# SPIRE Data Users Manual

**version 1.0.<undefined> , Document Number: SPIRE-RAL-DOC 003248**
**30 July 2010**

# SPIRE Data Users Manual

# Table of Contents

# Preface

# 1. Versioning

On the front page of this manual is a version number made of three digits. The first two digits follow a traditional versioning system (0.1, 0.2, ...), and the changes introduced with each version are detailed below. The third digit is the SPIRE build number to which each edition of the manual is associated. Also shown on the front page is the date of publication of the manual.

## 1.1. Changelog

**The following was changed for 1.0**

- Major updates to all sections to conform to data products and data processing as of the 4.0 branch.
- Added SPIRE Calibration chapter.
- Added aditional section on the Spectrum Explorer for SPIRE.
- Added reprocessing section for Small Map Mode.
- Expanded reprocessing section for spectrometer pipeline.

**The following was changed for 0.2**

- Updates to flow charts with respect to the 4.0 branch.

**The following was changed for v0.1**

- First version of the SDUM manual.

# Chapter 1. Introduction

## 1.1. Scope of this Data User's Manual

The purpose of this document is to provide a comprehensice reference for all SPIRE users in terms of the data structure users will encounter for on inspection of the different types of SPIRE observations, but also as a guide on how to reprocess the data and inspect the products through the full SPIRE pipeline. This document superceeds the SPIRE pipeline reduction formerly included in the HOWTOs document, but has been expaned to include all modes and insights on the data struture and types.

The data structure and reprocessing guide examples contained within the SPIRE Data Users Manual are based upon the HIPE 4.0 release - views may differ and examples may not work on previous and subsequent releases of HIPE.

For more information on obtaining HIPE and on how to install it, getting started with it, please go to the HIPE Quick Start Guide and the HIPE Owners Guide for a more more indepth overview of getting started with the HIPE environment.

## 1.2. SPIRE observing Modes

SPIRE observing modes for both the Photometer and the Spectrometer are provided as Astronomical Observation Templates (AOTs), and the way these AOTs are referred to may differ from resource to resource (Hspot, HIPE, etc). There are currently 6 available observaing modes in various levels of use and release, these are,

- **Large Map Mode(Scan Mapping, POF5):** Used for observations of large fields (>4x4 arcmins). The telescope is scanned building up a map, scan line by scan line. Scan lines can be orthonally cross-linked to produce high quality maps.
- **Small Map Mode (1x1 Small Scan Map, POF10):** Used for observations of large fields (>4x4 arcmins). This mode replaces the former small map 64-point Jiggle, POF3 mode. The new Small Scan Map mode consists of 2 orthoganal scan lines of fixed length. The mode operation and processing is essentially the same as the Large Map mode. For a given observation, the area covered by both scan legs defines a central square of side 5 arcmins although the length of the two orthogonal scan paths are somewhat longer than this. In practice, due to the position of the arrays on the sky at the time of a given observation, the guaranteed area for scientific use is a circle of diameter 5 arcmins.
- **Point Source Mode (7-point Jiggle, POF2):** Used for observations of point sources. The telescope stares at a target and the detector arrays are jiggled, using BSM, over the target using a 7-point pattern. The background is removed by chopping with the BSM and Nodding with the telescope.
- **Parallel Mode (Parallel):** Used for maps created with both SPIRE and PACS in parallel. These are essentially equivalent to Large Map observations.
- **Point Source Spectroscopy (SOF1):** Used for point source spectrocopy. The Spectrometer Mechanism (SMEC) mirror is scanned to produce a spectrum over the full wavelength range
- **Small Map Spectroscopy (SOF2):** Used for creating small spectrocopic maps. The Spectrometer Mechanism (SMEC) mirror is scanned to produce a spectrum over the full wavelength range while the BSM jiggles over 16 positions to produce an image map.

## 1.3. Structure of this document

Astronomer users will receive data that has already been processed through the standard pipelines to several Levels. The processing levels of the SPIRE pipeline and user deliverables are outlined below in Figure 1.1.

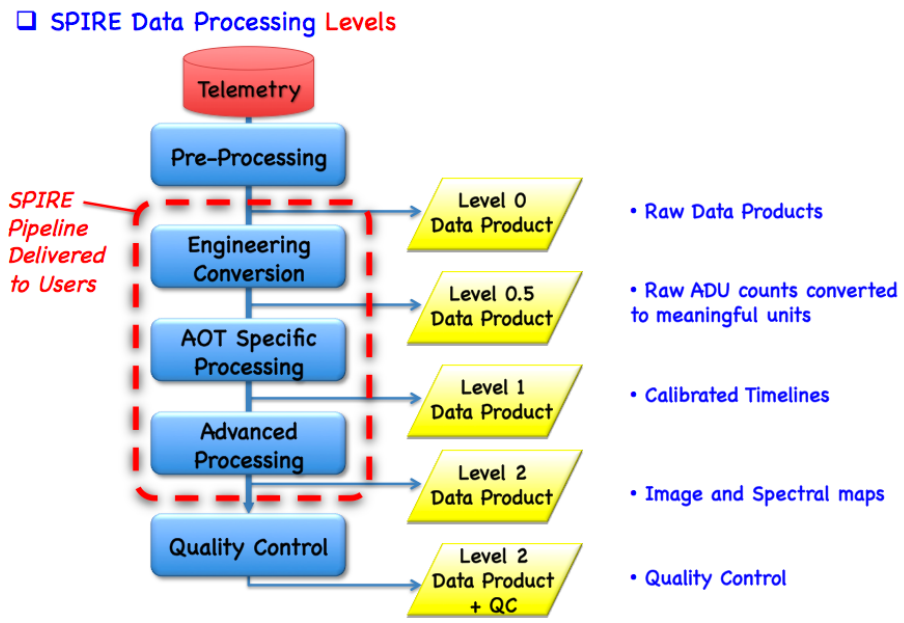


**Figure 1.1. The processing levels of the SPIRE pipeline and user deliverables.**

This document is divided into two broad topics. An introduction to the data structure as received from the Herschel Science Archive (HSA) is described in Chapter 2 which includes all relevant observation modes and processing Levels. The pipelines themselves and details on reprocessing your observations are covered in Chapter 4.

# Chapter 2. Looking at your data

## 2.1. SPIRE Observation Context Data Structure

### 2.1.1. Anatomy of a SPIRE Observation: Products, Pools, Storage, and Building Blocks

For the purposes of both this chapter and the next (on reprocessing your data), we assume that you have already downloaded a data set from the Herschel Science Archive and are familiar with how to put your data into a store and how to access your data from this store within HIPE. If you haven't, please look at the HIPE Quick Start Guide and the HIPE Owners Guide for instruction on how to do this.

Now you are the proud owner of a set of SPIRE observations. Before carrying out any processing its most likely that you will want to have a first look at your data. SPIRE observations are supplied in a highly organized structure that may be unfamiliar to previous astronomical datasets you have encountered.

All data within the HCSS processing system are passed around in containers referred to as **Products**. There are Products for every kind of data, e.g.;

- Raw and processed Detector Data Timelines
- Calibration Data
- Auxiliary (e.g. Pointing) Data
- Images
- Image Cubes
- Data Contexts
- .....

Products can contain the following (pictorially visualized in Figure 2.1);

- Meta Data
- One or more Datasets
- Processing History

**Datasets** can be;

- Array Tables
- Image arrays
- Composite (nested) Tables
- .....

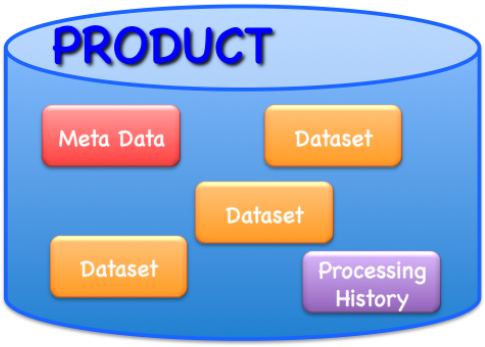


**Figure 2.1. General structure of a SPIRE data Product**

SPIRE (Herschel) Observations are accessed/downloaded and stored as a Pool of these products. A **Pool** is basically a directory that contains the original raw data, the results of the automatic pipeline processing and everything you need to process your observations again yourself ( e.g. spacecraft pointing, the parameters you entered in HSPOT when you submitted the proposal, and the pipeline calibration tables). Data that you reprocess yourself can also be stored into the same Pool or you may alternatively wish to save the results in a new Pool. If you wish to send someone a set of processed data for example, the entire Pool directory should be "tar"ed or archived and sent. Finally, once a Pool has been created, the pool's directory name must NOT be changed or HIPE will not be able to find the data.

In general, HIPE expects all your observation pool directories to be contained in a "**Local Store**" directory which can be thought of as a Super Repository for all Observation Pools on your hard disk. By default this directory resides in ~/.hcss/lstore but can be changed and renamed by by editing the HCSS user.props file. The structure of the Local Store is visualized in Figure 2.2

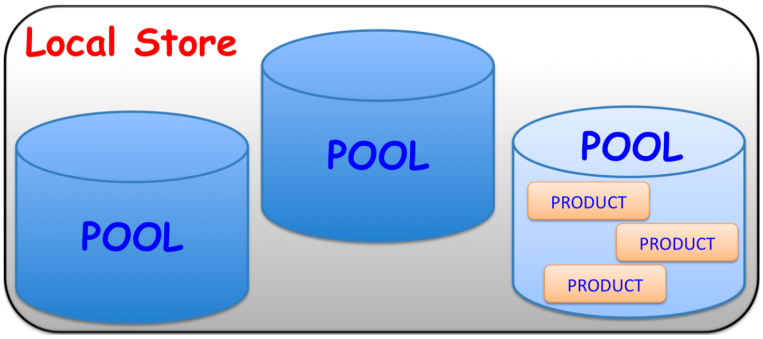


**Figure 2.2. General structure of the Local Store**

## 2.1.2. Linking it altogether: Introducing the Context

The smallest “piece” of SPIRE observational data is called a **Building Block**. These Building Blocks correspond to basic operations within an observation and as the name suggests every SPIRE AOT is built up from a combination of these building blocks. Building Blocks are usually in the form of Timeline Data Products.

Example building blocks may be;

- A scan line in a map
- A single 7 point Jiggle
- A set of Spectrometer scans

- A segment of housekeeping scans
- A motion of the Beam Steering Mirror (BSM)

Building Blocks and other Products are grouped into a context. A context is a special kind of product linking other products in a coherent description and can be thought of as an inventory or catalogue of products. The SPIRE processed observation consists of many such contexts within one giant **Observation context**. Therefore, Each set of building blocks have a context. Each Processing Level in the SPIRE pipeline has a context and the entire Observation has a context. Thus a complete observation may be thought of as a big SPIRE onion as depicted in Figure 2.3. Moreover, contexts are not just for building block products and higher processed data products, there are contexts for Calibration Products and contexts for Auxiliary Products (e.g. pointing) and even a context for Quality Control. The entire SPIRE Obseravtional Context is shown in Figure 2.4 for all products from the raw building block data to the final high level processed end products from the pipeline. This is the structure and content that you should receive for your SPIRE observation from the Herschal Science Archive (HSA).

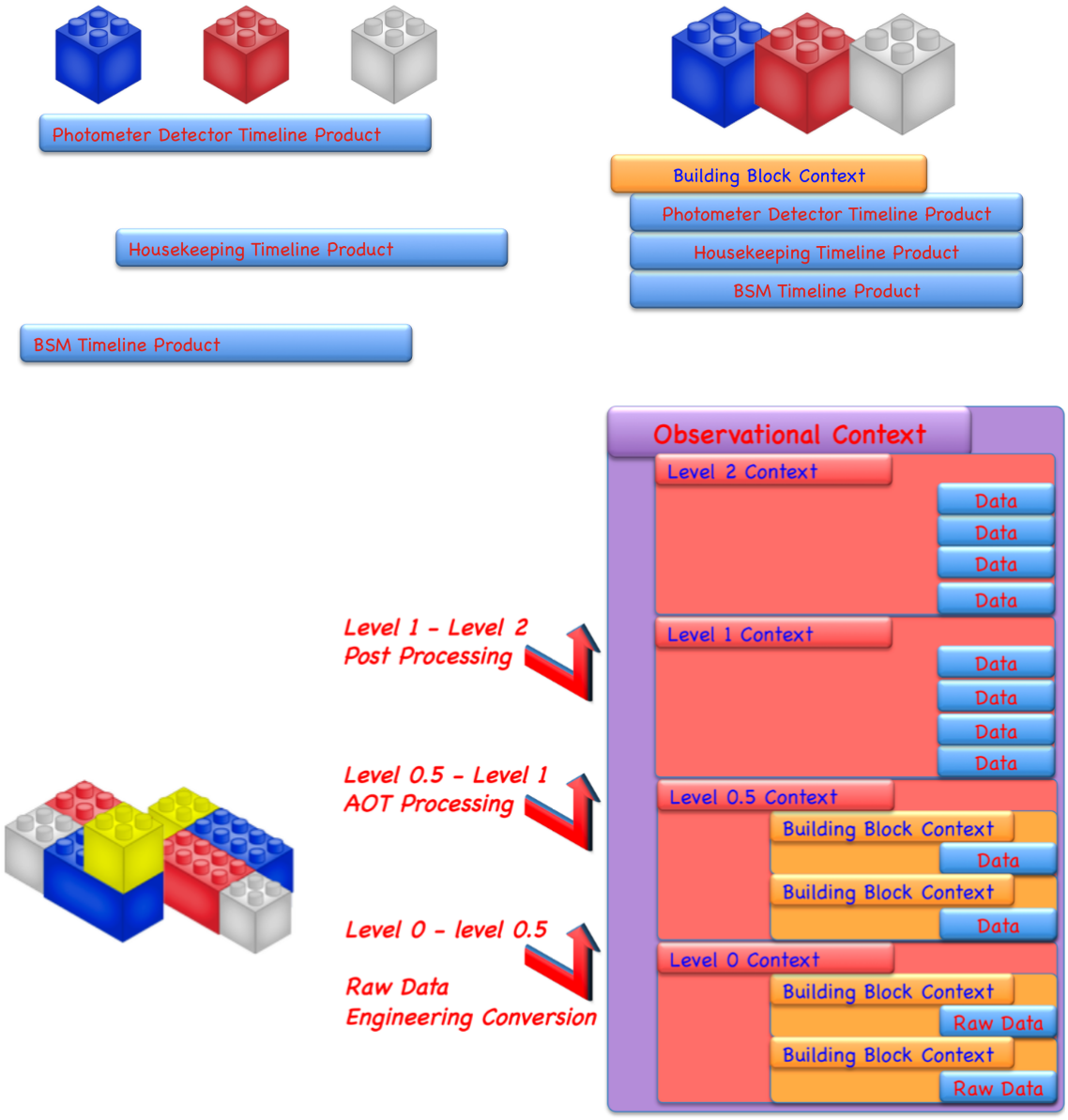


**Figure 2.3. The Context structure within HCSS. The smallest "piece" of SPIRE observational data are Building Blocks. Building Blocks and other Products are grouped into a context. All the data within an entire SPIRE observation are linked by an Observation Context.**

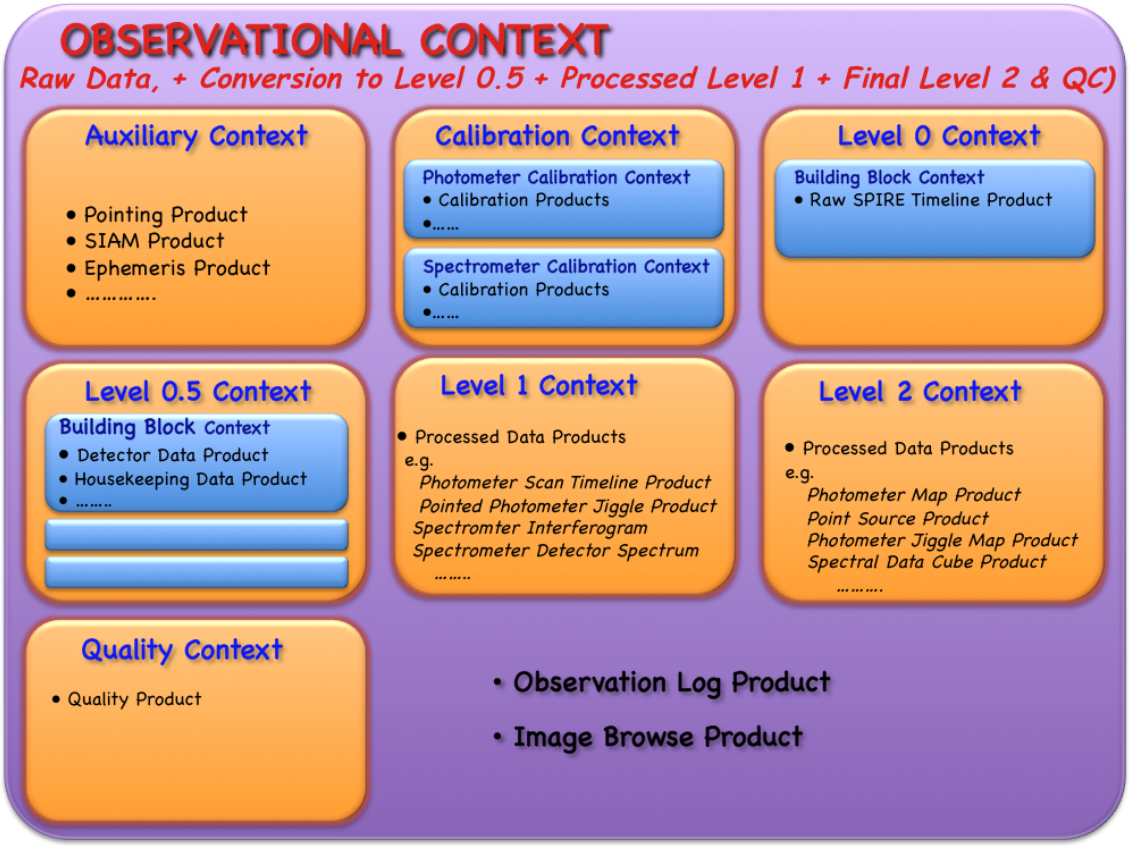


**Figure 2.4. The complete Observation Context of a SPIRE observation**

## 2.1.3. Looking at your Observation Context in HIPE

The Observation Context can be viewed directly within HIPE. It is assumed in this example that the data has already been downloaded from the archive and has already been stored in a pool named `GalaxyScanMap` in the Local Store. We therefore have to load this pool into the HIPE environment and extract the Observation Context for this observation. This is possible via a slightly convoluted route using the GUI but can also be accomplished painlessly with a the few lines of code shown below;

```
Pool  = -'GalaxyScanMap'                              # Select the pool name
storage=ProductStorage(Pool)                         # Register the pool
queryResults = storage.select(Query("type=='OBS'"))  # Query the pool
MyObsContext = queryResults[0].product               # Extract the  Context
```

The first line of code selects the desired Pool from our Local Store on disk. This Pool is read in to a storage area in memory (referred to as *Registering the Pool*") which we have decided to call `storage`. Once the Pool has been registered, it can then be *queried* for the observation context by searching the storage for the Product Type `OBS`. Finally, the Observation Context Product is stored in a variable we choose to call `MyObsContext`. After running the above lines we see five new entries `Variables` pane of HIPE shown in Figure 2.5. These variables have already been described above (Note: the `p` is simply a place holder). Double clicking on the `obsContext` in the variable list brings up the *Observation Context* observation in a new window as also shown in Figure 2.5. The Observation Context has `Summary`, `Meta-Data` and `Data` panes. The `Summary` pane contains information on the instrument, target position, observation ID, Operational Day and Observation Mode. The `Meta-Data` pane contains all relevant information on the Product necessary to describe and process the observation (including the information in the `Summary` pane). The Meta-Data for the observation context is summarized in Table 2.1. The Observation Context `Data` pane contains pointers to all other

contexts and data products contained in the Observation Pool. The `Data` pane contains many entries, listed below and in Figure 2.6 (See also Figure 2.4);

- `level 0`: The Level 0 context containing links to the Level 0 raw Data before any pipeline processing.
- `level 0.5`: The Level 0.5 context containing links to the Level 0.5 data products after the common engineering conversion has been made.
- `level 1`: The Level 1 context containing links to the Level 1 data products after AOT specific pipeline processing.
- `level 2`: The Level 2 context containing links to the final Level 2 data products from the pipeline.
- `calibration`: The Calibration context pointing to all calibration products required for the processing of SPIRE data.
- `auxiliary`: The context pointing to all .
- `logObsContext`: The context pointing to the reduction log that records the processing history of the data.
- `quality`: The Quality context pointing to the quality control products for this observation.
- `browseImageProduct`: The context pointing to thumbnail products.
- `browseProduct`: The context containing information from the HSA archive.

Note that the structure of the Observation Context can also be directly seen from the command line by typing, `print MyObsContext`;

```
HIPE> print MyObsContext
{description="Unknown", meta=[type, creator, creationDate, description, instrument,
 modelName,startDate, endDate, obsState, obsid, odNumber, cusMode, instMode],
datasets=[], history=None,
refs=[auxiliary,browseImageProduct,browseProduct,calibration,level0,
level0_5,level1,level2,logObsContext,quality]}
```

Here the Observation Context can be clearly seen to contain no data as such but rather a set of pointers or references to other different kinds of contexts. In the next section, the Observation Contexts for specific individual AOTs will be investigated in more detail allowing us to have a first look at our processed data!

**Table 2.1. Description of Meta Data in the SPIRE Observation Context**

| Meta Data | Description |
|---|---|
| odNumber | The Observational Day when the observation was made |
| obsid | The unique Observation ID (in decimal) |
| startDate | The start date of the observation in TAI, Zulu Time |
| endDate | The end date of the observation |
| creationDate | The creation date of this Product |
| creator | How the product was created (e.g. Standard Product Generation (SPG) version) |
| modelName | Whether the data is from Flight or Flight Spare, etc |
| obsState | |

| Meta Data | Description |
|---|---|
| | How far has the observation been processed by the pipeline (Level 0, 0.5, 1 or 2) |
| type | The Product Type (OBS = Observation Context) |
| instMode | The instrument mode (The AOTs defined internally as POF5 for Large Map Mode) |
| instrument | The instrument name, in this case SPIRE |
| cusMode | How the AOT is referred to in the observaion logs and scheduling (SpirePhotoLargeScan) |
| description | The Product name |

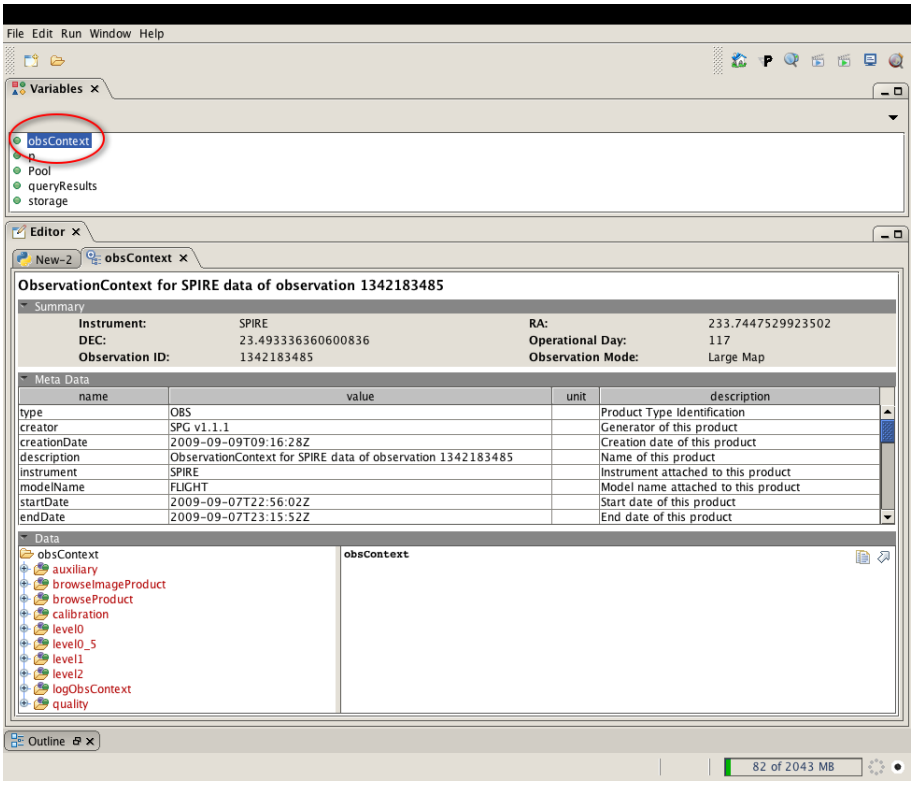


**Figure 2.5. The Observation Context within HIPE**

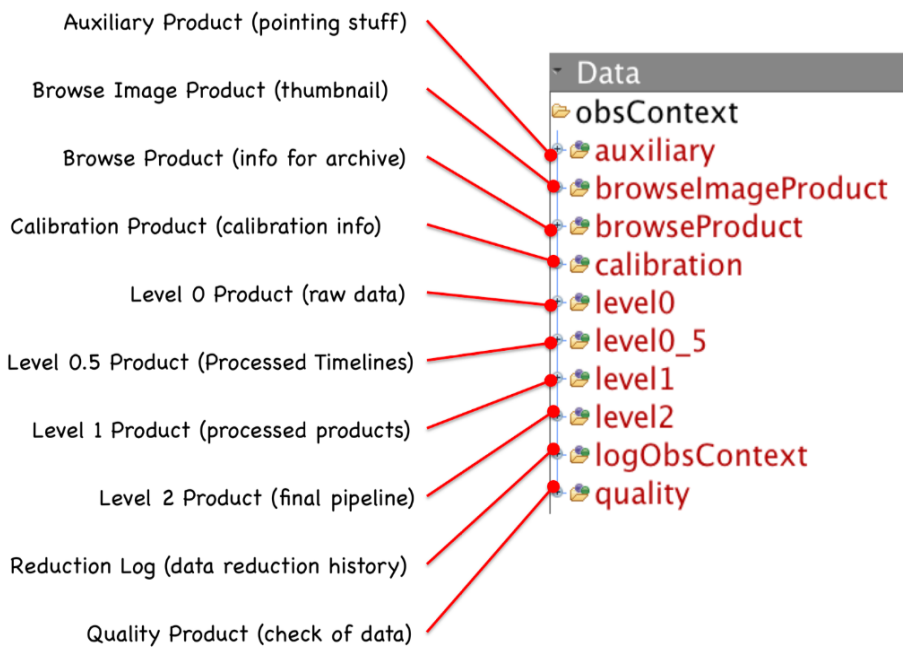


**Figure 2.6. Inside the Observation Context within HIPE.**

# 2.2. SPIRE Large Map and Parallel Mode Data Structure

## 2.2.1. A first look at your image maps (The Level 2 Data Product)

All the information for a given SPIRE observation is contained with the Observation Context (described in Section 2.1). In this section we shall see how to examine the data for a SPIRE **Large Map** observation, however this description applies equally to SPIRE **Parallel Mode** observations.

The observation we shall be looking at is a Large Map observation of the Planetary Nebulae NGC5315 taking during the Herschel-SPIRE PV phase. NGC5315 is at RA=13h53m57.00s, dec=-66d30'50.70" and was covered by scanning the photometer arrays 3 times each in orthoganal direction. The entire process was then repeated (i.e. this observation has 2 repetitions) giving in total 6 scans in each orthoganal direction making 12 scan lines in total.

It is assumed that the observation has already been downloaded into a Pool within your Local Store on your computer as described in section Section 2.1. The Observation Pool can be loaded into HIPE using the following 4 lines of Jython Code (where the **Pool** is whatever name you called your Pool for this observation in your Local Store on disk;

```
Pool  = -'OD117-ScanNGC5315-0x50001833'              # Select the pool name
storage=ProductStorage(Pool)                        # Register the pool
queryResults = storage.select(Query("type=='OBS'")) # Query the pool
MyObsContext = queryResults[0].product              # Extract the  Context
```

For this particular observation, we chose to call our Pool **OD117-ScanNGC5315-0x50001833** where **OD117** means the observation was made on Operational Day 117, **Scan** was the AOT mode,

**NGC5315** was the target name and **0x50001833** is the unique Observation ID in hexadecimal. Running the above script, reads the Observation Context into memory into the variable **MyObsContext** which appears in the **Variables** pane of HIPE (See Figure 2.7). Right Clicking (or CTRL-click for Apple Users) on the **MyObsContext** variable brings up another menu. Selecting `Open With -- Observation Viewer` will open the Observational Context for this observation. The structure of the Observation Context was explained in Section 2.1 and he we shall look at the data inside the Observational Context. We start with the final Product of the SPIRE Large Map pipeline - the image maps. The maps are Level 2 Products and can therefore be found within the Level 2 Context. The maps can be simply acccesed by clicking on the **level2** folder as shown in Figure 2.8, which reveals a SPIRE Photometer Map Product (or more technically `SimpleImage` Products) for each of the three SPIRE arrays (PSW, PMW, PLW). Each Photometer Map Product contains 3 Table Datasets corresponding to the image, error and coverage maps for each array and these are revelaed by *clicking* on the + sign next to the array folder.

The image map can be viewed by clicking on the appropriate array folder (PSW, PMW, PLW) or alternatively the image map can be displayed in a new window by right clicking on the appropriate array folder and selecting `Open With - Standard Image Viewer` from the drop down menu as shown in Figure 2.9. This action opens the image in the `Image Viewer` where the image can be panned, magnified etc. Colours, cut-levels, annotation options can be accessed by *right-clicking* anywhere on the image. The image, error and coverage maps can also be displayed individually by *clicking* on them or by *right-clicking* on the appropriate dataset and selecting `Open With - Image Viewer for ArrayDatasets` from the drop down menu. Finally, *right-clicking* on a given image dataset and selecting `Open With - Array Dataset Viewer` from the drop down menu shows the image (or error or coverage) in table form (Jy/beam for every pixel in the image) as shown in Figure 2.10.

If you want to extract the `SimpleImage` for the PSW, PMW or PLW array as a data cube containing the image, error and coverage maps to work with, rather than view it with the `Image Viewer`, on the command line type the rather exhaustive:

```
MyMapProduct=MyObsContext.refs["level2"].product.refs["PSW"].product
# Then to view each of the map datasets
Display(MyMapProduct.image)
Display(MyMapProduct.error)
Display(MyMapProduct.coverage)
```

where `MyMapProduct` can be any name we choose and the following syntax means from `MyObsContext` we want the Level 2 product PSW array Photometer Map Product. You will also notice that `MyMapProduct` now appears in the Variables Panel which can correspondingly be *right-clicked* on to show the various viewing options available for this product. The next 3 lines in the above script allow us to display the signal, error and coverage maps respectively.

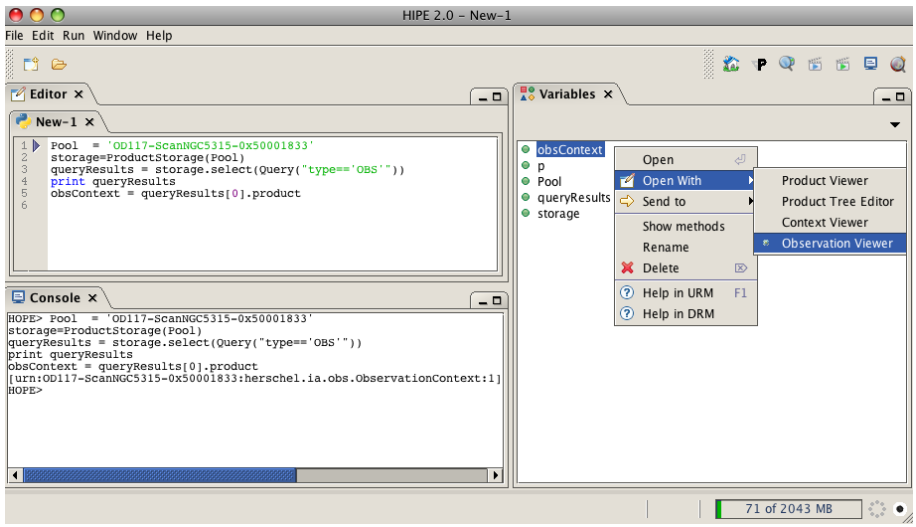


**Figure 2.7. Loading and viewing the Observation Context for the Large Map Observation.**

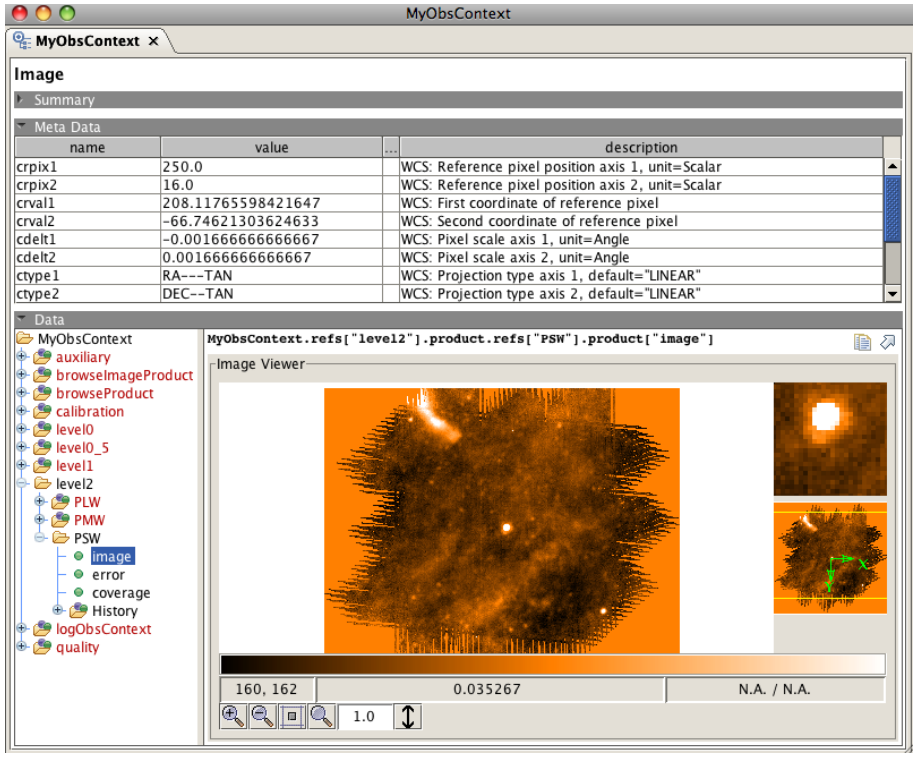


**Figure 2.8. Accessing the final Level 2 Product maps**

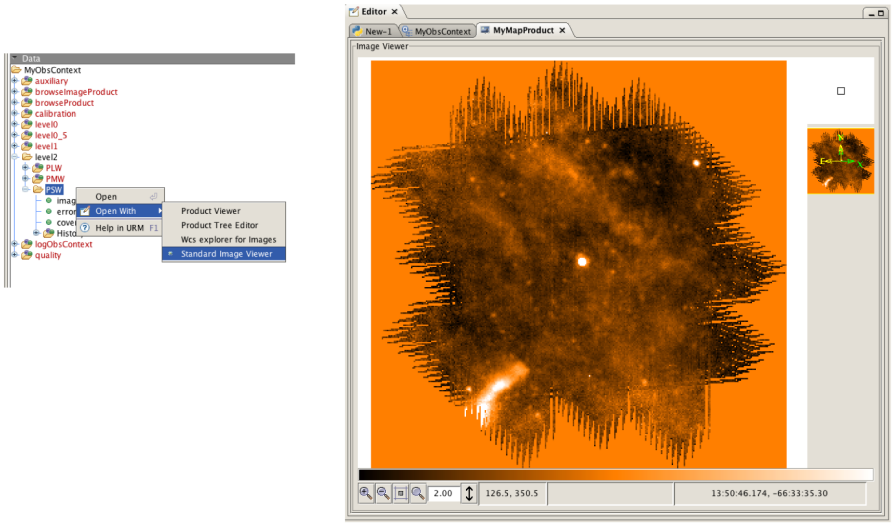


**Figure 2.9. Viewing the Level 2 Image Maps**

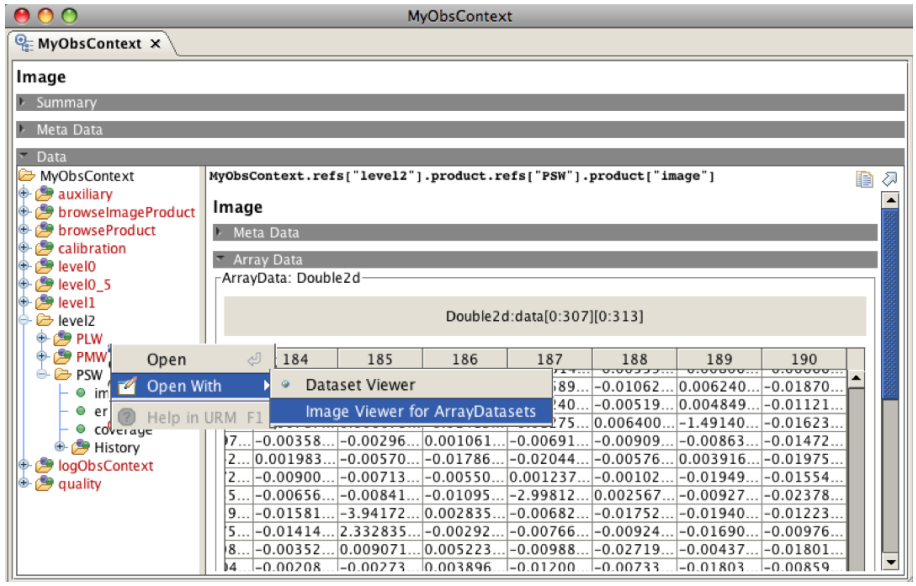


**Figure 2.10. Viewing the Level 2 Image Array Datasets**

## 2.2.2. Saving a map as a FITS file and reading it in again

It is possible that me may also want to look at our image maps in external applications such as DS9 for example and HIPE provides the tools for exporting our maps as conventional fits files. Following on from the previous example above we can send our `MyMapProduct(SimpleImage)` product to a FITS file by *right-clicking* on it in the variable list and selecting `Send To - FITS file` from the drop down menu. This will open the FITS writer panel as shown in Figure 2.11 where we can type in our desired filename and path. Click on `Accept` at the bottom of the panel to save the FITS file. This fits file will then be saved as a multi-extension fits file containing the image, error and coverage maps that can then be read into DS9 as a data cube and viewed. The same effect can be acheived on the command line by;

```
FitsArchive().save('mypath/myMap.fits', MyMapProduct)
```

which again saves the products as a multi-extension fits file containing the image, error and coverage maps.

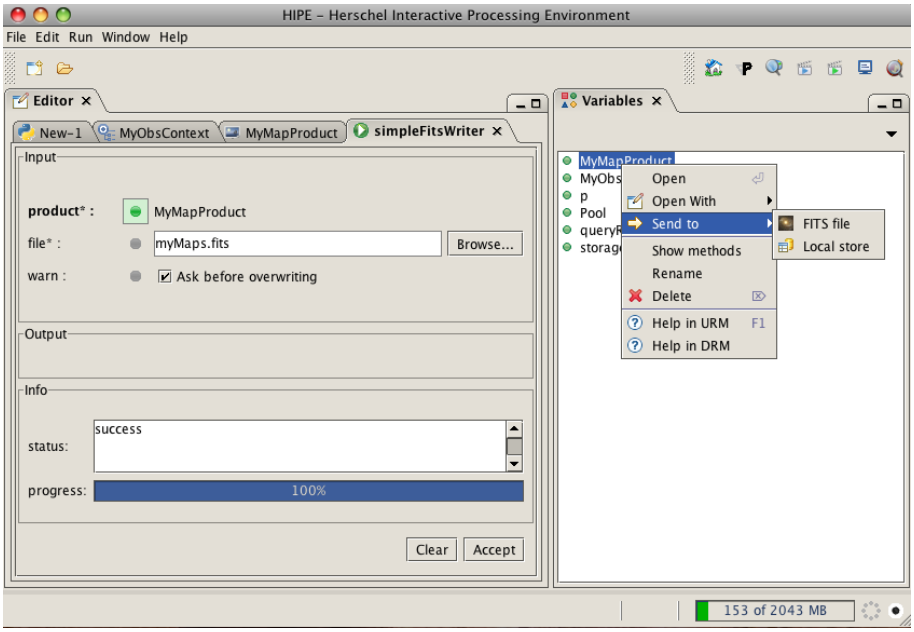


**Figure 2.11. Exporting Image Maps as FITS files**

Reading a FITS file into the HIPE session can be accomplished by either selecting `Open File` from the `File` menu in the top right hand corner of the HIPE window. Alternatively, from the command line;

```
myMap=simpleFitsReader('mypath/myMap.fits')
```

These FITS files are imported as a `simpleImage` and can be manipulated in the same manner as the `simpleImage` products described earlier in this section.

**Note**

The Photometer Map Products (data cubes for each array containing the image, error and coverage arrays) actually exist as fits files within the **Pool** for this observation in the Local Store. These can be found in the Pool for this example in the folder `/localstore/OD117-ScanNGC5315-0x50001833/herschel.ia.dataset.image.SimpleImage` (where the poolname is "OD117-ScanNGC5315-0x50001833"). The Photometer Map Products have the form `hspireplw.........pmp.fits`

## 2.2.3. Looking at the Level 1 Timeline Data

The image maps have been created from the individual timelines of detectors as they were scanned accross the target. These timelines are the Level 1 products from the Photometer Large Map Pipeline and are also available from the Observation Context. The Level 1 Large Map products are referred to as **Photometer Scan Products**. In Figure 2.12 we show how the Level 1 products can be accessed from the observational context. Note that within the Level 1 Context there are a total of 12 Products labelled from 0 to 12. These are all Photometer Scan Products. As noted earlier the map of NGC5315 was constructed by scanning the photometer arrays 3 times in each orthoganal direction twice making a total of 12 scan lines in total. Although the numbering system seems anonymous, the actual name of the Building Block can still be revealed by checking the Meta Data `bbTypeName` in the Photometer Scan Product (i.e. click on one of the folders numbered 1-12). The column names give the time, and then the signal for each detector on the arrays (not the first entry PSWR1, actually a resistor, measured

in Volts and the following bolometers measured in Jy and a thermistor (PSWT1) again measured in volts, etc.).

