NHSC/PACS Web Tutorials

Running the PACS Spectrometer pipeline for CHOP/NOD Mode

PACS-301

Pipeline Level 0 to 1 processing

Prepared by Dario Fadda
Updated by Babar Ali, February 2013
Updated by Steve Lord, Oct 2013

Introduction

This tutorial will guide you through the interactive spectrometer pipeline from loading raw data into HIPE to obtain calibrated data with astrometry in the case of chop/nod mode.

Pre-requisites

The following tutorials should be read before this one:

- ***PACS-101****: How to use these tutorials.*
- ***PACS-102****: Accessing and storing data from the Herschel Science Archive*
- ***PACS-103****: Loading scripts*

Sequel: *PACS-302* – Level 1-2 processing

Overview

Step 1 Check HIPE version and your local memory

Step 2 Set up script for the particular Obs ID

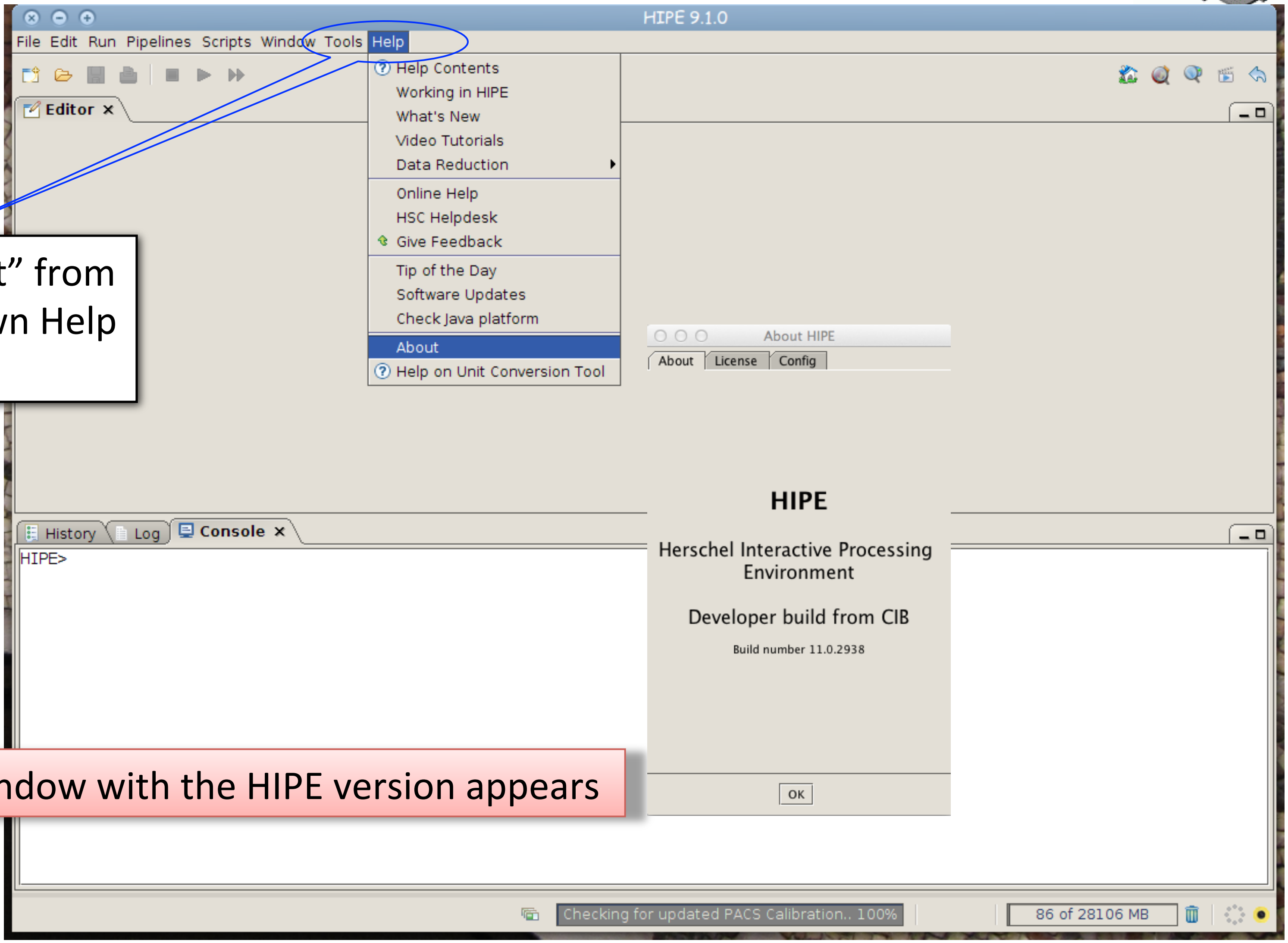
Step 3 Run the 0 → 0.5 pipeline

Step 4 Run the 0.5 → 1 pipeline

Step 1

Check HIPE version and memory allocation

The version used for the tutorial is 11.0 build 2938



HIPE>

HIPE

Herschel Interactive Processing Environment

Developer build from CIB

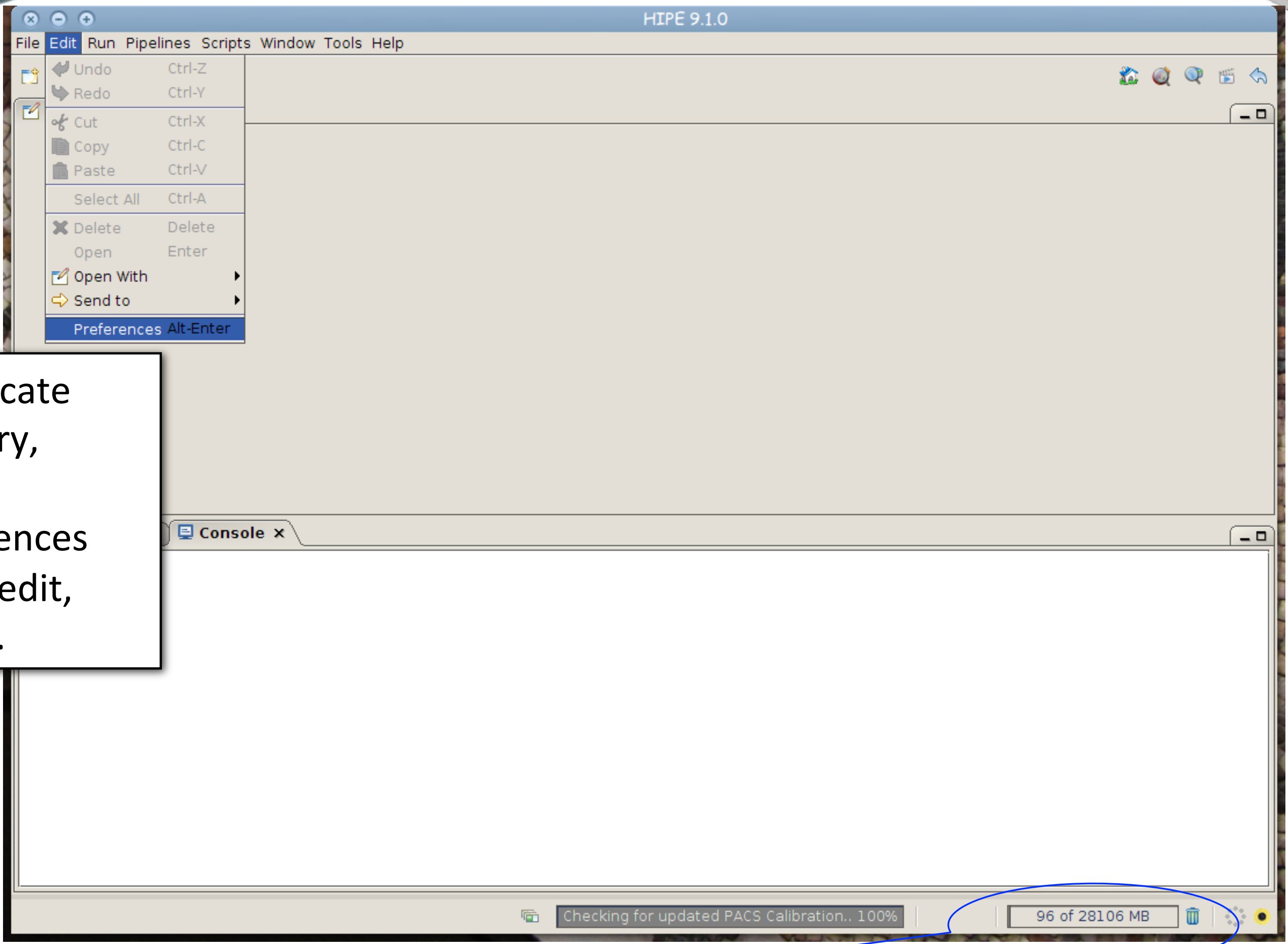
Build number 11.0.2938

Checking for updated PACS Calibration.. 100%

Select “about” from the drop down Help menu

A pop-up window with the HIPE version appears

nhsc.ipac.caltech.edu/helpdesk



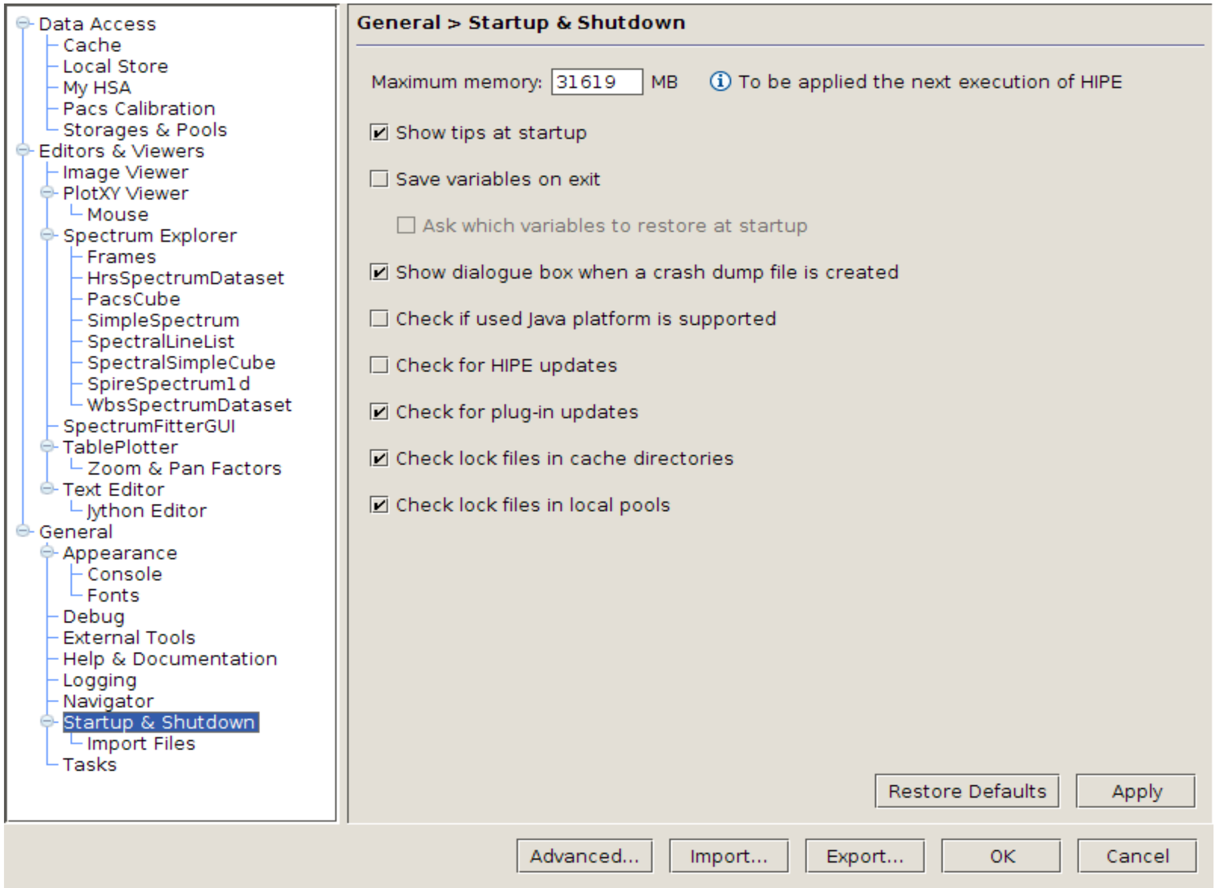
Checking for updated PACS Calibration.. 100%

96 of 28106 MB

To allocate memory, select preferences under edit, then ...

N.b.:, Memory used and available

…click on “Startup & Shutdown” and change the amount of memory



General > Startup & Shutdown

Maximum memory: 31619 MB ⓘ To be applied the next execution of HIPE

☑ Show tips at startup

☐ Save variables on exit

☐ Ask which variables to restore at startup

☑ Show dialogue box when a crash dump file is created

