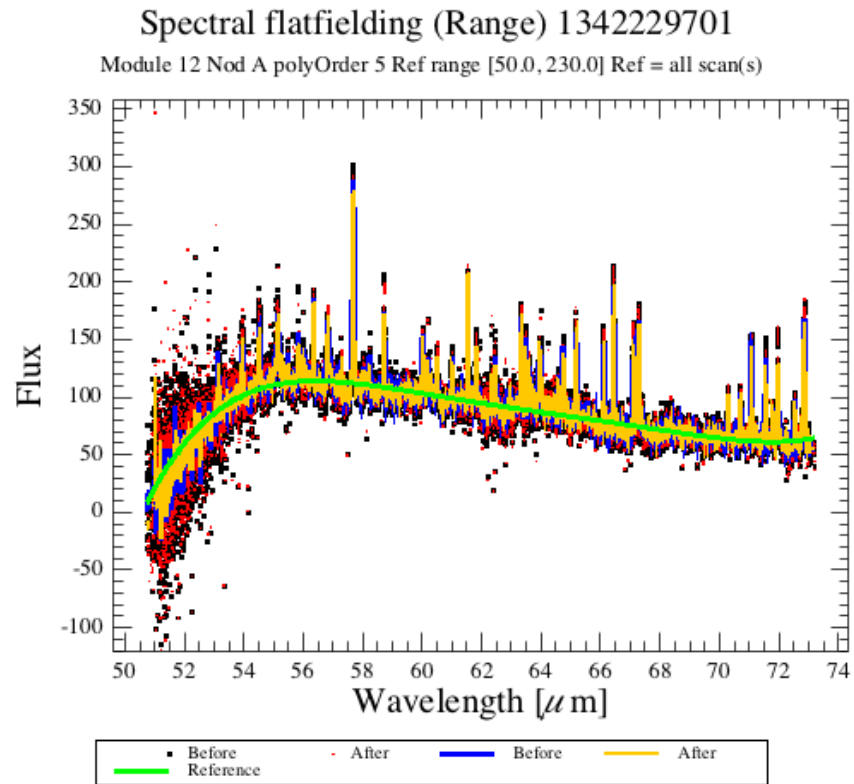


Figure 7.10. Before and after flatfielding (polynomial fitting): blue curve and black dots are from before; yellow curve and red dots are from after

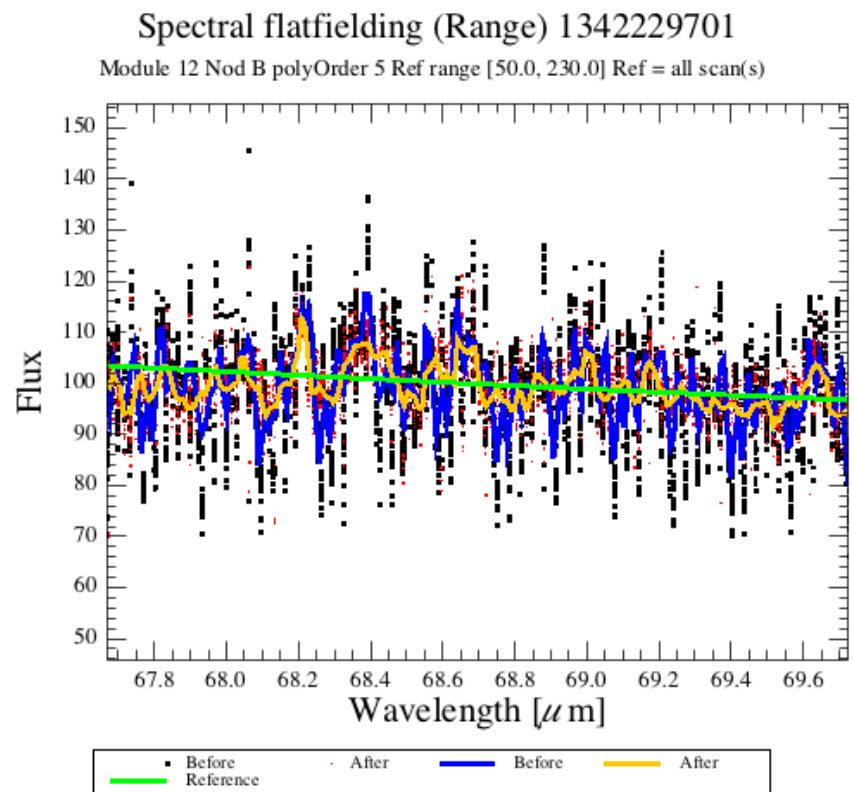


Figure 7.11. Before and after flatfielding: zoom

7.4.2.1. Things to watch out for

- **For short range observations**, less than about 5 microns, we recommend using the line scan flatfielding, i.e. running the line scan pipeline script, as this can produce superior results.
- **For SEDs**, `specFlatFieldRange` will deliver better results if you exclude the leak regions from your data during the flatfielding (`excludeLeaks=True`). These leak regions often contain large and spectrally rapid variations of the continuum flux, and this can render optimal fitting (especially for polynomials) of a reference spectrum more difficult. See the Observer's Manual to learn about these leak regions (herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb?template=viewprint).
- If you do want to include the red leak region (at wavelengths $>190\ \mu\text{m}$), see [Section 5.4](#) (`chopNod`) or [Section 6.4](#) (`unchopped`) for instructions. In this case, the spline fitting option of the flatfielding task will produce noticeably better results than the polynomial flatfielding.
- **Leak regions** are by default not included in the computation of the flatfielding (`excludeLeaks=True`). However, the flatfielding correction is applied to all the wavelengths in the spectra, and so leak regions of the spectrum are adjusted by a correction that does not apply to them. Therefore, these spectral ranges are given the mask NOTFFED. These data should be excluded (i.e. the mask should be activated) when the *PacsCubes* are turned into *PacsRebinnedCubes* by `specWaveRebin`, to avoid having very strange-looking spectra in the final cubes.
- **The order of the polynomial fitted** can have a strong effect on the final spectra when fitting to longer, bendy ranges. If you use polynomial fitting instead of the splines, it is strongly recommended that you check the final rebinned spectra, and if you are not happy with the flatfielding, start the whole process again (from beginning the flatfielding to creating the final rebinned cubes of the pipeline: `slicedFinalCubes` or `slicedDiffCubes`). One particular effect of having too high an order that S-curves can be induced in the spectra of the non-central (i.e. fainter) spaxels, as is demonstrated by the figure below. If you see this, it is almost always improved by reducing the order of the fit, or by using the spline fitting.

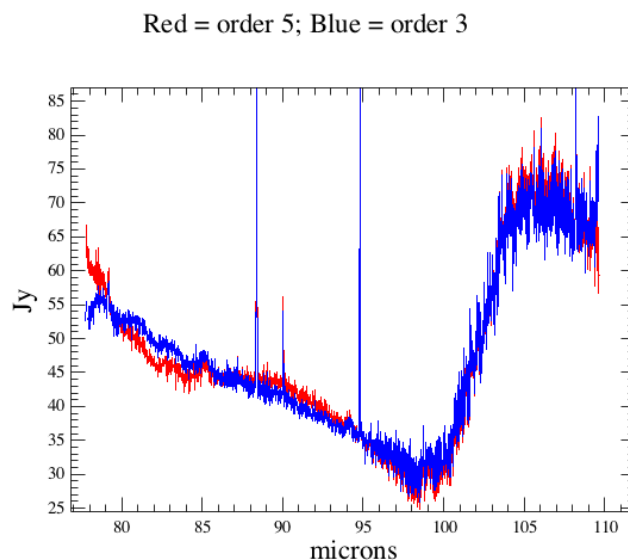


Figure 7.12. An example of an "S-curve" (red spectrum) that is a signature of flatfielding with too-high an order for the fitting (in `rangeScans`)

- A parameter added in HIPE 11 is `referenceScan`. **For data with very many grating scans such that the observation took several hours**, response drifts (with time) may have affected the detector during your observation. You can therefore chose to use as the reference spectrum either "all" grating scans (i.e. data from the entire sequence within the nod position), or only the "first" (that closest to when the calibration block was observed), or the "last" (that for which it is more likely

the response has stopped drifting). This parameter has been provided as a test, i.e. we have no recommendation as to which to use, you can test it out yourself on your data.

- **For sources with near-0 continuum levels**, pay attention to the result of the flatfielding, since with insignificant levels of continuum flux will make the results of the flatfielding task uncertain. If there is no differences in the SNR before and after flatfielding, you can even skip this pipeline step.

7.4.2.2. How to test the flatfielding

To test the flatfielding, you should save a copy of the slicedFrames or SlicedCubes before you do the flatfielding, so you can begin again from that product,

```
slicedFrames_b4 = slicedFrames.copy()

# or use this bit of the pipeline script
# (outputDir is defined earlier in the pipeline script)
name=nameBasis+"_slicedFrames_B4FF"
try:
    saveSlicedCopy(slicedFrames,name, poolLocation=outputDir)
except:
    print "Exception raised: ",sys.exc_info()
print "You may have to remove: ", outputDir+'/'+name
```

To then repeat pipeline steps, it is enough to

```
slicedFrames = slicedFrames_b4.copy()

# or
name=nameBasis+"_slicedFrames_B4FF"
slicedFrames = readSliced(name, poolLocation=outputDir)
```

and then continue with the pipeline, from the flatfielding until the task specAddNodCubes or specSubtractOffPosition.

There is a task to plot the spectra immediately before and after flatfielding: plotPixel, see [Section 10.2](#). This can be useful to see effect of the flatfielding on a single spaxel/module of a single *PacsCube/Frames* slice. But it is easier to judge the improvement of the flatfielding by looking at the final rebinned cubes of the pipeline ("slicedDiffCubes" or "slicedFinalCubes"). Here we include here a short script with which you can create a 5x5 plot of each cube of the final rebinned cubes to compare them. In this way you can see the global effect of different flatfielding options (or the effect of changing anything else in the pipeline).

First, to create a 5x5 plot from a *slicedFinalCubes* near the end of the pipeline, for any slice of your *slicedFinalCubes*:

```
slice=1
p12 = plotCube5x5(slicedFinalCubes.get(slice))
```

Its [URM](#) entry shows you how to limit the range plotted.

To over-plot spectra on a 5x5 panel window (i.e. to show the spectra of all 25 spaxels) requires a bit of scripting. We assume here you have two "slicedFinalCubes" both loaded in HIPE, and you want to over-plot the same slice from both:

```
p=PlotXY(titleText="two ffs compared")
slice=0
cube1=slicedFinalCubes1.get(slice)
cube2=slicedFinalCubes2.get(slice)
cnt=0
for x in range(5):
    for y in range(5):
        wv1=cube1.getWave(x,y)
        flx1=cube1.getFlux(x,y)
        wv2=cube2.getWave(x,y)
        flx2=cube2.getFlux(x,y)
        p.addLayer(LayerXY(wv1,flx1,line=1,\
```



```

        color=java.awt.Color.red),x,y)
    p.addLayer(LayerXY(wve2,flx2,line=1,\
        color=java.awt.Color.blue),x,y)
    p[cnt].xaxis.titleText="microns"
    p[cnt].yaxis.titleText="Jy"
    p[cnt+1].xaxis.titleText="microns"
    p[cnt+1].yaxis.titleText="Jy"
    print cnt
    cnt+=2
p.saveAsPNG("/Users/me/file.png")

```

7.5. Advice on background subtraction for the unchopped modes

The background subtraction for the unchopped modes is done by reducing on-source data and the off-source data until the rebinned cubes stage, separately. For unchopped line AOTs, the on-source and off-source data are contained within the same observation. For unchopped rangeScan AOTs, they are separate observations.

How to check for contamination in the background spectrum is explained in [Section 10.9](#). Here we give advice about using the pipeline task "specSubtractOffPosition". This task is used to subtract the off-source cubes from the on-source cubes for line and rangeScans. The algorithm for choosing which background spectrum to subtract is set by the parameter `algorithm`, and the choices are: the off-source closest in time to each on-source slice/cube; the mean of all off-source cubes; the time-weighted interpolation of the off-source cubes. The interpolation algorithm appears to work well, especially for mapping observations.

Playing with these parameter is worth doing for observations where there is more than one off-source pointing. This is particularly important because an off-source observation taken at the beginning of the observation, and hence right after the calibration block, can often suffer from a "long-term transient" that artificially increases its flux. This can result in very negative fluxes in the background-subtracted cubes. (This is why the continuum level in unchopped observations has a greater uncertainty than for chopNod observations.) This can be somewhat compensated for by trying a different value for `algorithm`, and you can also try the pipeline scripts called "...with transient correction", new to HIPE 13 (see [Chapter 6](#)).

To compare the results of different values for `algorithm`, you should save your slicedRebinnedCubes before you run the task `specSubtractOffPosition`

```

slicedRebinnedCubes_b4 = slicedRebinnedCubes.copy()

# or
name=nameBasis+"_slicedRebinnedCubes_B4offsub"
try:
    saveSlicedCopy(slicedRebinnedCubes,name, poolLocation=outputDir)
except:
    print "Exception raised: ",sys.exc_info()
print "You may have to remove: ", outputDir+'/'+name

```

the next use of `specSubtractOffPosition` should then work from this copy

```

slicedRebinnedCubes = slicedRebinnedCubes_b4.copy()

# or
name=nameBasis+"_slicedFrames_B4offsub"
slicedRebinnedCubes = readSliced(name, poolLocation=outputDir)

```

You can create the cubes containing the background spectra subtracted from each on-source cube in `slicedRebinnedCubes` by setting the parameter `withOffCubes` to `True`, e.g.

```

slicedFinalCubes = specSubtractOffPosition(allFinalCubes,copy=True,\
    algorithm="CLOSEST", withOffCubes=True)
# other choices: NEAREST, INTERPOLATED

```

In this case the output, "slicedFinalCubes", will be a *ListContext* with twice as many cubes in it: for each background-subtracted on-source cube in the *ListContext*, there is the background cube that was subtracted from the input on-source input cube to create it. You can tell which cubes are which using,

```
slicedSummary(slicedFinalCubes)
```

And then to grab any two cubes, extract the appropriate cube "slice", e.g. for slices 0 and 1

```
rebinnedCube_backsub = slicedFinalCubes.get(0)
rebinnedCube_back    = slicedFinalCubes.get(1)
```

You can use the Spectrum Explorer ([Section 10.5](#); [DAG chap. 6](#)), or the pipeline plotting tasks ([Section 10.2](#)) to compare the cubes.

7.6. Glitches, outliers, and saturation

Why read this section? PACS has two pipeline tasks to look for glitches: one runs at Level 0.5 and works on the time-line spectra and the other is found at Level 1 and works on the spectra organised vs. wavelength. There is one task that looks for saturation, at Level 0.5. If you suspect you have very bright and/or saturated lines in your spectra, you may want to divert a little from the pipeline scripts with respect to which masks are or are not included when the Level 2 cubes are built from the Level 1 cubes. This we explain here.

7.6.1. Saturation

Saturation is detected in the Level 0 to 0.5 stage of the pipeline with the task **specFlagSaturation**. As is explained in its [URM](#) entry, two masks are created: SATURATION and RAWSATURATION. SATURATION is a flag for any datapoint in the input *Frames* found to exceed a hard-wired saturation limit. RAWSATURATION is set similarly, but it is a more accurate detection of saturation as it works on the raw data (HPSRAW[R|B] in the *ObservationContext* or "slicedRawRamp" in the pipeline script). However, this raw-data product contains data only for the most responsive pixel of the central module of the PACS detector (spaxel 2,2=module 12, pixel 5 for the red camera and pixel 10 for the blue camera). If rawsaturation is found in this pixel, then all datapoints in the *Frames* of "slicedFrames" have a flag set to bad (True=1) in the RAWSATURATION mask, but only for the same time interval for which the saturated data occurs in the raw pixel.

However, it is worth checking this mask: just because the most responsible pixel is saturated for a certain time-interval does not necessarily mean that *all* other pixels will be also, since the signal level and the response of the other pixels of the detector are probably lower. If you do not believe that the RAWSATURATION mask carries useful information, then you should remember *to not activate* it during the rest of the running of the pipeline script. Keep an eye on the "activateMasks" tasks and exclude from them this mask, or run "activateMasks" a second time, to specifically deactivate this mask. See the [URM](#) entry for activateMasks to learn how to do this, or the examples in the pipeline scripts.

In the pipeline scripts, the *PacsRebinnedCubes* are created by averaging together the flux datapoints of the *PacsCubes* along a defined wavelength grid, by the task specWaveRebin. Thus, each wavelength bin of the *PacsRebinnedCubes* is the average of all the not-excluded datapoints of the *PacsCubes*. So, if a datapoint is flagged as bad for any active mask, e.g. SATURATION, then the average value in that *PacsRebinnedCube* bin does not include the saturated datapoint value. If, when running specWaveRebin, you however *do not* activate these saturation masks, then saturated data *will* be included in the *PacsRebinnedCubes*. In either case, a value is added to the "flag" dataset of the *PacsRebinnedCubes* to indicate that in this wavelength bin whether the saturation mask was, or was not, activated when the cube was created.

If you exclude saturated datapoints (as is the default of all the PACS pipeline scripts), any regions in the final spectra, that are saturated for all the repeats on that wavelength range, of the Level 2 cubes will have gaps (i.e. the data there will be NaN).

7.6.2. Glitches

In the beginning of the Level 0.5 part of the pipeline, the task `specFlagGlitchFramesQTest` searches for glitches (outliers, cosmic rays, ...), in the 16x25 pixels of time-line data, for each of your *Frames* slices (held as "slicedFrames"), creating a mask called GLITCH in the process. It uses the Q statistical test to do this: to learn what is done within the task you should consult its [URM](#) entry. There are parameters you could play with, but note that these have been much tested and the default parameter settings are good for practically all cases.

In the new unchopped pipeline scripts, a similar but less aggressive glitch detection task, `specFlagGlitchFramesMAD`, is used: this uses the Mean Absolute Deviation method instead of the Q test.

Then, in the Level 1 to 2 part of the pipeline, the task `specFlagOutliers` searches for glitches in the spectra, but along the wavelength domain; hence it is complimentary to the GLITCHes. It uses sigma clipping and creates a mask called OUTLIERS. This task works by first rebinning the data according to the specified wavelength grid, and then looks for all outliers by among the datapoints that fall in the new grid, going bin by bin, and flagging them as bad in its mask. You can see its [URM](#) entry to learn more (noting that changing the number of iterations does not seem to have a large effect, the sigma value is the most useful parameter to play with).

7.6.3. The interaction of saturation and glitch masks for targets with bright spectral lines

In the pipeline scripts, when building your Level 2 cubes it is assumed that you want to exclude the OUTLIER- and the GLITCH-masked data. There are a few cases, however, where you may not want to incorporate the GLITCH mask, or at least not all of it. These cases are where you have saturated or very bright lines. For these you may want to deviate from the default pipeline script a bit.

We have found that the Level 0-0.5 pipeline task `specFlagGlitchFramesQTest` can identify the peaks of bright (almost saturated) lines incorrectly as glitches. This is unimportant for the user until (and maybe even then not) they build the rebinned cubes with the task `specWaveRebin`. If you chose to divert from the pipeline and not exclude the masks SATURATION and RAWSATURATION (i.e. you *do* want these data to be included in your Level 2 cubes) you may find that the saturated lines *still* have NaN values at their peaks (i.e. gaps)—because these were *also* identified as GLITCHes. If you want to avoid this, then we recommend that you (i) look at all the bright spectral lines in your data before running the pipeline task `specFlagOutliers` to see if they have been masked as glitches (follow the examples given in [Section 10.4.4](#) and [Section 10.5](#)), and (ii) if they are, then do not activate the GLITCH mask before running both `specFlagOutliers` and `specWaveRebin` (in this case you would want to neither activate the saturation masks) but do activate OUTLIERS. You can check that `specFlagOutliers` along has detected all the glitches in your data (e.g. plot and compare the GLITCH and the OUTLIERS masks, following the examples in [Section 10.4](#)): on the whole we find that it does.

7.7. Spectrum errors in the final cubes

This section is about assessing the data errors of Level 2/2.5 PACS cubes, i.e. the data-point uncertainties. To learn about these calibration uncertainty for PACS spectroscopy **you should consult the Spectrometer Calibration Document** (available on the PACS calibration wiki: herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb).

Skipping straight to the conclusion: **the best way to understand the data errors of your spectra is by looking at the scatter in the continuum of your spectra.** However, do read [Section 7.2](#) for important information on the effect of the chosen wavelength grid on the apparent noise in a spectrum.

However, there are other potential sources of uncertainty that are unique to each observation. The effect of transients, for example—which are caused by a temporary change to the response of the detector after a cosmic ray hit, or when observing something faint after having observed something

bright—affect flux values and the smoothness of the spectra. The effect of telescope mispointings and pointing jitter during an observation also affect the smoothness of the spectra. The pipelines attempt to reduce the uncertainties by correcting or compensating for these effects (how this is done is explained in the pipeline chapters themselves), but it is never possible to correct for them completely. The effect of contamination in the background spectrum is also something the astronomer should be aware of, since the pipelines assume that the background position really was clear of contamination. One must consider that the telescope background spectrum can reach a few 100 Jy, and the absolute uncertainty of the determination and removal of this leads to an additional absolute error of less than +/-4Jy.

You should also appreciate that the two main methods for pipeline reducing chopNod observations—using the calibration block or using the telescope background—are different approaches to the same problem and will never produce the same results: see the advice offered in [Chapter 2](#) and [Section 3.4.2](#).

The PACS detectors are Ge:Ga photoconductors, and the raw signal we detect is a voltage. We do not receive the raw voltages from Herschel, but rather so-called "fitRamps" (a straight-line fit to the raw readouts), which have units of V/s (i.e. a low order polynomial is fit to the voltages taken over a reset interval [0.125s]). This means that we cannot establish the shot noise (photon statistics) of the incoming signal.

All elements in the detector circuit (capacitors, amplifier, sample holder, multiplexer,...) introduce noise on the measurement. One can distinguish between two types of noise:

1. *integrated noise*, which originates from the elements before the amplifier
2. *readout noise*, which originates from the elements after the amplifier

We can obtain a reasonable estimate of the noise in PACS spectra by making use of data redundancy, once we get to the stage of the pipeline where we deal with cubes. By "redundancy" we are referring to the fact that, for almost the entire wavelength grid for any observation model, any one wavelength bin on any point on the sky will contain more than one datapoint. With the 16 spectral pixels in each module(/spaxel), the multiple grating scans of the AOTs (there will have been at least two grating scans done in your observation), and the fact that the signal is constantly modulated (with moving grating scan, chopping, nodding,...), the PACS data are highly redundant, and we can get a fair idea of the scatter in the data—which can be considered an estimate of the noise—when these redundant data are folded together along a wavelength grid.

To get a fair representation of the noise one should only compare datapoints which really are redundant, i.e. which really sample the same resolution element spatially *and* spectrally. It is hence natural to introduce this particular noise dataset during the rebinning step, i.e. at the creation of the *PacsRebinnedCube* by the pipeline task `specWaveRebin`, or at the same stage when using the pipeline task `drizzle` instead. No noise or error dataset is introduced before that in the pipeline.

As the spectral rebinning of cubes works by averaging the fluxes of the input *PacsCubes* along a new wavelength grid, each wavelength point in the rebinned cube spectra will have an average and a standard deviation value. The "stddev" is divided by the square root of the number of datapoints in the bin (i.e. those not flagged as bad in the masks activated before the rebinning is done). When the *PacsRebinnedCubes* are then added together on nod by the pipeline task `specAddNodCubes` to create the final set of rebinned cubes, or when at the same stage during the `drizzle` task, the stddev dataset is propagated accordingly (see its [URM](#) entry).

We have verified that the stddev is a good measure of the error. The values in the "stddev" array of the *PacsRebinnedCubes* agree very well with the scatter directly estimated from continuum windows in the rebinned spectra. To be noted when considering this stddev array of these cubes, and hence also the propagated errors of the subsequent cubes, are:

- **If your wavelength grid has an `upsample` value >1 then your bins in the *PacsRebinnedCubes* are not independent:** for `upsample=2`, the second bin contains about half of the same datapoints as the first bin. This means that the stddev values are also not totally independent. However, the stddev for each bin really *is* the calculation of the scatter in that bin.

- The standard deviation measures only the scatter in the *PacsCubes*, it does not include sources of flux uncertainty such as those caused by pointing jitter or flatfielding, which tend to affect wavelength ranges slightly longer than a single bin.
- The flatfielding tasks apply a multiplicative correction, which means brighter lines are more strongly affected (in the absolute difference) than fainter lines.
- The task `specProject` propagates the "stddev" array of the rebinned cubes using standard error propagation, i.e. for each projected spaxel (which are smaller than the spaxels of the rebinned cubes), the error is $\text{SQRT}([\text{stddev of rebinned spaxel}] \times (\text{weight of the projection of that rebinned spaxel onto the projected output pixel})^2)$.
- The task `specInterpolate` does an interpolation of the "stddev" array of the rebinned cube, using the same algorithm that interpolates the flux array. The interpolation error (`specInterpolate` estimates the fluxes between irregularly gridded datapoints) is not included.
- The `drizzle` task computes its own errors—but these errors are also based on the scatter in the data of the input *PacsCubes* along the wavelength grid of the *SpectralSimpleCubes*. For more information on how `drizzle` handles errors, see [Section 7.9](#).
- Both methods to extract a calibrated point source spectrum (centred or not centred) propagate the stddev to create a weights!! check array ($1/\text{stddev}^{**2}$: from the central spectrum or as the quadratic sum of the central 9 spectra, depending on which output you adopt).
- As of the final SPG processing and the final HIPE pipelines, all Level 2 and 2.5 mosaic cubes and the rebinned cubes have a "weights!! check dataset, which is computed from the propagated "stddev" dataset.

7.7.1. Inspecting the StdDev and RMS arrays in the *PacsRebinnedCubes*

The task `plotCubesStddev` can be used to plot the "stddev" dataset in the *PacsRebinnedCubes* of `slicedFinalCubes` (`chopNod`) or `slicedDiffCubes` (`unchopped`) or the Level 2 HPS3DR[R|B] (direct from an *ObservationContext*).

The task `plotCubesStddev` (see also [Section 10.2](#)) plots, for a chosen spaxel: the spectrum (black), the $n\sigma \times \text{stddev}$ curve (blue), and the $n\sigma \times \text{continuum RMS}$ curve (green). In addition it prints to the Console two 1-sigma RMS values, in Jy and in W/m^2 , taken from the continuum curve from the plot in the region of highest coverage (where the spectral coverage is at its densest) and also its best value in the spectrum. It also shows a red marker for very low coverage spectral bins.

The standard deviation curve comes from the standard deviation calculated by `specWaveRebin`. Note that the stddev increases in the spectral lines because of the strong slope. *This is not the same as saying that "the error is higher" in the lines*, it is only the absolute value of the standard deviation that is higher.

The continuum RMS curve is calculated by `plotCubesStddev` itself in a different way for line scans and `rangeScans` (the Meta data indicated which type any observation is).

- **Line and short range (a few microns) scan observations:** the wavelength of highest coverage is determined (at the centre of the spectral range, where the spectral sampling is greatest); at the wavelengths with $\pm 50\%$ of that coverage value it is assumed the spectrum is continuum (i.e. that the spectral line is well within this range), and so a region around those two 50% points is used to calculate the RMS; this RMS is then extended to the rest of the spectrum with a correction for the difference in coverage at each wavelength point (i.e. the RMS is lower where the coverage is higher, and vice versa). This explains why the continuum RMS curve is lower in the spectral lines, at least those in the middle of the spectral range: the continuum RMS will be lowest where the coverage is highest.

The continuum RMS curve is computed in an automatic way, based purely on the coverage: hence it does not take into account any continuum curvature or spectral lines that happen to lie at the 50% coverage wavelength points. Use the parameter `wranges` to fine-tune this aspect of the task.

- *Longer rangeScan and SED observations:* since it is less obvious where the spectral lines will be found in these wider range spectra, and the datapoint coverage is more even across the spectrum, the task first clips out the spectral lines (by comparing the spectrum with its heavily-smoothed/filtered "baseline") and smooths the continuum (using a PACS task called `specBaselineEstimator`), then the smoothed continuum is subtracted from the spectrum, spectral lines are noted so they can be interpolated under, and it is from the residual that the RMS is calculated. The RMS curve is then plotted.

There is no correction for coverage in this version of the task, since for long ranges the coverage only drops at the very ends. However, this can cause a problem for short rangeScan observations—whether you are reducing them with the line *or* the rangeScan pipeline script—because for those the coverage does become as important as for line scan observations (it varies by as much over the spectral range). The parameter `isLineScan` can be used to force the task to behave as if it is working on a line scan: set to -1 the task uses the Meta data to determine what the observation is, set to 1 the task uses the line scan method, and set to 0 it uses the rangeScan method.



Tip

For short ranges the smoothing done by this version of `plotCubesStddev` may not work, in this case you need to change the filter used, the smoothing value, or use the line scan version instead (`isLineScan=1`).

The uncertainties indicated by this plots and values produced by `plotCubesStddev` are most reliable for flat stretches of continuum, and comparing spectral feature to the noise in the continuum is always the best way to establish the reliability of a measurement.