Each Photometer Scan Product contains 5 individual Table Datasets (and a Product containing the processing history) as shown in Figure 2.12 and defined below;

- **Signal Table:** A table containing the Sample Time (in seconds) and a column for the signal from every bolometer including both detector (in Jy/beam) and non-detector (e.g. thermistor, resistor in Volts) channels

- **Mask Table:** A table containing the Sample Time (in seconds) and a column for every bolometer including both detector and non-detector (e.g. thermistor, resistor) channels with a mask value corresponding to which processing flags have been raised. The masks are defined in the **SPIRE Pipeline User Guide** document

- **RA Table:** A table containing the Sample Time (in seconds) and a column for the RA on the sky in degrees for each detector (not including non-detector channels)

- **Dec Table:** A table containing the Sample Time (in seconds) and a column for the Dec on the sky in degrees for each detector (not including non-detector channels)

- **Temperature Table:** A table containing the Sample Time (in seconds) and a column for each Thermistor channel temperature (measured in Kelvin)

These individual Table Datasets correspond to data from a single scan line and can be viewed either as - by *right-clicking* - array tables (by selecting `Open With - Data Set Viewer`) or plotted (by selecting `Open With - Table Plotter`). Although the use of `Table Plotter` is beyond the scope of this document, an example is shown in Figure 2.13 where we have selected to plot the Sample Time against the Signal from the PSW D16 bolometer for this particular scan line.

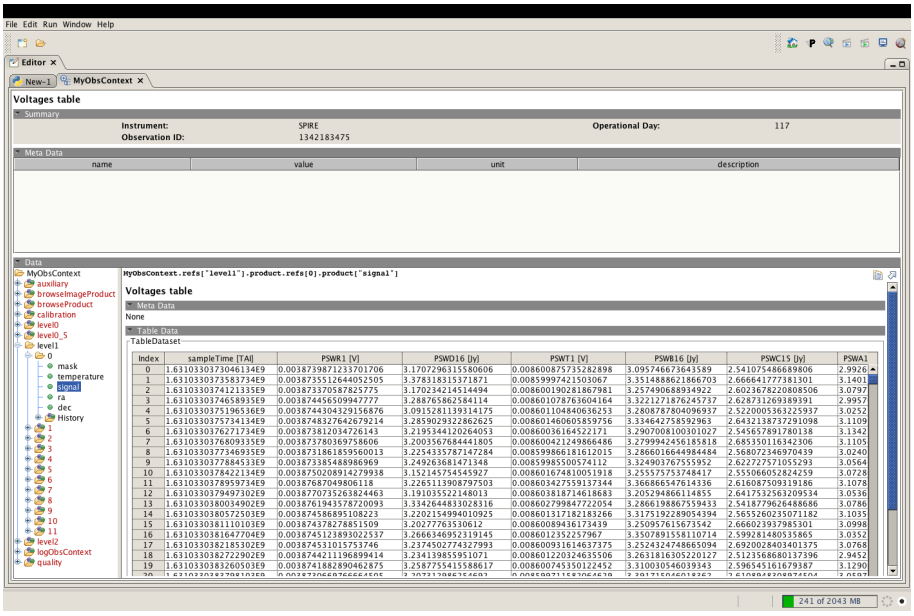


**Figure 2.12. Viewing the Level 1 Photometer Scan Products**

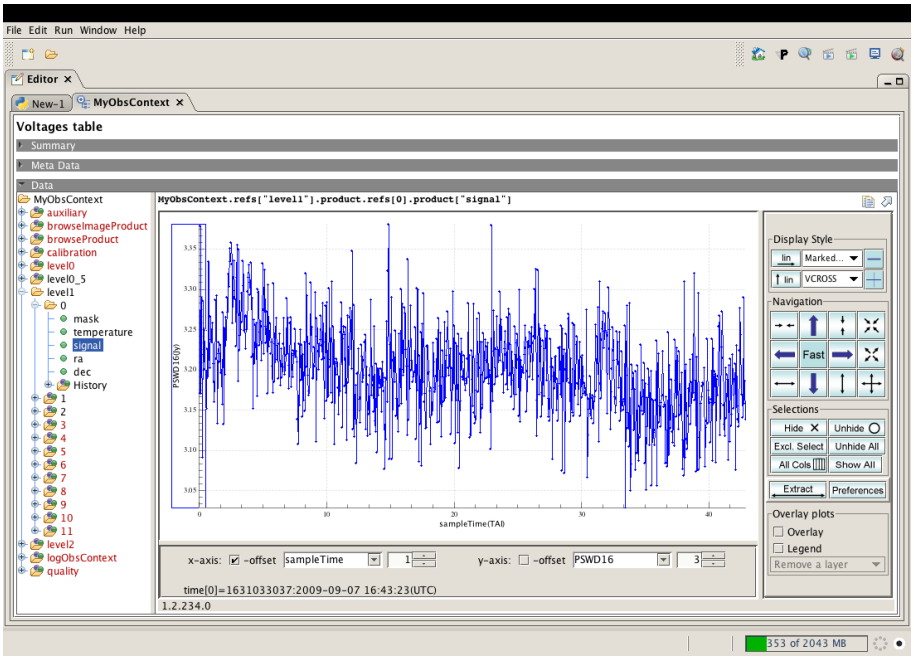


**Figure 2.13. Plotting Level 1 Photometer Scan Product Timeline Data**

Individual Table Data Sets can also be extracted from the Observational Context using the alternative command line script. Using Figure 2.13 as a guide we can see the following;

```
# Extract the Photometer Scan Product for the first Scan Line
ScanLine1=MyObsContext.refs["level1"].product.refs[0].product
# or extract the Photometer Scan Product for the second Scan Line
ScanLine2=MyObsContext.refs["level1"].product.refs[1].product
#
# Get the Signal Table from the first Scan Line
SignalScanLine1=ScanLine1['signal']
# Get the array of values for the Sample Time
TimeScanLine1=SignalScanLine1['sampleTime'].data
# Get the array of values for the PSW D16 Detector
PSWD16ScanLine1=SignalScanLine1['PSWD16'].data
print PSWD16ScanLine1
```

where `ScanLine1`, etc can be any name we choose and the following syntax means from `MyObsContext` we want the Level 1 product Photometer Scan Product for the first scan line (i.e. element [0]). You will also notice that `ScanLine1` now appears in the Variables Panel which can correspondingly be *right-clicked* on to show the various viewing options available for this product. The following lines show the procedure for extracting the second scan line (i.e. array element [1]) and go on to extract, for the first scan line the Signal Table Dataset. Finally the `sampleTime` and detector signal for the `PSWD16` detector are extarcted as normal arrays of numbers. The final list of variables in the HIPE Variable Pane is shown in Figure 2.14.

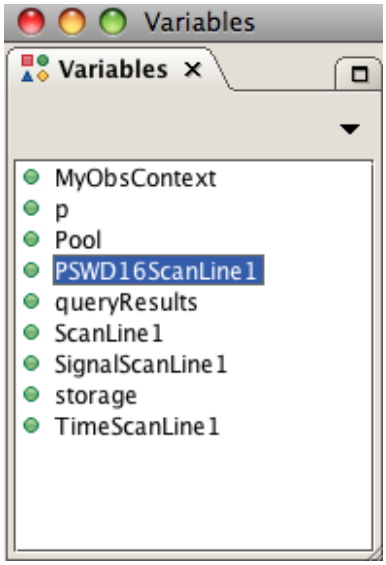


**Figure 2.14. Plotting Level 1 Photometer Scan Product Timeline Data variable list**

## 2.2.4. Looking at the Level 0.5 Timeline Data

These timeline data have been created by processing the raw Level 0 data through the Common Engineering Conversion (Level 0 - Level 0.5) Pipeline. The Level 0.5 data are the uncalibrated, uncorrected timelines measured in **Volts**. The level 0.5 products are also available from the Observation Context. The Level 0.5 context folder can be seen in the Observation Context and can be opened by *clicking* on the + next to the `level0_5` folder. The Level 0.5 context contains a lot more data than the Level 1 context and includes all the data necessary to process the observation and produce science quality data. In Figure 2.15 we show all the Level 0.5 data within the observation context. We see that there are a total of 31 entries in the list informatively labelled from 0 to 30 (Note that PCAL calibration flashes are no longer nade at the beginning of the observation since Operational Day OD302 for Large Map mode and OD341 for Parallel mode). This can be compared to a total of 12 entries that we saw for the Level 1 products. The Level 0.5 context contains all the building blocks used in the observation and in Figure 2.15 we show how this *Large-Map* observation was built up from the individual building blocks. In the figure, the building blocks can be divided into roughly 4 general types, configuration blocks, calibration blocks, science blocks and movement blocks. The type of building block can be revealed by *clicking* on a given number from 0-30 and scrolling down the `Meta data` window pane to the `BBtypeName` entry. The individual blocks are described below in Table 2.2;

**Table 2.2. Description of the Building Blocks in a Large Map Level 0.5 Context**

| BB number | BB Type | BB Hex prefix | Description |
|---|---|---|---|
| 0 | SpireBbObsConfig | 0xAF01 | Initial configuration |
| 1 | SpireBbPhotSerendipity | 0xA104 | Slew to target |
| 2 | SpireBbPOF5Config | 0xA050 | AOT configuration |
| 3 | SpireBbPOF5Init | 0xA051 | Initialize the AOT |
| 4 | SpireBbPcalFlash | 0xA801 | Photometer Calibration Lamp Flash |
| 5 | SpireBbScanLine | 0xA103 | A large map scan line |
| 6 | SpireBbMove | 0xAF00 | Scan Line turnaround movement |
| 7 | SpireBbScanLine | 0xA103 | A large map scan line |
| 8 | SpireBbMove | 0xAF00 | Scan Line turnaround movement |
| .. | SpireBbScanLine | 0xA103 | A large map scan line |
| .. | SpireBbMove | 0xAF00 | Scan Line turnaround movement |

| BB number | BB Type | BB Hex prefix | Description |
|---|---|---|---|
| .. | ... | ... | ..... |
| 27 | SpireBbScanLine | 0xA103 | A large map scan line |
| 28 | SpireBbMove | 0xAF00 | Scan Line turnaround movement |
| 29 | SpireBbPcalFlash | 0xA801 | Photometer Calibration Lamp Flash |
| 29 | SpireBbPOF5End | 0xA052 | End of AOT |

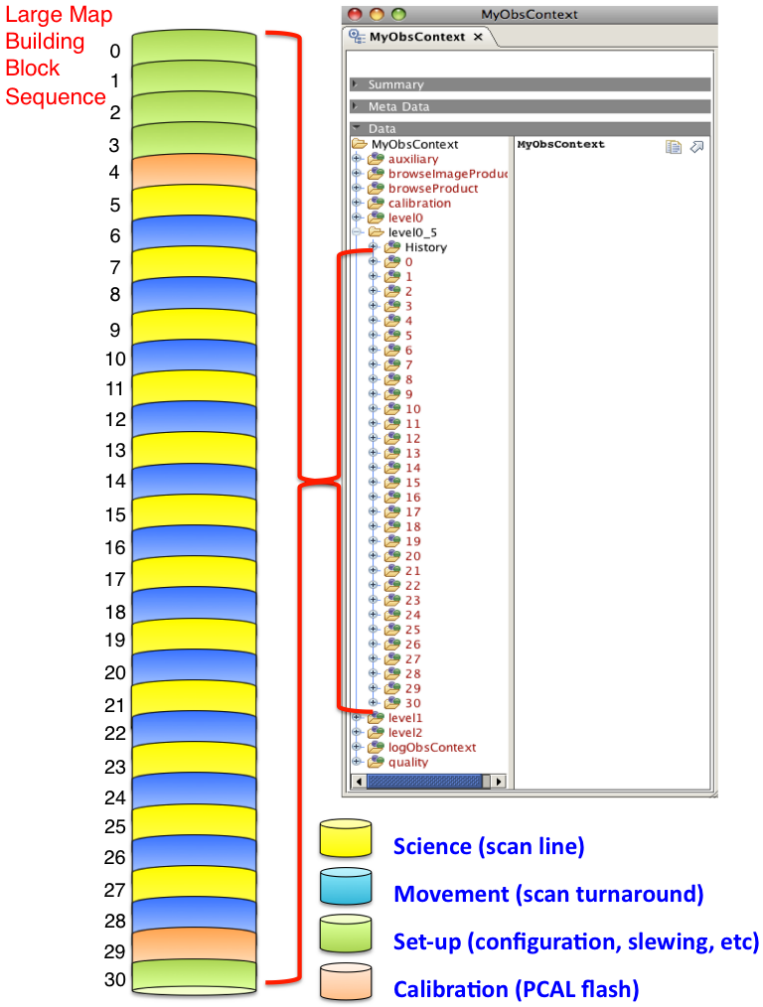


**Figure 2.15. Anatomy of Level 0.5 Building Block structure for a Large Map observation**

Looking at some of the individual entries in the Level 0.5 context, it can be seen that the individual Building Blocks are built up from a variety of different types of Products. *clicking* on the + sign for a given Building Block number reveals what Products a particular Building Block is made from. In Figure 2.16 the first handful of building blocks for our observation are opened to view the contents. The contents are a variety of Products referred to by ancronyms such as CHKT, NHKT, PDT, POT, SCUT, etc, described in order of importance below;

Example building blocks may be;

- **PDT:** The Photometer Detector Timeline contains the Level 0.5 detector data.
- **NHKT:** The Nominal House Keeping Timeline contains the housekeeping data with all the settings for this observation.

- **CHKT:** The Critical House Keeping Timeline contains all the critical parameters of the instrument such as the electronics.

- **SCUT:** The Sub Control Unit Timeline contains monitoring data for the instrument operation for this observation.

- **POT:** The Photometer Offset Timeline contains all the raw DC offsets in ADU that have already been used in the raw data processing to set the dynamic range of the detectors.

Note that Building blocks such as the Slewing (serendipity Building Block), Calibration flash and the scan line turnarounds all contain PDT data. Indeed, the scan line turnaround Building Block data **IS** used for scientific processing. The `CHKT`, `NHKT`, `POT`, `SCUT` Products all contain a `signal` table, containg data arrays and a `Mask` table containing flag information. The Level 0.5 `PDT` Photometer Detector Timeline Products contain 4 Table dataset arrays;

- **Voltage Table:** A table containing the Sample Time (in seconds) and a column for the signal measured in Volts for every bolometer including both detector and non-detector (e.g. thermistor, resistor) channels.

- **Resistance Table:** A table containing the Sample Time (in seconds) and a column for the Resistance measured in Ohms for every bolometer including both detector and non-detector (e.g. thermistor, resistor) channels.

- **Mask Table:** A table containing the Sample Time (in seconds) and a column for every bolometer including both detector and non-detector (e.g. thermistor, resistor) channels with a mask value corresponding to which processing flags have been raised. The masks are defined in the **SPIRE Pipeline User Guide** document

- **Temperature Table:** A table containing the Sample Time (in seconds) and the temperature of the 6 Thermistors (2 per array) in Kelvin.

- **Quality Table:** A table containing any Quality Flags raised for each detector.

In Figure 2.16 the PDT for the first Scan Line Building Block has been selected. *Right-clicking* and selecting `Open-with - Dataset Viewer`, opens the `voltage` table in a new window. Any of the Table Data Sets can also be viewed graphically by selecting `Open-with - Table Plotter` as shown in Figure 2.17. In the plot window the bolometer signal to plot can be selected from the `Y-axis` menu and many bolometers can be overlayed by ticking the `overlay` box (both circled in the plot window).

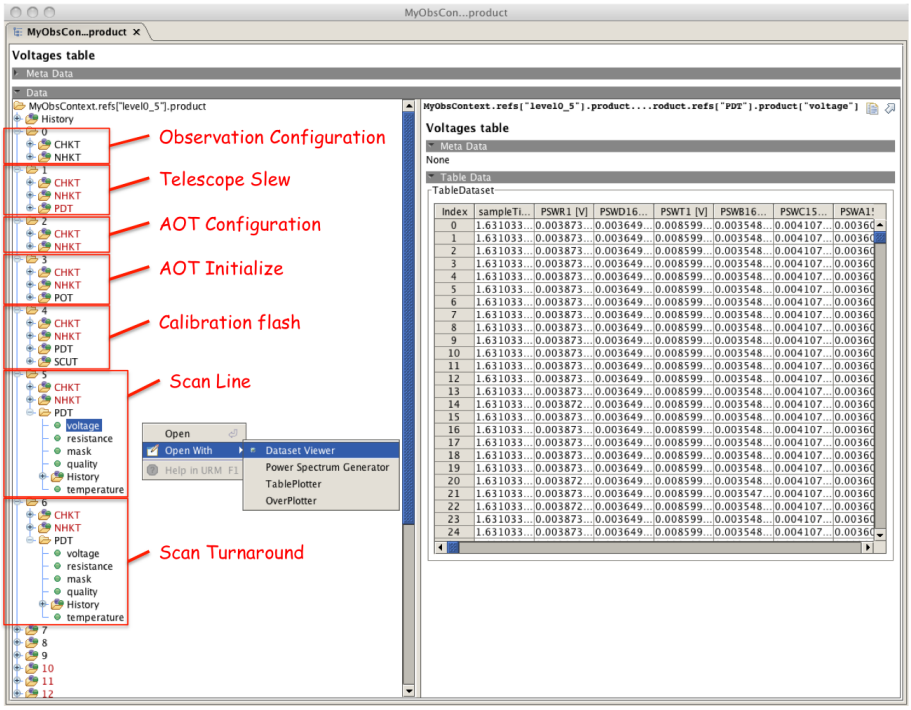


**Figure 2.16. Inside the Level 0.5 Building Block structure for a Large Map observation**

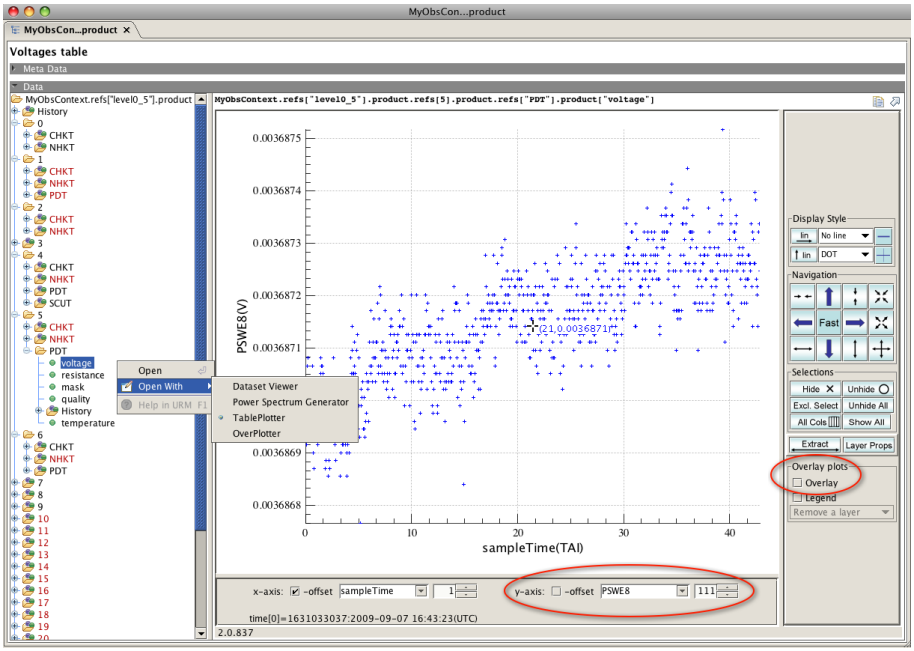


**Figure 2.17. Plotting the Level 0.5 data for a Large Map observation**

## 2.2.5. Looking at the Raw Level 0 Data

The Raw data formatted from the satellite telemetry is also available within the Observation Context. These are the Level 0 Products and will in most circumstances be of no general interest. The Level 0 Context, shown in Figure 2.18, contains 30 entries. Note that there is a significant difference in the Level 0 data structure compared to the Level 0.5 Products. In the Level 0.5 Products, each individual block in the observation has several data types (e.g. Scan line, Housekeeping data, etc - see Table 2.2 ). However, in order to reduce the raw data volume at the Level 0 stage, all the data types are concatonated into a single Level 0 product, referred to as a *Raw SPIRE Timeline* (RST) for each

building block, i.e. A single Level 0 product contains many seperate Table datasets. *Clicking* on a given number within the Level 0 context reveals the Level 0 Product for that particular building block. These products are the *raw* data versions of the Level 0.5 data and contain Table Datasets such as the Critical House Keeping timelines (`CHK`), Nominal House Keeping timelines (`NHK`), Raw Photometer Detector timelines (`PHOTF`), Raw Photometer Offset timelines (`PHOTOFF`) and Sub-Control Unit timelines (`SCUNOMINAL`). The Raw Photometer Detector Timeline ( `PHOTF`) Table Dataset can be viewed by *right-clicking* and selecting `Open-with - Dataset Viewer`, see Figure 2.18), we find quite a different structure to the Level 0.5 PDT datasets. There are 288 columns, one for every SPIRE channel, numbered not in the familiar `PSWE8`, `PSWE9` notation but rather as as `PHOTFARRAY001` -- `PHOTFARRAY288` which corresponds to their Channel Number (from an electrical designation). The signal is still in raw ADU and there are many different *time* columns which correspond to various measures of the data frames, telemetry packets and packet sequence counts, etc. The only flags are contained in the `PHOTFADCFLAGS` column which is set in the case of a problem with ADC process in telemetry. A full description of the data structure can be found in the Products Definition Document (HERSCHEL-HSC-DOC-0959) or the SPIRE Pipeline Description Document (SPIRE-RAL-DOC-002437).

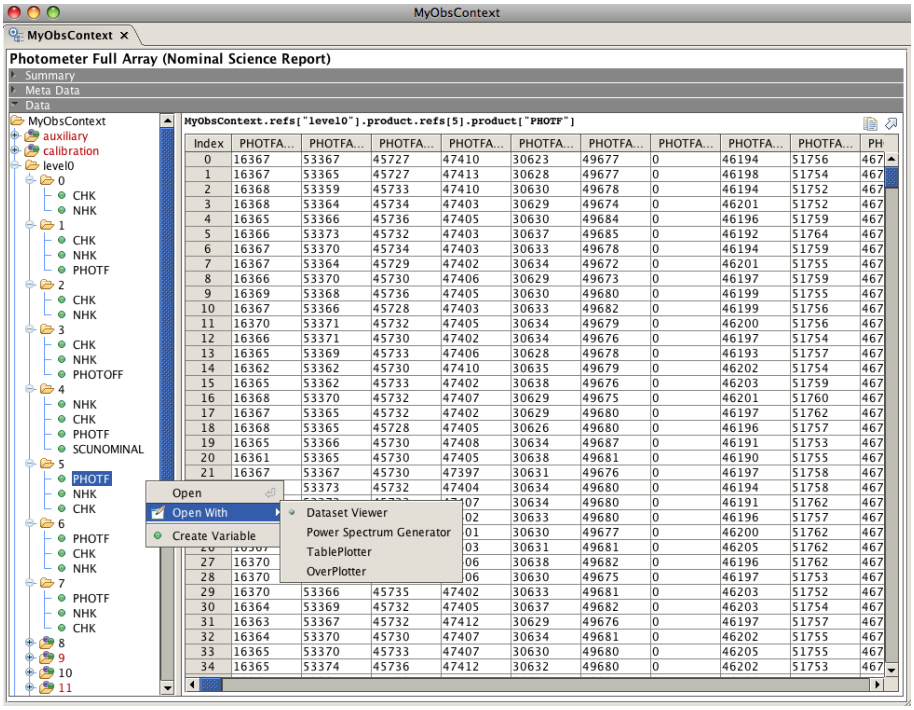


**Figure 2.18. The Level 0 Raw Data within the Observation Context**

# 2.3. SPIRE Small Map Mode Data Structure

## 2.3.1. A first look at your image maps (The Level 2 Data Product)

All the information for a given SPIRE observation is contained with the Observation Context (described in Section 2.1). In this section we shall see how to examine the data for a SPIRE **Small Map** observation, however this description applies equally to SPIRE **Large Map and Parallel Mode** observations. The Small Map mode operation is basically identical to the nominal Large Map Mode except that instead of a nominal 2x2 scan leg covergae of optional scan leg length, the Small Map

mode consists of a pair of orthoganl scans (i.e. a 1x1 scan, See Figure 2.19). For a given observation the area covered by both scan legs defines a central square of side 5 arcmins although the length of the two orthogonal scan paths are somewhat longer than this. In practice, due to the position of the arrays on the sky at the time of a given observation, the guaranteed area for scientific use is a circle of diameter 5 arcmins.

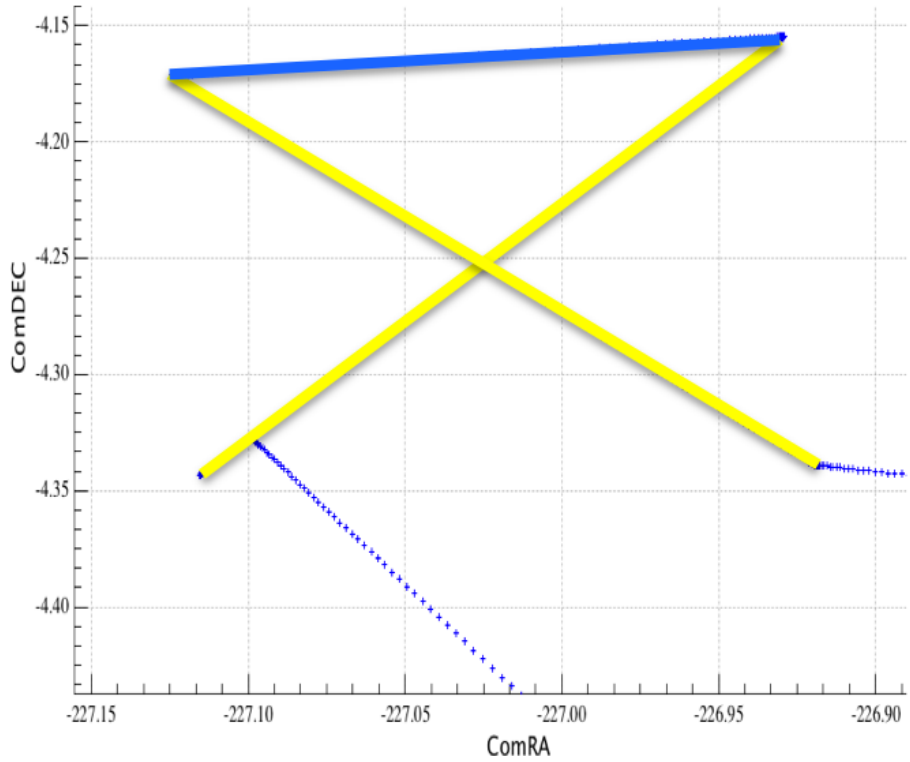


**Figure 2.19. The format of a Small Map observation.**

The observation we shall be looking at is a Small Map observation of a standard SPIRE calibration star Gamma Draconis taking during the Herschel-SPIRE routine calibration phase. Gamma Draconis is at RA=17h56m36.37s, dec=51d29'20.00" and was covered by scanning the photometer arrays once each in orthoganal directions. The entire process was then repeated a total of 4 times (i.e. this observation has 4 repetitions) giving in total 6 scans in each orthoganal direction making 12 scan lines in total.

It is assumed that the observation has already been downloaded into a Pool within your Local Store on your computer as described in section Section 2.1. The Observation Pool can be loaded into HIPE using the following 4 lines of Jython Code (where the **Pool** is whatever name you called your Pool for this observation in your Local Store on disk;

```
Pool  = -'OD358-SmallScanMapGammDra0x5000489F'        # Select the pool name
storage=ProductStorage(Pool)                         # Register the pool
queryResults = storage.select(Query("type=='OBS'"))  # Query the pool
MyObsContext = queryResults[0].product               # Extract the  Context
```

For this particular observation, we chose to call our Pool **OD358-SmallScanMapGammDra0x5000489F** where **OD358** means the observation was made on Operational Day 358, **SmallScanMap** was the AOT mode, **GammDra** was the target name and **0x5000489F** is the unique Observation ID in hexadecimal. Running the above script, reads the Observation Context into memory into the variable **MyObsContext** which appears in the **Variables** pane of HIPE (See Figure 2.20). Right Clicking (or CTRL-click for Apple Users) on the **MyObsContext** variable brings up another menu. Selecting `Open With` -- `Observation Viewer` will open the Observational Context for this observation. The structure of the Observation Context was explained in Section 2.1 and he we shall look at the data inside the Observational Context. We start with the final Product of

the SPIRE Large Map pipeline - the image maps. The maps are Level 2 Products and can therefore be found within the Level 2 Context. The maps can be simply acccesed by clicking on the **level2** folder as shown in Figure 2.21, which reveals a SPIRE Photometer Map Product (or more technically `SimpleImage` Products) for each of the three SPIRE arrays (PSW, PMW, PLW). Each Photometer Map Product contains 3 Table Datasets corresponding to the image, error and coverage maps for each array and these are revelaed by *clicking* on the + sign next to the array folder.

The image map can be viewed by clicking on the appropriate array folder (PSW, PMW, PLW) or alternatively the image map can be displayed in a new window by right clicking on the appropriate array folder and selecting `Open With - Standard Image Viewer` from the drop down menu as shown in Figure 2.22. This action opens the image in the `Image Viewer` where the image can be panned, magnified etc. Colours, cut-levels, annotation options can be accessed by *right-clicking* anywhere on the image. The image, error and coverage maps can also be displayed individually by *clicking* on them or by *right-clicking* on the appropriate dataset and selecting `Open With - Image Viewer for ArrayDatasets` from the drop down menu. Finally, *right-clicking* on a given image dataset and selecting `Open With - Array Dataset Viewer` from the drop down menu shows the image (or error or coverage) in table form (Jy/beam for every pixel in the image) as shown in Figure 2.23.

If you want to extract the `SimpleImage` for the PSW, PMW or PLW array as a data cube containing the image, error and coverage maps to work with, rather than view it with the `Image Viewer`, on the command line type the rather exhaustive:

```
MyMapProduct=MyObsContext.refs["level2"].product.refs["PSW"].product
# Then to view each of the map datasets
Display(MyMapProduct.image)
Display(MyMapProduct.error)
Display(MyMapProduct.coverage)
```

where `MyMapProduct` can be any name we choose and the following syntax means from `MyObsContext` we want the Level 2 product PSW array Photometer Map Product. You will also notice that `MyMapProduct` now appears in the Variables Panel which can correspondingly be *right-clicked* on to show the various viewing options available for this product. The next 3 lines in the above script allow us to display the signal, error and coverage maps respectively. The result is shown in Figure 2.24.

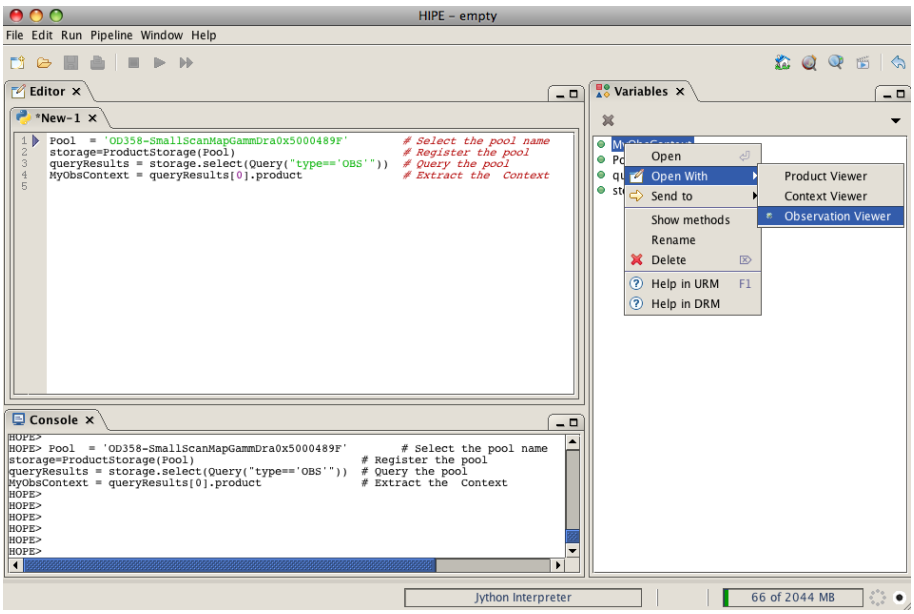


**Figure 2.20. Loading and viewing the Observation Context for the Small Map Observation.**

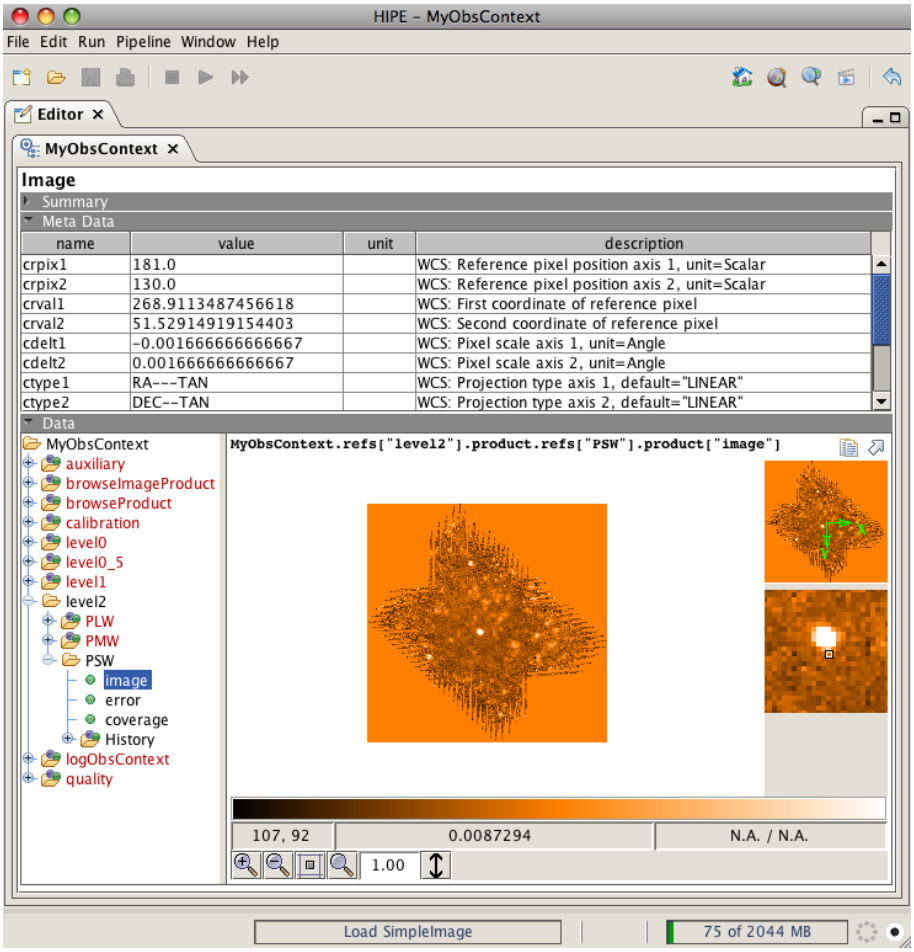


**Figure 2.21. Accessing the final Level 2 Product maps**

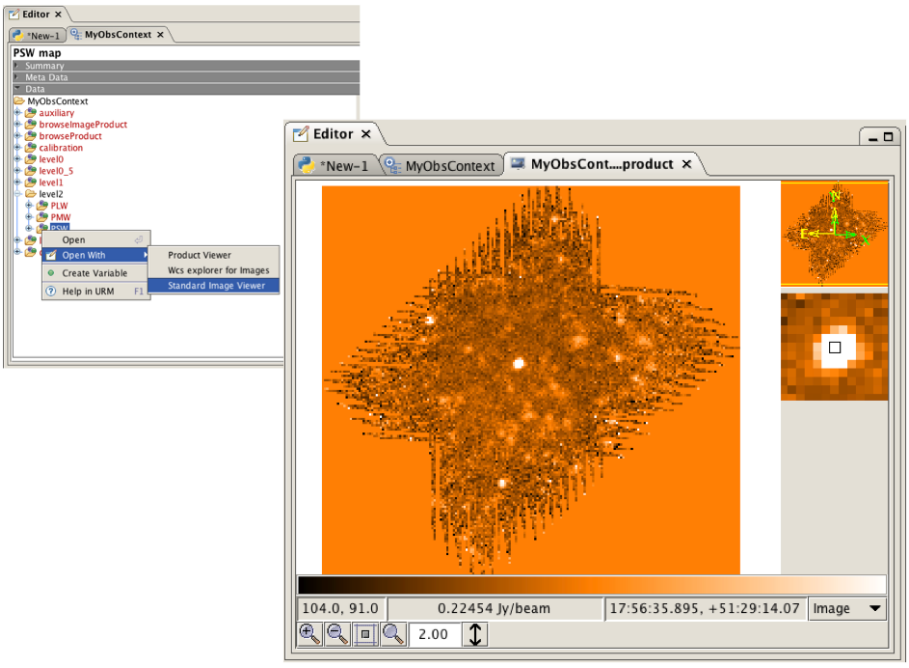


**Figure 2.22. Viewing the Level 2 Image Maps**

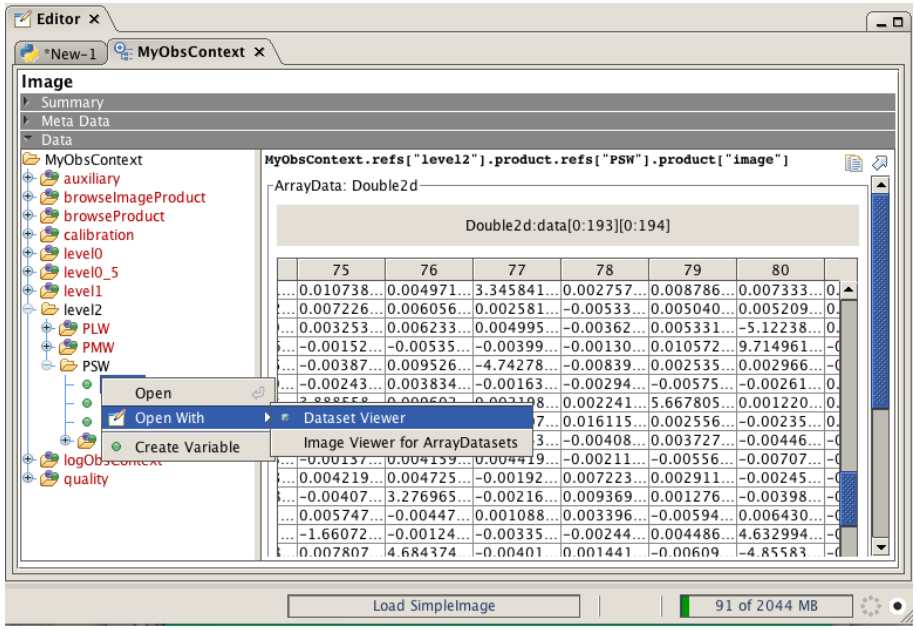


**Figure 2.23. Viewing the Level 2 Image Array Datasets**

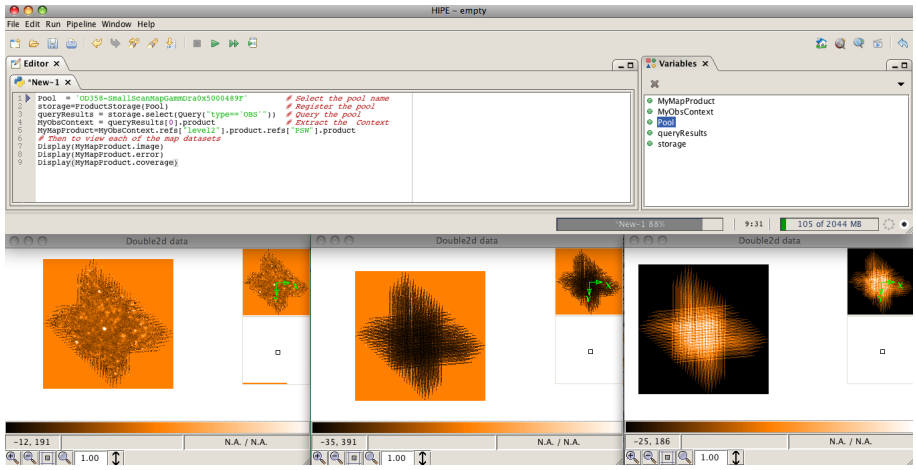


**Figure 2.24. Viewing the image cube for signal, error and coverage maps**

## 2.3.2. Saving a map as a FITS file and reading it in again

It is possible that me may also want to look at our image maps in external applications such as DS9 for example and HIPE provides the tools for exporting our maps as conventional fits files. Writing data out as FITS files and reading FITS files in is identical to the method described in the Large Map Section 2.2.2.

**Note**

The Photometer Map Products (data cubes for each array containing the image, error and coverage arrays) actually exist as fits files within the **Pool** for this observation in the Local Store. These can be found in the Pool for this example in the folder `/localstore/OD358-SmallScanMapGammDra0x5000489F/herschel.ia.dataset.image.SimpleImage` (where the poolname is "OD358-SmallScanMapGammDra0x5000489F"). The Photometer Map Products have the form `hspireplw.........pmp.fits`

## 2.3.3. Looking at the Level 1 Timeline Data

The image maps have been created from the individual timelines of detectors as they were scanned accross the target. These timelines are the Level 1 products from the Photometer Small Map Pipeline and are also available from the Observation Context. The Level 1 Small Map products are referred to as **Photometer Scan Products** and are exactly the same format as the Level 1 Large Map products described in Section 2.2.3. In Figure 2.25 we show how the Level 1 products can be accessed from the observational context. Note that within the Level 1 Context there are a total of 8 Products labelled from 0 to 8. These are all Photometer Scan Products. As noted earlier this map of Gamma Draconis was constructed by scanning the photometer arrays 4 times in each orthoganal direction once making a total of 8 scan lines in total. Although the numbering system seems anonymous, the actual name of the Building Block can still be revealed by checking the Meta Data `bbTypeName` in the Photometer Scan Product i.e. pull down the meta data information (circled in red in Figure 2.25) and then click on one of the folders numbered 1-8 in the context. The Figure shows the `signal` table for one scan line. The column names give the time, and then the signal for each detector on the arrays (not the first entry PSWR1, actually a resistor, measured in Volts and the following bolometers measured in Jy and a thermistor (PSWT1) again measured in volts, etc.).

Since the Small Map and Large Map modes are essentially the same, the further detailed structure of the Small Map Level 1 Product is described in the Large Map Section 2.2.3.

