The HIFI User's Manual

Hifi Editorial Board:

Max Avruch

Adwin Boogert

Tony Marston

Carolyn McCoey

Michael Olberg

Miriam Rengel

Russ Shipman

The HIFI User's Manual

Hifi Editorial Board:
Max Avruch
Adwin Boogert
Tony Marston
Carolyn McCoey
Michael Olberg
Miriam Rengel
Russ Shipman

Table of Contents

Chapter 1. Data Primer

A short introduction to the structure of Herschel HIFI data storage.

1.1. Data frames

The Herschel spacecraft stores data onboard (up two days' worth) until transmited to Earth. Science data, such as a WBS spectrometer readout, come naturally in sets, or Frames. Data frames are packetized for transmission from HSO to Earth. Along with House Keeping (HK) data they are downlinked to the tracking station and thence to the Mission Operation Center (MOC) at ESOC in Darmstadt, or to the latter directly. The data packets then flow from the MOC to the Herschel Science Center (HSC) at ESA's European Space Astronomy Centre (ESAC) in Madrid. The HIFI ICC copies the data from HSC, as well.

At ESAC, the data packets are 'ingested' into a database and the science data frames are reconstituted.

The combination of HK and science data creates a 'Level 0 Observational Data Product.'

1.2. Data Products

refs: Herschel_Data_Product_Document_partI_v0.95.pdf, [ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/releases/doc/ia/pal/doc/guide/html/pal-guide.html]

A Herschel Data Product consists of metadata keywords, tables with the actual data, and the history of the processing that generated the product. There are various product types (Observation, Calibration, Auxiliary, Quality Control, User Generated). The types of Observation Data Products:

1. Level -1: Raw data packets, separate HK and science frames as described above.

2. Level 0: HK and science frames grouped by time and building block ID (and perhaps other parameters?). As close to raw data as the as the typical user would find useful to be.

3. Level 0.5: data processed to an intermediate point adequate for inspection; for HIFI they are processed such that backend (spectrometer) effects are removed, essentially a frequency calibration.

4. Level 1: Detector readouts calibrated and converted to physical units, in principle instrument and observatory independent; for HIFI, essentially an intensity calibration. It is expected that Level 1 data processing can be performed without human intervention.

5. Level 2: scientific analysis can be performed. These data products are at a publishable quality level and should be suitable for Virtual Observatory access.

6. Level 3: These are the publishable science products with level 2 data products as input. Possibly combined with theoretical models, other observations, laboratory data, catalogues, etc. Formats should be Virtual Observatory compatible and these data products should be suitable for Virtual Observatory access.

1.3. Contexts

A Context is a subclass of Product, a structure containing references to Products and necessary metadata. A Context can contain Contexts, giving rise to Context 'trees.' Types:

1. ListContexts (for grouping products into sequences or lists, hardly used)

2. MapContexts (for grouping products into key,value dictionaries)

1.3.1. Herschel Observation Context

A MapContext instance serves as the organisational product unit for the Herschel Data Processing system. It contains the following contexts:

1. Level-0, Level-0.5, Level-1, Level-2, & Level-3(optional) Contexts
2. Calibration Context
3. Auxiliary Context
4. Quality Context
5. Browse product
6. Trend Analysis Context
7. optional Telemetry Context: not by default, only when the HSC deems it necessary because of a serious problem in the processing to level-0 data.

The uses of these Contexts will be described in Chapter 2.

Note that the descriptive modifiers "Product" and "Context" are often dropped conversationally.

Chapter 2. Running the HIFI pipeline

HIFI data is automatically processed through the HIFI pipeline before it can be accessed from the the Herschel Science Archive (HSA). The HIFI pipeline is used for processing data received from one or more of the four HIFI spectrometers into calibrated spectra or spectral cubes that are suitable for interactive analysis. The pipeline comprises four stages of processing:

1. Take data from the satellite and minimally manipulate it into time ordered Data Frames (a HifiTimeline, or HTP, for each spectrometer). This is a Level 0 data Product, which is the least processed data available to Astronomers.

2. Remove backend instrumental effects - essentially a frequency calibration. There are separate pipelines for the WBS and HRS spectrometers, and the result is a Level 0.5 Product.

3. Application of observing mode specific calibrations, i.e., subtraction of reference and off positions and intensity calibration using Hot/Cold loads. This is done by the Level 1 pipeline and resulting Level 1 Products are sets of frequency and intensity calibrated spectra.

4. The Level 2 pipeline removes further instrumental effects, such as standing waves, baseline slopes and offsets. Spectra are deconvolved and, depending on the observing mode, averaged or gridded into spectral cubes.

It is expected, especially in the early stages of the mission, that Level 2 products will need to be regenerated interactively by the Astronomer. Indeed, you may wish to re-run all or part of the pipeline to change defaults, use your own, or examine each step of processing. To that end, the ObservationContext that is obtained from the HSA contains, along with the Level 0-2 data Products, everything you need to reprocess your observations - calibration products, satellite data - as well quality, trend analysis, logging, and history products, which you can use to identify any problems with your data or its processing.

The following section explains how to re-run the pipeline using the HifiPipeline task. There are some aspects to using the HifiPipeline task that are only likely to be useful for calibration scientists and software engineers when looking at test or calibration data, and those are covered in the section after. However, if you get very involved in running the pipeline you may find some of this section to be useful.

The final section of this chapter describes your options when stepping through the pipeline (i.e., running each step manually) up to Level 1. Further processing steps, such as removal of standing waves or making cubes, have dedicated chapters elsewhere in this manual.

2.1. How to run the HifiPipeline task for Astronomers

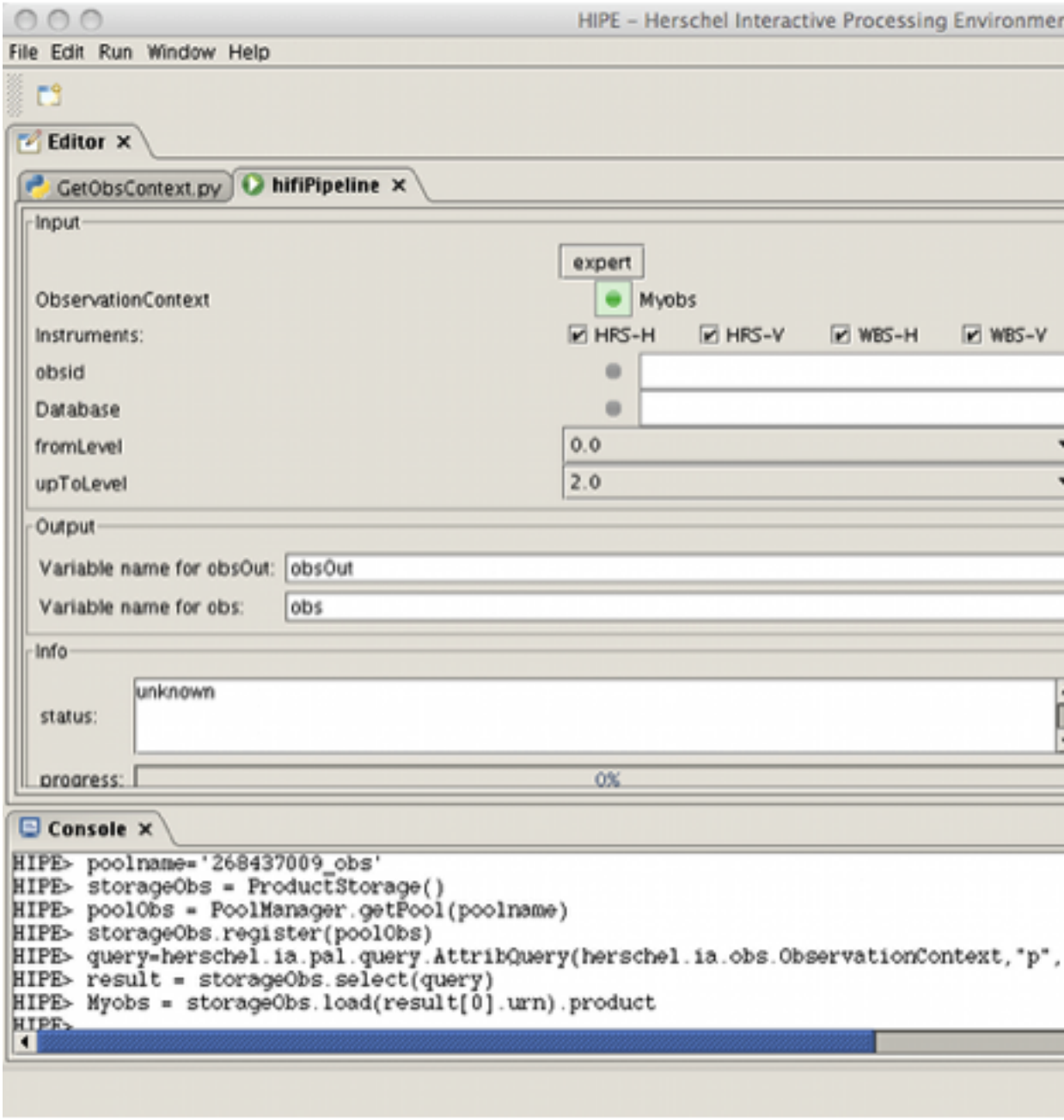
The HifiPipeline task links together the four stages of the pipeline described above and it can be used to reprocess ObservationContexts up to any Level, for any choice of spectrometer(s) and polarisation(s). You can also make your own algorithms - or modify the ones provided in the **`scripts/hifi/Pipeline`** directory in the installation directory of HIPE - and apply them to the pipeline. The HifiPipeline task can, of course, be run both from the GUI and the command line: we deal with the GUI first.

2.1.1. HifiPipeline task in the GUI

The HifiPipeline task is run from the GUI in the following fashion:

1. • Click once on an Observation Context in the Variables pane and the "hifiPipeline" Task will appear in the "Applicable Tasks" folder, double click on it to open the Task dialogue in the Editor view.

- Alternatively, open the "hifiPipeline" Task by double-clicking on it under the Hifi Category in the Tasks view.
- A "Hifi Pipeline" View is also available from the HIPE Window menu (under Show View) but it is not fully implemented yet.



The hifiPipelineTask appears in the "Applicable" Folder in the Tasks view after clicking on the Observation Context (MyObs) in the variable view.

Figure 2.1. HIFI pipeline task: default view

2. The default (or basic) dialogue allows you to re-process an already existing observation context, e.g. from the Herschel Science Archive, through the pipeline.

   - The way the data is to be reprocessed is defined in the Input section:

   a. If the hifiPipeline Task was opened from the "Applicable Tasks" folder then the Observation Context selected in the Variables View will automatically be loaded into the Task dialogue, and you will see its name by the observation context bullet, which will be green. Alternatively, drag the name of the observation context to be reprocessed from the Variables view to the observation context bullet.

   b. Select the spectrometers you wish to process data for by checking the desired instrument(s) and polarisation(s). Both H and V polarisations of both the Wide Band Spectrometer (WBS) and High Resolution Spectrometer (HRS) are checked by default.

   c. Select which levels to (re-)process from and to via the drop-down menus. By default the pipeline will process level 0 data up to level 2. If you are accessing data from the ICC database, you can process raw data (option -1). Data taken from the Herschel Science Archive (HSA) can be re-processed from level 0 (option 0) to levels 0.5 (option 0.5), 1 (option 1), or 2 (option 2)

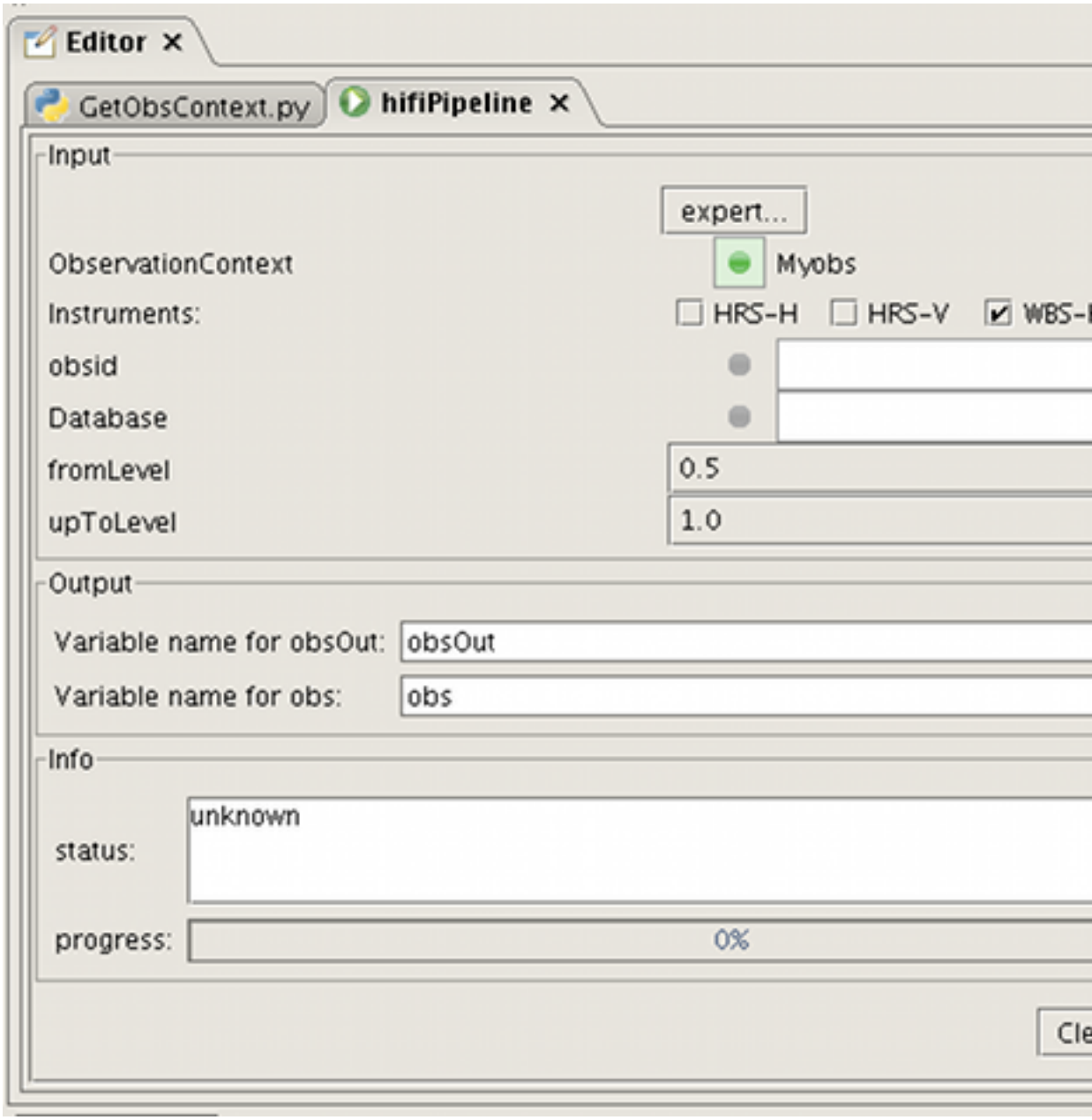
      If you try to re-process from a higher Level data than exists in the Observation Context then the hifiPipelineTask will automatically select the highest existing Level. For example, if you try to re-process from Level 0.5 to 1 but the ObservationContext only contains a Level 0 product then the pipeline will automatically run from Level 0 to Level 1.

   d. Ignore the entries for database and obsid; these can only be used by calibration scientists and should be removed in the future.

   e. At the moment, to supply your own algorithm to the pipeline (see Section 2.4) you must toggle to the expert view, where you can load a new algorithm from file. The ways you might want to modify the pipeline algorithms are discussed in Section 2.3. See the notes below about customizing pipeline algoriths.

   - In the Output section, choose the name of the observation context that will be produced or use the HIPE default, `obsOut`. The observation context contains all the products generated by the pipeline task and should be stored in a pool in your lstore (~/.hcss/lstore/hifi-pipeline) to be able to access it in a future session. The variable `obs` is also produced in order that the pipeline can be re-run without the need to reset IO parameters.

   - Click on "accept" to run the pipeline. The status ("running" if all is well, error messages if not) and the progress of the pipeline are given in the Info section at the bottom of the Task dialogue.



In this example, an already processed observation, 'Myobs', is being re-processed from Level 0.5 up to Level 1, both polarisations of the WBS spectrometer have been selected.