The standard deviation curve, and the array in the `PacsRebinnedCubes` it is taken from, gives you exactly that: the scatter in the data that created the spectra of the `PacsRebinnedCubes`. This includes all the effects of noise along the detector chain, from raw to Level 2. The continuum RMS, the shape of the continuum over an entire spectral range, includes this *and* also the sources of error that are time-dependent and so affect stretches of spectra larger than a single bin: examples are the pointing jitter (which creates "warps" in the spectrum), and long-term response drifts. The standard deviation is therefore most likely an underestimate of the true uncertainty of your data, and the continuum RMS then most likely a better estimate—for bright lines this is true, but we are still characterising this for other flux levels.

The Spectrum Explorer, which you can use to explore cubes, does not see the "stddev" or the "error" arrays, and you cannot inspect them with this viewer. But you can plot the stddev dataset of a `PacsRebinnedCube` with PlotXY following this script (zoom in on the plots to view the error bars/error spectra):

```
from java.awt import Color # for a quick way to specify colours

# Extract a cube from the SlicedPacsRebinnedCubes
# (usually called slicedFinalCubes or slicedRebinnedCubes
# in the pipeline scripts)
cube=slicedFinalCubes.get(0)
# Get the wave, flux, and stddev DoubleIcd for spaxel 2,2
wave=cube.getWave()
flux=cube.getFlux(2,2)
stddev=cube["stddev"].data[:,2,2]

# Now plot the wave and flux with PlotXY
p=PlotXY(titleText="spaxel 2 2")
p.addLayer(LayerXY(wave,flux,line=1))
# Now add stddev "bars"
p.errorY=[stddev,stddev]

# Or, plot the flux +/- stddev
```



```
p=PlotXY(titleText="spaxel 2 2")
p.addLayer(LayerXY(wave,flux,line=1,color=Color.black))
p.addLayer(LayerXY(wave,flux-stddev,line=1,color=Color.red))
p.addLayer(LayerXY(wave,stddev+flux,line=1,color=Color.red))
```

7.7.2. Inspecting the errors array of the projected/drizzled/interpolated *SpectralSimpleCubes*

The mosaic cubes of Level 2—projected and drizzled, all of which are *SpectralSimpleCubes*—can be inspected as can any cube in HIPE, using the Spectrum Explorer and associated tools (see the [DAG](#) chaps 6 and 7), but the error array cannot be explored with the Spectrum Explorer. Instead you can use PlotXY to plot the error spectra or to turn them into error bars, following this script:

```
from java.awt import Color # for a quick way to specify colours

# Extract a cube from the slicedProjectedCubes
cube=slicedProjectedCubes.refs[0].product
# Get the wave, flux, and error DoubleId for spaxel 12,12
wave=cube.getWave()
flux=cube.getFlux(12,12)
error=cube["error"].data[:,12,12]

# Now plot the wave and flux with PlotXY
p=PlotXY(titleText="spaxel 12 12")
p.addLayer(LayerXY(wave,flux,line=1))
# Now add error bars
p.errorY=[error,error]

# Or, plot the flux +/- stddev
p=PlotXY(titleText="spaxel 12 12")
p.addLayer(LayerXY(wave,flux,line=1,color=Color.black))
p.addLayer(LayerXY(wave,flux-error,line=1,color=Color.red))
p.addLayer(LayerXY(wave,error+flux,line=1,color=Color.red))
```

More examples of plotting errors with PlotXY can be found in chap. 3 of the [DAG](#).

7.7.3. The errors in the extracted central spectrum of point sources

The pipeline task `extractCentralSpectrum` ([Section 8.5](#)) extracts and point-source calibrated the spectrum of a point source located in the central spaxel, from the pipeline-final *PacsRebinnedCubes*. This task also extracts the "stddev" array and adds that to the "weights" array of the output spectrum. Weights are used in preference to errors in most HIPE spectral tasks, and are computed as the inverse square of the errors, i.e. $1/\text{stddev}^2$. If you write your extracted spectrum out as ASCII or FITS, look in this dataset/column to find your weights, which you can easily convert back to stddev values.

7.8. NaNs and flags in rebinned and mosaic cubes

Some comments that apply to all mosaic cubes and rebinned cubes.

- All mosaic cubes and the rebinned cubes have a flag array. This does *not* mean that there are data which are flagged as bad within these cubes, rather this flag array carries forward information about the masks that were active or inactive when the cubes were created. *To remind the reader, all data masked for instrumental effects, for saturation, and those falling outside of the band borders, are excluded* when the rebinned and mosaic cubes are created by the pipeline. The flag array is provided to allow one to be able to find out which masks were active, for any data-point, when a cube was created. It is expected that only the saturated data mask may be of interest to the user, and even then for a very few cases since saturated data is easy to spot (see next point).

- All the mosaicking tasks propagate NaN datapoints. To remind the reader: when the *PacsCubes* are turned into rebinned cubes, upon which `specInterpolate` and `specProject` work, the masked data are excluded. Similarly, when drizzle cubes are created, the masked data in the input *PacsCubes* are also excluded. Saturated data (from the masks called SATURATION and RAWSATURATION) are therefore excluded when the science cubes are created by the pipeline. Any datapoint in a rebinned or mosaic cube for which the input datapoints are masked, will create a NaN datapoint in the rebinned or mosaic cube.

This explains why, in the peaks of bright spectral lines or regions of high-flux continua, the data of the science cubes are NaN. Poor spectral sampling is also responsible for NaNs in the spectra, this occurring at the very edges of the spectral ranges or only for certain SEDs, for every other data-point. See [Section 7.6](#) for more information about the effect of the saturation and glitch masks on the rebinned of the *PacsCubes* by `specWaveRebin`.

The NaN regions are propagated into the mosaic cubes differently by `specProject`, `drizzle`, and `specInterpolate` because of the way their algorithms work. For the former two tasks, a NaN spectral region is usually of the same spatial extent as found in the input cube (e.g. just over one 9.4" spaxel). This is because the projection and drizzle algorithms only need to consider a small region around each input spaxel when creating an output spaxel; there are of the order of four "corners". For `specInterpolate`, however, there can be about eight corners, and, for each wavelength layer, NaNs are produced when even only one of these corners contains a NaN. The areas affected by NaNs are therefore larger than those from drizzled or projected cubes.

7.9. Drizzle: spectral-spatial resampling and mosaicking of cubes

Drizzle is a task that will perform a "drizzling" of input cubes onto an output grid, projecting the spectra from the input cubes onto the grid of the new, mosaic cube. In this section we summarise the important details of the drizzle task and its algorithm. Full details of the drizzling algorithm, and examples of it in use, are provided in the PhD thesis you can obtain from <http://fys.kuleuven.be/ster/pub/thesis-sara-regibo>. To learn how to use drizzle in a practical way, check out the pipeline scripts: for the default pipeline method ("Telescope normalisation") there is a dedicated script for drizzling chopNod line scans ("Telescope normalisation drizzled maps"), and for the "Calibration source and RSRF" script, the drizzle routines are at the end of the script.

7.9.1. When can I use drizzle?

Drizzle is designed to work over small wavelength ranges (i.e. line or small range spectroscopy)—a few microns at most but ideally $\sim 1\text{-}2\mu\text{m}$. This is partly related to processing time and memory requirements. But it is also related to the spatial grid that drizzle creates. It creates output cubes with a spatial grid such that the PACS beam is Nyquist sampled at the wavelengths of the input cubes. Since the PACS beam size varies with wavelength, it calculates the required spatial grid for the central wavelengths of the input cubes. These spatial grids will, of course, be slightly non-Nyquist at the wavelengths beyond or before the central wavelengths, but for small wavelength ranges this matters little. For long wavelength ranges it matters more.

To learn what defines a Nyquist sampling of the beam, see [Section 3.3](#).

7.9.2. Dealing with the off- and on-cubes (unchopped) and nod A and B (chopNod)

Here we give a little more detail on how the different pointings—nod A and B (chopNod), and off-source and on-source (unchopped)—are dealt with by drizzle, as it drizzles the nods separately (a difference with the other mosaicking pipeline task `specProject`).

Drizzle takes as input a *SlicedPacsCubes*, that is, a *ListContext* of *PacsCubes*. (This differs from *specProject* which works on *PacsRebinnedCubes*.)

- For *chopNod* AOTs, to correctly eliminate the telescope background the combined spectral cube must "see" the two nod positions with equal coverage. The two nod positions are drizzled separately, to the same spectral and spatial grid. This automatically takes care with any slight pointing offsets between nods. The fluxes of the nod A and B cubes are then averaged, assigning equal weights to the two positions. Spaxels in the resulting cube that did not receive flux in both positions are flagged (with the NO DATA flag).
- For *unchopped* AOTs the subtraction of the off-cubes from the on-cubes is done before drizzle is run, with the pipeline task *specSubtractOffPosition*.

7.9.3. Construction of the spatial grid

The construction of the spatial grid (the task *SpatialGrid*) uses parameters *oversample* and *upsample*, which have a similar meaning as they do for *wavelengthGrid* ([Section 7.2](#)).

The oversampling is incorporated in the calculation of the size of the spatial bins of the output cube grid. These spatial bins are shifted w.r.t. each other in steps of $(1/\text{upsample})$ times their size along the x- and y-axis. The size of the output spaxels should be smaller than those of the input spaxels (9.4 arcsec), and so more than one input spaxel can cover an output spaxel (and vice versa). The flux in any given output spaxel, at each wavelength (since the filling of the data in the spatial grid is done per wavelength of the spectral grid), is the sum of all contributions from the input spaxels, over the wavelength range of that wavelength's bin.

If you have a spatially oversampled observation (which is the recommended use-case for drizzle), you can improve the spatial resolution via the technique of *upsampling*. To calculate the flux in an output spaxel, this not only takes into account the fluxes that fall within that spaxel, but all fluxes that fall within a rectangular region of which the width and the height are a factor *upsample* larger than the spaxel.

7.9.4. pixFrac: the shrinking factor

The drizzling algorithm "shrinks" the pixels before they are drizzled. This is controlled by the parameter *pixFrac*. This shrinking of the input spaxels enables the elimination of the convolution of the sky plane with the PACS detector footprint, and allows for a more efficient outlier detection. The shrinking factor must be chosen (by the user) such that the resulting coverage is quasi-homogeneous in the region of interest, without creating holes. As the optimal shrinking factor will depend on what the target looks like, the user may have to run the algorithm a few times. In the pipeline a value of 0.6 is used for oversampled rasters and 0.3 for Nyquist sampled rasters (see [Section 3.3](#) to learn about over/under/Nyquist sampling with rasters), but see the above-references thesis for more information.

7.9.5. Outlier detection

You can input into drizzle *PacsCubes* that already have an OUTLIER mask created with the pipeline task *specFlagOutliers*. *SpecFlagOutliers* is a well-tested routine, and inspecting the flagged data is easy to do (and is documented in [Chapter 10](#)).

Drizzle also includes an outlier detection, and by default this is done (parameter *detectOutliers*). It works via an iterative #-clipping done on the input *PacsCubes*, but such that the clipping is done on the spectra of the output cube (smaller) spaxels, rather than on the input cube (larger) spaxels before they are then projected to the output spaxel-spectra. The user can specify the clipping level and the number of iterations. Detected outliers are placed in a DRIZZLE mask, and data so-masked will not be included in the construction of the final cubes. To see these masked data you need to open the *PacsCubes* output of the drizzled task: a copy of the input cubes but with the DRIZZLE mask (see the next section). These masked data can then be plotted e.g. with the pipeline helper tasks ([Section 10.2](#)) *plotCubes* and *plotCubesMasks*, or by using the Spectrum Explorer ([Section 10.5](#)).

The iterative #-clipping is similar to what `specFlagOutliers` does, but you should consider the two to be complimentary. There are some advantages in `drizzle`'s outlier detection: in particular, `specFlagOutliers` detects outliers by comparing *PacsCube* data along a fixed wavelength grid, for each spaxel, whereas the detection in `drizzle` looks at every spectral *and* spatial bin to find outliers: outlier detection should be more sensitive and efficient.

Consult the [URM](#) entry of `drizzle` to learn how to specify the relevant outlier detection parameters.

7.9.6. Drizzle error array

As discussed in [Section 7.7](#), one meaningful measure of the data errors in a cube can be computed from the dispersion (standard deviation) in the "dot cloud" that is the collection of spectra in each spaxel of the *PacsCubes* (if you use the Spectrum Explorer—see the *DAG* [chap. 6](#)—on these cubes, you will see what is meant by "dot cloud"). When `drizzle` constructs the spectral cube for unchopped observations or for any one nod position for chopped observations, the error on the flux in spaxel (x_o, y_o) , is:

$$\text{err}_{(x_i, y_i)} = \text{rms}_{(x_i, y_i)} / (n)^{1/2} \quad (7.1)$$

where n is the number of flux datapoints $d_{(x_i, y_i)}$ that are combined, and `rms` is the root-mean-square on the weighted mean $I_{(x_o, y_o)}$:

$$\sqrt{(\sum_{i=1}^n (s^2 \cdot d_{(x_i, y_i)} - I_{(x_o, y_o)})^2) / n} \quad (7.2)$$

where s^2 is the factor that ensures flux conservation. For chopped observations, the error for spaxel (x_o, y_o) in the combined cube can be calculated from the combined error for the two nod positions (the sqrt-sum-of-the-squares). It has been checked that this error is consistent with the continuum RMS in the final spectra, i.e. the spectral of the drizzled cubes (scaled by the square root of the coverage). Noting that the continuum rms is calculated over a larger spectral domain than a single spectral bin (which is what the propagated error gives), and that the propagated error does not include time-dependent factors, the two are consistent with each other.

The error is included in the drizzled cubes in the dataset called "error".

7.9.7. Structure of the output, and how to extract the sub-products that drizzle creates

The output from the `drizzle` task is a *ListContext* of *SpectralSimpleCubes*, where the raster positions have been combined, so there is one mosaic cube per wavelength range that was requested in the observation. This is what will be added to your Variables panel, and it will be called, if you are using the interactive pipeline scripts, `slicedDrizzledCubes`. In an *ObservationContext* it is called HPST3D-D[R|B] and is found at Level 2 or 2.5 (unchopped range). The datasets of the *SpectralSimpleCube* are given in [Section 10.3](#).

You can also get the following additional products out of `drizzle`:

- *First, you must not run drizzle the way it is written into the pipeline scripts.* The syntax used there will not allow you to extract the sub-products. The pipeline:

```
slicedDrizzledCubes = drizzle(slicedCubes, \
    wavelengthGrid=waveGrid, spatialGrid=spaceGrid)[0]
```

The `[0]` at the end returns the first product created by `drizzle`, which is the drizzled cube *ListContext*. To be able to also get the nod A and nod B drizzled cubes and the *PacsCube* with the DRIZZLE mask, you need to remove this `[0]`

```
result = drizzle(slicedCubes, \
    wavelengthGrid=waveGrid, spatialGrid=spaceGrid)
slicedDrizzledCubes = result[0]
```

- Now you can get the nod A and B drizzled cube *ListContexts*, which are the 2nd and 3rd cubes in result. Type,

```
resultNodAs = result[1]
resultNodBs = result[2]
nodA1 = resultNodAs.get(0) # get the first nodA cube
```

The first two are each a *ListContext* of *SpectralSimpleCubes*, a drizzled nod A or B cube per wavelength range present. The cubes, e.g. the third, can be inspected with the Spectrum Explorer (see [Section 10.5](#)). If you are not working with a chopNod observation, these cubes do not exist.

- Finally, to see the DRIZZLE mask, you want to extract the following.:

```
maskedCubes = result[3]
cube = maskedCubes.get(0) # get the first cube
```

The first is a *ListContext*, of *PacsCubes*, the second is a single cube. If outlier detection was switched off, this cube does not exist. If you are not working with a chopNod observation, and the nodA and nodB cubes do not exist, then the masked cube *context* will be number [1] rather than [3]. It is strongly recommended you check the data with this mask, *especially if you are using drizzle with sparse rasters (or single pointings) or choosing very small (or large) spaxels for the output.*

7.9.8. Using drizzle on longer wavelength ranges

Drizzle is designed to be used for observations of a short wavelength range. One reason is processing time and memory. The other reason is to do with the spatial grid, which the user specifies not by spaxel size but what sort of spatial sampling of the field-of-view is wanted: so not "3 arcsec please" but "1/3 of the PSF please". The spatial sampling chosen should give at least Nyquist sampling at the wavelength of your observation. Since the PSF increases with wavelength, then so will the spaxel size that is given to the drizzled cubes. To give the best results, it is therefore better to use drizzle on cubes of short wavelength ranges, so the spatial sampling is ideal for the entire stretch of the data.

If you want to use drizzle on a cube of long wavelength range, you need to cut it up into shorter ranges:

- *Unchopped rangeScans*: use the "Combine off-source with on-source" rangeScan script to reduce the data and subtract the off-source cubes. Then cut out the spectral region you want to drizzle

```
# You want to cut out the cubes of wavelength
# range 87.2,89.2
slicedDiffCubes = pacsExtractSpectralRange(slicedDiffCubes,[[87.2,89.2]])
```

You can then run drizzle on this product. See the [URM](#) to learn more about pacsExtractSpectralRange.

- *chopNod rangeScans*: use the "Telescope normalisation drizzled maps" script, which is provided for line scans. In specFlatFieldLine (which can be used on ranges of up to 5 microns), use the parameter maxrange to limit the flatfielding to the same spectral range you want to drizzle. You can use the same task "pacsExtractSpectralRange" on the first "slicedCubes" that the pipeline creates, just after the flatfielding, and then reduce the cut-up cubes through the rest of the pipeline.
- You can also use "Calibration source and RSRF" rangeScan pipeline script (if you chose not to use the Telescope normalisation method), to drizzle rangeScans, using the task pacsExtractSpectralRange on the slicedCubes created after the flatfielding (use either the parameter selectedRange in specFlatFieldRange to limit the flatfielding to the same spectral range you want to drizzle, or do the same as above and use the flatfielding from the line scan pipeline instead).

7.10. SpecProject

The task specProject takes as input the slicedFinalCubes (chopNod pipeline scripts) or slicedDiffCubes (unchopped pipeline scripts) or HPS3DR[R|B] (found at Level 2/2.5 in an observation) and

mosaics the different pointings in the raster together. The task is designed to be used for mapping observations but can be used for single pointings also: see [Chapter 9](#) for more information about the various mapping patterns of PACS observations.

SpecProject is a less complex task than drizzle, but it is the only projection task that can be used on long wavelength ranges to create a single mosaic cube for mapping observation. It can also be used to project the results of fitting to the spectra of rebinned cubes in a mapping observation, i.e. doing a 2d projection of fitting results; drizzle cannot do this.

The output grid that the mosaic cube is given can either be provided or it is computed to encompass all the pointings of the input cubes. The input spatial grid for each wavelength slice is projected on the output grid and the overlapping data are combined. The projection takes into account the overlap of the input spaxels with each of the output spaxels. This information is stored in two 3d arrays of size of the output grid: one containing the spaxel numbers of the (input) overlapping spaxels for each output spaxel, and one with their corresponding weights. The spectrum fluxes in the output grid are then the weighted sum of the contributing fluxes from the input spaxels. These weighted totals (per spaxel, per wavelength) are normalised, so that output spaxels with contributions from more than one raster pointing do not contain more flux simply because more than one pointing covers them. Flux is conserved by multiplying the flux by the input spaxel area divided by the output spaxel area. Errors are propagated using standard error propagation when the "error" or "stddev" dataset is found in the input data. See the [URM](#) entry for this task to learn more.

The main differences between the way specProject and drizzle work are:

- For specProject, the data must be rebinned first (with specWaveRebin) converting the *PacsCubes* into *PacsRebinnedCubes*. In this pipeline step, the wavelength grids within the input cubes are combined into one new wavelength grid in the output cube. The ra,dec array for each wavelength of the 25 spaxels is slightly different, and each wavelength sees a slightly different position on the sky compared to the next. This can be used to slightly increase the spatial sampling of the field of view. Drizzle takes advantage of this, but specProject does not (since it *averages* the wavelengths and the ra,dec arrays over the field of view). The drizzle cubes should look smoother.
- With specProject, the nod A and nod B rebinned cubes are first combined and then projected. Drizzle does not combine the nod positions until the very end, drizzling each one and then combining them. Since the two nod position do not coincide exactly in (Ra Dec), keeping them separate until the end also adds to the effective amount of spatial sampling in the observation. The drizzle cubes should look smoother.
- One extra provided in the drizzling algorithm is the shrinking of the input spaxels, which enables the elimination of the convolution with the detector footprint and a more efficient outlier detection.

7.11. SpecInterpolate

SpecInterpolate is designed to create mosaic cubes for tiling observations, where the beam is under-sampled, and it can also be used to map single pointings onto a regular spatial grid: see [Chapter 9](#) to learn more.

The task is not a re-mapping/projection such as drizzle and specProject. The task takes spatial grids of the input rebinned cubes, and creates a new spatial grid, with spaxels of the specified size using Delaunay triangulation to compute the new grid (this method finds an ideal arrangements of triangles with which an irregular grid can become a regular grid). The fluxes of the new spaxels are the computed by interpolating between the spaxels of the old grid and those of the new, and overlapping data are combined. This spatial interpolation is done for each wavelength 'slice' of the cube. The output grid can either be provided or it is computed to encompass all the pointings of the input cubes.

Since the new flux grid is an interpolation from the input flux grid, this means that the accuracy of the results will not be as high as with specProject or drizzle.

A word on NaN handling in specInterpolate. NaNs can be found in the spectra of the rebinned cubes mainly when: a data-point or string of data-points of the preceding *PacsCube* is saturated; or the

spectral sampling was optimised for one camera and ends up being poor for the other camera, resulting in wavelength bins that are not filled with data. These NaNs cannot be interpolated in the same way as real numbers can. Instead, the interpolation produces NaNs in the output cube for that wavelength slice when one of the corners of the Delaunay triangles contains a NaN for that same wavelength slice. The area affected by NaNs in `specInterpolate` are larger than the area affected by NaNs in `specProject`, since `specInterpolate` will usually have eight adjacent spaxels but `specProject` and `drizzle` only four.

The spatial grid for interpolated cubes are created using the Delaunay triangulation and *interpolation* from the old spatial grid to the new one. There is no *extrapolation* done at the edges of the field: the mapped field is therefore smaller than that achieved by the other mosaicking tasks (this is especially obvious when looking at cubes from pointed observations).

See [Figure 9.3](#) for a comparison of a cube produced by `specProject` and one produced by `specInterpolate`.

Note that errors are propagated by `specInterpolate`, but the `interpolate` error is not added to this.

7.12. The Pointing offset correction tasks

The tasks of the pipeline **Pointing offset correction (point sources)** are introduced in [Chapter 5](#), for the `chopNod` pipelines. This pipeline script is aimed at pointed observations of point sources, which are bright (at least $\sim 10Jy$) and centred within the central spaxel. More detail on these tasks can be found in their [URM](#) entries. Here we explain what the aim of these tasks is.

Remember that the spaxels of PACS are 9.4" on a side, while the FWHM of the beam is $\sim 9''$ in the blue to $\sim 13''$ in the red. When observing point sources, you will notice that almost all of the flux lies in the central spaxel, or at most the central 3x3. Summing up the spectra of these spaxels is not enough to obtain a correct spectrum of the point source. It is necessary to apply flux corrections to the extracted spectrum to account for the amount of flux that is lost: from between spaxels and because the beam is larger than the central 3x3 spaxel area. The point source flux losses applied by `extractCentralSpectrum` are to do exactly this: in analogy to photometry, we are applying an aperture correction.

However, these flux corrections ought to depend on where, exactly, within the central spaxel the point source is located, since if it is on the very edge, the flux losses will be different than if it is located exactly in the centre. The task `extractCentralSpectrum` can apply a second order correction to account for this, however it computes one correction for an entire observation.

Since PACS spectra are gathered in time, with the grating wheel incrementing forward and backwards along the wavelength range during an observation, if the telescope pointing was jittering about in time then it is also effectively jittering about "in spectrum". Hence, a better result can be obtained if the flux corrections follow how the source jitters about during an observation. It is this that the pointing offset pipeline attempts to do. The spectra coming out of `extractCentralSpectrum` should be smoother than otherwise.

There are two choices for computing the pointing offset correct, and which you use is determined by the value of `usePointingProduct`.

1. `usePointingProduct = 1` (the default): the POC (pointing offset correction) has two components, and this option requests that both are used.

The first component is the absolute offset, which is computed from the source itself by comparing the spatial distribution of its flux in the central 9 spaxels to that for a perfectly-pointed point source (which is taken from the beam calibration files). This relies entirely on the flux measured in the central 9 spaxels, hence it requires a good S/N (preferably continuum, not only line) in the observed spectrum to work.

The second is the pointing jitter, which is computed from the gyro-propagated pointing products that come with the `ObservationContext`. The jitter is derived from the gyro-propagated pointing products that come with the observation, rather than from the observation data itself, since it has

been found that this produces superior results. For this you need an SPG observation reduced with SPG 13 or higher.

2. `usePointingProduct = 0`: compute only the first component. Tests have shown that this gives an inferior result to `usePointingProduct = 1`, except for sources of continuum flux of more than 40 Jy over the entire spectral range.

Determining the time-resolved pointing offsets by comparing the flux distribution of your source with the beam files can be done for any chosen chunks of time. However, it is necessary that enough signal is used that the comparison of the observed flux distribution to the beam flux distribution produces reliable results. In the pipeline script, a median offset per nod (i.e. per slice) is determined (`perNod = True`). This is particularly aimed at weaker sources, where the data of an entire nod should be more reliable results than the data of a few wavelength points/grating positions, but is suitable for brighter sources also. However, it has been found that when fitting the position for an entire nod in one go, better results are obtained when using the parameter `smoothFactor` (the number of grating steps over which an average is taken when fitting the point source's position) to determine a set of values over the nod, and then taking the median of that set of values, rather than taking one median value for the entire nod, even for noisy data.

The pipeline tasks are as follows:

1. **specDetermineChiSquare**: determine a 2D ChiSq map that is the difference between the flux distribution of the observed point source and that of the beam. In the scripts we give the recommended value of `smoothFactor` to use, for fainter sources it may be worth trying a higher value. There are also parameters used in the interpolation of the beams (i.e. the sampling of the beam profile). Small is better, up to a point, but also much slower. Always start with the values in the pipeline script, and experiment around those values. Use the parameter `oversample` to change the spatial oversampling factor of the interpolation (between the data and the beams): higher numbers will work for brighter data.
2. **specDeterminePointingOffset**: using the ChiSq maps, determine the values of the pointing offsets, i.e. those which caused the measured flux offsets. A smoothing along the timeline can be applied here (the default is to not smooth), and the actual value of ChiSq to accept is also a parameter.
3. **specDeterminePreCalculatedPointing**: to calculate the pointing jitter during the entire observation, using the gyro-propagated pointing products. It is strongly recommended that you do apply this task, and then the jitter is derived from the pointing products but the mean position from the tasks above.
4. **specDeterminePointingOffsetFromPreCalculatedPointing**: to add the flux correction factors that correspond to the mean and the jittering pointing offsets.
5. **plotPointingOffset**: after these tasks, you can plot the determined pointing offsets to see where your source was during the observation.
6. **specApplyPointingCorrection**: and finally, apply the flux corrections to the central spaxels of the cube.

After this, the output "c9" or "c129" from `extractCentralSpectrum` can be used to get the point-source calibrated spectrum (at the end of the rest of the pipeline).

7.13. The transient corrections of the unchopped pipelines

"Transients" are events (e.g. cosmic rays) that affected the response of the detector immediately subsequent to the event. Some die away quickly, others take longer. In particular, the observation of the calibration block in the beginning of all unchopped observations often had a transient effect, resulting in an increased response immediately afterwards, followed by a slow normalisation. This results in a higher value of the signal at the beginning of the observation, i.e. a "transient".

In the chopNod observing mode the transients have a much smaller effect, because in the pipeline we work with the *differential* signal: where the chop-on and subsequent chop-off datapoints are taken very close in time to each other and subtracted from each other before being converted to Jy. But in the unchopped pipeline we deal with the absolute levels, and so the transients need to be corrected.

There are two pipeline methods for the unchopped mode, the first set of scripts ("Calibration source and RSRF") which are also the core of the SPG scripts for this mode, and the "...with transient correction" which contain new correction tasks. These latter can give better results, but they are interactive: you need to run them and check the results to be sure you are happy with the corrections applied. It is for this reason that the new scripts are not the SPG scripts. If dealing with unchopped observations, and you see very negative continuum fluxes in your cubes, or spectra that seem noisy or have sharp unexpected features in them, it is worth running these new pipeline scripts, *interactively*, and comparing the results.

However, note that even with the new transient correction tasks, negative fluxes in the continuum levels is possible, especially for sources with a low continuum in any case. The uncertainty in the continuum level for this mode, whichever pipeline is used to reduce the data, is much higher than for the chopNod mode. (See the spectroscopy calibration documentation on herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb to learn more about the uncertainties.)

The crucial transient correction pipeline tasks in the original and new pipeline scripts are explained in more detail here. Their use in the pipeline is explained in [Chapter 6](#).

The tasks of the original pipeline, "Calibration source and RSRF" for line scans:

- **specLongTermTransient:** This task will remove the effect of the long-term transient that occurs the beginning of an observation for the unchopped line mode.