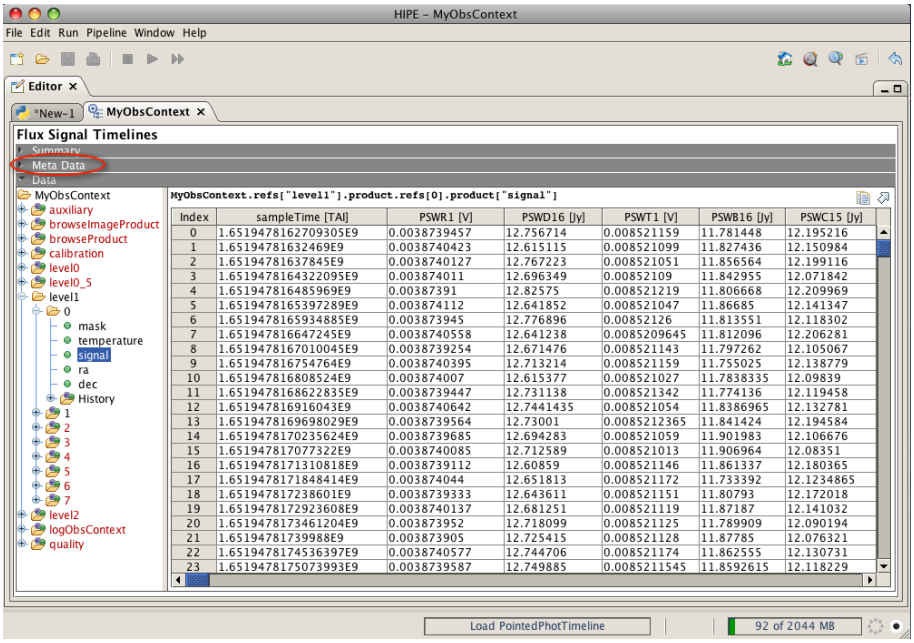


**Figure 2.25. Viewing the Level 1 Photometer Scan Products for the Small Map mode**

## 2.3.4. Looking at the Level 0.5 Timeline Data

The Level 2 maps and the Level 1 timeline products represent the output from the Small Map pipeline. These timeline data were created from the lower Level 0.5 data products (which were correspondingly created from processing the raw Level 0 data through the Common Engineering Conversion (Level 0 - Level 0.5) Pipeline). The Level 0.5 data are the voltage calibrated, timelines measured in **Volts** uncorrected for detector effects. These level 0.5 products are also available from the Observation Context. The Level 0.5 context folder can be seen in the Observation Context and can be opened by *clicking* on the + next to the `level0_5` folder. The Level 0.5 context contains a lot more data than the Level 1 context and includes all the data necessary to process the observation and produce science quality data. In Figure 2.26 we show all the Level 0.5 data within the observation context. We see that there are a total of 20 entries in the list informatively labelled from 0 to 20. This can be compared to a total of 8 entries that we saw for the Level 1 products. The Level 0.5 context contains all the building

blocks used in the observation and in Figure 2.26 we show how this *Small-Map* observation was built up from the individual building blocks. In the figure, the building blocks can be divided into roughly 4 general types, configuration blocks, calibration blocks, science blocks and movement blocks. The type of building block can be revealed by *clicking* on a given number from 1-20 and scrolling down the `Meta data` window pane to the `BBtypeName` entry. The individual blocks are are the same as for the **Large Map** mode and described previously in Table 2.2.

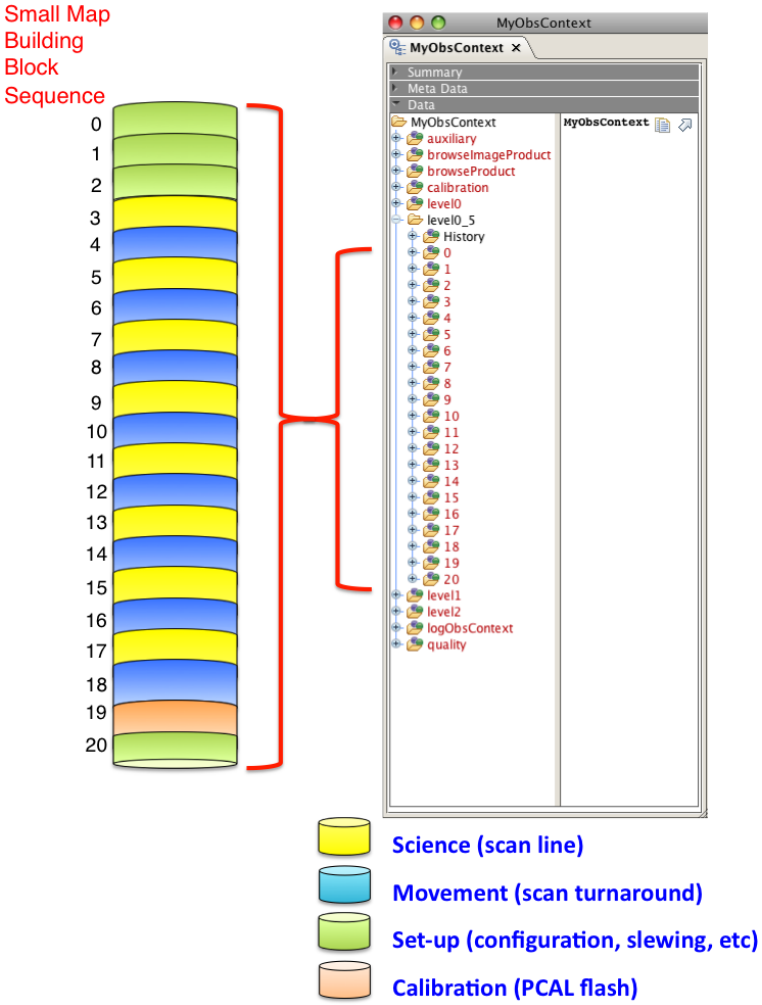


**Figure 2.26. Anatomy of Level 0.5 Building Block structure for a Small Map observation**

Since the Small Map and Large Map modes are essentially the same, the further detailed structure of the Small Map Level 0.5 Products is described in the Large Map Section 2.2.4.

## 2.3.5. Looking at the Raw Level 0 Data

The Raw data formatted from the satellite telemetry is also available within the Observation Context. These are the Level 0 products and will in most circumstances be of no general interest. Since the Small Map and Large Map modes are essentially the same, the further detailed structure of the Small Map Level 0 Products is described in the Large Map Section 2.2.5.

# 2.4. SPIRE Point Source Mode Data Structure

## 2.4.1. The Point Source Observation Mode

All the information for a given SPIRE observation is contained with the Observation Context (described in Section 2.1). In this section we shall see how to examine the data for a SPIRE **Point Source** observation. A point source observation carries out a staring observation of a point source. In order to recover the source successfully a 7-point hexagonal jiggle pattern is made around the source position. Sky backgrounds are removed by chopping using the Beam Steering Mirror (BSM) over a distance of plus/minus 63 arc sec and any emission due to the telescope structure is removed by nodding the entire telescope and repeating the chop=jiggle cycle.

The observation we shall be looking at is a Point Source observation of the Planetary Nebulae NGC5315 taking during the Herschel-SPIRE PV phase. NGC5315 is at RA=13h53m57.00s, dec=-66d30'50.70" and was covered by making 2 repetitions of the Point Source Mode which involves makes a pair of chopped and nod cycles at each of the 7 jiggle positions in the pattern.

It is assumed that the observation has already been downloaded into a Pool within your Local Store on your computer as described in section Section 2.1. The Observation Pool can be loaded into HIPE using the following 4 lines of Jython Code (where the **Pool** is whatever name you called your Pool for this observation in your Local Store on disk;

```
Pool  = -'OD117-7ptNGC5315-0x50001832'                # Select the pool name
storage=ProductStorage(Pool)                         # Register the pool
queryResults = storage.select(Query("type=='OBS'"))  # Query the pool
MyObsContext = queryResults[0].product               # Extract the  Context
```

For this particular observation, we chose to call our Pool **OD117-7ptNGC5315-0x50001832** where **OD117** means the observation was made on Operational Day 117, **7pt** was the AOT mode, **NGC5315** was the target name and **0x50001832** is the unique Observation ID in hexadecimal. Running the above script, reads the Observation Context into memory into the variable **MyObsContext** which appears in the **Variables** pane of HIPE (See Figure 2.27). Right Clicking (or CTRL-click for Apple Users) on the **MyObsContext** variable brings up another menu. Selecting `Open With` -- `Observation Viewer` will open the Observational Context for this observation. The structure of the Observation Context was explained in Section 2.1 and he we shall look at the data inside the Observational Context. We start with the final Product of the SPIRE Point Source pipeline - The Jiggled Photometer Product (JPP). The JPP is a Level 2 Product and can therefore be found within the Level 2 Context. The JPP can be simply acccesed by clicking on the **level2** folder as shown in Figure 2.28, which reveals a SPIRE Jiggled Photometer Product. *Right-clicking* on the JPP and selecting `Open With` - `Array Dataset Viewer` from the drop down menu shows the data in table form as shown in Figure 2.28. The JPP contains a Table Dataset with a row for each array with the following information;

- **Array Name:** A column listing each array PSW, PMW, PLW.
- **RA:** A column listing the final fitted Right Ascension for each array to the detected source within the 7-point Jiggle pattern for the target detector in decimal degrees
- **RA Error:** A column listing the errors on the Right Ascension for each array
- **Dec:** A column listing the final fitted Declination for each array to the detected source within the 7-point Jiggle pattern for the target detector in decimal degrees
- **Dec Error:** A column listing the errors on the Declination for each array
- **Signal:** A column listing the Gaussian fitted signal for the target detector for each array to the detected source within the 7-point Jiggle pattern in Jy (in beam flux)

- **Error:** A column listing the error on the fitted signalfor each array

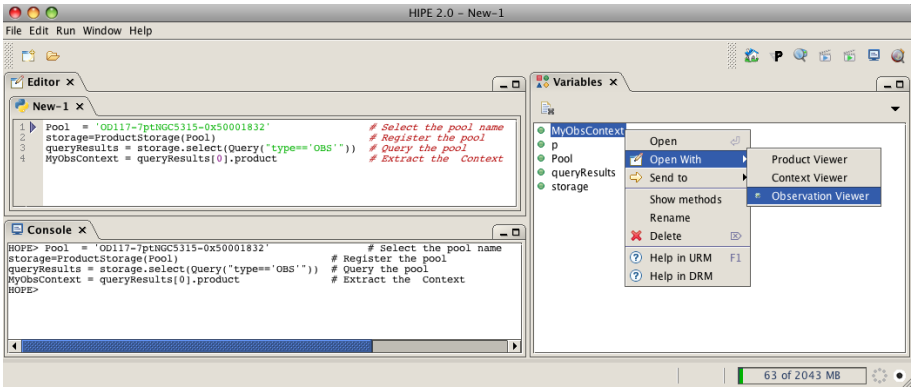


**Figure 2.27. Loading and viewing the Observation Context for the Photometer Point Source Observation.**

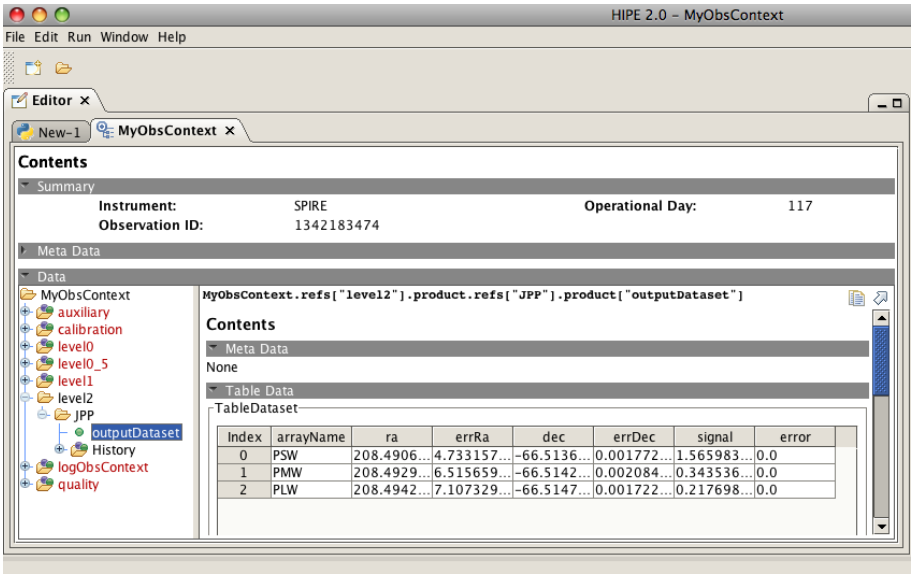


**Figure 2.28. Accessing the final Level 2 Product Jiggled Photometer Product**

## 2.4.2. Reading the JPP into memory and saving it as a FITS file and reading it in again

It is possible that me may also want to export our data and HIPE provides the tools for exporting data products as conventional fits files. The Level 2 JPP can be read into memory with the following admittadly long-winded command from the command line;

```
# read entire Product
myJPP=MyObsContext.refs["level2"].product.refs["JPP"].product
#
# read the RA data array
myRa=myJPP["outputDataset"]["ra"].data
print myRa
# read the RA for PSW array
myRaPSW=myJPP["outputDataset"]["ra"].data[0]
print myRaPSW
```

This creates a new entry `myJPP` in the Variables Pane of HIPE which can correspondingly be *right-clicked* on to show the various viewing options available for this product. The next 4 lines in the above script allow us to read in and print out the data for the Right Ascension for all arrays and for just the

PSW array (creating entries for `myRa` and `myRaPSW` in the variable pane). The JPP Level 2 Product can be saved as a FITS file by the following command line entry;

```
FitsArchive().save('mypath/myJPP.fits', myJPP)
```

where `mypath` is the desired path. Alternatively the product can be sent to a FITS file by *right-clicking* on it in the variable list and selecting `Send To - FITS file` from the drop down menu. This will open the FITS writer panel as shown in Figure 2.29 where we can type in our desired filename and path. Click on `Accept` at the bottom of the panel to save the FITS file.

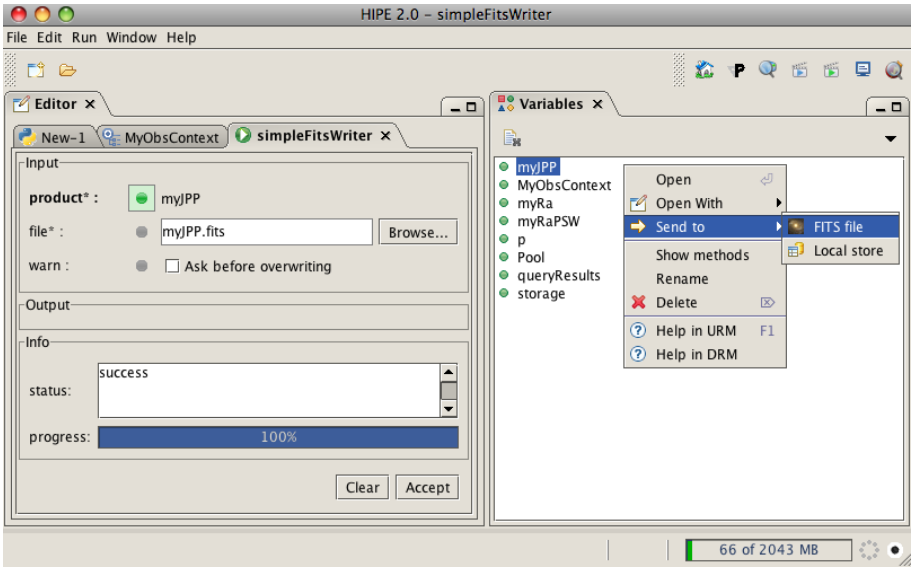


**Figure 2.29. Exporting the JPP as a FITS file**

Reading a FITS file into the HIPE session can be accomplished by either selecting `Open File` from the `File` menu in the top right hand corner of the HIPE window. Alternatively, from the command line;

```
myJPP=simpleFitsReader('mypath/myJPP.fits')
```

These FITS files are imported as an `JPP Product dataset` and can be manipulated in the same manner as described earlier throughout this section.

**Note**

The JPP actually exist as a fits file within the **Pool** for this observation in the Local Store. These can be found in the Pool for this example in the folder `/localstore/OD117-7ptNGC5315-0x50001832/herschel.spire.ia.dataset.JiggPhotProduct` (where the pool-name is "OD117-7ptNGC5315-0x50001832"). The JPP will have the `hspirephotometer.......jpp.fits`

## 2.4.3. Looking at the Level 1 Data for Point Source Observations

The final Level 2 Jiggle Photometer Product has been created from a Gaussian fit to the 7-point jiggle pattern of a target bolometer. The information on the individual jiggle positions for all bolometers is contained within the Level 1 Product and are also available from the Observation Context. The Level 1 Point Source mode product is referred to as the **Averaged Pointed Photometer Product (APPP)**. In Figure 2.30 we show how the Level 1 product can be accessed from the observational context. The

APPP holds information for each of the 7 jiggle positions for all bolometers after the signal has been demodulated (chopped) and de-nodded.

Each Averaged Pointed Photometer Product contains 7 individual Table Datasets (and a Product containing the processing history) as shown in Figure 2.30 and defined below;

- **Signal Table:** A table containing a column for the Jiggle ID (1-7 position) and a column for the signal from every detector channel (in Jy/beam)
- **Error Table:** A table containing a column for the signal error from every detector channel (in Jy/beam)
- **Dec Table:** A table containing a column for the declination on the sky in degrees for every detector channel
- **Dec Error Table:** A table containing a column for the errors in declination on the sky in degrees for every detector channel
- **RA Table:** A table containing a column for the right ascension on the sky in degrees for every detector channel
- **RA Error Table:** A table containing a column for the errors in right ascension on the sky in degrees for every detector channel
- **Mask Table:** A table containing the mask value for every detector channel corresponding to which processing flags have been raised. The masks are defined in the **SPIRE Pipeline User Guide** document

The APPP be viewed either - by *right-clicking* - array tables (by selecting `Open With - Data Set Viewer`) or plotted (by selecting `Open With - Table Plotter`). Although the use of `Table Plotter` is beyond the scope of this document, an example is shown in Figure 2.31 where we have selected to plot the Jiggle ID against the Signal from the PSW E10 bolometer for the APPP.

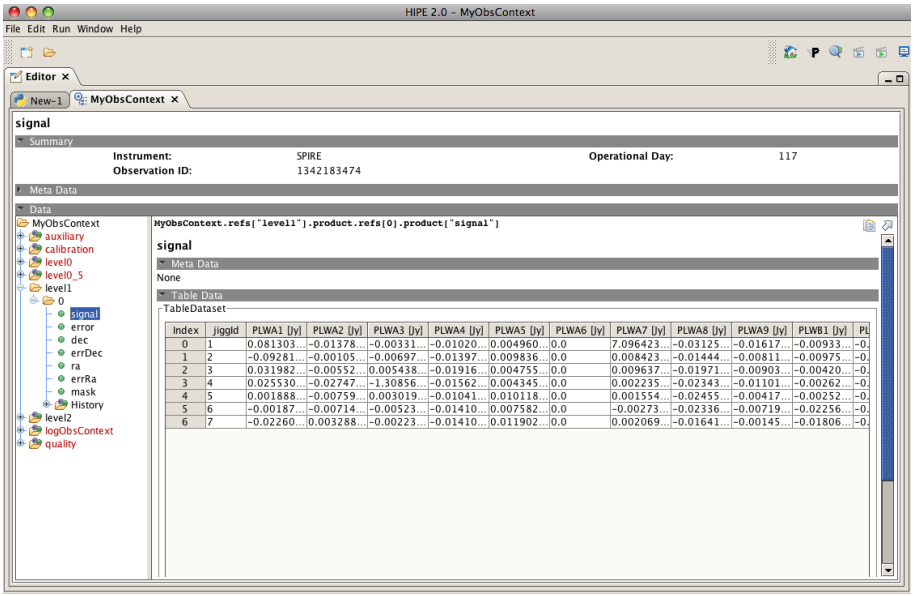


**Figure 2.30. Viewing the Level 1 Averaged Pointed Photometer Product**

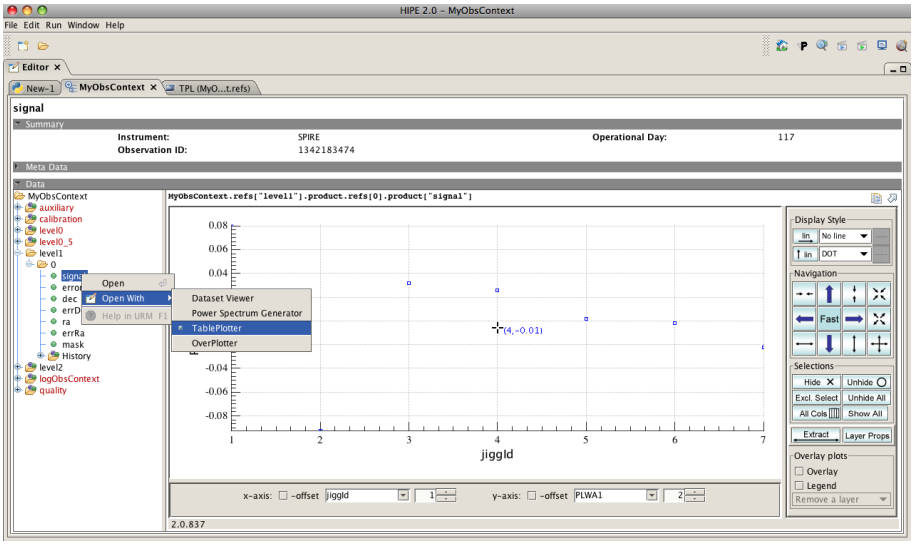


**Figure 2.31. Plotting Level 1 APPP Data Product**

## 2.4.4. Looking at the Level 0.5 Timeline Data for Point Source Observations

The Level 2 JPP and the Level 1 APPP products represent the output from the Point Source pipeline. These data products were created from the lower Level 0.5 data products (which were correspondingly created from processing the raw Level 0 data through the Common Engineering Conversion (Level 0 - Level 0.5) Pipeline). The Level 0.5 data are the voltage calibrated, timelines measured in **Volts** uncorrected for detector effects. These level 0.5 products are also available from the Observation Context. The Level 0.5 context folder can be seen in the Observation Context and can be opened by *clicking* on the + next to the `level0_5` folder. The Level 0.5 context contains a lot more data than the Level 1 context and includes all the data necessary to process the observation and produce science quality data. In Figure 2.32 we show all the Level 0.5 data within the observation context (Note that since Operational Day OD302 PCAL calibration flashes are no longer nade at the beginning of the observation). We see that there are a total of 23 entries in the list informatively labelled from 0 to 22. This can be compared to the single final product that we saw for the Level 1 data. The Level 0.5 context contains all the building blocks used in the observation and in Figure 2.32 we show how this *Point Source* observation was built up from the individual building blocks. In the figure, the building blocks can be divided into roughly 4 general types, configuration blocks, calibration blocks, science blocks and movement blocks. The type of building block can be revealed by *clicking* on a given number from 0-22 and scrolling down the `Meta data` window pane to the `BBtypeName` entry. The individual blocks are described below in Table 2.3. This observation involves two repetitions of the Point Source Mode. A single science building block consists an operation at a given Nod position (denoted A or B) and moving to the first jiggle position on the 7-point pattern, chopping 8 times on/off source, moving to the second position, .... until all 7 positions have been visited (plus one more at the centre). This operation is then repeated at the next nod positin (position B), repeated at B and then once more at nod position A. One repetiotion thus corresponds to a single ABBA nod cycle, therefore this observation will consist of 2 ABBA cycles.

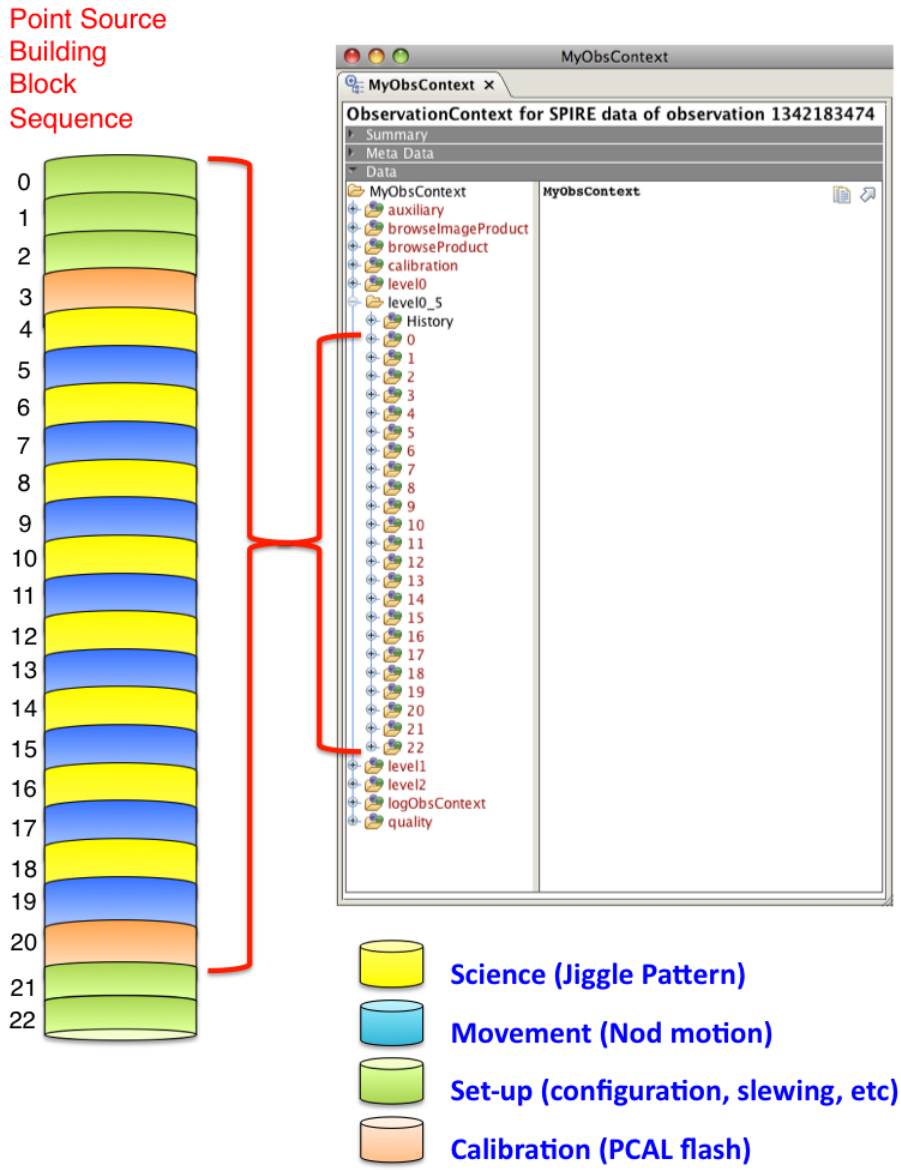


**Figure 2.32. Anatomy of Level 0.5 Building Block structure for a Point Source observation**

**Table 2.3. Description of the Building Blocks in a Point Source Mode Level 0.5 Context**

| BB number | BB Type | BB Hex prefix | Description |
|---|---|---|---|
| 0 | SpireBb_StartObsAll | 0xB6C8 | Begin Observation |
| 1 | SpireBbPOF2Config | 0xA020 | Initial configuration of the Point Source AOT |
| 2 | SpireBbPOF2Init | 0xA021 | Initialize the Point Source AOT |
| 3 | SpireBbPcalFlash | 0xA801 | Photometer Calibration Lamp Flash |
| 4 | SpireBbJiggle | 0xA321 | Carry out chopped motion around 7-point jiggle pattern at first nod position |
| 5 | SpireBbMove | 0xAF00 | Movment of Nod position (position A to B) |
| 6 | SpireBbJiggle | 0xA321 | Carry out chopped motion around 7-point jiggle pattern at second nod position |
| 7 | SpireBbMove | 0xAF00 | Movment of Nod position (dwell at position B) |
| 8 | SpireBbJiggle | 0xA321 | |

| BB number | BB Type | BB Hex prefix | Description |
|---|---|---|---|
| | | | Carry out chopped motion around 7-point jiggle pattern at second nod position |
| 9 | SpireBbMove | 0xAF00 | Movment of Nod position (position B to A) |
| 10 | SpireBbJiggle | 0xA321 | Carry out chopped motion around 7-point jiggle pattern at first nod position |
| 11 | SpireBbMove | 0xAF00 | Movment of Nod position (dwell at position A) |
| 12 -- 19 | ... | ... | Repeat entries 4-11 |
| 20 | SpireBbPcalFlash | 0xA801 | Photometer Calibration Lamp Flash |
| 21 | SpireBbPOF2End | 0xA022 | End of AOT |
| 22 | SpireBb_EndObsAll | 0xB6C7 | End Observation |

Looking at some of the individual entries in the Level 0.5 context, it can be seen that the individual Building Blocks are built up from a variety of different types of Products. *clicking* on the + sign for a given Building Block number reveals what Products a particular Building Block is made from. In Figure 2.33 the first handful of building blocks for our observation are opened to view the contents. The contents are a variety of Products referred to by ancronyms such as `CHKT`, `NHKT`, `PDT`, `BSMT`, `POT`, `SCUT`, etc, described in order of importance below;

Example building blocks may be;

- **PDT:** The Photometer Detector Timeline contains the Level 0.5 detector data.
- **BSMT:** The Beam Steering Mechanism Timeline contains the information of the BSM (chop and jiggle positions as a function of time).
- **NHKT:** The Nominal House Keeping Timeline contains the housekeeping data with all the settings for this observation.
- **CHKT:** The Critical House Keeping Timeline contains all the critical parameters of the instrument such as the electronics.
- **SCUT:** The Sub Control Unit Timeline contains monitoring data for the instrument operation for this observation.
- **POT:** The Photometer Offset Timeline contains all the raw DC offsets in ADU that have already been used in the raw data processing to set the dynamic range of the detectors.

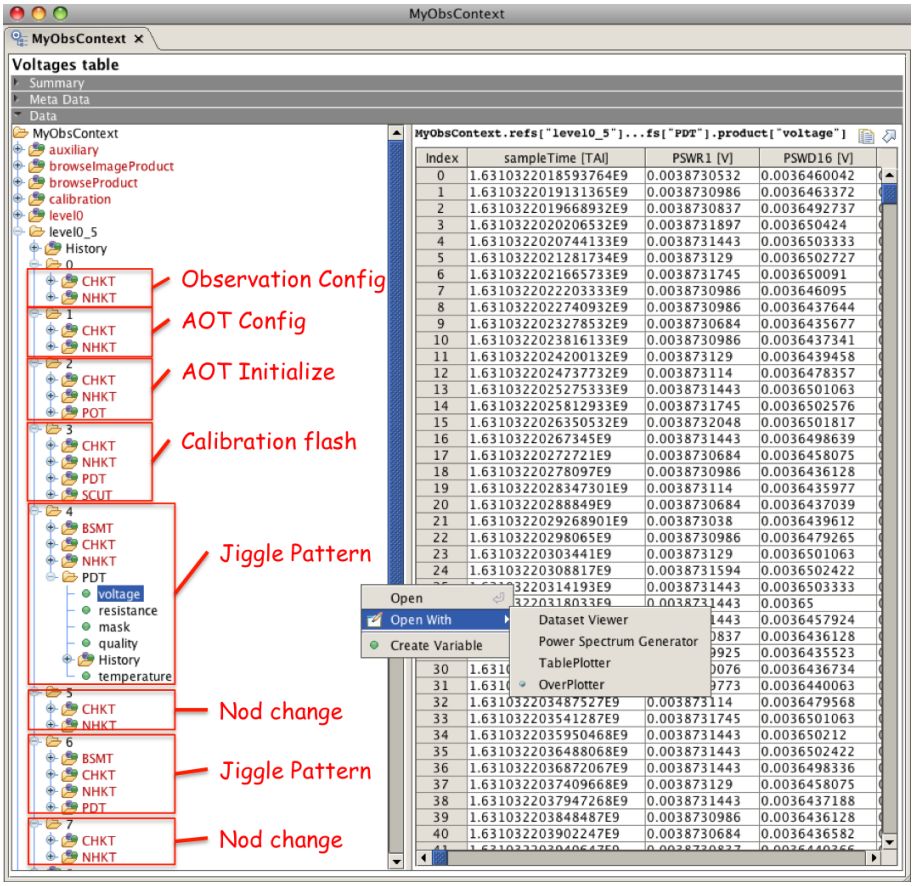


**Figure 2.33. Inside the Level 0.5 Building Block structure for a Point Source observation**

The CHKT, NHKT, BSMT, POT, SCUT Products all contain a `signal` table, containg data arrays and a `Mask` table containing flag information. The Level 0.5 PDT Photometer Detector Timeline Products contain 5 Table dataset arrays;

- **Voltage Table:** A table containing the Sample Time (in seconds) and a column for the signal measured in Volts for every bolometer including both detector and non-detector (e.g. thermistor, resistor) channels.

- **Resistance Table:** A table containing the Sample Time (in seconds) and a column for the Resistance measured in Ohms for every bolometer including both detector and non-detector (e.g. thermistor, resistor) channels.

- **Mask Table:** A table containing the Sample Time (in seconds) and a column for every bolometer including both detector and non-detector (e.g. thermistor, resistor) channels with a mask value corresponding to which processing flags have been raised. The masks are defined in the **SPIRE Pipeline User Guide** document

- **Quality Table:** A table containing any Quality Flags raised for each detector.

- **Temperature Table:** A table containing the Sample Time (in seconds) and the temperature of the 6 Thermistors (2 per array) in Kelvin.

In Figure 2.33 the PDT for the first Jiggle Building Block has been selected. *Right-clicking* and selecting `Open-with - Dataset Viewer`, opens the `voltage` table in a new window. Any of the Table Data Sets can also be viewed graphically by selecting `Open-with - Table Plotter` as shown in Figure 2.34. In the plot window the bolometer signal to plot can be selected from the `Y-axis` menu (circled in the plot window) and in this example the signal versus sample time for bolometer PSW E6 has been selected. In the figure, we also plot a marked line selected from the

`Display Style` box (also circled in the plot window). In Figure 2.34 the on and off chop positions and the circuit around the 7 jggle positions can be clearly seen.

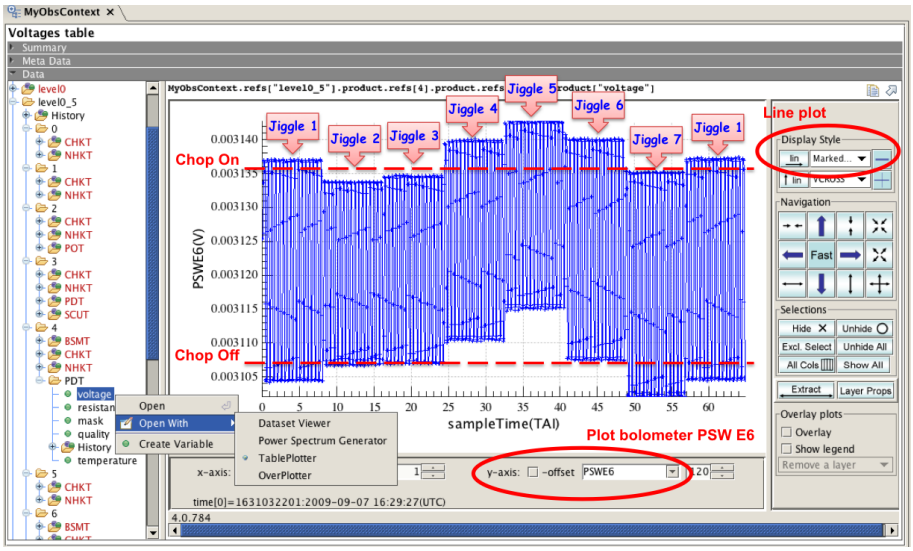


**Figure 2.34. Plotting the Level 0.5 data for a 7-point Jiggle Point Source observation**

# 2.4.5. Looking at the Raw Level 0 Data

The Raw data formatted from the satellite telemetry is also available within the Observation Context. These are the Level 0 Products and will in most circumstances be of no general interest. The Level 0 Context, shown in Figure 2.35, contains 23 entries. Note that there is a significant difference in the Level 0 data structure compared to the Level 0.5 Products. In the Level 0.5 Products, each individual block in the observation has several data types (e.g. Scan line, Housekeeping data, etc - see Table 2.3 ). However, in order to reduce the raw data volume at the Level 0 stage, all the data types are concatonated into a single Level 0 product, referred to as a *Raw SPIRE Timeline* (`RST`) for each building block, i.e. A single Level 0 product contains many seperate Table datasets. *Clicking* on a given number within the Level 0 context reveals the Level 0 Product for that particular building block. These products are the *raw* data versions of the Level 0.5 data and contain Table Datasets such as the Critical House Keeping timelines (`CHK`), Nominal House Keeping timelines (`NHK`), Raw Photometer Detector timelines (`PHOTF`), Raw BSM timelines (`BSNNOMINAL`), Raw Photometer Offset timelines (`PHOTOFF`) and Sub-Control Unit timelines (`SCUNOMINAL`). The Raw Photometer Detector Timeline ( `PHOTF`) Table Dataset can be viewed by *right-clicking* and selecting `Open-with - Dataset Viewer`, see Figure 2.35), we find quite a different structure to the Level 0.5 PDT datasets. There are 288 columns, one for every SPIRE channel, numbered not in the familiar `PSWE8`, `PSWE9` notation but rather as as `PHOTFARRAY001` -- `PHOTFARRAY288` which corresponds to their Channel Number (from an electrical designation). The signal is still in raw ADU and there are many different *time* columns which correspond to various measures of the data frames, telemetry packets and packet sequence counts, etc. The only flags are contained in the `PHOTFADCFLAGS` column which is set in the case of a problem with ADC process in telemetry. A full description of the data structure can be found in the Products Definition Document (HERSCHEL-HSC-DOC-0959) or the SPIRE Pipeline Description Document (SPIRE-RAL-DOC-002437).

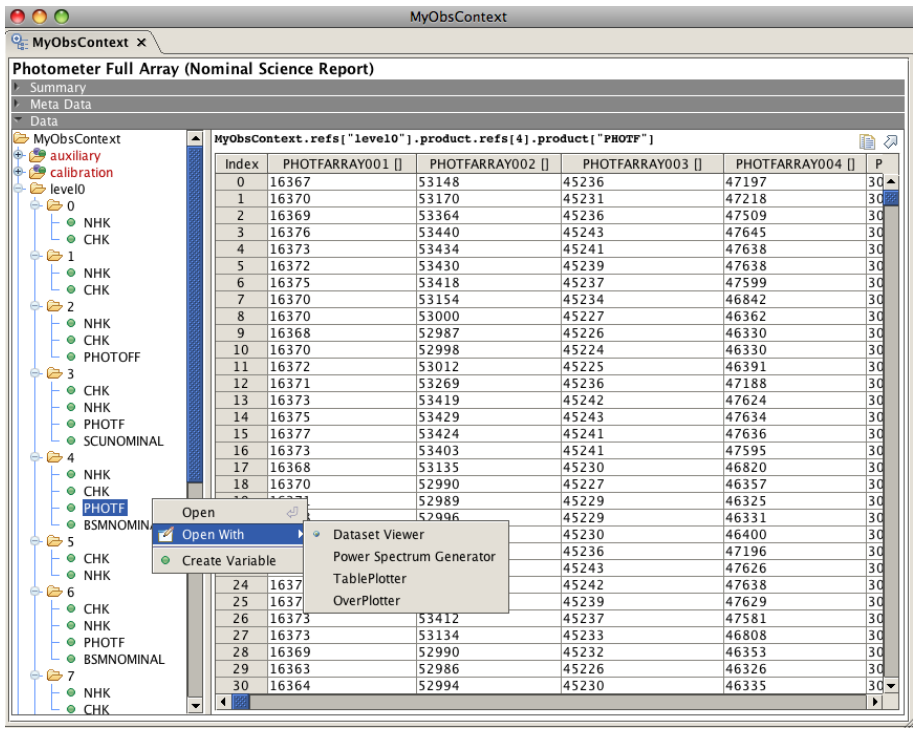


**Figure 2.35. The Level 0 Raw Data within the Observation Context**

# 2.5. SPIRE Spectroscopy Data Structure

## 2.5.1. SPIRE spectrometer introduction

This section is dedicated to familiarizing the reader with the appearance of the data from the SPIRE spectrometer within HIPE and how to visualize the data.

There are 6 different observing mode combinations for the SPIRE spectrometer (and within each of these, the spectral resolution could be High, Medium or Low). The corresponding pipeline script for each of the 6 combinations is shown belowin Figure 2.36:

| Pipeline | SOF1 | | SOF2 | | | |
|---|---|---|---|---|---|---|
| Spatial sampling | Sparse | | Intermediate | | Full | |
| Pointing | Point | Raster | Point | Raster | Point | Raster |

**Figure 2.36. SPIRE spectrometer modes**

The Level-1 data products returned by the standard pipeline are the same for all 6 observing combinations. They consist of the raw interferograms (in Volts), and the final spectra for each detector in the array calibrated assuming uniformly extended emission (in W/m2/Hz/sr). The Level-2 products, however, depend on whether the observation was “Sparse, Point” or not. For a Sparse-Point observation, the final product contains the data from the central detector pair calibrated assuming it is a point source (in Jy). For all other modes, the final product is a gridded spectral cube, calibrated assuming it is uniformly extended emission (in W/m2/Hz/sr). In all cases, for Level-1 and Level-2 spectra and spectral cubes, an unapodized and an apodized version is produced. This is summarised in Table 2.4 for the Level 1 products and Table 2.4 for the Level 2 products respectively:

**Table 2.4. Description of spectrometer Level-1 products**

| Pipeline | Sampling | Pointing | Product | Units |
|---|---|---|---|---|
| All | All | All | Interferogram before processing for every detector | V |
| All | All | All | Unapodized spectrum for every detector | W/m2/Hz/sr |
| All | All | All | Apodized spectrum for every detector | W/m2/Hz/sr |

**Table 2.5. Description of spectrometer Level-2 products**

| Pipeline | Sampling | Pointing | Product | Units |
|---|---|---|---|---|
| SOF1 | Sparse | Point | Single point spectrum for SSWD4/SLWC3 | Jy |
| SOF1 | Sparse | Raster | Gridded cube, map pixel 38”/70” | W/m2/Hz/sr |
| SOF2 | Intermediate | Point | Gridded cube, map pixel 19”/35” | W/m2/Hz/sr |
| SOF2 | Intermediate | Raster | Gridded cube, map pixel 19”/35” | W/m2/Hz/sr |
| SOF2 | Full | Point | Gridded cube, map pixel 9.5”/17.5” | W/m2/Hz/sr |
| SOF2 | Full | Raster | Gridded cube, map pixel 9.5”/17.5” | W/m2/Hz/sr |

The calibration for Level-1 spectra is based on observations of the emission from the Herschel telescope (i.e. observations of dark sky), and a model of its emission spectrum. As the telescope emission completely fills the beam in a uniform way, this gives a calibration that is appropriate for a smooth uniformly extended source. The units are given as brightness and so a measure of the beam area is necessary to convert to in-beam flux density.