Figure 2.2. HIFI pipeline task: default view

2.1.2. HifiPipeline in the command line

Below are some examples of running the HifiPipeline task from the command line, once again it is assumed that an ObservationContext called `Myobs` has been loaded into the session.

```
# Reprocess an ObservationContext up to Level 2 for all spectrometers
MyNewobs = hifiPipeline(obs=Myobs)
#
# Reprocess Myobs from Level 0.5 to Level 1, for all spectrometers
MyNewobs = hifiPipeline(obs=Myobs, FromLevel=0.5, UpToLevel=1)
#
```

```
# Now reprocess MyNewobs (which now contains data only up to Level 1) but only for
the WBS.
# WBS-H and WBS-V are the horizontal and vertical polarizations, respectively.
MyEvenNewerobs = hifiPipeline(obs=MyNewobs, apids=['WBS-H', -'WBS-V'])
#
# Reprocess Myobs from Level 0 to Level 0.5 for only horizontal polarization data
MyNewobs = hifiPipeline(obs=Myobs, apids=['WBS-H','WBS-V'], FromLevel=0,
UpToLevel=0.5)
#
# Now include your own algorithm for the Level 1 pipeline, for all spectrometers,
from Level 0 to 1
MyNewobs = hifiPipeline(obs=Myobs, FromLevel=0, UpToLevel=1, level1Algo=$full_path/
mylevel1Algo.py)
#
# Now the pipeline went wonky and you want to reset it!
hifiPipeline = hifiPipelineTask()
#
```

- The exact ordering of the arguments does not matter.
- What is an apid? "Application Program IDentifier": it is what the pipeline calls spectrometers.
- Note that to implement your own algorithm, you must load the algorithm script from wherever you saved it into HIPE and compile it (run it with >>) before you run the pipeline (see 2.4).

2.2. HifiPipeline tasks for Calibration Scientists

2.2.1. Expert hifiPipeline task

By toggling the "expert" button on the hifiPipeline task GUI, you may additionally control more detailed aspects of the pipeline set-up. There are code examples of their usage below.

Unless you specify otherwise when running the pipeline, it will look for the database (db), calibration pool (and aux pool for flight data) that you specify in your user.props file. See Section 14.2 for information about configuring your setup and the ICC databases.

Note that in the case that a database has been regenerated, you will need to delete your /.hcss/pal_cache in order to run the pipeline or generate an HTP. This is worth trying if you have difficulties after any major changes in database or configuration.

```

# Access the ICC simulations database and run the pipeline to generate an
observation context ('obs')
from scratch
obs = hifiPipeline(obsid=268435702, db="ds3@iccdb2.sron.rug.nl 0 READ")
#
# If the obsid is 10 integers long (i.e. after SOVT) then append an L
obs = hifiPipeline(obsid = 3221226279L, db="sovt2_fm_1_prop@iccdb1.sron.rug.nl 0
READ")
#
# Process for a selection of spectrometers
obs = hifiPipeline(obsid = 3221226279L, db="sovt2_fm_1_prop@iccdb1.sron.rug.nl 0
READ",
apids=['WBS-H','HRS-V'])
#
# Select the level to process to (from raw (-1) Level 1 (1) here)
obs = hifiPipeline(obsid = 3221226279L, db="sovt2_fm_1_prop@iccdb1.sron.rug.nl 0
READ",
FromLevel=-1, UpToLevel=1)
#
# If there is an ObsContext (Myobs) in the session, you can reprocess it from an
existing Level,
```

```
e.g Level 0.5 to 1
obs = hifiPipeline(obs = Myobs, FromLevel=0.5, UpToLevel=1)
#
# If the calibration tree is updated, you can re-populate the calibration products
in Myobs using:
obs = hifiPipeline(obs=Myobs, cal=1)
#
#For ILT data, provide the obsMode name -- the data itself does not have it:
obs = hifiPipeline(obsid=268516902, db="ilt_fm_5_prop@iccdb1.sron.rug.nl 0 READ",
obsMode="HifiPointModeLoadChop")
#
# Use a different tmVersion than the default in your user.props file
obs = hifiPipeline(obsid=268439922, db="ilt_par_5_prop@iccdb.sron.rug.nl 0 READ",
tmVersion="ilt-par")
#
# You can edit the algorithm of the pipeline tasks and use them in place of the
default algorithms,
# see Section 2.4
obs = hifiPipeline(obsid=1342179306L, db="hifi_icc_ops_1@iccdb1.sron.rug.nl 0
READ", wbsAlgo=myWbsAlgo,
hrsAlgo=myHrsAlgo,Level1Algo=myLevel1Algo)
#
# Provide your own palStore to which the pipeline will write to:
obs = hifiPipeline(obsid=1342179306L, db="hifi_icc_ops_1@iccdb1.sron.rug.nl 0
READ", palStore = myStore)
#
# Clear CachedStoreHandler to avoid a block due to none closed stores.  Note, this
closes ALL stores
# available in this cache and may affect other applications running in the session.
obs = hifiPipeline(obsid=1342179306L, db="hifi_icc_ops_1@iccdb1.sron.rug.nl 0
READ", clearCachedStoreHandler=1)
#
# If pipeline task is not behaving as you expect you could try a reset:
from herschel.hifi.pipelin import HifiPipelineTask
hifiPipeline = HifiPipelineTask()
```

2.2.2. Individual pipeline tasks

In addition to using the HifiPipeline task, one can run the underlying pipeline tasks too. These tasks handle ObservationContexts as well as HTPs, which can make using them more efficient to test one stage of the pipeline. The API for each of the tasks is identical and their functionalities are summarised in the examples below. GUI forms are also available.

Note

The Generic pipeline task is deprecated and the Level 1 and Level 2 pipeline tasks should be used instead

```
# Create a Level 0.5 HTP for WBS-H from raw data
htp=wbsPipelineTask(obsid=3221226570L, apid=1030,
db="sovt2_fm_1_prop@iccdb1.sron.rug.nl 0 READ")
#
# and then pass that to the Level 1 pipeline task
htp=level1PipelineTask(htp=htp, apid=1030)
#
# Create a Level 0.5 HTP for the HRS-H from an existing HTP, using your own
algorithm
htp=hrsPipelineTask(htp=htp, apid=1028 -, algo=myHrsAlgo)
#
# and pass that to the Level 1 pipeline, and defining your own palStore to write
the output to
htp = level1PipelineTask(htp=htp, apid=1028, palStore= myStore)
#
# Pass an existing ObservationContext to the Level2PipelineTask
obs = level2PipelineTask(obs=obs)
```

2.3. Running the Pipeline step by step

- Running the pipeline (or one part of the pipeline) step by step allows you to inspect the results of each step and change the default parameters of the pipeline. If you wish to create your own algorithm (which must be written in jython) for a part of the pipeline, then this will likely be your first step.
- It is not expected that there will be much need to customise the spectrometer pipelines (up to Level 0.5) and indeed there are only a steps of the spectrometer pipelines that have some options. It is more likely that you may wish to play with how off and reference spectra are subtracted in the Level 1 pipeline, although it is expected that the dafault settings should work well.
- To step through the pipeline you must work directly on the appropriate level HifiTimeLine (HTP - the dataset containing all the spectra, including calibration spectra, made during an observation for a given spectrometer). So the first thing you must do is extract the HTP you want to work on from your ObservationContext, see Section 14.6 for how to do this.
- When you select an HTP in the Variables view in HIPE you will notice that many tasks with names like DoWbsDark, mkFreqGrid. These are the names of all of the steps in the HIFI pipeline; mk... signifies a step where a calibration product is being made, Do... is a step where a calibration is applied. You can step through the pipeline using these tasks or (more efficiently) use and modify the scripts that are supplied with the software in the **`scripts/hifi/Pipeline`** directory in the installation directory of HIPE
- For information on the steps of each level of the pipeline (their names, the order to run them in, and what options you can change) see the HIFI Pipeline Specification document in the help, also here if you have access to the HIFI ICC pages.

2.4. How to customise pipeline algorithms

1. The pipeline algorithm scripts can be found in:
   - **WBS.** $BuildDir/scripts/hifi/pipeline/wbs/WbsPipelineAlgo.py
   - **HRS.** $BuildDir/scripts/hifi/pipeline/hrs/HrsPipelineAlgo.py
   - **Level 1.** $BuildDir/scripts/hifi/pipeline/generic/Level1PipelineAlgo.py
   - **Level 2.** $BuildDir/scripts/hifi/pipeline/generic/Level2PipelineAlgo.py
2. Open the algorithim you wish to customise in the editor, edit it (and save!)
3. Compile your algorithm by running the script with >>
4. Apply the algorithm to the pipeline as described in the sections above.

Chapter 3. Flags in HIFI data

Flags (also called masks) are identifiers of specific issues with the data, such as saturated pixels or a possible spur, that can affect the quality of the final product. Flags are used to identify affected data and to make a caution during its processing.

A Flag has a defined name and a value, which specifies the nature of the flag. The flags are divided into two categories, depending on whether they apply to an individual channel (pixel), or to a complete Dataframe. They are called *channel f lags*, and *column rowflags*, respectively.

Note

There are also *Quality Flags*, which are found in the Quality Product in the ObservationContaxt and are used to provide you with means to make a quick assessment of the quality of your data, they are discussed in chapter Chapter 4

3.1. Channel flags

Channel (or pixel) flags apply to individual pixels and are added as a column in the HTP. Their names are also added to the metadata of a dataset during processing and this is used for the history of the pipeline; it also means that you can tell that, e,g., the WBS pipeline has been applied if you see things like "isMasked" and "checkZero" in the metadata.

For each pixel there are 32 flags which can be set, currently 8 are defined, and the definition of the mask bits and values in HIFI data is given here:

Flag Name	Value	Description
Bad pixel	0	If this bit is set, the sample contains a bad pixel
Saturated pixel	1	If this bit is set, the sample was saturated
Not observed	2	If this bit is set, the sample is not observed
Not Calibrated	3	If this bit is set, the sample is not calibrated
In overlap region	4	If this bit is set, the sample is in the subband overlap region. I.e. it can be seen better in the adjacent subband.
Glitch detected	5	If this bit is set, the sample is not observed
Dark pixel	6	If this bit is set, the sample is used to measure the dark
Spur candidate	7	If this bit is set, the sample is a candidate to be a spur. It is a 'candidate' since not all things flagged by the spurfinder are necessarily spurs

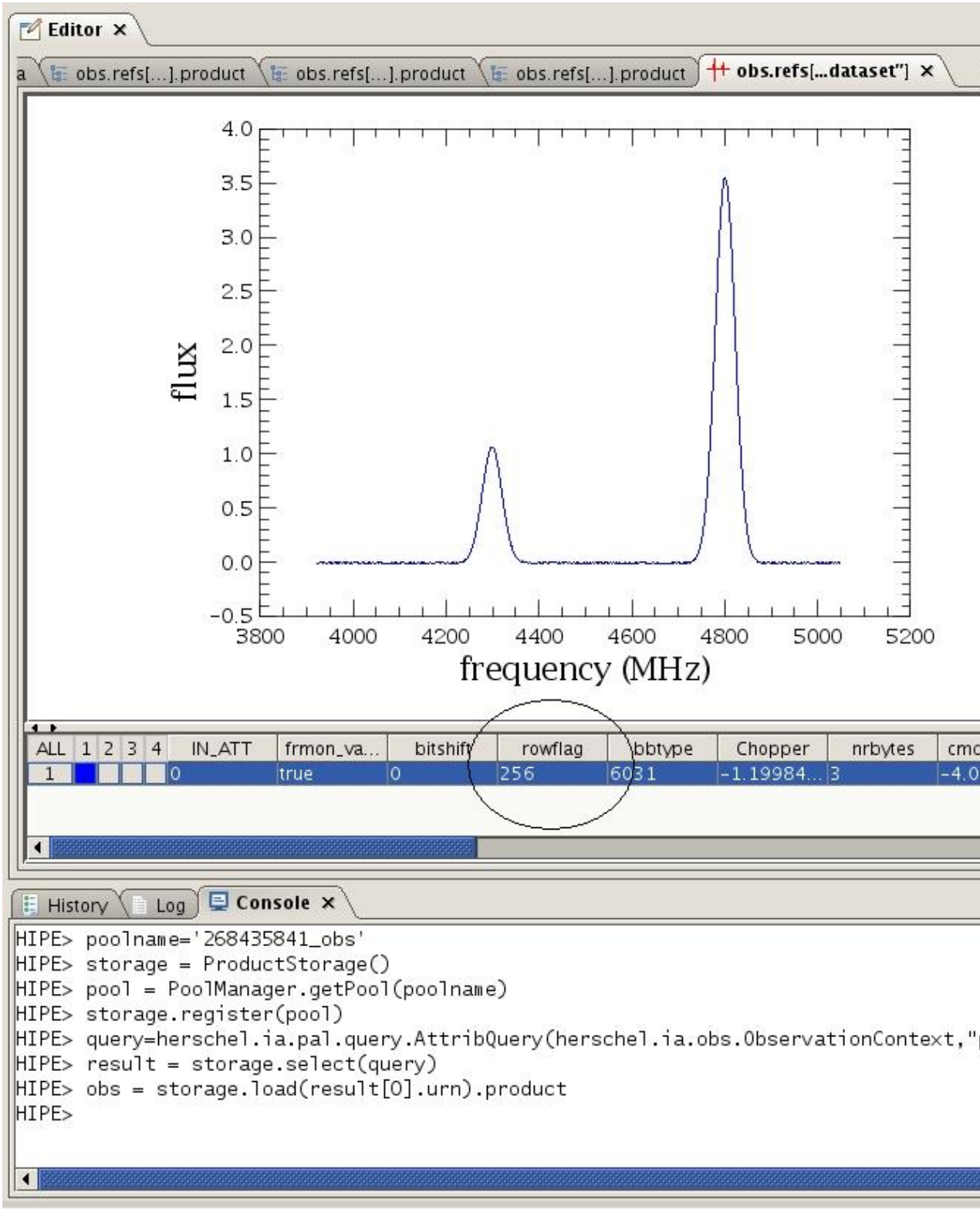
3.2. Column rowflags

Column rowflags (the "rowflag" column in the HIFI spectrum TableDataset) apply to the complete Dataframes (DF) or rows in a HifiSpectrumDataset (HSD).

For bit n the value is computed according to $value=2^{(n-1)}$. The first 5 bits are about the packets from which the DataFrame (DF) is reconstructed, and are unlikely to ever occur. Below is a table showing the current names and values of HIFI rowflags:

Flag Name	**Bit**	**Value**	**Description**
PacketOrder	1	1	Error in the packet order while constructing the DataFrame
PacketLength	2	2	Error in the packet length while constructing the DataFrame
TooMuchData	3	4	More data than can be fit in a DataFrame
FirstPacket	4	8	Error in the start packet while constructing the DataFrame
NoBlocks	5	16	No block information present while constructing the DataFrame
spare	6	32	
spare	7	64	
UnalignedHK	8	128	HK could not be aligned with DataFrames. When the columns "df_transfer" and "hk_transfer" in the TableDataset are different, bit 8 is set
noChopper	9	256	No valid Chopper information. Set when the flagbit is zero in the DFs, extracted from the HK packets if possible
noComChop	10	512	No valid Commanded Chopper information. Set when the flagbit is ero in the DFs, extracted from the HK packets if possible
noFreqMon	11	1024	No valid Frequency Monitor information. Set when the flagbit is zero in the DFs, extracted from the HK packets if possible
noLoCodeOffset	12	2048	No valid LO code offset information. Set when the flagbit is zero in the DFs, extracted

Flag Name	Bit	Value	Description
			from the HK packets if possible
noLoCodeMain	13	4096	No valid LO code main information. Set when the flagbit is zero in the DFs, extracted from the HK packets if possible.
BbidCorrection	14	8192	Correction of Bbid, see SPR 1963. Not relevant any more. It was during SOVT testing, but the onboard software has been corrected since
MixerCurrentDeviation	15	16384	Difference in mixer currents exceeds tolerance when applying DoRefSubtract.
MixerCurrentDeviation	16	32768	Difference in mixer currents exceeds tolerance when applying DoOffSubtract.
MixerCurrentDeviation	17	65536	Difference in mixer currents exceeds tolerance when applying DoFluxHotCold or MkFluxHotCold.
NoHotColdCalibration	18	131072	Division by the bandpass has not been carried through



Caption: Example of a HIFI spectrum TableDataset, which contains the "rowflag" column with a value of 256.

Chapter 4. Quality Flags

Quality Flags are raised during standard processing of HIFI data. Flags should be created from every processing step of the pipeline, from the initial creation of the HifiTimelineProduct (Level 0), through to the final product of Level2 processing. If all goes well, the flags will have their default values but if a certain processing step is unable to perform the action it was designed for the flag will take a different value. If the pipeline produces a flag other than the default value, this flag is promoted to the Quality Report. Thus the quality report is by definition a list of things identified as have gone wrong. A quality report is found from the ObservationContext:

```
obs.refs["quality"].product
```

Please note the difference between a quality flag and flagging data. In flagging data you identify that, for example, a given channel sample is saturated; if those channels are saturated repeatedly during the observation then the quality flag "SATURATEDNUMBER" wil be raised.

Below is a list of the current available types of quality flags for the HIFI pipeline, for each level. The format below gives flag name, flag description, and flag default value.

Level O Quality Flags.