☐ Check if used Java platform is supported

☐ Check for HIPE updates

☑ Check for plug-in updates

☑ Check lock files in cache directories

☑ Check lock files in local pools

The allocated memory should be a bit smaller than the total RAM of your computer. (e.g. 7.5 out of 8.0 Gbytes)
You must exit and restart HIPE to obtain the new amount of memory.

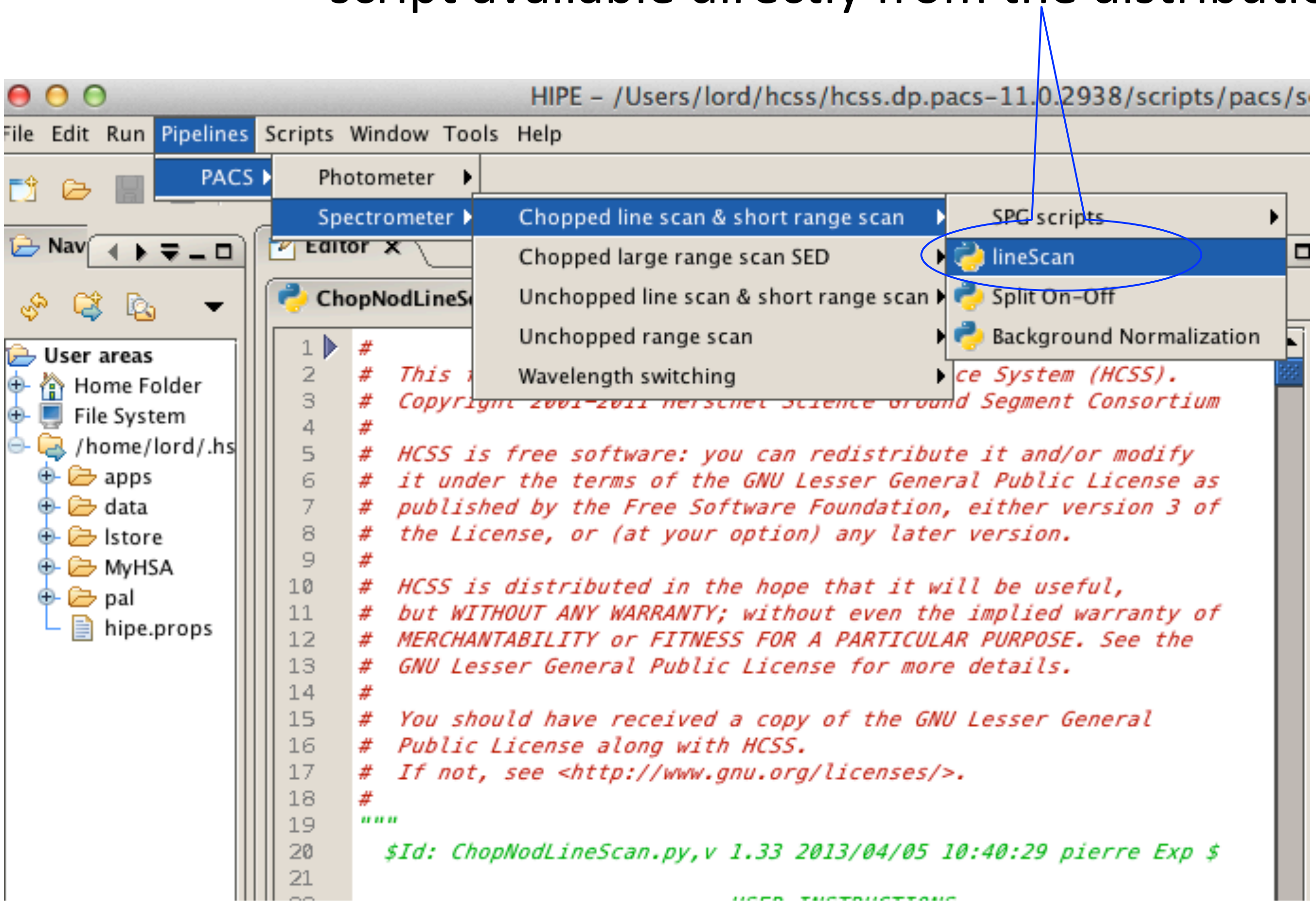
Step 2

Setup

Load pipeline script; load observation; check your data; and select the camera

Loading the script

The “linescan” script used in this tutorial corresponds to the script available directly from the distribution.

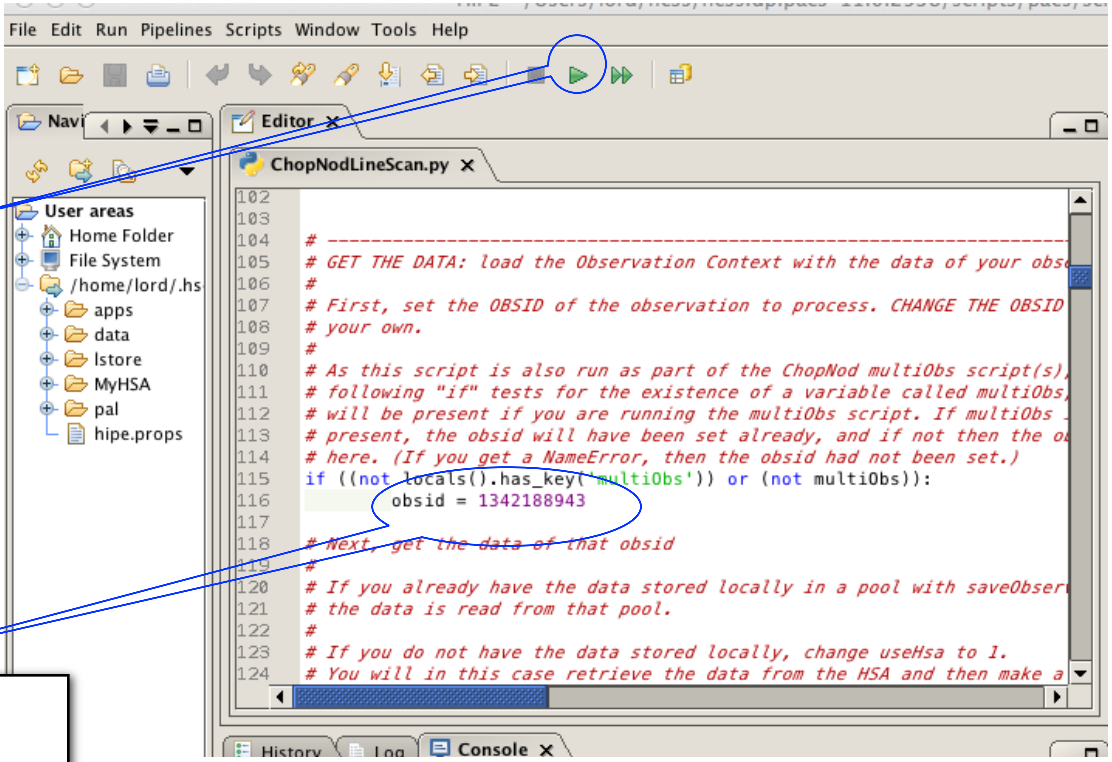


```
#
#  This                                    ce System (HCSS).
#  Copyright 2001-2011 Herschel Science Ground Segment Consortium
#
#  HCSS is free software: you can redistribute it and/or modify
#  it under the terms of the GNU Lesser General Public License as
#  published by the Free Software Foundation, either version 3 of
#  the License, or (at your option) any later version.
#
#  HCSS is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
#  GNU Lesser General Public License for more details.
#
#  You should have received a copy of the GNU Lesser General
#  Public License along with HCSS.
#  If not, see <http://www.gnu.org/licenses/>.
#
""""
  $Id: ChopNodLineScan.py,v 1.33 2013/04/05 10:40:29 pierre Exp $

```

Loading the observation

Once the script is loaded, one simply steps through the lines to execute it. But first modify it for OBSID of the observation desired. In the case of this tutorial, the observation was already saved into a pool in the user's local ~/.hcss/lstore directory (created when first installing HIPE). So one modifies the obsid in the script and clicks through using the green arrow….