For sparse-point observations, a conversion is applied to create a Level-2 spectrum calibrated assuming an unresolved point source. This correction takes account of the size of the beam, and also the difference in coupling efficiency for an extended and a point source. The correction is derived empirically by comparing the “extended” calibration derived from the telescope with a “point” calibration derived from Uranus, and the standard Herschel Uranus model. The beam size and coupling efficiency are determined from a combination of this empirical correction, and observations of the SPIRE spectrometer beam shape measured on Neptune.

For mapping observations, a spectral cube is created which re-grids the hexagonally packed detector arrays onto a rectangular grid. The units of the final data cube are W/m2/Hz/sr assuming uniformly extended emission.

In this example, a fully processed observation context is loaded into HIPE and inspected. Level-1 data products are extracted from the observation context and then visualized. Finally, portions of a data product are extracted and plotted, giving the user direct access to the data. The data, shown here, derive from an observation of the galaxy IC342. The observation was made on September 21, 2009, Herschel's Operational Day 130.

# 2.5.2. The Spectrometer Observation Context

## 2.5.2.1. Load an observation context into HIPE

In HIPE, one can access the observation contexts from data pools as follows:

1. Declare a ProductStorage: i.e. the name of the pool:

```
storage = ProductStorage("name-of-pool")
```

2. Query for an observation context which is identified by its type being equal to OBS:

```
results = storage.select(Query("type=='OBS'"))
```

3. Load the observation into the HIPE session:

```
observation = results[0].product
```

The introductory script loads three observation contexts from three separate data pools. Please refer to the script for the exact syntax. An observation context is a HIPE object which can contain several data products.

## 2.5.2.2. Inspect an observation context in HIPE

HIPE provides convenient GUI tools to inspect an observation context. Begin with the observation context for the low resolution observation (OBSID=0x50001AB8). In the HIPE Variables View, select lrObservation with a right mouse click and then Open With > Observation Viewer. HIPE will present the Summary view of the observation, including the image of four spectra, one unapodized and one apodized, derived from each of the center detectors of the two SPIRE spectrometer detector arrays: SLWC3 and SSWD4. Clicking the small arrow to the left of Summary in the observation viewer will hide the observation summary and present the detailed view of the observation context:

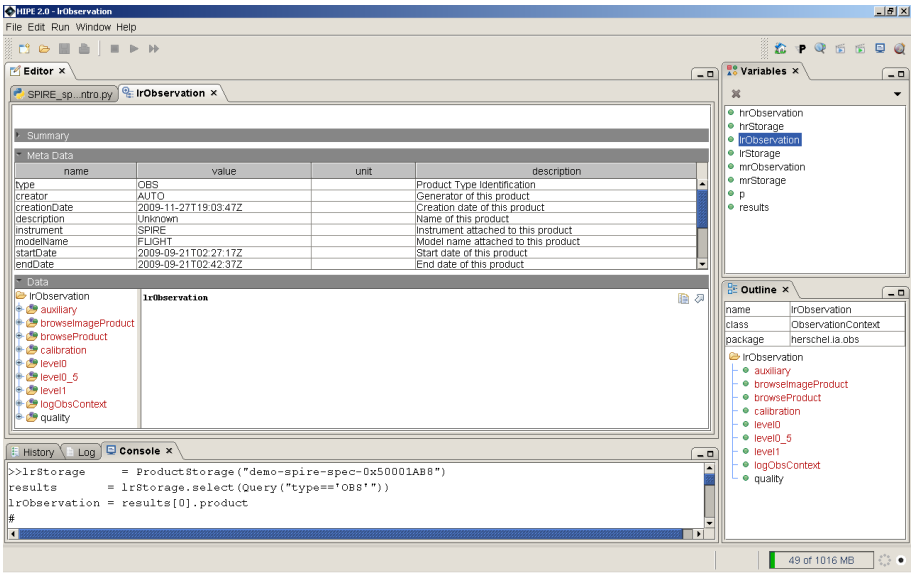


**Figure 2.37. Viewing the SPIRE observation context**

The viewing pane shows the many sub-contexts contained in the observation context in a folder-like layout.

Next, inspect the level-1 context. In the Data area of the Editor for lrObservation, select level1 with a right mouse click and select Open With > Context Viewer. Inside the Level 1 context there is one main entry named “Point_0_Jiggle_0_LR” which stands for the first and only raster point (index 0), the first and only jiggle position (index 0) at Low Resolution. This is the only building block contained in this observation. Double-click this building block to see the three entries it contains. Each one represents a different SPIRE spectrometer level-1 data product:

1. apodized_spectrum: Level 1 Apodized Spectrum Product

2. interferogram: Level 1 Interferogram Product

3. unapodized_spectrum: Level 1 Unapodized Spectrum Product

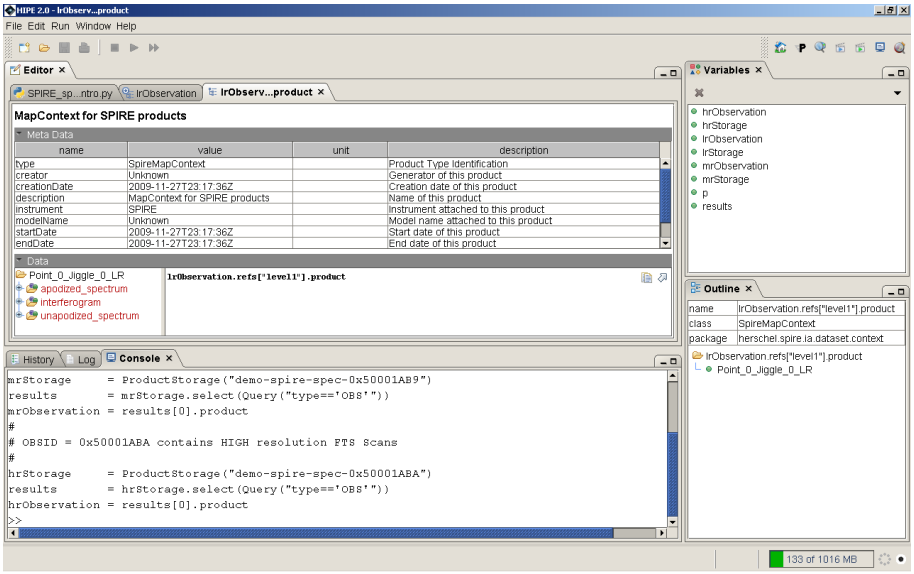


**Figure 2.38. Viewing the SPIRE Level 1 context**

# 2.5.3. The Spectrometer Level 1 Data Products

## 2.5.3.1. Extract the Level 1 data products

Before inspecting the contents of the level-1 data products, we first extract a selection of these products as separate variables in HIPE. The syntax required to access a level-1 product within an observation context is as follows:

```
Level1Product = observation.refs["level1"].product.
refs[BuildingBlock].product.refs[ProductName].product
```

For example, the following command will extract the level-1 interferogram product from the high resolution observation context:

```
hrInterferogram = hrObservation.refs["level1"].product.
refs["Point_0_Jiggle_0_HR"].product.refs["interferogram"].product
```

Note that the right hand side of this command is spelled out at the top of the Data area of the Context Viewer in HIPE. Clicking the copy icon at the top right corner will copy the command string into the clipboard and can then be pasted into the command console.

## 2.5.3.2. Inspect the Level 1 data products

HIPE offers dedicated visualization tools to inspect the level-1 interferogram and spectrum products.

The following steps demonstrate how one can inspect the contents of the datasets within a level-1 data product as tables. In this example, a dataset in the level-1 interferogram product of the high resolution observation is examined.

1. Select the hrInterferogram variable with right mouse click, select Open With > Product Viewer.

2. Scroll down to the bottom of the newly opened view. Within the folder-like structure, unfold Dataset 0001 by clicking the plus symbol to its left and select SLWC3 with a right mouse click. Select Open With > Dataset Viewer to view the numeric values of the dataset.

3. These values can be easily written into a text file with comma-separated values with the command quoted below. The equivalent command will work to save a particular spectrum into a text file:
```
asciiTableWriter(file="C:/SLWC3Interferogram.txt",
table=hrInterferogram["0001"]["SLWC3"])

asciiTableWriter(file="C:/SLWC3Spectrum.txt",
table=hrSpectrum["0000"]["SLWC3"])
```

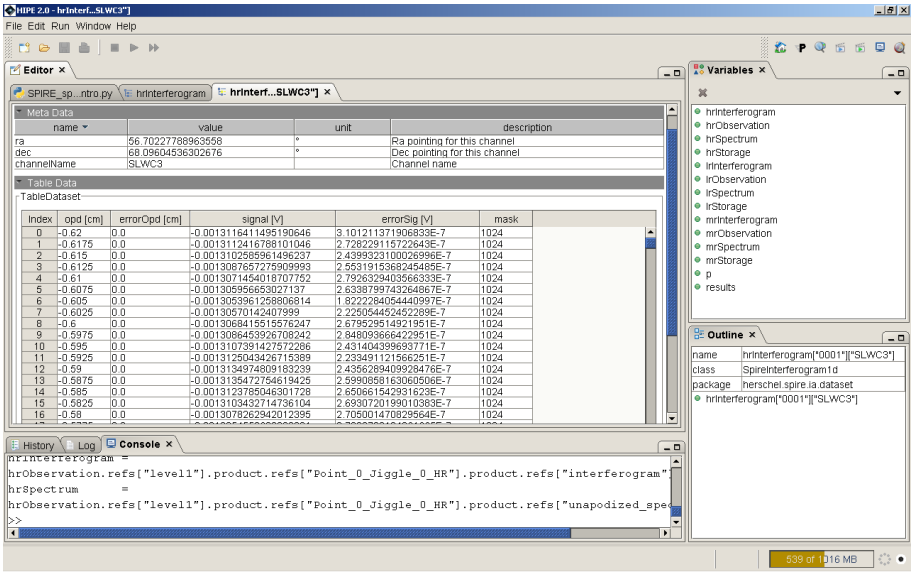

**Figure 2.39. Inspecting data from a level-1 product as tables**

The following steps demonstrate how one can conveniently plot the contents of the level-1 data product. In this example, the interferograms for a given detector in the level-1 interferogram product of the high resolution observation are examined.

1. In the Variables pane, select the hrInterferogram variable with right mouse click, select Open With > Spec SDI Explorer. Do the same for mrInterferogram, and lrInterferogram.

2. In the hrInterferogram view, select detector SLWC3 with a left mouse click. In the other views, select the same detector but do so with a *double-click* of the left mouse button to over-plot the interferograms.

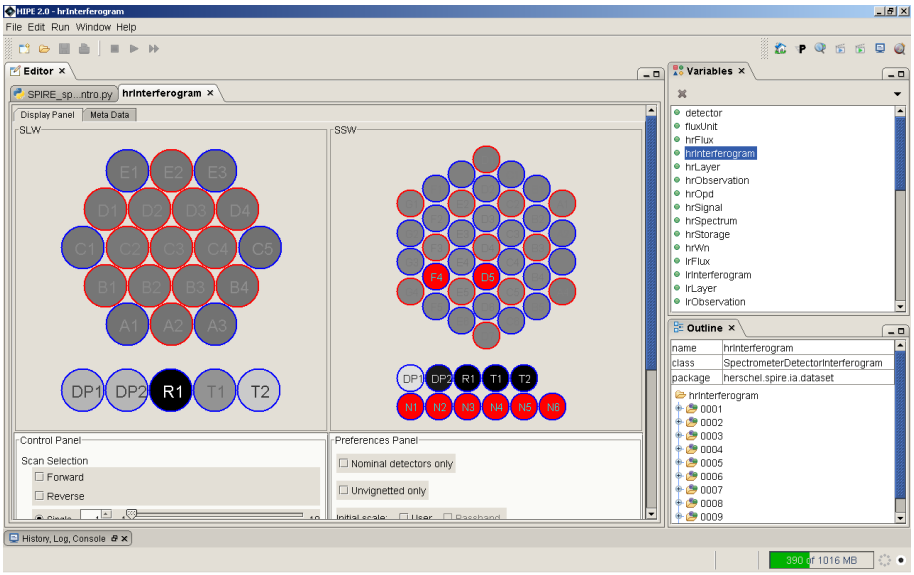

**Figure 2.40. The SDI Explorer allows to select and plot data from a level-1 interferogram product**

## 2.5.3.3. Extract and plot Level 1 data

The remainder of this chapter shows how to extract and plot interferograms and spectra:

1. Extract the individual data vectors from the product datasets

   General syntax:

   ```
   wave = spectrum[scanNumber][detector].getWave()
   flux = spectrum[scanNumber][detector].getFlux()
   ```

   Specific syntax:

   ```
   hrWn = hrSpectrum[0]["SLWC3"].getWave()
   hrFlux = hrSpectrum[0]["SLWC3"].getFlux()
   ```

2. Plot the results.

   General syntax:

   ```
   p = PlotXY()
   p.addLayer(LayerXY(x,y))
   ```

   Specific sample syntax:

   ```
   detector = "SLWC3"
   plotTitle = "Inspect Level 1 Spectra "+detector
   p = PlotXY(titleText = plotTitle)
   hrLayer = LayerXY(hrWn, hrFlux, name="HR")
   p.addLayer(hrLayer)
   ```

Using the above examples, the following plots should be displayed:

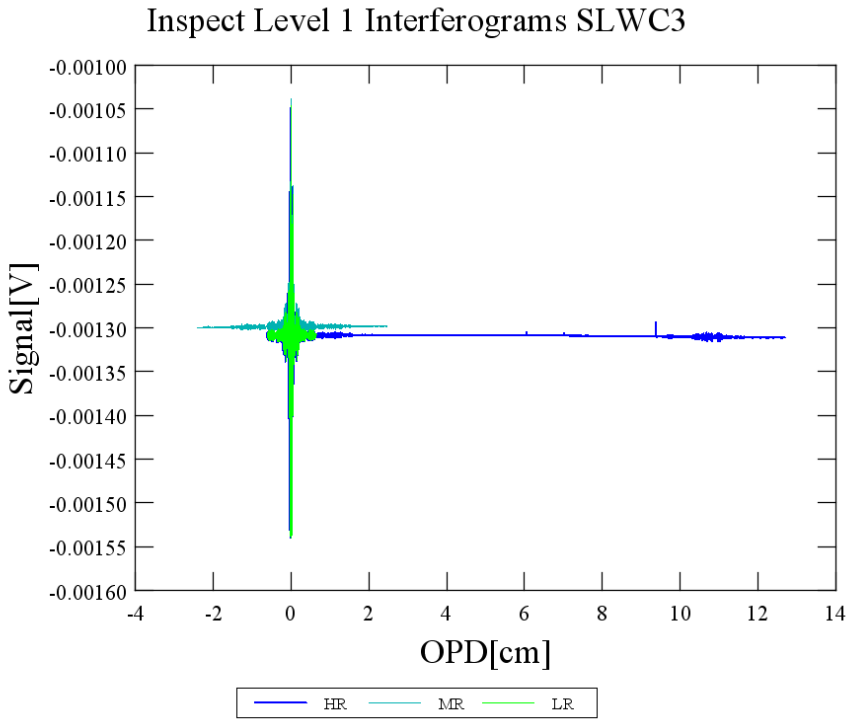


**Figure 2.41. Comparing three interferograms from the SLWC3 detector**

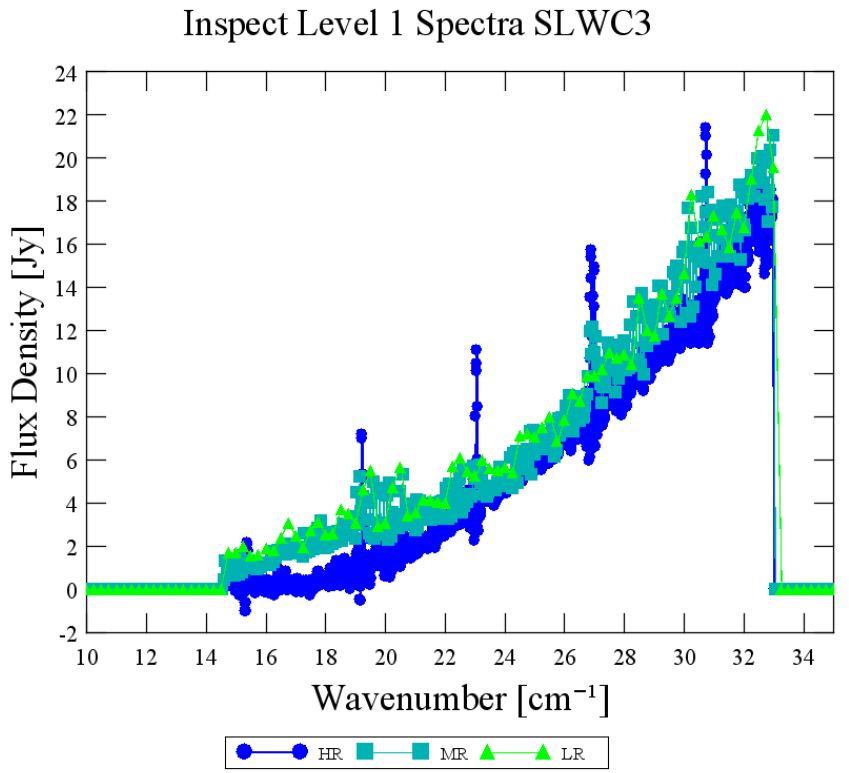


**Figure 2.42. Comparing three spectra from the SLWC3 detector**

# 2.5.4. Using SpecExplorer

The Spectrometer Detector Explorer, also known as SpecExplorer, is a GUI-based visualization tool that allows efficient inspection of the contents of the two SPIRE products: Spectrometer Detector Interferogram (SDI) and Spectrometer Detector Spectrum (SDS). The following sections detail the features of SpecExplorer.

## 2.5.4.1. Starting SpecExplorer

SpecExplorer is an application that can be called from the interactive data processing environment HIPE. At least one instance of one of the classes Spectrometer Detector Interferogram or Spectrometer Detector Spectrum must already be available in memory. For example, such a product can be loaded into memory via the Product Access Layer.

In HIPE, identify the product for visualization from the Products list and right-click the Spectrometer Detector Interferogram or Spectrometer Detector Spectrum product, follow the “Open With” menu entry and select the SpecExplorer from the drop-down menu (see Figure 2.43).

The SpecExplorer can also be called from the command line. SpecExplorer will visualize any Spectrometer Detector Interferogram (SDI) or Spectrometer Detector Spectrum product (SDS). In the HIPE command line window, we will load a product of the name SDS after entering the following commands:

```
from herschel.spire.ia.gui import SpecExplorer

SpecExplorer(SDS)
```

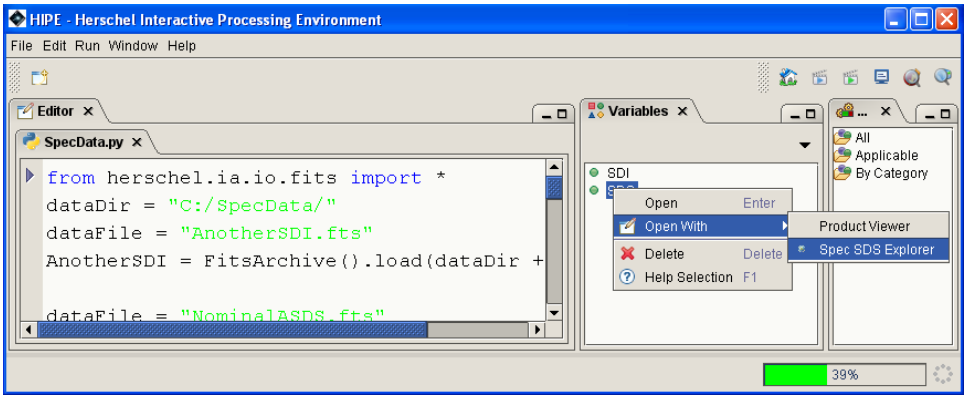


**Figure 2.43. Starting the SpecExplorer via HIPE.**

## 2.5.4.2. SpecExplorer Layout

The Graphical User Interface of the SpecExplorer is divided into four sections: The Bolometer Detector Arrays Spectrometer Long Wavelength (SLW) on the top to the left, the Spectrometer Short Wavelength (SSW) on the top to the right. The Control Panel is on the bottom left and the Preferences Panel is on the bottom right (see the Figure 2.44).

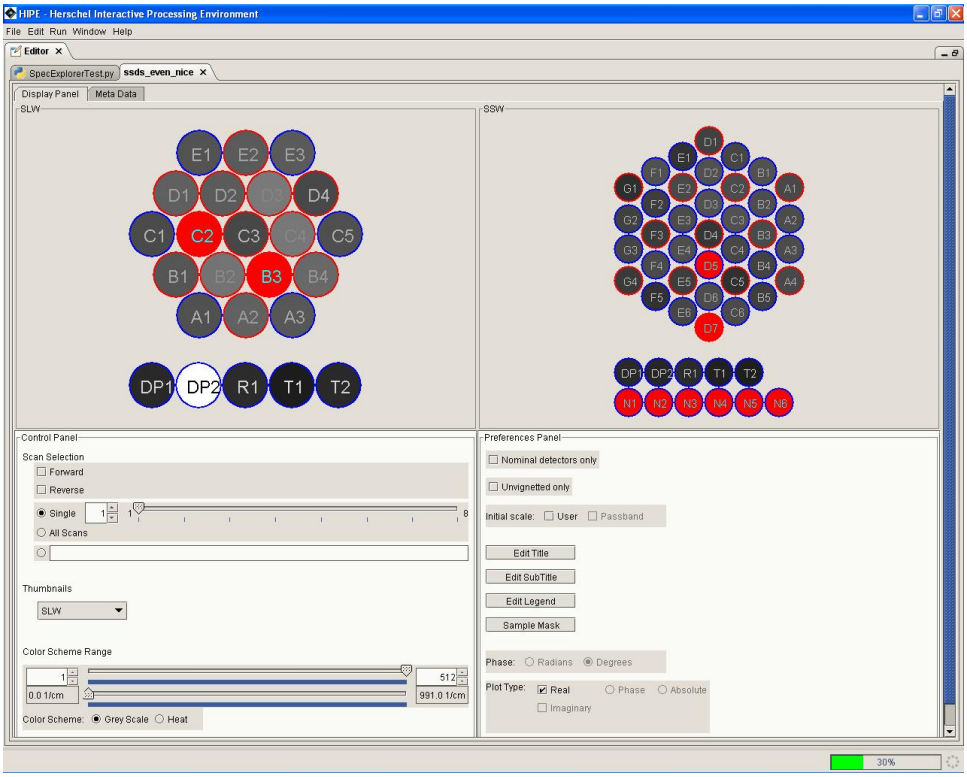


**Figure 2.44. SpecExplorer Graphical User Interface.**

## Bolometer Detector Arrays Display

The top panels of the SpecExplorer contain the display of the two detector arrays (see Figure 2.45).

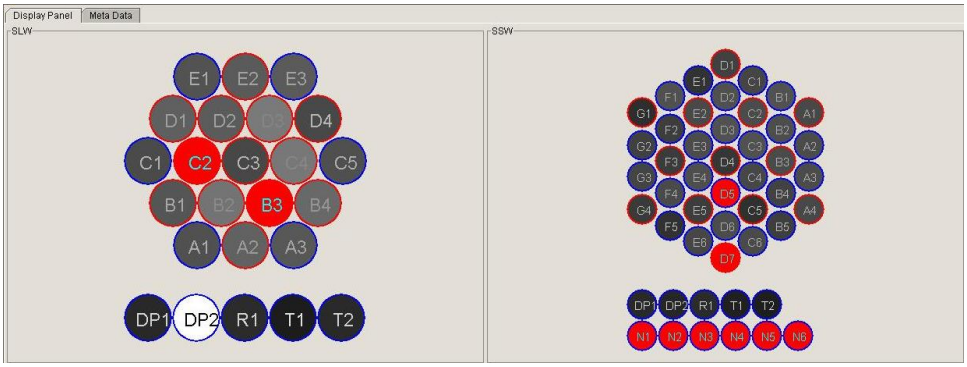


**Figure 2.45. SpecExplorer – Bolometer Detector Arrays Display.**

This display allows the user to select any of the detectors of the SPIRE spectrometer: The long wavelength array on the left and the short wavelength array on the right. The detector layout reflects their arrangement in a honeycomb pattern. Clicking on one or more of these detectors allows the user to plot the data said detector recorded.

## Control Panel

The Control Panel (see Figure 2.46) allows users to:

- Select a subset of the scans within the data product (Scan Selection).
- Create plots of many datasets on one page (Thumbnails).
- Define the fill colours for the detectors (Colour Scheme Range).

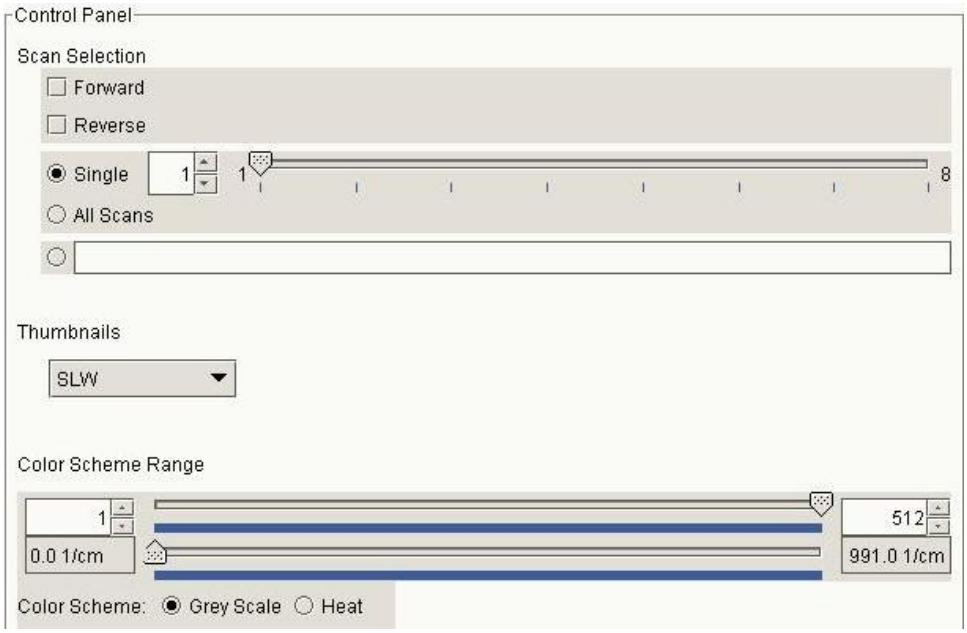


**Figure 2.46. SpecExplorer - Control Panel.**

## Scan Selection

This section of the Control Panel (see Figure 2.47) allows the user to select which scans are plotted:

- The *Forward* and *Reverse* buttons allow the plotting of scans based on the direction of those scans. Since all scans should be either forward or reverse, checking both options will plot all the scans in one product.
- The *Single* option allows the user to plot one scan at a time.
- The *All Scans* option plots all the scans of a given detector in the product.
- *Free Text:* Users can specify a range of scans to be plotted in a free text field. For example, if the user wishes to plot scans 1 through 4 and scans 7 and 8, this can be specified as follows in the user selected scans section: 1-4 7,8. The wildcard * will select all scans.

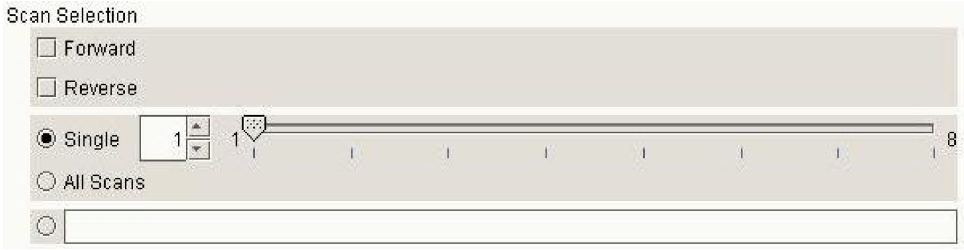


**Figure 2.47. Control Panel - Scan Selection.**

## Thumbnails

In order to enable the user to compare data from many detectors, the SpecExplorer allows the user to create numerous plots on a single page. The resulting data plots are small in order to fit all requested plots on one screen, leading to "thumbnail" images of the data. The result is a single window that contains a number of thumbnail data plots arranged in the same pattern as that of the detectors in the Bolometer Detector Array display, a honeycomb pattern. The scaling of the main plot window is applied to each thumbnail image. If no plot window is currently open, the selection of the Initial Scale on the Preferences Panel is applied. The Initial Scale is basically the scaling that allows for a focused view of the plotted data (see Figure 2.50). Three selections are available under the Thumbnail drop-down menu (see Figure 2.48):

1. *SLW* to plot data which were recorded by the detectors in the long wavelength detector array. Depending on the selection in the Preferences Panel, data are shown only for the nominal or unvignetted detectors.

2. *SSW* to plot data which were recorded by the detectors in the short wavelength detector array. Depending on the selection in the Preferences Panel, data are shown only for the nominal or unvignetted detectors.

3. *Co-Aligned* to plot data which were recorded by the co-aligned detectors in SLW and SSW. Depending on the selection in the Preferences Panel, data are shown for all the nominal or only the unvignetted co-aligned detectors.

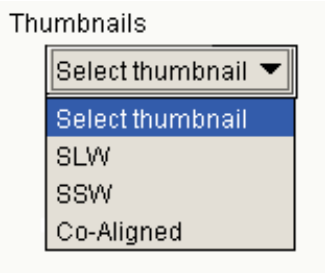


**Figure 2.48. Control Panel – Thumbnails – Thumbnails Selections.**

### Colour Scheme Range

This section allows the user to change and control the colour scheme used to determine the colours used in the honeycomb "images" for the Detectors Display (see Figure 2.49). Two colour schemes are available which both go from white (high values) to black (low values): Grey Scale and Heat. The values for the colour scheme are set to the average of the detector data within a user-specified data range. The range slider, and the indices and values displayed next to it, specify the abscissa range in the interferograms or spectra which is used to compute the average signal value and subsequently set the colours. Note that only the abscissa indices, not the values, can be entered by the user.

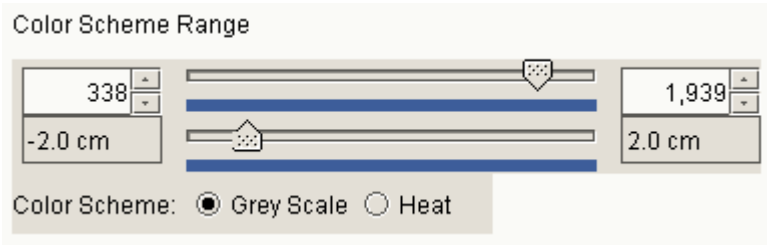


**Figure 2.49. Control Panel - Colour Scheme.**

## Preferences Panel

This panel (see Figure 2.50) allows the user to :

- Select which detectors are to be displayed in the Bolometer Detector Array Display and the Thumbnails.
- Select the initial scale of the plots.
- Customize the title, subtitle, and legend entries for the main plot area.
- Select whether spectral phase is given in units of radians or degrees. Note that the spectral phase $\phi(S)$ is defined as $\tan \phi\,(S)$ = Imaginary (S) / Real (S).
- Select the quantity used in the plot when complex data are presented, to either Real or Imaginary, or Phase, or Absolute. This selection is only available if the product contains complex data with real and imaginary components.

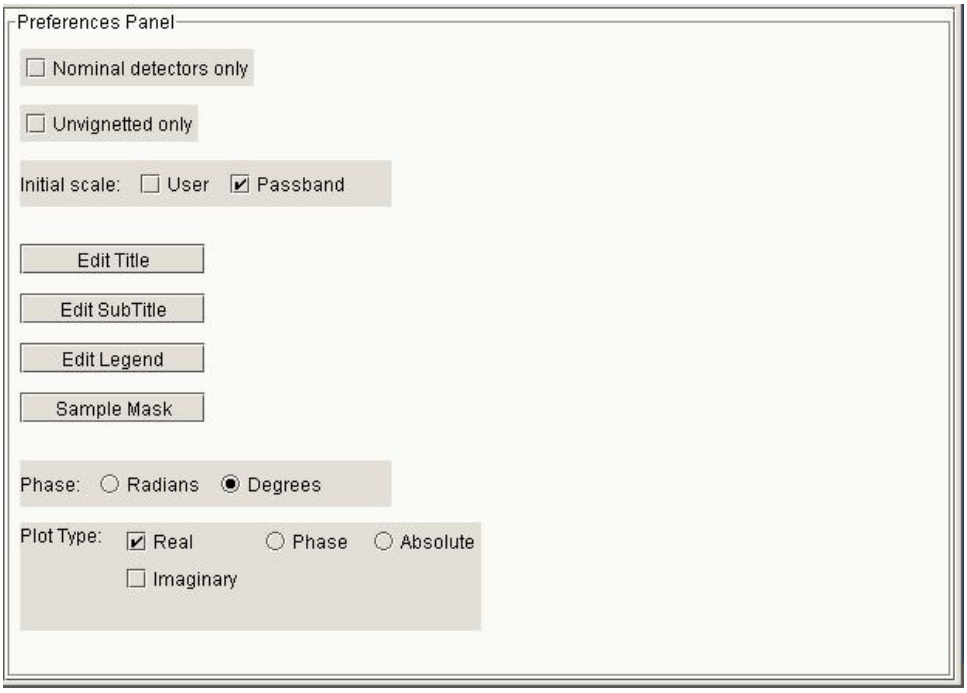


**Figure 2.50. SpecExplorer - Preferences Panel.**

The “Nominal detectors only” option allows the user to display only the nominal detectors in the detector arrays, i.e. those detectors which make sky observations. The “Unvignetted only” option allows the user to display only the unvignetted detectors in the detector arrays, i.e. those detectors which have an unvignetted Field of View through the Herschel telescope.

The two selection check boxes for the “Initial scale” allow the user to select whether the initial scale of a plot reflects the last user choice (“User”) or whether the plot presents the optical passband defined by the instrument, i.e. 10 – 35 $cm^{-1}$ for data from SLW and 25 – 55 $cm^{-1}$ for data from SSW and 10 – 55 $cm^{-1}$ if data are plotted from detectors from both arrays (“Passband”). The ordinate will scale with the data for the passband option. If neither box is checked, then the plot will self-scale according to the data.

The edit buttons allow the user to customize the title, subtitle, and legend of the main plot area. All descriptors from the data product are available regardless of the level where the metadata reside.

In case the spectral phase is plotted, the “Phase” section allows the user to plot the phase in Radians from $-\pi / 2$ to $+\pi / 2$ or in Degrees from -180° to 180°. This selection only applies to the first time when phase data are plotted. Changing this selection subsequently does not have any effect on the plot until a new plot is created.

The “Plot Type” section allows the user to specify which aspect of the spectral flux is plotted where it is given as a complex number. The absolute value of a complex number is given by $\sqrt{Imaginary(s)^2 + Real(s)^2}$.

## 2.5.4.3. Example 1: Plotting and Overplotting

In order to inspect data from a specific scan and detector from a specific product, perform the following steps:

1. Start the SpecExplorer for the product in question.

2. In the Scan Selection section of the Control Panel, specify which scan(s) should be plotted, e.g. all reverse scans.

3. In the Preferences Panel, click the buttons for edit titles, subtitles, and legends to customize these fields. Fields are populated by the entries in the inspected product and free text can be added by the user. Default title, subtitle, and legend information are stored and can be retrieved through the customization window. See Figure 2.51.

4. In the Preferences Panel, select the Initial Scaling needed, e.g. Passband.

5. In the Preferences Panel, select the Plot Type, e.g. the Absolute value of a complex number.

6. In the Detectors Display shown in Figure 2.44, single left mouse click the detector to plot its data, e.g. SLWD3. A new PlotXY window will open with the SSWD3 detector plotted as shown in Figure 2.52.

7. For an overplot, double-click with the left mouse button on an additional detector to plot its data, e.g. SSWC2 (see Figure 2.53). A single click would have created a new plot containing data from SSWC2 only.

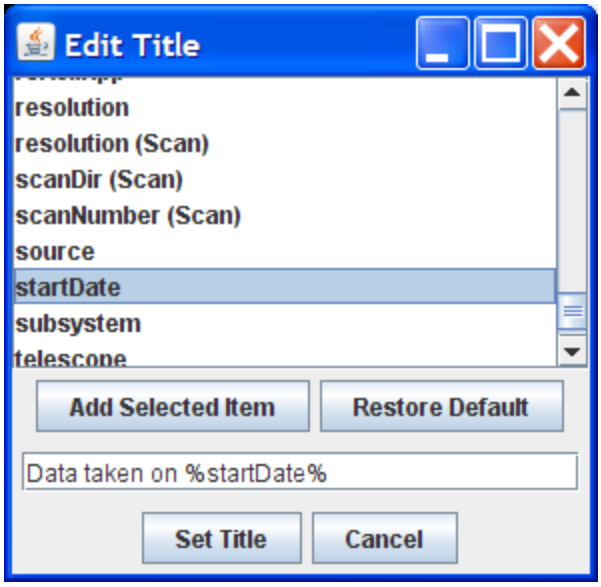


**Figure 2.51. Edit Title**

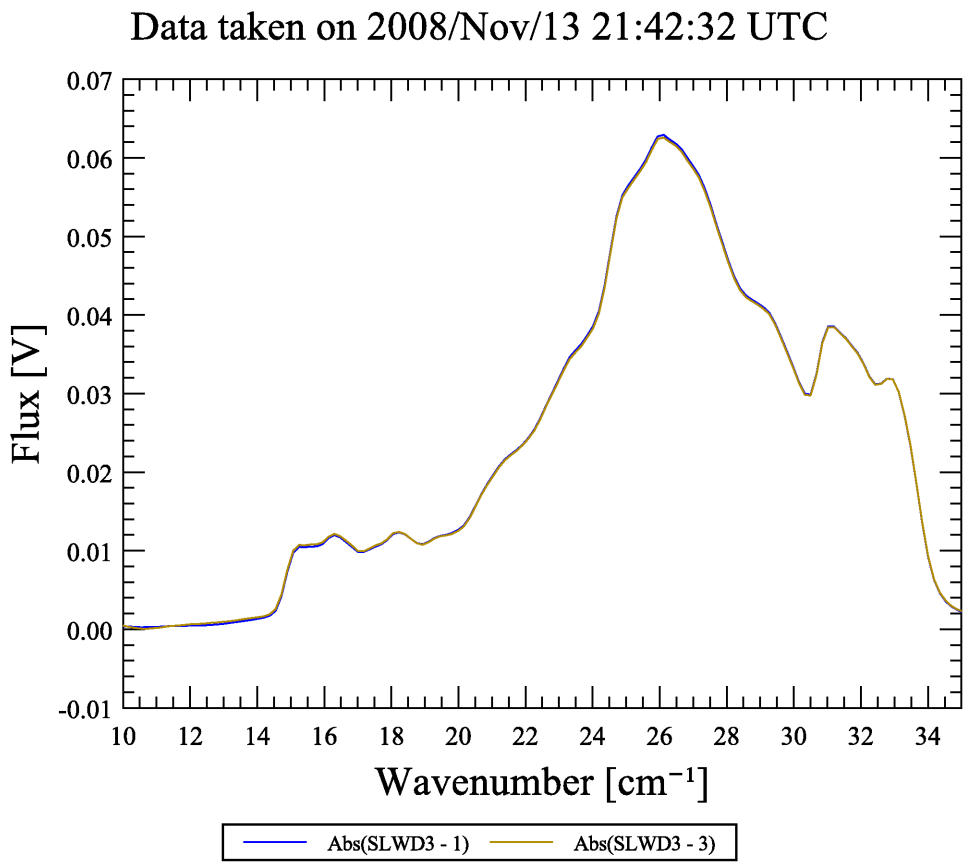


**Figure 2.52. Single plot of the reverse scans 1 and 3.**

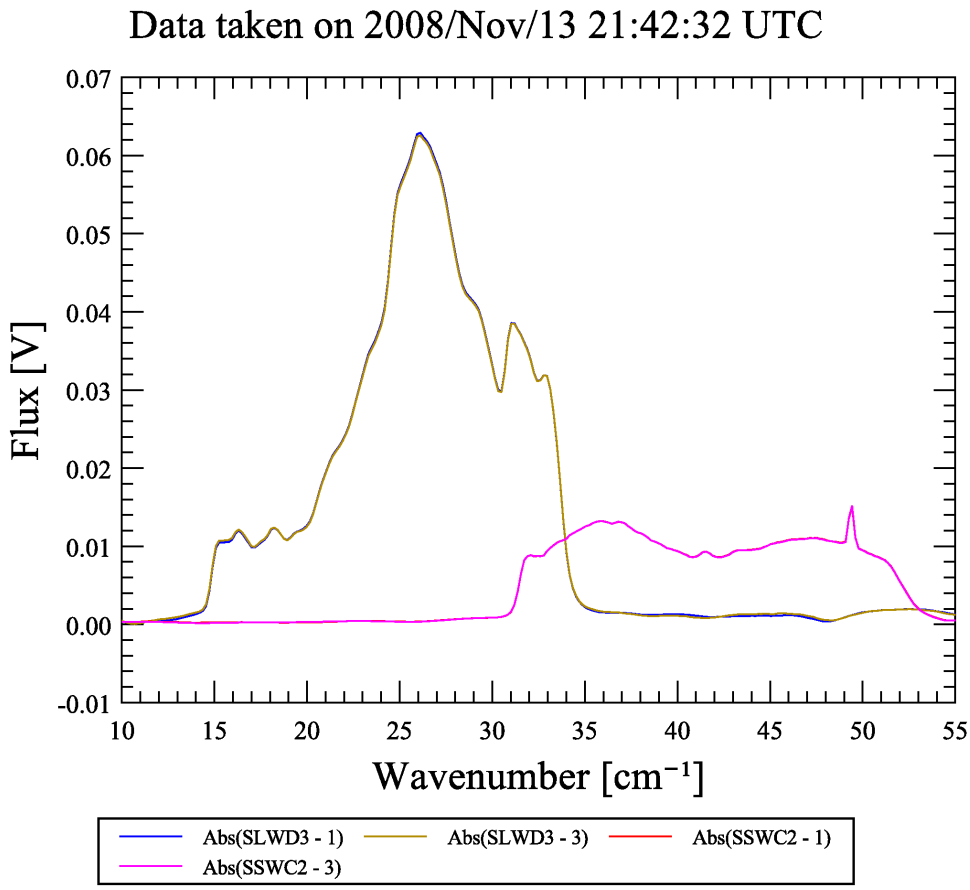


**Figure 2.53. Overplot of data from two different detectors in two different detector arrays.**

## 2.5.4.4. Example 2: Making a Thumbnail Image

In order to compare data from different detectors on the same page perform the following steps:

1. Start the SpecExplorer for the product in question.

2. In the Scan Selection section of the Control Panel specify which scan(s) should be plotted, e.g. scan number 1.

3. Open the main plot window by performing the steps in Section 2.5.4.3, select the range to be plotted on the thumbnail images, e.g. from 25 $cm^{-1}$ to 40 $cm^{-1}$.