Quality Flags
UNALIGNED_HK("unalignedHKdata","Percentage of Dataframes which have unaligned HK", 0.0)
NOCHOPPER("noChopperHKdata","Percentage of DFs having no chopper information", 0.0)
NOCOMCHOP("noCommandedChopperHKdata","Percentage of DFs having no commanded chopper information", 0.0)
NOFREQMON("noFrequencyMonitorHKdata", "Percentage of DFs having no frequency monitor information", 0.0)
NOLCOFFS("noLoCodeOffsetHKdata","Percentage of DFs having no LO Code offset information", 0.0)
NOLCMAIN("noLoCodeMainHKdata","Percentage of DFs having no LO Code main information", 0.0)
BBID_CORRECTION("bbidCorrection","Percentage of Bbids corrected according to commanded Bbids", 0.0)
DATAFRAMES_OUTOFORDER("dataframesOutOfOrder","Unordered or duplicate Dataframes found", false)
MISSING_DATA("missingData","Less data found than expected", false)
SURPLUS_DATA("surplusData","More data found than expected", false)

Level 0.5 Quality Flags: WBS.

Quality Flags
COMBFLAG(QWbsFreq.VALIDATE,"Flag for all COMB of the observation",false)
ZEROFLAG(QWbsZero.VALIDATE , "Flag for all Zero of the observation",false)
SPIKENUMBER(QWbsSpikes.NUMBER, "Maximum number of spikes detected in a Comb", 0)
SATURATEDNUMBER("pixelSaturated","Maximum number of saturated pixel detected in a single spectrum",0l)
SDARKFLAG("darkFlag","Spectrum contains saturated dark ",false)
BADPIXELS("badPixels","Number of channels marked as BAD due repeated saturations",0l)

Level 0.5 Quality Flags: HRS.

Quality Flags
NOQDC("noQDC", "No Quantization Distortion Correction could be processed.",false)
FASTQDC("fastQDC", "Fast Quantization Distortion Correction processed. Not optimal.",false)
NOPOWCOR("noPowerCorrection","No Power Correction could be processed.",false)

Level 1.0 Quality Flags.

Check data structure.

Quality Flags
OBSERVINGMODE("observingMode","Observing mode not recognized - consult the pipeline configuration xml file.", false)
UNKNOWNBBTYPE("unknownBbType","Bbtype not known.", false)

Check freq grid.

Quality Flags
FREQUENCYDRIFT("maxFreqDrift", "Unacceptable maximum drift in the frequency grid detected.", false)
FREQUENCYCHECKS("noFreqChecks", "Frequency checks and/or frequency grouping failed.", false)

Check phases.

Quality Flags
CHOPPERPATTERN("chopperPattern", "Pattern observed for the Chopper not as expected in all datasets.", false)
CHOPPERVALUES("chopperValues", "Number of distinct Chopper values not as expected in all datasets.", false)
LOFPATTERN("lofPattern", "Pattern observed for the LoFrequency not as expected in all datasets.", false)
LOFVALUES("lofValues", "Number of distinct LOF values not as expected in all datasets.", false)
BUFFERPATTERN("bufferPattern", "Pattern observed for the buffer not as expected in all datasets.", false)
BUFFERVALUES("bufferValues", "Number of distinct buffer values not as expected in all datasets.", false)
PHASECHECKS("noPhaseChecks", "Not all phase checks could not be carried through or completed.", false)

Hot/cold-calibration.

Quality Flags
HOTCOLDDATA("hotcoldData","Data measured from hot and cold loads not sufficient for hot/cold calibration.", false)
TSYSFLAG("tsysFlag","Hot/cold calibration not successful.", false)
INTENSITYCALIBRATION("intensityCalibration", "Intensity calibration not or not for all spectra carried through.", false)

Channel weights.

Quality Flags
CHANNELWEIGHTSFLAG("channelWeights","Problem occurred while computing channel-dependent weights. No weights added.", false)

Reference subtraction.

Quality Flags
REFSUBTRACTIONFLAG("refSubtraction", "Reference subtraction not processed - maybe identification of phases not successful.", false)

Off smooth.

Quality Flags
NOOFFBASELINE("noBaseline", "No off baseline could be calculated.", false)

Off subtraction.

Quality Flags
ONOFFSEQUENCE("onoffSequence","ON/OFF datasets not in expected sequence (...-ON-OFF-ON-OFF-... or ...-ON-OFF-OFF-ON-ON-....", false)
ONOFFPAIRSIZE("onoffLength", "Some ON/OFF dataset pairs found with unequal number of rows.", false)
ONOFFPROCESSING("onoffProcessing", "More ON- than OFF-datasets found in the data - not all ON-datasets could be processed with OFF-dataset(s).", false)
OFFBASELINESUBTRACTION("offBaselineSubtraction", "No off baseline subtraction carried through since no off baseline data available.", false)
DATALOSSINAVERAGE("average", "Some data has been lost while computing the average over many datasets.", false)

Chapter 5. Viewing Spectra

5.1. Introduction

HIFI spectra can be visualised in several ways, at various levels of sophistication and user-friendliness. Here the PlotXY and SpectrumExplorer packages are described.

5.2. Basic Spectrum Viewing: the PlotXY Package

PlotXY() is the basic package to plot arrays of data points in the HCSS, and it can be used to plot HIFI spectra as well. It has a lot of options, making the plots highly configurable. Here is an example of plotting a HIFI spectrum:

- Get the frequency and flux data to be plotted from the spectrumdataset 'sd':

  ```
  freq=sd.getWave().get(0)

  flux=sd.getFlux().get(0)
  ```

- The simplest possible plot:

  ```
  out=PlotXY(freq, flux)
  ```

- When plotting multiple spectrum datasets, say 'sd1' and 'sd2' in one figure:

  #get the wavelengths and fluxes to be plotted

  ```
  freq1=sd1.getWave().get(0)

  flux1=sd1.getFlux().get(0)

  freq2=sd2.getWave().get(0)

  flux2=sd2.getFlux().get(0)
  ```

  #create the plot variable

  ```
  p=PlotXY()
  ```

  #create the plots in batch mode

  ```
  p.batch=1
  ```

  #define the layer variable

  ```
  ll=[]
  ```

  #remove any non-numbers (NaN's, Infinites etc.)

  ```
  valid=flux1.where(IS_FINITE)
  ```

  #create layer for first plot

  ```
  l=LayerXY(freq1[valid],flux1[valid])
  ```

  #append to layer variable

ll.append(l)

#repeat the above for the 2nd plot to be overlaid

valid=flux2.where(IS_FINITE)

l=LayerXY(freq2[valid],flux2[valid])

ll.append(l)

#define the plot layers that have just been created

p.layers=ll

#get out of batch mode. This actually creates the plot

p.batch=0

- And this is how some common features of the plot are modified.

p.setYrange([0, 1.5])

p.setTitleText("This is an example plot")

5.3. Viewing with SpectrumPlot

It is also possible to display spectra without taking apart the data format as is described in the previous section. All Herschel spectra types can be displayed with the `SpectrumPlot` package.

If `spectrum` is a Herschel Spectral type (Spectrum1d, Spectrum2d) then:

```
splot=SpectrumPlot(spectrum, useFrame=1)
```

will simply display the spectrum along with some standard header information. The `useFrame=1` allows for the possiblility of creating a plot without actually viewing it at first, but as the last step. The SpectrumPlot module is build on PlotXY, and so many of the features you would use in `PlotXY` you can also use for `SpectrumPlot`. Below are a few examples:

```
from herschel.ia.toolbox.spectrum.gui import SpectrumPlot
from herschel.ia.gui.plot.renderer.StyleEngine.ChartType import HISTOGRAM,LINECHART
#
#
Creating the plot
sp=SpectrumPlot(spectrum,useFrame=1)
#
#
adding a second spectrum to the plot
sp.add(spectrum2)
#
#Start
fresh again
p = SpectrumPlot(spectrum, useFrame=1)
#
#get
graphs
g0 = p.getGraphs()[0]
g2 = p.getGraphs()[2]
#
#display as line graph or histogram
g0.layer.style.chartType = HISTOGRAM
g2.layer.style.chartType
= LINECHART
#
```

```
#add
annotations
g0.layer.addAnnotation(Annotation(4000,1,"My
annotation"))
g0.layer.addAnnotation(Annotation(5000,0.98,"My
annotation"))
#
#select
a range of data
g0.layer.xaxis.addMarker(AxisMarker(4200,4400))
g2.layer.xaxis.addMarker(AxisMarker(6000,6500))
```

These last lines will produce the following plot:

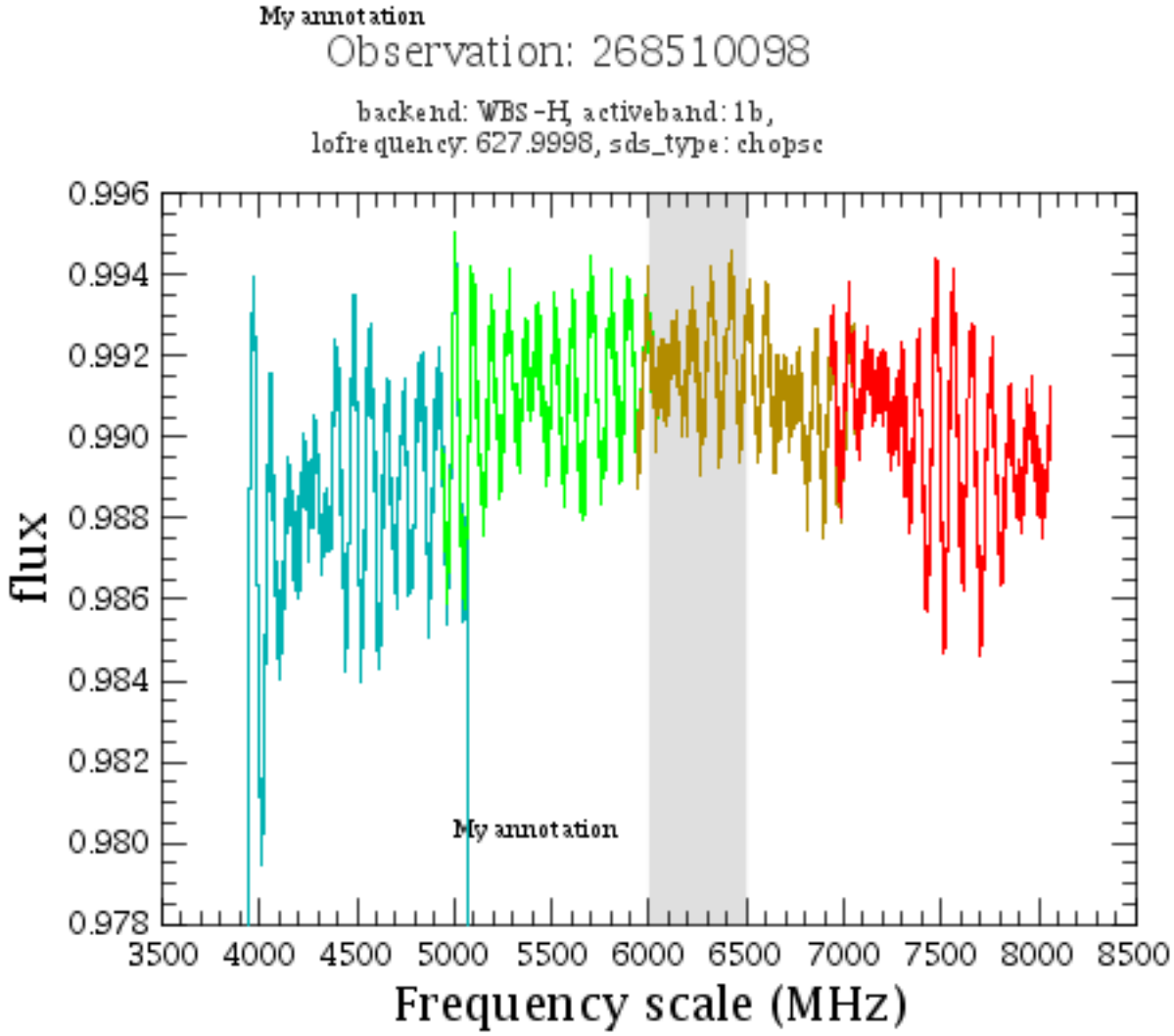
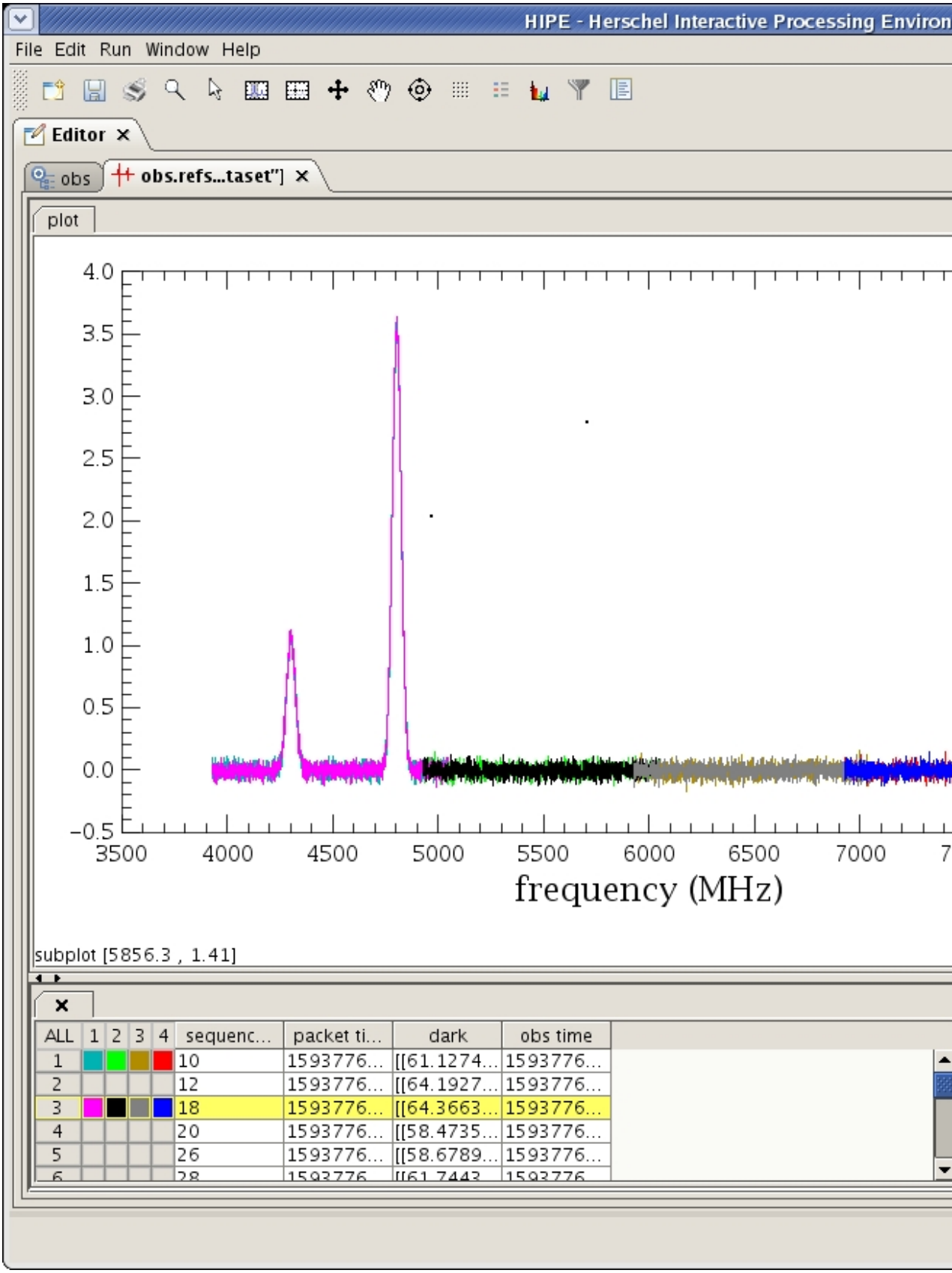


Figure 5.1.