Hit the green arrow to step through the entire script

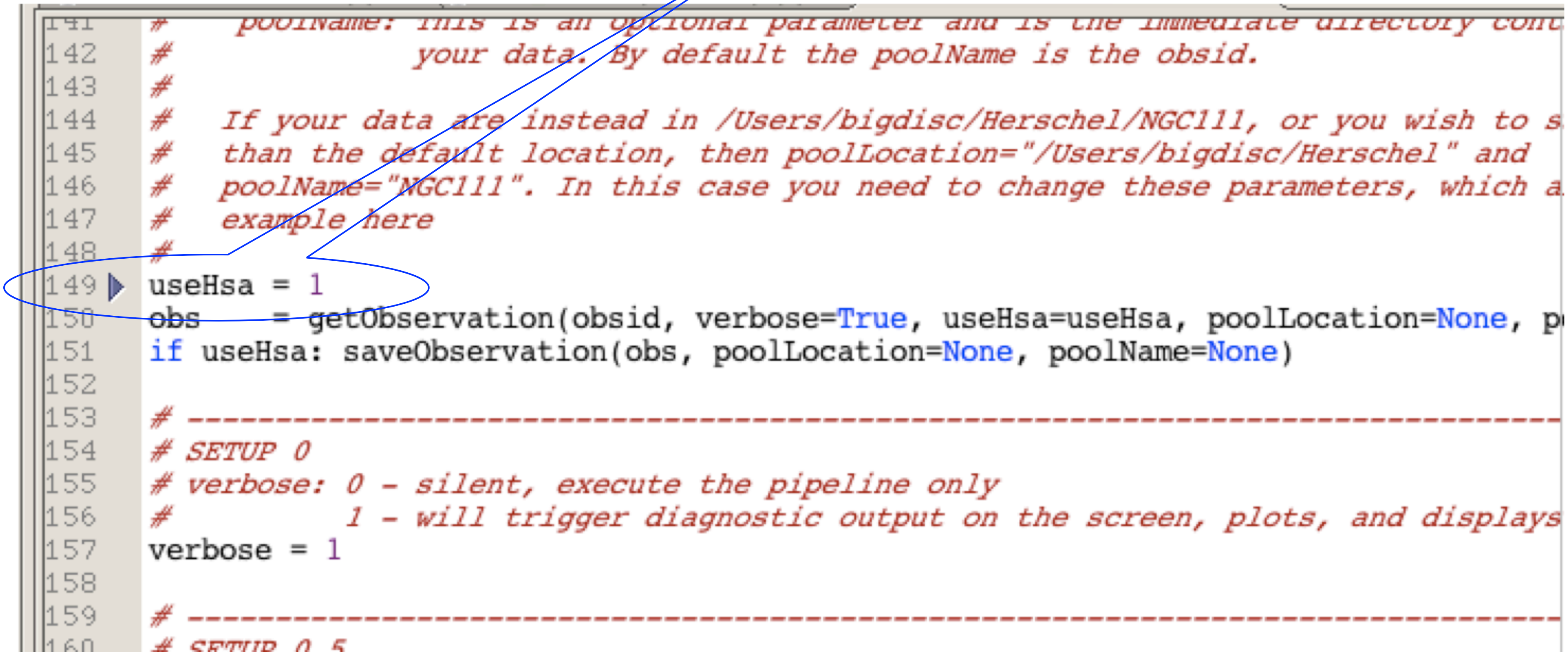


```
# -------------------------------------------------------------------
# GET THE DATA: load the Observation Context with the data of your obs
#
# First, set the OBSID of the observation to process. CHANGE THE OBSID
# your own.
#
# As this script is also run as part of the ChopNod multiObs script(s),
# following "if" tests for the existence of a variable called multiObs,
# will be present if you are running the multiObs script. If multiObs
# present, the obsid will have been set already, and if not then the o
# here. (If you get a NameError, then the obsid had not been set.)
if ((not locals().has_key('multiObs')) or (not multiObs)):
        obsid = 1342188943

# Next, get the data of that obsid
#
# If you already have the data stored locally in a pool with saveObser
# the data is read from that pool.
#
# If you do not have the data stored locally, change useHsa to 1.
# You will in this case retrieve the data from the HSA and then make a
```

Modify this line. Here we set obsid to 1342186799.

Loading the observation

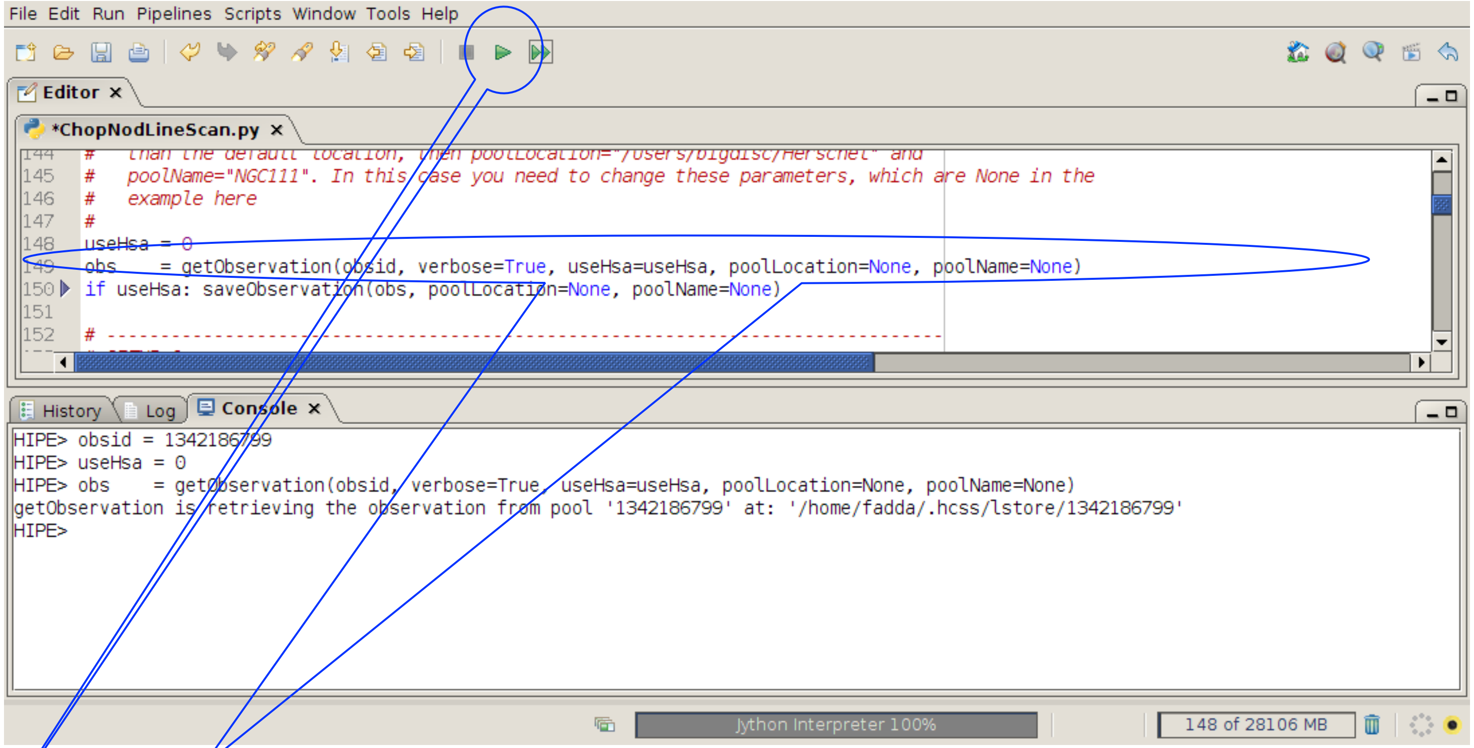
If the data is not stored as a local pool, you want to tell the script to acquire the data from HSA. In this case edit the line to useHsa=1



```
#    poolName: This is an optional parameter and is the immediate directory cont
#              your data. By default the poolName is the obsid.
#
#   If your data are instead in /Users/bigdisc/Herschel/NGC111, or you wish to s
#   than the default location, then poolLocation="/Users/bigdisc/Herschel" and
#   poolName="NGC111". In this case you need to change these parameters, which a
#   example here
#
useHsa = 1
obs    = getObservation(obsid, verbose=True, useHsa=useHsa, poolLocation=None, p
if useHsa: saveObservation(obs, poolLocation=None, poolName=None)

# ------------------------------------------------------------------------------
# SETUP 0
# verbose: 0 - silent, execute the pipeline only
#          1 - will trigger diagnostic output on the screen, plots, and displays
verbose = 1

# ------------------------------------------------------------------------------
# SETUP 0.5
```