4. In the Preferences Panel, check whether to get thumbnail images from all, only the nominal, or only the unvignetted detectors, e.g. “Unvignetted only”.

5. From the Thumbnails drop-down menu on the Control Panel, select to get thumbnail images from SLW, SSW, or the co-aligned detectors on SLW and SSW (see Figure 2.54).

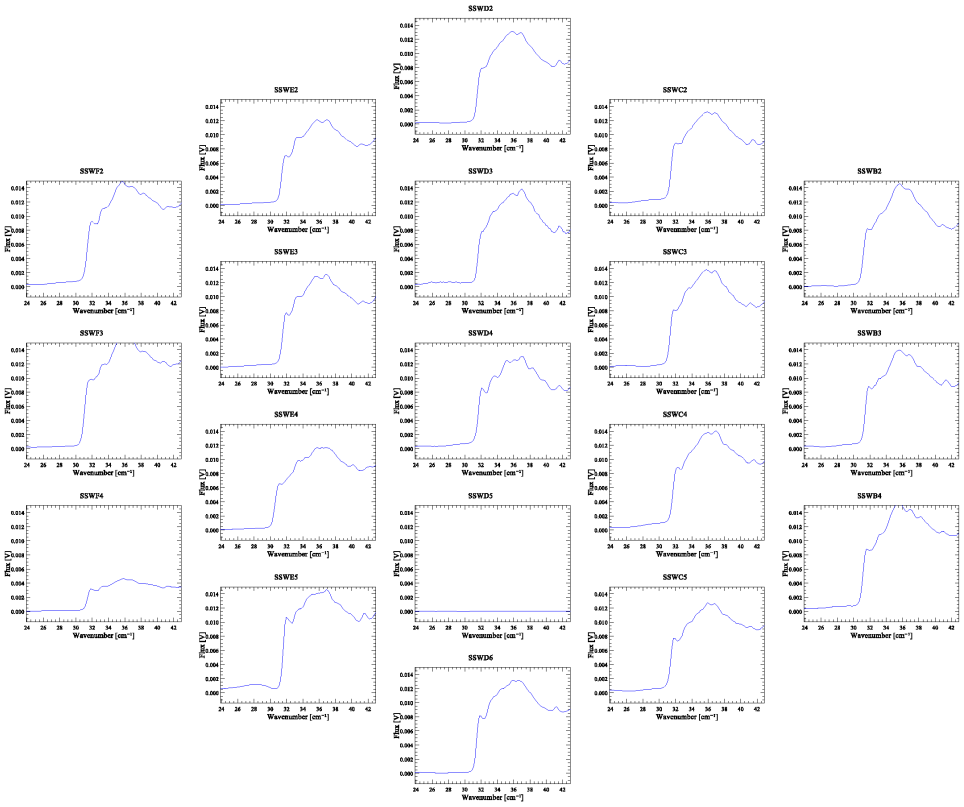


**Figure 2.54. Thumbnail images of the unvignetted SSW detectors in the spectral region selected in the main plot window.**

# 2.5.5. The Spectrometer Level 0.5 Data Products

The Level 1 data products were created from the lower Level 0.5 data products (which were correspondingly created from processing the raw Level 0 data through the Common Engineering Conversion (Level 0 - Level 0.5) Pipeline). The Level 0.5 data are the voltage calibrated, timelines measured in **Volts** uncorrected for detector effects. These level 0.5 products are also available from the Observation Context. The Level 0.5 context folder can be seen in the Observation Context and can be opened by *clicking* on the + next to the `level0_5` folder. The Level 0.5 context contains a lot more data than the Level 1 context and includes all the data necessary to process the observation and produce science quality data. In Figure 2.55 we show all the Level 0.5 data within the observation context. We see that there are a total of 15 entries in the list informatively labelled from 0 to 14. This can be compared to the single final product that we saw for the Level 1 data. The Level 0.5 context contains all the building blocks used in the observation and in Figure 2.55 we show how this *spectrometer* observation was built up from the individual building blocks. In the figure, the building blocks can be divided into roughly 4 general types, configuration blocks, calibration blocks, science blocks and movement blocks. The type of building block can be revealed by *clicking* on a given number from 0-14 and scrolling down the `Meta data` window pane to the `BBtypeName` entry. The individual blocks are described below in Table 2.6.

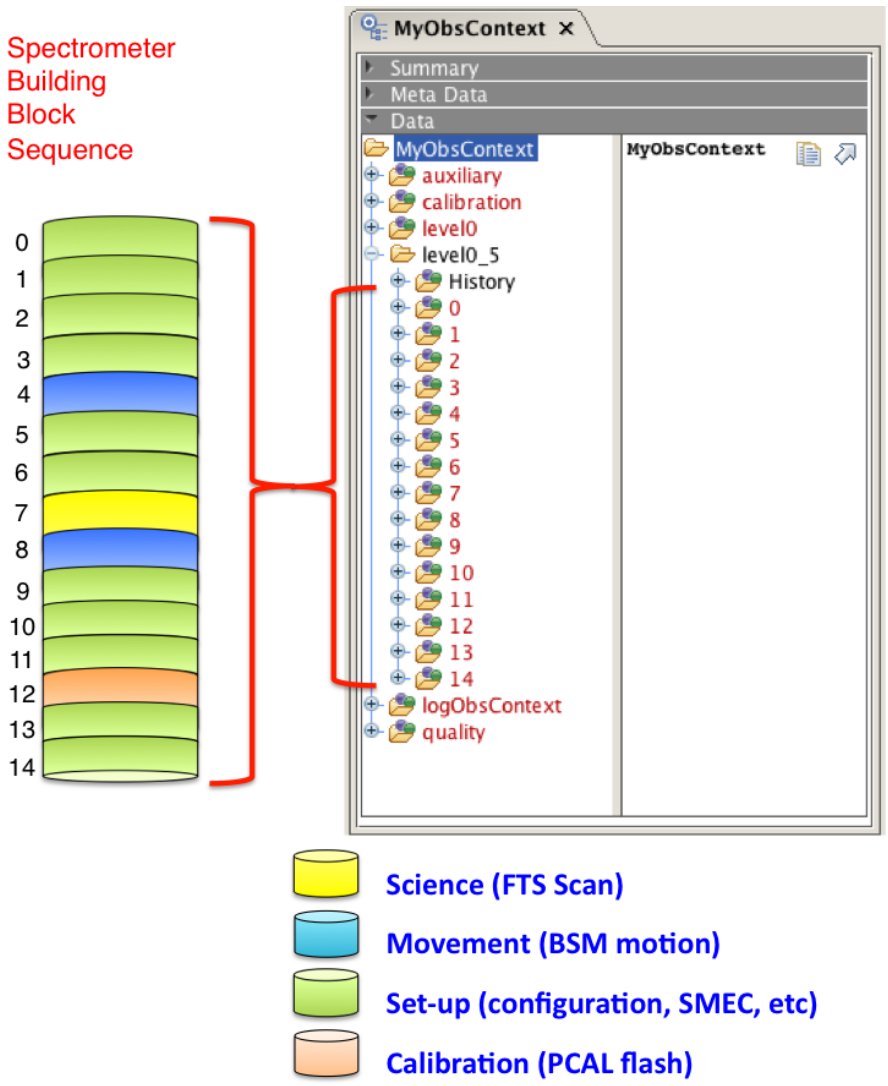


**Figure 2.55. Anatomy of Level 0.5 Building Block structure for a spectrometer observation**

**Table 2.6. Description of the Building Blocks in a Spectrometer Level 0.5 Context**

| BB number | BB Type | BB Hex prefix | Description |
|---|---|---|---|
| 0 | SpireBb_StartObsAll | 0xB6C8 | Begin Observation |
| 1 | SpireBbSOF1Config | 0xA0B0 | Initial configuration of the Spectrometer SOF1 AOT |
| 2 | SpireBbSmecInit | 0x8213 | Initialize the SMEC |
| 3 | SpireBbSOF1Init | 0xA0B1 | Initialize the AOT |
| 4 | SpireBbBsmMove | 0xA107 | Move BSM to position for this set of FTS scans |
| 5 | SpireBbSetBsmSam-pling | 0x8641 | Set BSM sampling rate for FTS scanning |
| 6 | SpireBbSetSmecSam-pling | 0x8642 | Set SMEC sampling rate for FTS scanning |
| 7 | SpireBbFtsScan | 0xAF00 | Science FTS scans |
| 8 | SpireBb_BsmMove | 0xB6CC | Reset BSM position after scanning |

| BB number | BB Type | BB Hex prefix | Description |
|---|---|---|---|
| 9 | SpireBb_MoveSmec2Home | 0xB6C2 | Move SMEC to home position after scanning |
| 10 | SpireBbSetBsmSampling | 0x8641 | Reset BSM sampling rate after scanning |
| 11 | SpireBbSetSmecSampling | 0x8642 | Reset SMEC sampling rate after scanning |
| 12 | SpireBbPcalFlash | 0xB6B9 | Calibration Lamp Flash |
| 13 | SpireBb_MoveSmec | 0xB6C3 | Move SMEC to rest position |
| 14 | SpireBbSOF1End | 0xA0B2 | End AOT Observation |

Looking at some of the individual entries in the Level 0.5 context, it can be seen that the individual Building Blocks are built up from a variety of different types of Products. *clicking* on the + sign for a given Building Block number reveals what Products a particular Building Block is made from. In Figure 2.56 the first handful of building blocks for our observation are opened to view the contents. The contents are a variety of Products referred to by ancronyms such as `CHKT`, `NHKT`, `SDT`, `BSMT`, `SOT`, `SCUT`, etc, described in order of importance below;

Example building blocks may be;

- **SDT:** The Spectrometer Detector Timeline contains the Level 0.5 detector data.
- **BSMT:** The Beam Steering Mechanism Timeline contains the information of the BSM.
- **SMECT:** TheSpectrometer Mechanism Timeline contains the information of the position of the SMEC (the moving FTS mirror) as a function of time.
- **NHKT:** The Nominal House Keeping Timeline contains the housekeeping data with all the settings for this observation.
- **CHKT:** The Critical House Keeping Timeline contains all the critical parameters of the instrument such as the electronics.
- **SCUT:** The Sub Control Unit Timeline contains monitoring data for the instrument operation for this observation.
- **SOT:** The Spectrometer Offset Timeline contains all the raw DC offsets in ADU that have already been used in the raw data processing to set the dynamic range of the detectors.
- **MCUET:** The Mechanism Control Unit Engineering Timeline contains information on the SMEC (position sensors etc).

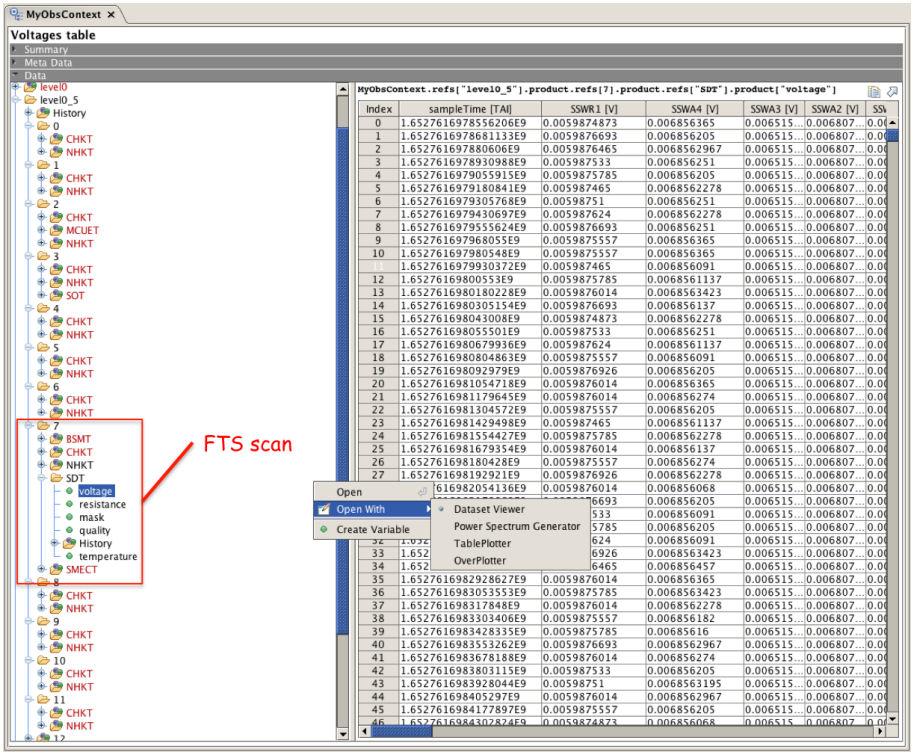


**Figure 2.56. Inside the Level 0.5 Building Block structure for a spectrometer observation**

The `CHKT`, `NHKT`, `BSMT`, `SOT`, `SCUT` Products all contain a `signal` table, containg data arrays and a `Mask` table containing flag information. The Level 0.5 `SDT` Spectrometer Detector Timeline Products contain 5 Table dataset arrays;

- **Voltage Table:** A table containing the Sample Time (in seconds) and a column for the signal measured in Volts for every bolometer including both detector and non-detector (e.g. thermistor, resistor) channels.

- **Resistance Table:** A table containing the Sample Time (in seconds) and a column for the Resistance measured in Ohms for every bolometer including both detector and non-detector (e.g. thermistor, resistor) channels.

- **Mask Table:** A table containing the Sample Time (in seconds) and a column for every bolometer including both detector and non-detector (e.g. thermistor, resistor) channels with a mask value corresponding to which processing flags have been raised. The masks are defined in the **SPIRE Pipeline User Guide** document

- **Quality Table:** A table containing any Quality Flags raised for each detector.

- **Temperature Table:** A table containing the Sample Time (in seconds) and the temperature of the Thermistors in Kelvin.

In Figure 2.56 the SDT Building Block has been selected. *Right-clicking* and selecting `Open-with - Dataset Viewer`, opens the `voltage` table in a new window. Any of the Table Data Sets can also be viewed graphically by selecting `Open-with - Table Plotter` as shown in Figure 2.57. In the plot window the bolometer signal to plot can be selected from the `Y-axis` menu (circled in the plot window) and in this example the signal versus sample time for bolometer SSW D4 has been selected. In the figure, the forward and reverse scans of the SMEC can be seen.

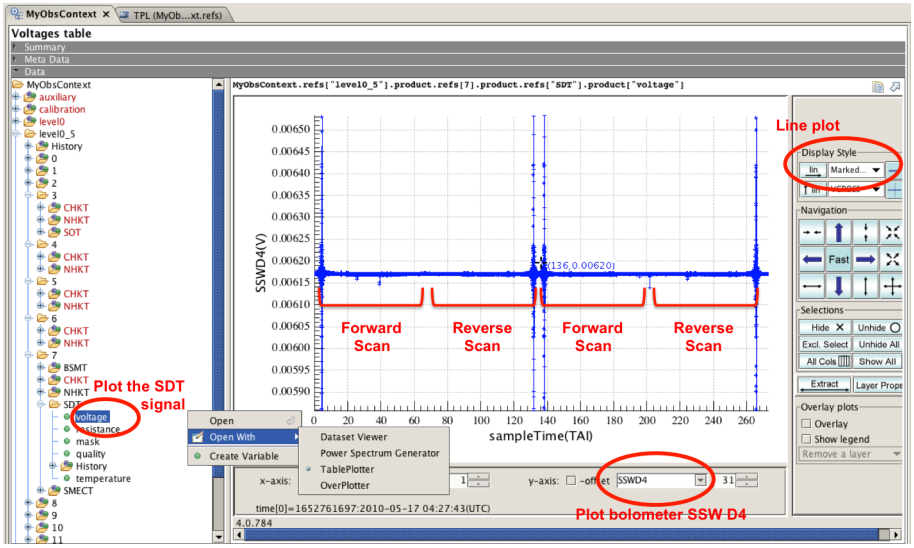


**Figure 2.57. Plotting the Level 0.5 data for a Spectrometer observation**

# 2.5.6. Looking at the Raw Level 0 Data

The Raw data formatted from the satellite telemetry is also available within the Observation Context. These are the Level 0 Products and will in most circumstances be of no general interest. The Level 0 Context, shown in Figure 2.58, contains 15 entries. Note that there is a significant difference in the Level 0 data structure compared to the Level 0.5 Products. In the Level 0.5 Products, each individual block in the observation has several data types (e.g. Scan line, Housekeeping data, etc - see Table 2.6 ). However, in order to reduce the raw data volume at the Level 0 stage, all the data types are concatonated into a single Level 0 product, referred to as a *Raw SPIRE Timeline* (`RST`) for each building block, i.e. A single Level 0 product contains many seperate Table datasets. *Clicking* on a given number within the Level 0 context reveals the Level 0 Product for that particular building block. These products are the *raw* data versions of the Level 0.5 data and contain Table Datasets such as the Critical House Keeping timelines (`CHK`), Nominal House Keeping timelines (`NHK`), Raw Spectrometer Detector timelines (`SPECF`), Raw SMEC timelines (`SMECSELECT`), Raw BSM timelines (`BSNNOMINAL`), Raw Spectrometer Offset timelines (`SPECOFF`) and Sub-Control Unit timelines (`SCUNOMINAL`). The Raw Spectrometer Detector Timeline ( `SPECF`) Table Dataset can be viewed by *right-clicking* and selecting `Open-with - Dataset Viewer`, see Figure 2.58), we find quite a different structure to the Level 0.5 SDT datasets. There are 72 columns, one for every SPIRE channel, numbered not in the familiar `SSWD4`, `SLWC3` notation but rather as as `SPECFARRAY001` -- `SPECFARRAY072` which corresponds to their Channel Number (from an electrical designation). The signal is still in raw ADU and there are many different *time* columns which correspond to various measures of the data frames, telemetry packets and packet sequence counts, etc. The only flags are contained in the `SPECFADCFLAGS` column which is set in the case of a problem with ADC process in telemetry. A full description of the data structure can be found in the Products Definition Document (HERSCHEL-HSC-DOC-0959) or the SPIRE Pipeline Description Document (SPIRE-RAL-DOC-002437).

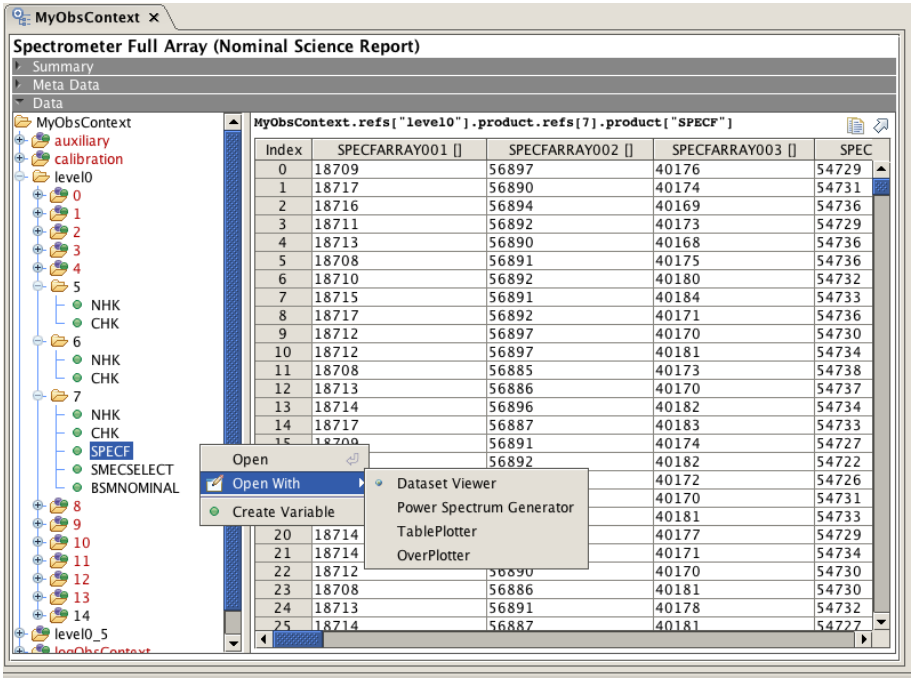


**Figure 2.58. The Level 0 Raw Data within the Observation Context**

# Chapter 3. SPIRE Calibration Data

## 3.1. SPIRE Calibration Explained

### 3.1.1. The SPIRE Calibration Context

Calibration data is attached to the Observation Context for every observation. This section describes how to access, understand and update (if necessary) the calibration data. The calibration context which contains all of the SPIRE calibration products for both Photometer and Spectrometer can be extracted from the Observation Context as follows (where the observation context has alraedy been read into a variable called `obs`):

```
cal = obs.calibration
```

The view when this is visualised in the Observation or Context Viewer is shown in Figure 3.1. This viewer shows that there are two sub-contexts – one for Photometer and one for the Spectrometer, as well as some products that are common and so listed separately. The individual calibration products are contained within the “phot” and “spec” calibration contexts.

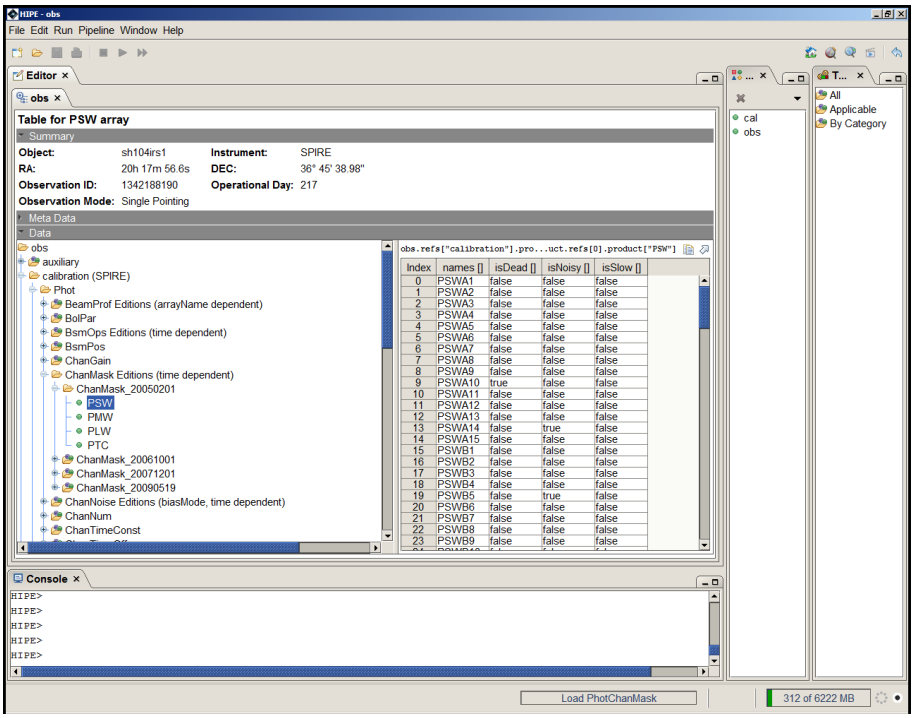


**Figure 3.1. The SPIRE calibration context.**

### 3.1.2. The SPIRE Calibration Tree

The calibration of the SPIRE instrument is likely to be improved throughout the mission and beyond as we gain better understanding of the instrument performance. The collection of all calibration products for SPIRE are referred to as the “Calibration Tree”, and as this is updated, the calibration tree number changes. The version of the calibration tree is contained within the metadata of the calibration context, for example:

```
print obs.calibration.version
spire_cal_4_0
```

Calibration trees are often (but not always) related to a particular version of Hipe.

## 3.1.3. SPIRE Calibration Product Editions

Several calibration products have different contents depending on the conditions of the observation (for example, the values may change at different times, or may depend on whether “bright” or “nominal” mode was used, etc.). These are referred to as “editions”. The Calibration Context Viewer lists the dependency of the editions next to each calibration product, and gives access to all of the different editions (shown as an example in Figure 3.2).

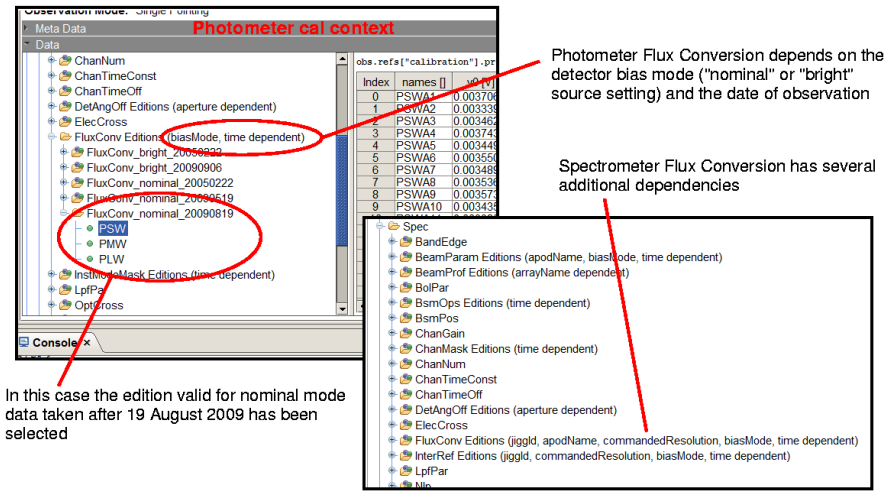


**Figure 3.2. SPIRE calibration editions.**

In a script, in simple cases (such as time dependency), the Observation Context can select the correct edition automatically. In other cases, the variables upon which the product depends must be supplied to get the correct product from a List. Some examples of accessing individual products in a script from the phot and spec contexts are: Spectrometer band edges product, which has no dependency,

```
bandEdge = obs.calibration.spec.bandEdge
```

Photometer channel mask product (details which detectors are defined as dead, or noisy) – the correct time dependent edition for this observation is selected automatically,

```
chanMask = obs.calibration.phot.chanMask
```

Flux Conversion products are selected automatically for the Photometer (dependency is on whether nominal or bright mode was used and the observing date), but for the Spectrometer, where the product also depends on jiggle position, spectral resolution and apodization function, the correct product must be selected from a List,

```
photFluxConv = obs.calibration.phot.fluxConv
specFluxConv =
obs.calibration.spec.fluxConvList.getProduct(6, -"HR", -"unapod", -"nominal",
obs.startDate)
```

When the SPIRE calibration products are saved in FITS file format, the naming convention for the individual product edition files is derived from “SCal” (for SPIRE Calibration), plus “Phot” or “Spec” (for Photometer of Spectrometer), the name of the product, the dependencies (if there are any), and the version number of that particular edition. For example:

```
SCalPhotBolPar_v3.fits
SCalSpecInterRef_12_CR_nominal_20050222_v1.fits
```

Time dependency is specified in the file name by the start date at which the edition becomes valid.

## 3.1.4. Updating a Calibration Tree

When an observation is processed by the HSC and placed into the Herschel Science Archive, it has the particular calibration tree of the time attached (and used in the automatic pipeline). It is possible to update this calibration tree that is attached to the observation, either to a more recent version, or to a previous version (e.g. to determine the effect of an update in calibration products). The latest calibration tree can be downloaded from the HSA directly from within Hipe, using,

```
calNew = spireCal(calTree="spire_cal_4_0")
```

This will pop-up a dialog box asking for user login and password for the HSA. Alternatively, the calibration tree can be loaded from a .jar file (if you have one) that has been saved on the local disk,

```
calNew = spireCal(jarFile="spire_cal_4_0.jar")
```

Once the updated calibration tree has been downloaded, it can be added to the Observation Context to replace the existing tree using,

```
obs.calibration.update(calNew)
```

To save this change, the Observation Context would then need to be written out to a pool on the local disk.

## 3.1.5. Updating Individual Calibration Products

The tasks in the pipeline take individual calibration products as input. This means that any individual calibration product can be supplied directly to the task if an updated test version is available. The name/filename of calibration product used is recorded in the processing history of the data.

## 3.1.6. Removing Calibration Products from the Tree

It is possible to remove some calibration products from the Calibration Context if it is taking up too much disk space. For example, the Spectrometer calibration context is quite large - if only the Photometer calibration products are needed, the Spectrometer part of the Calibration Context can be removed using:

```
obs.calibration.spec.refs.clear()
```

The modified calibration tree could then be written back to the disk (if desired) as a new pool,

```
poolName = -"spire_cal_4_0_phot"
store = ProductStorage(poolName)
store.save(obs.calibration)
```

## 3.1.7. Further Information

Further details of (expert) methods to control or manipulate the calibration tree can be found in the SPIRE Developer's Reference Manual API documentation (javadoc) in the entry listed under:

```
herschel.spire.ia.cal.SpireCal
```

# Chapter 4. Reprocessing your data

## 4.1. Introduction

Now that you have inspected your data products, you may feel that you would like to reprocess your data from the Level 0.5 products onwards, and in time to diverge away from the standard pipeline processing provided by the HSC. This section provides an overview of the steps required to process your datasets from Level 0.5 onwards, and on how to inspect your final Level 1 and Level 2 products.

## 4.2. Reprocessing SPIRE Large Map and Parallel Mode Data

### 4.2.1. Prerequites

The Large Map mode is essentially the same as the SPIRE component of the Parallel Mode - for both modes, this processing guide will allow you to reprocess your data. For this data reprocessing example, we assume that you wish to reprocess your data starting from Level 0.5 products. For this data reprocessing example, we will be using the Large Map observation (obsID: 1342183475) of NGC 5315. We will in this example assume that you have received the engineering pipeline processed Level 0.5 data products from the HSC, and have stored them in a storage pool "1342183475_POF5_NGC5315", either by a direct download or through HIPE.

You can access the POF5 pipeline processing script by clicling on 'Pipeline' on the top bar within HIPE, selecting 'SPIRE' and then clicking on 'Photometer Large Map pipeline script (POF5)' - the script will open up in the Editor window within HIPE.

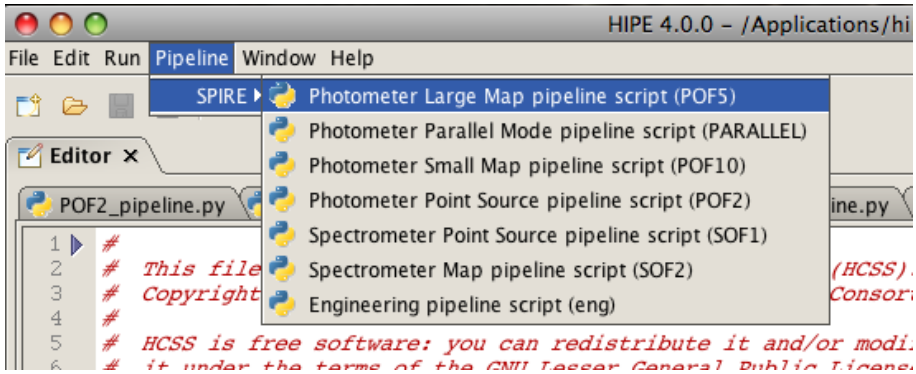


**Figure 4.1. Selecting the POF5 pipeline script**

To start processing, first, we need to make sure that you have imported all needed classes and task definitions required to run the POF5/Large Map pipeline:

```
# Import all needed classes
from herschel.spire.all import *
from herschel.ia.all import *
from herschel.ia.task.mode import *
from herschel.ia.pg import ProductSink
from java.lang import *
from java.util import *
from herschel.ia.obs.util import ObsParameter
```

```
from herschel.ia.pal.pool.lstore.util import TemporalPool

# Import the script tasks.py that contains the task definitions
from herschel.spire.ia.pipeline.scripts.POF5.POF5_tasks import *

# Input definition
from herschel.spire.ia.pipeline.scripts.POF5.POF5_input import *

# Import the script obsLoader.py that allows to load an ObservationContext from a
storage.
from herschel.spire.ia.scripts.tools.obsLoader import *
```

We must search our local pool "1342183475_POF5_NGC5315" for our observation context. We will run the ObsLoader pop-up window and input the ObsID and the name of the local pool to load the observation context, and open an pop up dialog box to take inputs such as if we wish to look at plots of intermediately processed pipeline products, the type of map-making (naive or MadMap) and which point you wisg to start processing from (e.g. Level 0):

```
# Open the input dialog to enter inputs
inputs.openDialog()

# Open a dialog to load the ObservationContext if -"obs" is not defined.
try:
 obsid=obs.obsid
except NameError:
 loader=ObsLoader()
 obs=loader.getObs().product
pass
```

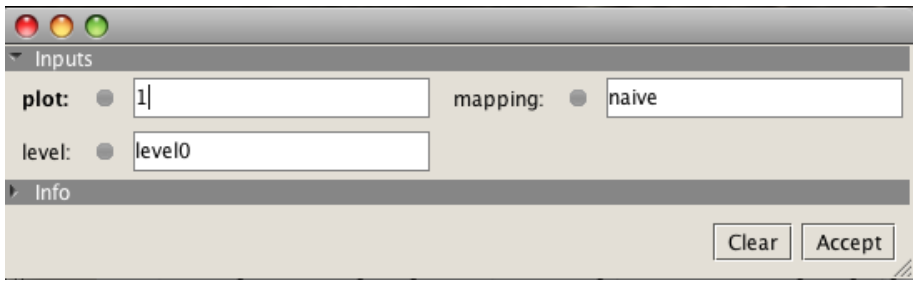


**Figure 4.2. Setting parameters for processing**

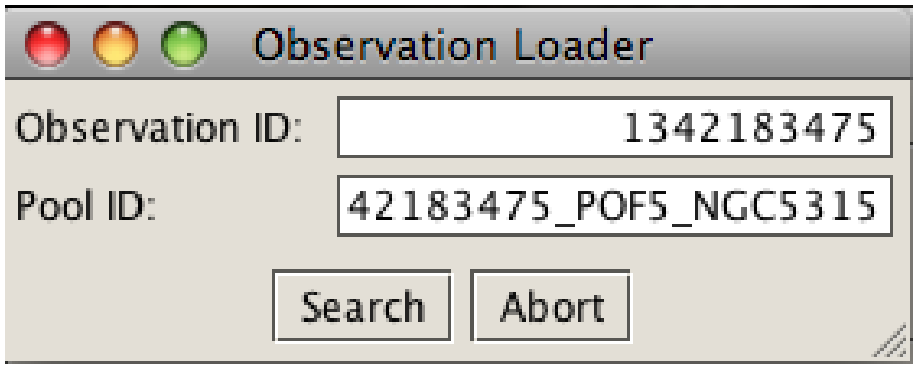


**Figure 4.3. Using ObsLoader to load the observation**

The pipeline also includes a check that the data really is SPIRE data, by raising a BadDataException if the data isn't:

```
# Check that the data are really SPIRE data
if obs.instrument -!= -"SPIRE":
    raise BadDataException("This ObservationContext cannot be processed with this
pipeline: it contains -"+obs.instrument+" data, not SPIRE data")
```

Next, we shall create a creator variable to store the relevant origin metadata for the Level and Level 2 contexts, a logger to follow the progress of the pipeline's execution, set up the time origin for any output plots and then finally, extract the ObsId of our observation and the calibration and auxillary products required for processing the POF10 pipeline:

```
# this is used to put in the creator metadata of level 1 and level 2 context the
version of SPG or of the pipeline
# that was executed
creator=herschel.share.util.Configuration.getProperty("hcss.ia.dataset.creator", -"$Revision:
1.68 $")

#create a logger for the pipeline
logger=TaskModeManager.getMode().getLogger()

# Shift of time origin for plots
t0=obs.startDate.microsecondsSince1958()*1e-6
obsid=obs.obsid
print -"processing OBSID=",obsid,"("+hex(obsid)+")"

# Extract from the observation context the calibration products that
# will be used in the script
bsmPos=obs.calibration.phot.bsmPos
bsmOps=obs.calibration.phot.bsmOps
detAngOff=obs.calibration.phot.detAngOff
elecCross=obs.calibration.phot.elecCross
optCross=obs.calibration.phot.optCross

# Extract from the observation context the auxiliary products that
# will be used in the script
hpp=obs.auxiliary.pointing
siam=obs.auxiliary.siam
```

We set up the Product Sink to perform our processing instead of simply using only memory and then we initialise it:

```
# Set this to FALSE if you don't want to use the ProductSink
# and do all the processing in memory
tempStorage=Boolean.TRUE

# Initialize the ProductSink with a TemporalPool that will be removed when the
# HIPE session is closed, in case of interactive mode.
# The TemporalPool is created in a directory starting from the path defined by the
# var.hcss.workdir property. If this directory is inaccessible or not convenient,
please
# change this property to a proper value.
if TaskModeManager.getType().toString() == -"INTERACTIVE" and tempStorage:
    pname="tmp"+hex(System.currentTimeMillis())[2:-1]
    tmppool=TemporalPool.createTmpPool(pname,TemporalPool.CloseMode.DELETE_ON_CLOSE)
    ProductSink.getInstance().productStorage=ProductStorage(tmppool)
pass
```

# 4.2.2. Level 0 to Level 0.5 Processing (Optional)

If you do not have Level 0.5 products to hand, you will need to make the engineering conversion first from the raw Level 0 products - basically, we are converting the raw telemetry in the form of products into engineering units such as bolometer voltages and resistances timelines. We can run the engineering conversion pipeline from the Level 0 products obtained from the HSA to obtain our Level 0.5 products using:

```
# From Level 0 to Level 0.5
if inputs.level=="level0":
 # Make Engineering conversion of level 0 products
 level0_5= engConversion(obs.level0,cal=obs.calibration, tempStorage=tempStorage)
```

```
 # Add the result to the observation in level 0.5
 obs.level0_5=level0_5
else:
 level0_5=obs.level0_5
pass
```

## 4.2.3. Level 0.5 to Level 1 Processing

Now, we can process our data from Level 0.5 to Level 1. Looping over the scan lines to start building up the map, we take the engineering products to calculate the BSM angles and the SPIRE pointing product. We then perform a number of corrections to the data, after which we will have produced the Level 1 pipeline data product. The pipeline for Level 0.5 to Level 1 processing involves the following sequence of processing modules. The pipeline works on a Photometer Detector Timeline (PDT) and requires the Nominal Housekeeping Timeline (NHKT). Additional auxilliary products are required for the telescope pointing information. The figure below outlines the steps required to process the Small Map pipeline.