5.4. The SpectrumExplorer Package

The SpectrumExplorer package allows one to visualize HIFI, PACS, and SPIRE SpectrumDatasets in a userfriendly, interactive way. To activate it, click on a SpectrumDataset in the Variables window with the right mouse button and select 'Open With' and 'Spectrum Explorer'. This will plot the spectrum. If the SpectrumDataset is wrapped in an ObservationContext or a TimelineProduct, double-click on it in the Variables window and then click down the product tree to the desired SpectrumDataset. Then you will see something like this:



In the shown example of the HIFI instrument, individual WBS sub-bands or scans can be plotted by clicking on the appropriate boxes in the bottom panel and removed by double-clicking. The plot can be modified interactively or other actions can be performed after clicking the appropriate button at

the top panel. The same actions can also be selected by clicking the right mouse button on the plot window. From left to right:

- button 1: new... (file, script)
- button 2: save the plot to a file. One can chose from a number of file formats.
- button 3: print the plot.
- button 4: this is the default mode when SpectrumExplorer is started. Change the horizontal and vertical plot ranges by drawing a rectangular box using the left mouse button. Also, one can scroll the spectrum along the horizontal and vertical axes by clicking on an axis with the left mouse button and then moving the mouse or using the mouse wheel. Control-left mouse button will un-zoom the plot (or use the Zoom option under the right mouse button).
- button 5: highlight a plotted spectrum by clicking on it with the left mouse button. Then with the right mouse button, under 'Spectrum' chose an operation to be applied to the selected spectrum. Subsequently a GUI for the particular operation will appear. Enter any details and click on 'Accept' in the GUI. Note that currently this will only work if the dataset was selected directly from the Variables window, not from the ObservationContext or TimelineProduct tree. In the latter case, create a variable first by dragging the dataset from the Outline window to the Variables window.
- button 6: highlight spectral ranges by using the left or middle mouse buttons. This will create a vertical grey bar. In the near future (likely HIPE 1.2) all sorts of tasks can be applied to the marked area(s), e.g. flagging datapoints and fitting lines and Gaussians.
- button 7: select specific data points with the left mouse button. In the near future (likely HIPE 1.2) one will be able to flag or remove these points.
- button 8: click on a spectrum and drag it to another or a new panel (note that dragging to the left and top of the original panel is mot possible). The spectrum can also be dragged to the Variables window where is will be stored as a new variable.
- button 9: pan through the spectrum by clicking the left mouse button and moving the mouse.
- button 10: only show the active plot panel, and change the axis ratio in order to fit the screen. Click button 7 again to show all panels.
- button 11: add (or remove) a grid to the plot
- button 12: display or hide the plot legend
- button 13: switch between line and histogram mode
- button 14: apply a filter to the attributes of a SpectrumDataset, such as chopper position, flags, etc. The filter GUI will appear at the bottom right of the SpectrumExplorer window. Attributes can be selected from a drop down menu. The filter values can be entered as ranges, e.g. sequence numbers 18-20 can be selected by entering '18-20' or '19+/-2' under 'filter'.
- button 15: any plot parameter (plot range, titles, colors etc.) can be modified in the same way as for the PlotXY() package.

Finally, it is also possible to...

- ...overplot multiple SpectrumDatasets. If the dataset is listed in the Variables window, click and drag it to the area under the SpectrumExplorer plot window, and a new tab for that dataset will appear that can be selected subsequently.
- ...add, remove, or invert axes by right-mouse clicking on an axis. In the future it will be possible to change the units of the axis as well (e.g. USB/LSB/IF/Vlsr for HIFI).

Chapter 6. Changing to LSB/USB and Velocity

6.1. Changing HIFI Frequency Scales

In practice there at two methods of altering the HIFI frequency scales: using the Spectrum Explorer GUI or from the command line. These two approaches differ in one fundamental way. The command line tasks will actually change the data, by resetting the frequency to upper/lower sideband representation or velocity. The GUI only changes what is seen in the SpectrumExplorer, the data themselves are not changed.

There are four fundamental ways of representing the frequency scale for HIFI: the intermediate frequency (default), the upper sideband frequency, the lower sideband frequency, or by velocity.

One final note, currently the HIFI pipeline is providing the "final" spectra represented in both USB and LSB. The level 2 product names are tagged LSB or USB it is still possible from these spectra to transform back to IF or the other sideband.

6.1.1. Changing Spectral Views

The SpectrumExplorer provides internal means of viewing spectra. These views are only for display purposes and do not change the data.

6.1.1.1. LSB/USB

Assuming you have activated a spectrum in a SpectrumExplorer window. To move between a spectrum seen in the Intermediate Frequency, USB or LSB, right mouse click on the frequency access (not the title of the access, but the axis itself). A pull down menu for the access will appear.

6.1.1.2. Velocity

6.1.2. Change Spectral Views from the command line

6.1.2.1. LSB/USB

The task to convert the actual frequency scale in a HifiTimelineProduct or HifiSpectrumDataset is called ConvertFrequencyTask. Assuming spectrum is the variable name for a HifiSpectrumDataset with the frequency scale of the data in expressed as IF frequencies.

```
cft=ConvertFrequencyTask()
cft(sds=spectrum,to="lsbfrequency")
```

Of course, it is also possible to convert to the upper sideband. for this the keyword is "usbfrequency".

```
cft(sds=spectrum, to='usbfrequency')
```

To convert back to the IF, use:

```
cft(sds=spectrum, to='frequency')
```

The ConvertFrequencyTask works equally well on the HifiTimelineProduct itself. In this case all the internal HifiSpectrumDatasets are converted. This is not something you should do in the early stages (before level 0.5) of the HIFI pipeline. For example on a level 1 HifiTimelineProduct:

```
cft=ConvertFrequencyTask()
cft(htp=hifitimelineproduct, to='frequency')
```

Note

Direct application of the ConvertFrequencyTask changes the data listed in the spectrum. Conversion back to the original IF scale is possible, just use the to='frequency' option.

6.1.2.2. Velocity

The ConvertFreqencyTask also works to convert the frequency scale to a velocity scale once given the reference frequency.

```
cft=ConvertFrequencyTask()
cft(sds=spectrum,to='velocity',reference=576.268,inupper=False)
```

In the above example, I had to specify the reference frequency in GHz and whether this reference frequency is for the upper (inupper=True) or lower (inupper=False) sideband.

Another call to ConvertFrequencyTask using "to = 'frequency'" will undo the change to velocity as well.

6.1.2.3. Review of ConvertFrequencyTask

The ConvertFrequencyTask works on HifiSpectrumDatasets or HifiTimelineProducts. The task uses the keywords "sds" for HifiSpectrumDataset and "htp" for HifiTimelineProducts. The conversion of frequencies is done using the "to" keyword. The following table shows the various possibilities:

to=	**Description**	**Other keywords necessary**
frequency	Converts to the Intermediate Frequency scale.	None
usbfrequency	Converts to the Upper side band Frequency scale.	None
lsbfrequency	Converts to the lower side band Frequency scale.	None
velocity	Converts to the velocity scale in km/s	reference=reference frequency, inupper=(True or False)

Chapter 7. Mathematical Operations on Spectra

Please see the Data Analysis Guide for more up-to-date information.

Chapter 8. HIFI Standing Wave Removal Tool

8.1. Introduction FitHifiFringe

FitHifiFringe is a tool to remove standing waves from level 1 and level 2 HIFI spectra. It makes use of the general sine wave fitting task FitFringe, but has been adapted to read HIFI SpectrumDatasets, and provide input and defaults applicable to HIFI spectra. For details on the sine wave fitting method, please consult the FitFringe manual.

FitHifiFringe is being tested on PV data. It can be applied to all bands, with the caveat that the standing waves in HEB bands 6 and 7 are not sine waves, and hence can only be fitted in an approximate way by fitting a combination of many sine waves.

Also note that presently FitHifiFringe can only be applied to WBS, not HRS, spectra.

8.2. Running FitHifiFringe

FitHifiFringe can be run by clicking on a WBS level 1 or 2 SpectrumDataset variable and then double-clicking on the applicable task. Alternatively it can be run on the command line as follows:

```
fhf = FitHifiFringe()

sds_out = fhf(sds1=sds_in,nfringes=2,midcycle=150.)
```

The input sds_in is a WBS level 1 or level 2 SpectrumDataset. The output sds_out SpectrumDataset is identical to the input, but with the fitted sine waves subtracted from the flux columns.

The following input parameters are allowed:

- nfringes: number of sine waves to be fitted [DEFAULT: 1]
- midcycle: This is an important parameter the user can supply. It is the typical standing wave period in the spectrum (in MHz). Any structure with periods much longer than that will be considered baseline, and no sine waves will be fitted to it. Any narrow peaks (spurs, emission or absorption lines) will be masked. [DEFAULT: 176 MHz]
- cycle: start of sine wave period search range [DEFAULT: 2727 MHz]
- plot=False: only show plot of end-result for each scan [DEFAULT: 3 plots per scan: (1) period versus Chi^2 (2) a before/after plot including the line mask (3) the before/after plot and the subtracted sine wave]
- ncycle: number of cycles to check [DEFAULT: 450]
- averscan=True: determine standing waves on average of all scans, and then subtract this from each scan [DEFAULT: process each scan separately] NOTE: this option only available in HIPE > 1.2

Chapter 9. Fitting Spectra

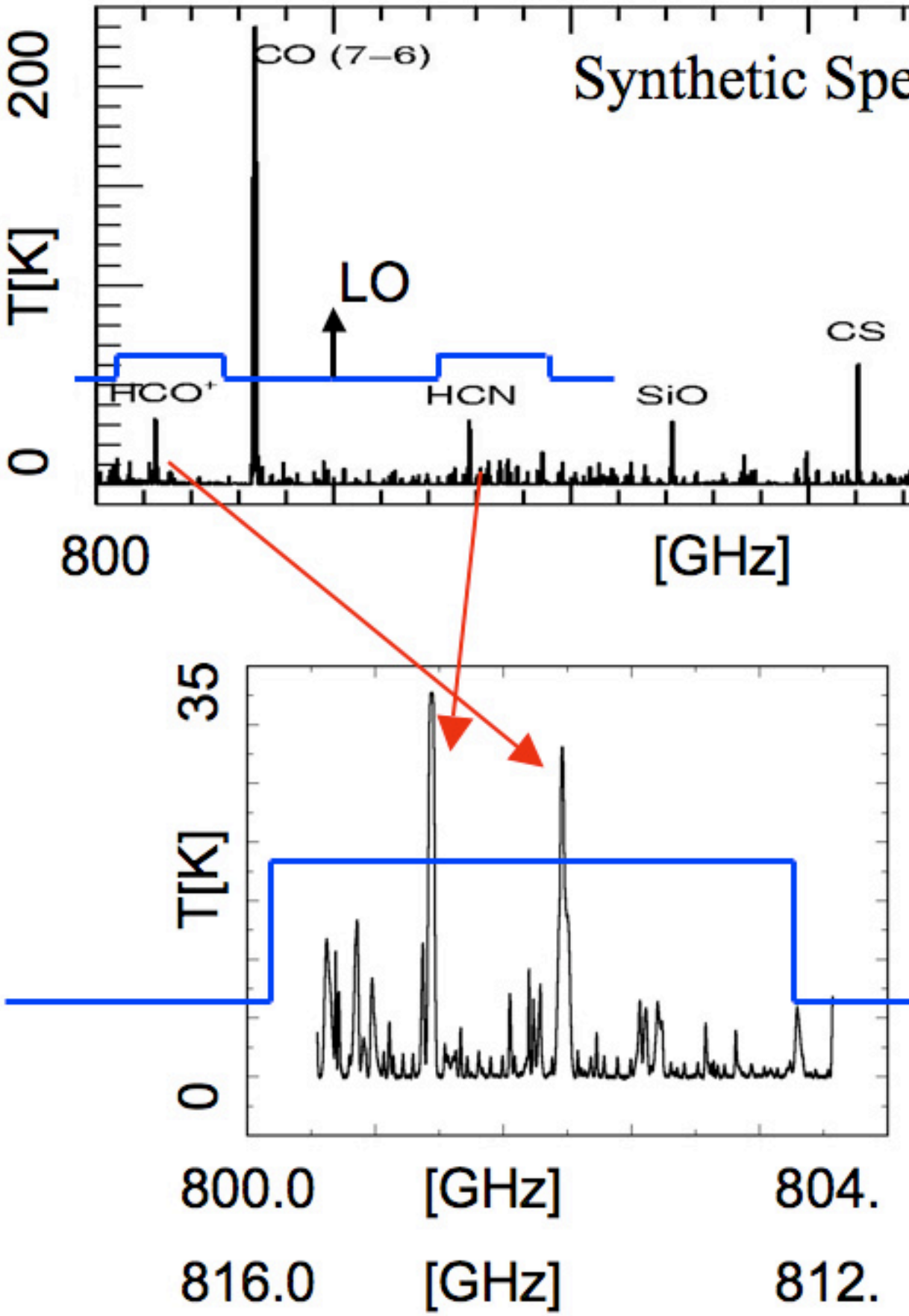
Please see the Spectrum Fitting chapter in the Data Analysis Guide.

Chapter 10. Sideband Deconvolution

The deconvolution tool is the post-Level 2 processor to separate the "folded" double sideband (DSB) data inherently produced by the heterodyne process into a single sideband (SSB) result. See the figure below. Fluxes (F_DSB) in the DSB spectrum are given by:

F_DSB(nu_IF) = g_u*F_sky(nu_LO+nu_IF) + g_l*F_sky(nu_LO-nu_IF)

where nu_LO+/-nu_IF are sky frequencies, and g_l and g_u are sideband gain (imbalance) factors, typically close to 1. The deconvolution is used to reduce WBS Spectral Surveys, which are collections of observations taken at many LO settings so as to constrain the solution. The algorithm finds a SSB solution that best models the observed DSB observations through iterative chi-square minimization (Comito and Schilke 2002). The algorithm automatically runs twice, first with gain factors set to 1.0 for stability, and then a second time, starting with the SSB solution of the first run, but this time allowing the gain values to be optimized as well.



Double sideband spectrum

The deconvolution tool is run AFTER the level 2 pipeline in HCSS 1.2. The level 2 pipeline performs the following tasks:

- splits the data into upper and lower sideband representations
- applies a gain correction specific to the LO frequency and sideband of the spectra
- corrects frequencies for spacecraft radial velocities
- resamples the spectra onto a fixed grid. For WBS this is done at 0.5 MHz spacing, with the first frequency snapped to the nearest 0.5 MHz

Any HIFI observation context will contain Level 2 products if run through the standard product generation.

10.1. Running the Deconvolution Tool

Assuming the observation context is named "MyObsContext", the user can run the deconvolution task on the command line with the default parameters by simply invoking:

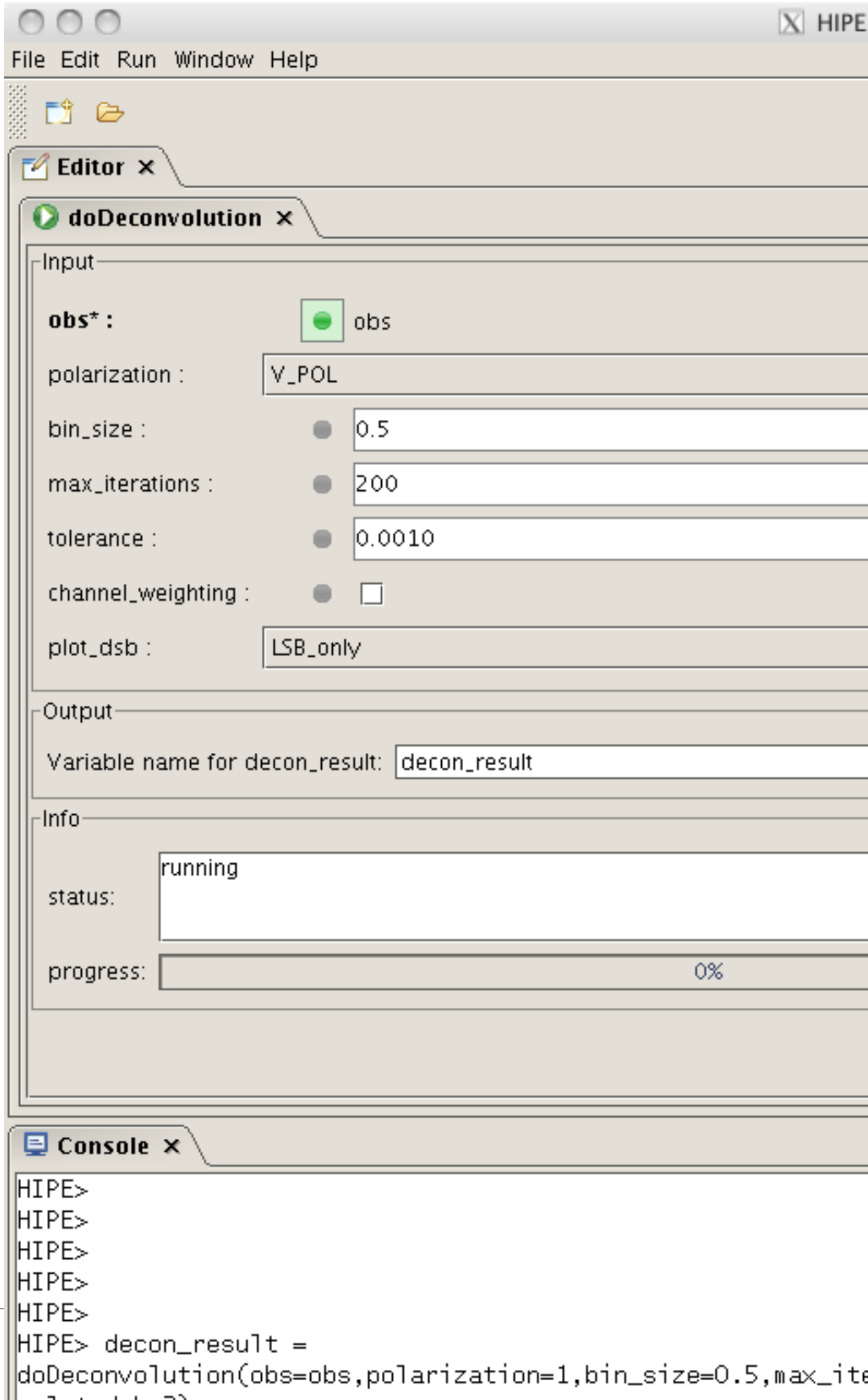
```
result=DoDeconvolution()(obs=MyObsContext)
```