The effect of the transient on the signal can be modelled as a linear combination of exponential functions. Upward- and downward-pointed transients can occur, and these are modelled by different functions. The task finds the optimal parameters for the transient models by minimising the scatter of points in the final spectrum after correcting for the inflated response in increments. The final spectrum is built up from a time-series of sampled datapoints derived separately from up-and-down grating scans (i.e. the different directions over the grating over are dealt with separately because they have intrinsically slightly different flux levels). Note that the transient mainly affects the first two up-and-down scans. The correction algorithm works separately for each spaxel/module.

- **plotTransient:** This task plots the data taken at the beginning of the observation scaled to that taken at the end: if the data at the beginning slope up or down to meet a baseline that continues until the end of the observation, then you have a transient. The plot is the data (black points) from the first science *Frames* (slice = 1) for the central module of the IFU (module = 12) normalised to the signal level in the final grating scan data-chunk. The various grating scans cover the same wavelength range as each other (and there are at least two grating scans in any observation): since the effect of a transient is to increase the signal levels, a transient will make the first spectral scan stand out from those of later in the observation. The second call to this task plots the corrected data as red points.

The tasks of the new line scan transient correction pipelines:

- **plotLongTermTransientAll:** This task plots the data taken at the beginning of the observation normalised to the expected background telescope flux. The on- and off-source data are plotted as blue and green points. Transient-affected data will stand out as jumps with trailing tails. If the data at the beginning slope down to meet a baseline that continues until the end of the observation, then you have the long-term transient that often occurs just after the calibration block has been observed. This long-term transient is what is corrected for in the next task, and then a second call to plotLongTermTransientAll will plot the long-term transient fit as a red line.

You can chose which module (0—24: these correspond to the spaxels: [Section 10.6](#)) to plot.

- **specLongTermTransientAll:** This task tries to compute the long-term transient along the entire history of the signal (therefore using all the slices in the input product), by normalising the signal to the expected telescope background flux. (This normalised signal is what is plotted in plot-

LongTermTransientAll.) If the source does not have a substantial continuum, and the telescope background model correctly approximates the actual measured background, then the correction this task produces (and which is then applied by the subsequent task) will produce a superior final result. This task has in particular been developed to obviate the problem of negative fluxes found in some observations, which is due to the fact that the off-target observation is performed first, just after the calibration block which causes a transient that artificially enhances the measured flux.

- **specApplyLongTermTransient:** This task applies the correction computed by the task spec-LongTermTransientAll.
- **plotTransient:** see above.
- **specLongTermTransCorr:** Sudden differences in flux between two adjacent slices cause long-term transients in the detector response. This task fits a transient model to the signal normalised to the last wavelength scan, and applies the correction to the data.
- **specMedianSpectrum:** This task computes a first guess of the spectrum for each spatial module of the *Frames* ("modules" later become "spaxels") using the median value of the flux (per wavelength) from the 16 pixels that feed each module. The median spectrum is used to normalise the signal by the next task, specTransCorr, to compute the transient correction.
- **specTransCorr:** This task identifies the discontinuities in the signal caused by cosmic ray hits, and fits the transients in the detector response that occur after the cosmic ray hits, using a model of the transients computed by the PACS team. The result is applied to the signal to free it from the effect of cosmic ray hits, i.e. short-term transients. Any signal which is too damaged to be corrected is instead flagged in a new mask: UNCORRECTED.

The tasks of the new pipeline for range scans:

- **plotTransient:** see above.
- **specLongTermTransCorr:** see above.
- **specUpDownTransient:** in range spectroscopy the variation of the continuum during a wavelength scan can be so large that it induces a transient behaviour in the signal. This affects the up-scans and down-scans differently, since normally the blue end is brighter than the red end of the wavelength range. This task therefore treats the up-scans and down-scans separately, median-correcting all the scans to the global mean. The correction applied is at the halfway point between each up- and down-scan.
- **specMedianSpectrum:** This task computes a first guess of the spectrum for each spatial module of the *Frames* ("modules" later become "spaxels") using the median value of the flux (per wavelength) from the 16 pixels that feed each module. The median spectrum is used to normalise the signal by the next task, specTransCorr, to compute the transient correction.
- **specTransCorr:** see above.

Note that the transient corrections tasks separate out the data into populations. PACS data were taken continuously, while the grating was moving and while the instrument was moving on the sky (e.g. for rasters). Hence there is a population of data that were taken during movements and which will have been masked as bad, and a population that are good data. There are additional populations of datapoints that come from different parts of the instrument. The PACS signal-detector was constructed from 25 rows of 16 pixels: each pixel was in fact a separate detector with its own response, and each row of 25 corresponds to a single spaxel and hence to a single position on the sky. The signal stream from each pixel for each module is also a separate population. Finally, the response of each pixel differs if you are moving along the grating towards increasing or towards decreasing wavelength, and this makes additional populations.

Chapter 8. Dealing with point and slightly extended sources

8.1. Introduction

To extract the correctly-calibrated spectrum of a point source, it is not enough to sum up the field-of-view of a rebinned cube: point source corrections *must* be applied.

For slightly extended sources—small enough to fit in the field-of-view of a single pointing (about 47"x47"), with a FWHM of too not much larger than a spaxel (9.4")—there is a task to correctly calibrate the spectrum. This correction is based on the difference between the morphology of the source and that of a point source; hence you need to know the surface brightness distribution of the slightly-extended source for the results to be meaningful.

In this chapter you will find:

- Combining PACS and SPIRE spectra for point sources: [Section 8.2](#).
- Extracting the spectra of point sources that are not located in the central spaxel: [Section 8.4](#).
- Extracting the spectra of point sources that are located in the central spaxel: [Section 8.5](#).
- Extracting the spectra of slightly extended that are centred in the central spaxel: [Section 8.6](#).
- Correcting for the uneven illumination of the PACS IFU using the forward modelling tool: [Section 8.7](#).

8.2. Combining the PACS and SPIRE full SED for point sources

It is possible to observe the entire SED, from 50 to 680 μm , with two PACS observations—covering the bands B2A and B2B and their accompanying R1 ranges—and one SPIRE observation—covering the SSW and SLW bands—and we have provided a way to look at these data contained within a single spectral product, rather than as 6 separate spectral products. Comparing PACS and SPIRE *cubes* in a scientifically meaningful way (whether for full SED coverage or spectra of shorter regions) is not so straightforward—it would require, for example, adjusting the data for the differences in the beam sizes and spectral resolutions of the instruments, particularly when working with extended sources. But for single spectra of point sources extracted from the cubes it is a more simple matter.

In the HIPE Scripts menu is a script to do this combining, working from some public obsids; *Scripts#PACS Useful scripts#Spectroscopy: Combine PACS and SPIRE spectra*.

Note that this task does not mathematically combine the spectra, it simply pushes them together into the same *Spectrum1d*. The advantages of this is that you can keep them together, including writing the *Spectrum1d* out to disk as FITS. A *Spectrum1d* is a class of spectrum that contains columns of flux, weight, flag, wavelength, and segment: with this segment number you can distinguish spectra that came from different sources (e.g. PACS can get segment numbers 1 to 4 and SPIRE can get segment numbers 5 and 6).

The process is quite straightforward. Open the script and run the following steps:

1. Extract the point source spectra from your cubes; this has to be done on a PACS HIPE build for PACS data and a SPIRE HIPE build for SPIRE data, or on an all-instrument build, since these tasks are instrument-specific.

For PACS there are two tasks to extract the point source spectrum (see [Section 8.4](#) and [Section 8.5](#)), and the combining script includes an example of how to run the most commonly-used task.

For SPIRE this is done in a task that is provided (as a jython function) in the combining script.



Tip

If you don't have an all-instrument build, you can run e.g. the PACS part in a PACS build of HIPE, write the file out as FITS, and then open a SPIRE build and read the PACS file back in to HIPE.

2. The PACS and SPIRE spectra will each be pushed into two *Spectrum1d* products, one for PACS and one for SPIRE, in which the different input spectra will have different segment numbers. If you are working with full SED coverage data then you will have 4 PACS spectra which you push into a *Spectrum1d* and 2 SPIRE spectra which you push into a *Spectrum1d*, but you can work with more or fewer spectra. This can be done on any build of HIPE, as the tasks involved are instrument-independent. For PACS you are offered the chance to set flags for bad data/regions and for SPIRE you are offered the chance to change the wavelength units to micrometers.
3. Finally you will combine the PACS *Spectrum1d* with the SPIRE *Spectrum1d* product into a single, new *Spectrum1d*, with all spectra being converted to the same X-axis units ("GHz", "micrometer", "cm-1"). The flux units must be Jy.
4. Note that the PACS spectra have a "weights" column, while the SPIRE spectra have an "error". we have added an column to the PACS+SPIRE *Spectrum1d*. In the script we add an error column to the PACS product so it matches the SPIRE one, but it is filled with 0 (it is actually easy to fill it with correct values, as the weights are based on the errors).

You can view all the *Spectrum1d* you create with the Spectrum Explorer. The new spectra will appear in the *Variables* pane and the Spectrum Explorer should be the default viewer for them. You can also inspect their Meta data, we have copied over the relevant Meta data from the input products. Use the Product viewer or Dataset viewer to see Meta data. Finally, any *Spectrum1d* can be saved as a FITS file.

8.3. Spectral line skew induced by off-centred sources

For point or point-like sources that are off-centred in a spaxel (any spaxel) in the dispersion direction, the spectral lines will change from Gaussian to Skewed-Gaussian. The greater the offset, the greater the skew, and the direction of the skew (to the short or long wavelengths) depends on the side of the spaxel the source is off-centred towards. This affected every single pointing, so if you are working with mapping observations of point-like sources, the skew will find its way into the mosaic cubes also. More detail is given in the Observer's Manual on the website: [here](#)

To tell which direction is what, open any rebinned cube Standard Cube Viewer or the Spectrum Explorer (this will not work on a drizzled, projected, or interpolated cube).

- The dispersion direction: is the up-down direction. So e.g. if a source is located between spaxel 2,1 (module 7) and 2,2 (module 12: central), or spaxel 2,2 and 2,3 (module 17), then it has moved in the dispersion direction. The spaxel coordinates are given in a box at the bottom left of the viewers. (You can also check the table in [Section 10.6](#)).
- The chopping direction: is the left-right direction in a rebinned cube, perpendicular to the dispersion direction.

8.4. Extracting point source spectra that are not located in the central spaxel

The spectrum of a point source must be extracted from the *PacsRebinnedCube(s)* in the slicedFinalCubes (chopNod pipeline scripts) or slicedDiffCubes (unchopped pipeline scripts): you *cannot* do this

on a projected, interpolated, or drizzled cube. If working directly from an observation, you will be looking for the HPS3DR[R|B] product at Level 2 (or HPS3DRBS[R|B] at Level 2.5 if there is one).

In this section we explain what to do if your point source *does not* lie in the central spaxel, or falls between spaxels. If your task lies in the central spaxel, go to the next section.

The tasks you will use for the non-central are also detailed in the script of the HIPE Scripts menu: *Scripts#PACS Useful scripts#Spectroscopy: point source loss correction (any spaxel)*. See this script to learn how run the necessary tasks.

- **undoExtendedSourceCorrection:** before doing the extraction of the spaxel containing your point source, it is necessary to remove the extended source correction with this task.
- **extractSpaxelSpectrum:** a simple task that extracts the spectrum from any spaxel and returns a spectrum in the *SimpleSpectrum* format.
- **pointSourceLossCorrection:** this task takes the extracted spectrum and applies the point source correction. This correction is based on a PSF model (including the wings), and has been verified at a 15 wavelengths (see its [URM](#) entry). This correction is contained in the calibration tree (`pointSourceLoss`).

You can open the Spectrum Explorer (see the *DAG* [chap. 6](#)) on the spectrum created, and you can also save it to disk as a FITS file: right-click on the product in the Variables panel to access that option, or use the `simpleFitsWriter`.

Note that the point source must be centred *within* a spaxel for the correction to be at its most reliable. To check you can look at your rebinned or interpolated cube e.g. with the Standard Cube Viewer or the Cube Toolbox. However, even in this case, the resulting uncertainty is included in the published accuracy values for the absolute flux calibration (you can also consult the PACS public wiki for the Observer's Manual and calibration information and documentation (currently at herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb?template=viewprint).

We also point out that the point-source correction applied by `extractSpaxelSpectrum` is that based on the beam of the central spaxel. The beams of the other spaxels are similar, but not the same. Hence, the correction applied as explained here will always be inferior to that applied to centrally-located point sources. But it is still recommended to applying no correction at all. Alternatively, one can use the "Forward modelling tool", with a point-source (located in whichever spaxel) as the input model: see [Section 8.7](#).

8.5. Extracting point source spectra that are located in the central spaxel

For observations of point sources located in the central spaxel, you should use **extractCentralSpectrum** to extract and calibrate the point source spectrum. If your point source is located well outside of the central spaxel, see the previous section for the alternative task.

Note that the point source must be centred *within* a spaxel for the correction to be at its most reliable. To check you can look at your rebinned or interpolated cube e.g. with the Standard Cube Viewer or the Cube Toolbox. If your source is slightly offset, then the `c9` spectrum (being based on the sum of the central 3x3 spaxels) will give a more correct flux. It is also possible to try the "Pointing offset" pipeline script if your observation is `chopNod` ([Section 5.2](#)). In any case, even for slightly offset sources, the resulting uncertainty is included in the published accuracy values for the absolute flux calibration (you can also consult the PACS public wiki for the Observer's Manual and calibration information and documentation (currently at herschel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb?template=viewprint).

Errors: the "stddev" dataset contained in the rebinned cubes this task works on is propagated by the task: the outputs have a "weights" dataset that is $1/\text{stddev}^2$. This is calculated either only from the central spectrum or is the quadratic sum of the errors of the central 9 spectra.

8.5.1. What

The task `extractCentralSpectrum` is included in most of the interactive pipeline scripts, and is also part of the PACS Useful script: *Scripts#PACS Useful scripts#Spectroscopy: point source loss correction (central spaxel)*, where we begin directly from an *ObservationContext* downloaded from the HSA. The task in the pipeline works on rebinned cubes (*PacsRebinnedCubes*): the individual cube slices of the product called "slicedFinalCubes" or "slicedDiffCubes". If working directly from an *ObservationContext*, you will be looking for the HPS3DR[R|B] product at Level 2 (or HPS3DRBS[R|B] at Level 2.5 if there is one).

The task `extractCentralSpectrum` starts by removing the extended source correction factor, then extracts spectra from the cube and applies the appropriate corrections. It produces three output spectra, two of which are more robust against pointing jitter and slight pointing offsets, but are not suitable for the faint spectra or those with uncertain continuum levels. The point source loss correction applied by this task is taken from the same calibration file (`pointSourceLoss`) that is used by the task `pointSourceLossCorrection` (previous section). The correction is based on a PSF model (including the wings) and has been verified at a 15 wavelengths.

The task returns three spectra:

1. **c1**: The central spaxel spectrum with the "c1-to-total" point-source correction applied; this is for cases where the source spectrum is almost completely confined to the central spaxel, i.e. faint sources. This output is, in fact, the same result as obtained for non-centrally located point sources (previous section).

If you used the Pointing offset correction (point sources) pipeline, the "c1" spectrum should not be used. The correction applied by this pipeline is to all 9 central spaxels, and this renders the "c1" output invalid.

2. **c9**: The summed spectrum in the 3x3 central spaxel "box" with the "c9-to-total" point-source correction applied. The effect of slight pointing jitter and slight pointing offsets (##0.5 spaxel) during an observation will be to move flux out of the central spaxel, but in almost all cases most of the flux will still be within this central 3x3 "superspaxel". Using this spectrum is therefore more robust to these slight movements than c1 is. However, this product should only be used if the source is bright enough to have noticeably more signal in the 3x3 spaxel box than in the central one, and not a worse SNR.
3. **c129**: A combination of the two: the spectrum of c1 scaled on c9. The computation of the scaling is based on the ratio of the spectrum of the central spaxel to the spectrum of the central 3x3 spaxels, comparing that from the data to that from the calibration tree (see the [URM](#) entry for `extractCentralSpectrum` for more information). The scaling can be a single value (line scans) or a smoothed curve (range scans). This spectrum is the preferred one for sources bright enough that the SNR of c1 is better than that of c9, but where there is still noticeable signal in c9, and as it includes c9 this spectrum is also more robust to pointing jitter than is c1.

If working with unchopped observations, or if you reduced data longwards of 190 microns with the special RSRF for these wavelenght, the "c129" spectrum should not be used. The scaling used to create this spectrum is invalid for these modes.

8.5.2. How to run the task

The syntax to run this task and the parameters to specify are included in [Chapter 4](#) and in the useful script, and that is not repeated here. See also the [URM](#) entry for `extractCentralSpectrum` to learn more about its parameters and an explanation of what it does. Most of the parameters you need to think about are those used to control the scaling of c1 to c9 that is then incorporated in c129. You can request a wavelength-dependent or -independent scaling.

- **For range scan and SEDs** you can apply a wavelength-dependent scaling. Pointing jitter affects the fluxes measured by PACS, but because telescope jitter is time-dependent and the gathering of the

signal from the source is *also* time dependent (the grating moves along the wavelength range with time), the jitter *also* affects the shape of the spectrum. Applying a wavelength-dependent scaling is a second order "correction": it will not improve the flux accuracy any further, it will only (slightly) correct the spectral shape of the continuum.

In this case, there are some parameters to choose that control the smoothing of the scaling. Two filters can be applied to the scaling "spectrum", and the associated filter widths—`width` and `preFilterWidth`—help you achieve the smoothest result that still reflects the global shape of the scaling spectrum. Alternatively ask for the wavelet filtering method instead of smoothing. The filtering/smoothing is done to the scaling curve, not to the actual extracted spectrum. Finally, you could try the task with the "median" smoothing (that used for line scans) and compare the results.

- **For line scans** we recommend a wavelength-independent scaling: `smoothing="median"` or `isLineScan=1`. The second order "correction" for the effects of jitter used for range scans is not done here, as it cannot be reliably calculated for the short wavelength range of line scans; the effect of the jitter for these data is anyway usually not significant.

The task will produce two plots if you run it with `verbose=1`. The plots will appear in the *Editor* pane and show:

1. The "Central9/Central" plot shows a blue and a green curve overplotted on the data they are created from.
 - The light grey (data) and blue (smoothed fit) is the ratio of the spectrum from the central spaxel ("s1") to the sum of the central 3x3 spaxel box ("s9"), compared to the ratio expected for a perfectly-pointed point source. This comparison gives you an idea of how well pointed and/or how point-like your source is. The mean value of this curve is printed to screen as "(median) correction", together with the relative error of the data the "correction" was calculated from (RMS/\sqrt{n}), from the grey spectrum with n being the number of datapoints in the spectrum), and the RMS in those data. **If the printed mean value is greater than 20-30%, this probably means that your source is slightly extended, too faint for a reliable calculation, or not well-enough centred in the central spaxel.** In this case you should not use this task or accept that the result will be uncertain. If the RMS value are high, the ratio is also likely to be uncertain.
 - The green is the smoothed version of the thick black curve, which is the ratio of "s9" point source corrected (i.e. "c9") to the spectrum from the central spaxel ("s1"), and is the amount of shift that the "c129" spectrum has compared to the central spaxel spectrum.

2. A plot of all three spectra, c1, c9, and c129, together with labeled colours.

8.6. Extracting the spectra of slightly extended sources

For sources that are slightly extended—those that are still entirely contained within the central 3x3 spaxels of the IFU (a practical limit being a diameter of about 15")—the correct integrated flux can be obtained using two tasks in HIPE. First a point-source calibrated spectrum is extracted and then a correction from that to the semi-extended source morphology is made.

The `specExtendedToPointCorrection` task calculates the coupling of the source distribution model with the PACS beam profile (wavelength-dependent), i.e. the difference between the correction for a point source and the extended source based on the difference in the morphology of the two (the surface brightness distribution). This correction is essentially the amount of flux expected inside the central spaxel, or the central 3x3 spaxels, from the slightly-extended source compared to a point source, and this is always less than 1.0. The task uses an ellipse Top Hat function as the default source model but others can be provided. When dealing with a sources that is not centred within the central spaxel, it is necessary that the surface brightness model includes this "offset".

With respect to the first part of the job—extracting a point-source spectrum: the recommendation is that the task `extractCentralSpectrum` is used, with the output "c1" only for faint sources; "c129", or

"c9" for unchopped observations, otherwise. The best results will always be obtained for sources which are either centrally-located, or at least cross into the central spaxel, and so `extractCentralSpectrum` is used as the first part of this semi-extended correction task-set. For sources which only cross into the central spaxel (i.e. which are not centrally-located within it) the recommendation is still to use `extractCentralSpectrum`, taking the 3x3 output of the task (c129 or c9), and ensuring that your source model includes the correct position of the source in the FoV. Formally it is possible to apply the corrections for a semi-extended source which does not have any flux in the central spaxel, by using the point source correction tasks for non-centrally located sources (as explained in [Section 8.4](#)), instead of `extractCentralSpectrum`. However, be aware that if the source does not cross into the central spaxel then it is located close on the edge of the FoV: not only is the point-source correction factor less correct outside of the central spaxel (see the advice in [Section 8.4](#)), but it is also possible that some of the source actually falls out of the FoV.

The task—**specExtendedToPointCorrection**—produces a correction spectrum that must be divided into the point source extracted spectrum you get from your cube (using either **extractCentralSpectrum**, or **extractSpaxelSpectrum** preceded by **undoExtendedSourceCorrection** and followed by **pointSourceLossCorrection**). For instructions on the use of `extractCentralSpectrum`, see [Section 8.5](#), and [Section 8.4](#) for the alternative.

The inputs are explained in its [URM](#) entry. These include:

- A *SimpleSpectrum* the output of the point source tasks. You must also input a rebinned cube from which the wavelength grid is taken.
- *If adopting the default ellipse morphology*, you need to input the major and minor axes of the ellipse and position angle.
- *Or you can input your own model shape* (this requires an ability to script in HIPE). You can define a model shape in HIPE, "MySource", in the following way (example is for a Gaussian source with a 5" diameter):

```
from herchel.pacs.spg.spec.beams import ExtendedSource
from math import *
#
class MySource(ExtendedSource):
    sigma = 0.0
    def __init__(self,diameter):
        self.sigma = diameter/2.3548
#
    def getFlux(self,relRa,relDec):
        flux = exp(-(0.5*(relRa/self.sigma)**2+0.5*(relDec/self.sigma)**2))
        return flux
#
# Gaussian source with 5.0 arcsec diameter
source = MySource(5.0)
epCorr = specExtendedToPointCorrection(spectrum,calTree=calTree,\
    userSource=source)
```

- *The offset of the source* from the central spaxel (in arcsec).

See the [URM](#) entry for the task for guidance on its use and parameters. The parameter `central3x3` determines if the point source correction was calculated over the central spaxel or the central 3x3 spaxels. It is important that this parameter is coordinated with the output of the point source correction tasks: i.e. what you use in one you must use in the other. If you set this parameter to *False* then the correction spectrum is to correct the fluxes of the central spaxel only, suitable for use with `extractSpaxelSpectrum+pointSourceLossCorrection` or "c1" from `extractCentralSpectrum`. If you set this parameter to *True*, then you can take either the second or the third ("c9" and "c129" in the pipeline scripts) output from `extractCentralSpectrum` and you cannot use the alternative point source task. The default value is *False*.

After running `specExtendedToPointCorrection` and one of the point source extraction tasks, divide one output by the other. This can be done using the task *divide* which will run on the command line, via its own GUI (see "Tasks"), or via the Spectrum Explorer. On the command line this is a simple as:


```
result = divide(ds1=PointSourceSpec, ds2=ExtendedCorrection)
```

where `ExtendedCorrection` is the result of `specExtendedToPointCorrection`, and `PointSourceSpec` is the output of either `extractCentralSpectrum`, or the task-set `undoExtendedSourceCorrection + extractSpaxelSpectrum + pointSourceLossCorrection`.

The final spectrum is not referenced to a beam of a certain size, but to the entire source flux distribution. The application of this task is recommended for sources whose spatial extent falls within the central 3x3 spaxels (around 15" diameter being a practical limit). For larger sources, it is preferable to integrate the spectra of the entire 5x5 spaxel PACS rebinned cube, but it will still be necessary to compare the total flux levels to e.g. a PACS photometer map to be sure that the complete flux of the source has been accounted for (here see [Section 8.7](#) for important information.)

8.7. Correcting for the uneven illumination: the forward modelling tool

The spaxels of the PACS IFU are not evenly illuminated, with the result that there is effectively some flux loss between the spaxels over most of the PACS wavelength range. To correct for this an extended source correction was created, and this is applied to *all* observations by the pipeline. This fully corrects the integrated fluxes for all fully-extended sources (those with a flat or gradual flux gradient of no more than 20% within a single 47" square FoV). For point sources sources, this correction is taken out before the point source corrections are applied, and the fluxes will still be correct. This also applies to semi-extended sources if using the extended-to-point correction in HIPE. However, the uneven illumination of the PACS IFU FoV affects the flux distribution in the rebinned cubes and the spectral calibration in surface brightness. This may result in an incorrect integrated spectrum extracted from an aperture in spectral cubes that contain:

- crowded fields (e.g. multiple point sources)
- off-centred point sources
- semi-extended structures (especially if not extracted using the provided tools: [Section 8.6](#))
- sources which are extended but with steep flux gradients

The inaccuracy in the flux will depend on the morphology of the source at the wavelength of your observation and its coupling to the detector's beam efficiencies on every pointing (mapping) pattern. If you do aperture photometry on your source (from small sources to the entire field), you could end up under- *or* over-estimating the flux in the aperture. To estimate the inaccuracy in the flux, if you know (or can model/estimate) the surface brightness distribution of the source (i.e. its morphology at the wavelength of your observation), you can apply a "forward projection" in HIPE: this takes in your input surface brightness model/image and a model spectrum, and working with the pointing/mapping pattern of your observation, it produces result which folds in the uneven illumination. You can then compare the modelled result to your observed result.

8.7.1. Description of the "forward modelling" process

The task `pacSpecFromModel` simulates PACS cubes according to a user-provided source flux distribution model, spatially and spectrally folding in the imperfect response via the multiplication by the beam efficiencies of each detector on every pointing in the mapping pattern. The task requires a reference observation to obtain the raster pattern information and a source model. The source model can be input via:

- An existing task called `smallExtendedSpectralSource`, that models spatially a centred symmetric Gaussian function and spectrally a continuum with a first-order polynomial and a line with a Gaussian profile, whose amplitude is scaled to the continuum

- A user-provided jython class that extends the class *ExtendedSpectralSourceAdapter*. It must contain a method named "getFlux". An example of such a function is:

```

from herschel.pacs.spg.spec.beams import ExtendedSpectralSourceAdapter
from math import *

class MyGaussSpectral(ExtendedSpectralSourceAdapter):
    fwhm = 0.0
    offRa = 0.0
    offDec = 0.0
    contFlux = 0.0
    contSlope = 0.0
    lineAmplitude = 0.0
    lineCenter = 0.0
    lineWidth = 0.0

    def __init__(self, fwhm, offRa, offDec, contFlux, contSlope, \
                 lineAmplitude, lineCenter, lineWidth):
        self.sourceSize = fwhm/2.3548
        self.offRa = offRa
        self.offDec = offDec
        self.contFlux = contFlux
        self.contSlope = contSlope
        self.lineAmplitude = lineAmplitude
        self.lineCenter = lineCenter
        self.lineWidth = lineWidth

    def getFlux(self, relRa, relDec, wavelength):
        r = sqrt((relRa-self.offRa)**2 + (relDec-self.offDec)**2)
        spectrum = self.contFlux + self.contSlope * (wavelength-self.lineCenter)
        spectrum += self.lineAmplitude * \
            exp(-0.5 * ((wavelength-self.lineCenter)/self.lineWidth)**2)
        flux = spectrum * exp(-0.5 * (r / self.sourceSize)**2)
        return flux

    def getFluxImage(self, relRa, relDec, wavelength):
        r = sqrt((relRa-self.offRa)**2 + (relDec-self.offDec)**2)
        spectrum = self.contFlux + self.contSlope * (wavelength-self.lineCenter)
        spectrum += self.lineAmplitude * \
            exp(-0.5 * ((wavelength-self.lineCenter)/self.lineWidth)**2)
        self.fluxImage = spectrum * exp(-0.5 * (r / self.sourceSize)**2)
        return self.fluxImage

    def renderGaussLayer(self, size=109, dpix=0.5, wavelength=66.38):
        self.image=Double2d(size, size)
        for y in range( size ):
            for x in range( size ):
                xim = (-dpix)*(x-(size/2))
                yim = (dpix)*(y-(size/2))
                self.image.set(y,x, self.getFluxImage(xim, yim, wavelength))
        return self.image

    def getSourceModelImage(self, size=109, dpix=0.5, wavelength=66.38):
        return self.renderGaussLayer(size, dpix, wavelength)

sourceModel =
    MyGaussSpectral(sourceSize,offRa,offDec,contFlux,contSlope,lineAmplitude,\
                   lineCentre,lineWidth)
sourceModelImage = SimpleImage(image=sourceModel.getSourceModelImage(109, 0.5,
66.38))
d=Display(SimpleImage(image=sourceModel.getSourceModelImage(109, 0.5, 66.38)), \
          title='continuum layer Gaussian model')
d=Display(SimpleImage(image=sourceModel.getSourceModelImage(109, 0.5,
lineCentre)), \
          title='line layer Gaussian model')

```

Requirements: The comparison with the observed cube pipeline product must be in units of Jy/beam, thus it is necessary then to remove extended source correction factor first. This can be done via the function *undoExtendedSourceCorrection* if working on *PacsRebinnedCube* (which will be the case if

you are working on pointed observations, or wish to create model interpolated or projected cubes to compare to those created from your observation), or *undoExtendedSourceCorrectionSlicedCubes* if working with *PacsCubes* (which will be the case if you wish to compare model to observed drizzled cubes). To import these functions (so you can use them) can be done with:

```
from herchel.pacs.spg.spec.ExtractCentralSpectrumTask import *
```

In order to obtain consistent units of Jy/beam in the modelled PACS observation, a scaling factor of 100 must be multiplied to the final product of *PacsSpecFromModel* task. Scripts and explanation for this tool will be provided on the PACS documentation pages on the Herschel Science Centre web-pages, currently at herchel.esac.esa.int/twiki/bin/view/Public/PacsCalibrationWeb, and the Herschel Explanatory Legacy Library pages. You can also raise a HSC helpdesk ticket to request the "FMT" package.