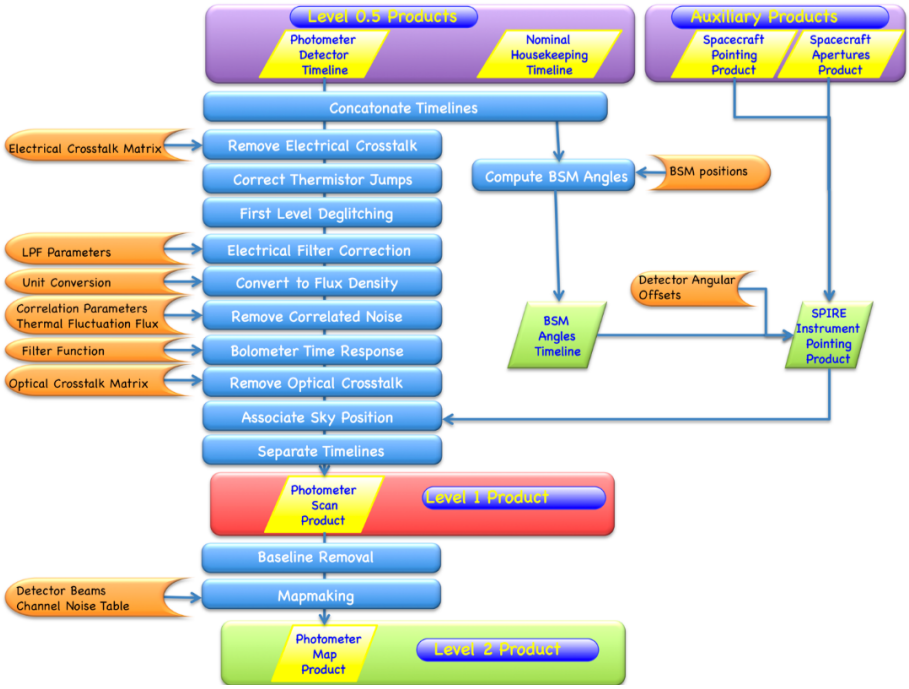


**Figure 4.4. The SPIRE POF5 Photometer Large Map pipeline.**

In order to execute these steps in the most efficient manner possible, we execute a number of pipeline tasks within a single loop. A simplified vesion of this loop, adapted from the POF10 pipeline script, is given below:

```
if inputs.level=="level0" or inputs.level=="level0_5":
 # Create Level1 context
 level1=Level1Context(obsid)
 for key in level0_5.meta.keySet():
  if key -!= -"creator" and (not key.endswith("Date")) and key -!= -"fileName" and \
  key -!= -"type" and key -!= -"description":
   level1.meta[key]=level0_5.meta[key].copy()
 level1.creator=creator
 bbids=level0_5.getBbids(0xa103)
 nlines=len(bbids)
 print -"number of scan lines:",nlines
 #
 # Loop over scan lines
 for bbid in bbids:
  block=level0_5.get(bbid)
```

```
  print -"processing BBID="+hex(bbid)
  # Now move to engineering data products
  pdt  = block.pdt
  nhkt = block.nhkt

  #
  # access and attach turnaround data to the nominal scan line
  bbCount=bbid & 0xFFFF
  pdtLead=None
  nhktLead=None
  pdtTrail=None
  nhktTrail=None
  if bbCount >1:
   blockLead=level0_5.get(0xaf000000L+bbCount-1)
   pdtLead=blockLead.pdt
   nhktLead=blockLead.nhkt
   if pdtLead -!= None and pdtLead.sampleTime[-1] < pdt.sampleTime[0]-3.0:
    pdtLead=None
    nhktLead=None
  if bbid < MAX(Long1d(bbids)):
   blockTrail=level0_5.get(0xaf000000L+bbCount)
   pdtTrail=blockTrail.pdt
   nhktTrail=blockTrail.nhkt
   if pdtTrail -!= None and pdtTrail.sampleTime[0] > pdt.sampleTime[-1]+3.0:
    pdtTrail=None
    nhktTrail=None
  pdt=joinPhotDetTimelines(pdt,pdtLead,pdtTrail)
  nhkt=joinNhkTimelines(nhkt,nhktLead,nhktTrail)
  #
  # calculate BSM angles
  bat=calcBsmAngles(nhkt,bsmPos=bsmPos)
  #
  # create the SpirePointingProduct
  spp=createSpirePointing(detAngOff=detAngOff,bat=bat,hpp=hpp,siam=siam)
  #
  # run electrical crosstalk correction
  pdt=elecCrossCorrection(pdt,elecCross=elecCross)
  #
  # run the deglitch
  pdt=waveletDeglitcher(pdt, scaleMin=1.0, scaleMax=8.0, scaleInterval=5,
holderMin=-1.9,\
   holderMax=-0.3, correlationThreshold=0.69)
  #
  # run electrical Low Pass Filter response correction
  pdt=lpfResponseCorrection(pdt,lpfPar=lpfPar)
  #
  # run the flux conversion
  fluxConv=fluxConvList.getProduct(pdt.meta["biasMode"].value,pdt.startDate)
  pdt=photFluxConversion(pdt,fluxConv=fluxConv)
  #
  # run the temeperature drift correction

 tempDriftCorr=tempDriftCorrList.getProduct(pdt.meta["biasMode"].value,pdt.startDate)
  pdt=temperatureDriftCorrection(pdt,tempDriftCorr=tempDriftCorr)
  #
  # run bolometer time response correction
  pdt=bolometerResponseCorrection(pdt,chanTimeConst=chanTimeConst)
  #
  # run optical crosstalk correction
  pdt=photOptCrossCorrection(pdt,optCross=optCross)
  #
  # add pointing
  psp=associateSkyPosition(pdt,spp=spp)
  #
  # cut the timeline back to scan line range.
  # If you want include turnaround data in map making, call the following
  # task with the option -"extend=True"
  psp=cutPhotDetTimelines(psp)

  # Store Photometer Scan Product in Level 1 product storage
  if tempStorage:
   ref=ProductSink.getInstance().save(psp)
```

```
   level1.addRef(ref)
  else:
   level1.addProduct(psp)
  #
  print -"Completed BBID=0x%x (%i/%i)"%(bbid,count+1,nlines)
  # set the progress
  count=count+1
  inputs.progress = 20+(60*count)/nlines
 #
 if level1.count == 0:
  logger.severe("No scan line processed due to missing data. This observation
CANNOT be processed!")
  print -"No scan line processed due to missing data. This observation CANNOT be
processed!"
  raise MissingDataException("No scan line processed due to missing data. This
observation CANNOT be processed!")
 #
 obs.level1=level1
 # promote to LEVEL1_PROCESSED
 obs.obsState = ObservationContext.OBS_STATE_LEVEL1_PROCESSED
else:
 level1=obs.level1
pass
```

To break this up into its constituent parts, first of all, we create the Level 1 context:

```
 # Create Level1 context
 level1=Level1Context(obsid)
 for key in level0_5.meta.keySet():
  if key -!= -"creator" and (not key.endswith("Date")) and key -!= -"fileName" and \
  key -!= -"type" and key -!= -"description":
   level1.meta[key]=level0_5.meta[key].copy()
 level1.creator=creator
 bbids=level0_5.getBbids(0xa103)
 nlines=len(bbids)
 print -"number of scan lines:",nlines
 #
```

We loop over the scan lines and attach the engineering data to the scan lines:

```
 # Loop over scan lines
 for bbid in bbids:
  block=level0_5.get(bbid)
  print -"processing BBID="+hex(bbid)
  # Now move to engineering data products
  pdt  = block.pdt
  nhkt = block.nhkt

  # access and attach turnaround data to the nominal scan line
  bbCount=bbid & 0xFFFF
  pdtLead=None
  nhktLead=None
  pdtTrail=None
  nhktTrail=None
  if bbCount >1:
   blockLead=level0_5.get(0xaf000000L+bbCount-1)
   pdtLead=blockLead.pdt
   nhktLead=blockLead.nhkt
   if pdtLead -!= None and pdtLead.sampleTime[-1] < pdt.sampleTime[0]-3.0:
    pdtLead=None
    nhktLead=None
  if bbid < MAX(Long1d(bbids)):
   blockTrail=level0_5.get(0xaf000000L+bbCount)
   pdtTrail=blockTrail.pdt
   nhktTrail=blockTrail.nhkt
   if pdtTrail -!= None and pdtTrail.sampleTime[0] > pdt.sampleTime[-1]+3.0:
    pdtTrail=None
    nhktTrail=None
  pdt=joinPhotDetTimelines(pdt,pdtLead,pdtTrail)
```

```
  nhkt=joinNhkTimelines(nhkt,nhktLead,nhktTrail)
```

Next, we calculate the BSM angles and compute the SPIRE Pointing Product:

```
  # calculate BSM angles
  bat=calcBsmAngles(nhkt,bsmPos=bsmPos)
  #
  # create the SpirePointingProduct
  spp=createSpirePointing(detAngOff=detAngOff,bat=bat,hpp=hpp,siam=siam)
  #
```

We now perform a number of corrections to the data - electrical crosstalk correction, deglitching, electrical Low Pass Filter response correction, flux conversion, temeperature drift correction, bolometer time response correction. Most of these are dependent on the calibation products you imported earlier in the pipeline, so to tweak, you must supply the updated relevant calibration product. The exception is deglitching, where you can edit the input parameters to the module directly. Discussion of the parameters of this module is beyond the scope of this discussion - instead, please inspect the relevant section of the SPIRE Users Manual for a more in-depth treatment of the paarmeters for this module.

```
  # run electrical crosstalk correction
  pdt=elecCrossCorrection(pdt,elecCross=elecCross)
  #
  # run the deglitch
  pdt=waveletDeglitcher(pdt, scaleMin=1.0, scaleMax=8.0, scaleInterval=5,
holderMin=-1.9,\
   holderMax=-0.3, correlationThreshold=0.69)
  #
  # run electrical Low Pass Filter response correction
  pdt=lpfResponseCorrection(pdt,lpfPar=lpfPar)
  #
  # run the flux conversion
  fluxConv=fluxConvList.getProduct(pdt.meta["biasMode"].value,pdt.startDate)
  pdt=photFluxConversion(pdt,fluxConv=fluxConv)
  #
  # run the temeperature drift correction

 tempDriftCorr=tempDriftCorrList.getProduct(pdt.meta["biasMode"].value,pdt.startDate)
  pdt=temperatureDriftCorrection(pdt,tempDriftCorr=tempDriftCorr)
  #
  # run bolometer time response correction
  pdt=bolometerResponseCorrection(pdt,chanTimeConst=chanTimeConst)
  #
  # run optical crosstalk correction
  pdt=photOptCrossCorrection(pdt,optCross=optCross)
```

We then add the pointing product, and cut the timeline back to follow the scan line range:

```
  # add pointing
  psp=associateSkyPosition(pdt,spp=spp)
  #
  # cut the timeline back to scan line range.
  # If you want include turnaround data in map making, call the following
  # task with the option -"extend=True"
  psp=cutPhotDetTimelines(psp)
  #
```

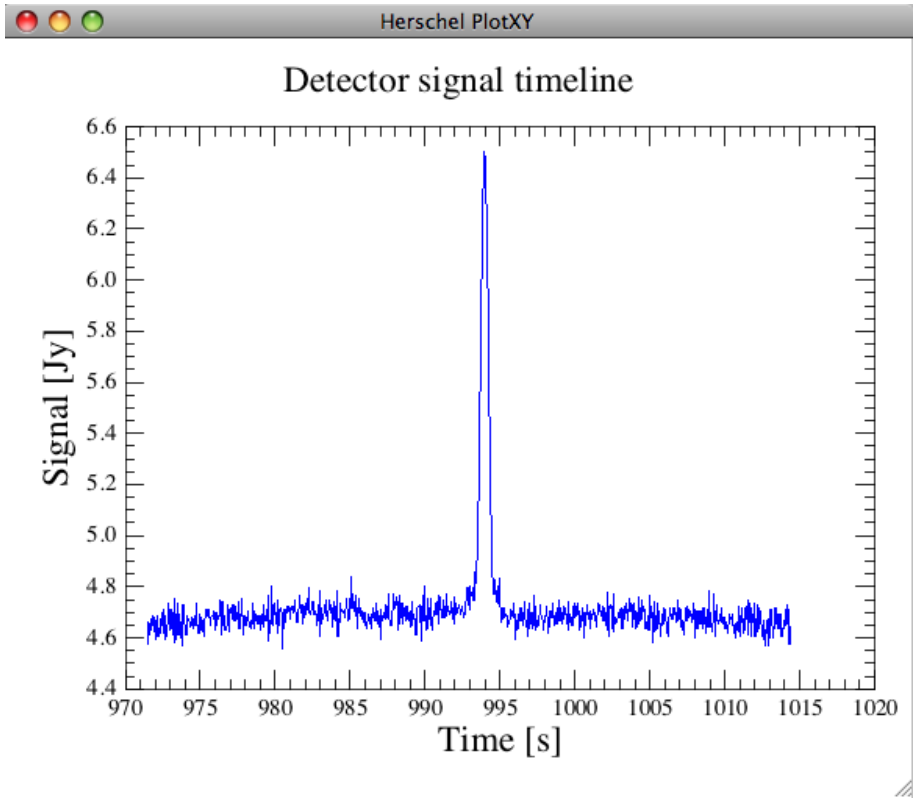


**Figure 4.5. Detector signal timelines**

Finally for this stage of the pipeline, we shall store our product in the Level 1 context:

```
 if tempStorage:
  ref=ProductSink.getInstance().save(psp)
  level1.addRef(ref)
 else:
  level1.addProduct(psp)

 obs.level1=level1
 # promote to LEVEL1_PROCESSED
 obs.obsState = ObservationContext.OBS_STATE_LEVEL1_PROCESSED
```

And that brings us to the end of our Level 1 processing for Small Map.

## 4.2.4. Level 1 to Level 2 Processing

Our next step is to convert our Level 1 photometer product into our final maps, which constitute the Level 2 products for the Small Map pipeline.

```
if inputs.mapping -!= -'none':
 #
 # Flag to switch on and off the baseline removal
 useRemoveBaseline=True
 #
 # Create a SpireListContext to be used as input of map making
 scans=SpireListContext()
 #
 # Run baseline removal and populate the map making input
 for i in range(level1.count):
  if useRemoveBaseline:
   psp=level1.getProduct(i)
   psp=removeBaseline(psp,chanNum=chanNum)
   if tempStorage:
    ref=ProductSink.getInstance().save(psp)
```

```
    scans.addRef(ref)
   else:
    scans.addProduct(psp)
  else:
   scans.addRef(level1.refs[i])
 pass
 #
 # Run mapmaking
 if inputs.mapping == -'naive':
  mapPlw=naiveScanMapper(scans, array="PLW")
  inputs.progress=85
  mapPmw=naiveScanMapper(scans, array="PMW")
  inputs.progress=90
  mapPsw=naiveScanMapper(scans, array="PSW")
 else:

 chanNoise=obs.calibration.phot.chanNoiseList.getProduct(level1.getProduct(0).meta["biasMode"].valu
\
   level1.getProduct(0).startDate)
  mapPlw=madScanMapper(scans, array="PLW",chanNoise=chanNoise)
  inputs.progress=85
  mapPmw=madScanMapper(scans, array="PMW",chanNoise=chanNoise)
  inputs.progress=90
  mapPsw=madScanMapper(scans, array="PSW",chanNoise=chanNoise)
 pass
 #
 # Create a context with level 2 products (maps) and attach it to the observation
context
 level2=MapContext()
 for key in level1.meta.keySet():
  if key -!= -"creator" and key -!= -"creationDate":
   level2.meta[key]=level1.meta[key].copy()
 level2.creator=creator
 level2.type="level2context"
 level2.description="Context for SPIRE Level 2 products"
 level2.meta["level"]=StringParameter("20", -"The level of the product")
 level2.refs.put("PLW",ProductRef(mapPlw))
 level2.refs.put("PMW",ProductRef(mapPmw))
 level2.refs.put("PSW",ProductRef(mapPsw))
 obs.level2=level2
 #
 # promote to LEVEL2_PROCESSED
 obs.obsState = ObservationContext.OBS_STATE_LEVEL2_PROCESSED
 #
 # Create browse product and image
 createRgbImage=CreateRgbImageTask()
 browseProduct=createRgbImage(red=mapPlw,green=mapPmw,blue=mapPsw,percent=98.0,redFactor=1.0,
\
  greenFactor=1.0,blueFactor=1.0)
 #
 # Populate metadata of the browse product
 for par in ObsParameter.values():
  if obs.meta.containsKey(par.key) and par.key -!= -"fileName":
   browseProduct.meta[par.key]=obs.meta[par.key].copy()
 pass
 browseProduct.startDate=obs.startDate
 browseProduct.endDate=obs.endDate
 browseProduct.instrument=obs.instrument
 browseProduct.modelName=obs.modelName
 browseProduct.description="Browse Product"
 browseProduct.type="BROWSE"
 #
 # Attach the browse product to the ObservationContext
 obs.browseProduct=browseProduct
 #
 # Generate the browse image
 from herschel.ia.gui.image import ImageUtil
 imageUtil = ImageUtil()
 browseProductImage=imageUtil.getRgbTiledImage(\
  browseProduct["red"].data, browseProduct["green"].data,
browseProduct["blue"].data)
 obs.browseProductImage=browseProductImage.asBufferedImage
```

```
pass
```

Assuming that we have requested mapping products, we first of all flag the pipeline to perform baseline subtraction, set up a List Context to be used as input to the map making, perform the baseline subtraction on our Level 1 product and store it in our List Context:

```
# Flag to switch on and off the baseline removal
useRemoveBaseline=True
#
# Create a SpireListContext to be used as input of map making
scans=SpireListContext()
#
# Run baseline removal and populate the map making input
for i in range(level1.count):
 if useRemoveBaseline:
  psp=level1.getProduct(i)
  psp=removeBaseline(psp,chanNum=chanNum)
  if tempStorage:
   ref=ProductSink.getInstance().save(psp)
   scans.addRef(ref)
  else:
   scans.addProduct(psp)
 else:
  scans.addRef(level1.refs[i])
pass
```

We use this as input for the map maker. The type of map produced is dependent on your input at the very start of the pipeline - naive or MadMap.

```
# Run mapmaking
if inputs.mapping == -'naive':
 mapPlw=naiveScanMapper(scans, array="PLW")
 inputs.progress=85
 mapPmw=naiveScanMapper(scans, array="PMW")
 inputs.progress=90
 mapPsw=naiveScanMapper(scans, array="PSW")
else:

 chanNoise=obs.calibration.phot.chanNoiseList.getProduct(level1.getProduct(0).meta["biasMode"].valu
\
   level1.getProduct(0).startDate)
 mapPlw=madScanMapper(scans, array="PLW",chanNoise=chanNoise)
 inputs.progress=85
 mapPmw=madScanMapper(scans, array="PMW",chanNoise=chanNoise)
 inputs.progress=90
 mapPsw=madScanMapper(scans, array="PSW",chanNoise=chanNoise)
pass
```

Three maps are each produced for PSW, PMW and PLW, and are visible through the Product Viewer by right-clicking on the required variable in the Variable pane and selecting 'Open With':

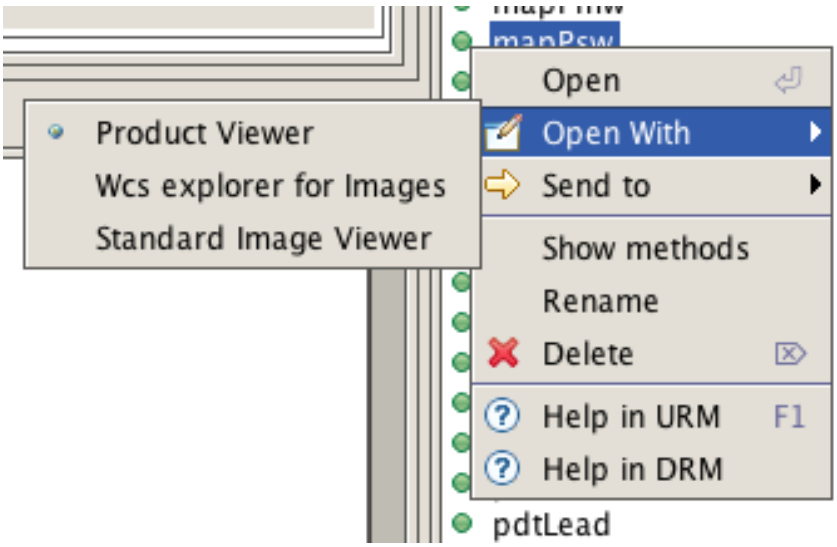


**Figure 4.6. Selecting the Product Viewer**

the actual map with fluxes (denoted as 'image');

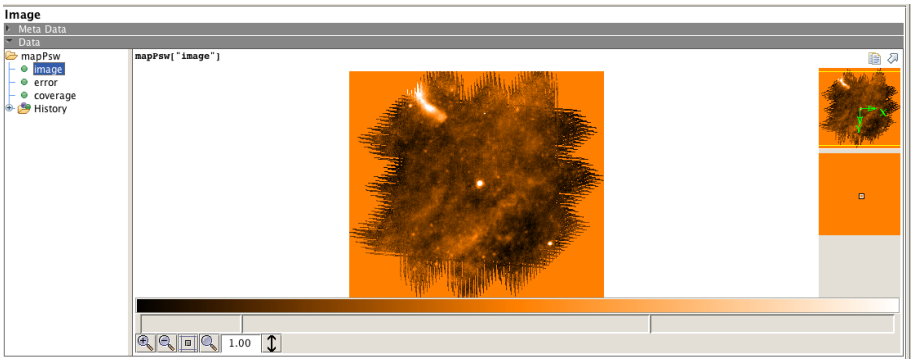


**Figure 4.7. Setting parameters for processing**

the statistical flux error map (denoted as 'error'):

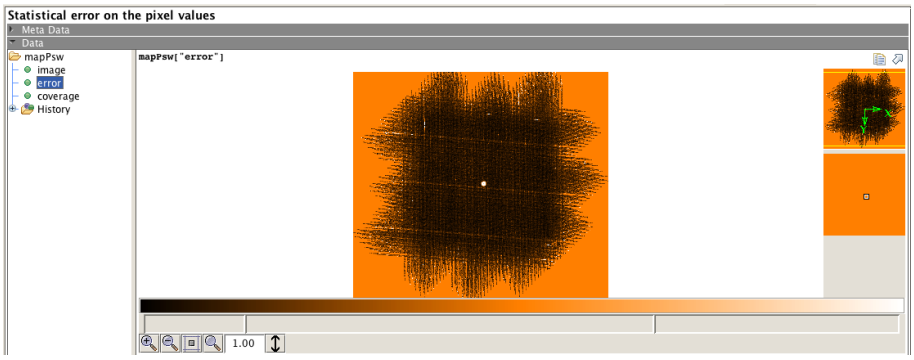


**Figure 4.8. Setting parameters for processing**

and an image which shows the coverage map for our scans (denoted as 'coverage'):

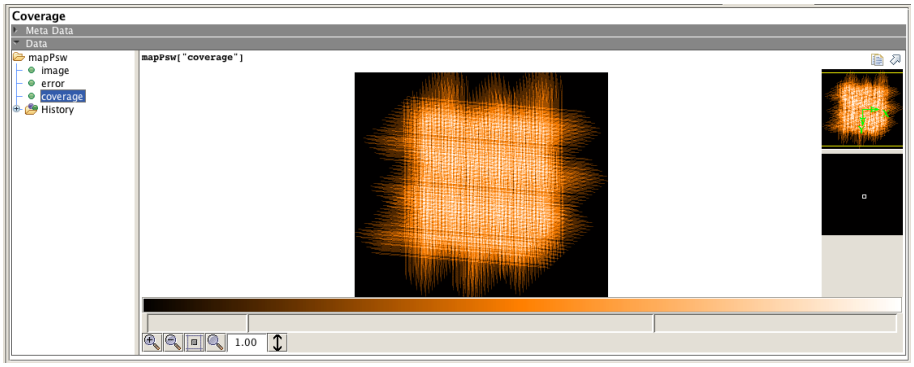


**Figure 4.9. Setting parameters for processing**

We can export our images to FITS files by right clicking on the respective variable in the Variable pane (e.g. mapPsw), and selecting 'Send To -> FITS file'.

We finally scan create a context to store our maps as Level 2 products, and attach them to the observation context.

```
 # Create a context with level 2 products (maps) and attach it to the observation
context
 level2=MapContext()
 for key in level1.meta.keySet():
  if key -!= -"creator" and key -!= -"creationDate":
   level2.meta[key]=level1.meta[key].copy()
 level2.creator=creator
 level2.type="level2context"
 level2.description="Context for SPIRE Level 2 products"
 level2.meta["level"]=StringParameter("20", -"The level of the product")
 level2.refs.put("PLW",ProductRef(mapPlw))
 level2.refs.put("PMW",ProductRef(mapPmw))
 level2.refs.put("PSW",ProductRef(mapPsw))
 obs.level2=level2
 #
 # promote to LEVEL2_PROCESSED
 obs.obsState = ObservationContext.OBS_STATE_LEVEL2_PROCESSED
```

Congratulations! You have now re-processed your Small Map data all the way to the final Level 2 maps!

---

# 4.3. Reprocessing SPIRE Small Map Data

## 4.3.1. Prerequites

For this data reprocessing example, we will be using the Small Map observation (obsID: 1342195871) of the star Gamma Draconis. We will in this example assume that you have received the engineering pipeline processed Level 0.5 data products from the HSC, and have stored them in a storage pool "1342195871_POF10_GammaDra", either by a direct download or through HIPE.

You can access the POF10 pipeline processing script by clicling on 'Pipeline' on the top bar within HIPE, selecting 'SPIRE' and then clicking on 'Photometer Small Map pipeline script (POF10)' - the script will open up in the Editor window within HIPE.

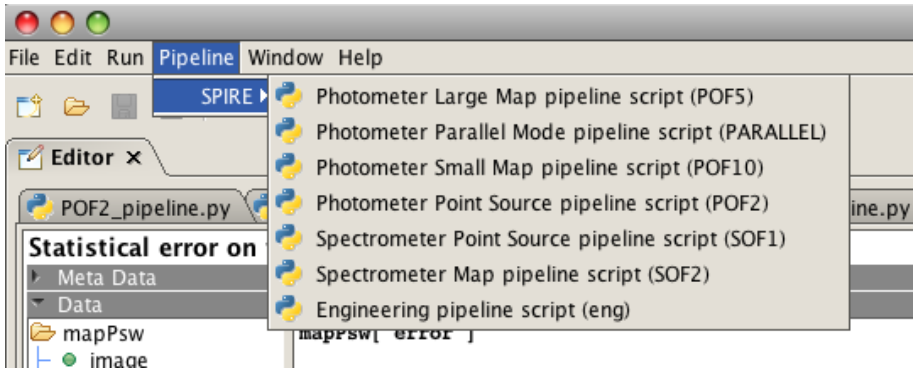


**Figure 4.10. Selecting the POF10 pipeline script**

To start processing, first, we need to make sure that you have imported all needed classes and task definitions required to run the POF2/point source pipeline:

```
# Import all needed classes
from herschel.spire.all import *
from herschel.ia.all import *
from herschel.ia.task.mode import *
from herschel.ia.pg import ProductSink
from java.lang import *
from java.util import *
from herschel.ia.obs.util import ObsParameter
from herschel.ia.pal.pool.lstore.util import TemporalPool

# Import the script tasks.py that contains the task definitions
from herschel.spire.ia.pipeline.scripts.POF10.POF10_tasks import *

# Input definition
from herschel.spire.ia.pipeline.scripts.POF10.POF10_input import *

# Import the script obsLoader.py that allows to load an ObservationContext from a
storage.
from herschel.spire.ia.scripts.tools.obsLoader import *
```

We must search our local pool "1342195871_POF10_GammaDra" for our observation context. We will run the ObsLoader pop-up window and input the ObsID and the name of the local pool to load the observation context, and open an pop up dialog box to take inputs such as if we wish to look at plots of intermediately processed pipeline products, the type of map-making (naive or MadMap) and which point you wisg to start processing from (e.g. Level 0):

```
# Open the input dialog to enter inputs
inputs.openDialog()

# Open a dialog to load the ObservationContext if -"obs" is not defined.
try:
 obsid=obs.obsid
except NameError:
 loader=ObsLoader()
 obs=loader.getObs().product
pass
```

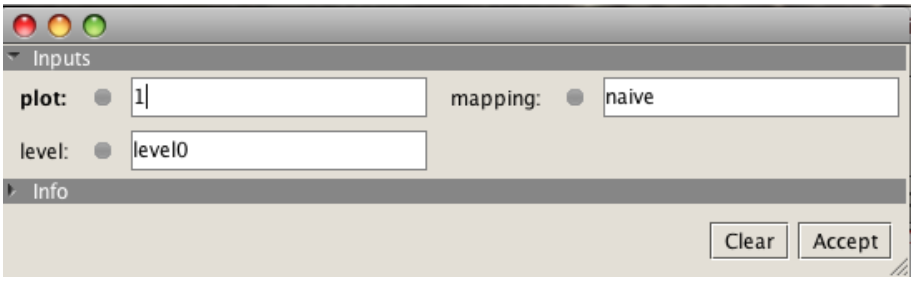


**Figure 4.11. Setting parameters for processing**

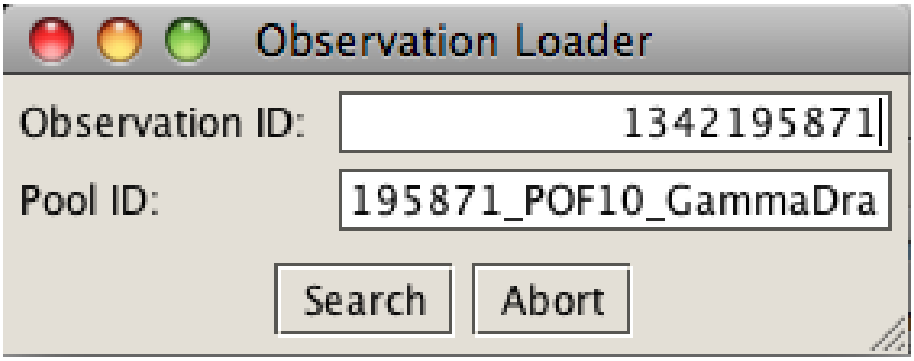


**Figure 4.12. Using ObsLoader to load the observation**

The pipeline also includes a check that the data really is SPIRE data, by raising a BadDataException if the data isn't:

```
# Check that the data are really SPIRE data
if obs.instrument -!= -"SPIRE":
    raise BadDataException("This ObservationContext cannot be processed with this
pipeline: it contains -"+obs.instrument+" data, not SPIRE data")
```

We set up the Product Sink to perform our processing instead of simply using only memory and then we initialise it:

```
# Set this to FALSE if you don't want to use the ProductSink
# and do all the processing in memory
tempStorage=Boolean.TRUE

# Initialize the ProductSink with a TemporalPool that will be removed when the
# HIPE session is closed, in case of interactive mode.
# The TemporalPool is created in a directory starting from the path defined by the
# var.hcss.workdir property. If this directory is inaccessible or not convenient,
please
# change this property to a proper value.
if TaskModeManager.getType().toString() == -"INTERACTIVE" and tempStorage:
    pname="tmp"+hex(System.currentTimeMillis())[2:-1]
    tmppool=TemporalPool.createTmpPool(pname,TemporalPool.CloseMode.DELETE_ON_CLOSE)
    ProductSink.getInstance().productStorage=ProductStorage(tmppool)
pass
```

Next, we shall create a creator variable to store the relevant origin metadata for the Level and Level 2 contexts, a logger to follow the progress of the pipeline's execution, set up the time origin for any output plots and then finally, extract the ObsId of our observation and the calibration and auxillary products required for processing the POF10 pipeline:

```
# this is used to put in the creator metadata of level 1 and level 2 context the
version of SPG or of the pipeline
# that was executed
creator=herschel.share.util.Configuration.getProperty("hcss.ia.dataset.creator", -"$Revision:
1.2.2.2 $")

#create a logger for the pipeline
logger=TaskModeManager.getMode().getLogger()

# Shift of time origin for plots
t0=obs.startDate.microsecondsSince1958()*1e-6
obsid=obs.obsid
print -"processing OBSID=",obsid,"("+hex(obsid)+")"

# Extract from the observation context the calibration products that
# will be used in the script
bsmPos=obs.calibration.phot.bsmPos
bsmOps=obs.calibration.phot.bsmOps
detAngOff=obs.calibration.phot.detAngOff
elecCross=obs.calibration.phot.elecCross
optCross=obs.calibration.phot.optCross

# Extract from the observation context the auxiliary products that
# will be used in the script
hpp=obs.auxiliary.pointing
siam=obs.auxiliary.siam
```

## 4.3.2. Level 0 to Level 0.5 Processing (Optional)

If you do not have Level 0.5 products to hand, you will need to make the engineering conversion first from the raw Level 0 products - basically, we are converting the raw telemetry in the form of products into engineering units such as bolometer voltages and resistances timelines. We can run the engineering conversion pipeline from the Level 0 products obtained from the HSA to obtain our Level 0.5 products using:

```
# From Level 0 to Level 0.5
if inputs.level=="level0":
 # Make Engineering conversion of level 0 products
 level0_5= engConversion(obs.level0,cal=obs.calibration, tempStorage=tempStorage)
 # Add the result to the observation in level 0.5
 obs.level0_5=level0_5
else:
 level0_5=obs.level0_5
pass

# set the progress
inputs.progress=20
# counter for computing progress
count=0
```

## 4.3.3. Level 0.5 to Level 1 Processing

Now, we can process our data from Level 0.5 to Level 1. Looping over the scan lines to start building up the map, we take the engineering products to calculate the BSM angles and the SPIRE pointing product. We then perform a number of corrections to the data, after which we will have produced the Level 1 pipeline data product. The pipeline for Level 0.5 to Level 1 processing involves the following sequence of processing modules. The pipeline works on a Photometer Detector Timeline (PDT) and requires the Nominal Housekeeping Timeline (NHKT). Additional auxilliary products are required for the telescope pointing information. The figure below outlines the steps required to process the Small Map pipeline.

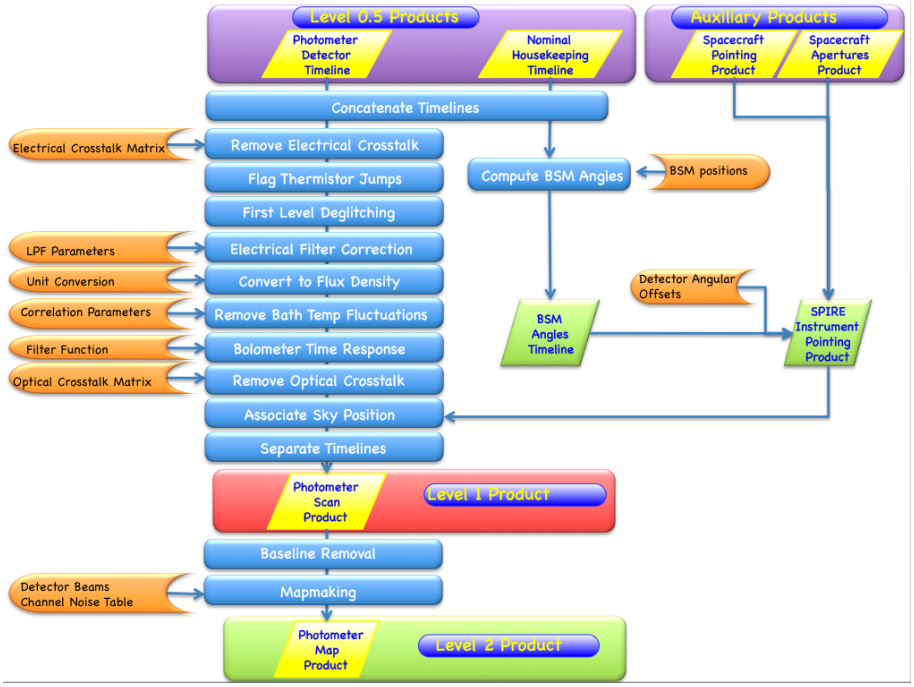


**Figure 4.13. The SPIRE POF10 Photometer Small Map pipeline.**

In order to execute these steps in the most efficient manner possible, we execute a number of pipeline tasks within a single loop. A simplified vesion of this loop, adapted from the POF10 pipeline script, is given below:

```
# From Level 0.5 to Level 1
if inputs.level=="level0" or inputs.level=="level0_5":
 # Create Level1 context
 level1=Level1Context(obsid)
 for key in level0_5.meta.keySet():
  if key -!= -"creator" and (not key.endswith("Date")) and key -!= -"fileName" and \
  key -!= -"type" and key -!= -"description":
   level1.meta[key]=level0_5.meta[key].copy()
 level1.creator=creator
 bbids=level0_5.getBbids(0xa103)
 nlines=len(bbids)
 print -"number of scan lines:",nlines
 #
 # Loop over scan lines
 for bbid in bbids:
  block=level0_5.get(bbid)
  print -"processing BBID="+hex(bbid)
  # Now move to engineering data products
  pdt  = block.pdt
  nhkt = block.nhkt
  #
  # access and attach turnaround data to the nominal scan line
  bbCount=bbid & 0xFFFF
  pdtLead=None
  nhktLead=None
  pdtTrail=None
  nhktTrail=None
  if bbCount >1:
   blockLead=level0_5.get(0xaf000000L+bbCount-1)
   pdtLead=blockLead.pdt
   nhktLead=blockLead.nhkt
   if pdtLead -!= None and pdtLead.sampleTime[-1] < pdt.sampleTime[0]-3.0:
    pdtLead=None
    nhktLead=None
  if bbid < MAX(Long1d(bbids)):
   blockTrail=level0_5.get(0xaf000000L+bbCount)
   pdtTrail=blockTrail.pdt
```

```
   nhktTrail=blockTrail.nhkt
   if pdtTrail -!= None and pdtTrail.sampleTime[0] > pdt.sampleTime[-1]+3.0:
    pdtTrail=None
    nhktTrail=None
  pdt=joinPhotDetTimelines(pdt,pdtLead,pdtTrail)
  nhkt=joinNhkTimelines(nhkt,nhktLead,nhktTrail)
  #
  # calculate BSM angles
  bat=calcBsmAngles(nhkt,bsmPos=bsmPos)
  #
  # create the SpirePointingProduct
  spp=createSpirePointing(detAngOff=detAngOff,bat=bat,hpp=hpp,siam=siam)
  #
  # run electrical crosstalk correction
  pdt=elecCrossCorrection(pdt,elecCross=elecCross)
  #
  # run the deglitch
  pdt=waveletDeglitcher(pdt, scaleMin=1.0, scaleMax=8.0, scaleInterval=5,
holderMin=-1.9,\
   holderMax=-0.3, correlationThreshold=0.69)
  #
  # run electrical Low Pass Filter response correction
  pdt=lpfResponseCorrection(pdt,lpfPar=lpfPar)
  #
  # run the flux conversion
  fluxConv=fluxConvList.getProduct(pdt.meta["biasMode"].value,pdt.startDate)
  pdt=photFluxConversion(pdt,fluxConv=fluxConv)
  #
  # run the temeperature drift correction

 tempDriftCorr=tempDriftCorrList.getProduct(pdt.meta["biasMode"].value,pdt.startDate)
  pdt=temperatureDriftCorrection(pdt,tempDriftCorr=tempDriftCorr)
  #
  # run bolometer time response correction
  pdt=bolometerResponseCorrection(pdt,chanTimeConst=chanTimeConst)
  #
  # run optical crosstalk correction
  pdt=photOptCrossCorrection(pdt,optCross=optCross)
  #
  # add pointing
  psp=associateSkyPosition(pdt,spp=spp)
  #
  # cut the timeline back to scan line range.
  # If you want include turnaround data in map making, call the following
  # task with the option -"extend=True"
  psp=cutPhotDetTimelines(psp)

  # Store Photometer Scan Product in Level 1 product storage
  if tempStorage:
   ref=ProductSink.getInstance().save(psp)
   level1.addRef(ref)
  else:
   level1.addProduct(psp)
  #
  print -"Completed BBID=0x%x (%i/%i)"%(bbid,count+1,nlines)
  # set the progress
  count=count+1
  inputs.progress = 20+(60*count)/nlines
  raise MissingDataException("No scan line processed due to missing data. This
observation CANNOT be processed!")
 #
 obs.level1=level1
 # promote to LEVEL1_PROCESSED
 obs.obsState = ObservationContext.OBS_STATE_LEVEL1_PROCESSED
else:
 level1=obs.level1
pass
```

To break this up into its constituent parts, first of all, we create the Level 1 context:

```
 # Create Level1 context
```

```
level1=Level1Context(obsid)
for key in level0_5.meta.keySet():
 if key -!= -"creator" and (not key.endswith("Date")) and key -!= -"fileName" and \
 key -!= -"type" and key -!= -"description":
  level1.meta[key]=level0_5.meta[key].copy()
level1.creator=creator
bbids=level0_5.getBbids(0xa103)
nlines=len(bbids)
print -"number of scan lines:",nlines
#
```

We loop over the scan lines and attach the engineering data to the scan lines:

```
# Loop over scan lines
for bbid in bbids:
 block=level0_5.get(bbid)
 print -"processing BBID="+hex(bbid)
 # Now move to engineering data products
 pdt  = block.pdt
 nhkt = block.nhkt

 # access and attach turnaround data to the nominal scan line
 bbCount=bbid & 0xFFFF
 pdtLead=None
 nhktLead=None
 pdtTrail=None
 nhktTrail=None
 if bbCount >1:
  blockLead=level0_5.get(0xaf000000L+bbCount-1)
  pdtLead=blockLead.pdt
  nhktLead=blockLead.nhkt
  if pdtLead -!= None and pdtLead.sampleTime[-1] < pdt.sampleTime[0]-3.0:
   pdtLead=None
   nhktLead=None
 if bbid < MAX(Long1d(bbids)):
  blockTrail=level0_5.get(0xaf000000L+bbCount)
  pdtTrail=blockTrail.pdt
  nhktTrail=blockTrail.nhkt
  if pdtTrail -!= None and pdtTrail.sampleTime[0] > pdt.sampleTime[-1]+3.0:
   pdtTrail=None
   nhktTrail=None
 pdt=joinPhotDetTimelines(pdt,pdtLead,pdtTrail)
 nhkt=joinNhkTimelines(nhkt,nhktLead,nhktTrail)
```

Next, we calculate the BSM angles and compute the SPIRE Pointing Product:

```
 # calculate BSM angles
 bat=calcBsmAngles(nhkt,bsmPos=bsmPos)
 #
 # create the SpirePointingProduct
 spp=createSpirePointing(detAngOff=detAngOff,bat=bat,hpp=hpp,siam=siam)
 #
```

We now perform a number of corrections to the data - electrical crosstalk correction, deglitching, electrical Low Pass Filter response correction, flux conversion, temeperature drift correction, bolometer time response correction. Most of these are dependent on the calibation products you imported earlier in the pipeline, so to tweak, you must supply the updated relevant calibration product. The exception is deglitching, where you can edit the input parameters to the module directly. Discussion of the parameters of this module is beyond the scope of this discussion - instead, please inspect the relevant section of the SPIRE Users Manual for a more in-depth treatment of the paarmeters for this module.

```
 # run electrical crosstalk correction
 pdt=elecCrossCorrection(pdt,elecCross=elecCross)
 #
 # run the deglitch
```

```
  pdt=waveletDeglitcher(pdt, scaleMin=1.0, scaleMax=8.0, scaleInterval=5,
holderMin=-1.9,\
   holderMax=-0.3, correlationThreshold=0.69)
  #
  # run electrical Low Pass Filter response correction
  pdt=lpfResponseCorrection(pdt,lpfPar=lpfPar)
  #
  # run the flux conversion
  fluxConv=fluxConvList.getProduct(pdt.meta["biasMode"].value,pdt.startDate)
  pdt=photFluxConversion(pdt,fluxConv=fluxConv)
  #
  # run the temeperature drift correction

 tempDriftCorr=tempDriftCorrList.getProduct(pdt.meta["biasMode"].value,pdt.startDate)
  pdt=temperatureDriftCorrection(pdt,tempDriftCorr=tempDriftCorr)
  #
  # run bolometer time response correction
  pdt=bolometerResponseCorrection(pdt,chanTimeConst=chanTimeConst)
  #
  # run optical crosstalk correction
  pdt=photOptCrossCorrection(pdt,optCross=optCross)
```

We then add the pointing product, and cut the timeline back to follow the scan line range:

```
  # add pointing
  psp=associateSkyPosition(pdt,spp=spp)
  #
  # cut the timeline back to scan line range.
  # If you want include turnaround data in map making, call the following
  # task with the option -"extend=True"
  psp=cutPhotDetTimelines(psp)
  #
```

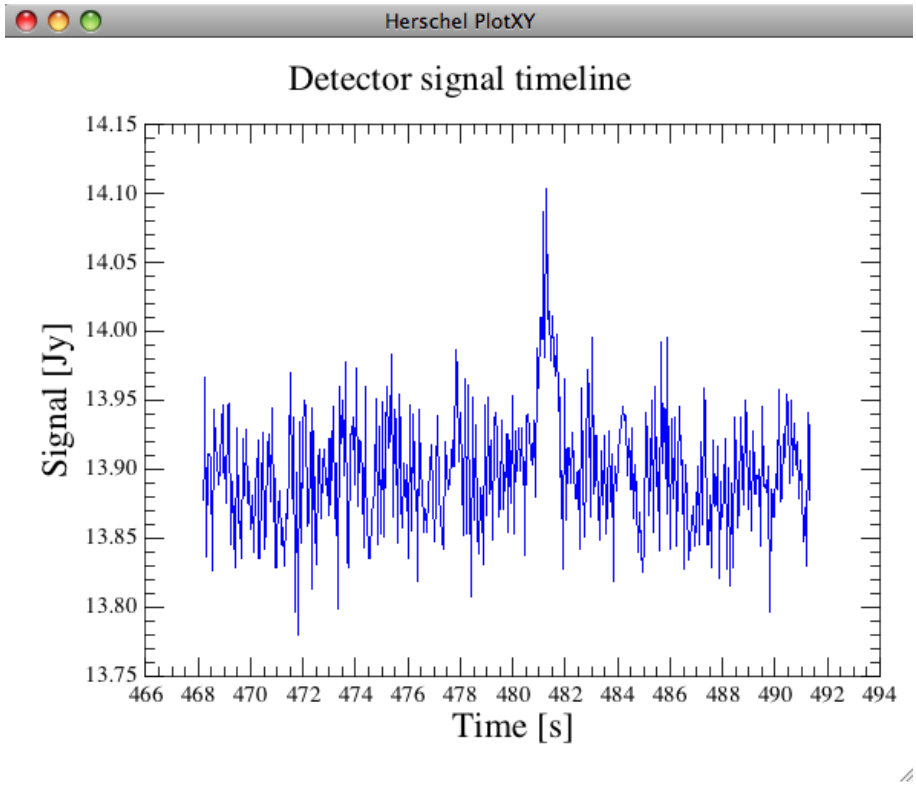


**Figure 4.14. Detector signal timelines**

Finally for this stage of the pipeline, we shall store our product in the Level 1 context:

```
  if tempStorage:
   ref=ProductSink.getInstance().save(psp)
   level1.addRef(ref)
  else:
   level1.addProduct(psp)

  obs.level1=level1
  # promote to LEVEL1_PROCESSED
  obs.obsState = ObservationContext.OBS_STATE_LEVEL1_PROCESSED
```

And that brings us to the end of our Level 1 processing for Small Map.