The default parameters can be modified as follows:

```
result=DoDeconvolution()(obs=MyObsContext,
polarization=1,max_iterations=200,
tolerance=0.001,channel_weighting = True)
```

- max_iterations: Tells decon to stop after a specified number of interations if it has not converged by then
- tolerance: Specifies the tolerance of the solution. Lower values means it takes longer to converge
- bin_size : Tells deconvolution to bin data into a specified size (in Mhz). Effectively smooths the input
- polarization : Observations contexts store H and V polarizations. You can specify which to reduce with this option. 0=H, 1=V
- channel_weighting : Toggles whether or not the deconvolution uses the weight values in the data

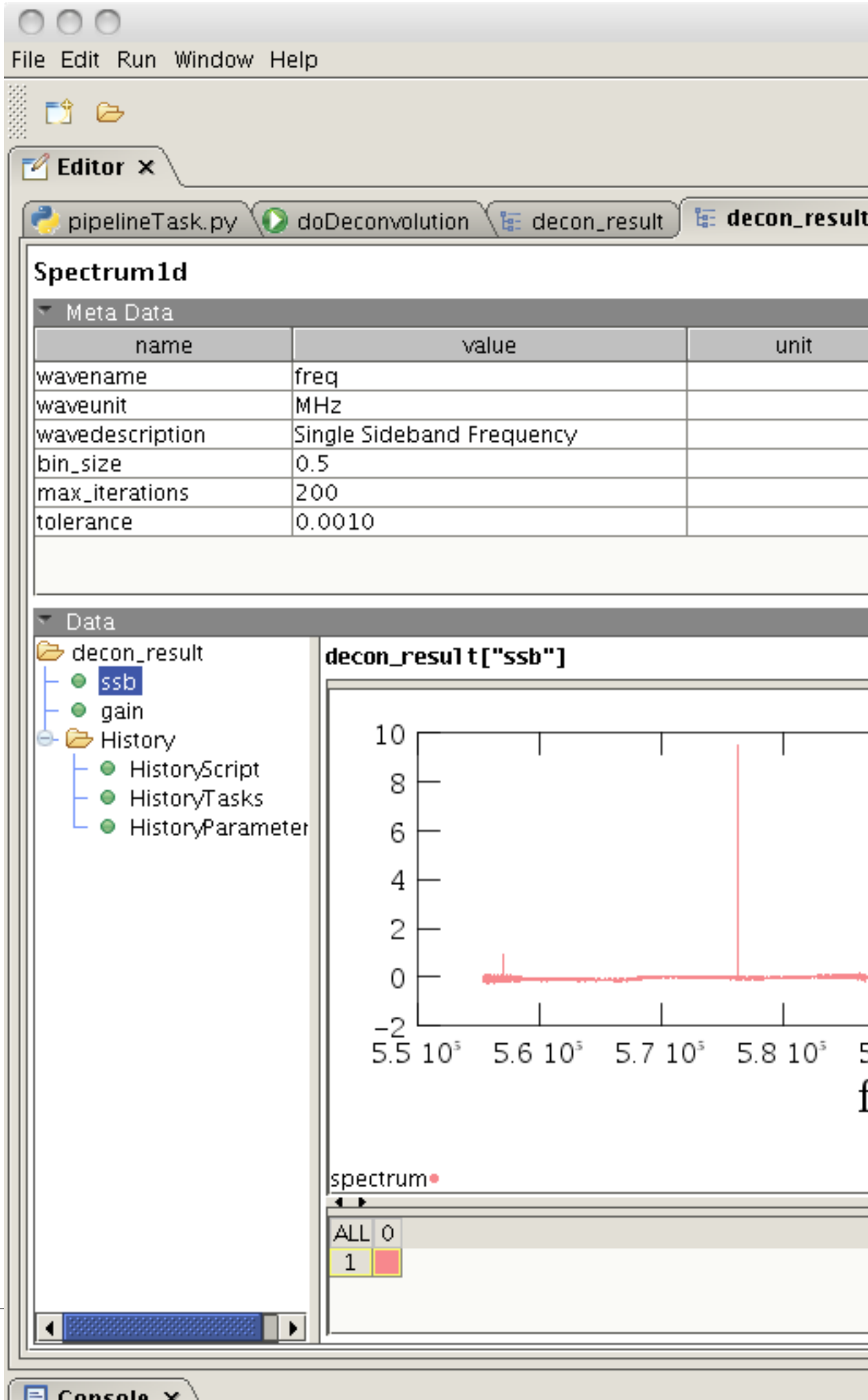
The Deconvolution Tool can also be run from a GUI by clicking on the the obs context in the "Variables" window, then double clicking 'doDeconvolution' in the "Task" list.


HIPE
File Edit Run Window Help
Editor
doDeconvolution
Input
obs* :
obs
polarization :
V_POL
bin_size :
0.5
max_iterations :
200
tolerance :
0.0010
channel_weighting :
plot_dsb :
LSB_only
Output
Variable name for decon_result:
decon_result
Info
status:
running
progress:
0%
Console
HIPE>
HIPE>
HIPE>
HIPE>
HIPE>
HIPE> decon_result =
doDeconvolution(obs=obs,polarization=1,bin_size=0.5,max_ite

Like other GUIs in the system, once the 'Accept' button is hit, the command line version is written in the console window so users know exactly how the task was called. This output can be cut and paste into user scripts for repeatability.

10.2. Viewing Deconvolution Results

- The output product result can be viewed with the product viewer.
- The single sideband result (ssb) is a dataset that can be viewed with the SpectrumExplorer. On the command line, it can be extracted from the product as follows: **`ssb=decon_result["ssb"]`** This contains the deconvolved spectrum, and is the primary output of the tool.
- The dataset "gain" can be viewed with dataset inspector. On the command line, it can be extracted from the product with: **`gains=decon_result["gain"]`** The deconvolution tool can estimate the sideband gains due to the redundant nature of the data taking. These estimates are stored per LO tuning in this product.
- The meta data added to ssb includes number of iterations and the tolerance, as can be seen in the HIPE screenshot below.


File Edit Run Window Help
Editor
pipelineTask.py
doDeconvolution
decon_result
decon_result
Spectrum1d
Meta Data
name
value
unit
wavename
freq
waveunit
MHz
wavedescription
Single Sideband Frequency
bin_size
0.5
max_iterations
200
tolerance
0.0010
Data
decon_result
ssb
gain
History
HistoryScript
HistoryTasks
HistoryParameter
decon_result["ssb"]
10
8
6
4
2
0
-2
5.5 10⁵
5.6 10⁵
5.7 10⁵
5.8 10⁵
spectrum
ALL
0
1
Console

Chapter 11. How to make a spectral cube

Spectral cubes from OTF mapping observations are produced as part of the SPG pipeline and are a level 2 product. However, re-processing of spectral cubes from a Level 2 product is likely desirable; this is done using the doGridding Task after calibrations of baseline, sideband gain, and antenna temperature.

The default operation of the task is to select the science datasets from an HTP and create a cube for each given spectrometer subband. Each slice of the cube is produced by computing a two dimensional grid covering the area of the sky observed in a mapping mode. For each pixel in the grid, the task computes a normalized Gaussian convolution of those spectra (equally weighted) falling in the convolution kernel around that pixel. After running the task you will have an array of cubes, one for each subband.

The SimpleCube product can be analyzed with the CubeSpectrumAnalysisToolbox.

11.1. Making a Spectral Cube via the command line

As a part of the automated (SPG) pipeline, Dogridding handles ObservationContexts but if you are making a cube yourself then you should use a Level-2 HifiTimelineProduct (HTP). As of HCSS 1.0, the HTP contains the satellite pointing information required to run the task but if you are using old data you must also supply the pointing, SIAM and uplink products. Some examples of usage are below

- **Data selection:**

  Make cubes for all the subbands, then display the first one:

```
cubes = doGridding(htp=htp)
cubes_count = len(cubes)
cube = cubes[0]
Display(cube)
```

  Or you might automatically create a separate variable for each cube as in the following routine:

```
# get a separate variable for each cube computed for each subband
for subband in range(len(cubes)):
    cube = cubes[subband]
    subband = cube.meta['subband'].value
    cube_name = -"cube_%d" % subband
    vars()[cube_name] = cube
```

  The medata of each cube will include a "subband" parameter stating the subband of the spectra which was used to compute the cube. This can be checked with,

  `print cube.meta['subband']`

- You may select just a part of the spectrum for each subband to be processed, that is, to generate the cube for a range of the channels of the given spectra. This can be done by providing a "channels" input, which is an Int2d array. This has to contain as many rows as subbands are to be processed. Each row must have two elements, the start and end channel to be read.

  The next example shows how to create a cube for the first and fourth subbands of a given spectrometer, reading just the channels 200 to 1200 in the first one, and the channels 400 to 700 in the second:

```
channelRanges = Intd2()
channelRanges.append(Int1d([200,1200]), 0)  # 0 means append row wise
channelRanges.append(Int1d([400,700]), 0)

cubes = doGridding(htp=htp, subband = Int1d([1,4]), channels=channelRanges)
```

- Select datasets by type: the default action is to take the science data sets that are on the source and this is normally sufficient. However, there may be observations where the dataset type to be read to make the cube has a different dataset type (e.g. an engineering observations whose type is called "other", instead of "science"). You can also select the off positions too.

```
cubes = doGridding(htp=htp, datasetType="science", ignoreOffs=false)
```

- Select some datasets by index instead of picking all the "science" datasets (datasetType is ignored if this is used): here we select subbbands 2 and 4, and datasets 3, 4, and 5 from the HTP. The weighting can also be specified to be "equal" (this is default) or that computed in DoChannelWeights in the Level 1 pipeline ("selection"):

```
cubes = doGridding(htp=htp, subbands=Int1d([2,4]), datasetIndices=([3,4,5]),
weightMode="selection")
cubes = doGridding(htp=htp, subbands=Int1d([2,4]),
dataset_indices=Int1d([3,4,5]), weightMode="selection")
cube_subband_2 = cubes[0]
cube_subband_4 = cubes[1]
```

- **Geometry:**

  Specify the (antenna) beam size:

  You can specify which is the half power beam width of the instrument i.e. the beam width. In the case of HIFI, case the beam is symmetric hence a single value is needed. However, one might in principle provide two different sizes along the x and y axis, thus specifying the dimensions of an elliptical beam.

  When this input is not provided the gridding task computes a default value for the HIFI beam size, based on a known function of the observed frequency. At present the formula used for the default case is:

  HPBW = 75.44726 * wavelength[mm]

```
# specify the size of the beam
cubes = doGridding(htp=htp,beam=Double1d([15.4]))

# specify the size of the beam. In this case the beam is wider along the
vertical axis.
cubes = griddingTask(htp=htp,beam=Double1d([10., 20.]))
```

  If the beam size is specified, and the pixel size is not specified, the pixel size will be function of the beam size taking into account the Nyquist criterion and the smooth factor (if any given). Usually, for nyquist sampling, the default pixel size becomes half the beam size.

- Specify the type of filter:

  By default the convolution is performed with a gaussian filter function, however, the user can specify other filter types.

```
cubes = doGridding(htp=htp, filterType="box")
```

  At present the available filter functions are box function (best for Raster maps) and a Gaussian function (best for OTF). Other filter functions maybe added in next releases.

```
# the default filter type is gaussian
cubes = doGridding(htp=htp, filterType="gaussian")
```

- Specify the parameters of the filter along each axis:

  The parameters that characterize each filter can be modified. For example, to use a box filter with a different length:

```
parameters = [Double1d([0.5]), Double1d([1.5])]
cubes =
doGridding(htp=htp,weightMode="equal",filterType="box",filterParams=parameters)
```

  The next example specifies the parameters length and sigma of the Gaussian filter function, when using a gaussian filter (default case).

```
# the -"influence area" is the area surrounding a grid point
# where the algorithm must pick up all the available data points.
influence_area = 1.95  # length in pixels
# sigma of the gaussian function times SQRT(2)
sigma_sqrt2 = 0.3      # in pixels
xFilterParameters = Double1d([influence_area, sigma_sqrt2])
# default case:
influence_area = 1.8; sigma_sqrt2 = 0.36
yFilterParameters = Double1d([influence_area, sigma_sqrt2])

cubes = doGridding(htp=htp,filterType="gaussian", xFilterParams=
xFilterParameters, yFilterParams = yFilterParameters)

# it is also possible to pass both set of parameters in a single input:
parameters = [Double1d([1.8,0.4]), Double1d([1.6,0.3])]
cubes =
doGridding(htp=htp,weightMode="equal",filterType="gaussian",filterParams=parameters)
```

  The following example modifies the default parameters of the box filters (their length):

```
# customize a box filter i.e. set the length of the pixel, measured in pixels
parameters = [Double1d([0.5]), Double1d([1.5])]
cubes =
doGridding(htp=htp,weightMode="equal",filterType="box",filterParams=parameters)
```

  Note: bear in mind that the default values of each type of filter are thought to optimize the convolution.

- Specify the size of the pixels:

  The user can choose a pixel size different from the pixel size computed by default (based on other inputs and on the angular dimensions of the observed area).

  The pixel size must be given in seconds of arc. For example, to assign a pixel size of 20 arcsec along both axes the user can specify the `pixelSize` input

```
cubes =
doGridding(htp=htp,weightMode="selection",filterType="gaussian",pixelSize=Double1d([15]))
```

  And to assign a different pixel size along the x and y axis, the given Double1d must have two elements. For example, to get pixels 15 arcsec wide and 25 tall, the pixelSize input should be Double1d([15,25]):

```
cubes =
doGridding(htp=htp,weightMode="selection",filterType="gaussian",pixelSize=Double1d([15,25]))
```

By default the pixel size is computed so that it is optimal, based on the other parameters given to the task. If neither beam size nor smooth factor have been provided, the task will compute a default HPBW and then it will choose pixel size equal to the half of this HPBW, i.e. it will assume that the sampling was done with following the nyquist criterion. The default pixel size will be the biggest of the values (HPBW/4) and (HPBW/(2*smoothFactor)). By default the smoothFactor is 1.0 (no smoothing factor applied), so that the default pixel size becomes the half of the beam size.

If the map dimensions in pixels were specified, the pixel size will be simply the division of the area actually observed by the number of pixels specified in the map size parameter.

If an smooth factor is provided, the pixel size will be the largest of HPBW/4 and (HPBW/2)*smoothFactor

- Specify the dimensions of the map:

  The user can specify the size of the map, in pixels, by means of the mapSize parameter. For example:

  ```
  cubes = doGridding(htp=htp, mapSize=Int1d([10,20]))
  cube = cubes[0]
  ```

  will create a cube 10 pixels wide and 20 pixels high. When this parameter is not specified the task computes the optimal dimensions taking into account the (antenna) beam size as well as the area of the sky covered by the input spectra.

- Specify the reference pixel

  The user can specify which is the reference pixel of the grid. It is also possible to define the coordinates of that reference pixel. If the latter is not provided the reference pixel will provide the coordinates, measured in pixels, of the projection centre, which is, in its turn, computed as the center of the coordinates of the input spectra (usually the centre of the map). Hence if the user provides a reference pixel, the user is defining where, in the regular grid, lies the centre of the observed area.

  Please note that the convention for the pixels computed for the regular grid of the output cubes is that the (0,0) pixel corresponds to the center of bottom-most, left-most pixel of the regular grid. Please note that this differs in -1 from the usual convention for FITS images, where the center of the bottom-most, left-most pixel has coordinates (1.0, 1.0).

  If the user specifies only this refPixel input and the user does not specify the coordinates of that pixel, this will computed so that it gets the pixel coordinates of the centre of the input spectra.

  For example, if we want to force that the reference pixel is the pixel (3.5 , 4.0), then the refPixel input will be Double1d([3.5, 4.]). If no refPixelCoordinates are provided, then the centre of the coordinates of the input spectra will be locate at the pixel (3.5, 4.0) of the regular grid i.e. at the FITS pixel (4.5, 5.0) from the bottom-most, left-most pixel of the cube. This means that the value of the CRPXI1 parameter of the result cube will be equal to 4.5 and the value of the CRPIX2 parameter will be equal to 5.0 (remind that the cube header uses the usual FITS convention about pixel coordinates).

  ```
  cubes = doGridding(htp=htp,refPixel = Double1d([3.5, 4.0]))
  ```

  By setting both refPixel and the refPixelCoordinates input explained below, the user can place the regular grid at any arbitray location, although the user is adviced to let the task automatically compute these so that the grid is located at a suitable place fully covering the observed spectra.

- Specify the coordinates of the reference pixel:

In addition to choosing a reference pixel, the user can also specify its celestial coordinates i.e. the longitude and latitude of the point chosen as the reference pixel of the cubes to be made by the gridding task.