Caveats: The task *PacsSpecFromModel* does not model wavelength shifts nor the skew effect in that occur in emission lines due to offset of point-like and semi-extended source from the centre of a spaxel. Note also that this task is memory greedy when used on mapping observations.

Chapter 9. The post-pipeline tasks for mapping observations and extended sources

9.1. Introduction

In this chapter you will find sections on the post-pipeline tasks for extended sources, observed with a single pointing or as a raster. The sections are:

- The native footprint of the PACS IFU and of the rebinned cubes: [Section 9.2](#).
- Tiling and pointed observations: [Section 9.3](#).
- Oversampled and Nyquist-sampled mapping observations: [Section 9.4](#).
- Correcting for the uneven illumination of the PACS IFU using the forward modelling tool: [Section 9.5](#).
- Fitting the spectra of extended sources in rebinned cubes, and then making mosaic images from the fitting results: [Section 9.6](#).
- Creating mosaic cubes with different spaxel sizes (but not re-reducing the data); combining separate observations into a single mosaic cube: [Section 9.7](#).

There are three tasks provided to create mosaic cubes for PACS mapping observations: `drizzle`, which is explained in [Section 7.9](#); `specInterpolate`, which is explained in [Section 7.11](#); `spectProject`, which is explained in [Section 7.10](#).

There is also a Useful script which can be used to create all of these cubes, *Scripts#PACS useful scripts#Spectroscopy: Post-processing for extended sources*, and you may want to look at the Useful script that can be used to create versions of these cubes with an equidistant wavelength grid, *Spectroscopy: Re-create the standalone browse products*.

9.2. The native footprint of the PACS IFU and sampling of the beam

The first of the Level 2/2.5 cubes created by the pipeline, and the earliest cubes that can be used for science (for any type of observation or observing mode) are the rebinned cubes: these can be found in the *contexts* called "slicedFinalCubes" or "slicedDiffCubes" in the pipeline scripts, or HPS3DR[R|B] in the *ObservationContext*. These cubes are of class *PacsRebinnedCube*. There is one cube for each pointing (for mapping observations) and wavelength range (where more than one range was requested for the observation).

The rebinned cubes have a sky footprint of the instrument, which is 5 spaxels along one direction and 5 spaxels along the perpendicular, with a slightly irregular grid:

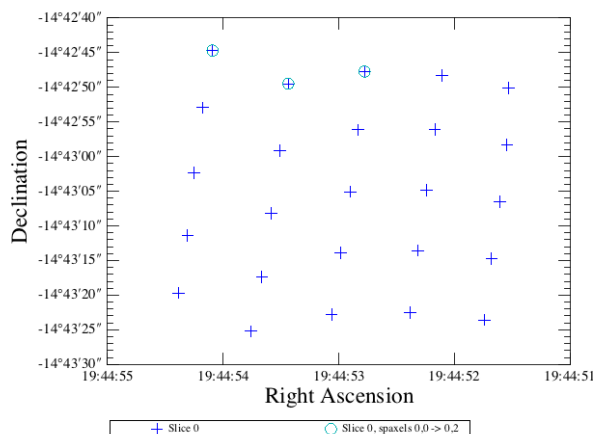


Figure 9.1. Sky footprint of the *PacsRebinnedCube* (and *PacsCube*)

A rebinned cube viewed in HIPE will display with a 5x5 regular grid: see the central image of [Figure 9.3](#). This is purely because the viewers cannot display images with an irregular grid. The rebinned cubes do have an RA, Dec array, but the spatial coordinates of the WCS is defined in spaxel coordinates: 0,0; 0,1 and etc., rather than in sky coordinates.

The physical on-sky spaxel size for the PACS integral field unit was chosen as a compromise between mapping efficiency (if the spaxels are very small, so will be the field-of-view) and sampling the beam (if the spaxels are too large, the spatial resolution will be terrible). Ideally the instrument should Nyquist sample the PSF (i.e. at least 2 to 3 spaxels should fit across the FWHM of the PSF), e.g.:

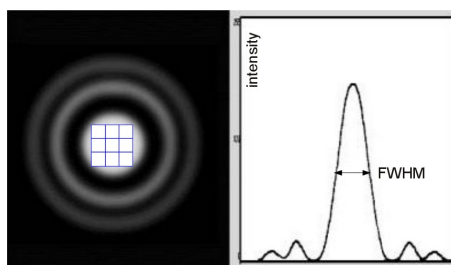


Figure 9.2. PSF Nyquist sampling

However, since the PACS spaxels are so large that 2-3 of them do not fit across the detector, any single pointing does not have Nyquist sampling, and these observations suffer the problems of *spatial undersampling*. Tiling observations, which are essentially a series of single pointings, suffer in the same way.

To overcome this undersampling limitation, a mapping AOT was developed: the field-of-view was observed with a raster pattern with sub-spaxel size pointing offsets, such that when the cubes are combined, you are then effectively properly sampling the beam. See the PACS Observer's Manual, chap. 6, on the HSC Documentation page [\[here\]](#) for an explanation of this mode.

9.3. Tiling and pointed observations

For pointed or tiling observations of extended sources, PACS recommends that the science measurements are made from the rebinned cubes. Since the PACS beam has not been fully sampled for these observations, the flux distribution of the observed source will always be an approximation of the true situation: think of it as the gaps in the spatial sampling corresponding to gaps in the collected signal.

For these observations we provide **interpolated cubes** (HPS3DI[R|B]), created with the task `specInterpolate`. This task creates a regular grid via Delaunay triangulation from the input grids of the rebinned cube(s), and the flux array is then interpolated from the input cubes onto this regular grid. The interpolated cubes are a good approximation of the observed patch of sky for these observations, but

the best spectra to measure for science are those of the rebinned cubes, where the flux array has not been subjected to any interpolation. The spaxel sizes of interpolated cubes are 3" in SPG 14 (4.7" was used in SPG 13). Interpolated cubes are also provided in the *ObservationContext* downloaded from the HSA for Nyquist-sampled mapping observations, but for mapping observations the projected or drizzled ones should be preferred for science.

We also provide **projected cubes** (HPS3DP[R|B]) in the *ObservationContext* for these observations. For the pointed mode, the spaxel sizes of projected cubes are 0.5" and for for tiling mode, they are 1.5". Whether you use the projected or interpolated cubes to browse through your observed field-of-view is a matter of personal preference: the interpolated cubes are easire to load into viewers because they have few spaxels, and the "images" from these cubes give a better view of extended sources. In addition, note:

- specProject is built on an algorithm that assume your datapoints, at each wavelength, are at least Nyquist sampling the beam. For any other type of observation, you could introduce artifacts into the resulting cube. For these undersampled observations this can be avoided by having very *small* spaxel sizes—0.5" pointed/1.5" tiling. In this way the original footprint of the rebinned cubes still stands out, and you get an image that looks like that of the rebinned cubes, but now with a regular grid. This is demonstrated in the figure below.
- specInterpolate is the recommended task to use for tiling observations. The SPG pipeline adopts a spaxel size of 3", but the best value for your observation depends on the wavelength of the data, the morphology of the source, and the details of the raster. Spaxel sizes larger than the 9.4" native size are not recommended, and spaxels of 3" .4.7" .6" are good value to try. In either case, the resulting fluxes and morphology are indicative, since these observations are all undersampled.

The spatial grid for interpolated cubes are created using the Delaunay triangulation and *interpolation* from the old spatial grid to the new one. There is no *extrapolation* done at the edges of the field: the mapped field is therefore smaller than that achieved by the other mosaicking tasks (this is especially obvious when looking at cubes from pointed observations).

- Please see [Section 7.8](#) for some general comments about mosaic cubes concerning the propagation of NaNs and flags.
- *Never run drizzle*. It is inappropriate for these cubes.

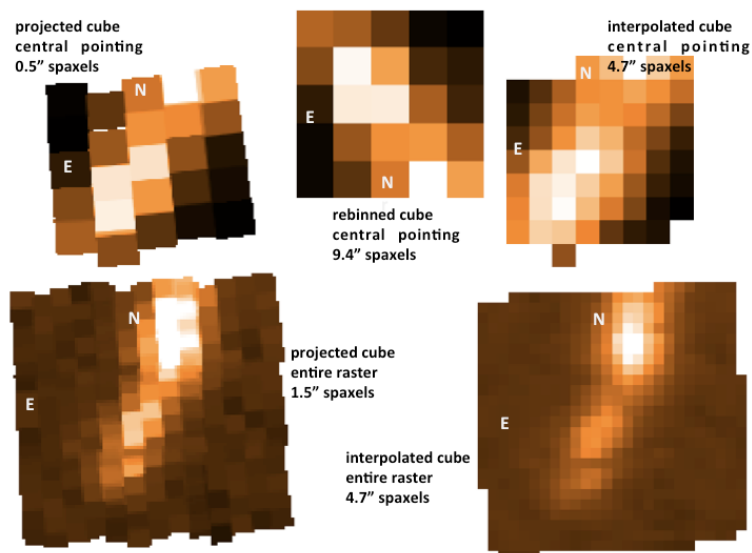


Figure 9.3. Some examples of cubes for a tiling observation. Top centre is the rebinned cube taken from the central position in the raster, with spaxels of 9.4" and the approximate N-E axes indicated. Left and right of that are the same cube but interpolated (right) and projected (left), with N-E indicated and spaxel sizes as you would get from an observation created by SPG 13. Bottom are the entire raster made into a mosaic with specInterpolate (right) and specProject (left), with N-E indicated and spaxel sizes as you would get from an observation created by SPG 13

9.4. Oversampled and Nyquist sampled mapping observation

The pipeline tasks `specProject` and `drizzle` are used on mapping observations that properly sample the beam. These tasks:

1. project the fluxes of the input cubes onto a regular sky grid, and so mosaic together the individual cubes of a mapping observation
2. take advantage of the mapping pattern to better sample and map the morphology of the source and recover the correct flux levels of extended sources in the field-of-view
3. create cubes with a WCS along the spatial axes: one expressed in sky coordinates, with RA and Dec
4. work on the image, error/weight, and flag datasets of the input cubes (and any other datasets that are present), these then being propagated into the output mosaic cube; NaN values are also propagated as NaN

The spatial-spectral projection that are the core of the tasks `specProject` and `drizzle` creates mosaic cubes from the rasters (HPD3DP[R|B] and HPS3DD[R|B]), with spaxels of 3" where only a projected cube is present, and a bit less if both a drizzled and a projected cube is present (and in which case, both cubes will have exactly the same WCS). (We also provide projected cubes for tiling observation (1.5" spaxels) and pointed observations (0.5" spaxels), although these cubes are mainly provided for ease-of-viewing.)

These cubes can be used for science measurements, since the algorithms used by `specProject` and `drizzle` are sufficiently sophisticated to ensure the fluxes and morphology are well represented in the mosaic cubes. `Drizzle` is to be preferred, but these cubes are only provided for line scan observations (a limitation of the task).

Please see [Section 7.8](#) for some general comments about mosaic cubes concerning the propagation of NaNs and flags.

9.5. Correcting for the uneven illumination: the forward modelling tool

The spaxels of the PACS IFU are not evenly illuminated, with the result that there is effectively some flux loss between the spaxels over most of the PACS wavelength range. To correct for this an extended source correction was created, and this is applied to *all* observations by the pipeline. This fully corrects the integrated fluxes for all fully-extended sources (those with a flat or gradual flux gradient of no more than 20% within a single 47" square FoV). However, the uneven illumination of the PACS IFU FoV affects the flux distribution in the rebinned cubes and the spectral calibration in surface brightness. This may result in an incorrect integrated spectrum extracted from an aperture in spectral cubes that contain:

- crowded fields
- off-centred point sources
- semi-extended structures (especially if not extracted using the provided tools: [Section 8.6](#))
- sources which are extended but with flux gradients

The inaccuracy in the flux will depend on the morphology of the source at the wavelength of your observation and its coupling to the detector's beam efficiencies on every pointing (mapping) pattern. If you do aperture photometry on your source (from small sources to the entire field), you could end up under- or over-estimating the flux in the aperture. To estimate the inaccuracy in the flux, if you know

(or can model/estimate) the surface brightness distribution of the source (i.e. its morphology at the wavelength of your observation) and can provide a model spectrum, you can apply a "forward projection" in HIPE: this takes in your input surface brightness model/image, and working with the pointing/mapping pattern of your observation, it produces result which folds in the uneven illumination.

9.5.1. Description of the "forward modelling" process

The task *pacSpecFromModel* simulates PACS cubes according to a user-provided source flux distribution model, spatially and spectrally folding in the imperfect response via the multiplication by the beam efficiencies of each detector on every pointing in the mapping pattern. The task requires a reference observation to obtain the raster pattern information and a source model. The source model can be input via:

- An existing task called *smallExtendedSpectralSource*, that models spatially a centred symmetric Gaussian function and spectrally a continuum with a first-order polynomial and a line with a Gaussian profile, whose amplitude is scaled to the continuum
- A user-provided jython class that extends the class *ExtendedSpectralSourceAdapter*. It must contain a method named "getFlux". An example of such a function is:

```
from herchel.pacs.spg.spec.beams import ExtendedSpectralSourceAdapter
from math import *

class MyGaussSpectral(ExtendedSpectralSourceAdapter):
    fwhm = 0.0
    offRa = 0.0
    offDec = 0.0
    contFlux = 0.0
    contSlope = 0.0
    lineAmplitude = 0.0
    lineCenter = 0.0
    lineWidth = 0.0

    def __init__(self, fwhm, offRa, offDec, contFlux, contSlope, \
                 lineAmplitude, lineCenter, lineWidth):
        self.sourceSize = fwhm/2.3548
        self.offRa = offRa
        self.offDec = offDec
        self.contFlux = contFlux
        self.contSlope = contSlope
        self.lineAmplitude = lineAmplitude
        self.lineCenter = lineCenter
        self.lineWidth = lineWidth

    def getFlux(self, relRa, relDec, wavelength):
        r = sqrt((relRa-self.offRa)**2 + (relDec-self.offDec)**2)
        spectrum = self.contFlux + self.contSlope * (wavelength-self.lineCenter)
        spectrum += self.lineAmplitude * \
            exp(-0.5 * ((wavelength-self.lineCenter)/self.lineWidth)**2)
        flux = spectrum * exp(-0.5 * (r / self.sourceSize)**2)
        return flux

    def getFluxImage(self, relRa, relDec, wavelength):
        r = sqrt((relRa-self.offRa)**2 + (relDec-self.offDec)**2)
        spectrum = self.contFlux + self.contSlope * (wavelength-self.lineCenter)
        spectrum += self.lineAmplitude * \
            exp(-0.5 * ((wavelength-self.lineCenter)/self.lineWidth)**2)
        self.fluxImage = spectrum * exp(-0.5 * (r / self.sourceSize)**2)
        return self.fluxImage

    def renderGaussLayer(self, size=109, dpix=0.5, wavelength=66.38):
        self.image=Double2d(size, size)
        for y in range( size ):
            for x in range( size ):
                xim = (-dpix)*(x-(size/2))
                yim = (dpix)*(y-(size/2))
                self.image.set(y,x, self.getFluxImage(xim, yim, wavelength))
```

```

        return self.image

    def getSourceModelImage(self, size=109, dpix=0.5, wavelength=66.38):
        return self.renderGaussLayer(size, dpix, wavelength)

sourceModel =
MyGaussSpectral(sourceSize,offRa,offDec,contFlux,contSlope,lineAmplitude,\
    lineCentre,lineWidth)
sourceModelImage = SimpleImage(image=sourceModel.getSourceModelImage(109, 0.5,
    66.38))
d=Display(SimpleImage(image=sourceModel.getSourceModelImage(109, 0.5, 66.38)), \
    title='continuum layer Gaussian model')
d=Display(SimpleImage(image=sourceModel.getSourceModelImage(109, 0.5,
    lineCentre)), \
    title='line layer Gaussian model')

```

Requirements: The comparison with the observed cube pipeline product must be in units of Jy/beam, thus it is necessary then to remove extended source correction factor first. This can be done via the function *undoExtendedSourceCorrection* if working on *PacsRebinnedCube* (which will be the case if you are working on pointed observations, or wish to create model interpolated or projected cubes to compare to those created from your observation), or *undoExtendedSourceCorrectionSlicedCubes* if working with *PacsCubes* (which will be the case if you wish to compare model to observed drizzled cubes). To import these functions (so you can use them) can be done with:

```
from herschel.pacs.spg.spec.ExtractCentralSpectrumTask import *
```

In order to obtain consistent units of Jy/beam in the modelled PACS observation, a scaling factor of 100 must be multiplied to the final product of *PacsSpecFromModel* task. Scripts and explanation for this tool will be provided on the PACS documentation pages currently at herschel.esac.esa.int/wiki/bin/view/Public/PacsCalibrationWeb, and the Herschel Explanatory Legacy Library pages.

Caveats: The task *PacsSpecFromModel* does not model wavelength shifts nor the skew effect in that occur in emission lines due to offset of point-like and semi-extended source from the centre of a spaxel. Note also that this task is memory greedy when used on mapping observations.

9.6. Fitting spectral cubes, and making maps from the results

In the HIPE Scripts menu we have provided a series of scripts with which you can fit PACS spectra and make images from the fitting results. There are currently three such scripts:

1. *Spectroscopy: Fitting mapping observations (mosaic cubes)*. Working from the interpolated, drizzled, or projected cubes. Fit a spectral line in the cubes, and create images from the fitting results, e.g. integrated flux and velocity.
2. *Spectroscopy: Fitting mapping observations (pre-mosaic cubes)*. Working from the rebinned cubes for a mapping observation. Fit a spectral line in each cube, then mosaic together the fitting results to create mosaic images.
3. *Spectroscopy: Fitting single pointing cubes*. Working from the interpolated cubes, fitting a spectral line and making qualitative maps of the fitting results.

These scripts are aimed at extended sources. Each script includes a description of its purpose and what the input cubes are, and they all start from a public observation that you can test them on. The aim of these scripts is that you can learn how to fit PACS spectral cubes on the command line, and you can learn what you can do with the fitting results afterwards. You can then use these scripts as a seed for your own fitting scripts.

The difference between script 1 and 2 is that for the first, you fit a single mosaic cube and create fitting images therefrom, for the second you fit the individual rebinned cubes of the raster, and then mosaic

the fitting results (i.e. a 2d mosaic, rather than a 3d mosaic). Which approach to prefer is a personal choice, as the work involved is slightly different. For both, the fitting is semi-interactive: for each cube you fit you have to choose a reference spaxel, fit its spectrum with your model (Gaussian+polynomial) and check and refine the fit, and then fit the entire cube with that refined model. Observations of sources that vary a lot over the field-of-view may produce better results with approach 2, because the reference spectrum chosen from each single 5x5 rebinned cube may be more representative of the other spectra in that cube, than a single reference spectrum chosen from an entire raster—but this is not a hard and fast rule. The first two scripts have the same observation as the example to run on, and you can compare the results.

The process of fitting your cube and turning the fits results into maps uses a product called a *PacsParameterCube*, or more generally a *ParameterCube*. This is not a cube in the sense of a dataset with axes RA, Dec, and wavelength, but rather a 3d product that holds fitting results for defined models performed on spectral cubes.

To learn more about fitting, we refer you to the *DAG* [chap. 7](#). The command-line fitting in the PACS scripts use the Spectrum Fitter, which is the heart of the Spectrum Fitter GUI, and both are explained in detail in the *DAG*.

9.7. Changing the spaxel size of SPG cubes; mosaicing unrelated observations

Two other scripts are provided in the Scripts menu of HIPE. The scripts are themselves well documented.

1. *Spectroscopy: Create mosaic cubes with any spaxel size*. This script contains what are essentially the final stages of the pipeline, and it is aimed at those who do not wish to re-reduce their data, but simply create a mosaic cube with a different spaxel size to that provided via the HSA. The most common use-case is probably for drizzled cubes: the spaxel size is dependent on the wavelength, and hence cubes of different ranges but from the same observation will have slightly different spaxel sizes.
2. *Spectroscopy: Mosaic multiple observations*. This script is for those who have separate observations which cross over spatially and spectrally, and for which they wish to create a single mosaic cube from the data from all obsids. The use-case here is rare (since the HSC did not like to have the same part of the sky observed with the same spectral settings), but some observations in the archive do match this scenario.

Chapter 10. Plotting and inspecting PACS data

10.1. Introduction

In this chapter you will find:

- An explanation of the pipeline helper tasks ([Section 10.2](#)), including saving slices to disc, and plotting: plot the instrument movements v.s. the time-line spectrum; the spectra of the pipeline products—*Frames* and the various cubes—in various useful ways; the standard deviation (a measure of the spectral error); the pointing (including overplotting on an image).
- Working with sliced products ([Section 10.3](#)).
- How to plot PACS spectral data subjected to various selections, so you can focus on checking one aspect of your data at a time ([Section 10.4](#)); in the course of which you will learn something about how PACS data are organised.
- How to use the Spectrum Explorer to explore PACS *Frames* and cubes spatially and spectrally ([Section 10.5](#)).
- The PACS IFU (integral field unit) sky footprint ([Section 10.6](#)).
- The PACS footprint viewer: a cube footprint overlaid on an image ([Section 10.7](#)).
- PACS product viewer, to inspect Level 0 and 1 data along the time-line ([Section 10.8](#)).
- A quick guide to comparing the off-source and on-source cubes for unchopped mode observations, starting from an observation gotten from the HSA ([Section 10.9](#)).
- How to know the PACS spectral resolution at a given wavelength ([Section 10.10](#)).
- Working with masks ([Section 10.11](#)) [this has an advanced use-case].
- Working with the Status and Block tables ([Section 10.12](#)) [this has an advanced use-case].

10.2. Pipeline helper tasks explained

Here we briefly explain the pipeline helper plots. Examples of their use will be found in the pipeline scripts, and while many have been taken out of the Track 13 pipelines, they are still in the build: consult the PACS [URM](#) entries to see the full set of task parameters and how to use them.

- **obsSummary**: this provides a human-readable table summarising the observing details and the data layout. It is run on the entire *ObservationContext*:

```
obsSummary(obs)
```

The listing you will see is taken from the Meta data, and includes proposal information, an instrument configuration summary, and a table of the different defined parts of the observation, such as: wavelength regions covered, any flux information that the proposer put in HSPOT when defining the observing request, and whether any of the data is OUTOFBAND.

OUTOFBAND is a mask created as you pipeline process your data, and it indicates that some (or all) of the spectral region covered in a dataset is outside of the filter function of that camera and

band. Why this would happen: if the observer defined a blue camera region to cover a particular line, and if the accompanying spectrum from the red camera happens to fall outside of the red filter band (partially or completely), those data are OUTFORBAND and will be unusable.

- **saveSlicedCopy** and **readSliced** are tasks that can be used to save sliced products—which are *contexts* rather than single files—to a "pool" on disc. The products saved to disc are saved as FITS files, but they need to be organised in a certain way so that HIPE can understand the relationship between them when it reads the *context* back in. The use of these tasks is demonstrated in the pipeline scripts, and they are used thus:

```
name = nameBasis+"_slicedCubes"
saveSlicedCopy(slicedCubes, name, poolLocation=outputDir)
slicedCubes = readSliced(name, poolLocation=outputDir)
```

The "slicedCubes" is saved to a disc location [outputDir]/name and within there the data are organised as FITS files. See their [URM](#) entries to learn more about the use of these tasks.



Note

One thing to note: if you wish to grab these data directly from disc, rather than via "readSlices", and are used to inspecting PACS data on disc (e.g. HSA tarball downloads) note that the organisation of the FITS files here is slightly different and the file-names will not reflect the level and product class as they should otherwise. Hence, to be sure you know what you are doing, if you use `saveSlicedCopy` to write to disc, use `readSliced` to read back in.

- **slicedSummary**: gives a printed summary of each slice in the pipeline sliced products (e.g. the "slicedFrames", or "slicedCubes") including wavelength information (line, band, ranges) and pointing (raster, nod, nodCycle). In the beginning of the pipeline there is only one slicedFrames slice, then there are several as the data are sliced, then towards the end of the pipeline there are fewer as data are combined. An example output of a slicedFrames from Level 0:

```
HIPE> slicedSummary(slicedFrames)
noSlices: 1
noCalSlices: 0
noScienceSlices: 1
slice# isScience nodPosition nodCycle rasterId lineId band      dimensions
...
0      true      [""]      0      0 0      [0]      ["B2","B3"]
[18,25,12976] ...
```

The columns are: slice number; whether the slice contains any science data (True), or only calibration data (False); the nod position (A|B); the nod cycle order number; the raster sequence; the line identification number (differs with wavelength range); the band; the data dimensions; the wavelength range; and whether the data are on or off source, or both.

- **slicedSummaryPlot**:

```
slicedSummaryPlot(slicedFrames)
```

This will plot a summary of the instrument movements during an observation, with an optional timeline spectrum over-plotted. Run on a slicedFrames from Level 0 will show the grating movements in engineering units with the signal from the central pixel of the detector over plotted, taken from the central pixel (8) of the central module, which is the very centre of the field-of-view of PACS. The X-axis is readout, and moving along the axis moves you along in time. From this plot you can see how your source signal varies with grating position (with wavelength). The parameter `signal=1` or `0` can be used to chose whether to plot the signal or not.

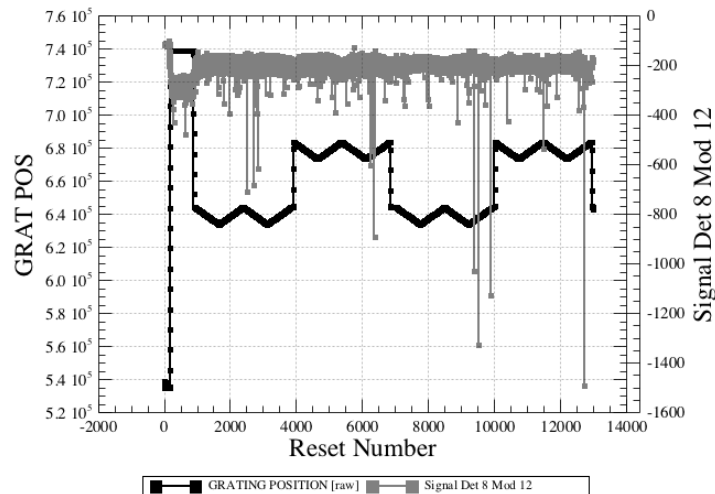


Figure 10.1. The plot from `slicedSummaryPlot` run at the very beginning of the pipeline: grey is the signal, black are the grating movements. Each grating scan is done twice, in opposite directions, creating a V shape, and the 2 sets of Vs in each nod (A and B) because there are two wavelength ranges in these data. The signal is noisy due to the presence of glitches. The block of data on the very left is from the calibration block. When run later in the pipeline there will be additional curves on the plot.

Run on a `slicedFrames` from Level 0.5/1, after more pipeline tasks have been run on the data, more/other information is shown: wavelength (in microns), unitless numbers that represent the calibration source position (0=looking at the source, 1,2=calibration source 1 or 2), band (4 of them), nod A or nod B (0=not in that nod, 1=are in that nod), and the "gratscan", the grating scan counter (0,1,2,... and + or - depending on which direction along the grating one is moving).

- **slicedPlotPointing:**

```
slicedPlotPointing(slicedFrames, plotBoresight=False) # or True
```

This will plot the position of each spaxel in the nod A (on- and off-source) and nod B (off- and on-source) positions; the "boresight", which for PACS is roughly the centre of the detector at chopper 0; and the programmed source position taken from the Meta data. Note that for solar system objects, this may mean that your source falls off of the centre of the pointing cluster, since the Meta data position does not include solar system coordinate corrections.

By default only the last slice will be plotted (though you can ask for all). Since there is only one slice in the first, Level 0, `slicedFrames` you work with in the pipeline, for this product all the pointing movements are plotted, and you will see a long grey track, with a cloud of coloured points at one end of it. Zoom in on this cloud to see the footprint of the IFU at the various chop and nod positions during that slice of the observation.

The task also prints the Position Angle (in degrees and radians) to the HIPE *Console*.