Loading the observation

Next step, we load the observational context (a structure containing all the observational data, information about them and calibration data).



```
#   than the default location, then poolLocation="/Users/bigdisc/Herschel" and
#   poolName="NGC111". In this case you need to change these parameters, which are None in the
#   example here
#
useHsa = 0
obs    = getObservation(obsid, verbose=True, useHsa=useHsa, poolLocation=None, poolName=None)
if useHsa: saveObservation(obs, poolLocation=None, poolName=None)

# -----------------------------------------------------------------------------
```

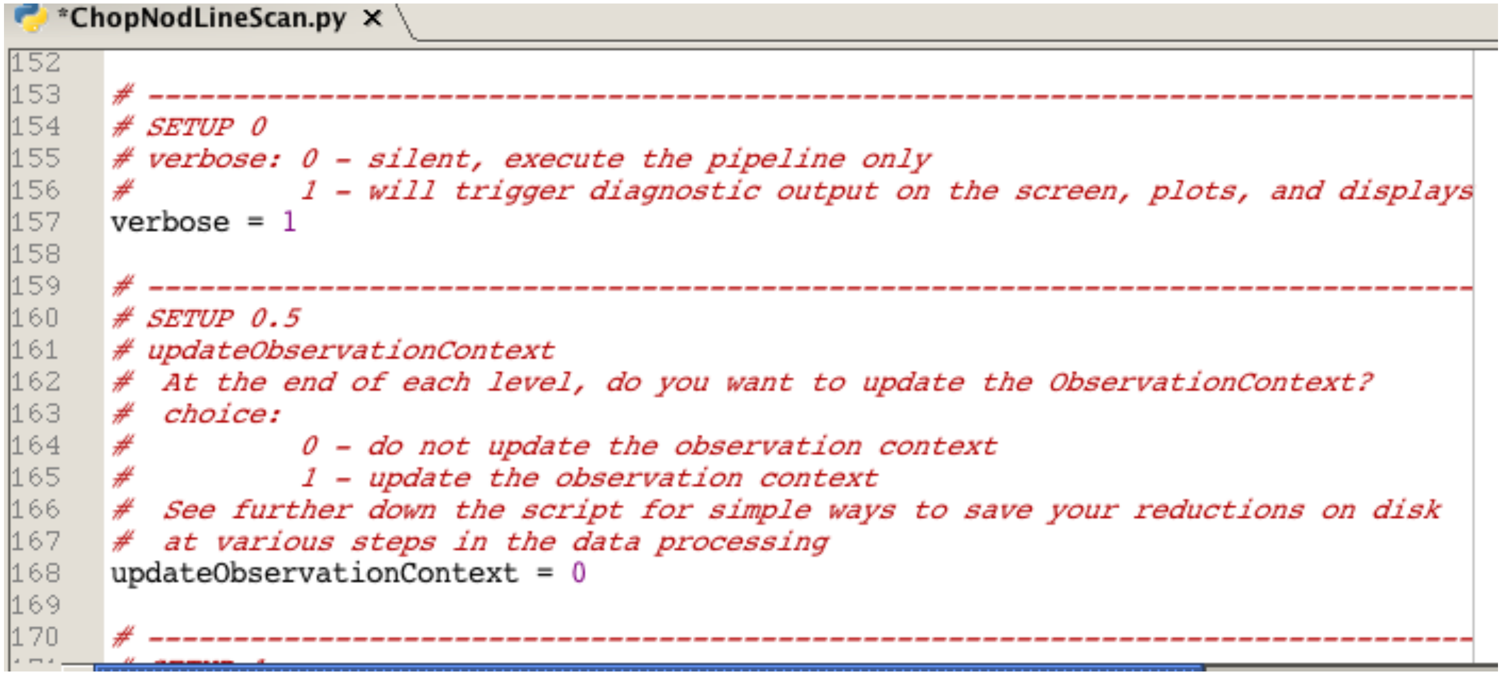
```
HIPE> obsid = 1342186799
HIPE> useHsa = 0
HIPE> obs    = getObservation(obsid, verbose=True, useHsa=useHsa, poolLocation=None, poolName=None)
getObservation is retrieving the observation from pool '1342186799' at: '/home/fadda/.hcss/lstore/1342186799'
HIPE>
```

Click through this line using the green arrow.

Observation Selection

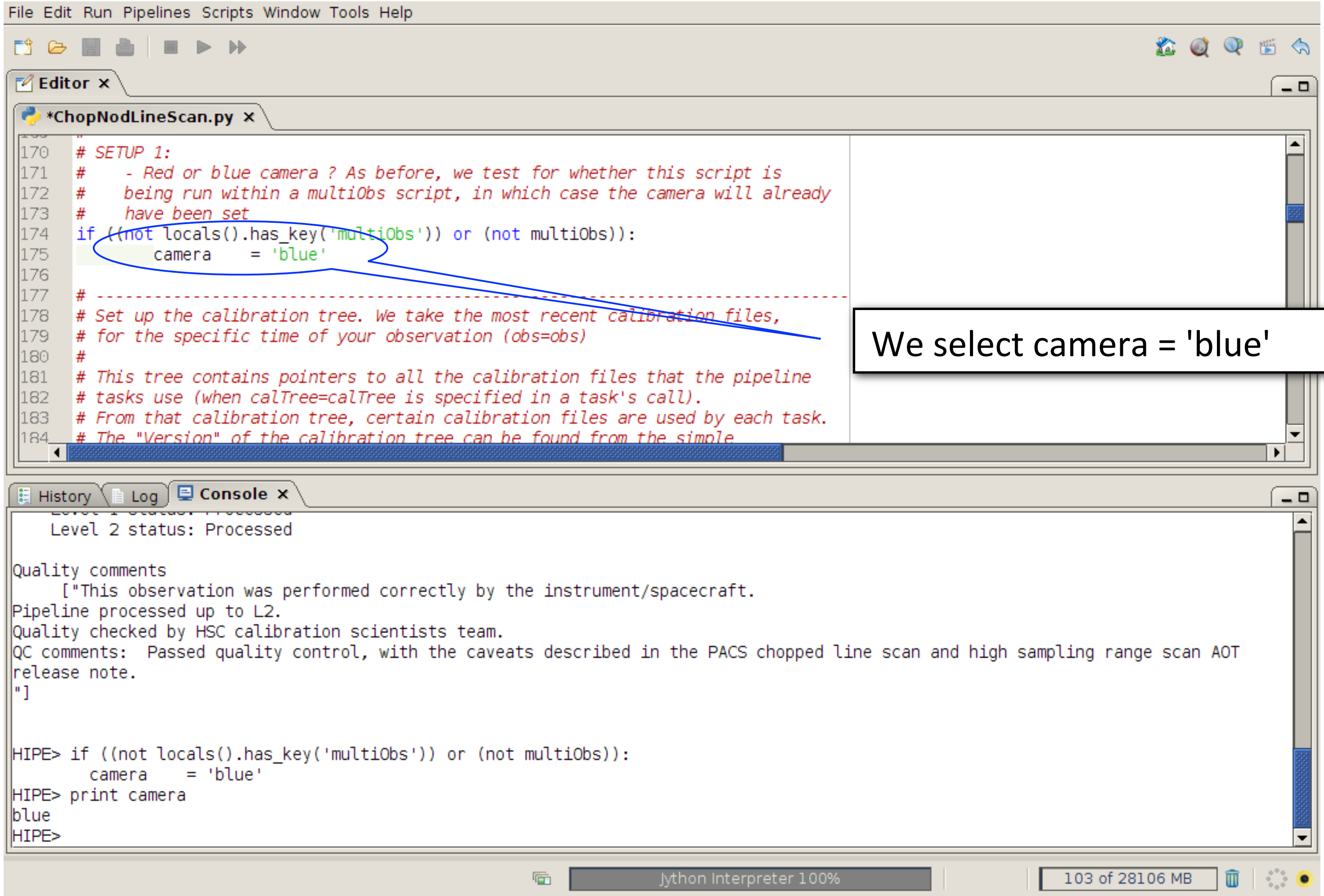
For this data set, we will concern ourselves with the line observed in the PACS blue channel.

Click through (using the green arrow) the remaining lines, choosing your preferences as described in the comments until you get to the channel section.



```
# ----------------------------------------------------------------------------
# SETUP 0
# verbose: 0 - silent, execute the pipeline only
#          1 - will trigger diagnostic output on the screen, plots, and displays
verbose = 1

# ----------------------------------------------------------------------------
# SETUP 0.5
# updateObservationContext
#  At the end of each level, do you want to update the ObservationContext?
#  choice:
#          0 - do not update the observation context
#          1 - update the observation context
#  See further down the script for simple ways to save your reductions on disk
#  at various steps in the data processing
updateObservationContext = 0

# ----------------------------------------------------------------------------
```

Setting the camera



```
# SETUP 1:
#    - Red or blue camera ? As before, we test for whether this script is
#    being run within a multiObs script, in which case the camera will already
#    have been set
if ((not locals().has_key('multiObs')) or (not multiObs)):
        camera    = 'blue'

# -----------------------------------------------------------------------------
# Set up the calibration tree. We take the most recent calibration files,
# for the specific time of your observation (obs=obs)
#
# This tree contains pointers to all the calibration files that the pipeline
# tasks use (when calTree=calTree is specified in a task's call).
# From that calibration tree, certain calibration files are used by each task.
# The "Version" of the calibration tree can be found from the simple
```