For instance, to make that the reference pixel is located at the coordinates (RA,DEC) = (308.9, 40.36) degrees, a refPixelCoordinates input can be provided with these coordinates. Let's say, in addition, that the user wants that these reference pixel is the (0,0) pixel located at the bottom left corner of the image. Then refPixel = (0,0). Then the user should call doGridding like in the following example:

```
refPixel = Double1d([0,0])
longitude = 307.9
latitude = 40.36
refPixelCoordinates = Double1d([longitude, latitude])
cubes = doGridding(htp=htp,refPixel=Double1d([0,0]),
refPixelCoordinates=Double1d([longitude, latitude]))
# or:
cubes = doGridding(htp=htp,refPixel=refPixel,
refPixelCoordinates=refPixelCoordinates)

#one can check that cubes[i].wcs.crval1 == longitude and cube[i].wcs.crval2 ==
latitude:
print cubes[i].wcs.crval1 == longitude # 1, True
print cube[i].wcs.crval2 == latitude # 1, True
```

Please note that if only the refPixelCoordinates input is provided, the user will be choosing the coordinates of the centre of the map.

By setting both refPixel and refPixelCoordinates the user can place the regular grid at any arbitrary location, although the user is adviced to let the task automatically compute these so that the grid is located at a suitable place fully covering the observed spectra.

11.2. Using Gridding Task

Another task is available, called GriddingTask, to make cubes of images. It works with any dataset or product that happens to implement the SpectrumContainer or SpectrumContainerBox interfaces. It can also work with a collection of SpectrumContainer's. Said without using the Java jargon means that it can accept various simple inputs, such as an Spectrum2d or an Spectrum1d since these are SpectrumContainers.

You may also creat your own collection of datasets, and pass it to the GriddingTask in order to provide the spectra to be read to make a cube by performing an spatial regridding (a convolution) of these spectra onto a regular grid computed based on the coordinates of the given spectra (and on optional inputs about the shape of the grid which can be given by the end users).

GriddingTask and the Spectrum Toolbox. The user can make use of the spectrum selection tools of the spectrum toolbox, to perform any selection of spectra followed by the usage of the GriddingTask to create a cube for each segment of the spectra in these selections. The following example shows how to combine SelectSpectrum with the GriddingTask:

```
# first, create an instance of the SelectSpectrum task
selector = herschel.hifi.pipeline.util.tools.SelectSpectrum()
# use SelectSpectrum to get a single HifiSpectrumDataset with the spectra that
fulfill certain criteria
# e.g. here one selects those spectra where its containing dataset has bbtype
equal to 6022.
selected = selector(htp=htp, selection_lookup={'bbtype':[6022]},
return_single_ds=Boolean.TRUE -)
# one might have a glance at the spectra in the -"selection" dataset e.g. in the
TablePlotter
```

```
cube = griddingTask(selected)
cubes = griddingTask.cubes
```

Making a SpectralSimpleCube with the GriddingTask .

```
#-------------------------------------------------------------------------------
#  make a dataset with all the spectra from all the science datasets
#  (isLine == true => bbtype == 6022)
#-------------------------------------------------------------------------------
selector = herschel.hifi.pipeline.util.tools.SelectSpectrum()
selected = selector(htp=htp, selection_lookup={'bbtype':[6022]},
return_single_ds=Boolean.TRUE -)

scienceOnIndices = htp.summary['isLine'].data.where(\
htp.summary['isLine'].data == Boolean.TRUE)
bbid = htp.summary['Bbid'].data[scienceOnIndices]

#another way of selecting...

selected = selector(htp=htp, \
                    selection_lookup={'bbtype':bbid[0]}, \
                    return_single_ds=Boolean.TRUE -)

#-------------------------------------------------------------------------------
# the gridding task that can work with any SpectrumContainer or collection of
SpectrumContainers, like the selected above
#-------------------------------------------------------------------------------

cube = griddingTask(container=selected)

#get a point spectrum from this selection,
ds_spectrum = selected.getPointSpectrum(1)
ds_segment = ds_spectrum.getSegment(3) # read its third subband -: get
SpectralSegment.
plotSegment = PlotXY(ds_segment.wave,ds_segment.flux,xtitle='Frequency
(MHz)',ytitle='Intensity')

#-------------------------------------------------------------------------------
# now let's play with the result cubes
#
#
# each cube is a SpectralSimpleCube which in its turn is an SpectrumContainer
# hence we profit all the spectrum toolboxes: arithmetics, statistics, etc.
# and we can e.g.directly obtain a point spectrum as for any other SpectrumContainer

row = 0; column = 10;
spectrum = cube.getPointSpectrum(row,column)

print spectrum.getLongitude()
print spectrum.getLatitude()


print spectrum.segmentIndices
# you can check that the cube, hence its spectra has a single -"segment" or subband

segment = spectrum.getSegment(0)
# or...
segment = spectrum.getSegment(spectrum.segmentIndices[0])

plotSpectrum = PlotXY(segment.getWave(),segment.getFlux(),xtitle='Frequency
(MHz)',ytitle='Intensity')

# There are several ways to visualize a cube such as
# the CubeSpectrumAnalysisToolbox:

cat = CubeSpectrumAnalysisToolbox(cube)

# you can also visualize it with the SpectrumExplorer,
# since the cube is an SpectrumContainer

# or simply display it as a cube of images:
```

```
display = Display(cube)
```

Optional inputs for the GriddingTask.

Most of the optional inputs of the DoGridding task are also applicable to the griddingTask namely: weightMode, filterType, mapSize, refPixel, refPixelCoordinates, pixelSize, smoothFactor, filterType, filterParams, detail, extrapolate and the input Wcs

In addition, there are other optional inputs which are specific to this task, namely container and containerBox

11.3. Using the GUI to make a Spectral Cube

The doGridding Task can be found in the "Applicable" folder of the Tasks view when an HTP is selected in the variable view; double-click on it to open the dialogue in the Editor View. The GUI is still a prototype and it is recommended to use the command line to run DoGridding for the time being.

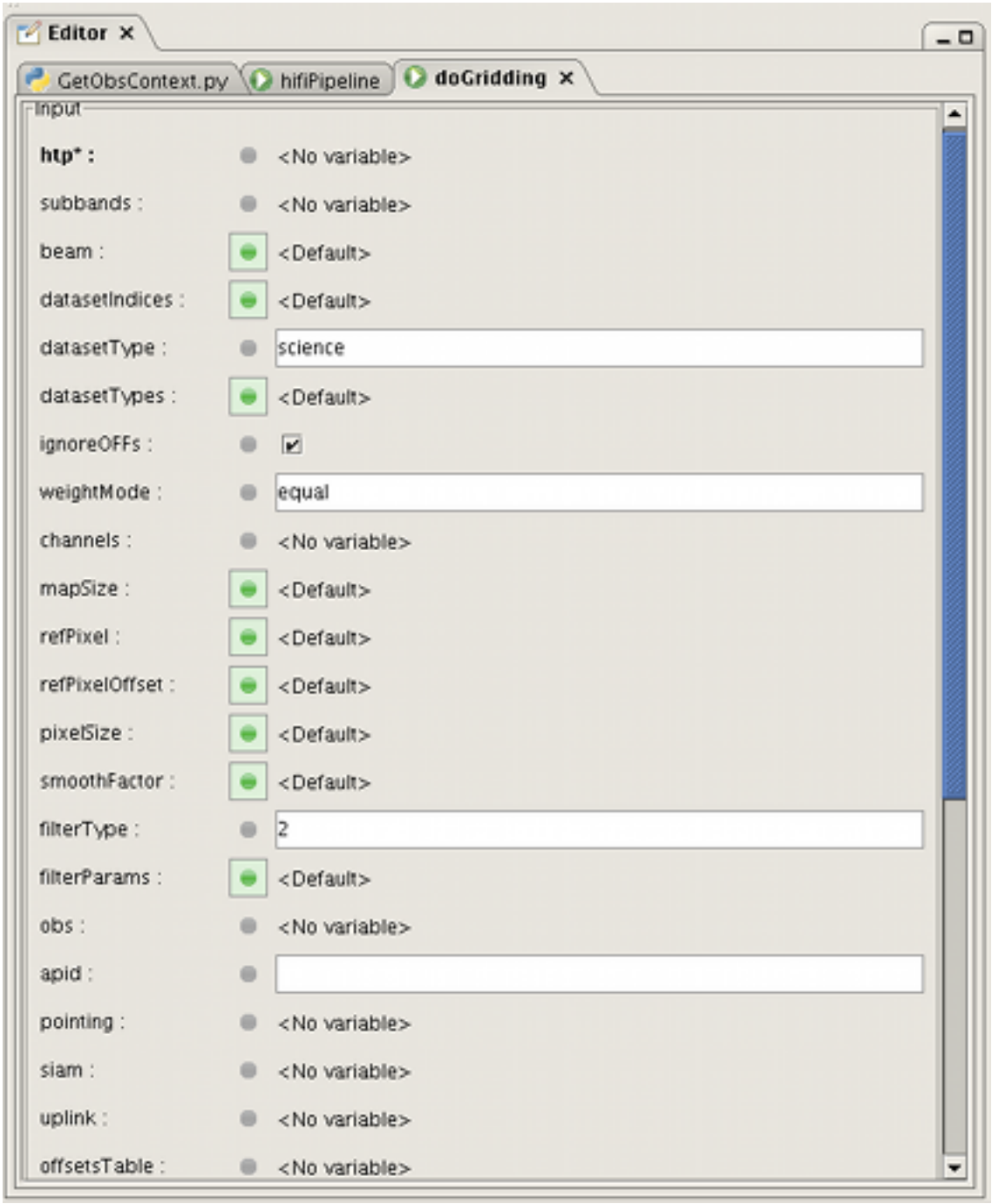


Figure 11.1. The doGridding task GUI form

Chapter 12. Trend Analysis

To be written

Chapter 13. Memory Issues

On occasion, one can run into the `java heap space` error when using HCSS software, especially when running the pipeline. Here are some things to help:

1. **User release.** Choose the "Advance" installation and increase the maximum amount of memory available to HIPE (the "User" installation allocates 1 Gb by default).

2. Modify the memory allocation (`java.vm.memory.min` and `java.vm.memory.max`) in `${install.dir}/installed.properties`

3. The "garbage collection" command `System.gc()` is also useful to force clearing memory. HIPE will automatically do this when memory becomes too full.

4. **Swap Store Properties:** It is possible to use the hard disk as swap space to preserve the memory available in HIPE. The following properties are defined to preserve computer memory. This becomes especially useful when pipeline processing long observations on a laptop, or on a pc with a 32 bit Operating System (TBC) and with average or limited memories capacities. However, any Task that uses or changes any HifiProduct (e.g., HifiTimelineProduct) will benefit from the use of swap space.

   The following properties can be defined (in the `user.props` file or using the "Hifi Product" tab in `propgen`) to set or to configure the Swap mechanism.

   - **hcss.hifi.pipeline.product.memory = true:** Setting the value of this property to "true" enables the swap mechanism. Note that the default value is "false".

   - **hcss.hifi.pipeline.product.swapstore = "swapStore".** This is the name of the LocalStore where the temporary data will be saved. The default location is:`${user.home}/.hcss/lstore/swapStore` .

   - **hcss.hifi.pipeline.product.swapratio = 0.25:** This property determines how much the swap mechanism is used and is used to set the threshold level of free memory. When a new dataset is set or retrieved from the HifiProduct, the HifiProduct will check the size of the dataset and the free memory in the system. If the condition:

     (memory free)*swapratio < dataset size

     is met, then all the floating datasets contained in the HifiProduct will be saved in the swap store.

     This property should have value between 0 and 1 and has a default value of 0.25.

     If the value is 0 all datasets will be always stored in the swap store. This is safe, but it could create performance delay (in the time needed to process the pipeline) due to the access time to the hard disk.

     In the case of long observations, setting the property to 1 could be dangerous because memory problems (like `Java.heap.space exception`), may still occur, although the pipeline will try to have the best performance possible.

   - **hcss.hifi.pipeline.product.savedisk = true:** This property determines whether an existing observation in the swap store should be overwritten or not. It is strongly suggested to keep the value = true, otherwise the space used in the hard disk will increase in proportion to the number of times a product is saved in the swap store.

Note

SwapUtil Class: At the moment, the pipeline does not clean the swap store after the processing. To avoid the swap store completely filling the hard disk when many

observations are processed, it is suggested one manually remove the swap store by either deleting the swapStore directory, or in HIPE:

```
from herschel.hifi.pipeline.product import SwapUtil
SwapUtil.delete()
```

Chapter 14. Notes for Calibration Scientists

14.1. Input/Output of Spectra

14.1.1. Accessing Spectra

HIFI spectra can be stored in various formats, and there are various ways to access them.

14.1.1.1. Accessing Spectra in a DbPool

At the ICC, HIFI pre- and post-pipeline data are stored in online databases called DbPools, and can be retrieved relatively easily and stored locally.

Product Pools

A Product Pool or Pool is a database where Products are stored. The database is implemented in different ways, for example as a directory tree on disk (a LocalStore product pool), or a Versant database (a DbPool). What matters is that they can talk to the software in the Product Access Layer (PAL) of HCSS. Let's consider concrete examples of HCSS routines for data processing. A useful page when pondering jython scripts for HCSS: [ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/releases/doc/api/index.html]

Product Access Layer

One should not work directly with pools. The ProductStorage interface is meant to satisfy your need to query, load, and save products in pools. A set of methods residing in herschel.ia.pal are meant to provide a uniform interface between you and the data you want, which may be located far-flung pools.

The PAL can talk to several types of pools, some of which are: CachedPool, DbPool, HsaPool, LocalStore (note despite the name, it's a pool, not to be confused with the ProductStorage class mentioned below). A DbPool is a Versant database, usually remote but not necessarily. A LocalStore is a database in the form of a directory structure full of FITS files on a disk mounted on the user's machine. A CachedPool is, unsurprisingly, a locally cached copy of products and metadata retrieved from remote pools.

The methods in herschel.ia.pal.ProductStorage() provide an interface to pools.

```
HIPE> from herschel.ia.pal import *
HIPE> store = ProductStorage()
```

"store" is ready to accept and interface with pools I choose to register within it.

How do I grab a pool and stick it into store? Use the PoolManager. If I take a shot in the dark:

```
HIPE> firstpool = PoolManager.getPool('foo')
HIPE> store.register(firstpool)
HIPE> print store.getPools()
[herschel.ia.pal.pool.lstore.FitsProductPool:foo]
```

What happened: hipe looked at the property 'hcss.ia.pal.pool.foo'

```
HIPE> from herschel.share.util import Configuration
HIPE> print Configuration.getConfigurationProperty("hcss.ia.pal.pool.foo")
None
```

Because in this case it is not defined, a new Product Pool of type hcss.ia.pal.defaulttype was created and put in store:

```
HIPE> print Configuration.getConfigurationProperty("hcss.ia.pal.defaulttype")
lstore
```

By the way, these config variables are also visible using the propgen gui.

Let's connect to a remote pool, a versant database at the SRON ICC. I might think I can use the PoolManager getPool() method again after changing hcss.ia.pal.defaulttype to "db"

```
HIPE> Configuration.setProperty("hcss.ia.pal.defaulttype","db")
HIPE> print Configuration.getConfigurationProperty("hcss.ia.pal.defaulttype")
db
```

and perhaps other settings, but it doesn't work, in my experience.

Use the DbPool class. The important Configuration parameters for DbPool methods are var.database.devel and var.database.server

```
HIPE> print Configuration.getConfigurationProperty("var.database.devel")
hifi_icc_ops_1@iccdb1.sron.rug.nl XYZ READ
HIPE> print Configuration.getConfigurationProperty("var.database.server")
@iccdb1.sron.rug.nl
```

But don't change these. Instead, create a new configuration property, associate it with the DbPool, and register it in the store:

```
HIPE>
Configuration.setProperty("interesting_db", -"hifi_ops_obs_icc_1@iccdb1.sron.rug.nl
0 READ")
HIPE> store.register(DbPool.getInstance("interesting_db"))
```

Now I can peruse the product pools in my product storage:

```
HIPE> print store.getPools()
[herschel.ia.pal.pool.lstore.FitsProductPool:foo,
herschel.ia.pal.pool.db.DbPool@3be4a2c]
```

The first pool you register is automatically your 'prime' pool and is writable:

```
HIPE> print store.getWritablePool()
herschel.ia.pal.pool.lstore.FitsProductPool:foo
```

NOTE! There seems to be a bug preventing reading and writing from the same store. In my experience, writing to pool foo will fail. Open a second store and register foo there.

So, finally, let's take some data products from the dbpool and save them in our localstore pool for offline use. There is a gui, but I can't recommend it, I haven't figured out how to use it.

```
HIPE> result = browseProduct(store)
```

I prefer the command-line.