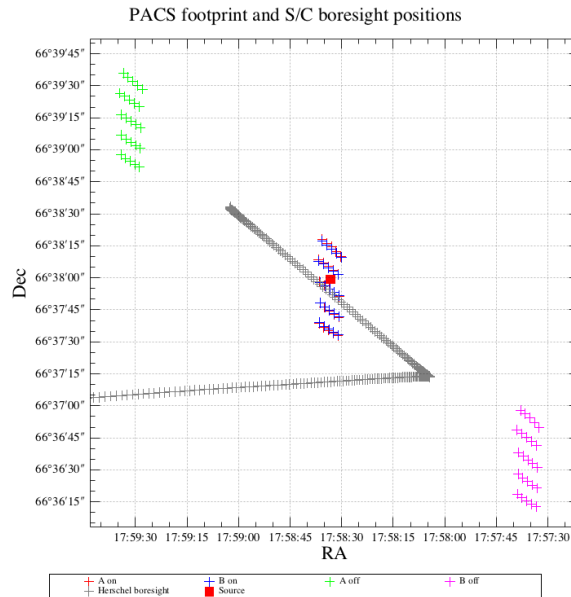


Figure 10.2. The plot from `slicedPlotPointing`. The grey curve shows the end of the slew on to the source, and the coloured crosses are labeled.

- **`slicedPlotPointingOnOff`:** Does the same as the previous task, but for the unchopped observations.

```
slicedPlotPointingOnOff(slicedFrames)
```

- **`maskSummary`:** Lists all the masks in all slices and their status: 1 for active (will be considered), and 0 for not.

```
maskSummary(slicedFrames) # or slicedCubes
```

- **`plotSignalBasic`:** This task plots the spectrum of a single slice at a time.

```
plotSignalBasic(slicedFrames, slice=0, titleText="something")
```

Plotted is 5h3 spectrum of the central pixel (8,12) for all unmasked, i.e. good data (those for which the masks have been activated). See its [URM](#) entry to learn how to plot other parts of your observation, e.g. other pixels.

When looking at your spectrum, bear in mind that the off-source and on-source data may both be plotted, together. See [Section 10.4.3](#) to learn how to separate the two when plotting. Alternatively, wait until the off-source data have been subtracted (in the `chopNod` pipeline this is done by the task `specDiffChop`, in the unchopped pipelines it is not done until later in the pipeline, and `plotSignalBasic` does not work on those data).



Tip

The PACS spectrometer detector array has a size of 18,25, with science data being contained in pixels 1 to an including 16 (out of 18) only.

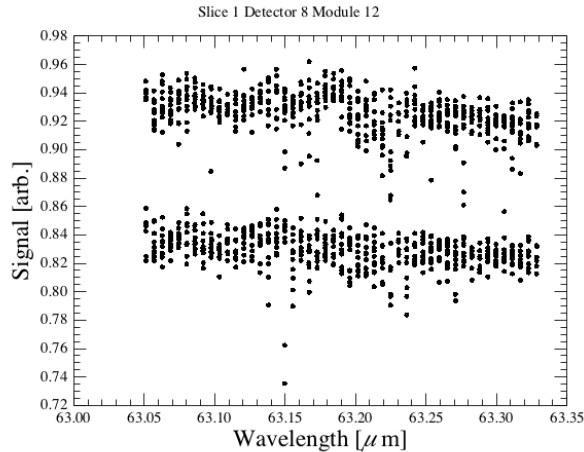


Figure 10.3. The plot from `plotSignalBasic`, where you can see the off (lower) and on (higher) line of datapoints

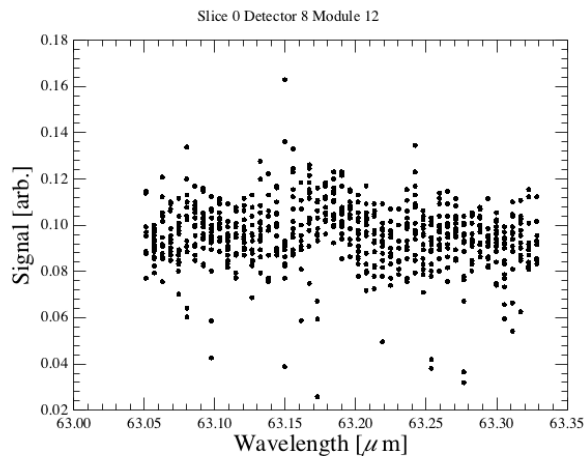


Figure 10.4. The plot from `plotSignalBasic`, after the off signals have been subtracted

- **plotPixel:**

```
slice = 0
x,y = 2,2
plotPixel(slicedCubes.get(slice), x=x, y=y, ...)
```

This task is particularly useful to use before and after the flatfielding and the transients correction tasks. It will plot all the pixels of a chosen spaxel of a *PacsCube*, in a different colour. You can chose (`rebin=1`) to overplot on each pixel's dot cloud a curve that is the rebinned version of the same spectrum (with a rough rebinning, not that which is done later in the pipeline). The dots show the original data, but with the curve you can see the spectrum more clearly. The URM is this time not so helpful to see a full listing of the parameters of this task, instead type:

```
print plotPixel.__doc__
```

It can be difficult to see the curve over the dots for some spectra, even when you do isolate a single spaxel, but (i) if you right-click on the plot to access the Properties, you can go to the Style location for each layer (each spectrum is a new layer) and change e.g. the line width or dot size, and (ii) you can use the `offset` parameter of the `plotPixels` task to offset the individual spectra from each other by an additive amount. Or, of course, you can zoom on the plot.

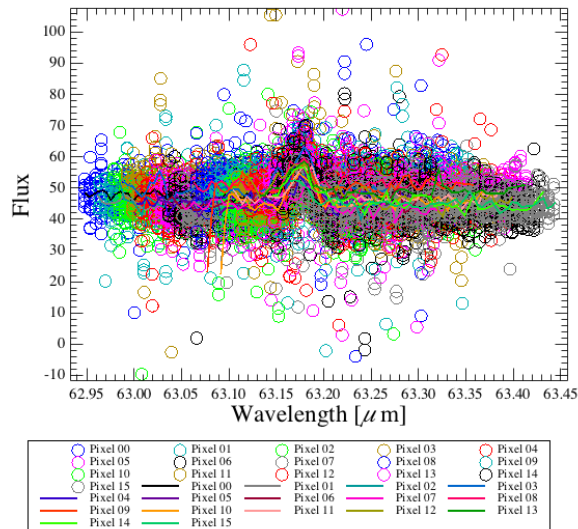


Figure 10.5. The plot from `plotPixels`, showing the *PacsCube* data in dots and a rebinned version of those data as a curve.

- **plotCubes**: plots the spectrum of the central spaxel of all slices in a *SlicedPacsCubes*, including only data that are good, as identified in the masks that are specified in the call. The different slices come up in different colours. The plot caption tells you what colour is what:

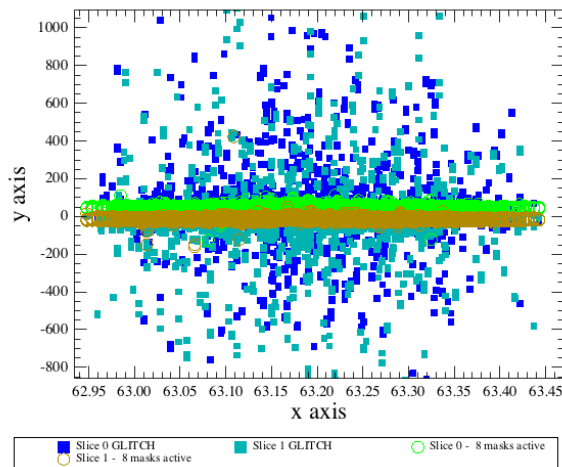


Figure 10.6. `plotCubesMasks` bad data (blue squares) over plotted with `plotCube` good data (brown/green circles)

There are parameters to allow over-plotting in different colours, stroke widths, and whether to plot single spaxels or the sum of several. See [URM](#) entry for this task.

To plot one slice only (creating an easier-to-read plot) then change the input from a *SlicedPacsCube* containing all the slices to one containing only a single slice, using `selectSlices`

```
x,y=2,2 # centre of the IFU
masks = String1d(["GLITCH"])
p1 = plotCubes(slicedCubes, x=x, y=y, masks=masks)

temp = selectSlices(slicedCubes,sliceNumber=[0])
plotCubesMasks(temp)
```


If you run the task using the first example above, the output of this task is put in a plot variable "p1". Some subsequent tasks can overplot on this, for example: `plotCubes` (see below) which can plot all the good data. In this way you can see the good from the bad.

Used together with `plotCubesMasks`, you can overplot good and bad data.

```
x,y=2,2 # centre of the IFU
masks = String1d(["GLITCH"]) # to plot only these bad-masked data
p1 = plotCubesMasks(slicedCubes, x=x, y=y, masks=masks)

# get all the mask names that are active
masks = slicedCubes.get(0).getActiveMaskNames()
# overplot on p1 all the good datapoints
p2 = plotCubes(slicedCubes, x=x, y=y, masks=masks)
```

- **plotCubesMasks** plots the spectrum of the central spaxel of all slices in a *SlicedPacsCubes*, including only data that are bad, as identified in the masks that are specified in the call. See `plotCubes` (above) to learn more.
- **plotCubesRaDec**:

```
overlay = None # or 'WISE', 'MSX', obsid, image name
plotCubesRaDec(slicedRebinnedCubes, subtitleText="something", overlay=overlay)
```

This task will plot the RA and Dec of the *PacsRebinnedCube* slices, towards the end of the pipeline. The values are taken from the "ra" and "dec" datasets of the cubes (created by the pipeline task `specAssignRaDec`). The different coloured crosses are the different cubes, these should be nod A and nod B at all the raster pointings you have, if both nods are present in the plotted product.

Note: it is normal to see a slight offset between the nods (this is found for all observation), and that this offset varies with spaxel (it will be least in the centre of the cube). You will also be able to see clearly the slightly irregular spacing in the sky footprint of these cubes.

Three spaxels are circled in the plot: these are the spaxels 0,0; 0,1; 0,2. See [Section 10.6](#) for an explanation of the footprint of PACS and the relationship between module numbering and spaxel numbering and [Section 10.4.6](#) to see how to plot the sky footprint of PACS.

If you want to plot only a single slice (to make the plot easier to read), then you can select the slice out of the *ListContext* within the call to the task, using the `selectSlices` task, e.g.:

```
p9 = plotCubesRaDec(selectSlices(slicedRebinnedCubes, \
    sliceNumber=[0]))
```

It is possible to plot the pointing on a image of that patch of the sky using the parameter `overlay`. Choosing "WISE" or "MSX" will result in a search in those archives for an image at the position of the observation. Specify instead an obsid (1342... number) to plot on an image taken from a PACS photometry observation of that position on the sky. Or if you already have an image loaded in HIPE, give its name that will be shown.

- **plotCube5x5** will create a 5x5 panel display of the spectra from each spaxel. Such plots are useful to get a broad view on a rebinned cube, ideally the last one of the pipeline

```
plotCubes5x5(slicedFinalCubes, frameTitle = "something")
```

- **plotCubesStddev**: This is a useful task to produce a plot that you can use to inspect the standard deviation and continuum RMS datasets, for a single spaxel, for all slices (or only one) for *PacsRebinnedCubes*. This should be done at the end of the pipeline, when you have very little left to do to the data.

```
plotCubesStddev(slicedFinalCubes, plotLowExp=1, plotDev=0, \
    nsigma=3, isLineScan=-1, spaxelX=x, spaxelY=y, verbose=1, \
```

```
calTree=calTree, wranges=None)
```

The parameter `isLineScan` determines whether the task is used in a way appropriate for line scans (1), rangeScans (0), or either (the task determines this itself by looking at the Meta data to find out what whether the observation is line or range). Consult its [URM](#) entry for a full listing of the parameters.

The main plot will be embedded in the *Editor* pane of HIPE. In black is the spectrum, red dots are low-exposure bins (they should only be at the edges, otherwise you set a bad wavelength grid when running the task `wavelengthGrid`), and a green and a blue line are noise spectra: RMS and stddev. In green are also plotted the spectral ranges used to estimate the RMS.

The blue is the stddev dataset, this having been created by `specWaveRebin` and which is propagated by `specAddNodCubes`. It is a measure of the standard deviation in each bin, based on the datapoints from the *PacsCubes* that created the *PacsRebinnedCubes* you are looking at now. The green is the RMS dataset taken from the spectrum of the *PacsRebinnedCubes*. Both will be multiplied by a factor that you set in the task, by default this is 3 (i.e. 3-sigma curves). **See [Section 7.7](#) for a more detailed explanation of the difference between stddev and the RMS spectra, and to understand the limitations of this task.**

The task also sends some text to the Console, listing the values of the continuum RMS in Jy and W/m²: the best value and that at the maximum coverage wavelength.

If you wish to run this task coming directly from the *ObservationContext*, rather than the pipeline, then you need to extract the same input product from the observation:

```
slicedFinalCubes = obs.level2.red.rcube.product # or blue
# also necessary
calTree=getCalTree()
```



Tip

For too-short ranges scans the smoothing done `plotCubesStddev` may not work, in this case you need to change the filter used, the smoothing value, or use the line scan version instead (`isLineScan=1`).

- **plotTransient** is used in the unchopped pipeline scripts to plot data in such a way that long-term transients which often occur at the beginning of an observation stand out.

```
slice=1
module=12
plotTransient(slicedFrames, slice=slice, module=module, updown=1..)
```

In the pipeline scripts you can see an example of this task with added colours and legends.

The task will plot the spectrum of the chosen module, but normalised to the signal at the end of the observation (the final "scan")

The parameters `slice` and `module` have the same meaning that they do in all other tasks: which cube "slice" to plot and which module (from 0-24) to plot the spectrum of. The parameter `updown` sets whether the normalisation is to the entire final scan (up the grating and down again) or only the final up or down part of the scan (the signal differs intrinsically between up and down, hence this choice). Consult the [URM](#) entry for this task for a full listing of the parameters.

- **plotLongTermTransientAll** is used in the new transient correction pipeline scripts for unchopped observations of line scan mode.

```
# necessary, if not done
calTree=getCalTree()
module = 12 # for the central module
plotLongTermTransientAll(slicedFrames, obs=obs, step=0, \
    module=module, calTree=calTree)
```

The task plots the signal of a spaxel as a function of the time from the beginning of the observation. The signal is normalised to the expected background telescope flux. The off- and on-position data are plotted as blue and green points. In the pipeline script the task is run twice, the first plotting the data (the example above) and the second time overplotting the result of the long-term transient fit as a red curve. Consult the [URM](#) entry for this task for a full listing of the parameters.

- **plotPointingOffset** from the "Pointing offset correction (point sources)" pipeline script. This task plots the measured centre of the point source with respect to the centre of the central spaxel, as determined by the unique tasks of this pipeline, in unit of arcsec. When you run this task, the mean values are also printed to console.

10.3. Interacting with sliced products

There are various methods to interact with the sliced products (see the [PACS DRM](#) entries for the particular product to learn more). For the average user, the most commonly-used methods are:

```
# SlicedFrames, SlicedPacsCube, SlicedPacsRebinnedCube: extract a slice
# The example is from a sliceFrames, but replace "Frames" with "PacsCube"
# or "PacsRebinnedCube" for the other products

# Get a slice number "slice" (where 0 is the first slice)
# The product returned is a Frames
slice = 1
frames = slicedFrames.get(slice)

# Get science slice (i.e. excluding the calibration slices)
# number "slice". The product returned is a Frames
frames = slicedFrames.getScience(slice)

# Another method to get slice number "slice"
# The product returned is a Frames
frames = slicedFrames.refs[slice].product

# Get a slice another way
# The product returned is a SlicedFrames
slicedFrames_sub=slicedFrames.selectAsSliced(slice)

# Using a task; see the URM to see all the parameters
# you can select on. A SlicedFrames is returned
# Here it is possible to specify more than one slice number for sliceNumber
slicedFrames_sub = selectSlices(slicedFrames, sliceNumber=[0,1,3])

# replace, add etc (will work for all PACS sliced products)
# Adds a new frame to the end
slicedFrames.add(myframe1)
# Inserts a new frame before the given slice no. (1 here)
slicedFrames.insert(1,myframe1)
# Removes slice no. 1 (which is the second slice, not the first)
slicedFrames.remove(1)
# Replaces slice no. 1 with a new Frames
slicedFrames.replace(1,myframe)

# concatenate any number of SlicedXXX using a task
sliceFrames_big = concatenateSliced([slicedFrames1, slicedFrames2])

# ListContext: the SpectralSimpleCube context called
# slicedDrizzed|Projected|InterpolatedCubes
# The result is held in a basic ListContext
slice = 1
# The product returned is a SpectralSimpleCube
pcube = slicedPCubes.refs[slice].product
# Add a new cube on to the end
slicedPCubes.refs.add(ProductRef(newPCube))
# Add a new cube to a particular position (the first, position 0)
slicedPCubes.refs.add(1, ProductRef(newPCube))
# Remove a cube from a particular position
```

```
slicedPCubes.refs.remove(1)
```

For the full details of the methods that work on *SlicedFrames* etc. (including the new spectrum tables), look up the class names in the [PACS DRM](#). For the *ListContexts* of *SpectralSimpleCubes* you need to consult the [HCSS URM](#). You can also extract slices from a *SlicedXXX* list in a more varied way by selecting on the *MasterBlockTable*: see [Section 10.12](#) for a guide.



Tip

As with all other products in HIPE, their names are not the actual "thing" itself, but rather a pointer to the address in memory where the "thing" is held. That means that if you extract a slice out of a *SlicedXXX* list and change it, you are not only changing the copy, but also the original, as they both point to the same address.

Thus, if you want to work on a copy and not have any changes you make be done also to the original, you need to make a "deep copy" of the original by adding `.copy()` to the end of the sentence:

```
# A deep copy of a SlicedFrames
slicedFrames_new = slicedFrames.copy()
```

This will work for the *SlicedFrames*, *SlicedPacsCubes*, and *slicedPacsRebinnedCubes*, as well as for the *Frames*, *PacsCube*, and *PacsRebinnedCube*. It will work in the *Spectral-SimpleCubes* also, but not for the *ListContext* that contains these (projected, drizzled, or interpolated) cubes.

The pipeline task `selectSlices`, which reads a *SlicedXXX* in and writes a *SlicedXXX* out, also creates a fully independent product.

10.4. Plotting PACS spectral data yourself: how; and what are you looking at?

In the pipeline scripts we occasionally direct you to the **Spectrum Explorer** to inspect your PACS data. The Spectrum Explorer can be used to inspect *Frames* and all the cubes products in a friendly way. Its use with PACS Level 0.5 and 1 *Frames* and *PacsCubes* is explained in this PDRG, in [Section 10.5](#). To learn how to use the Spectrum Explorer on Level 2 cubes, see instead the *DAG* [chap. 6](#). In the pipeline scripts we also have various pipeline "helper" plotting tasks that allow you to see your data at various stages of the pipeline; these were explained in the previous section of this chapter.

In this section we describe the more "manual" methods to plot data using PlotXY, and at the same time we will introduce you to some of the things you may want to check as you run the pipeline. See the *DAG* [chap. 3](#) to read up on PlotXY—we explain here how to use it rather than what it is.



Note

If you are coming to this section directly from one of the pipeline chapters, you need to know that you can use PlotXY on one *Frames* or cube at a time.

To extract the *Frames* from the *slicedFrames* product that the pipeline runs on:

```
myframe=slicedFrames.get(0) # get the first slice
```

To know how many "myframes" you can extract:

```
print slicedFrames.getNumberOfFrames()
```

Bear in mind that once the data have been sliced by the pipeline task `pacSliceContext`, the first slice will include only calibration block datapoints. Then, at Level 1, the calibration slice is removed.

The same syntax will work for *PacsCubes* and *PacsRebinnedCubes*, where for these the calibration block slice will have been removed, leaving only science slices.

For the *SpectralSimpleCubes* you use this syntax instead:

```
cube=slicedProjectCube.refs[0].product # get the first one
```

See [Section 10.3](#) to learn more about selecting out slices.

10.4.1. General considerations: what difference the Level makes

A few general considerations when you plot PACS spectra. The spectrum plot you see will naturally depend on what type of observation you are looking at and at what pipeline stage your data are at. Before the end of Level 0.5 there is no slicing done, and one consequence of this is that the data contain the calibration block as well as all the wavelength ranges and raster pointings and repeats that you requested in the AOT. Hence if you plot all the data you could see a very messy-looking spectrum.

- The calibration source data are taken at the key wavelengths only, and so when plotted against wavelength (rather than time/array position) they form a small bunch of points at one wavelength, which may not fall within the wavelength range requested for your science.
- Data from multiple raster pointings will cover the same wavelength range as each other but the spectral features will probably look different.
- Repeats on grating scan and nod will produce plotted spectra with slightly different baseline levels.
- If the on- and off-source spectra are both present, two spectra of a different baseline level will be plotted.

As you proceed along the pipeline you slice on nod, repeat, pointing, wavelength, slice out the calibration block, and subtract the off- from the on-source data: all of this will change the plotted spectrum. But note that the repeated grating scans (there will always be at least 2 grating scan runs, one up and one down the wavelength range) will always have a slightly different base-level.

What will a Level 0, 0.5 or 1 spectrum look like?: In a spectrum taken from a *Frames*, each of the 16 active pixels (numbers 1 to 16 inclusive; 0 and 17 are not useful) that feed into each spaxel sample a wavelength range that is slightly shifted with respect to the next pixel. Hence if you over-plot several pixels of a single module (e.g. 1 8 and 16 of module 12) for a line scan AOT (after the wavelength calibration has been done), you will see something similar to this:

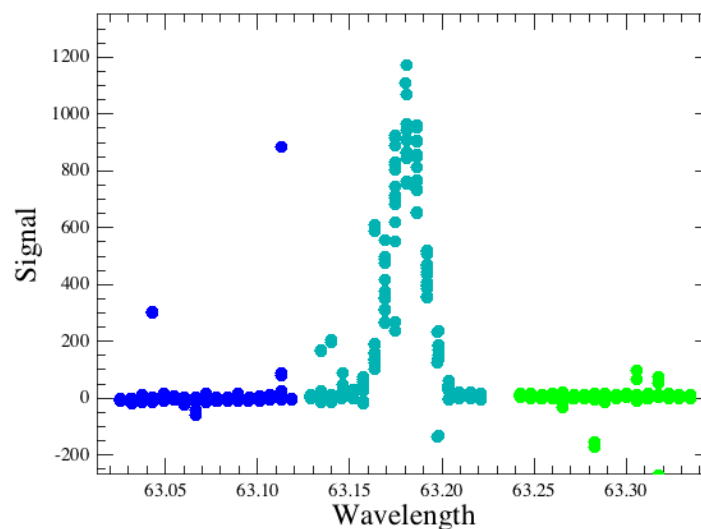


Figure 10.7. An example of the spectral from three different pixels (1, 8, 16) of a single module

where in the plot here the dark blue is pixel 1,12; light blue is 8,12; green is 16,12. (The observation this plot was taken from was done early in the mission, and the range sampling changed afterwards, so this plot is a slight exaggeration of what most observations will show.)

The dispersion is also important in determining what you see when you plot a single pixel. If your dispersion is sparse (Nyquist/SED mode default sampling) 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.

10.4.2. Basic plotting, and comparing the before- and after-task spectra

Here we show you (i) the basics of plotting PACS spectra and (ii) how to over-plot a before-a-task on an after-a-task spectrum. Where in the pipeline you could usefully do this is when: subtracting the telescope background; flatfielding; detecting glitches; rebinning the spectra.

10.4.2.1. A basic plot of signal vs time or wavelength

To plot just the signal of *Frames* 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]"
```

(If the data have been flux calibrated then the Y-axis unit will be Jy.) You use the `.getSignal(8,12)` method because `PlotXY` requires a *Double1d* array to plot, and with this method you extract the signal dataset of pixel 8,12 of your *myframe* as a *Double1d*.

The same is true for any PACS cube, but the method has a slightly different name:

```
p=PlotXY(mycube.getFlux(2,2), titleText="your title",\
line=1) # or line = 0 for dots: appropriate for PacsCubes
p.xaxis.title.text="Readouts"
p.yaxis.title.text="Flux (Jy)"
```

It is not necessary to specify `p=PlotXY()`, you could type `PlotXY()`, but with the first you can add more things to the plot afterwards (more data, annotations., axes titles, etc.).



Tip

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. You can find more information about the methods for the *Frames* product by looking in the [PACS DRM](#).

For the cubes and *Frames* you can also use the other syntax to access your spectrum as a *Double1d*:

```
# To extract the entire signal/flux array
signal = myframe.signal[8,12,:]
flux = mycube.flux[:,2,2]
# To extract only the first 100 signal/flux datapoints
signal = myframe.signal[8,12,0:101]
flux = mycube.flux[0:101,2,2]
# Or
signal = myframe.getSignal(8,12)[0:101]
flux = mycube.getFlux(2,2)[0:101]
```

Note the difference in order of the spatial and spectral axes between the *Frames* and all the cubes!

To plot a spectrum of signal versus wavelength,

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

```

titleText="title",line=0)
p.xaxis.title.text="Wavelength [μm]"
p.yaxis.title.text="Signal [Volt/s]"

```

10.4.2.2. Plot all the spectra of a module of a *Frames*, or several spaxels of a cube

To plot all the spectra of a module of a *Frames*, or several spaxels of a cube together, in one colour, then in the PlotXY command use the RESHAPE command: when you extract out the spectra of 16 pixels you are getting with a *Double2d* array rather than a *Double1d*—RESHAPE can be used to "reshape" the 2d to 1d, suitable then for use in PlotXY. The following script is to plot all the pixels (1 to 16 inclusive) of module 12 (the central spaxel of the field of view) for all readouts:

```

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

```

For the cubes the same principle applies, to select more than one spaxel to plot you can use RESHAPE,

```

p=PlotXY(RESHAPE(mycube.wave[2:4,2,:]),\
RESHAPE(mycube.flux[2:4,2,:]),titleText="title",line=1)
p.xaxis.title.text="Wavelength [μm]"
p.yaxis.title.text="Flux [Jy]"

```

However, for the *SpectralSimpleCube* that is the outcome of the pipeline task specProject or the task drizzle, note that the wavelength dataset is always a 1d dataset, so you do not need to specify the spaxel number nor do you need to use RESHAPE. You use the following syntax:

```

# For a SpectralSimpleCube, the wavelengths are access via:
wve = mycube.getWave()
wve = mycube.getWave()[0:101] # the first 100 datapoints
wve = mywave.wave[0:101] # also the first 100 datapoints

```

while for the flux dataset you continue to use RESHAPE when you extract more than one spaxel at a time.

10.4.2.3. Over-plotting from separate products

To over-plot two spectra of one or two *Frames*, for example before and after subtracting the off-source data from the on-source data for chopNod observations:

```

# extract a frame before you run the task specDiffChop
frame1=slicedFrames.get(1)
# plot all 16 spectra of the central module. As this is a 2d
# array but PlotXY can only handle 1d arrays, you need to
# RESHAPE the data
p=PlotXY(RESHAPE(frame1.wave[1:17,12,:]), \
RESHAPE(frame1.signal[1:17,12,:]))
# then run specDiffChop and again select a frame
frame2=slicedFrames.get(1)
# now over-plot the data after the chop offs have been removed
p.addLayer(LayerXY(RESHAPE(frame1.wave[1:17,12,:]), \
RESHAPE(frame1.signal[1:17,12,:]))

```

To over-plot two spectra from one or two *PacsCubes*, you can add a new layer to the plot created, for example:

```

sig1=mycube.getFlux(2,2)
wve1=mycube.getWave(2,2)
# if you want to get rid of NaNs (tho they are invisible)
idx=sig1.where(IS_FINITE(sig1))
sig1=sig1[idx]
wve1=wve1[idx]
#

```

```

p = PlotXY(wve1,sig1,titleText="your title",line=0)
p[0].setName("spaxel 2,2")
sig2=mycube.getFlux(1,1) # mycube or a different cube
wve2=mycube.getWave(1,1)
# if you want to get rid of NaNs (tho they are invisible)
idx=sig2.where(IS_FINITE(sig2))
sig2=sig2[idx]
wve2=wve2[idx]
#
p.addLayer(LayerXY(wve2,sig2,line=0))
p[1].setName("spaxel 1,1")
p.xaxis.title.text="Wavelength [Å]"
p.yaxis.title.text="Jy"
p.getLegend().setVisible(True)

```

So, to compare the spectra from a *Frames* or cube before and after a task has been run, you make sure you have a separate copy of the before and after so you can over-plot them.

10.4.2.4. Plotting a before and an after task spectrum

An example of plotting before and after flatfielding, or before and after the transients correction, is provided by the pipeline helper task plotPixels. To do this yourself, if working on a *Frames* it will be much easier to see the improvement if you use PlotXY to plot an entire spaxel, i.e. all 16 pixels. To do this, and to plot only the clean (not masked) data, you can use the following example:

```

# Before
frames1 = slicedFrames_b4.get(1)
# get the wave and flux for pixels 1 to 16 of module 12 (the central)
# for the entire time-line dataset. Format them from a 2d array to 1d
# array with the RESHAPE command, so PlotXY can handle them.
wave = RESHAPE(frames1.wave[1:17,12,:])
flux = RESHAPE(frames1.signal[1:17,12,:])
# find the indices of the NOT masked data (GLITCH and UNCLEANCHOP,
# which are generally the worst data)
# and RESHAPE these also
ind1 = RESHAPE(frames1.getMask("GLITCH").not())[1:17,12,:])
ind2 = RESHAPE(frames1.getMask("UNCLEANCHOP").not())[1:17,12,:])
# combine the good-data indices
ind = ind1.where(ind1 & ind2)
# plot
p=PlotXY(wave[ind],flux[ind],titleText="Flatfielding",line=0)
p[0].setName("before")

# After
frames2 = slicedFrames_after.get(1)
wave = RESHAPE(frames2.wave[1:17,12,:])
flux = RESHAPE(frames2.signal[1:17,12,:])
ind1 = RESHAPE(frames2.getMask("GLITCH").not())[1:17,12,:])
ind2 = RESHAPE(frames2.getMask("UNCLEANCHOP").not())[1:17,12,:])
ind = ind1.where(ind1 & ind2)
p.addLayer(LayerXY(wave[ind],flux[ind],line=0))
p[1].setName("after")
p.xaxis.title.text="Wavelength [Å]"
p.getLegend().setVisible(True)

```

See [Section 10.4.4](#) to learn more about plotting masks.

10.4.3. Plotting the on-source vs. off-source spectra for chopNod mode observations



Note

Comparing the on-source and off-source spectra is more usefully done in the late stages of the pipeline, when the spectral rebinning has been done, and the spectra have been flatfielded. It is for this reason that the script "Split on-off" is provided in the Pipeline menu of HIPE. The code snippets in this section work on data that are before these crucial steps have been executed.

For chopNod AOTs, this means plotting data from the on-chops and those from the off-chops. As is discussed in the [PACS Observer's Manual](#), the chopNod AOT sees PACS nodding between an A and a B pointing, and in each nod PACS chops between blank sky and the source. The pipeline task specDiffChop subtracts the off-chops (blank sky) from on-chops. There are 4 chopNod combinations possible (chop- nodA, chop+ nodB, chop+ nodA, chop- nodB), the former two are off-source and the latter two are on-source. After the pipeline task specDiffChop you are left with nod A (chop+) and nod B (chop-), both of which contain the on-source data with the off-source data subtracted. You may want to check that your off-spectrum does not have any spectral lines in it, as that would indicate that your off position was sitting on something astronomically interesting.

To compare the spectra from chop+ and chop- for a single *Frames*, of a single nod (necessarily, as the data are sliced on nod) from your *SlicedFrames* before running specDiffChop you could do something like this:

```
myframe = slicedFrames.get(1) # e.g.
stat = myframe.getStatus("CHOPPOS")
on = (stat == "+large") # a boolean, true where +large
on = on.where(on) # a Selection array
off = (stat == "-large")
off = off.where(off)
p=PlotXY(myframe.getWave(8,12)[on],myframe.getSignal(8,12)[on],line=0)
p.addLayer(LayerXY(myframe.getWave(8,12)[off],\
  myframe.getSignal(8,12)[off],line=0))
p[0].setName("on")
p[1].setName("off")
p.xaxis.title.text="Wavelength [ $\mu\text{m}$ ]"
p.getLegend().setVisible(True)
```

NOTES (for this script and the following one):

- To run this script you 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 ([Section 10.12](#)) or >print UNIQ(myframe.getStatus("CHOPPOS")); then change the third and fifth lines of the script appropriately
- The script worded as it is here works with a *Frames* of nod B, where the plus chopper position ("+large" here) is off-source and the minus chopper position is on-source. For a *Frames* of nod A you need to make the appropriate swap. To know what nod your selected *Frames* is, use the pipeline task slicedSummary on the SlicedFrames, as explained in the pipeline chapters.
- Remember that you need to run this script on a *Frames* that *has* got chop on and chop off data. Thus you must use a *SlicedFrames* from *before* the task specDiffChop.

This is not the only way to select these chopper positions, but it is conceptually the easiest to explain. To combine this with a selection of only the good data (i.e. not masked), as these can make the plot very difficult to inspect, you need to exclude the data that have been masked. There are a only few masks you need to exclude to produce nicer plots.

Note that the red detector has a bad pixel in the central spaxel (pixel 7,12). The blue detector has no bad pixels in the central spaxel. (The PACS Product Viewer [Section 10.8](#)) will show you where the bad pixels are—look at the BADPIXEL mask.)