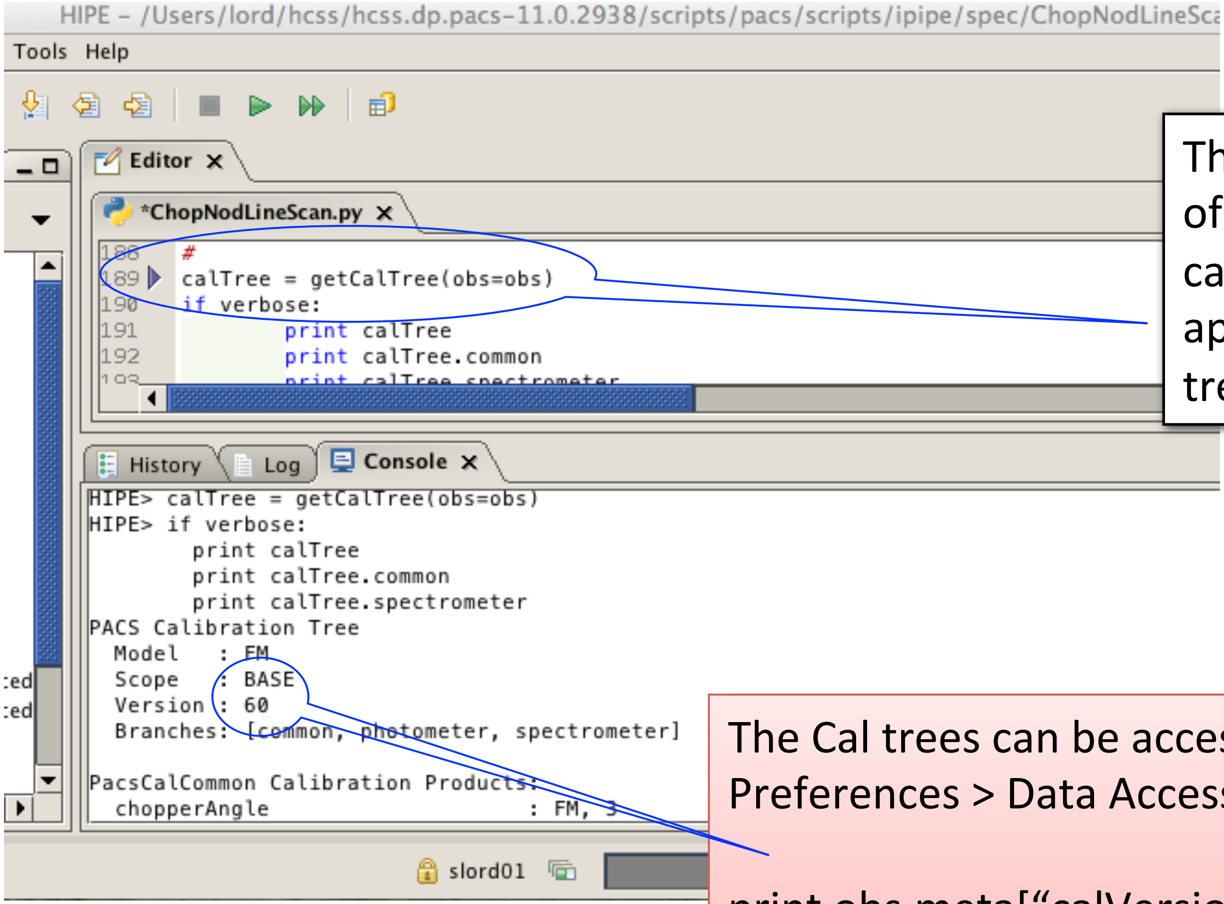
```
    Level 2 status: Processed

Quality comments
     ["This observation was performed correctly by the instrument/spacecraft.
Pipeline processed up to L2.
Quality checked by HSC calibration scientists team.
QC comments:  Passed quality control, with the caveats described in the PACS chopped line scan and high sampling range scan AOT
release note.
"]


HIPE> if ((not locals().has_key('multiObs')) or (not multiObs)):
        camera    = 'blue'
HIPE> print camera
blue
HIPE>
```

We select camera = 'blue'

After selecting the camera, we can check what camera we selected by simply printing: "print camera"

Setting the calibration tree

Finally, we set the calibration tree.



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

```
HIPE> calTree = getCalTree(obs=obs)
HIPE> if verbose:
        print calTree
        print calTree.common
        print calTree.spectrometer
PACS Calibration Tree
  Model    : FM
  Scope    : BASE
  Version  : 60
  Branches: [common, photometer, spectrometer]

PacsCalCommon Calibration Products:
  chopperAngle                          : FM, 3
```

This reads the time stamp of our obs and applies the calibration from the appropriate calibration tree.

The Cal trees can be accessed and updated from Preferences > Data Access > Pacs Calibration.

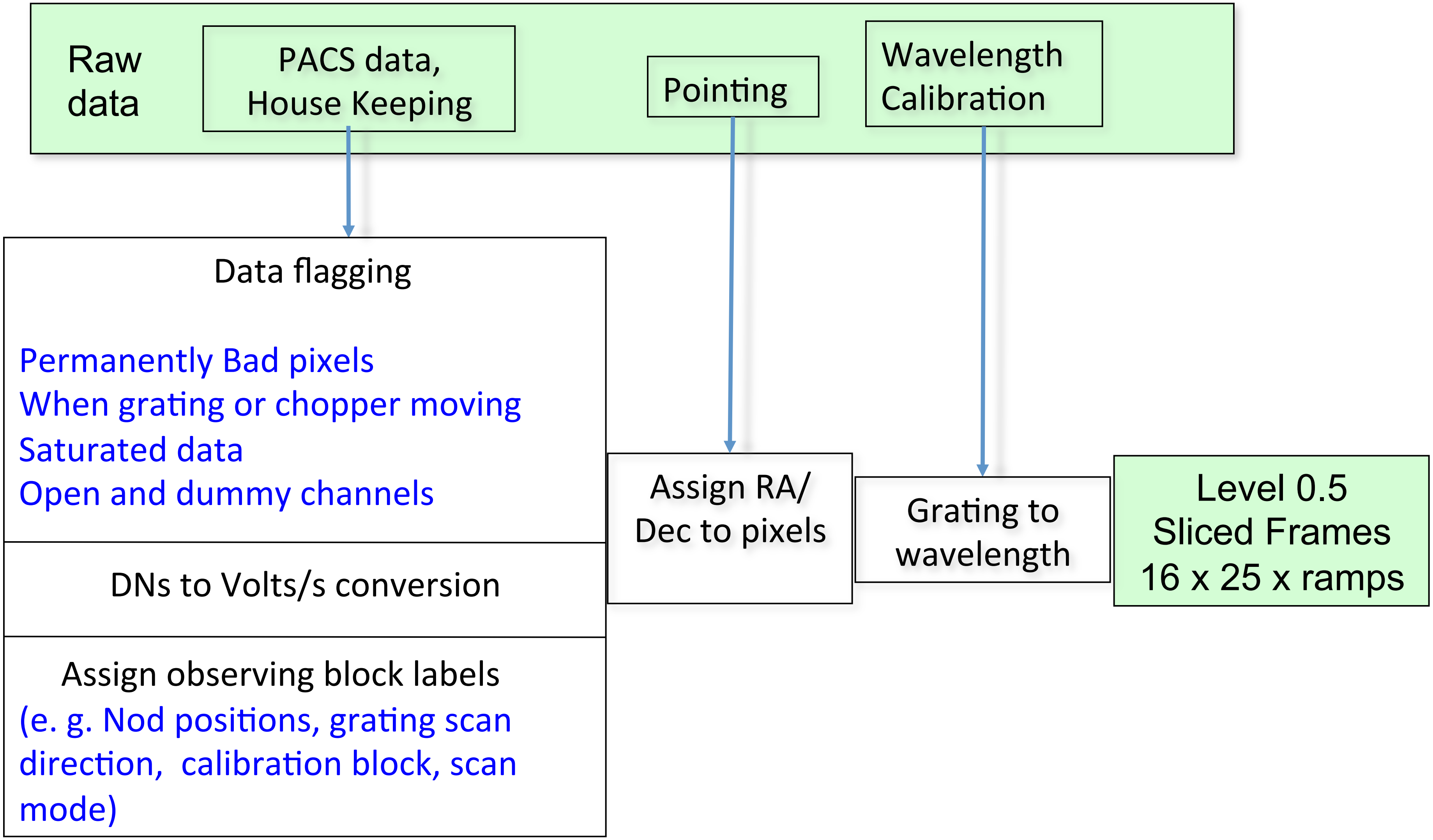
print obs.meta[“calVersion”] shows the calibration used in current observation.

Step 3

Run the 0 → 0.5 pipeline

Basic calibration (pointing,
wavelength calibration, slicing)

Level 0 → 0.5



Raw data

PACS data, House Keeping

Pointing

Wavelength Calibration

Data flagging

Permanently Bad pixels
When grating or chopper moving
Saturated data
Open and dummy channels

DNs to Volts/s conversion

Assign observing block labels
(e. g. Nod positions, grating scan direction, calibration block, scan mode)