How are products in the pool identified? By URNs (Universal Resource Name). A product always has a urn. You may also 'tag' products, and you may find some convenient tags have been created for you (they have). But being naive, let's get a list of all ObservationContexts in our store. Remember, ObservationContexts are the basic units of Herschel data processing. Pools are databases, so it's not surprising that we can query them. For information on query formats, see package herschel.ia.pal.query: [http://www.rssd.esa.int/SD-general/Projects/Herschel/hscdt/releases/doc/api/herschel/ia/pal/query/package-summary.html#package_description]

```
HIPE> q = Query(ObservationContext,"p","1")
HIPE> qresult = store.select(q)
```

This will return a list of all objects of type ObservationContext in all pools registered in store. It's actually a list of ProductRefs - references to ObservationContexts. That's to conserve memory. DON'T do a query on type Product on the current database, it will take hours if not days to return. The "p" is just a placeholder, used in more selective queries (see link above).

qresult is a list of ProductRefs:

```
HIPE> print qresult[0]
urn:db_hifi_ops_obs_icc_1:herschel.ia.obs.ObservationContext:39036
HIPE> a = qresult[0]
HIPE> print a.__class__
herschel.ia.pal.ProductRef
HIPE> print a.meta
{type=OBS,creator=HifiPipeline,creationDate=2009-08-15T16:50:23.224000
TAI (1629046223224000),description=another
observation...,instrument=HIFI,modelName=FLIGHT,
    startDate=2009-08-01T21:43:16.000000 TAI
(1627854196000000),endDate=2009-08-01T21:49:40.000000 TAI (1627854580000000),

obsState=LEVEL0_PROCESSED,obsid=1342181162,odNumber=80,obsMode=HifiEngSwitchonLO,origin=Instrument
Control Centre,aorLabel=Calibration_mo_13-SwitchonLO4a-OD80,
    aot=HifiEngSwitchonLO,equinox=2000.0,missionConfig=MC_H19_P13_S17_PV,object=No
Pointing,raDeSys=ICRS,ra=309.79251954150624,dec=68.0193187761005,posAngle=358.13260329747203,
    raNominal=0.0,decNominal=0.0,telescope=Herschel Space
Observatory,observer=unknown,proposal=Calibration_pvhifi_8,pointingMode=No-pointing}
```

Probably, there's an ObsID in which you're interested, for example 1342181163:

```
HIPE> from herschel.ia.numeric import *
HIPE> obsids = Long1d([(i.meta["obsid"].value) for i in qresult])
HIPE> print obsids
[1342181162L, 1342180840L, -... -, 1342180571L]
HIPE> idx = obsids.where(obsids == 1342181163L)
HIPE> my_obscontext = qresult[idx.toInt1d()[0]].product
HIPE> print my_obscontext.__class__
herschel.ia.obs.ObservationContext
HIPE> print my_obscontext
{description="another observation...", meta=[type, creator, creationDate,
description, instrument, modelName, startDate, endDate,
    obsState, obsid, odNumber, obsMode, origin, aorLabel, aot, equinox,
missionConfig, object, raDeSys, ra, dec, posAngle, raNominal,
    decNominal, telescope, observer, proposal, pointingMode], datasets=[History],
history=None, refs=[auxiliary,calibration,level0,level0_5,
    level1,level2,logObsContext,quality,trendAnalysis]}
```

Or another way, using the tag "observations" which has been created in the current database by Peer:

```
   HIPE> observations = store.load(store.getUrnFromTag("observations")).product
   -"observations" is a MapContext, a hash of ObservationContexts indexed by
ObservationID keys:
   HIPE> my_2nd_obscontext = observations.getProduct("1342181163L")
```

Now save the observationcontext to our localstore pool. There's a rub here: remember that an observation context contains references to other contexts. Those references might point to products in pools we have not yet registered. If you try to save the observationcontext now it will fail for want of being able to access these other data.

At present, you must register the dbpools for at least the Auxiliary and Calibration contexts:

```
   HIPE> Configuration.setProperty("aux_dbpool","hifi_ops_aux@iccdb1.sron.rug.nl 0
READ")
   HIPE> aux = DbPool.getInstance("aux_dbpool")
   HIPE> Configuration.setProperty("cal_dbpool","hifi_ops_cal@iccdb1.sron.rug.nl 0
READ")
   HIPE> cal = DbPool.getInstance("cal_dbpool")
   HIPE> store.register(cal)
   HIPE> store.register(aux)
```

Further, there seems to be a bug (SPR HCSS-7885) loading and saving to the same store. If you try to save to the prime pool in your store, it will fail silently. Until this is fixed, create a second store in which you have only your local store for saving, and write to that:

```
   HIPE> store2 = ProductStorage()
   HIPE> writepool = PoolManager.getPool('write_pool')
   HIPE> store2.register(writepool)
   HIPE> store2.save(my_obscontext)
```

14.1.1.2. Accessing Spectra in Local Pool

By default, spectral products are stored on a local disk in 'local store' format in the directory .hcss/lstore/. The user can modify the format and storage location by adding the following properties to the configuration files in the user's .hcss directory (usually .hcss/user.props) before starting the HIPE session:

#store data in a local store ('lstore')

`hcss.ia.pal.defaulttype = lstore`

#default location of data storage

`hcss.ia.pal.pool.lstore.dir = ${user.home}/.hcss/lstore`

Regardless of data format or location, spectra are loaded into the HIPE session with the following three steps:

1. register the pools of interest, so they can be accessed. For example if pools 'test1' and 'test2' exist, they are registered as follows:

   `storage = ProductStorage()`

   `pool1 = PoolManager.getPool("test1")`

   `pool2 = PoolManager.getPool("test2")`

```
storage.register(pool1)

storage.register(pool2)
```

2. obtain the unique 'urn' reference numbers of the spectrum products of interest. This can be obtained
   - interactively:

     ```
     result=browseProduct(storage)
     ```

   - from the command line

     ```
     query = Query("creator == 'my name'")

     result = storage.select(query)
     ```

3. finally, load the spectral products. In this case, the first one in the list of returned urn's:

   ```
   prod = storage.load(result[0].urn).product
   ```

14.1.1.3. Accessing Fits Files

Fits files can be loaded from disk as follows:

```
fa=FitsArchive()

prod = fa.load("filename")
```

14.1.2. Exporting Spectra

HIFI spectra can be stored in various ways. While users may import data from the HSA or, for calibration scientists, the Versant database (section "Accessing Versant Database"), they usually store their products on a local disk in one of the following ways.

14.1.2.1. Export to Local Pool

Spectra can be stored in fits format in a 'local store'. Set up this storage, and its location, in the configuration files in the user's .hcss directory:

#store data as local store ('lstore')

```
hcss.ia.pal.defaulttype = lstore
```

#default locations of data storage

```
hcss.ia.pal.pool.lstore.dir = ${user.home}/.hcss/lstore
```

To export spectra, first create a pool called, e.g., 'test', and then save the product in it:

```
storage = ProductStorage()

pool = PoolManager.getPool("test")

storage.register(pool)

storage.save(product)
```

14.1.2.2. Export to Fits File

To export spectra to fits files, issue the following HIPE commands:

```
fa=FitsArchive()

fa.save("filename", product)
```

14.2. Database, Binstruct and MIB

HIFI data are uniquely labeled by an obsid and are stored in HIFI ICC Versant Databases. Go to the Download page from the ICC internal pages to download Versant software. See the Databases page to find which database an obsid is associated with, and ICC Database contents for information about the observation.

Databases are named according to the data's test environment, for example IST data is held in databases labelled `ist_fm_x_prop` (x is an integer), while simulator data is stored in `ds3`, and flight data in `hifi_ops_obs_x`.

- You can change database within HIPE by, for example:

```
from herschel.share.util import Configuration
Configuration.setProperty("var.database.devel", hifi_icc_ops_1@iccdb1.sron.rug.nl
0 READ)
```

- You can also set your preferred default database in the `user.props` file in your .hcss directory. For example:

```
# Set simulator database as default
var.database.devel = ds3@iccdb2.sron.rug.nl 0 READ
```

The `hifi-cal` database contains calibration information needed to process data through the pipeline. There are two ways of getting this information into a Herschel DP session:

1. Use the databases at the HIFI ICC, a developer version of HIPE should be configured to do this. If you have a problem, check that this line is in your properties file:

```
hcss.binstruct.hifi.mib.pal.database = hifi-cal@iccdb1.sron.rug.nl 0 READ
```

2. Or install the `hifi-cal` locally on your machine from a copy of that at the ICC. First, download the local store copy of the MIB (hifi-cal) from the ICC download page. and unpack this in your `.hcss` directory.

   Current MIB (13 Sept 2009) . Use hifi-cal-26-Jun-2009.tgz

   Next, change the Binstruct property in user.props so that it will look for a local copy of the database:

```
# Comment out setting that points to ICC database
#hcss.binstruct.hifi.factory = herschel.hifi.share.binstruct.HifiBinstructFactory
hcss.binstruct.hifi.factory = herschel.binstruct.DefaultBinstructFactory
hcss.binstruct.hifi.mib.pal.poolname = hifi-cal
#
# FOR 1.1 Track ALSO INCLUDE
hcss.ia.obs.cal.release = developer
```

   This will now point to a local pool to get the MIB (Mission Information Base) needed to create the HifiTimelineProduct.

MIB Updates

Please note that each time the MIB (or calibrations) is updated these will be present in the database `hifi-cal_dev` but, if you are using a local pool, you will have to update it yourself. Do this by reloading the hifi-cal from the hifi-icc pages.

The MIB data are needed to define and interpret uplink commands and downlink housekeeping. The housekeeping is interpreted using the binstruct module and are associated with a MIB via the TmVersion tables, which map a MIB to a period in time. We have different tm-version tables for each test environment (e.g., "ilt-qm", "ilt-fm", "ilt-par", "ist-fm"). You can set the tmversion with the following binstruct property:

```
hcss.binstruct.hifi.mib.pal.tm_version_map = ilt-fm
```

The TmVersion to apply per database is as follows:

Table 14.1. TMVersions to apply

Database	tm_version table	MIB type
ilt_par_*	ilt-par	**hcss.binstruct.hifi.services = herschel.binstruct.mib.MibPalServi**
ilt_fm_*	ilt-fm	**hcss.binstruct.hifi.services = herschel.binstruct.mib.MibPalServi**
ist_fm_*	ist-fm	**hcss.binstruct.hifi.services = herschel.binstruct.mib.MibPalServi**
sovt_*	ist-all	**hcss.binstruct.services = herschel.binstruct.hpsdb.HpsdbPalS**
hifi_icc_ops_1, ds*	flight	**hcss.binstruct.services = herschel.binstruct.hpsdb.HpsdbPalS**

Note that the binstruct properties for the test environments have an additional substring (hifi) to make them ICC dependent. The TmVersion tables for sovt_ and hifi_icc_ops_1 data are merged so satellite housekeeping can also be inspected.

The automatic use of the different tmVersion tables to associate HouseKeeping data via its mission configuration name will be available when SPR 2111 is implemented.

In the meantime, use the following workaround to change the MIB on the fly

```
#workaround allowing to compare data distributed over the different setups:
#assume tmversion is by default -"ilt-fm", i.e. mydb is one of the ilt databases
tmversion = Configuration.getProperty('hcss.binstruct.mib.pal.tm_version_map')
# apply this data, for ex:
hdf = accessDataFrameTask(obsid = myobsid, apid = myapid, db
= -"ilt_fm_5_prop@iccdb.sron.rug.nl 0 READ")
hk = accessPacketTask(obsid = myobsid, apid = myapid, db
= -"ilt_fm_5_prop@iccdb.sron.rug.nl 0 READ")
htp = HifiTimelineProduct(hdf,hk)
# now switch to next instrument setup:
Configuration.setProperty('hcss.binstruct.mib.pal.tm_version_map', -"ist-fm")
# Now important reset binstruct:
from hcss.binstruct import InstrumentProperties
InstrumentProperties.getInstance().initialize()
#
continue with data available in one of the -"ist-fm" databases
hdf = accessDataFrameTask(obsid = myobsid, apid = myapid, db
= -"ist_fm_4_prop@iccdb.sron.rug.nl 0 READ")
hk = accessPacketTask(obsid = myobsid, apid = myapid, db
= -"ist_fm_4_prop@iccdb.sron.rug.nl 0 READ")
htp = HifiTimelineProduct(hdf,hk)
#
# that's it!
```

14.3. Accessing Versant Database

Level 0 spectra are obtained from the Versant database as follows:

1. before starting HIPE, set the relevant server and database in $HOME/.hcss/default.props or in $HOME/.hcss/myconfig. For example:

   `var.database.server = @iccdb.sron.rug.nl 0 READ`

   `var.database.devel = ilt_fm_5_prop${var.database.server}`

2. then in HIPE, set the obsid to be retrieved: **`obsid=268494774`**

3. get the housekeeping data: **`hk=AccessPacketTask()(obsid=obsid, apid=1026)`**

4. get the data frames: **`df=AccessDataFrameTask()(obsid=obsid, apid=1030)`**. The meaning of common APIDs:

   ```
   apid=1028 HRS prime/horizontal polarization
   apid=1029 HRS prime/vertical polarization
   apid=1030 WBS prime/horizontal polarization
   apid=1031 WBS prime/vertical polarization
   ```

5. next step is often the creation of a HifiTimeline product from the retrieved data frames and housekeeping data:

   `htp=HifiTimelineProduct(df,hk)`

6. note that steps c., d., and e. can be very slow, if run on computers located far away from the Versant database, even for obsids with short integration times.

14.4. Accessing Versant Database with Web Interface

The ICC databases can now be accessed using a web interface (especially useful for Mac users!) using CIB 1.1, or newer.

To use the web interface, the following properties must be set in your user.props file:

```
hcss.access.connection=herschel.access.net.NetworkConnection
hcss.access.authentication=true
hcss.access.url=http://129.125.20.1:8181/hcss/tm
```

Dataframes and packets are accessed as above and an HTP can then be generated but, for the moment, it is not properly formed and successfully stepping through the pipeline will require some manipulation. The first time you do this a pop-up will ask for a username and password (contact Kevin for these), which will then be stored in encrypted format in your Hipe.props file.

It is not possible to run the pipeline task via the web interface; however, the ICC product pools can be accessed by the following:

```
from herschel.ia.pal.pool.http import HttpClientPool
pool = HttpClientPool ("http://129.125.20.1:8181/hcss/pal", -"sim3-
obs@iccdb1.sron.rug.nl 0 READ", -"username", -"password")
storage = ProductStorage (pool)
storage.authenticate()
```

```
ref = storage.select (Query (ObservationContext, -"1"))
```

print ref

will return a list of urns. To load an Observation Context into your session, here the 5th in the list, use:

obs = storage.load(ref[4].urn).product

14.5. How to configure the CIB (for use on a non-ICC cluster machine)

1. The CIB is correctly configured on the ICC cluster machines but to use it on your own machine you need to set the property path manually. Here is one way to do it:

   - Install build of choice and make a soft link between it and a generically named target, e.g.,

     ln -s hcss.icc.hifi-1.1.*1627* hcss.icc.hifi-current

   - In your .cshrc/.tcshrc file add

     setenv HCSS_PROPS ${HOME}/.hcss/user.props:${HOME}/hcss.icc.hifi-current/config/hifi-new.props

     For bash, **export HCSS_PROPS=${HOME}/.hcss/user.props:${HOME}/hcss.icc.hifi-current/config/hifi.props**

   - For builds earlier than 1624 you also need to set in your .hcss/user.props

     var.hifi.dir=var.hcss.dir

2. To run the pipeline you should then only need:

   obs=hifiPipeline(obsid=*obsid*, db=*database*)

3. In order to use a local MIB and auxilliary data from the ICC, put in your .hcss/user.props file:

   hcss.ia.pal.store.spgstore = {pipelineout, cal-local, aux-icc}

   Other options:

   - cal-icc, uses calibration from ICC (default)
   - cal-hsa, uses calibration from the HSA pool
   - aux-hsa, uses auxiliary data from the HSA pool

   The scripts these options invoke can be found in the config directory and adapted if need be for, e.g., databases at the NHSC or Cologne.

4. Some other properties maybe useful to set in .hcss/user.props:

```
# Allocate memory to HIPE
java.vm.memory.min=64m
java.vm.memory.max=1900m
#
# Set up pipeline stores
hcss.ia.pal.store.spgstore={pipelineout, cal-local, aux-icc}
#
# Access to HSA (needed for cal-hsa, or aux-hsa)
hcss.ia.pal.pool.hsa.haio.login_usr= # Contact ICC for username
hcss.ia.pal.pool.hsa.haio.login_pwd= # Contact ICC for password
```

```
hcss.ia.pal.pool.hsa.haio.apply_authentication=true
```

14.6. Navigating HIFI Products: How to get a spectrum data set from an ObsContext

The following script shows how to extract an ObservationContext from the Flight Products database pool at the ICC:

```
from herschel.share.util import Configuration

Configuration.setProperty("hifi-obs","hifi_ops_obs_1@iccdb1.sron.rug.nl
0 READ")

storage = ProductStorage()
storage.register(DbPool.getInstance("hifi-obs"))
#below I retrieved the urn number from the internal database listing
that Kevin provides for
# hifi_ops_obs_1
urn="urn:db_0_10090:herschel.ia.obs.ObservationContext:24256"
urn="urn:db_0_10090:herschel.ia.obs.ObservationContext:42136"
#
#The next step loads the reference
ref=storage.load(urn)
#
#The final step will actually download the ObservationContext into your
HIPE session
# Note at this step you can retrieve all levels of HIFI pipeline
processing, AUX product and Quality #Products.  All these products are
not yet loaded in your session, but they will be once you request
#them.
obs=ref.product
#
# To get the LoTrendTable which is produce for every observation
lotrend=obs.refs["trendAnalysis"].product.refs["LoTrendTable"].product["dataset"]

# To get the HIFI level1 timeline product for WBS-H
htp_wbsh=obs.refs["level1"].product.refs["WBS-H"].product
```

14.7. HIFI Housekeeping

14.7.1. Introduction

Data from onboard sensors monitoring HIFI or Herschel performance are stored in the databases along with the science data. They are available to 'expert' users who have access to the raw data packets, but are not visible to the 'general user,' ie. you must have access privileges to a database at the HSC or ICC.