## 4.3.4. Level 1 to Level 2 Processing

Our next step is to convert our Level 1 photometer product into our final maps, which constitute the Level 2 products for the Small Map pipeline.

```
if inputs.mapping -!= -'none':
 #
 # Flag to switch on and off the baseline removal
 useRemoveBaseline=True
 #
 # Create a SpireListContext to be used as input of map making
 scans=SpireListContext()
 #
 # Run baseline removal and populate the map making input
 for i in range(level1.count):
  if useRemoveBaseline:
   psp=level1.getProduct(i)
   psp=removeBaseline(psp,chanNum=chanNum)
   if tempStorage:
    ref=ProductSink.getInstance().save(psp)
    scans.addRef(ref)
   else:
    scans.addProduct(psp)
  else:
   scans.addRef(level1.refs[i])
 pass
 #
 # Run mapmaking
 if inputs.mapping == -'naive':
  mapPlw=naiveScanMapper(scans, array="PLW")
  inputs.progress=85
  mapPmw=naiveScanMapper(scans, array="PMW")
  inputs.progress=90
  mapPsw=naiveScanMapper(scans, array="PSW")
 else:

 chanNoise=obs.calibration.phot.chanNoiseList.getProduct(level1.getProduct(0).meta["biasMode"].valu
\
   level1.getProduct(0).startDate)
  mapPlw=madScanMapper(scans, array="PLW",chanNoise=chanNoise)
  inputs.progress=85
  mapPmw=madScanMapper(scans, array="PMW",chanNoise=chanNoise)
  inputs.progress=90
  mapPsw=madScanMapper(scans, array="PSW",chanNoise=chanNoise)
 pass
 #
 # Create a context with level 2 products (maps) and attach it to the observation
context
 level2=MapContext()
 for key in level1.meta.keySet():
  if key -!= -"creator" and key -!= -"creationDate":
   level2.meta[key]=level1.meta[key].copy()
 level2.creator=creator
 level2.type="level2context"
 level2.description="Context for SPIRE Level 2 products"
 level2.meta["level"]=StringParameter("20", -"The level of the product")
```

```
 level2.refs.put("PLW",ProductRef(mapPlw))
 level2.refs.put("PMW",ProductRef(mapPmw))
 level2.refs.put("PSW",ProductRef(mapPsw))
 obs.level2=level2
 #
 # promote to LEVEL2_PROCESSED
 obs.obsState = ObservationContext.OBS_STATE_LEVEL2_PROCESSED
 #
 # Create browse product and image
 createRgbImage=CreateRgbImageTask()
 browseProduct=createRgbImage(red=mapPlw,green=mapPmw,blue=mapPsw,percent=98.0,redFactor=1.0,
\
  greenFactor=1.0,blueFactor=1.0)
 #
 # Populate metadata of the browse product
 for par in ObsParameter.values():
  if obs.meta.containsKey(par.key) and par.key -!= -"fileName":
   browseProduct.meta[par.key]=obs.meta[par.key].copy()
 pass
 browseProduct.startDate=obs.startDate
 browseProduct.endDate=obs.endDate
 browseProduct.instrument=obs.instrument
 browseProduct.modelName=obs.modelName
 browseProduct.description="Browse Product"
 browseProduct.type="BROWSE"
 #
 # Attach the browse product to the ObservationContext
 obs.browseProduct=browseProduct
 #
 # Generate the browse image
 from herschel.ia.gui.image import ImageUtil
 imageUtil = ImageUtil()
 browseProductImage=imageUtil.getRgbTiledImage(\
  browseProduct["red"].data, browseProduct["green"].data,
browseProduct["blue"].data)
 obs.browseProductImage=browseProductImage.asBufferedImage
pass
```

Assuming that we have requested mapping products, we first of all flag the pipeline to perform baseline subtraction, set up a List Context to be used as input to the map making, perform the baseline subtraction on our Level 1 product and store it in our List Context:

```
 # Flag to switch on and off the baseline removal
 useRemoveBaseline=True
 #
 # Create a SpireListContext to be used as input of map making
 scans=SpireListContext()
 #
 # Run baseline removal and populate the map making input
 for i in range(level1.count):
  if useRemoveBaseline:
   psp=level1.getProduct(i)
   psp=removeBaseline(psp,chanNum=chanNum)
   if tempStorage:
    ref=ProductSink.getInstance().save(psp)
    scans.addRef(ref)
   else:
    scans.addProduct(psp)
  else:
   scans.addRef(level1.refs[i])
 pass
```

We use this as input for the map maker. The type of map produced is dependent on your input at the very start of the pipeline - naive or MadMap.

```
 # Run mapmaking
 if inputs.mapping == -'naive':
  mapPlw=naiveScanMapper(scans, array="PLW")
```

```
 inputs.progress=85
 mapPmw=naiveScanMapper(scans, array="PMW")
 inputs.progress=90
 mapPsw=naiveScanMapper(scans, array="PSW")
else:

chanNoise=obs.calibration.phot.chanNoiseList.getProduct(level1.getProduct(0).meta["biasMode"].valu
\
   level1.getProduct(0).startDate)
 mapPlw=madScanMapper(scans, array="PLW",chanNoise=chanNoise)
 inputs.progress=85
 mapPmw=madScanMapper(scans, array="PMW",chanNoise=chanNoise)
 inputs.progress=90
 mapPsw=madScanMapper(scans, array="PSW",chanNoise=chanNoise)
pass
```

Three maps are each produced for PSW, PMW and PLW, and are visible through the Product Viewer by right-clicking on the required variable in the Variable pane and selecting 'Open With':

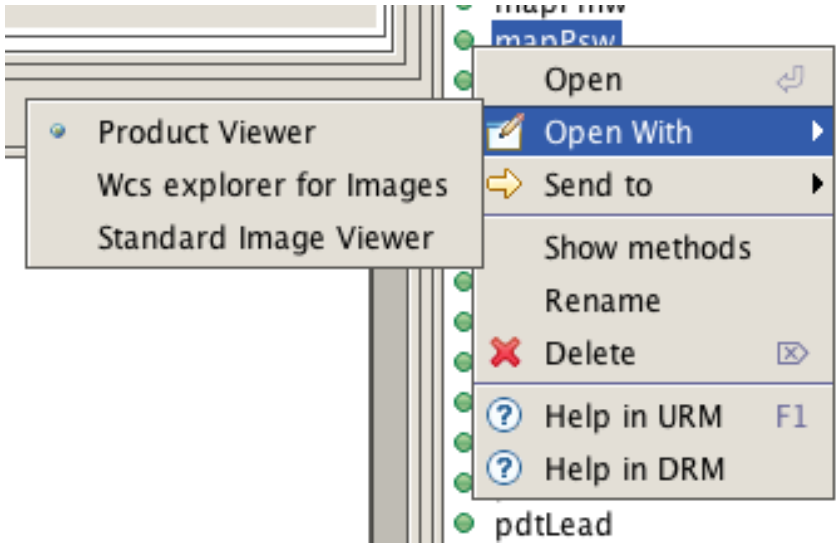


**Figure 4.15. Selecting the Product Viewer**

the actual map with fluxes (denoted as 'image');

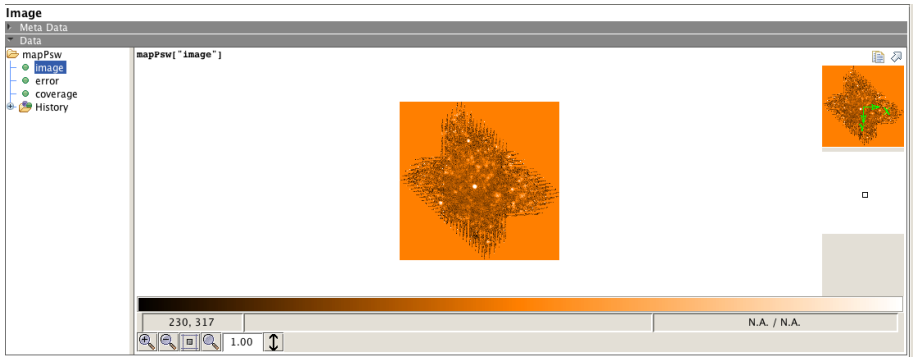


**Figure 4.16. Setting parameters for processing**

the statistical flux error map (denoted as 'error'):

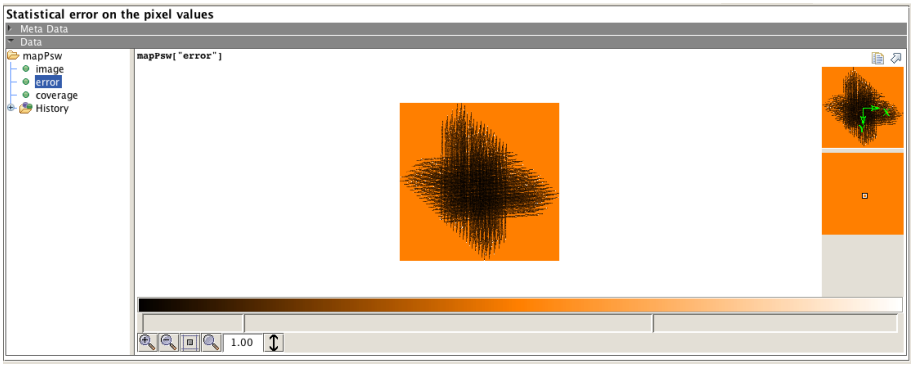


**Figure 4.17. Setting parameters for processing**

and an image which shows the coverage map for our scans (denoted as 'coverage'):

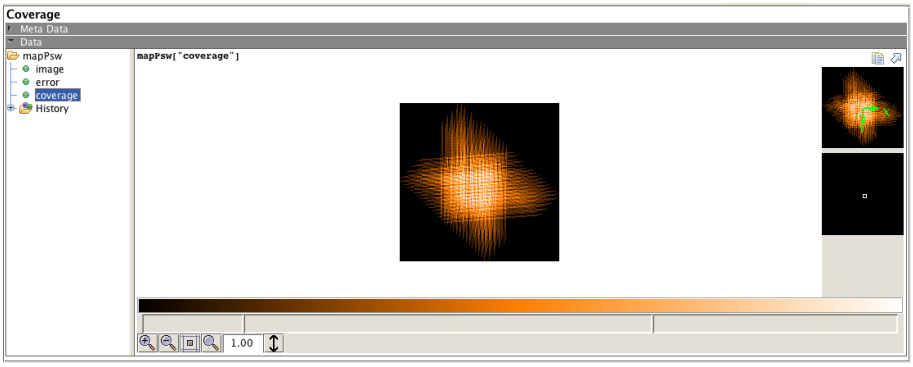


**Figure 4.18. Setting parameters for processing**

We can export our images to FITS files by right clicking on the respective variable in the Variable pane (e.g. mapPsw), and selecting 'Send To -> FITS file'.

We finally scan create a context to store our maps as Level 2 products, and attach them to the observation context.

```
 # Create a context with level 2 products (maps) and attach it to the observation
context
 level2=MapContext()
 for key in level1.meta.keySet():
  if key -!= -"creator" and key -!= -"creationDate":
   level2.meta[key]=level1.meta[key].copy()
 level2.creator=creator
 level2.type="level2context"
 level2.description="Context for SPIRE Level 2 products"
 level2.meta["level"]=StringParameter("20", -"The level of the product")
 level2.refs.put("PLW",ProductRef(mapPlw))
 level2.refs.put("PMW",ProductRef(mapPmw))
 level2.refs.put("PSW",ProductRef(mapPsw))
 obs.level2=level2
 #
 # promote to LEVEL2_PROCESSED
 obs.obsState = ObservationContext.OBS_STATE_LEVEL2_PROCESSED
```

Congratulations! You have now re-processed your Small Map data all the way to the final Level 2 maps!

# 4.4. Reprocessing SPIRE Point Source Mode Data

## 4.4.1. Prerequisites

For this data reprocessing example, we will be using the Point Source observation (obsID: 1342183474) of NGC 5315. We will in this example assume that you have received the engineering pipeline processed Level 0.5 data products from the HSC, and have stored them in a storage pool "1342183474_POF2_NGC5315", either by a direct download or through HIPE. The figure below outlines the steps required to process the Jiggle pipeline.

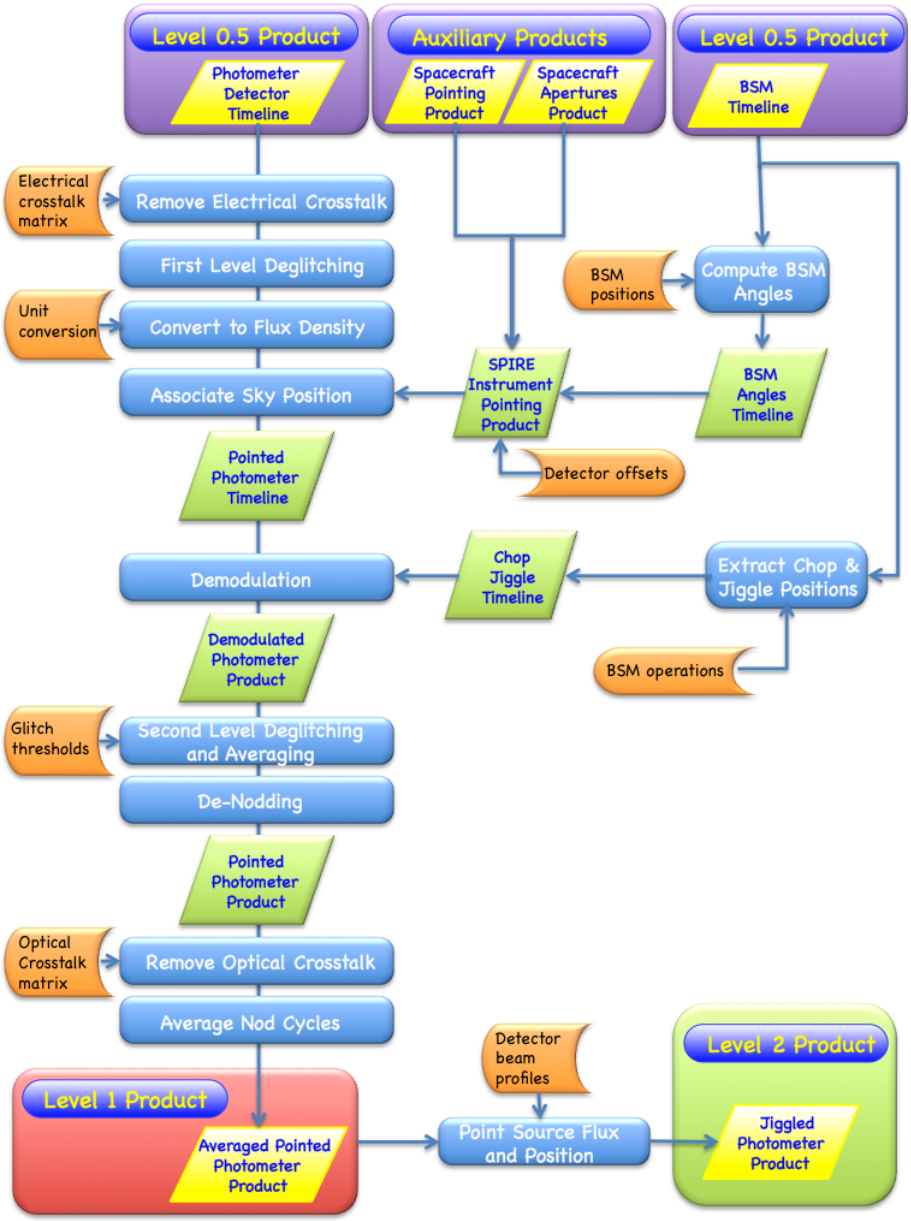


**Figure 4.19. The SPIRE POF2 Photometer Point Source pipeline.**

You can access the POF2 pipeline processing script by clicling on 'Pipeline' on the top bar within HIPE, selecting 'SPIRE' and then clicking on 'Photometer Point Source pipeline script (POF2)' - the script will open up in the Editor window within HIPE.

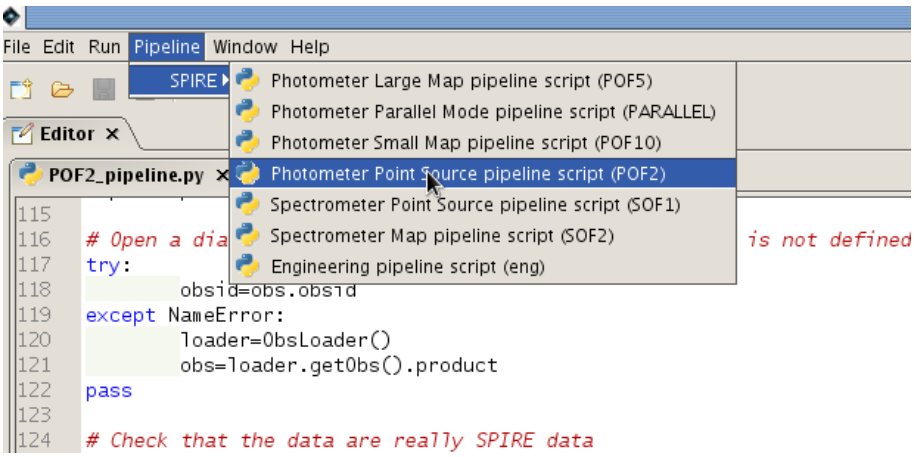


**Figure 4.20. Selecting the POF2 pipeline script**

To start processing, first, we need to make sure that you have imported all needed classes and task definitions required to run the POF2/point source pipeline:

```
# Import all needed classes
from herschel.spire.all import *
from herschel.spire.util import *
from herschel.ia.all import *
from herschel.ia.task.mode import *
from java.lang import Long
from java.util import *

# Import the script tasks.py that contains the task definitions
from herschel.spire.ia.pipeline.scripts.POF2.POF2_tasks import *

# Import the script input.py that contains the input definitions
from herschel.spire.ia.pipeline.scripts.POF2.POF2_input import *

# Import the script obsLoader.py that allows to load an ObservationContext from a
storage.
from herschel.spire.ia.scripts.tools.obsLoader import *
```

We must search our local pool "1342183474_POF2_NGC5315" for our observation context. We will run the ObsLoader pop-up window and input the ObsID and the name of the local pool to load the observation context, and open an pop up dialog box to take inputs such as if we wish to look at plots of intermediately processed pipeline products:

```
# Open the input dialog to enter inputs
inputs.openDialog()

# Open a dialog to load the ObservationContext if -"obs" is not defined.
try:
 obsid=obs.obsid
except NameError:
 loader=ObsLoader()
 obs=loader.getObs().product
pass
```

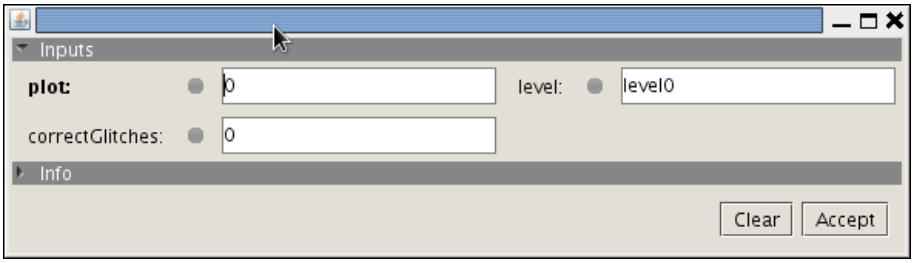


**Figure 4.21. Using ObsLoader to load the observation**

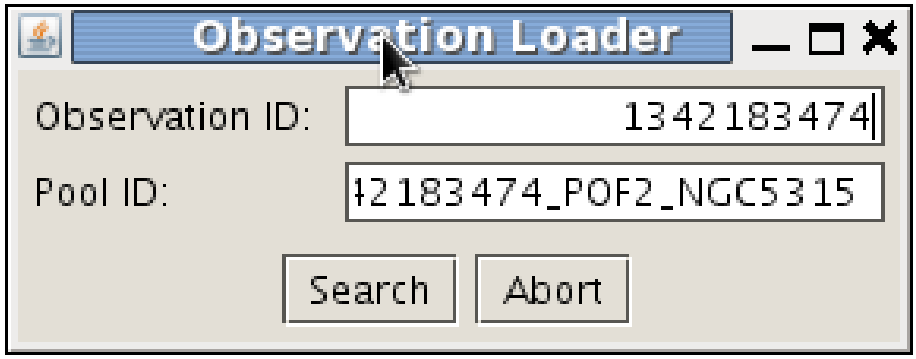


**Figure 4.22. Using ObsLoader to load the observation**

The pipeline also includes a check that the data really is SPIRE data, by raising a BadDataException if the data isn't:

```
# Check that the data are really SPIRE data
if obs.instrument -!= -"SPIRE":
    raise BadDataException("This ObservationContext cannot be processed with this
pipeline: it contains -"+obs.instrument+" data, not SPIRE data")
```

Next, we shall create we shall create a creator variable to store the relevant origin metadata for the Level and Level 2 contexts, a logger to follow the progress of the pipeline's execution, set up the time origin for any output plots and then finally, extract the ObsId of our observation and the calibration and auxillary products required for processing the POF2 pipeline:

```
# this is used to put in the creator metadata of level 1 and level 2 context the
version of SPG or of the pipeline
# that was executed
creator=herschel.share.util.Configuration.getProperty("hcss.ia.dataset.creator", -"$Revision:
1.68 $")

#create a logger for the pipeline
logger=TaskModeManager.getMode().getLogger()

# Shift of time origin for plots
t0=obs.startDate.microsecondsSince1958()*1e-6
obsid=obs.obsid
print -"processing OBSID=",obsid,"("+hex(obsid)+")"

# Extract from the observation context the calibration products that
# will be used in the script
bsmPos=obs.calibration.phot.bsmPos
bsmOps=obs.calibration.phot.bsmOps
detAngOff=obs.calibration.phot.detAngOff
elecCross=obs.calibration.phot.elecCross
optCross=obs.calibration.phot.optCross

# Extract from the observation context the auxiliary products that
# will be used in the script
hpp=obs.auxiliary.pointing
siam=obs.auxiliary.siam
```

## 4.4.2. Level 0 to Level 0.5 Processing (Optional)

If you do not have Level 0.5 products to hand, you will need to make the engineering conversion first from the raw Level 0 products - basically, we are converting the raw telemetry in the form of products into engineering units such as bolometer voltages and resistances timelines. We can run the engineering conversion pipeline from the Level 0 products obtained from the HSA to obtain our Level 0.5 products using:

```
# From Level 0 to Level 0.5
if inputs.level=="level0":
 # Run Engineering Conversion of level 0 products
 level0_5= engConversion(obs.level0,cal=obs.calibration)
 # Add the result to the observation in level 0.5
 obs.level0_5=level0_5
else:
 level0_5=obs.level0_5
#
# set the progress
inputs.progress=20
# counter for computing progress
count=0
```

## 4.4.3. Level 0.5 to Level 1 Processing

Now, we can process our data from Level 0.5 to Level 1. Looping over each BBID, we first convert the BSM telemetry into a Y, Y and Z angle timeline and then into a chopper id/jiggle id timeline. We can use these to create the SPIRE pointing product. We then perform a number of corrections to the data, after which we will have produced the Level 1 pipeline data product. In order to execute these steps in the most efficient manner possible, we execute a number of pipeline tasks within a single loop. A simplified vesion of this loop, adapted from the POF2 pipeline script, is given below:

```
# From Level 0.5 to Level 1
if inputs.level=="level0" or inputs.level=="level0_5":
 #
 dpparr=[DenodInput()]
 nrep=1
 nblocks=len(level0_5.getBbids(0xa321))
 #
 for bbid in level0_5.getBbids(0xa321):
  print -"Starting BBID=",hex(bbid)
  block=level0_5.get(obsid,bbid)
  # Get basic engineering data products
  pdt  = block.pdt
  bsmt = block.bsmt

  # run the task to convert BSM telemetry in a Y angle and Z angle timeline
  bat=calcBsmAngles(bsmt,bsmPos=bsmPos)

  # run the task to convert BSM telemetry in a chopper id & jiggle id timeline
  cjt = calcBsmFlags(bsmt, bsmOps=bsmOps)
  #
  #create the SpirePointingProduct
  spp=createSpirePointing(detAngOff=detAngOff,bat=bat,hpp=hpp, siam=siam)
  #
  # run the electrical crosstalk correction
  pdt=elecCrossCorrection(pdt,elecCross=elecCross)
  #
  # run the deglitch
  pdt=waveletDeglitcher(pdt, scaleMin=1.0, scaleMax=8.0, scaleInterval=5,
holderMin=-1.6,\
   holderMax=-0.1, correlationThreshold=0.6, correctGlitches=inputs.correctGlitches)
  #
  # run the flux conversion
```

```
 fluxConv=obs.calibration.phot.fluxConvList.getProduct(pdt.meta["biasMode"].value,pdt.startDate)
  pdt=photFluxConversion(pdt,fluxConv)
  #
  # associate the sky position
  ppt=associateSkyPosition(pdt,spp=spp)
  #
  # run the Demodulation task
  dpp = demodulate(ppt, cjt=cjt)
  #
  # second level deglitching
  dpp = secondDeglitching(dpp)
  #
  # average on jiggle position
  dpp = jiggleAverage(dpp)

  #
  ncyc=((dpp.bbCount-1)/4)+1
  if ncyc >= nrep+1:
   for k in range(ncyc-nrep):
    dpparr.append(DenodInput())
   dpparr[ncyc-1].addProduct(dpp)
   nrep=ncyc
  else:
   dpparr[ncyc-1].addProduct(dpp)
  print -"Completed BBID=0x%x (%i/%i)"%(bbid,count+1,nblocks)
  # set the progress
  count=count+1
  inputs.progress = 20+(60*count)/nblocks
 #
 # denodding
 ppps=[]
 for i in range(nrep):
  denin=dpparr[i]
  if denin.count == 0:
   print -"nod cycle -",i," doesn't have any data"
   logger.severe("nod cycle -"+i.toString()+" doesn't have any data")
   continue
  ppp=deNodding(denin)
  ppps.append(ppp)
 #
 if len(ppps) == 0:
  print -"No PPP produced due to missing data. This observation CANNOT be
processed!"
  logger.severe("No PPP produced due to missing data. This observation CANNOT be
processed!")
  raise MissingDataException("No PPP produced due to missing data. This
observation CANNOT be processed!")
 #
 for i in range(len(ppps)):
  # run the optical crosstalk correction
  ppps[i]=photOptCrossCorrection(ppps[i],optCross=optCross)
 #
 # averaging over nodding
 appp = nodAverage(ppps)
 #
 # Add level 1 context to observation context
 level1=Level1Context(obsid)
 for key in level0_5.meta.keySet():
  if key -!= -"creator" and (not key.endswith("Date")) and key -!= -"fileName" and \
  key -!= -"type" and key -!= -"description":
   level1.meta[key]=level0_5.meta[key].copy()
 level1.creator=creator
 level1.addProduct(appp)
 obs.level1 = level1

 # promote to LEVEL1_PROCESSED
 obs.obsState = ObservationContext.OBS_STATE_LEVEL1_PROCESSED
else:
 level1=obs.level1
 appp=level1.getProduct(0)
pass
```

To break up this loop into its constituent parts, first of all, we set dpparr as an array to host input the data after it has been demodulated later in the pipeline, and obtain the number of buidling blocks from the Level 0.5 products:

```
dpparr=[DenodInput()]
 nrep=1
 nblocks=len(level0_5.getBbids(0xa321))
```

Next, we grab the engineering products from the Level 0.5 output product:

```
block=level0_5.get(obsid,bbid)
# Get basic engineering data products
pdt  = block.pdt
bsmt = block.bsmt
```

The next required step is to convert the BSM telemetry into a Y angle and Z angle timeline, and a chopper ID/jiggle ID timeline, respectively - using this ouput product in conjunction with the pointing and the SIAM files, we can create the SPIRE Pointing Product:

```
# run the task to convert BSM telemetry in a Y angle and Z angle timeline
bat=calcBsmAngles(bsmt,bsmPos=bsmPos)

# run the task to convert BSM telemetry in a chopper id & jiggle id timeline
cjt = calcBsmFlags(bsmt, bsmOps=bsmOps)

#create the SpirePointingProduct
spp=createSpirePointing(detAngOff=detAngOff,bat=bat,hpp=hpp, siam=siam)
```

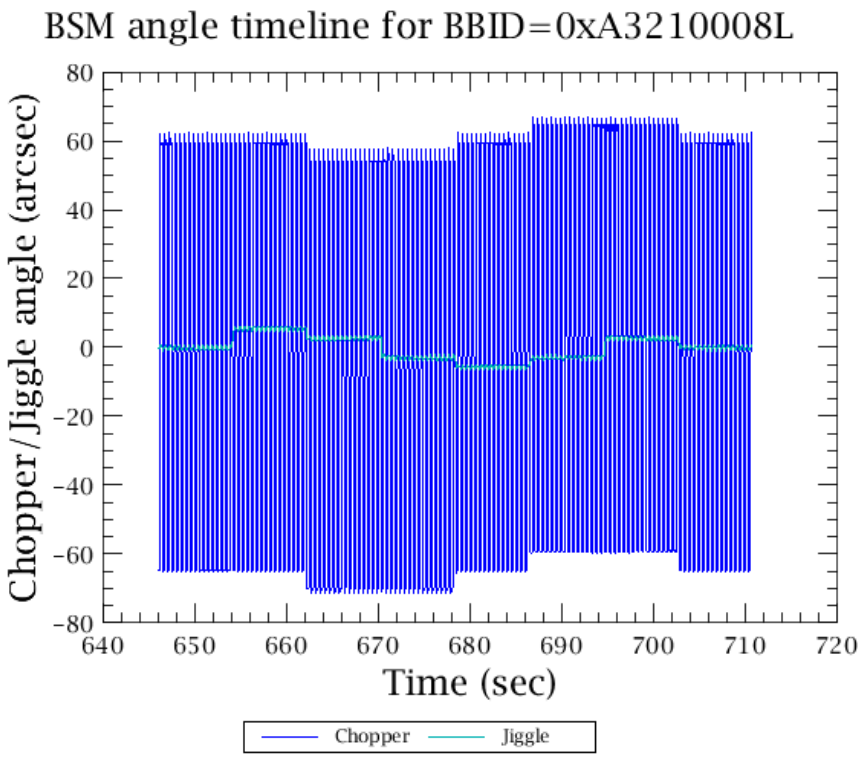


**Figure 4.23. BSM Angle Timeline**

We then in turn provide a number of corrections to our Level 0.5 datasets - electrical crosstalk correction, deglitching, flux conversion, sky position association, demodulation of the data, second level deglitching and averaging of the demodulated data. The deglitching task parameters can be tweaked as required - see the SPIRE Users Manual for a more in-depth discussion of the parameters for this task, and the ranges allowed.

```
# run the electrical crosstalk correction
pdt=elecCrossCorrection(pdt,elecCross=elecCross)

# run the deglitch
pdt=waveletDeglitcher(pdt, scaleMin=1.0, scaleMax=8.0, scaleInterval=5,
holderMin=-1.6,\
   holderMax=-0.1, correlationThreshold=0.6, correctGlitches=inputs.correctGlitches)

# run the flux conversion
fluxConv=obs.calibration.phot.fluxConvList.getProduct(pdt.meta["biasMode"].value,pdt.startDate)
pdt=photFluxConversion(pdt,fluxConv)

# associate the sky position
ppt=associateSkyPosition(pdt,spp=spp)


# run the Demodulation task
dpp = demodulate(ppt, cjt=cjt)

# second level deglitching
dpp = secondDeglitching(dpp)

# average on jiggle position
dpp = jiggleAverage(dpp)
```

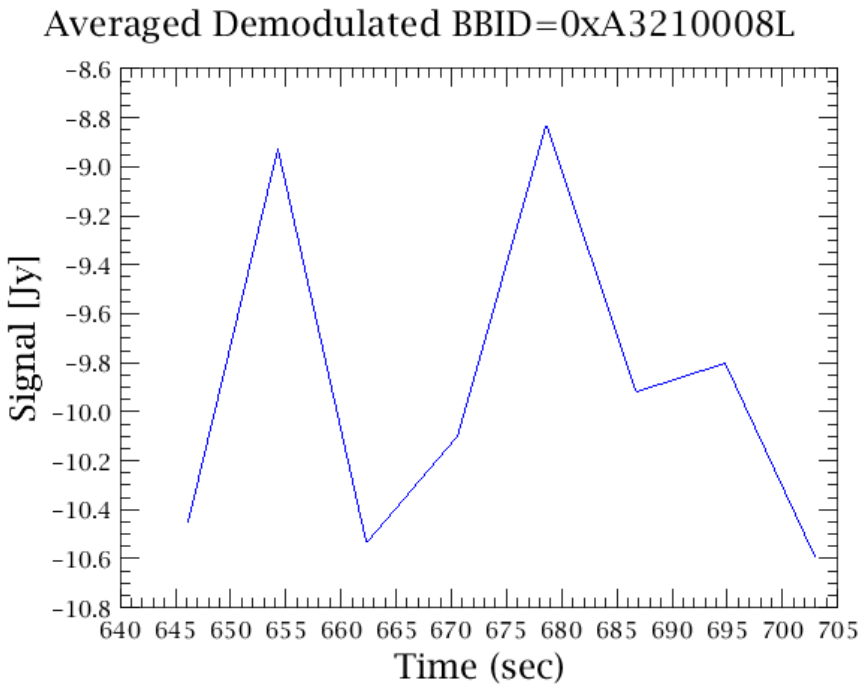


**Figure 4.24. Averaged demodulated data**

This is repeated over the full number of nod cycles, and the averaged, demodulated output is appended to the dpparr output product:

```
  #
  ncyc=((dpp.bbCount-1)/4)+1
```

```
 if ncyc >= nrep+1:
  for k in range(ncyc-nrep):
   dpparr.append(DenodInput())
  dpparr[ncyc-1].addProduct(dpp)
  nrep=ncyc
 else:
  dpparr[ncyc-1].addProduct(dpp)
 print -"Completed BBID=0x%x (%i/%i)"%(bbid,count+1,nblocks)

 # set the progress
 count=count+1
 inputs.progress = 20+(60*count)/nblocks
```

The demodulated data is then denodded, the optical crosstalk correction is applied and finally, the data is nod averaged:

```
denodding
ppps=[]
for i in range(nrep):
 denin=dpparr[i]
 ppp=deNodding(denin)
 ppps.append(ppp)
      #
for i in range(len(ppps)):
 # run the optical crosstalk correction
 ppps[i]=photOptCrossCorrection(ppps[i],optCross=optCross)
#
# averaging over nodding
appp = nodAverage(ppps)
```

This nod averaged product forms the Level 1 output product, and is written out as such from the pipeline. The Level 1 context is then added to the observation context, finishing Level 1 processing.

```
# Add level 1 context to observation context
level1=Level1Context(obsid)
for key in level0_5.meta.keySet():
 if key -!= -"creator" and (not key.endswith("Date")) and key -!= -"fileName" and \
 key -!= -"type" and key -!= -"description":
  level1.meta[key]=level0_5.meta[key].copy()
level1.creator=creator
level1.addProduct(appp)
obs.level1 = level1

# promote to LEVEL1_PROCESSED
obs.obsState = ObservationContext.OBS_STATE_LEVEL1_PROCESSED
```

# 4.4.4. Level 1 to Level 2 Processing

In the final step of the POF2 Point Source pipeline processing, we can obtain the final Level 2 products for Point Source Observations, by passing the APP to the "pointSourceFlux" module, and by inspecting the output JPP product:

```
# user products
jpsfp = pointSourceFit (appp)
jpp   = sourceFlux (jpsfp)
```

This completes the standard POF2 pipeline.

Congratulations! You have now successfully reprocessed your point source data from Level 0 to the final Level 2 user products! Additional and more detailed information regarding the data processing modules and the data at the various levels of processing can be found in the SPIRE Users Manual.

# 4.5. SPIRE Spectroscopy Data Processing

## 4.5.1. Reprocessing SPIRE spectrometer data

In the standard processing applied at the HSC, the SPIRE spectrometer pipeline is divided into two scripts - one for sparse observations (SOF1) and one for intermediate (4 point jiggle), or fully (16 point jiggle) sampled observations (SOF2). In either case the observation may consist of a set of telescope pointings in a raster pattern on the sky. The sequence of processing steps applied to the data are illustrated schematically in Figure 4.25.

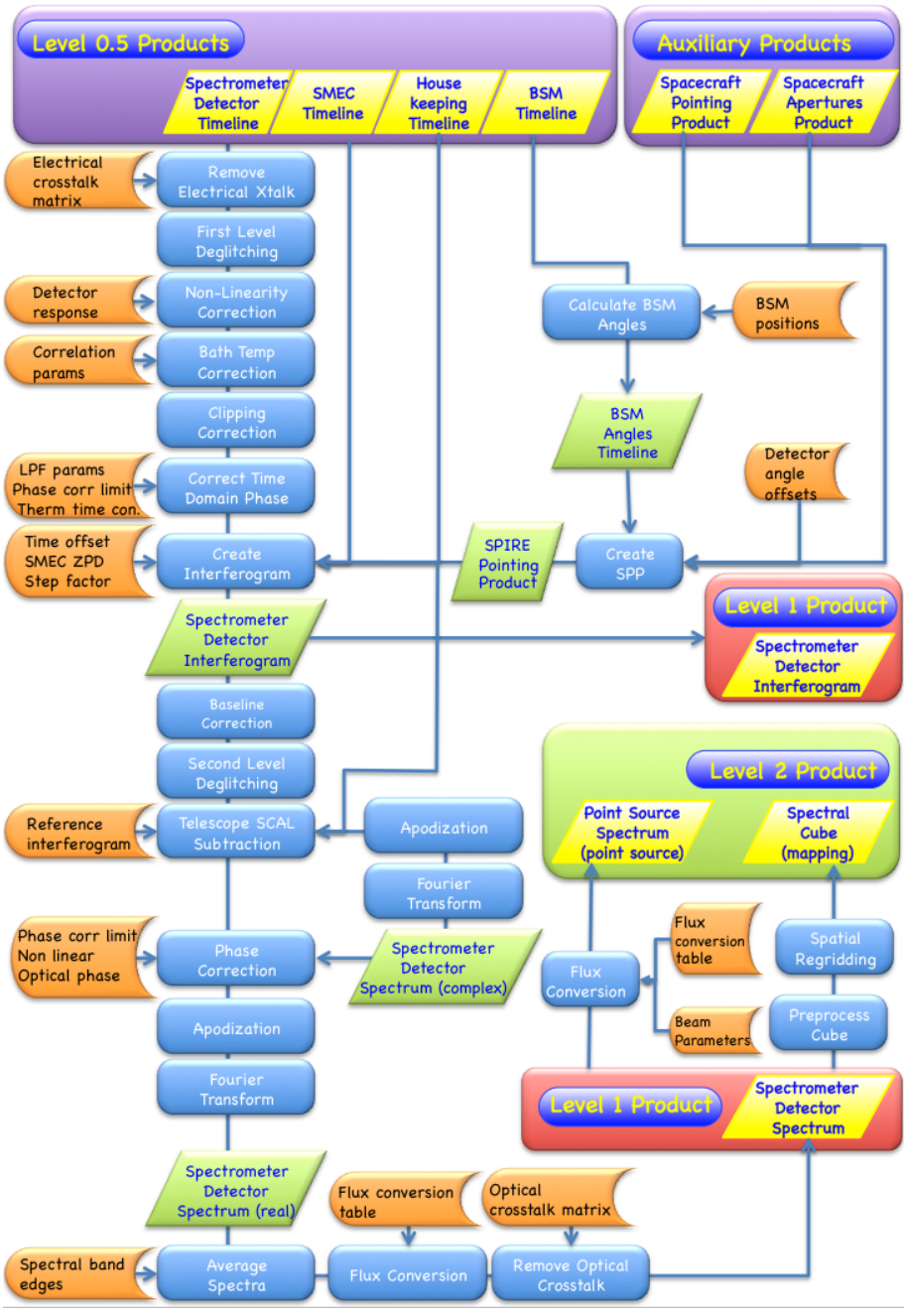


**Figure 4.25. The SPIRE Spectrometer pipeline.**

Reprocessing your observation using the simple steps described in this chapter may improve your results significantly over the output of the standard pipeline for HCSS v4 (or earlier) for two main reasons:

1. The standard pipeline up to HCSS v4 uses a single sky measurement to remove the telescope and instrument background emission. In reality, both background levels vary from day to day. Day-dependent reference observations are now available to subtract a background that better matches the thermal conditions during the science observation.

2. The calibration products evolve rapidly at this point. It is very likely that your standard pipeline results were generated using an older calibration tree version. Updated calibration products can be applied using the latest calibration context.

While the data reprocessing described here usually results in an improvement to the final spectrum, there may still be some spectral artifacts (particularly for weaker sources). These will be further dealt with in HCSS v5.

Currently, the reprocessing steps outlined here apply only to sparsely sampled observations. Further details of the additional processing needed for mapping observations will be added for HCSS v5.

In order to reprocess your data, you have to do the following:

1. Download the raw data from the HSA

2. Update the calibration tree attached to the observation to spire_cal_4_0

3. Get the SCalSpecInterRef calibration product closest in time to your observation from the SPIRE ICC website

4. Re-run the pipeline from level-0.5 data using updated calibration files

The following sections explain these steps in more detail using an interactive pipeline script that is reproduced in full at the end of the chapter. The two standard pipeline scripts for the Spectrometer are also available from within Hipe (from the "pipeline" button in the taskbar), but they contain many lines of code specific to automated processing in the standard processing environment. Eventually, the simpler interactive script will also be available from the taskbar in Hipe, but not yet in HCSS v4.