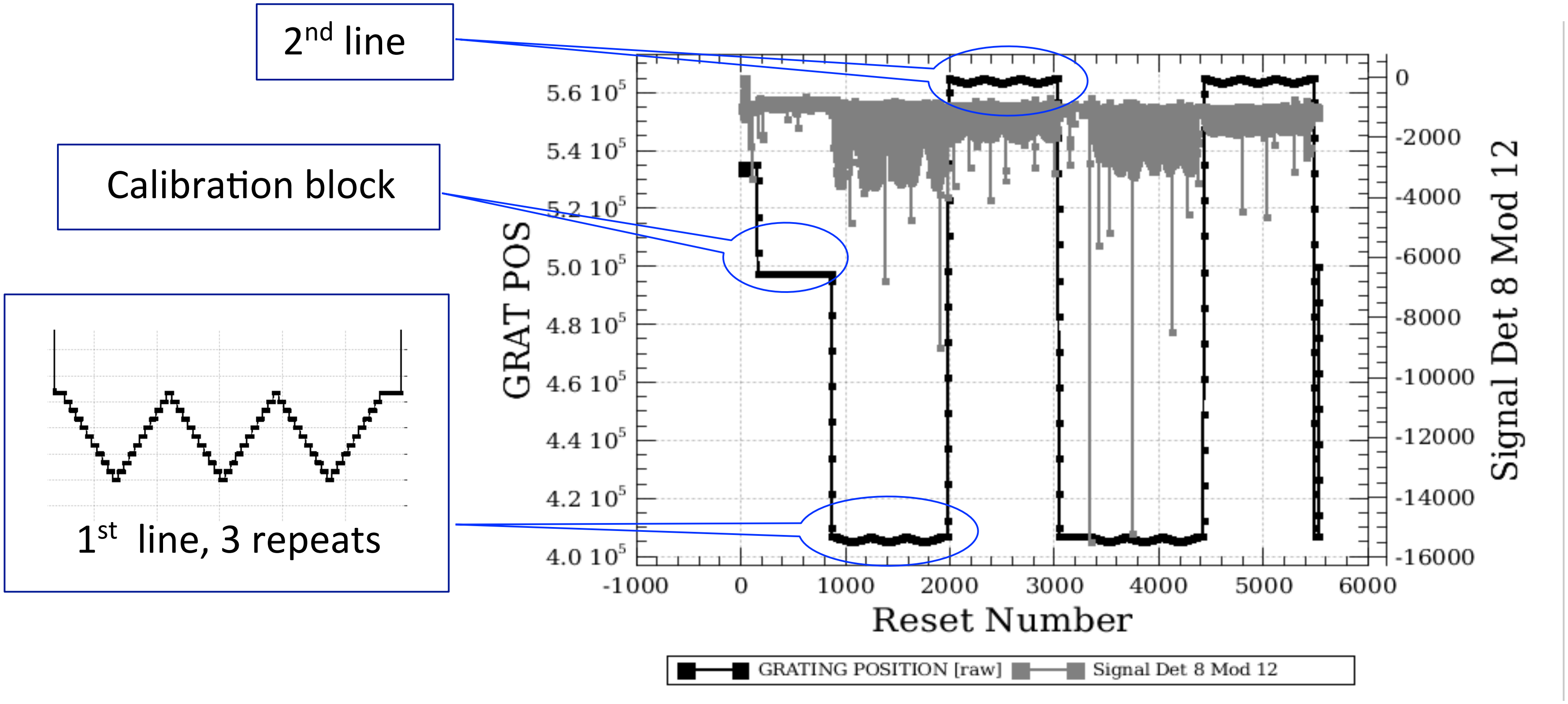
Assign RA/Dec to pixels

Grating to wavelength

Level 0.5
Sliced Frames
16 x 25 x ramps

Check: level 0

From now on, we will step through the script line by line using the green arrow on the menu bar. The first step consists in extracting the 0-level products from the observation context.



2nd line

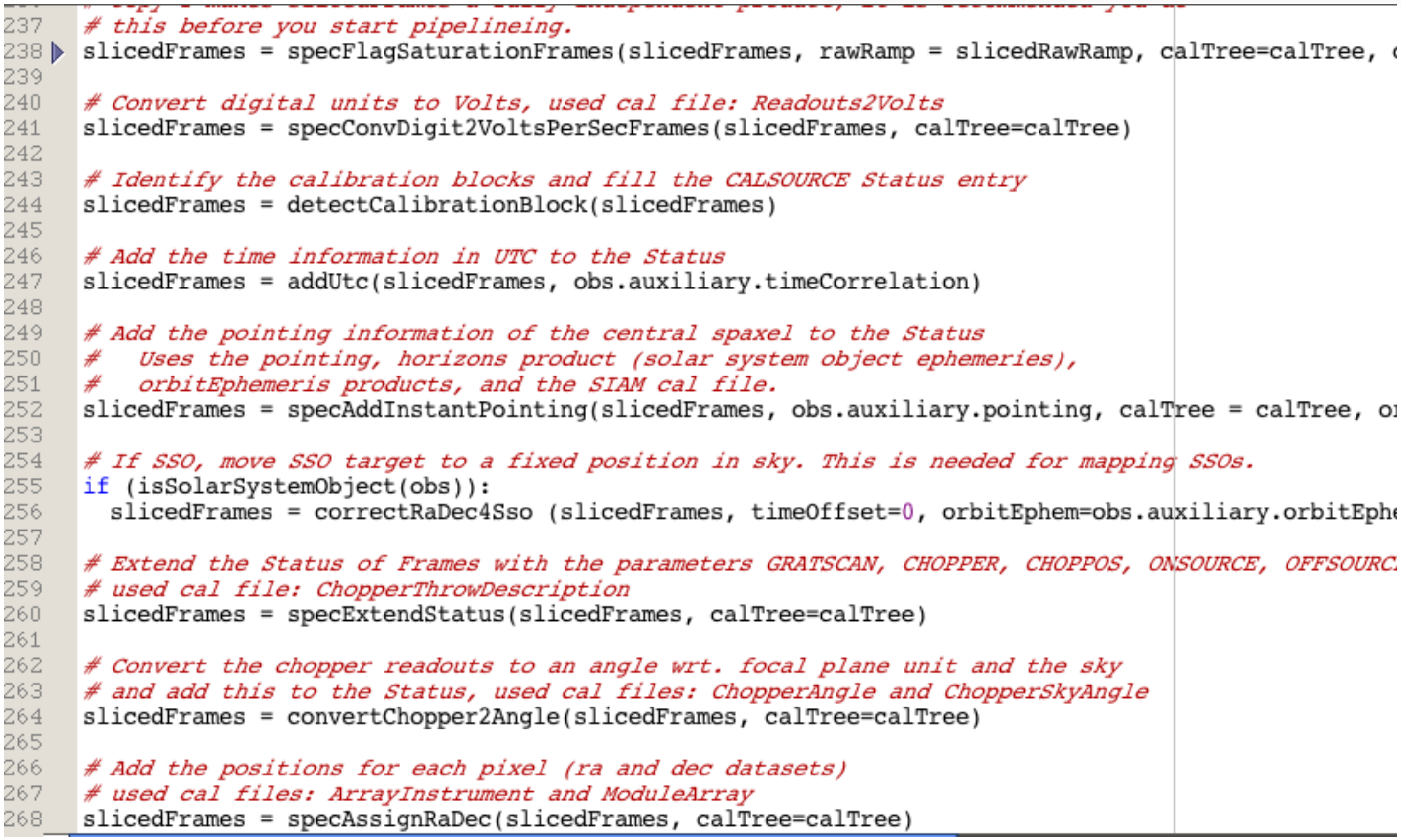
Calibration block

1st line, 3 repeats

The plot appears after: ‘p0 = slicedSummaryPlot(slicedFrames,signal=1)’
In our case, after the calibration block, we can identify two different lines observed 3 times in the two nod positions.