Telemetry (TM) data are produced on board the spacecraft and collected and processed by the Spacecraft Management Unit (SMU). TM is formmatted in compliance with the CCSDS-compatible ESA Packet Telemetry Standard. TM is prioritized based on content for downlink during the Daily Telecommunication Period (DTCP). There are four types of TM packets stored on board: Telecommand verification packets, event packets, Housekeeping (HK) packets, and Science packets.

TMIngestion is the processing of raw telemetry packets into TmSourcePackets and storage in the operational database at the HSC (see here also). All TM packets are stored in the HSC database; in addition, TM packets containing science data are processed to produce Data Frames which are then also stored in the HSC database. The format of TMSourcePackets can be found here. The HIFI-specific HK packets are described in this document.

The Housekeeping (HK) section walks through some basic guidelines to accessing, viewing, and exporting HK data.

See also Chapter 3. "DP Commands" of the User's Reference Manual for more examples.

14.7.2. Accessing Housekeeping

HK data can be both digital and analog and has some similarities in structure with the dataframes packets. Two methods for obtaining HK data are described here: (a) command line, in which case HK packets and science Dataframes are queried from the database and stored in an array or table, and (b) a GUI created by the the script ObservationSelector(), available in the user-scripts area of the 'expert' HIPE installation. At the time of writing, the GUI method fails due to recent code changes in HCSS, so beware.

A brief description of how to get Level 0 data is given in Section 14.1.1. Data frames in the database are basically accessed after setting the relevant server and database.

As described in the HIFI HK ICD document, the APIDs for periodic HK data are 1026 (Primary chain) and 1027 (Redundant), and for essential HK data 1024 (P) and 1025 (R).

The HIFI backend APIDs are HRS-H:1028, HRS-V:1029, WBS-H: 1030, and WBS-V:1031.

To put all the periodic HK data for HRS-H (P) for a certain obsid into a TableDataset:

```
from herschel.hifi.hrs import *
obsid=3221226629L
apid   = 1028
hkapid = 1026
db = -"sovt2_fm_4_prop@iccdb1.sron.rug.nl 0 READ"
hk = accessPacketTask(obsid = obsid, apid = hkapid, db = db)
hkDataset = HrsHK().createHKDataset(apid, -"All", hk)
```

To view the HK data in a graphical table:

```
HrsHKViewTask()(model="FM", apid=apid, hk_list=hk)
```

To create a TimelineProduct from the HK and Data Frames:

```
df = accessDataFrameTask(obsid=obsid, apid=apid)
htp = HifiTimelineProduct(df, hk)
```

A set of convenience scripts for accessing HK data is available in the user-supplied scripts directory. These scripts will be installed with HIPE if you choose the 'Expert' installation. For example,

```
from herschel.hifi.scripts.users.volker import all
```

NOTE: code examples below this point are obsolete. Update coming.

`dataframes = ObservationSelector() (obsid, spectrometer, gui)`.

To display the values of all digital and analog HK values of the HRS into a JTable:

```
from herschel.hifi.hrs.task.import *
hkDataset = HrsGetHKTask() (apid=1028, obsid=268494774)
HrsViewHKTask() (model="FM", apid=1028, hk=hkDataset)
```

To print to an ASCII table file information on periodic HK for LO during an observation:

```
lo = LO_HK_dump()
t(268494774,5,10)
```

To print to ASCII table file spectra taken with either WBS. ASCII table spectra also accompanied by a set of useful HK data

```
t = WBS_Export_ascii()
t(observation=268494774, plot="V")
```

To read and select data set for an spectrometer (wbs or hrs): **`observation=ReadObservation()(obsid=5861, spectrometer="wbs-h")`**

To retrieve system temperature and bandpass/gain determined in a Hot-Cold measurement:

```
allhot=SelecScans()(spectra,validbbs=Intlb([2223])
allcold=SelecScans()(spectra,validbbs=Intlb([2222])
hot=SingleHifiSpectrum(allhot)
cold=SingleHifiSpectrum(allcold)
ct=ComputeTsys()
noise=ct(hot,cold,eta_hot=0.96)
gamma=ct.gamma
```

To compute the system temperature using Hot-Cold load results from the HRS:

```
t = RadioMetry()
t(observation=268494774)
```

To compute system temperature using Hot-Cold load results from the WBS (across the IF band):

```
t = RWbs_HotCold_fast_bare()
t(observation=268494774)
```

To compute the WBS IF noise temperature (across the IF band):

```
t = Wbs_IFNoise()
ifout=t(observation=268494774)
```

From the retrieved data frames and HK data a HIFI timeline product can be created. A HifiTimelineProduct is an extension of HifiProduct. It also contains a summary table and a listing of the types of the HSDs.

`df = AccessDataFrameTask()(apids=[1030,1031],obsid=268435473)`

`htp=HifiTimelineProduct(df,hk)`

To get start and end time: **`FineTime start [INPUT, OPTIONAL, default=null]`** and **`FineTime end [INPUT, OPTIONAL, default=null]`**, respectively

To retrieve the HK packets containing Gascell information from a database:

- Use AccessPacketTask on apid 2025 for the provided obsid. You should also provide a database name (full version, including the server name). If you do not, then the query will be applied to the current database. This task instantiates AccessPacketTask only once.

Example:

```
obsids = [268449660, 268449664, 268449665, 268449670]
database = -'ilt_fm_2_prop@iccdb.sron.rug.nl 0 READ'
accessGascellHk = AccessGascellHk()
accessGascellHk.db = database
```

```
# for obsid in obsids:
hk_pack = accessGascellHk(obsid = obsid)
print -'obsid', obsid, -'has', len(hk_pack), -' gascell packets.'
```

To convert HK packets containing Gascell information into a table use the hk_tools.spy and tables_tools.spy routines to create the table. The list of the gascell HK parameters which are tabulated are given in the Chapter 3 of DP Commands.

To generate a table data set of the HK data: **TableDataset selected [OUTPUT, OPTIONAL, default=None]**

To create a tabla data set of HRS HK converted values (with as many columns as HRS HK mnemonics exist):

```
from herschel.hifi.hrs.task import *
hk = HrsGetHKTask()(apid=1028, obsid=268435622, type="All")
```

Helper class for defining the GUI components for AccessHkParamTask:

ahpt = AccessHkParamTask()

ahpt(db=ilt_fm_5_prop@iccdb.sron.rug.nl, obsid=268505167, apid=2017, params = ["FPU_room_temp", "FPU_shutter_temp"])

To store HK information in a PAL: **Hk_Store()(obsid=1732, lstore="HK_data", desc="Diplexer scan HK")**

As mentioned in the Chapter 4 of the Generic Pipeline description, some HK data are important for the Generic Pipeline. The following HK data are important when processing the generic pipeline:

- LO-frequency ("LoFrequency"): Generally, for grouping comparable spectra or to check and identify phases in the FSwitch observations.
- Chopper position ("Chopper"): To check and identify phases.
- Buffer ("buffer"): Alternative to check and identify phases.
- Observation time ("obs time"): For interpolating possibly drifting intensity scales (hot/cold measurements) or "background" obtained by blank sky measurements.
- Hot/Cold load temperature("hot_cold"): Used in the intensity calibration (determination of the bandpass and the system temperature).

A list of helpers class for defining the GUI components for the following generic tasks:

- AccessDataFrameTask: Allows access to DataFrames in the database. Helpers: AccessComponents, AccessDataFrameComponents.
- AccessPacketTask: Allows access to TmSourcePackets in the database. Helper: AccessComponents, AccessPacketComponents.
- AccessHkParamTask: Allows direct parameter retrieval from TmSourcePackets in the database(s). Helper: AccessHkParamComponent.

A description of the helpers is given in Chapter 3 of the User Manual.

Some parameter names:

- FPU_room_temp
- FPU_shooter_temp

Example of how to retrieve them:

```
from herschel.hifi.generic.task import AccessHkParamTask
ahpt = AccessHkParamTask()
ahpt(db=ilt_fm_5_prop@iccdb.sron.rug.nl, obsid=268505167, apid=2017, params =
["FPU_room_temp","FPU_shooter_temp"])
composite=ahpt.paramresult
print composite
```

14.7.2.1. Useful examples

1.- Accessing HK information for a given ObsID. Product: Table [time, FPU Temperature]

Warning

HcssConnection.get() task passes all the dataframes into the user's JIDE session. This can be memory intensive.

```
# Import needed packages
from herschel.access import *
from herschel.access.util import *
from herschel.binstruct import *
from herschel.pus import *
# Access HK packets associated with ObsID = 1400
pk = PacketAccess(1400)
# Connect to the default database to find the packets
hk_set = HcssConnection.get(pk)
# Create an empty Java array list -- needed for the PacketSequence routine below.
arrList = java.util.ArrayList()
# Loop around adding the HK dataset into the array arrList
for x in range(len(hk_set)):
arrList.add(hk_set[x])
# Look at our array
print arrList
# -...but to get something sensible we need packets in a time order.
pseq = PacketSequence(arrList)
# Get a listing of the parameter types contained in
print pseq
# Find packets in the sequence which contain information on temperatures within
the focal plane unit (FPU)
seq_FPU_Temp = pseq.select(TypeEquals("FPU_Temperatures"))
# Find out parameters contained in the selected packets by obtaining the HK
parameter names from the first selected packet in the sequence
par_FPU_Temp = seq_FPU_Temp[0].getParametersContained()
# Print out to the DP session the names of all the parameters contained
print par_FPU_Temp
# Choose the FPU Temperature parameter you want to get info on...and get a time
ordered set of HK data for it
# The output file plot_fpu_hk is a TableDataset with one column for time (a
Finetime of microseconds since 1 January 1958)
# and one for the value of the parameter (RAW rather than engineering value). Here
we choose the parameter FPU_b_body_top for the table
# output and get the converted values (in degrees K)
plot_fpu_hk = seq_FPU_Temp.getConvertedMeasures(["FPU_b_body_top"])
time = Double1d(plot_fpu_hk[0].data/1000000.0) # puts time into seconds
data = Double1d(plot_fpu_hk[1].data)
# Plot the timeline of the HK data over the time period of the observation
(obsid=1400) by plotting the table
p = PlotXY(time, data, style=Style(line=8, color=Color.black))
# Resize the window
p.height = 400
p.width=600
# Give a layer/legend name...
p[0].name="time plot"
# -...and add a title
p.title.text="FPU temperature"
```

2.- Accessing HK data for a given time period (covering several ObsIDs). Product: a plot time vs. Mixer Voltage.

Note

If you choose to sample long time period data this will prove to be memory intensive.

```
# Import needed packages for handling databases and HK data
from herschel.access import *
from herschel.access.util import *
from herschel.binstruct import *
from herschel.pus import *
# And this package deals with times.
from herschel.share.fltdyn.time import *
# Enter a start and stop time for HK information. We enter Java Dates, given as
year (-1900), Month (-1),
# day, hour, minute, second. Our start_time is therefore 01:10:00 on 25 October 2004
start_time = java.util.Date(104, 9, 25, 1, 10, 0)
# stop_time is 01:15:00 on the same day
stop_time = java.util.Date(104, 9, 25, 1, 15,0)
# Need to convert final numbers into a FineTime used in database.
start_1 = DateConverter.dateToFineTime(start_time)
# Date/time of start for plotted data
prod_date = DateConverter.fineTimeToDate(start_1)
# Ditto for stop time
stop_1 = DateConverter.dateToFineTime(stop_time)
end_date = DateConverter.fineTimeToDate(stop_1)
# Initialize some parameters
pk=0
hk_set = 0
# Get object ready for sorting packets.
pseq = PacketSequence()
# Set up the query for accessing packets of HK data. Here we ask for packets with
an APID of 1026, which carries
# HIFI HK data. The database identified by the user's properties is accessed for
packets of this type
# between the given start and stop FineTimes.
pk = PacketAccess(1026,start_1,stop_1)
# Now we know where to look, we can get the packets! First we create an array with
the packets in
hk_set = HcssConnection.get(pk)
# -...then we loop over the array to get the contents and
# put packets into our packet sequence
for x in range(len(hk_set)):
  pseq.add(PusTmSourcePacket(hk_set[x].getContents()))
# Now we get the parameters in the packets that we can plot.
seq_HIFI_HK = pseq.select(TypeEquals("HIFI_HK_rev_3"))
# Pick out some of them
mnemonics = ["HF_AH1_MXMG_V", -"HF_AV1_MXMG_V"]
# -...and get their converted (physical unit) measurements. -"plot_HIFI_HK" is a
TableDataset with a first column measuring time
# and the next 2 columns holding the HK parameter values at those times. We can
now plot any of the parameters versus
# time, or against each other, by picking out the appropriate column of the table.
Plot_HIFI_HK = seq_HIFI_HK.getConvertedMeasures(mnemonics)
# This is what to do to set up the plot. Since time is in microseconds we convert
it to seconds first.
# Get the first column and divide by 1 million
time = plot_HIFI_HK[0].data/1000000.0
# Measure time on the plot from the beginning of the observation....We subtract
the initial time value
plot_time = time -- time[0]
# Plot the two voltages contained in columns 2 and 4
h_voltage = plot_HIFI_HK[1].data
v_voltage = plot_HIFI_HK[2].data
# Plot the data
p = PlotXY(plot_time, h_voltage, style=Style(line=8, color=Color.black))
# Resize the window
p.height = 400
p.width=600
# Change the legend
p[0].name = -"H Mixer Plot"
# Change the axis labels...
```

```
p.xaxis.title.text="Time (hours)"
p.yaxis.title.text="Mixer voltage [V]"
# -...and add a title
p.title.text="Plot of Mixer Voltages. Start: -"+str(prod_date)+\
  -"End: -"+str(end_date)
# Now we can also overlay the second voltage trend in blue.
p[1]=LayerXY(plot_time, v_voltage, name= -"V Mixer Plot", \
 style=Style(color=Color.blue))
```

14.7.3. Viewing Housekeeping

There are several ways to display the dataframes:

1.-

```
dd = DisplayDataFrameTask()
dd.dataframe=df[0]
dd.gui = 1
```

2.-

```
browse_df = BrowseDfQuery()(gui=1)
#feedback dataframes back into your session using:
dfs = browse_df.query
df = browse_df.selected
# feedback dataframes back into your session using:
dfs = browse_df.query
df = browse_df.selected
```

3.-

```
#to pass an existing store this task should reuse:
dfs = AccessDataFrameTask()(store=myStore, -.....)
```

The BrowseHK task is used to display HK data associated with an observation stored in any accessable database. To change to a new database type the name of the database into the local database text field located on the database tab. The fastest way to access data from the database is to lookup data using a specific obsid. If the times in the time selector panel are the same the script will automatically get the begin and end time of the observation and add it to the query string. If the times are different this will override the automatic process and search the database based on those times. The execute button is used to aquire data from the database. The exit button is used to close the application.

```
# interactive mode (Database: ilt_dm_10 obsid: 268435762)
# interactive mode (Database: ilt_fm_2 obsid: 268435480)
from herschel.hifi.generic.task.BrowseHk import *
hkBrowser = BrowseHk()
hkBrowser()
# This only brings up the browser. Once selections are made, and the
# execute button is pressed, then one can export pk-params(s) in their
# session
#
# resulting_pks = hkBrowser.query -// array[TmSourcePacket]!!
# selected_pk = hkBrowser.selected -// TableDataset!!
```

The array of TmSourcePackets (HK data) returned from the query to the database: **`Array(TmSourcePacket) query [OUTPUT, OPTIONAL, default=None]`**

To create a TableDataset of HRS HK converted values: **`hk = HrsGetHKTask() (apid=1028, obsid=268435622, type="All")`** or alternatevely **`TableDataset selected [OUTPUT, OPTIONAL, default=None]`**

To plot a stream of HK data:

```
#In a new -.py file add the lines (call it qlaHkPlot.py)
from herschel.hifi.generic.process.hkplot import *
DataFlowManager(hkPlot())
Thread.sleep(315360000)  #one year in seconds
#Then from the terminal window type the following
jylaunch qlaHkPlot.py
```

To plot and export a time series of HIFI HK data for a selected ObsID

```
HkPlotter()(obsid = obsid, apid = apid)
HkPlotter()(obsid, apid)
```

There is an alternative task to plot HK data, which is interactive, in order to analyze and plot WBS functional test: WbsCheckFt .