# 4.5.2. Options available to the user

1 **Set your OBSID and pool name**

Specify the observation ID (obsid) and data pool name that contains the level 0.5 data for your observation, i.e. the pool name where you have stored the data downloaded from the archive.

```
myObsID    = [obsid]
myDataPool = [pool name]
```

2 **Choose to limit the number of detectors**

Setting this option to be true will result in only the central detectors SLWC3 and SSWD4 being processed. If this option is set to be false, all the detectors will be processed. Selecting only the central detectors saves both processing time and memory usage.

```
processOnlyCenterDetectors = 1
```

3 **Select the dark sky reference interferogram**

Using a reference interferogram which was taken under different conditions is the main cause of errors in the resulting spectra. To produce an accurate spectrum, a reference interferogram taken in

similar conditions to that of the source observation must be subtracted. Often, the best choice for a reference interferogram is the one which was taken closest in time to the source observation. The reference interferogram calibration files can be downloaded from the SPIRE ICC website http://www.spire.rl.ac.uk/icc/InterRefFiles.html The reference observations taken each day may be shallower than the source observation and introduce noise. Eventually, the goal is to provide deep reference observations that match the thermal conditions during the science observation.

```
myInterRef = -"[Enter path here + filename]"
```

For example:

```
myInterRef = \
"SCalSpecInterRef_CR_nominal_20050222_50002AA3_average_fourier_ALL_DETS.fits"
```

4 **Choose to apodize interferograms or not**

Choosing apodize = 1 will apply the standard apodization function to the interferogram, reducing the ringing in the instrument line shape wings at the cost of spectral resolution. Setting apodize = 0 will avoid apodization altogether and preserve the best spectral resolution available from the SPIRE spectrometer.

```
apodize = 0
```

5 **Define the output directory**

The output directory defined here will be used to save the resulting FITS files containing the final spectra.

```
outDir = -"[Enter path here]"
```

# 4.5.3. Detailed description of the processing script

## 4.5.3.1. Define some Jython "Methods"

The methods shown at the beginning of the script are used for merging the observation building blocks together. These methods are needed for data taken prior to OD 302 since a calibration building block was sometimes inserted into the middle of an observation. Since OD 302, calibration building blocks are only placed at the end of each observation, and the spectral scans are not divided up.

```
def mergeNhkt(nhkts):
    for i in range(1, len(nhkts)):
        nhkts[0]['signal'].addRowsByIndex(nhkts[i]['signal'])
        nhkts[0]['mask'].addRowsByIndex(nhkts[i]['mask'])
        nhkts[0].meta['endDate'].value = nhkts[-1].meta['endDate'].value
    return nhkts[0]

def mergeSdis(sdis):
    bigSdi = SpectrometerDetectorInterferogram()
    bigSdi.meta = MetaData(sdis[0].meta)
    scanNumbers = []
    for sdi in sdis:
        scanNumbers.append(sdi.getNumScans())
    if len(sdis) == 1:
        return sdis[0]
    i=0
```

```
    for sdi in sdis:
        toAdd = SUM(scanNumbers[i+1:len(scanNumbers)])
        for scanNumber in sdi.getScanNumbers():
            thisScan = sdi.removeScan(scanNumber)
            thisScan.setScanNumber(thisScan.getScanNumber()+toAdd)
            bigSdi.setScan(thisScan)
        i=i+1
    return bigSdi
```

## 4.5.3.2. Define the central detectors and thermistors and dark pixels

The value of the boolean variable "processOnlyCenterDetectors" was defined in the user input section. Here, we specify the names of the central detectors, thermistors, and dark pixels so that if "processOnlyCenterDetectors" was set to 1, we know which detectors to keep and which to remove later in the script.

```
detsToKeep = ["SLWC3", -"SSWD4"]
therms = ["SLWT1", -"SLWT2", -"SSWT1", -"SSWT2", \
          -"SSWDP1", -"SSWDP2", -"SLWDP1", -"SLWDP2"]
firstCut = therms
firstCut.extend(detsToKeep)
```

## 4.5.3.3. Load an observation context into HIPE

The following lines read in the observation from the local storage.

```
storage = ProductStorage(myDataPool)
obs     = storage.select(MetaQuery(ObservationContext,\
             -"p","p.meta['obsid'].value == %iL"%myObsID))[0].product
```

## 4.5.3.4. Attach the latest calibration tree to the observation

In general, the appropriate calibration files will be included with the observation data. However, only very recent data in the HSA have the latest calibration files attached. Therefore, the following two lines fetch the v4 calibration tree from the HSA (will ask for username and password), and update the observation context by attaching them. If you run the script for many observations, you could read the calibration context from the HSA, save it as a local pool to disk, and read it in from there (rather than fetching from the HSA for every observation).

```
cal = spireCal(calTree="spire_cal_4_0")
obs.calibration.update(cal)
```

## 4.5.3.5. Start processing from the Level 0.5 products

Use the following code in HIPE to load in the Level 0.5 products relevant to an FTS Scan building block (bbid = 0xa106XXXX: the first scanning building block is 0xa1060001) - there is usually only one scanning building block that contains the observation data (except for long observations before OD302 - see above). The following lines set up empty Jython lists to contain the results for each building block, and starts a loop over the scanning building blocks.

```
sdis  = []
nhkts = []
for bbid in obs.level0_5.getBbids(0xa106):
```

## 4.5.3.6. Extracting the Spectrometer Detector Timeline

The following line extracts the detector timeline product from the observation context. In the sample data given below, the observation contains 20 spectral scans, see Figure 4.26. The data taken during one scan are described in more detail in Figure 4.27.

```
sdt   = obs.level0_5.get(myObsID, bbid).sdt
```

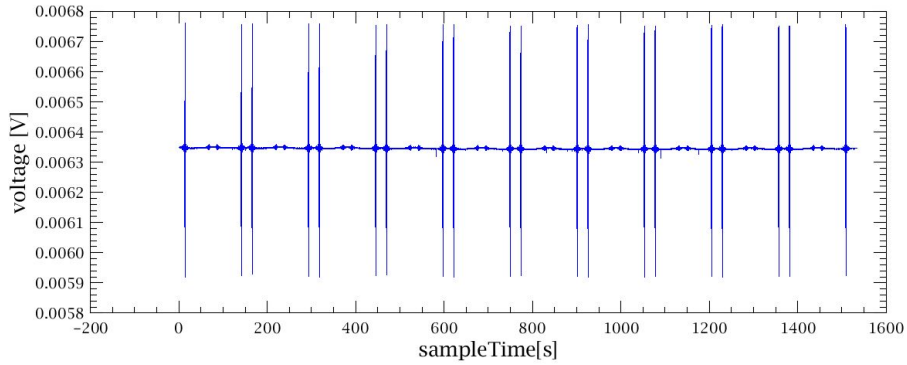


**Figure 4.26. The timeline of the SLWC3 detector for 20 scans.**

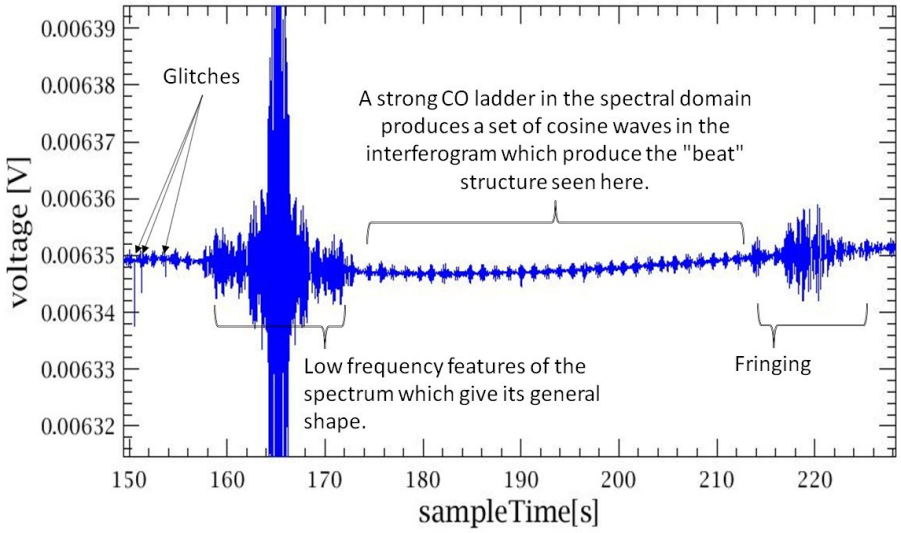


**Figure 4.27. Annotated interference pattern for one scan.**

## 4.5.3.7. Nominal House Keeping Timeline

The following line extracts the nominal housekeeping timeline product from the observation context. This product contains the instrument "housekeeping" data - for example, temperatures of various instrument components, as well as voltages, phases, etc.

```
nhkt  = obs.level0_5.get(myObsID, bbid).nhkt
```

For example, follow the steps below to view instrument thermometry:

1. Open the nhkt file from the variables list by double clicking on it.

2. Right-click on "Signal" and open with "TablePlotter"

3. Activate the drop-down menu for the different axes to plot any of the instrument sensor measurements, e.g. SCALTEMP.

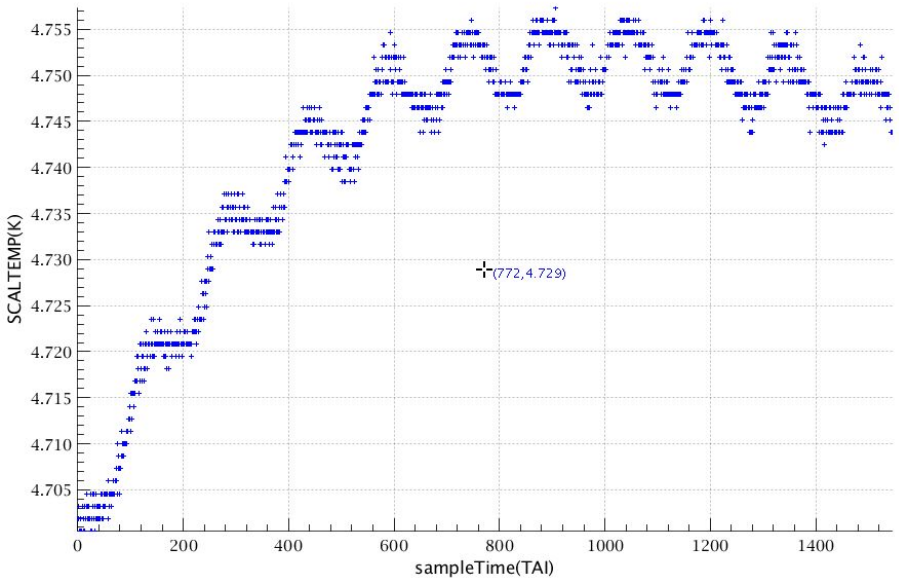


**Figure 4.28. SCALTEMP, a good indication of the temperature of the optical bench, as a function of time during the observation.**

## 4.5.3.8. Spectrometer MEChanism Timeline

The following line extracts the spectrometer mirror mechanism timeline product from the observation context. This product contains data concerning the position of the mirror that is mounted on the linear translation stage mechanism.

```
smect = obs.level0_5.get(myObsID, bbid).smect
```

The TablePlotter can be used in the same fashion as with the NHKT product to produce a sample plot of the mirror scan distance during the observation.

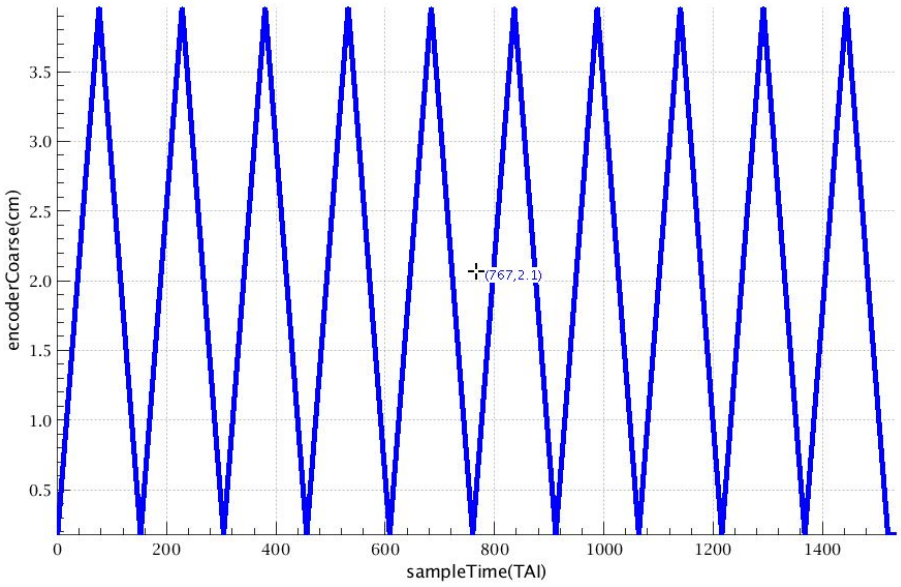


**Figure 4.29. The position of the stage mechanism as a function of time during the observation.**

The correlation between the SCALTEMP plot in Figure 4.28 and the mirror stage mechanism plot in Figure 4.29 indicates that the instrument temperature is affected by the operation of the stage mechanism.

## 4.5.3.9. Removing unnecessary channels

At this point, we remove the detectors that are not required (using the list of detectors to keep which was defined above).

```
if (processOnlyCenterDetectors):
    for chan in sdt.channelNames:
        if chan not in firstCut:
            sdt.removeColumn(chan)
```

## 4.5.3.10. Apply first level deglitching

The waveletDeglitcher removes the effects of cosmic rays from the SDT and replaces the gaps with a polynomial fit since an interferogram with gaps cannot undergo a correct Fourier transform. We recommend not altering the parameters of this correction.

```
sdt = waveletDeglitcher(sdt,reconstructionPointsAfter=3, \
                        reconstructionPointsBefore=2, \
                        correctGlitches=Boolean.TRUE,\
                        scaleMin=1, scaleMax=8, scaleInterval=5,\
                        holderMin=-1.4, holderMax=-0.6,\
                        correlationThreshold=0.85,\
                        optionReconstruction="polynomialFitting",\
                        degreePoly=6, fitPoints=8)
```

## 4.5.3.11. Apply the Non-linearity and Temperature Drift Corrections

The non-linearity correction is required because the response of the bolometric detectors is non-linear for substantially increased flux rates. The reponse of the bolometers also depends upon their operating temperature. Therefore temperature drift and non-linearity corrections must be applied together.

```
sdt = specNonLinearityCorrection(sdt, \
                            nonLinCorr=obs.calibration.spec.nonLinCorr)
sdt = temperatureDriftCorrection(sdt, \
                            tempDriftCorr=obs.calibration.spec.tempDriftCorr)
```

## 4.5.3.12. Remove the thermistor channels

After the temperature drift correction the thermistor channels can be removed.

```
if (processOnlyCenterDetectors):
    for chan in sdt.channelNames:
        if chan not in detsToKeep:
            sdt.removeColumn(chan)
```

## 4.5.3.13. Correct the detector signals for clipping

The detectors have a finite dynamic range and can saturate when observing a particularly strong source. This results in an interference pattern peak which has been "clipped" off, eventually leading to an

incorrect spectrum. Consequently, saturated detector signals must be reconstructed prior to the Fourier transform.

```
    sdt = clippingCorrection (sdt)
```

## 4.5.3.14. Correct the detector signals for time shifts

The thermal response of the detectors and the read-out electronics is not instantaneous and imparts a time delay to the recorded signals that is corrected here.

```
    sdt = timeDomainPhaseCorrection(sdt,\
                                lpfPar=obs.calibration.spec.lpfPar,\
                                chanTimeConst=obs.calibration.spec.chanTimeConst)
```

## 4.5.3.15. Create a Spire Pointing product

The SPIRE pointing product allows the calculation of the position on the sky that the instrument detectors were viewing. This is different from the line of sight of the Herschel telescope for potentially three reasons:

1. The detector arrays are offset from the boresight of the Herschel telescope.

2. If you are using more than just the central detectors, these are offset by a different angle.

3. The beam steering mirror can also alter the angular offset of the detector.

```
    bat = calcBsmAngles(nhkt, bsmPos=obs.calibration.spec.bsmPos)
    spp = createSpirePointing(hpp=obs.auxiliary.pointing, siam=obs.auxiliary.siam, \
                              detAngOff=obs.calibration.spec.detAngOff, bat=bat)
```

## 4.5.3.16. Interpolate SDT and SMECT to create Interferograms

With the knowledge of the optical path difference from the SMECT combined with the SDT that has been corrected for non linear response, temperature drift, clipping and signal time delay, a level-1 interferogram can produced. The interferogram (i.e. the signal as a function of optical path difference) for each scanning building block is appended to the list of Spectrometer Detector Interferograms (SDIs) which is then merged into a single SDI outside of the loop.

```
    sdi = createIfgm(sdt=sdt, smect=smect, nhkt=nhkt, spp=spp, \
                     smecZpd=obs.calibration.spec.smecZpd,\
                     chanTimeOff=obs.calibration.spec.chanTimeOff,\
                     smecStepFactor=obs.calibration.spec.smecStepFactor,\
                     interpolType= -"spline")
    sdis.append(sdi)
    nhkts.append(nhkt)
sdi  = mergeSdis(sdis)
nhkt = mergeNhkt(nhkts)
```

## 4.5.3.17. Subtract the interferogram baseline and apply second level deglitching

By running the pipeline script line by line, you can inspect the results at each stage to gain a more visual representation of the alterations from each processing step. If you compare the SDI before and

after baseline correction, you will notice that the baseline after the correction should be at zero (but note that once the correction has been applied it overwrites the input variable!).

```
sdi = baselineCorrection(sdi, type="fourier", threshold=4)
sdi = deglitchIfgm(sdi, deglitchType="MAD", thresholdFactor=4)
```

## 4.5.3.18. Subtract the reference interferogram from the source interferograms

The following step subtracts the reference interferogram contained in the calibration file specified in the user input section of the script.

```
interRef = fitsReader(myInterRef)
sdi = telescopeScalSubtraction(sdi, interRef=interRef, nhkt=nhkt)
```

The resulting interferogram is shown in Figure 4.30.

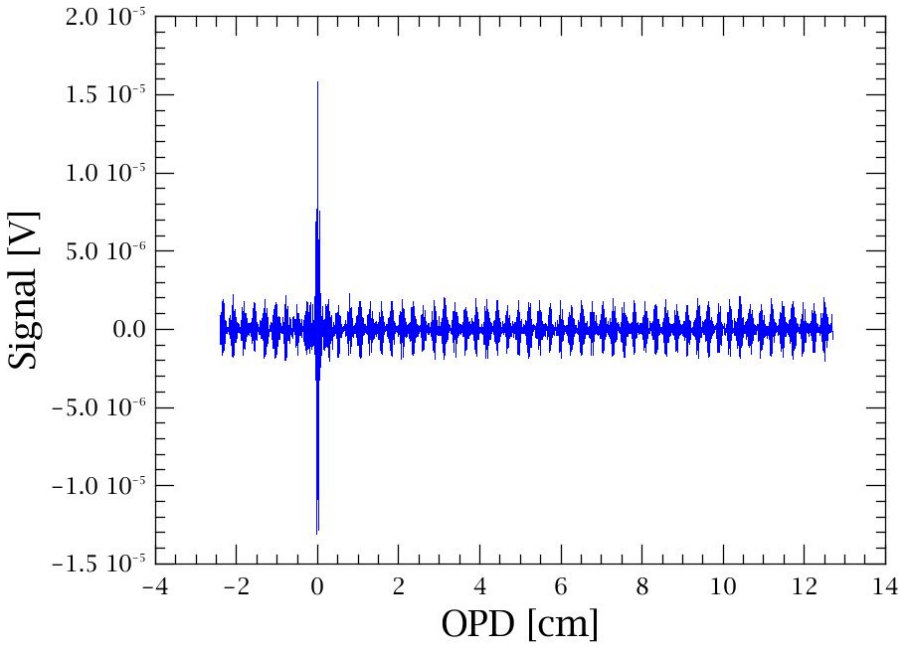


**Figure 4.30. The source interferogram.**

## 4.5.3.19. Apply Interferogram Phase correction

Since the strong signal in the SDI is not always exactly centred at zero in optical path difference, this must be corrected before the Fourier transform to produce the final spectrum. This correction is achieved by extracting the symmetric double-sided portion of the SDI (see Figure 4.31), followed by a Fourier transform which will produce a spectrum containing both real and imaginary parts (see Figure 4.32). The arctan of the ratio of the imaginary over the real part of the complex spectrum is commonly referred to as phase. The phase correction adjusts the central position of the extracted symmetric portion of the SDI so as to eliminate the imaginary part of the spectrum. Once this phase correction is found, the same correction is applied to the whole SDI prior to the Fourier transform to produce the almost finalised spectrum.

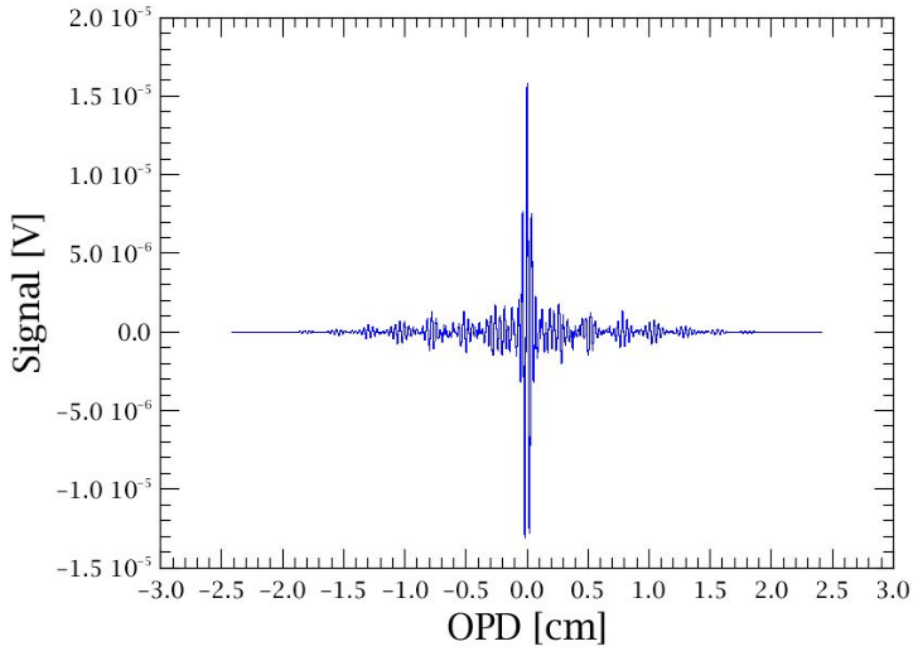


**Figure 4.31. The double-sided portion of the interferogram.**

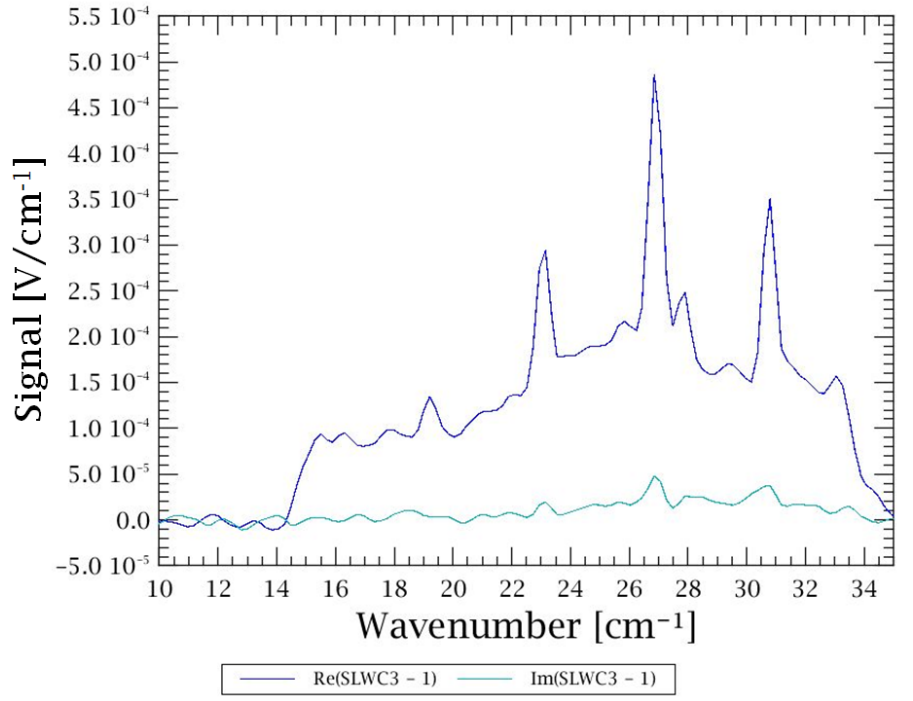


**Figure 4.32. The complex spectrum from the double-sided interferogram.**

```
presdi = apodizeIfgm(sdi, apodType="prePhaseCorr", apodName="aNB_20")
dsds   = fourierTransform(sdi=presdi, ftType="prePhaseCorr", zeroPad="None")
sdi    = phaseCorrection(sdi, sds=dsds, \
                         polyDegree=2, pcfSize=127, \
                         nlp=obs.calibration.spec.nlp, \
                         phaseCorrLim=obs.calibration.spec.phaseCorrLim)
```

Also, if the "apodize" variable is set to True, the interferogram is apodized prior to the Fourier transform using the standard apodization function.

```
if apodize:
    sdi = apodizeIfgm(sdi, apodType="postPhaseCorr", apodName="aNB_15")
```

## 4.5.3.20. Transform the Phase-corrected interferograms

The phase corrected SDIs can now undergo the Fourier transform to produce the almost finalised spectrum (see Figure 4.33). The spectrum stretches from 0 to approximately 200 wavenumbers for both detectors, covering a much wider spectral range than the optical passband of the instrument.

```
ssds = fourierTransform(sdi=sdi, ftType="postPhaseCorr", zeroPad="standard")
```

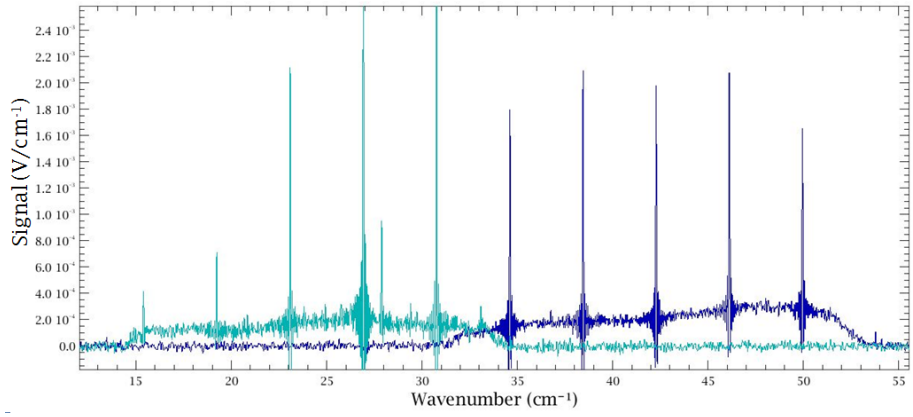


**Figure 4.33. The phase-corrected spectrum from a single scan.**

## 4.5.3.21. Read in the appropriate flux conversion calibration products

The following lines get the correct flux conversion and beam parameter calibration products depending on whether apodization has been selected in the user input section.

```
if apodize:
    fluxConv = obs.calibration.spec.fluxConv
    beamParam = obs.calibration.spec.beamParamList.getProduct(1, -"nominal", \
                                                              obs.startDate)
else:
    fluxConv = obs.calibration.spec.fluxConvList.getProduct("HR", -"unapod", \
                                                           -"nominal", obs.startDate)
    beamParam = obs.calibration.spec.beamParamList.getProduct(0, -"nominal", \
                                                              obs.startDate)
```

## 4.5.3.22. Average the spectra and remove out-of-band data

The spectra can be averaged by combining all of the observed scans, or by combining forward and backward scans of the mirror separately. In the following line, "separateScanDirections" is set to zero, indicating that all scans are to be averaged together. If it was set to 1, forward and backward scans would be kept separately. The keyword boolean "INCLUDE_OOB" is set to 0 to truncate spectral data to the optical passband, the scientifically useful wavenumber range - see the resulting spectrum in Figure 4.34. If this keyword was set to 1, the spectral data would not have been truncated. The slight undulation of the spectrum is caused by the Relative Spectral Response Function of the SLW and SSW filters.

```
ssds = averageSpectra(ssds, separateScanDirections=0, \
                      INCLUDE_OOB=0, bandEdge=obs.calibration.spec.bandEdge)
```

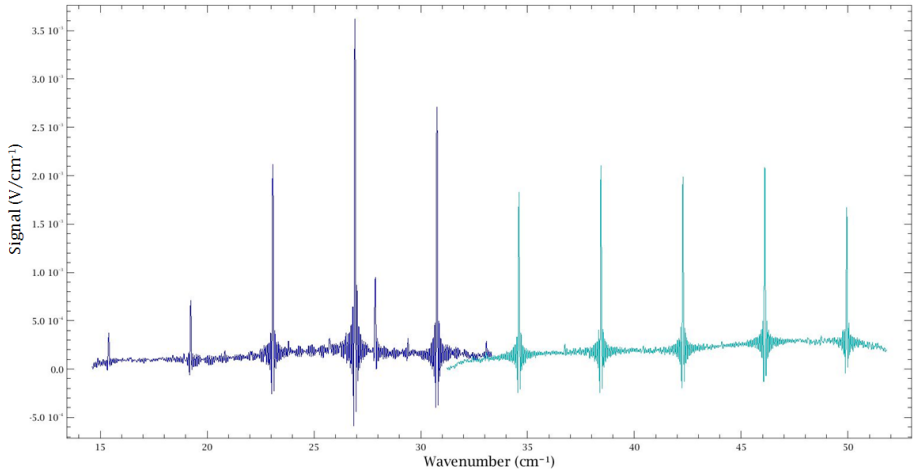


**Figure 4.34. The average spectrum from the source observation.**

## 4.5.3.23. Flux conversion

The following commands correct the flux values for the RSRF and convert them into units of Janskys ($10^{-26}$ W/m$^2$/Hz). The initial conversion applies a correction assuming a uniformly extended source. This is then saved in a copy called "extended". The original data for the central detectors is then further corrected (using the beam parameters calibration product) assuming a point source, see Figure 4.35.

```
ssds = specFluxConversion(ssds, fluxConv=fluxConv)
extended = ssds.copy()
pointSourceSds = specFluxConversion(sds=ssds, fluxConv=fluxConv, \
                                    beamParam=beamParam, APPLY_POINT_SOURCE=1)
```

The variable "extended" is a copy of the ssds product prior to the point source flux conversion, since the pointSourceSds command would overwrite the original ssds variable.

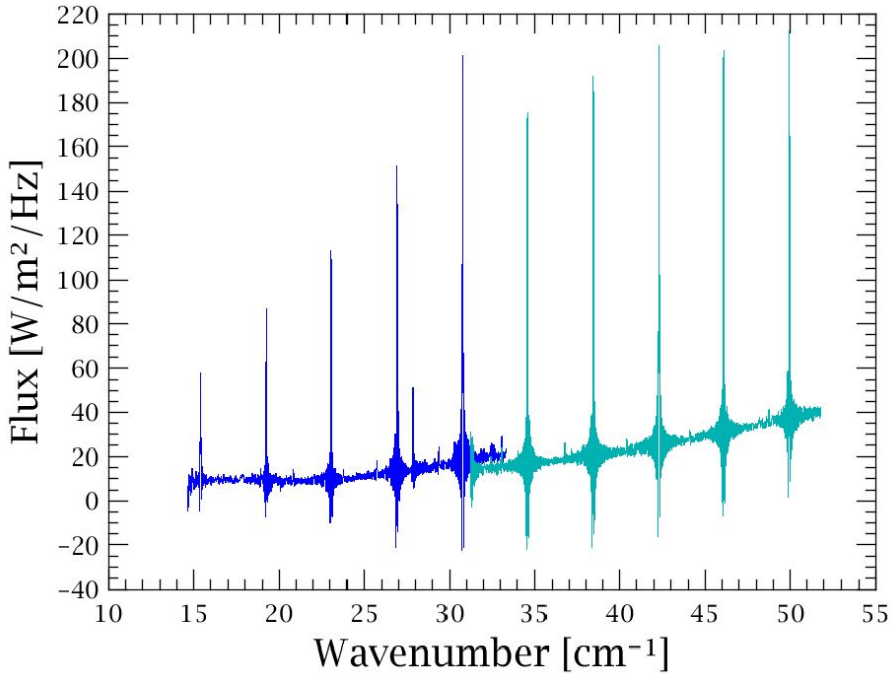


**Figure 4.35. The final, flux-calibrated source spectrum.**

in HCSS v4, the label for the flux unit for point sources is incorrect. Currently, the labeled specifies $W/m^2/Hz$, when in actual fact, the unit should be Jy ($10^{-26}$ $W/m^2/Hz$).

There may still be some effects of incorrect telescope and instrument subtraction (particularly for weaker sources). In the spectrum shown here, there is a slight flattening below 20 $cm^{-1}$ due to a mismatch in instrument temperatures. The step between the signal from the two detector arrays may either be due to incorrect telescope temperature in the reference, or to the source being extended in the beam. These remaining corrections will be addressed in HCSS v5.

### 4.5.3.24. Save the resulting spectra

The following lines store the extended and point source calibrated spectral products into fits files for further analysis.

```
if apodize:
    simpleFitsWriter(extended, -"%s%i_finalSpectrum_extended_apodized.fits"\
                     %(outDir,myObsID))
    simpleFitsWriter(pointSourceSds, -"%s%i_finalSpectrum_point_apodized.fits"\
                     %(outDir,myObsID))
else:
    simpleFitsWriter(extended, -"%s%i_finalSpectrum_extended.fits"\
                     %(outDir,myObsID))
    simpleFitsWriter(pointSourceSds, -"%s%i_finalSpectrum_point.fits"\
                     %(outDir,myObsID))
```

## 4.5.4. The processing script

```
###############################################################################
#  Purpose:  A simplified version of SPIRE SOF1 pipeline script distributed
#            with HIPE 4.0.  This is for data reprocessing by a user using
#            the latest SPIRE calibration products and a user specified background
#            interferogram for telescope and instrument subtraction.
#
#            In addition, the user has the options of (i) processing only
#            the central detectors (SSWD4/SLWC3) to speed up processing and
```

```
#            to lighten the memory load, and (ii) producing either unapodized
#            or apodized spectra.
#
#            The results are two FITS files containing the final spectra with
#            extended-source and point-source flux calibration, respectively.
#
#  Usage:    The user needs to specify the options in the simple user input
#            section at the beginning of the script.
#
#  Updated: 29/07/2010
#
###############################################################################

###############################################################################
# >>>>>>> User_selectable_options:
#
# (A) Specific OBSID and the name of the data storage in your Local Pool:
myObsID    = [obsid]
myDataPool = [pool name]
#
# (B) Only processing the center detector if processOnlyCenterDetectors = 1,
#     or all detector channels otherwise:
processOnlyCenterDetectors = 1
#
# (C) Provide the file name for the dark sky reference interferogram,
myRefInter = -"[Enter path here + filename]"
#
# (D) The final spectrum will be unapodized (if apodize = 0) or apodized (if
#     apodize = 1):
apodize = 0
#
# (E) Specify the output directory for writing the resulting spectrum into a
#     FITS file:
outDir = -"[Enter path here]"
#
# >>>>>>> End_of_user_choices
###############################################################################

# Define some Jython -"methods" (to merge building blocks together):
def mergeNhkt(nhkts):
    for i in range(1, len(nhkts)):
        nhkts[0]['signal'].addRowsByIndex(nhkts[i]['signal'])
        nhkts[0]['mask'].addRowsByIndex(nhkts[i]['mask'])
    nhkts[0].meta['endDate'].value = nhkts[-1].meta['endDate'].value
            return nhkts[0]

def mergeSdis(sdis):
    bigSdi = SpectrometerDetectorInterferogram()
    bigSdi.meta = MetaData(sdis[0].meta)
    scanNumbers = []
    for sdi in sdis:
        scanNumbers.append(sdi.getNumScans())
    if len(sdis) == 1:
        return sdis[0]
    i=0
    for sdi in sdis:
        toAdd = SUM(scanNumbers[i+1:len(scanNumbers)])
        for scanNumber in sdi.getScanNumbers():
            thisScan = sdi.removeScan(scanNumber)
            thisScan.setScanNumber(thisScan.getScanNumber()+toAdd)
            bigSdi.setScan(thisScan)
        i=i+1
    return bigSdi
###############################################################################

# Define the central detectors and thermistors and dark pixels:
detsToKeep = ["SLWC3", -"SSWD4"]
therms = ["SLWT1", -"SLWT2", -"SSWT1", -"SSWT2", \
          -"SSWDP1", -"SSWDP2", -"SLWDP1", -"SLWDP2"]
firstCut = therms
firstCut.extend(detsToKeep)
```

```
# Load in an observation context into HIPE:
storage = ProductStorage(myDataPool)
obs     = storage.select(MetaQuery(ObservationContext,\
             -"p","p.meta['obsid'].value == %iL"%myObsID))[0].product

# get the latest calibration tree relevant to HCSS v4 from the HSA
cal = spireCal(calTree="spire_cal_4_0")
# attatch it to observation context
obs.calibration.update(cal)

# Start to process the observation from Level 0.5
# Process each SMEC scan building block (0xa106) individually, append to a list,
# and then merge.
sdis  = []
nhkts = []
for bbid in obs.level0_5.getBbids(0xa106):
    sdt   = obs.level0_5.get(myObsID, bbid).sdt
    nhkt  = obs.level0_5.get(myObsID, bbid).nhkt
    smect = obs.level0_5.get(myObsID, bbid).smect

    # remove all detectors except the center ones, termistors and dark channels:
    if (processOnlyCenterDetectors):
        for chan in sdt.channelNames:
            if chan not in firstCut:
                sdt.removeColumn(chan)

    # Do the 1st level deglitching:
    sdt = waveletDeglitcher(sdt,reconstructionPointsAfter=3, \
                            reconstructionPointsBefore=2, \
                            correctGlitches=Boolean.TRUE,\
                            scaleMin=1, scaleMax=8, scaleInterval=5,\
                            holderMin=-1.4, holderMax=-0.6,\
                            correlationThreshold=0.85,\
                            optionReconstruction="polynomialFitting",\
                            degreePoly=6, fitPoints=8)

    # Run the Non-linearity and Temp Drift correction steps
    sdt = specNonLinearityCorrection(sdt, \
                              nonLinCorr=obs.calibration.spec.nonLinCorr)
    sdt = temperatureDriftCorrection(sdt, \
                              tempDriftCorr=obs.calibration.spec.tempDriftCorr)

    # Now also remove thermistors and dark pixels if the user wants process
    # the central detectors only:
    if (processOnlyCenterDetectors):
        for chan in sdt.channelNames:
            if chan not in detsToKeep:
                sdt.removeColumn(chan)

    # Do clipping repair if needed:
    sdt = clippingCorrection (sdt)

    # Time domain phase correction:
    sdt = timeDomainPhaseCorrection(sdt,\
                               lpfPar=obs.calibration.spec.lpfPar,\
                               chanTimeConst=obs.calibration.spec.chanTimeConst)

    # Add pointing info:
    bat = calcBsmAngles(nhkt, bsmPos=obs.calibration.spec.bsmPos)
    spp = createSpirePointing(hpp=obs.auxiliary.pointing, siam=obs.auxiliary.siam, \
                              detAngOff=obs.calibration.spec.detAngOff, bat=bat)

    # Create interferogram:
    sdi = createIfgm(sdt=sdt, smect=smect, nhkt=nhkt, spp=spp, \
                     smecZpd=obs.calibration.spec.smecZpd,\
                     chanTimeOff=obs.calibration.spec.chanTimeOff,\
                     smecStepFactor=obs.calibration.spec.smecStepFactor,\
                     interpolType= -"spline")

    # Append this building block to the list:
    sdis.append(sdi)
    nhkts.append(nhkt)
```

```
# Merge all the building blocks into one:
sdi  = mergeSdis(sdis)
nhkt = mergeNhkt(nhkts)

# Baseline correction and 2nd-level deglitching:
sdi = baselineCorrection(sdi, type="fourier", threshold=4)
sdi = deglitchIfgm(sdi, deglitchType="MAD", thresholdFactor=4)

# Subtract a background in the interferogram domain:
interRef = fitsReader(myRefInter)
sdi = telescopeScalSubtraction(sdi, interRef=interRef, nhkt=nhkt)

# Phase correction:
presdi = apodizeIfgm(sdi, apodType="prePhaseCorr", apodName="aNB_20")
dsds   = fourierTransform(sdi=presdi, ftType="prePhaseCorr", zeroPad="None")
sdi    = phaseCorrection(sdi, sds=dsds, \
                         polyDegree=2, pcfSize=127, \
                         nlp=obs.calibration.spec.nlp, \
                         phaseCorrLim=obs.calibration.spec.phaseCorrLim)

# Creat the apodized interferogram:
if apodize:
    sdi = apodizeIfgm(sdi, apodType="postPhaseCorr", apodName="aNB_15")

# Fourier transform to the spectral domain:
ssds = fourierTransform(sdi, ftType="postPhaseCorr", zeroPad="standard")

# Get the flux conversion calibration products:
if apodize:
    fluxConv = obs.calibration.spec.fluxConv
    beamParam = obs.calibration.spec.beamParamList.getProduct(1, -"nominal", \
                                                               obs.startDate)
else:
    fluxConv = obs.calibration.spec.fluxConvList.getProduct("HR", -"unapod", \
                                                        -"nominal", obs.startDate)
    beamParam = obs.calibration.spec.beamParamList.getProduct(0, -"nominal", \
                                                               obs.startDate)

# Average scans together and apply flux conversion:
ssds = averageSpectra(ssds, separateScanDirections=0, \
                      INCLUDE_OOB=0, bandEdge=obs.calibration.spec.bandEdge)
ssds = specFluxConversion(ssds, fluxConv=fluxConv)

# keep a copy of the level-1 spectra (extended source calibration):
extended = ssds.copy()

# Also apply point-source flux calibration:
pointSourceSds = specFluxConversion(sds=ssds, fluxConv=fluxConv, \
                                    beamParam=beamParam, APPLY_POINT_SOURCE=1)

# Save the final spectra to FITS (both extended and point source calibrated):
if apodize:
    simpleFitsWriter(extended, -"%s%i_finalSpectrum_extended_apodized.fits"\
                     %(outDir,myObsID))
    simpleFitsWriter(pointSourceSds, -"%s%i_finalSpectrum_point_apodized.fits"\
                     %(outDir,myObsID))
else:
    simpleFitsWriter(extended, -"%s%i_finalSpectrum_extended.fits"\
                     %(outDir,myObsID))
    simpleFitsWriter(pointSourceSds, -"%s%i_finalSpectrum_point.fits"\
                     %(outDir,myObsID))

### End of the script
```