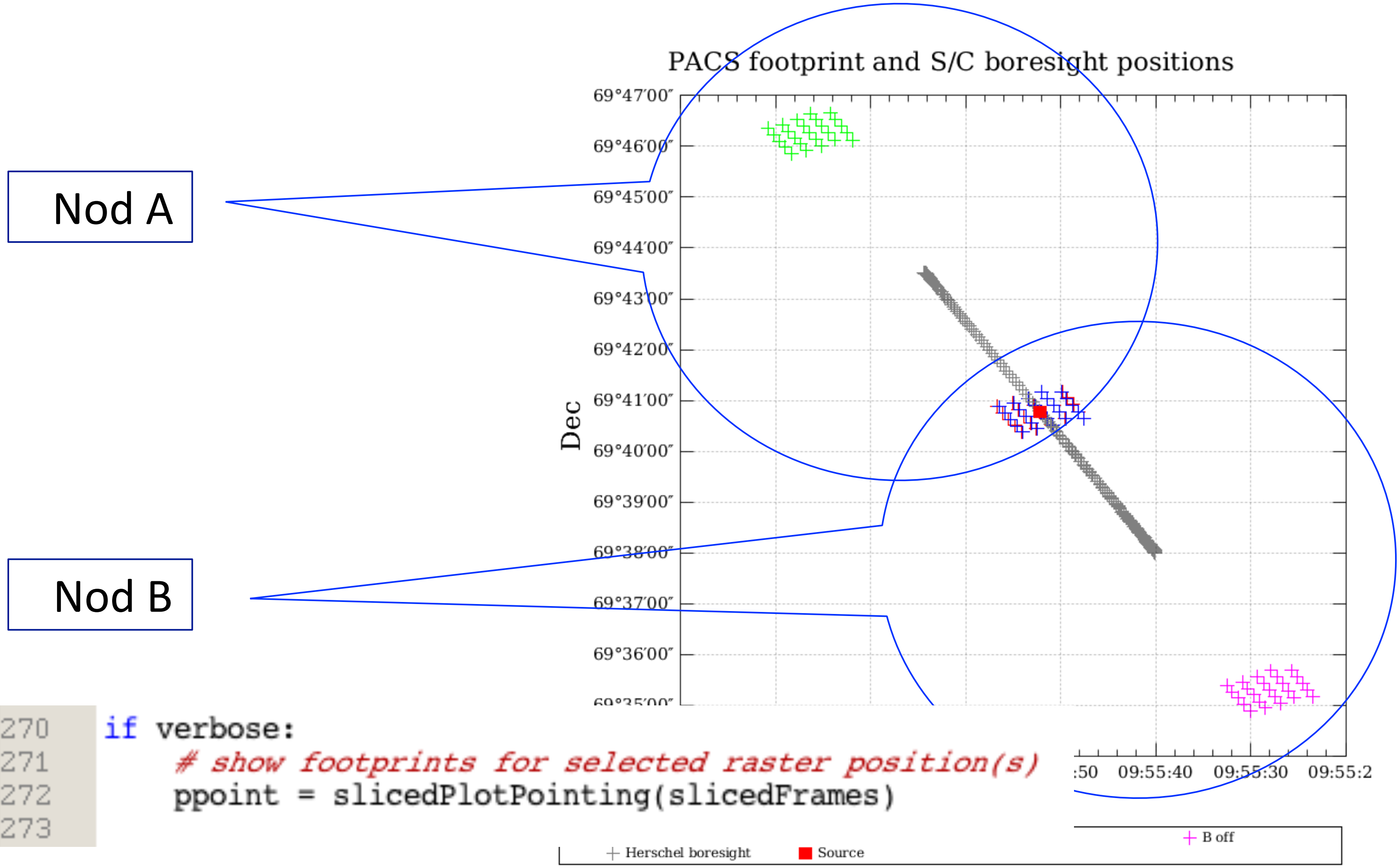
Continue …

With remaining Level 0 to 0.5 processing steps as outlined in slide 17. Step through with the green arrow.



```
# this before you start pipelineing.
slicedFrames = specFlagSaturationFrames(slicedFrames, rawRamp = slicedRawRamp, calTree=calTree, 

# Convert digital units to Volts, used cal file: Readouts2Volts
slicedFrames = specConvDigit2VoltsPerSecFrames(slicedFrames, calTree=calTree)

# Identify the calibration blocks and fill the CALSOURCE Status entry
slicedFrames = detectCalibrationBlock(slicedFrames)

# Add the time information in UTC to the Status
slicedFrames = addUtc(slicedFrames, obs.auxiliary.timeCorrelation)

# Add the pointing information of the central spaxel to the Status
#   Uses the pointing, horizons product (solar system object ephemeries),
#   orbitEphemeris products, and the SIAM cal file.
slicedFrames = specAddInstantPointing(slicedFrames, obs.auxiliary.pointing, calTree = calTree, o

# If SSO, move SSO target to a fixed position in sky. This is needed for mapping SSOs.
if (isSolarSystemObject(obs)):
  slicedFrames = correctRaDec4Sso (slicedFrames, timeOffset=0, orbitEphem=obs.auxiliary.orbitEphe

# Extend the Status of Frames with the parameters GRATSCAN, CHOPPER, CHOPPOS, ONSOURCE, OFFSOURC
# used cal file: ChopperThrowDescription
slicedFrames = specExtendStatus(slicedFrames, calTree=calTree)

# Convert the chopper readouts to an angle wrt. focal plane unit and the sky
# and add this to the Status, used cal files: ChopperAngle and ChopperSkyAngle
slicedFrames = convertChopper2Angle(slicedFrames, calTree=calTree)

# Add the positions for each pixel (ra and dec datasets)
# used cal files: ArrayInstrument and ModuleArray
slicedFrames = specAssignRaDec(slicedFrames, calTree=calTree)
```

Check: footprint



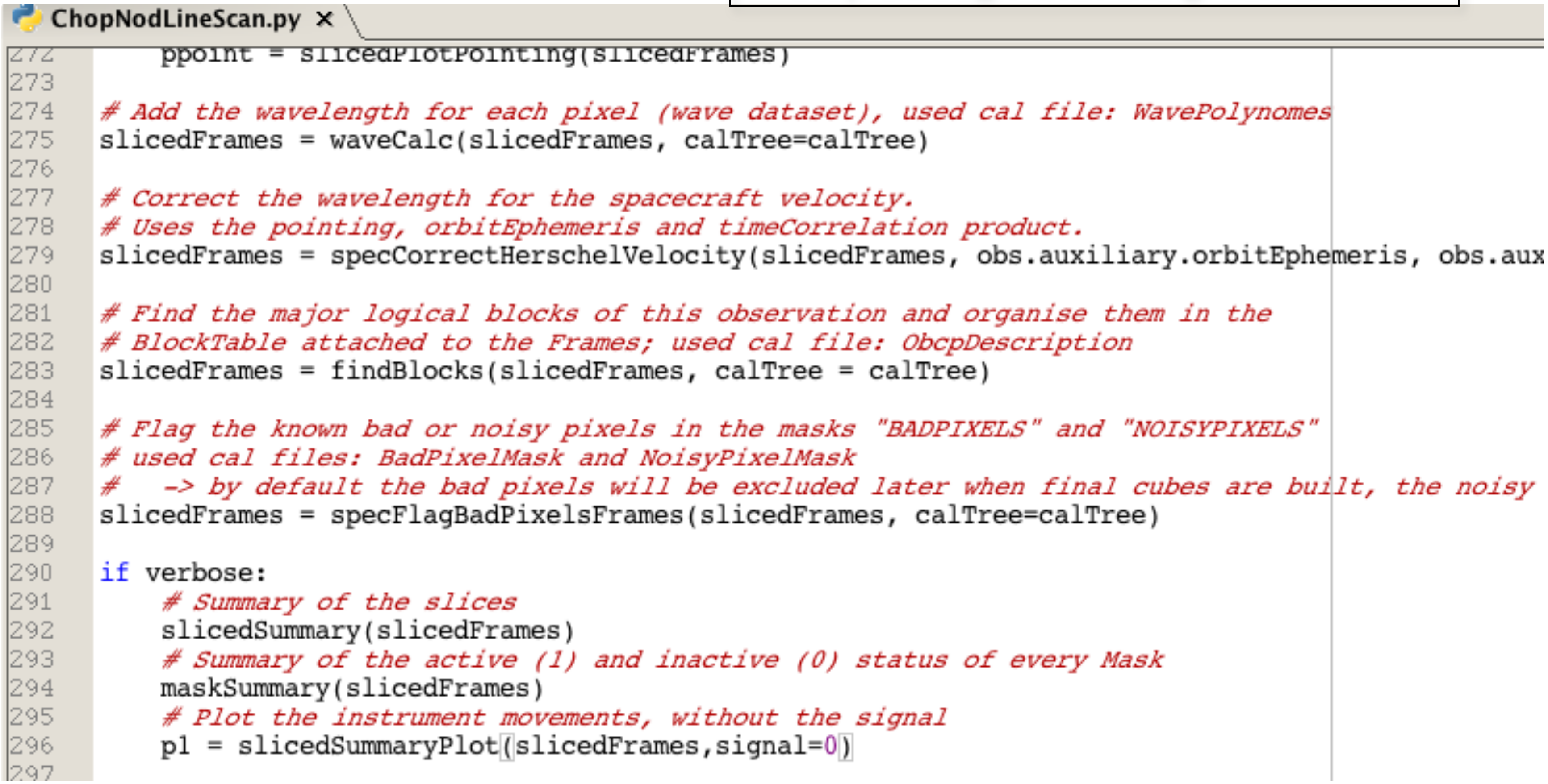
Nod A

Nod B

```
if verbose:
    # show footprints for selected raster position(s)
    ppoint = slicedPlotPointing(slicedFrames)
```