```
x=8
y=12
myframe = slicedFrames.get(1)
stat = myframe.getStatus("CHOPPOS")
on = (stat == "+large")
off = (stat == "-large")
# to give the NOT masked data a boolean value of 1 instead of 0
msk1 = myframe.getMask("GLITCH").not()[x,y,:]
msk2 = myframe.getMask("UNCLEANCHOP").not()[x,y,:]
# then, locate all places where the masks are all 1 (ie, good)
goodon = msk1.where(msk1 & on & msk2)
goodoff = msk1.where(msk1 & off & msk2)
p=PlotXY(myframe.getWave(x,y)[goodon],myframe.getSignal(x,y)[goodon])
```

```
p.addLayer(LayerXY(myframe.getWave(x,y)[goodoff],\
  myframe.getSignal(x,y)[goodoff]))
p[0].setName("on")
p[1].setName("off")
p.xaxis.title.text="Wavelength [ $\mu\text{m}$ ]"
p.getLegend().setVisible(True)
```

This script will work for any single "myframe" slice from a *SlicedFrames* before you run the pipeline task `specDiffChop`, since you have `chop+` and `chop-` in all slices. After running that task, for the nod A frames you will have only `chop+` and for the nod B slices only `chop-`, as these are the respective on-source positions.

For unchopped line scan observations, see [Section 10.9](#).

10.4.4. Plotting the good and bad data

We explain more about masks in [Section 10.11](#). Here we show you how to plot masked data. You can do this on *Frames* and *PacsCubes* products, but not *PacsRebinnedCubes* or *SpectralSimpleCubes*, since these cubes carry no masks.

These are examples of plotting all datapoints for pixel 8,12 for a *Frames* slice, and then over-plotting only the ones that are masked for UNCLEANCHOP (or "GLITCH" or ...):

```
# EXAMPLE 1: plot good and over-plot bad
flx = myframe.getSignal(8,12)
wve = myframe.getWave(8,12)
p = PlotXY(wve,flx,line=0)
index_bad=myframe.getMaskedIndices(String1d(["UNCLEANCHOP"]),8,12)
p.addLayer(LayerXY\
  (wve[Selection(index_bad)],flx[Selection(index_bad)],line=0))
p[0].setName("all")
p[1].setName("good")
p.xaxis.title.text="Wavelength [ $\mu\text{m}$ ]"
p.getLegend().setVisible(True)

# EXAMPLE 2: plot good only, eg for GLITCH and BADPIXELS
masks=String1d(["GLITCH","BADPIXELS"])
flx = myframe.getUnmaskedSignal(masks,8,12)
wve = myframe.getUnmaskedWave(masks,8,12)
p = PlotXY(wve,flx,line=0,titleText="Good data only")
p.xaxis.title.text="Wavelength [ $\mu\text{m}$ ]"
p.getLegend().setVisible(True)
# and for the same on a PacsCube
masks=String1d(["GLITCH","BADPIXELS"])
flx = myframe.getUnmaskedSignal(masks,8,12)
wve = myframe.getUnmaskedWave(masks,8,12)
p = PlotXY(wve,flx,line=0,titleText="Good data only")
p.xaxis.title.text="Wavelength [ $\mu\text{m}$ ]"
p.getLegend().setVisible(True)
```

See the examples [Section 10.4.2](#) in to know the slight differences in syntax necessary to do this for a cube. You can also consult the [PACS DRM](#) for the product you want to plot to see what new methods (the `.getXXX`) there are for selecting good/bad data or indices.

What does the first example do?

- The first two lines are how you extract out of "myframe" the fluxes and wavelengths and put each in a new variable (which is of class *Double1d*).
- Next open 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 an *Int1d* variable called `index_bad` the X-axis (wavelength) array indices of myframe where the data have been flagged for the specified mask. The parameters are the mask name (listed in a *String1d*) and the pixel coordinates (8,12). If you want to enter more

than one mask name you do that as `String1d(["MASK1","MASK2"])`. Note that you can use the method `getUnmaskedIndices` to select out the indices of the unmasked data points, but for a *Frames* product only.

- 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_bad)]` will select out of `wve` those array indices that correspond to the index numbers in `index_bad`. You need to use the "Selection()" syntax because you are doing a selection on an array.

If you wish to extract the masked data (for a single mask) for several pixels/spaxels at once, you can work on the datasets, rather than use the `.getXXX()` method (see [Section 10.4.2](#) also):

```
wve=myframe.wave[8,12,:] # wavelengths for pixel for pixel 8,12
sig=myframe.signal[8,12,:]
msk=myframe.getMask("GLITCH")[8,12,:] # mask for pixel 8,12

# then, locate all places where the mask is 1 (ie, true, ie bad) and
# make that into a Selection
bad=msk.where(msk == 1)

# and to select these data:
wve_bad=wve[bad]
sig_bad=sig[bad]

# so you can now plot
PlotXY(wve_bad,sig_bad,line=0)
```

The masks you are most likely to look at (because if you wish to change any, it will be these) are SATURATION, GLITCH and OUTLIERS; note that UNCLEANCHOP and BADPIXELS data will be noisy.

To combine the masks (which are boolean arrays), so you can do the above on more than one mask type at a time, use `&` or `|` ("and" or "or"), which follow the usual rules for boolean combining (i.e. if there are any 0s, the `&` gives a 0; if there are any 1s, the `|` gives you 1):

```
msk = (msk1 & mks2 & mks3)
msk = (msk1 | mks2 | mks3)
```

and then you can select as before:

```
bad=msk.where(msk == 1)
wve_bad=wve[bad]
```

Masks are 0 for NOT masked, and 1 for IS masked. If you want to locate readouts that are bad for any of the masks, then you will need to use `|`, and to exclude these then from your plotting you select all `"msk == 0"`. After this you then you treat "msk" as in the previous script snippet.

(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.)

Note that the red detector has a bad pixel in the central module (pixel 7,12). The blue detector has no bad pixels in the central module.

10.4.5. Plotting Status values and comparing to the signal and masks

The Status table is explained in [Section 10.12](#). There are some Status data that you may want to compare to your spectrum, although note that this will not provide a diagnostic of your data, rather it will help you understand how the pipeline works.

Here we explain how to plot the Status data for CPR and GPR, which are the chopper and grating position values, against the signal with the data masked for a moving chopper (UNCLEANCHOP)

and grating (GRATMOVE) highlighted. The idea here is that where the data are masked as "taken while moving" you should also see that the Status data indicate the chopper or grating were moving. This will be done on a *Frames* product. The plot you will produce will show how the chopper and grating were moving during the entire length of your observation. You will see how many grating scan directions you had (grating scan direction 0 followed by 1 produces a triangle-like shape in the plot; the chopper switches between 2 positions for each grating step).

First, to plot Status data and the signal together:

```
# 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.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. The "l1." commands create new layers which are then added to the plot with `p.addLayer()`. Each new layer is plotted in a different colour.)

If you zoom in tightly on the plot you just made you should see something like this:

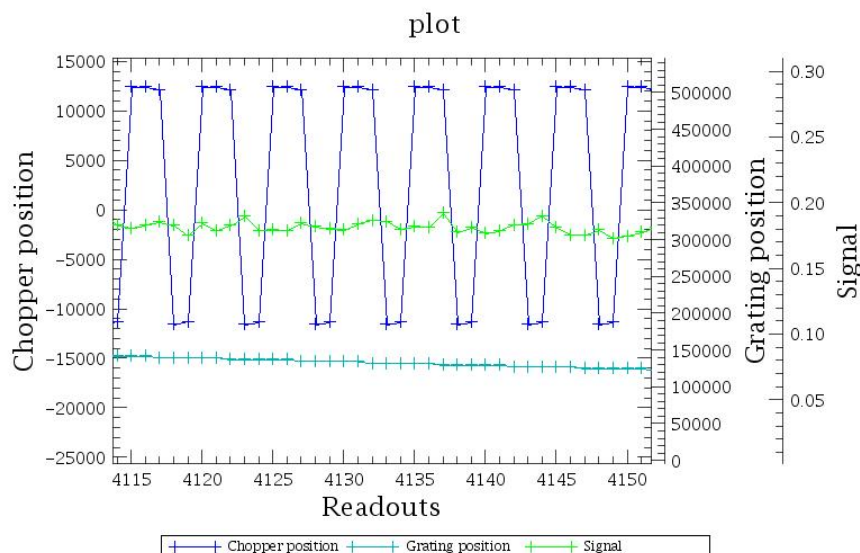


Figure 10.8. A zoom in on PlotXY of the grating and chopper movements for a Frames

What the figure above shows is that the chopper (dark blue) is moving up and down, with 2 readings taken at chopper minus and 3 readings taken at chopper plus. The grating (light blue) 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 (green) 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 because the data of this plot are old).

To plot the Status data with good- and bad-masked data:

```

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.getMasksIndices(StringId(["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.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)

```

This will product a plot similar to this:

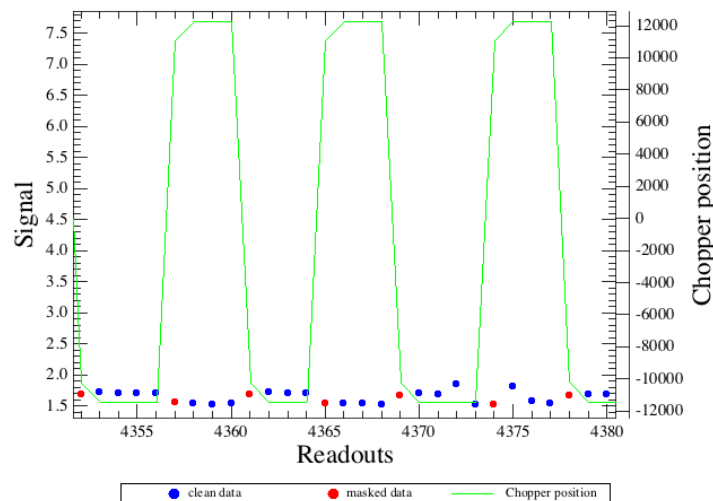


Figure 10.9. Masked/unmasked data v.s. Status

You can see that the red datapoints (masked) were taken at the same time that the chopper 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. 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) will be very slight.

10.4.6. Plotting the pointing

Once you have run the pipeline tasks to calculate the pointing you can plot the RA Dec movements of the central pixel (i.e. where was PACS pointing?). Positions are held in three places in your obser-

variations: the programmed pointing from your AOR is given in the Meta data (see the PPE [chp. 4](#) in *PACS Products Explained*), the Status contains the RA and Dec for the central pixel, and the RA and Dec for each pixel is held in separate datasets attached to the *Frames*.

To plot the central spaxel's position in your cube (the "boresight"):

```
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:

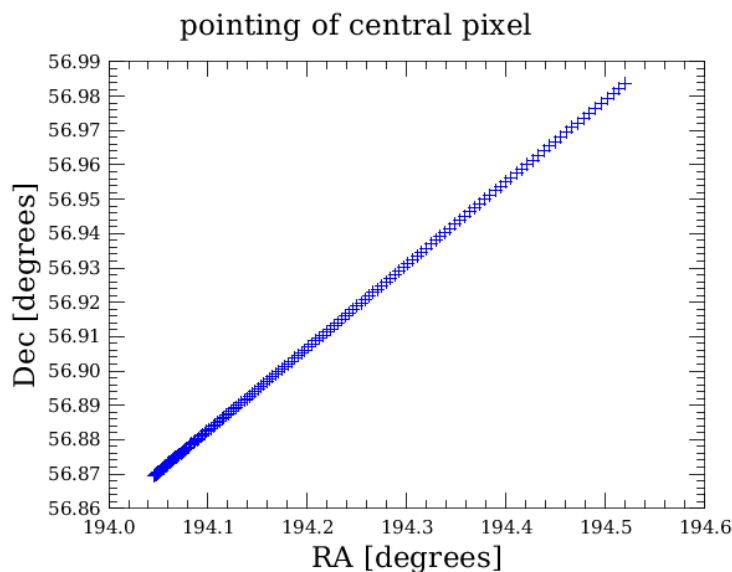


Figure 10.10. Movement of PACS during an observation

The footprint of the integral field unit that is PACS, the spectrometer, is not an exact square. It looks like then whenever you look at the spatial grid of your cubes with the various viewers and GUIs that HIPE offers, but in fact the footprint is different. This is demonstrated in [Section 10.6](#).

The footprint of the cube on the sky can be plotted. You need to select a time-point to plot (you could plot all, but that could make for a *very* slow plot as there are very many time-points in an observation). Then you do the following:

```
pixRa=RESHAPE(myframe.ra[:, :-1])
pixDec=RESHAPE(myframe.dec[:, :-1])
plotsky=PlotXY(pixRa, pixDec, line=0)
plotsky[0].setName("spaxels")
srcRa=slicedFrames.meta["raNominal"].value
srcDec=slicedFrames.meta["decNominal"].value
# note, "ra" can also be found in the ObservationContext as well
# as the SlicedFrames or Frames
plotsky.addLayer(LayerXY(DoubleIcd([srcRa]),DoubleIcd([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 (notice the slight offset of the second row of spaxels from the right):

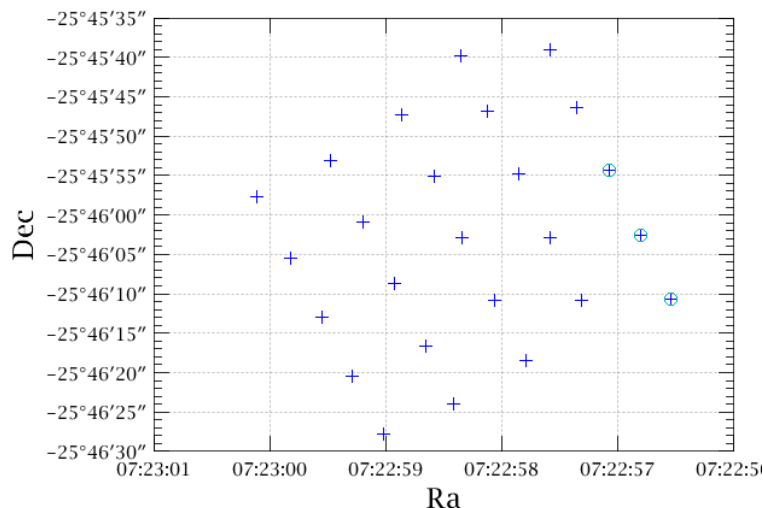


Figure 10.11. The IFU footprint. Crosses are the module centres and circled ones are modules 0 (bottom-most),1,2 (top-most). Module 5 is immediately to the left of module 0.

Explanation of the script:

- `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 (`-1`) gives you a 2D array.
- `myframe.ra/dec` are the `ra/dec` datasets attached to a *Frames*, which is not the same as the `Ra/DecArray` in the Status. "`ra`" and "`dec`" have dimensions `X,Y,Z` and were produced by the task `specASignRaDec`, whereas `Ra/DecArray` are 1D (they are just for the central pixel), and were produced by the task `specAddInstantPointing`.
- The "`-1`" is how you ask to plot the `ra` for the last element of the array.
- `srcRa` and `srcDec` 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 plot 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 the *SlicedFrames* that corresponds to each pointing (each slice is in fact a unique pointing), and then you can use the same commands as written above to plot the pointings of each *Frames*.

To make the same plot for two *PacsRebinnedCubes*, the following script will work:

```
#select the final ra and dec of a PacsRebinnedCube
# extract the cube with something like
#rebinnedCube1=slicedRebinnedCubes.get(0)
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 two raster pointings, from two *PacsRebinnedCubes* you have taken out of your rebinnedCube list.

10.5. Using the Spectrum Explorer on PACS spectra

The **Spectrum Explorer** (SE) is a HIPE tool for looking at spectra—single or multiple (i.e. cubes)—and via it you can gain access to the Spectrum and Cube Toolboxes and the Spectrum Fitter GUI. Here we explain the parts of the SE that are PACS-specific; to learn about the SE in general you should consult the *DAG* [chap. 6](#).

For PACS products—the *Frames* and the *PacsCubes*, *not* the *ListContexts* that they are contained within when being pipeline processed—what is unique in the Spectrum Explorer, and which is explained in this section, are a few extra panels which allow you to select, to plot (i) in the case of *Frames*, only certain pixels of the module, or an entire module, and (ii) for both, only masked or unmasked data, and (ii) for both, only certain blocks of data. These extra panels are located on the very left and the very right of the bottom of the GUI. Everything else will be the same as explained in the *DAG* [chap. 6.4](#).



Note

If you use the SE on *Frames* and *PacsCubes*, it will mainly be for inspection. Many of the Spectral maths tools that you can access via the SE will not work on *Frames* or *PacsCubes*, because they require the data to be held in monotonic arrays (which is not the case for these products). Spectral fitting via the SE (using the SpectrumFitterGUI) does work on *PacsCubes*, and if you exclude the masked data when fitting the template spectrum (i.e. that which you fit upon first opening the SE) then this is taken into account with the multifitter..



Note

If you are coming to this section directly from one of the pipeline chapters, you need to know that you can use the Spectrum Explorer on one *Frames* or cube at a time.

To extract the *Frames* from the *slicedFrames* product that the pipeline runs on:

```
myframe=slicedFrames.get(0) # get the first slice
```

To know how many "myframes" you can extract:

```
print slicedFrames.getNumberOfFrames()
```

Bear in mind that once the data have been sliced by the pipeline task *pacSliceContext*, the first slice will include only calibration block datapoints. The, at Level 1, the calibration slice is removed.

The same syntax will work for *PacsCubes* and *PacsRebinnedCubes*, where for these the calibration block slice will have been removed, leaving only science slices.

For the *SpectralSimpleCubes* you use this syntax instead:

```
cube=slicedProjectCube.refs[0].product # get the first one
```

See [Section 10.3](#) to learn more about selecting out slices.

Due to the large number of datapoints in *PacsCubes* and *Frames* products, the SE is slower when used on such data when of a large wavelength range, such as the full SED.

To call up the SE on your PACS product you do the usual right-click on the product in the *Variables* pane. The GUI will open in the Editor panel:

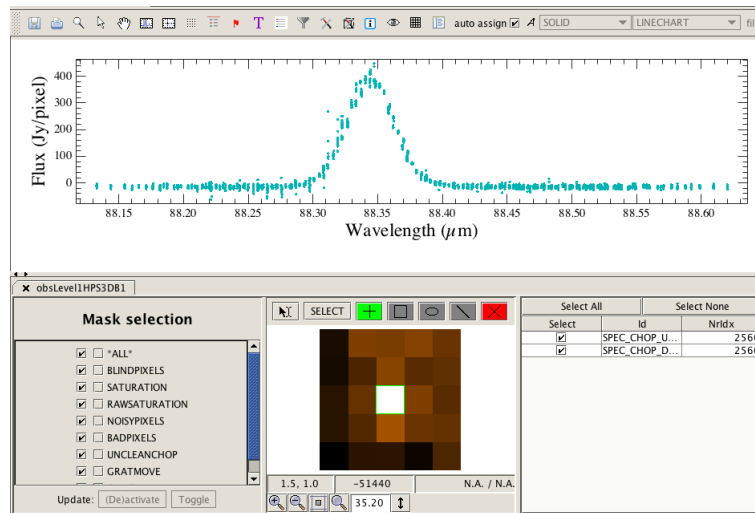


Figure 10.12. The Spectrum Explorer on a *PacsCube* (screenshot from HIPE 12)

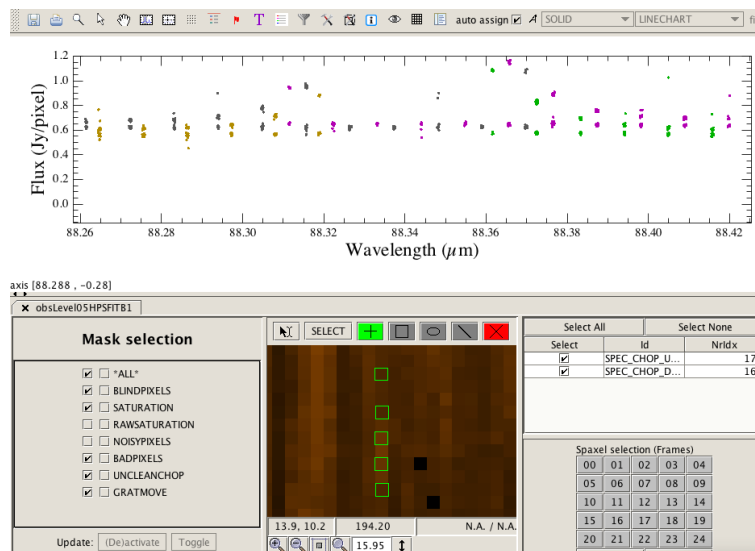









Figure 10.13. The Spectrum Explorer on a *Frames* (screenshot from HIPE 12)

If you undock the tab you will have more room to play with, and especially to see the cube/frames image at the bottom, from which you will select spectra to view and where you will find the various selection panels. (By undocking you also rearrange the panels, useful because sometimes the SE opens up with some panels squashed).

Undock and arrange the divider(s) to your liking. The image you see at the bottom is a single wavelength slice of the product you opened on: of size 5x5 spaxels for a *PacsCube* [or *PacsRebinnedCube*]; 18x25 for a *Frames* (noting that the top and bottom rows do not hold astronomical data, and each column is a single module/spaxel); and NxM spaxels for a *SpectralSimpleCube*.

For PACS cubes from Track 14 onwards, a "flip" has been added to the WCS such that when they are displayed in the Spectrum Explorer or Standard Cube Viewer, they have the same orientation as when using the Footprint viewer (the flip is top-left to bottom-right), and one that is more natural for most astronomers. A "flip" drop-down menu is provided at the bottom of the Cube viewer and Spectrum Explorer to allow you to flip further.

To view the spectrum from a spaxel of a cube or pixel of a *Frames*, do the following, in the cube image at the bottom of the GUI (this is called the *Data Selection panel*):

- No spectra will be shown initially when you open Spectrum Explorer.
- *Spectra are displayed* by left-clicking in a spaxel/pixel from the cube image in the *Data Selection panel*. There are four modes of selection indicated by four *region selection* icons above the image: single spaxel/pixel, rectangular or elliptical region, and along a line. The default mode upon startup is the single spaxel/pixel, and you can select any number of these one after the other. *But for the other modes, you always need to select the mode before each and every selection.*
 -  single spaxel/pixel: click on the icon and then on a spaxel/pixel to select one at a time
 -  all spaxels/pixels in a rectangular region: click on the icon and then single click on the cube image for a rectangle to appear; change the rectangle by moving the corners and move the whole shape by moving its centre
 -  all spaxels/pixels in an elliptical region: click on the icon and then single click on the cube image for an ellipse to appear; change the shape by moving the corners and move the whole shape by moving its centre
 -  all spaxels/pixels along a line: click on the icon and then in the cube image and a line appears (don't click the edge of the cube or one end of the line will go off the image); change the line by moving the ends, and move the whole line from its centre
 -  is to "edit" a selection (i.e. select it to be removed, to change its size, or change its location); select this and then the shape you want to edit
 -  is to pan the cube image, useful for when you have zoomed on it and want to access a part that has fallen out of the panel display space.
- *Spectra are removed* for single pixels/spaxels by clicking on them again. Spectra are removed for selections of shapes, or for everything selected, by using the "delete selection"  icon. The order of actions is to "activate" what you want to remove, and then to press the red cross.

To "activate" individual selections (single spaxels or entire regions) you can: (i) press the edit icon and then left-click on the selection; (ii) to select more than one use shift-left-click; (iii) to select everything, press the button that says "select" (or "deselect")—pressing more than once to get the action you want is OK to do.

- After removing spectra with the method above, or after having selected a shape, you need to select the single spaxel icon again to select single spaxels.
- For *Frames*, you can select and de-select all the spectra in a module by clicking in the "Spaxel Selection (Frames)" selection panel (once=select, twice=deselect).

To learn more about changing the plot properties, style, zooming and panning, selecting spectra, see the *DAG* [chap. 6.4](#).

To select parts of your spectra to view, i.e. to use the PACS-product specific panels, you do the following:

- **On a Frames or PacsCube:**
 - You can select, from the Block selection panel, which block to display the spectra of (from the selected pixels or spaxels that are displayed). This panel is on the right of the Data Selection panel, above the "Spaxel Selection (Frames)" for *Frames*, and it is the only selection panel on the right for *PacsCubes*.

By default, all blocks are shown. The blocks you can select to remove/include are usually only the different grating scan runs (of which there is a minimum of two in any dataset), since your data are sliced on all other blocks (wavelength range and pointing). Do this by clicking on the check box.

- You can also view data that are/are not masked with the "Mask selection" panel to the left of the Data Selection panel. Click to add the check mark to the box in the left column of boxes to *hide* data flagged as bad in the mask, and click again to remove the check to show these data.

If you want to highlight the bad data, click to add the check mark to the box on the right column next to the mask's name. These will appear in the plot as a red both highlighted with a red diamond.

Note that there may be some red diamonds without red dots in them: this is because those data were also flagged in another mask, and so the data themselves are not shown because most likely the mask they are flagged it has a check in the left column of boxes [don't show these data please]. If you were to ask to see the bad data for that other mask, the red dots will then appear.

- You can also edit masks with the SE, see [Section 10.11](#) to learn how to do this.
- **On PacsRebinnedCube or the SpectralSimpleCubes:** see the *DAG* [chap. 6](#) for more advice, because for these products there are no special PACS-panels.

With the SE you can also edit your masked datapoints, this is explained in [Section 10.11](#).

10.6. The PACS IFU footprint

Here we show you the sky and detector footprint of the integral field unit that is the PACS spectrometer. We also explain the relationship between the pixels of the detector and the spaxel of the cubes.



Note

A **spaxel** is a spatial pixel, that is a pixel on the sky that contain a whole spectrum, i.e. the pixel units of the 3d cube. PACS's spaxels have an intrinsic sky size of 9.4" square, but if you spatially resample your cube (e.g. using the pipeline task `specProject`) then your new spaxels will be smaller, but they are still called spaxels.

For *Frames* you can see the detector layout when you open it in the PPV ([Section 10.8](#)). The detector has 25 "modules" which correspond to the 25 spaxels (they are called modules at the detector level, spaxels at the cube level) and each has 18 "pixels", of which the central 16 contain science data:

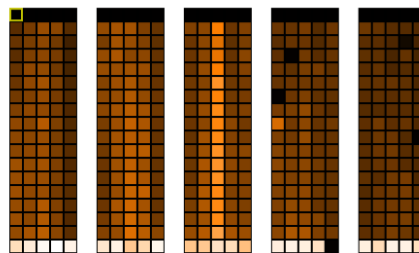


Figure 10.14. The PACS spectroscopy detector layout

Pixel `row=0,col=0` is at the top left (as you view in the PPV) and pixel `row=17,col=24` is at the bottom right.

In the *PacsCube* and *PacsRebinnedCube*, which have 5x5 spaxels, the footprint on the sky is not a perfect square, because one column of spaxels is offset from the others. However, when you view these cubes as images, e.g. when you open the Spectrum Explorer or Standard Cube Viewer, it will be displayed as a 5x5 square. Remember that this is not its sky layout!

We explain in [Section 10.4.6](#) how to plot the footprint of the cube. That will produce a plot similar to:

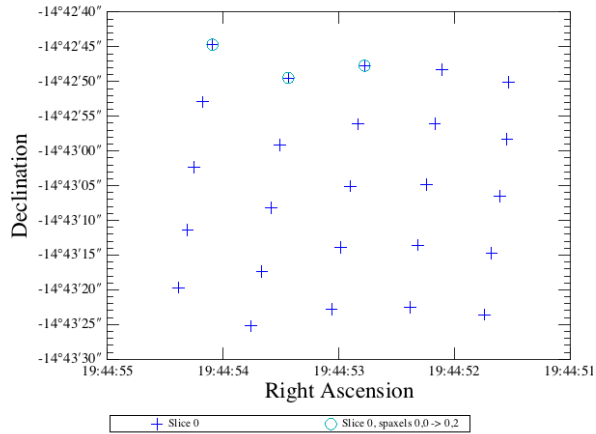


Figure 10.15. Sky footprint of the *PacsRebinnedCube* and *PacsCube*

The following table relates the spaxel coordinates in a cube (*PacsCube* or *PacsRebinnedCube*) to those in the preceding *Frames*. Note that when displaying *PacsCubes* and *PacsRebinnedCubes* created by SPG 14, they are flipped wrt the cubes created in earlier SPGs: X now increments as you move your mouse up and down, and Y as you move your mouse left and right. The coordinates reported at the bottom left of a display (an image display or a cube display) are given in order (Y,X) [row, column].

Table 10.1. Coordinates of the Frames to the *PacsCube*

Module in Frames	Spaxel (x,y) in <i>PacsCube</i>
0	0,0
1	1,0
2	2,0
3	3,0
4	4,0
5	0,1
6	1,1
7	2,1
8	3,1
9	4,1
10	0,2
11	1,2
12	2,2
13	3,2
14	4,2
15	0,3
16	1,3
17	2,3

18	3,3
19	4,3
20	0,4
21	1,4
22	2,4
23	3,4
24	4,4

Which can, scripturally, be repeated in this way, for a cube of any dimensions:

```
# for a cube with dimensions (wavelength, 8,6)
row=8
column=6
for r in range(row):
    for c in range(column):
        idx = (column * r) + c
        print "cube coordinate",r,c," index:",idx

# for the same cube, to go back and forth for a single coordinate
row = 8 # cube dimension
column = 6 # cube dimension
specIndex = 2
c = specIndex %column
r = specIndex/column
print r,c # spaxel coordinate
```

Despite the irregular sky footprint of the *PacsCubes* and *PacsRebinnedCubes*, when you open these cubes with a viewer (e.g. the Spectrum Explorer or the Standard Cube viewer) you will see a regular 5x5 layout. This is because the viewers cannot handle irregularly gridded images, and so effectively it is displaying them in cube coordinates (i.e. 5 pixels up and 5 pixels along: see the central image of [Figure 9.3](#)). In fact, the WCS of the *PacsRebinnedCubes* is defined in this way, rather than in sky coordinates. So you will find that, for *PacsRebinnedCubes*, `crpix1|2` (the first spaxel number) are 1,1; `cdelt1|2` (the spaxel increments) are also 1,2; and `crval1|2` (the location of `crpix1|2`) are 0,0. Both types of cubes do still have RA and Dec information for each spaxel (these are held in the "ra" and "dec" datasets), but this information is not in the WCS. If you want to know the sky coordinates of the spaxels, you can do the following:

```
# rcube = a single PacsRebinnedCube
x,y=2,2
print MEAN(NAN_FILTER(rcube["ra"].data[:,x,y]),\
           MEAN(NAN_FILTER(rcube["dec"].data[:,x,y]))
# where the NAN_FILTER is to remove NaNs from the computation of
# the mean
```

The coordinates of the *SpectralSimpleCube*, the projected, interpolated, or drizzled cubes, will depend on the details of the spatial projection and what you ask for when you run those tasks. These cubes do have a WCS, and the coordinates for individual spaxels can be gotten via the WCS,

```
wcs=pcube.getWcs() # get the WCS of the projected cube
crpix1=pcube.getWcs().getCrpix1() # spaxel coordinates of first spaxel
crval1=pcube.getWcs().getCrval1() # RA coordinates of first spaxel
crval2=pcube.getWcs().getCrval2() # Dec coordinates of first spaxel
cdelt1=pcube.getWcs().getCdelt1()*3600.0 # Ra spaxel offsets in arcsec
```

See the [SG](#) to learn more about working with a WCS with cubes. See [Chapter 9](#) to learn more about the differences between *PacsRebinnedCube* and projected or drizzled cube. See [Section 10.7](#) to learn more about a PACS footprint viewer, where you can over-plot a cube's footprint on an image.

10.7. The PACS Spectral Footprint Viewer

This viewer—the `pacSpectralFootprint` viewer—will over-plot the footprint of the spectrometer cube (a *PacsCube* or a *PacsRebinnedCube*) on any image with a WCS, e.g. one retrieved with the VO Tool Aladin, or any Herschel image. This task will become an *Applicable Task* in the *Task* pane when you click on a PACS cube or on an image, **but you should only open it on the image**: opening on the cube gives you a blank view.

To run the task:

1. Click on an image in the *Variables* pane, and then open the task in the (*Applicable*) *Tasks* View. The GUI appears in the *Editor* area. If you do not open the task on an image, after opening you can drag and drop an image into the upper part of the viewer. You can use the buttons on the lower-left of the display area of the GUI to zoom.
2. Then drag-and-drop the cube from the *Variables* pane into the display area of the footprint GUI.
3. If the product is a *ListContext* of cubes (i.e. the container that hold the cubes as you pipeline process your data), then the footprints of all the cubes will be drawn (if there are many, this will take some time). If you have very many repetitions in your data, then each repetition will be drawn and this can take a while.
4. You can select to view one or more IFU footprints in the table which is below the display (see figure below), by clicking on the coloured entry (click to see it flash, control (or command) click to stop it flashing), or to remove it click on "Remove" to see it flash and again to remove it.

Command line usage: You can run the task from the console as follows:

```
fp = pacSpectralFootprint(some_image_with_wcs)
# Then the "fp" variable has some methods to add more footprints,
# or to access the Display inside for e.g. custom annotations:
fp.addFootprint(some_cube)
fp.display.zoomFactor = 2
fp.display.setCutLevels(0.95)
```

consult its [URM](#) entry to learn more, and the [PACS DRM](#) to find out about its methods.



Note

Pay attention when comparing the display from the footprint viewer with that from the Standard Cube Viewer or the Spectrum Explorer: to give them the same orientation (position angle apart) PACS cubes from Track 14 onwards have had a "flip" added to the WCS such that when they are displayed in the Spectrum Explorer or Standard Cube Viewer, they have the same orientation as when using the Footprint viewer (the flip is top-left to bottom-right), and one that is more natural for astronomers. A "flip" drop-down menu is provided at the bottom of the Cube viewer and Spectrum Explorer.

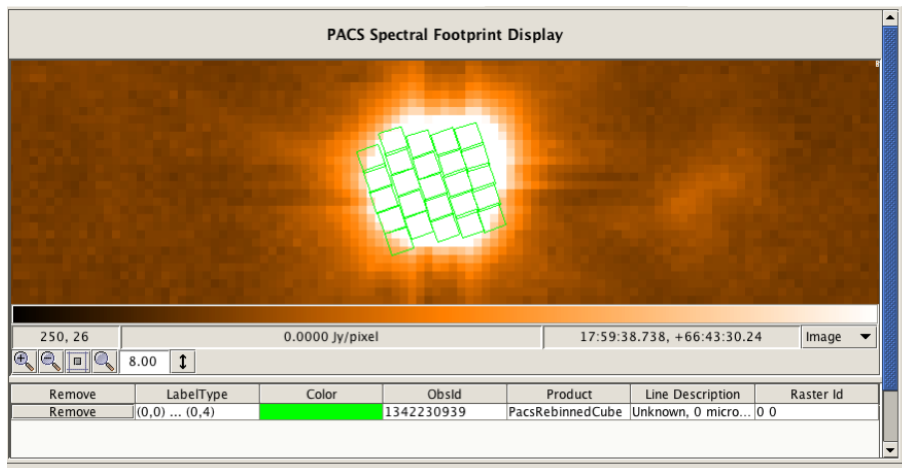


Figure 10.16. The PACS spectral footprint viewer

If you want to add WCS coordinates to the footprint viewer, a right-click on the image will give access to the normal image viewer menu, which includes adding axes. On the command line the following example can be followed:

```
obs=getObservation(1342179004,useHsa=1)
image = obs.refs["level2"].product.refs["HPPMAPR"].product
obs=getObservation(1342188943,useHsa=1)
scubes = obs.level2.red.rcube.product
fp = pacsSpectralFootprint(image)
fp.addFootprint(scubes)
fp.display.setCutLevels([0.0,0.2])
leftAxis = fp.display.getLeftaxis()
leftAxis.setWorldCoordinates(True)
```

10.8. PACS product viewer (PPV)

With the PACS product viewer (PPV) you can see the detector layout and the time-line dataset in all the detector pixels for the spectrometer. This GUI can be called up either with a right click on a *Frames* in the Variables panel (it does not work on maps or cubes), or on the command line:

```
MaskViewer(myframe)
```



Note

If you are coming to this section directly from one of the pipeline chapters, you need to know that you can use the PPV on one *Frames* or cube at a time.

To extract the *Frames* from the *slicedFrames* product that the pipeline runs on:

```
myframe=slicedFrames.get(0) # get the first slice
```

To know how many "myframes" you can extract:

```
print slicedFrames.getNumberOfFrames()
```

Bear in mind that once the data have been sliced by the pipeline task *pacsSliceContext*, the first slice will include only calibration block datapoints. The, at Level 1, the calibration slice is removed.

The same syntax will work for *PacsCubes* and *PacsRebinnedCubes*, where for these the calibration block slice will have been removed, leaving only science slices.

For the *SpectralSimpleCubes* you use this syntax instead:

```
cube=slicedProjectCube.refs[0].product # get the first one
```

See [Section 10.3](#) to learn more about selecting out slices.

With the PPV you can look at the data in the form of signal vs readout (=time), for each detector pixel, one by one. It also allows you to plot flagged data and over-plot Status values on the signal datapoints. A screenshot of the PPV:

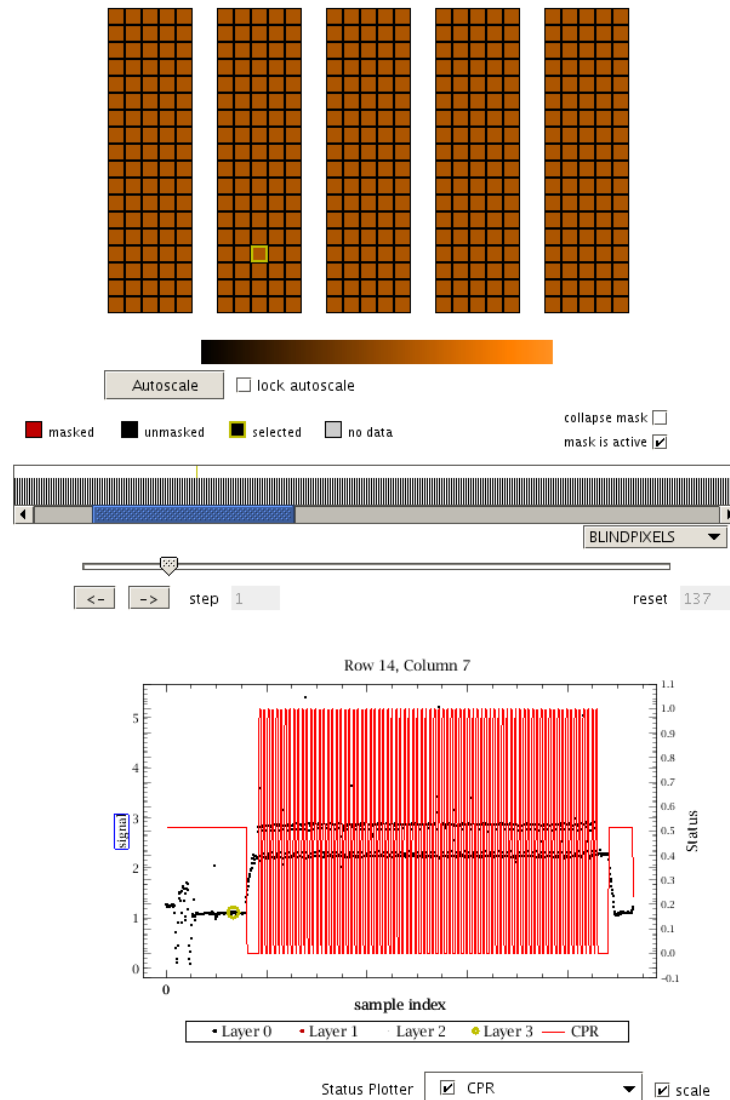


Figure 10.17. The PACS Product Viewer

At the top is a layout of the detector with colour-coded values, from which you can select pixels to see their time-line spectrum in the plot at the bottom of the viewer. (For spectroscopy, the 25 columns are the 25 modules/spaxels, and in each are 18 pixels containing the spectral data, although pixel row 0 and 17 contain no astronomical data.) In the middle is a (time-line) scroll-bar. Above the plot panel is a mask selection box. If you select from the Masks list, you will see in the plot below the datapoints that are so-masked turn red (the red dots may be small, so look carefully). If you select the "collapse mask" radio button, you will see datapoints highlighted in red that are masked for anything. At the same time, in the image at the top of the GUI, the pixels for which the masked data are the current readout/time-point will also be outlined in red. The current readout is selected with the scroll-bar, and a yellow circle will follow the scroll-bar movements in the plot below. The "autoscale" button is to help with the colour gradient in the detector view. At the very bottom of the PPV you can select some Status parameters to over-plot on the spectrum.

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

You can also edit the masks with the PPV: see [Section 10.11](#).

10.9. Looking at the background spectrum for unchopped mode AOTs, directly from an observation

If you want to check the background spectra for unchopped mode observations, e.g. to see if there is contamination present, it is a matter of comparing the off-source cubes from the on-source cubes.

For unchopped range this is straightforward because the two are separate observations. Extract the cubes from each observation—using the rebinned cubes is probably the most appropriate of the cubes to try first; if there is more than one cube there, you can use "slicedSummary" to figure out what the spectral and spatial coverage of each cube is, e.g.

```
# first red rebinned cube, on source and off source
onCubeContext = obsOn.level2.red.rcube.product
slicedSummary(onCubeContext)
offCubeContext = obsOff.level2.red.rcube.product
slicedSummary(offCubeContext)
onCube = onCube.get(0) # the first cube slice
offCube = offCube.get(0) # here also
```

and then right-click on one of the cubes in the Variables panel, and select the *Spectrum Explorer*. Then drag-and-drop the other cube into the Spectrum Explorer. They can now be compared to each other. To learn how to use the Spectrum Explorer on Level 2 cubes, see instead the *DAG* [chap. 6](#).

For the unchopped line mode, if starting from an observation gotten from the HSA rather than working your way through the pipeline, a few extra commands are necessary since the on-source and off-source rebinned cubes are not immediately available. Get one of the Level 2 cube types, run some pipeline tasks on them (to remove the outliers, which will make for a much cleaner), and then using a wavelengthGrid, turn the cubes into the rebinned cubes:

```
slicedCubes = obs.level2.red.cube.product # or blue
upsample = 2
upsample = 4
calTree = getCalTree()
waveGrid=wavelengthGrid(slicedCubes, oversample=2, upsample = upsample,\
  calTree = calTree)
slicedCubes = activateMasks(slicedCubes, StringId(["GLITCH","UNCLEANCHOP",\
  "SATURATION","GRATMOVE", "BADFITPIX", "BADPIXELS"]), exclusive = True,\
  copy=copyCube)
slicedCubes = specFlagOutliers(slicedCubes, waveGrid)
slicedCubes = activateMasks(slicedCubes, StringId(["OUTOFBAND","GLITCH",\
  "UNCLEANCHOP","SATURATION","GRATMOVE", "BADFITPIX", "OUTLIERS",\
  "BADPIXELS"]), exclusive = True)
slicedRebinnedCubes = specWaveRebin(slicedCubes, waveGrid)

slicedOffRebinnedCubes = selectSlices(slicedRebinnedCubes, onOff=['off'])
slicedOnRebinnedCubes = selectSlices(slicedRebinnedCubes, onOff=['on'])

slicedSummary(slicedOffRebinnedCubes)
slicedSummary(slicedOnRebinnedCubes)

offCube = slicedOffRebinnedCubes.get(0) # for example
onCube = slicedOnRebinnedCubes.get(0) # for example
```

Comparing the onCube and offCube can be done with the Spectrum Explorer.

To look at the spectrum that has been subtracted from an unchopped observation (line or rangeScan) as computed by the pipeline task specSubtractOffPosition, you can run this task in the usual way but specify two additional parameters:

- `withOffCubes = True`: the output of this task then includes the off-subtracted cubes (as usual) and the cubes containing the spectra that were subtracted. Use "slicedSummary" to see the order that these off-cubes are held in the output
- `copy = True`: should be set if you want to run `specSubtractOffPosition` again on the same input `slicedRebinnedCubes` but with different parameters. This is necessary to avoid overwriting the input `slicedRebinnedCubes` with the output (off-subtracted) `slicedRebinnedCubes`.

**Note**

For the unchopped mode, the off-source data are nod A, and the on-source data are nod B.

10.10. How to know the PACS spectral resolution at a given wavelength

When fitting spectral lines, it is often useful to be able to make an initial guess of the FWHM of the line based on the instrumental resolution. The spectral resolution depends on the spectral order: order 1 for band R1, order 2 for bands B2A and B2B, order 3 for band B3A. Use `obsSummary(obs)` to know what spectral band(s) are in your observation. The HIPE task `getSpecResolution` gives the instrumental resolution in km/sec.

```
calTree=getCalTree()
resolution = getSpecResolution(order, wavelength, calTree=calTree)

# then
speedC = herschel.share.unit.Constant.SPEED_OF_LIGHT.value
speedC = speedC/1000. # into km/s
line = 88.0 # an example wavelength
fwhm = resolution / speedC
```

10.11. Masks and masking

Masks are a way to avoid having to actually remove bad datapoints; instead of removing or replacing the data we flag them as "bad". This can be done any number of times—in the pipelines we have tasks that create individual mask datasets with flags (0 or 1) for glitches, for saturation, because they were taken while parts of the instrument were still moving, or because they require a particular warning. This means any one data point can be flagged as being bad for saturation but good for glitches: you need to consider all the important masks if you want to select out bad data. An individual mask is a data array with the same amount of storage as the array of the science data, so a mask holds a state for every science data value. If the data is 3d, then the masks are also 3d. The only masks that are 2d are those that mask out whole pixels, these being the `BADPIXEL` and `NOISYPIXELS` masks. The masks generally have self-explanatory names, and the mask state values are 1=True for bad data, **is masked**, and all others are 0=good, **not masked**. They are held as boolean arrays. The whole group of masks that are created (by the pipeline or by you) are held in the "Mask" container of a *Frames* or *PacsCube*. In this way we differ to SPIRE and HIFI in our handling of masks.

For spectroscopy we also use the concept of a `MasterMask`. This is a collapse of all the "active" masks contained in your *Frames* or *PacsCube*. The task `activateMasks` (which you use when running the pipeline) is used to activate the masks you want to consider, and at the same time (and by default) can deactivate all others. Hence you can chose which masks to include in your pipeline processing, and this way you can also create your own masks and have them considered.

The masks that are created by the spectroscopy pipeline are:

- `BLINDPIXELS`: pixel masked out by the `DetectorSelectionTable`, so applied already at Level 0. (This mask will be blank for spectroscopy).

- **BADPIXELS**: bad pixel masked during pipeline processing by the task `specFlagBadPixelsFrames`.
- **SATURATION**: entire saturated pixels or individual saturated readouts, found by the task `specFlagSaturationFrames`.
- **RAWSATURATION**: If the central detector pixel is saturated for a readout, then this mask is set to 1 for all pixels for that time-point, as a warning that "the central pixel is saturated here, maybe the other pixels are also". This is also created by the task `specFlagSaturationFrames`, but the reason that it looks at the central pixel only is because it is only for this pixel that the data are downlinked from Herschel in "raw" format, rather than "fit" format. Saturation is easier to spot in these data.
- **GLITCH**: Readouts (signals) affected by glitches (cosmic rays), and detected with the task `specFlag-GlitchFramesQtest`.
- **OUTLIERS**: Masking of outliers of the *PacsCubes*. This differs from the GLITCH mask in that the task that create the OUTLIERS operates along the wavelength dimension, rather than the time dimension.
- **UNCLEANCHOP**: masking of unreliable readouts/signals taken during chopper transitions (i.e. taken while the chopper was still moving) by the task `flagChopMoveFrames`.
- **GRATMOVE**: masking of readouts (signals) taken during grating movements: `flagGratMove-Frames`.
- **NOISYPIXELS**: noisy pixels mask is added by the task `specFlagBadPixelsFrames`, a warning mask based on on-ground testing of the detectors.
- **OUTOFBAND**: a warning mask, used for the large rangeScan and SED AOTs, to indicate data that are out of band: e.g. where the investigator defined, in HSPOT, a range in the red such that the "free" range in the blue falls out of the range of the blue filter. These data will be useless, and not worth including in the final cubes, but you can chose to keep them. Added by the task `waveCalc`.
- **INVALID**: a mask created by `specDiffChop` when working in the background normalisation pipeline. Any data that are 0 in the denominator (i.e. on+off=0.0) are assigned as INVALID. This is avoid creating infinite values later.
- **INLINE**: a mask added by one of the tasks—`maskLines`—that you can run when flatfielding line scan AOTs. Is to identify spectral lines so they are ignored when the continuum is fitted as part of the flatfielding process.
- **OUTLIERS_FF**: created during the flatfielding of line scan AOTs. It sets to bad any datapoints that after flatfielding are outliers when before flatfielding they were not. It is an information (or warning) mask.
- **OUTLIERS_B4FF**: a mask also added during the flatfielding process. It is an information mask, showing all the OUTLIERS that were detected before flatfielding was done on the data.
- **UNCORRECTED**: produced by the transients correction task `specTransCorr`; for data that cannot be corrected.
- **NOTFFED**: data not flatfielded by the rangeScan flatfielding task
- **NODATA**: a mask produced by `drizzle`, indicates that there is was no input data from both nod positions when it combined these nods when making drizzle cubes (`chopNod` pipeline scripts).
- **DRIZZLE**: a mask of data clipped out by a sigma-clippng outlier detection done by the task `drizzle`.

You can see what masks you have by looking at your *Frames* or *PacsCube* with a Product viewer, with the PACS product viewer (PPV: see [Section 10.8](#)), and with the Spectrum Explorer (SE: see [Section 10.5](#)). You can also plot spectra and without masked datapoints, as is shown in [Section 10.4.4](#).

Changing the masks is something you are rarely likely to want to do. It is not recommended that you edit any masks of the pipeline, with the possible exceptions of the OUTLIER mask, in rare occasions the GLITCH mask (as is explained in the pipeline chapters), or the NOISYPIXELS masks. One reason for doing this would be if you wanted to force certain datapoints to be masked without creating your own mask: you instead add them to a pre-existing mask. We show you how to do this below. You can, of course, create your own masks (we also explain this below), for example if you had your own deglitching routine. Scripting requires you understand how to interact with the Masks class, meaning you should be comfortable with reading up on the class in the [PACS DRM](#). Here we show you the basics of interacting with masks on the command line and with various GUIs.



Note

If you are coming to this section directly from one of the pipeline chapters, you need to know that you can use inspect the masks on one *Frames* or *PacsCube* at a time. (The other types of cubes do not have Masks.)

To extract the *Frames* from the *slicedFrames* product that the pipeline runs on:

```
myframe=slicedFrames.get(0) # get the first slice
```

To know how many "myframes" you can extract:

```
print slicedFrames.getNumberOfFrames()
```

Bear in mind that once the data have been sliced by the pipeline task *pacsSliceContext*, the first slice will include only calibration block datapoints. The, at Level 1, the calibration slice is removed.

The same syntax will work for *PacsCubes* and *PacsRebinnedCubes*, where for these the calibration block slice will have been removed, leaving only science slices.

See [Section 10.3](#) to learn more about selecting out slices.

10.11.1. Editing masks for a few bad values

You can use the **Pacs Product Viewer** (PPV: see [Section 10.8](#) to learn more about using the PPV) to edit masks. This will only work for *Frames* products. Chose it via the right-click menu on your *Frames* in the Variables panel.

First, open the PPV on your *Frames*. Then click a pixel in the detector array to see signal time-line plot at the bottom of the PPV. When you spot some datapoints you want to toggle the masks of (i.e. mask them or unmask them), first make sure you are looking at the right mask. Masks to view are selected from a Masks drop-down menu in the middle of the PPV. From the bottom of that drop-down menu, you can also chose add your own mask. You can only edit masks that exist (or that you create yourself within the PPV).

Select the points you want to mask/unmask in the plot panel, by drawing a rectangle around them: press the shift button on your keyboard; the mouse cursor will get a "+" to indicate that a drag will add a selection of datapoints; drag a rectangle with the mouse. All values within the rectangle will get selected (and will be colour-coded in yellow). If you need to edit the selection, hold down the alt-button on your keyboard. A "-" sign at the mouse cursor will indicate that if you drag a new rectangle, all previously selected values within the new rectangle will get deselected. After you have selected your points, shift right-click again in the plot to choose one of the two masking options offered, as this figure shows:

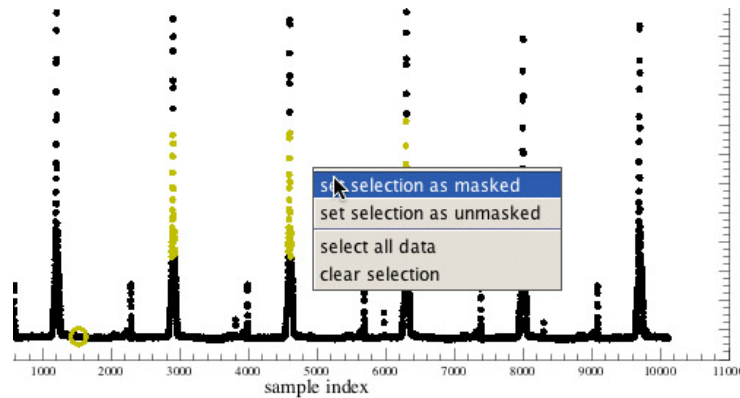


Figure 10.18. Editing masks with the PPV

As soon as you toggle the state of your selected datapoints, the *Frames* is changed. Masked datapoints will be highlighted in red.

Masks can also be edited within the **Spectrum Explorer**: see [Section 10.5](#) to learn about the PACS-specific parts of the Spectrum Explorer. You can do this to the *Frames* or *PacsCube* stage of your data, although it will be easier to do it for the *PacsCubes*. To edit single or small groups of datapoints you do the following, after opening the SE on your product:

1. Select a single spaxel or pixel, to display its spectrum in the plot at the top of the SE.
2. Display the flagged datapoints for a mask you wish to look at by ticking the right box next to the mask name in the Mask selection panel, on the left of the SE GUI (see [Figure 10.12](#): the bad-flagged datapoints will be highlighted with red diamonds around them. You cannot add new masks in the SE, you can only edit masks that already exist, and this is why you first need to highlight the data of a mask.
3. Right-click in the plot panel and select, from the menu that appears, *Tools>SelectPoints*. If you do not select this tool, then the next movement will zoom on the plot instead of selecting points.
4. You can now use the left mouse button to draw a box around the point(s) you want to *either mask or unmask*. If the datapoints you are selecting are already masked, then the next action will unmask them, and vice versa. You can also click on a single point to select it alone, but you have to select *exactly so* and so drawing a box is easier. The datapoints selected will be outlined in circles.
5. Now click the "Toggle" button at the bottom of the Mask selection box. This will do the following: "points that were masked are now not masked" and "points that were not masked are now masked". The datapoints toggled will now appear in the colour/symbol appropriate to their new status.

Note that any datapoints selected that are flagged in other masks (those you are not showing) will not have their state affected in those other masks.

6. You have now changed those datapoints. As soon as you toggle, you change! You can repeat actions 5 and 6 as often as you wish. You can also change your selection box before you Toggle by redrawing the box around the previously-enboxed points. If you want to do any more zooming, remember to right-click in the plot to access *Tools>Zoom*, otherwise your next mouse—plot interaction will be the selection of another box, rather than a zoom or pan.
7. To remove the selection of the datapoints (which adds circles around them) so you can see your newly-toggled spectrum more cleanly, right click in the mass of selected points and chose *Point Selection>Deselect*. Now you can see the red datapoints with red diamonds highlighting any new datapoints you just toggled to be flagged as bad (or you will see a lack of red diamonds around datapoints you just toggled to be flagged as good). If you click to add a check to the right box next to the mask name you were just working on, you will again see the spectrum without the bad data.

Some warnings:

- If you are worried about messing up the mask editing, we recommend you save the *PacsCube* or *Frames* you are working on first: `>newcube=mycube.copy()`. Remember: as soon as you toggle, you change! If you are working on a *Frames* or *PacsCube* within a *ListContext*, then save the entire *ListContext* before editing any element in it. See [Section 10.3](#) to learn about how to save, remove, replace products within a *ListContext*.
- You cannot toggle a datapoint that is only a red diamond, it must be also a dot, i.e. you cannot toggle a datapoint that is only Highlighted, it must also be unHidden. If you want to toggle a datapoint that you can only see as a red diamond, you will need to unHide (deselect the left select button for that Mask) for the other masks until you can see the datapoint as a dot.
- If you want to look for and flag bad datapoints without caring what mask they belong to, you may be tempted to look at the default displayed spectrum, as all the masked data are hidden (all the left-column check boxes are checked), and start selecting points to toggle. Unfortunately, you will only be able to select points, you will not be able to toggle. You need to toggle *within* an already existing mask, so you do need to select a mask. You could, of course, always create a new mask yourself ([Section 10.11.2](#)) and use that one for your datapoint editing. (But then remember to use it in the pipeline!)

To remove the selection of the datapoints (which adds circles around them, making it hard to see the data underneath), right click in the mass of selected points and chose Point Selection->Deselect. Now you can see the red datapoints with red diamonds highlighting any new datapoints you just toggled to be flagged as bad (or you will see a lack of red diamonds around datapoints you just toggled to be flagged as good). If you click to add a check mask to the right box next to the mask name you were just working on, you will again see the spectrum without the bad data, including those you just toggled.

It should be obvious that this way of manipulating masks is only useful if you are doing a small amount of editing. For example (and has explained in the pipeline chapters) in some cases very bright lines can be falsely identified as being GLITCHed. Your options are to either ignore that mask altogether, or to go into the SE and remove the mask for those bright lines. (You can do this just before rebinning the *PacsCube* to turn it into a *PacsRebinnedCube*).

10.11.2. Manipulating masks on the command-line

10.11.2.1. The basics

To extract or identify which datapoints are masked as good or bad using the command line (e.g. if you want to plot the data) the easiest way is to use the methods on the *Frames* or *PacsCubes* product directly. Some of these methods were used in [Section 10.4.4](#) and a summary of the most useful methods that work on *Frames* and *PacsCubes* is:

```
# get the entire mask as a Bool3d
mask=myframe.getMask("MaskName")
# returns the indices of good data
idx=myframe.getUnmaskedIndices("Maskname", row, col)
# returns the indices of the bad data
idx=myframe.getMaskedIndices("Maskname", row, col)
# returns the signal for Frames
signal=myframe.getUnmaskedSignal("Maskname", row, col)
# returns the flux for PacsCube
flux=myframe.getUnmaskedFlux("Maskname", row, col)
# returns the wavelength for both
wave=myframe.getUnmaskedWave("Maskname", row, col)
# example of use on myframe, a Frames product
PlotXY(myframe.getUnmaskedWave("GLITCH",8,12), \
myframe.getUnmaskedSignal("GLITCH",8,12))
```

You will have to look at the [PACS DRM](#) entry for *Frames* and *PacsCube* to learn all the variations on these methods (e.g. you can specify different things inside the ()) and read up on all other methods.

**Note**

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

Working on masks and adding new ones directly: To extract a mask as an entity, you can use these methods

```

masks = myframe.getMask() # get all the masks
mask = myframe.getMask("GLITCH") # get a single mask
print mask.dimensions # how big is it
mask = myframe.getMask("GLITCH")[8,12,:] # the mask for a single pixel
mask = myframe.getMask("GLITCH")[8,12,45:67] # and a range of readouts

```

The first extracts the entire Mask extension, creating an *AMask* class object (which is a *Mask* class object, and to work with these classes you can look them up in the [PACS DRM](#)). The second extracts the dataset of a single mask, and the class of product created is a *Bool3d*. The fourth and fifth show you one way how to extract the booleans of a mask for a single pixel for all or for only certain readouts.

To change a single masked datapoint, you first need to find the spatial (pixel or spaxel) coordinates and the temporal coordinate (the array's index). You can do this on a *Frames* or a *PacsCube*, the examples here are all *Frames*. The masks have the same dimensions as the spectra,

```

print myframe.getMask("UNCLEANCHOP").dimensions
--> 18,25,13600
print myframe.dimensions
--> 18,25,13600

```

thus one (admittedly rather laborious) way to locate the bad/good points is to plot the spectrum vs. array index (i.e. `PlotXY(myframe.getSignal(8,12))`), to visualise masked data, and locate the index value of any data you do or do not like (see below for instructions). You can also use the PPV to look at the time-line spectra of your *Frames* products, to determine their index (the x-axis) values. You can then change those single entries (paying attention to the values in the ()): these here are just an example):

```
myframe.setMask("UNCLEANCHOP", 3,5,1823,True)
```

to set the UNCLEANCHOP mask for index 1823 for pixel 3,5 to True.

This ".setMask" method is also the way you can change larger sections of masks, or to add 1s and 0s to your own mask. The .setMask method allows you to modify single masked points or an entire section of the 3D mask array (which you may want to do if you have created your own 3d mask). You can consult the [PACS DRM](#) to learn about the various of .setMask that will allow you to add a whole mask in one go.

As an example, let's say you want to add a pixel of a *Frames* to the BADPIXELS mask, for a single *Frames* slice of a *SlicedFrames*.

```

# slicedFrames is your SlicedFrames
# First extract the Frames you want to edit
# Here I want to add pixels 1,12 and 1,13 to the BADPIXELS as bad
frame=slicedFrames.get(0)
sz=frame.getDimensions()[2]
for i in range(sz):
    frame.setMask("BADPIXELS",1,12,i,True)
    frame.setMask("BADPIXELS",1,13,i,True)
# Once you have done all editing and checked them,
# Replace the original Frames with the edited one
slicedFrames.replace(0,frame)

```

To add a new mask, do the following:

```

# slicedFrames is your SlicedFrames
# First extract the Frames you want to edit

```



```
# Here I want to add pixels 1,12 and 1,13 to the BADPIXELS as bad
frame=slicedFrames.get(0)
sz=frame.getDimensions()[2]

# Create a Bool3d with all values set to 0
mybool3d=Bool3d(18,25,sz,False)
st=0
end=3210
# Now change some of them. I want to add pixels 1,12 and 1,13 as bad
# between readouts 0 and 3210
mybool3d.set(1,12,Range(st,end),True)
mybool3d.set(1,13,Range(st,end),True)

# Now add that mask
frame.addMaskType("MyMask","my chosen bad data")
frame.setMask("FuckedPixels",mybool3d)
# And add that Frames back into slicedFrames
slicedFrames.replace(0,frame1)
```

You can remove masks with

```
myframe.removeMask("MYGLITCH")
```

And rename then with

```
myframe.renameMask("OUTLIERS", "OUTLIERS_1")
```

The PPV does not work on a cube, so to see the time line/array position of a bad datapoint you need to PlotXY your *PacsCube* spectrum/a. But otherwise what we have written here is the same for a *PacsCube* as for a *Frames*.

Working on the actual masked data: To locate masked data (as opposed to the looking at the boolean masks themselves) you use methods such as:

```
flx=myframe.getSignal(8,12)
wve=myframe.getWave(8,12)
index_cln=myframe.getUnmaskedIndices(String1d(["UNCLEANHOP"]),8,12)
flx_clean=flx[Selection(index_cln)]
wve_clean=wve[Selection(index_cln)]
```

will allow you to locate all the readouts (e.g. to plot the spectrum) that have NOT been masked as UNCLEANCHOP. To locate the readouts that have not been masked for anything, you can either specify all masks in the String1d(["", "", ""]), or, for *PacsCubes* (but not, currently, for *Frames* although that may change), you can use `.getUnmaskedIndices(8,12)`.

To locate all the points that HAVE been masked (e.g. GLITCH), you can use the following syntax:

```
# either (should work for Frames, may not for PacsCube)
index_dirty=myframe.getMaskedIndices(String1d(["UNCLEANHOP"]),8,12)
# or, if that method does not work on your product, do the following:
s1 = myframe.getMask("GLITCH")[8,12,:]
# to use it to select on the spectrum from pixel 8,12,
# to see the glitched data itself, the following is necessary:
sla=s1.where(s1==True).toInt1d() # locate the GLITCHed data
# apply that to your data
flux = myframe.getSignal(8,12)
flux_glitch=flux[Selection(sla)]
```

(to get all the data that have NOT been glitch masked, you can replace the True with False in `sla=s1.where(s1==True).toInt1d()`—an alternative to the `.getUnmaskedIndices` method.)

10.11.3. The "flag" array in cubes

In the *PacsRebinnedCubes* created by `specWaveRebin` and the *SpectralSimpleCubes* created by `drizzle` there is a flag array instead of a Masks one. You can see this when using the Product viewer on a

cube. "Masks" are not the same as "flags". These both are provided for the same purpose—identifying bad datapoints—but they are very different sorts of datasets and found in different types of products. Masks, as discussed here and as found in PACS data as you pipeline process them, are a PACS-only arrangement: they cannot be understood by HCSS (i.e. general) tasks. The general tasks that you can find in HIPE, which work on any product, understand only "flags". So, we have given our Level 2 cubes (the *PacsRebinnedCube*, the *SpectralSimpleCube*, and also the extracted point source corrected spectra) a "flags" dataset, the information in here being based on the information held in the Masks: each mask has a corresponding bit in the flag integers. (This really is not very important to understand, but you may want to be aware of the fact.)

The values in the flag array include the 1s and 0s from the following Masks for the *PacsRebinnedCube*: SATURATION and RAWSATURATION and all masks **not** activated when the cube was created. For the cube created by drizzle, only the NOD mask is included, indicating whether a datapoint includes data from both nod positions or not.

If you are interested in reading these flags in these cubes, you can adapt the following:

```
# get the flag array from your cube called "cube"
flag = cube.getFlag()
# what Masks went into this array?
print flag.flagTypes()
# get a Bool3d of the data in a particular flag/mask
saturat = flag.getFlag("SATURATION")
# turn that into a Double3d
saturation = Double3d(saturat)
# the dimensions of the cube this came from
print saturation.dimensions

# Plot the spectrum of a spaxel and overplot the
#saturation flag value
wave=cube.getWave()
flux=cube.getFlux(2,2)
p=PlotXY(wave,flux,line=1)
flag=saturation[:,2,2]
p.addLayer(LayerXY(wave,flag,line=0))
```

10.12. Using the Status and BlockTables to perform selections on data

Most astronomers will not need to interact with the Status or BlockTable. However, these two can be used to select data chunks out of an observation, for example to work with or plot, and advanced users may wish to do so. Here we outline the methods that can be used to do this, starting by listing the columns of information that the Status and BlockTables contain.

10.12.1. The Status Table

The Status table generally has the same length as that of the readout/time axis of the flux dataset in a *Frames* or *PacsCube*, so for each astronomy datapoint, each column of the Status will have an entry. Most, but not all, of the Status entries are a single value, others are arrays.

The Status changes as the pipeline processing proceeds, it is added to as information is calculated (e.g. RA and Dec, spacecraft velocity) or modified (e.g. when the length of the *Frames* is reduced after the pipeline task *specDiffCs*), and information in the Status is used by other pipeline tasks.

To inspect the Status you can use the Dataset viewer or one of the plotters (access via the usual right-click on "Status" when you view your product via the Observation Viewer). The entries in the Status table are of mixed type—integer, double, boolean, and string. The ones that are plain numbers can be viewed with the Table/OverPlotter, the others you need to look at with the Dataset Viewer. (Note: over-plotting to see clearly the entries that have very different Y-ranges can be a bit difficult.)

Most of the Status columns are given in this table. For those not listed, hovering the mouse over the column label will bring up a tooltip about that column.

Table 10.2. Status information

Name	(S)pec/(P)hot,Type,Dim,Unit	Description
AbPosId	S+P Bool 1	on and off raster nod information (0=A, 1=B)
AcmsMode	S+P String 1	Attitude Control and Measurement System mode
AngularVelocityX/Y/Z Error	S+P Double 1 arcsec/sec	Angular velocity along the spacecraft axes
Aperture	S+P String 1	instrument aperture used
BAND	S+P String 1	band
BBID	S+P Long 1	Building block identifier
BBSEQCNT	S+P Int 1	building block sequence counter
BBTYPE	S+P Int 1	building block type
BLOCKIDX	S+P Int 1	a reference to the BlockTable: value taken therefrom
BSID	P Int 1	a 1 byte field (1 word used in the packet) containing instrumental information of the bolometers
CALSOURCE	S+P Int 1	calibration source number (0=not, 1 or 2)
CAPA	S Double 2 pF	Electric capacitance
CHOPFPUANGLE	S+P Double 1 degree	chopper position angle wrt the optical axis of the focal plane unit
CHOPPER	S Int 1	combination of CHOPPER-PLATEAU and CALSOURCE
CHOPPOS	S String 1	the chopper position (e.g. +/- throw)
CHOPSKYANGLE	S+P Double 1 arcmin	chopper position angle wrt as an angle on the sky
CPR	S+P Int 1	Chopper position as encoded by the MEC
CHOPPERPLATEAU	S+P Int 1	a number identifying that the chopper is in one of its set-positions
CRDC	S Int 1	Number of readouts since the last SET_TIME command to the DMC—used with TMP1 and 2 to calculate the FINETIME. Current ReadOut Count: current values of the readouts counter, starting from Nr and decreasing, a value of 0 signals a destructive readout and the end of an integration interval.

CRECR	S Int 1	CRDC number of readouts since the last SET_TIME command to the DMC.
CRDC	P Int 1	5x the number of OBT (on board) clock ticks since the last SET_TIME command to the DMC—used with TMP1 and 2 to calculate the FINETIME
CRDCCP	P Int 1	Current Readout Count in Chopper Position. This counter is reset each time the chopper starts moving
DBID	P	Data Block ID: of the block of detector arrays whose data are included in this packet (data in this timestamp).
DecArray/Err	S+P Double 1	Dec of the boresight (centre of the detector), from pointing product and FINETIME
DMCSEQACTIVE	S+P Int 1	DecMec sequence active? 0=no 1=yes
DP_WHICH_OBCP	S+P Int 1	On Board Control Procedure
DXID	S+P Int 1	Detector selection table identifier (this is a table which is used to select which detector pixels to read out)
FINETIME	S+P Long 1	Time in units of microsecond. Atomic time (SI seconds) elapsed since the TAI epoch of 1 Jan. 1958 UT2.
GRATSCAN	S Int 1	counter of grating scans
GPR	S Int 1	grating position an encoded by the MEC
GratingCycle	S Int 1	the index within the grating cycle, within the LABBA or ABAB chopping cycle
IndexInCycle	S Int 1	the index within the ABBA or ABAB chopping cycle
IsAPosition/IsBPosition	S+P Bool 1	at B or A nod position>
IsConstantVelocity	S+P Bool 1	is the satellite velocity constant?
IsOutOfField	S+P Bool 1	out of field?
IsOffPos	S+P Bool 1	off position flag
IsSerendipity	S+P Bool 1	serendipity mode; was the instrument active during e.g. slew periods
IsSlew	S+P Bool 1	slew of the satellite
LBL	S+P Int 1	Label
Mode	S+P String 1	observing mode
NodCycleNum	S+P Long 1	pointing nod cycle number

NrReadouts	Int 1	number of readouts that were used for the on-board-reduced value
OBSID	S+P Long 1	Observation identifier
ON/OFFSOURCE	S Bool 1	whether on or off source
OFF_RESETIDX	S Int 1	reset indices of the used off-source frames
OnTarget	S+P Bool 1	on target flag
PaArray/Err	S+P Double 1	Position angle of the boresight (centre of the detector), from pointing product and FINE-TIME
PIX	S+P Int 1	Counter for synchronisation to SPU housekeeping
PIXEL	S+P Int 1	the detector pixel number
RaArray/Err	S+P Double 1	RA of the boresight (centre of the detector), from pointing product and FINETIME
RasterLine/ColumnNum	S+P Long 1	raster line and column number
RAWSAT	Boolean	value of the rawsaturation mask (which contains values for one spectral pixel of the PACS detector only)
Repetition	S+P Int 1	repetition number
RESETINDEX	S+P Int 1	a counter of resets
RCX	S+P Int 1	(detector) Raw Channel Index
RollArray	S+P	obsolete; see PaArray
RRR	S Int 1	Readouts in Ramp (number)
SCANDIR	S Int 1	grating scanning direction
ScanLineNumber	S+P Long 1	scan line number
TMP1	S+P Int 1	Timing parameter: "time stamp" of the SET_TIME to the DMC, set only at the beginning of the observation. Unit of seconds
TMP2	S+P Int 1	see TMP1. In 1/65536 fractional seconds
Utc	S+P Long 1	UTC (not used during pipeline processing but is set by pipeline task)
VLD	S+P Int 1	Science data is value (0xff) or is not (0x00)
VSC	S Double 1	the spacecraft velocity to correct the wavelengths; negative is moving towards the source
WASWITCH	S Bool	wavelength switching mode or not
WPR	S+P Int 1	Wheel position encoded by the MEC

10.12.2. Selecting discrete data-chunks using the Status

Here we show a few ways to work with Status data, including using it to select on the astronomical data (the fluxes). **Note:** The intension is to show the methods that can be used to work with these data: the code itself will only produce a plot at the end for a Frames taken from Level 0.5 (from Level 1 there is no "off" chop data), for a chopNod AOT (not for wavelength switching or unchopped rangeScans, which do not have "on" and "off" chop within the same observation), and for a "large" throw (other choices are "small" and "medium").

```
# Preamble: to test the following on an example observation
obs = getObservation(1342238131, useHsa=1)
myframe = obs1.level0_5.red.fitted.product.refs[1].product
# a red frames, level 0.5 -> change as you wish

# What are the dimensions of a Status column in a Frames
print myframe.getStatus("RaArray").dimensions

# Two ways to read Status data
# what are all the "BAND"s in a Frames
print UNIQ(myframe.status["BAND"].data)
print UNIQ(myframe.getStatus("BAND"))

# Extract the Status column called CHOPPOS from a Frames
# You can do the same for other columns
# Result will be a DoubleIcd, IntIcd, or StringIcd, in most cases
stat = myframe.getStatus("CHOPPOS")

# Extract out certain data using Status values
# get the on chops: here the throw is "large";
# "small" and "median" are also values
on = (stat == "+large") # a boolean, true where +large
on = on.where(on) # turn that into a Selection array
# get the off chops
off = (stat == "-large")
off = off.where(off)
wave = myframe.getWave(8,12)
offwave = wave[off]
# To check this, you could plot the selected data
p=PlotXY(myframe.getWave(8,12)[on],myframe.getSignal(8,12)[on])
p.addLayer(LayerXY(myframe.getWave(8,12)[off],myframe.getSignal(8,12)[off]))
```

More scripts following the example here can be found in [Section 10.4.5](#). You can also look up the *PacsStatus*, *Frames* and *PacsCubes* classes in the [PACS DRM](#) to see what (java) methods they have. And to learn more about scripting in HIPE, we refer you to the [SG](#).

10.12.3. BlockTable and MasterBlockTable

The BlockTable is a listing of the data blocks of your *Frames* and *PacsCubes*. In the spectroscopy pipeline we slice our data according to a (mainly) astronomical logic (pointing, wavelength, etc.), therefore, as well as the BlockTables of the individual slices of the *SlicedFrames* or *SlicedPacsCubes*, there is a MasterBlockTable that is a concatenation of the individual BlockTables of all the slices.

10.12.4. Selecting discrete data-chunks using the block table

Blocks are chunks of data grouped following an instrument and/or astronomical logic: for example two grating scans will be two blocks, and two pointings in a raster will be two blocks. The data themselves are not organised in this block structure, rather the BlockTable contains pointers to the data so that tasks can identify where the blocks are by querying with the BlockTable. It also gives a handy overview of your observation. Blocks are created in and used by the pipeline.

To find the BlockTable to look at it, click on the "+" next to a *Frames* or *PacsCube* listed in the Observation Viewer. To see the MasterBlockTable for a *SlicedFrames*, click on the + next to the HPSFIT[R|B] in the Observation Viewer. Open the Dataset inspector (right-click on the BlockTable to access the viewer via the menu) to see something similar to this screenshot:

Index	Obcp	DMSActive	ChopperPlateau	CalSource	Filter	Start...	EndIdx	NrIdx	StartFine...	EndFineT...	Raster	Id	Description	OnSource	OffSource1	OffSource2	Nodc
0	111	1	2	0	1	0	150	150	1629773...	1629773...	0.0	UNDEFINED	Undefined	-1	-1	-1	0
1	0	1	2	0	0	150	152	2	1629773...	1629773...	0.0	UNDEFINED	Undefined	-1	-1	-1	0
2	35	1	2	0	0	152	164	12	1629773...	1629773...	0.0	SPEC_CHO...	Chopped ...	3	5	-1	2
3	0	0	0	0	0	164	166	2	1629773...	1629773...	0.0	UNDEFINED	Undefined	-1	-1	-1	0
4	35	1	1	1	0	0	144	144	1629773...	1629773...	0.0	SPEC_CHO...	Chopped ...	3	5	-1	2
5	35	1	1	1	0	144	288	144	1629773...	1629773...	0.0	SPEC_CHO...	Chopped ...	19	21	-1	2
6	35	1	1	1	0	288	432	144	1629773...	1629773...	0.0	SPEC_CHO...	Chopped ...	3	5	-1	2
7	35	1	1	1	0	432	575	143	1629773...	1629773...	0.0	SPEC_CHO...	Chopped ...	19	21	-1	2
8	0	0	0	0	0	575	637	62	1629773...	1629773...	0.0	UNDEFINED	Undefined	-1	-1	-1	0
9	35	1	1	0	0	144	144	144	1629773...	1629773...	0.0	SPEC_CHO...	Chopped ...	3	5	-1	2
10	35	1	1	0	0	144	287	143	1629773...	1629773...	0.0	SPEC_CHO...	Chopped ...	19	21	-1	2

Figure 10.19. The MasterBlockTable

The entries that identify nod, grating scan (direction), raster pointing, wavelength range, etc. can be found in the table below. Many of the BlockTable columns come directly from the Status, and so can be similarly understood. If you hover with the cursor over the column titles, a banner explanation will pop up. The BlockTable entries are:

Table 10.3. BlockTable columns

Name	Description
Band	spectral band
CalSource	0,1,2 for neither, calsource 1, calsource 2
ChopperPlateau	taken from the Status table
Description	explanation of keyword
DMSActive	DecMec sequence active? 0=no 1=yes
Filter	filter
FramesNo	slice number, the <i>Frames</i> number in the <i>Sliced-Frames</i> that this block comes from
GenericOnOffSource	0=not science, 1=on, 2=off, -1=not defined (i.e. the block has chopping, so both on and off are included)
GPRMin/Max	smallest and largest grating position
Id	keyword describing this block
index	a simple counter
IsOutOfField	true=out of field
LineDescription	line description taken from the Meta data
LineId	line identification number (a simple counter)
Min/MaxWave	smallest and largest wavelength
NodCycleNum	nod cycle number
NoddingPosition	0=none, 1=A, 2=B
NrIdx	number of indices in this block
Obcp	on board control procedure number
OffSource1/2	first and second off-source position label (some AOTs have 2 off positions)
OnSource	on-source position label
Raster	raster number
RasterColumn/LineNum	raster column and line numbers

Repetition	repetition cycle number (used in photometry)
ResLen	reset length (number of resets that fed into each readout)
ScanDir	scanning direction (0,1)
Start/EndFineTime	start and end FINETIME of this block
Start/EndIdx	starting/end index of this block
WaSwitch	wavelength switching active or not

In the MasterBlockTable, be aware that, per block, the StartIdx and EndIdx entries are relative to the slice that is that block. The entry that identifies the slice number is "FramesNo".

To use the [Master]BlockTable to query your data you can use the following examples

```
# Preamble: to test the following on an example observation
obs=getObservation(1342209728, useHsa=1)
slicedFrames = obs.level0_5.red.fitted.product
myframe = slicedFrames.refs[1].product
slicedCubes = obs.level1.red.cube.product # red cubes, level 1
mycube = slicedCubes.refs[0].product

# Get listing of all columns of the BlockTable of a Frames
# and the MasterBlockTable of a SlicedFrames
print myframe.blockTable
print slicedFrames.masterBlockTable
print mycube.blockTable
print slicedCubes.masterBlockTable

# Select the blocks "SPEC_UPSCAN_ALL" and "SPEC_DOWNSCAN_ALL"
# and create two new Frames from them
# This command will overwrite "myframe" so copy it first
myframe_cp = myframe.copy()
myframe_up=myframe.select(myframe.getBlockSelection("SPEC_UPSCAN_ALL"))
myframe = myframe_cp.copy()
myframe_down=myframe.select(myframe.getBlockSelection("SPEC_DOWNSCAN_ALL"))

mycube_cp = mycube.copy()
mycube_up=mycube.select(mycube.getBlockSelection("SPEC_UPSCAN_ALL"))
mycube = mycube_cp.copy()
mycube_down=mycube.select(mycube.getBlockSelection("SPEC_DOWNSCAN_ALL"))

# To be sure this has worked, plot the data from one
# detector pixel
p=PlotXY(myframe_up.getWave(8,12), myframe_up.getSignal(8,12),line=0)
p.addLayer(LayerXY(myframe_down.getWave(8,12), myframe_down.getSignal(8,12),line=0))
p[0].setName("up")
p[1].setName("down")
p.getLegend().setVisible(True)

p=PlotXY(mycube_up.getWave(2,2), mycube_up.getFlux(2,2),line=0)
p.addLayer(LayerXY(mycube_down.getWave(2,2), mycube_down.getFlux(2,2),line=0))
p[0].setName("up")
p[1].setName("down")
p.getLegend().setVisible(True)

# The easiest way to select from a SlicedFrames, a slicedPacsCubes
# or SlicedPacsRebinnedCubes, i.e. from an entire observation, not just
# working on one slice at a time, is with the pipeline helper task
# "selectSlices". This is taken from the pipeline scripts: to select
# on any of these parameters, enter the selection keywords in the relevant
# []. The output of this task is another SlicedFrames|PacsCubes.
verbose = 1
lineId = []
wavelength = []
rasterLine = []
rasterCol = []
nodPosition = ""
nodCycle = []
```

```

band      = ""
scical    = ""
sliceNumber = []
sCubes = selectSlices(slicedCubes, lineId=lineId, wavelength=wavelength,\
    rasterLine=rasterLine,\
    rasterCol=rasterCol, nodPosition=nodPosition, nodCycle=nodCycle,\
    band=band, scical=scical,\
    sliceNumber=sliceNumber, verbose=verbose

# A listing of the PacsCubes (or Frames) in a slicedCubes or slicedFrames
slicedSummary(slicedCubes)

# More general examples for querying a SlicedCubes(or Frames) to identify
# which slices satisfy a condition, and then select them out.
#
# Get the raster column numbers from a SlicedFrames
nodSlicesCol = slicedCubes.getMasterBlockTable().getRasterColumnNumber()
framesNo = slicedCubes.getMasterBlockTable().getFrameNumbers()
# select on it, e.g. you want the slices from raster column 2:
select = UNIQ(framesNo.get(nodSlicesCol == 2))
# Now select out the slices that have this column number and place the
# in a new SlicedfCubes
select = select.getArray()
sCubes = selectSlices(slicedCubes,sliceNumber=select)

```

You can look up the *Frames* and *PacsCubes* classes in the [PACS DRM](#) to see the other methods that work with these classes. Bear in mind that since the data at Level 0.5 onwards is already sliced by raster position, wavelength, etc., within any one slice you cannot usefully select for these aspects. However, similar methods to those given here will work on a *SlicedFrames* or *SlicedPacsCubes*, allowing you to select on the entire observation. To learn the syntax of these methods, look up these classes in the [PACS DRM](#) to see what methods are provided for them. And if you completely extract the [Master]Block-Table from a [*Sliced*]Frames (e.g. HIPE>status=myframe.getBlockTable()), the [PACS DRM](#) entry for the its class (*ObcpBlockTable* or *MasterBlockTable*) can be consulted to see what methods you can use on that. Finally, to learn more about scripting in HIPE, we refer you to the [SG](#).