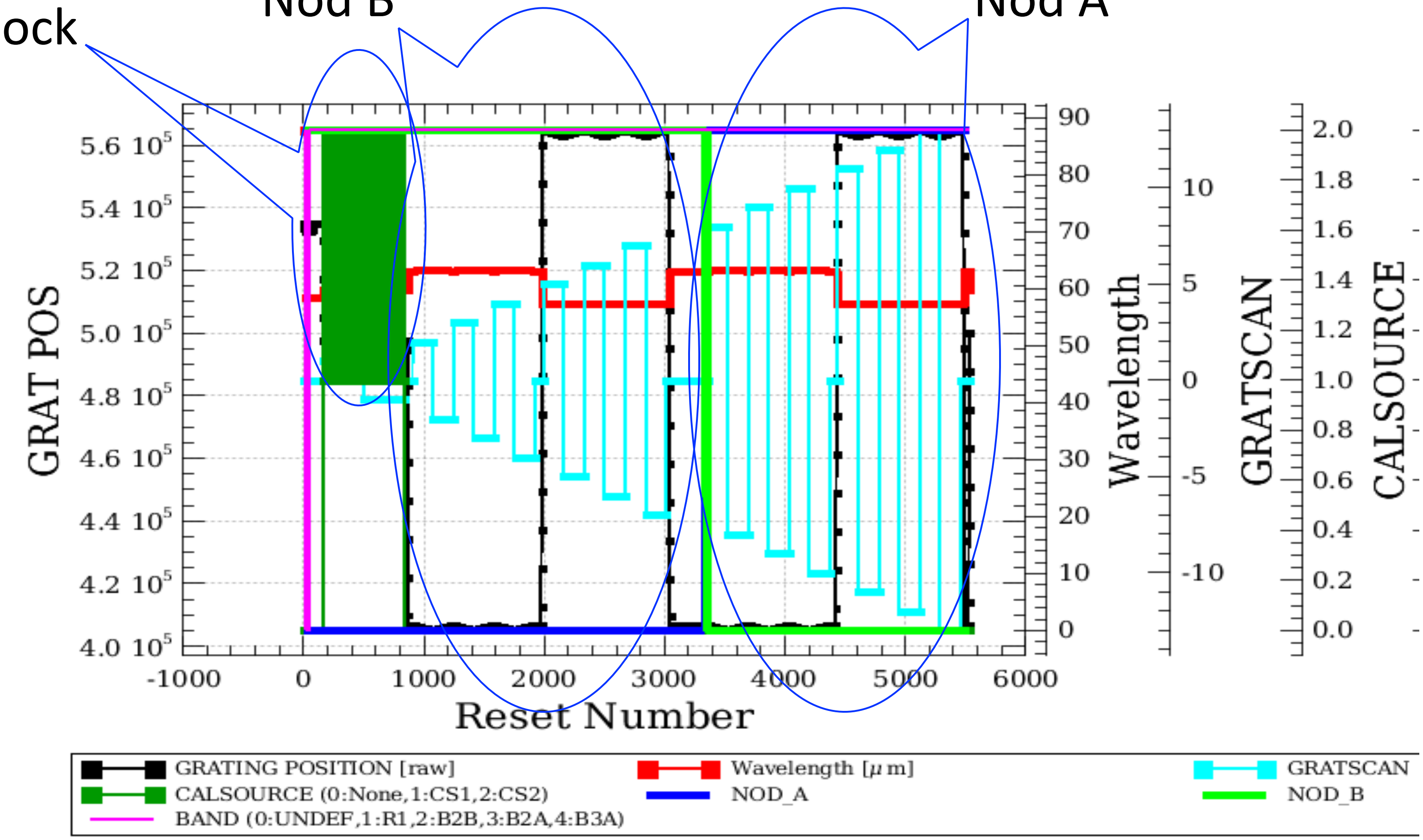
Continue …

With remaining Level 0 to 0.5 processing steps as outlined in slide 17. Step through with the green arrow.



```
    ppoint = slicedPlotPointing(slicedFrames)

# Add the wavelength for each pixel (wave dataset), used cal file: WavePolynomes
slicedFrames = waveCalc(slicedFrames, calTree=calTree)

# Correct the wavelength for the spacecraft velocity.
# Uses the pointing, orbitEphemeris and timeCorrelation product.
slicedFrames = specCorrectHerschelVelocity(slicedFrames, obs.auxiliary.orbitEphemeris, obs.aux

# Find the major logical blocks of this observation and organise them in the
# BlockTable attached to the Frames; used cal file: ObcpDescription
slicedFrames = findBlocks(slicedFrames, calTree = calTree)

# Flag the known bad or noisy pixels in the masks "BADPIXELS" and "NOISYPIXELS"
# used cal files: BadPixelMask and NoisyPixelMask
#   -> by default the bad pixels will be excluded later when final cubes are built, the noisy
slicedFrames = specFlagBadPixelsFrames(slicedFrames, calTree=calTree)

if verbose:
    # Summary of the slices
    slicedSummary(slicedFrames)
    # Summary of the active (1) and inactive (0) status of every Mask
    maskSummary(slicedFrames)
    # Plot the instrument movements, without the signal
    p1 = slicedSummaryPlot(slicedFrames,signal=0)
```

Check: after slicedSummaryPlot(slicedFrames…)



Cal Block

Nod B

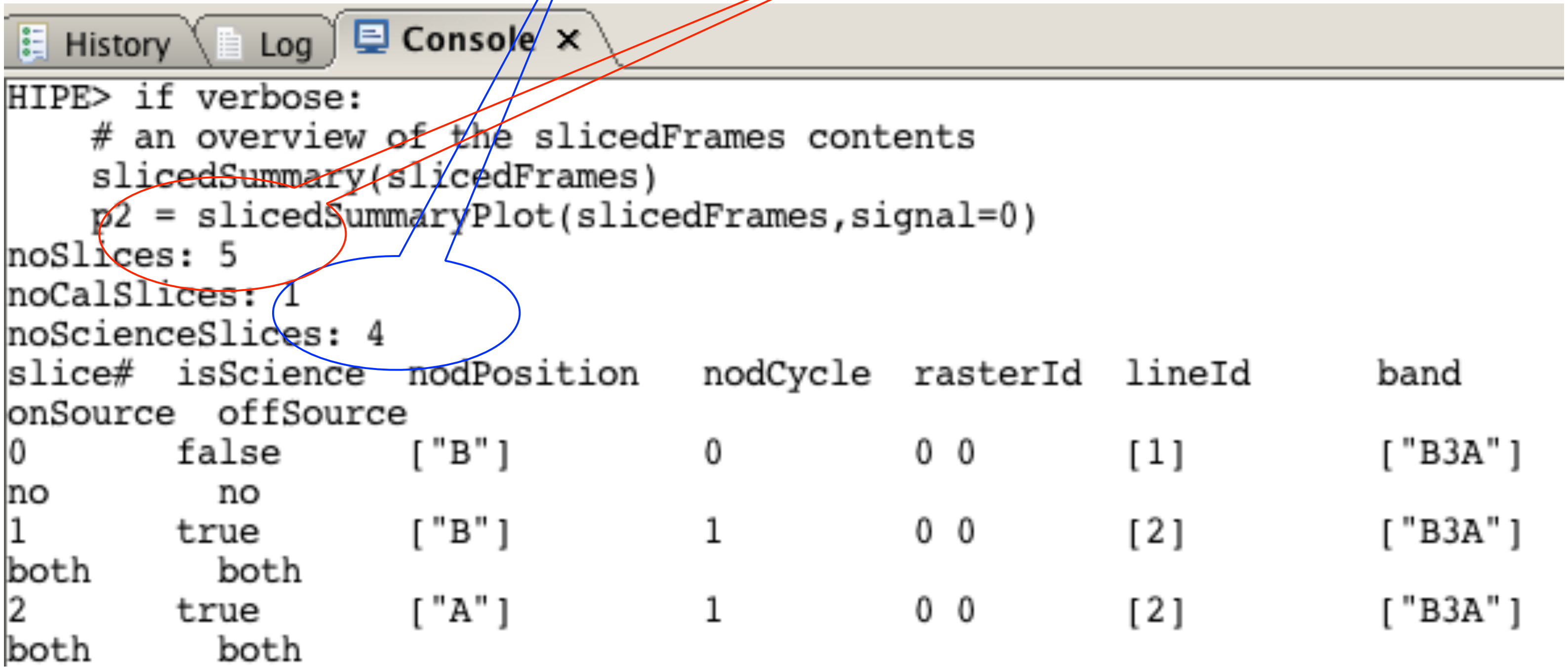
Nod A

There are two lines (two wavelengths in red). Grating scans are numbered positive if upscans and negative if downscans.

nhsc.ipac.caltech.edu/helpdesk

Check: slcedSummary(slicedFrames)

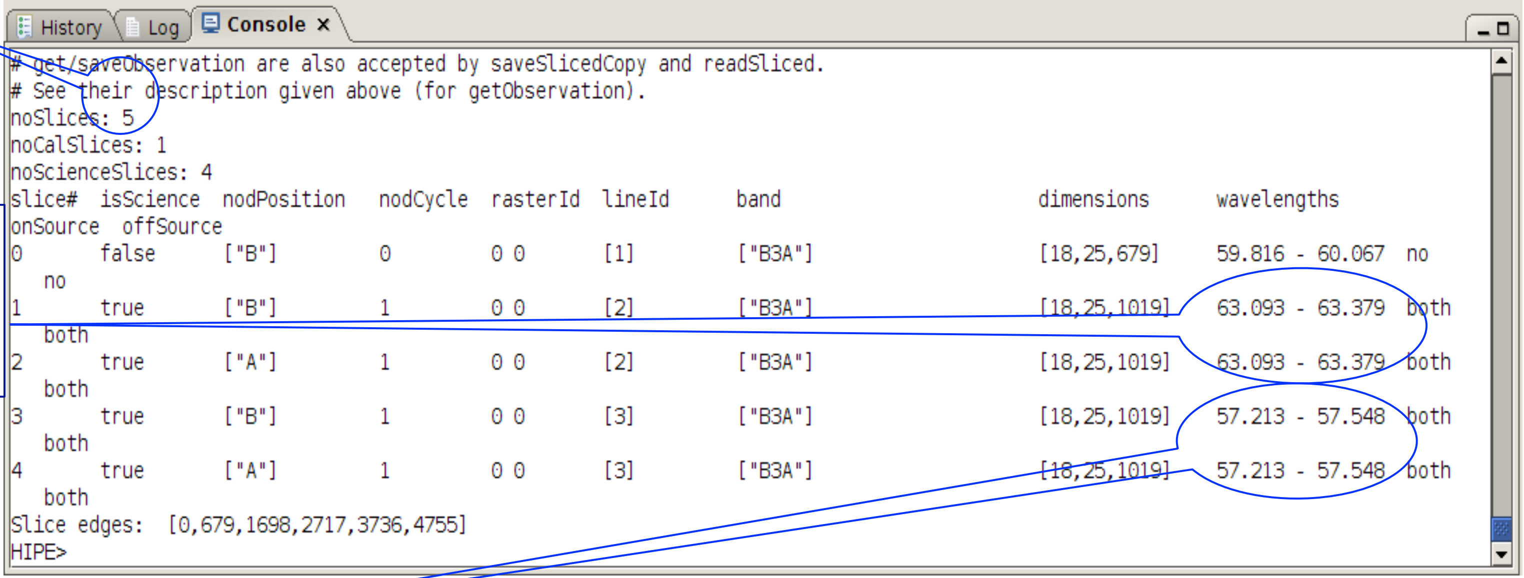
The slicing of the data is performed according to rules made explicit in the pipeline. In our example, two lines are observed in two nodding positions. So, we expect 4 slices plus an initial slice containing the calibration block.



```
HIPE> if verbose:
    # an overview of the slicedFrames contents
    slicedSummary(slicedFrames)
    p2 = slicedSummaryPlot(slicedFrames,signal=0)
noSlices: 5
noCalSlices: 1
noScienceSlices: 4
slice#  isScience  nodPosition   nodCycle  rasterId  lineId      band
onSource  offSource
0        false      ["B"]        0         0 0       [1]         ["B3A"]
no        no
1        true       ["B"]        1         0 0       [2]         ["B3A"]
both      both
2        true       ["A"]        1         0 0       [2]         ["B3A"]
both      both
```

Check: after slicing

5 slices !



```
# get/saveObservation are also accepted by saveSlicedCopy and readSliced.
# See their description given above (for getObservation).
noSlices: 5
noCalSlices: 1
noScienceSlices: 4
slice#  isScience  nodPosition   nodCycle  rasterId  lineId     band                      dimensions     wavelengths
onSource  offSource
0       false      ["B"]         0         0 0       [1]        ["B3A"]                   [18,25,679]    59.816 - 60.067  no
   no
1       true       ["B"]         1         0 0       [2]        ["B3A"]                   [18,25,1019]   63.093 - 63.379  both
   both
2       true       ["A"]         1         0 0       [2]        ["B3A"]                   [18,25,1019]   63.093 - 63.379  both
   both
3       true       ["B"]         1         0 0       [3]        ["B3A"]                   [18,25,1019]   57.213 - 57.548  both
   both
4       true       ["A"]         1         0 0       [3]        ["B3A"]                   [18,25,1019]   57.213 - 57.548  both
   both
Slice edges:  [0,679,1698,2717,3736,4755]
HIPE>
```

Line 1 – B & A nodes

Line 2 – B & A nodes

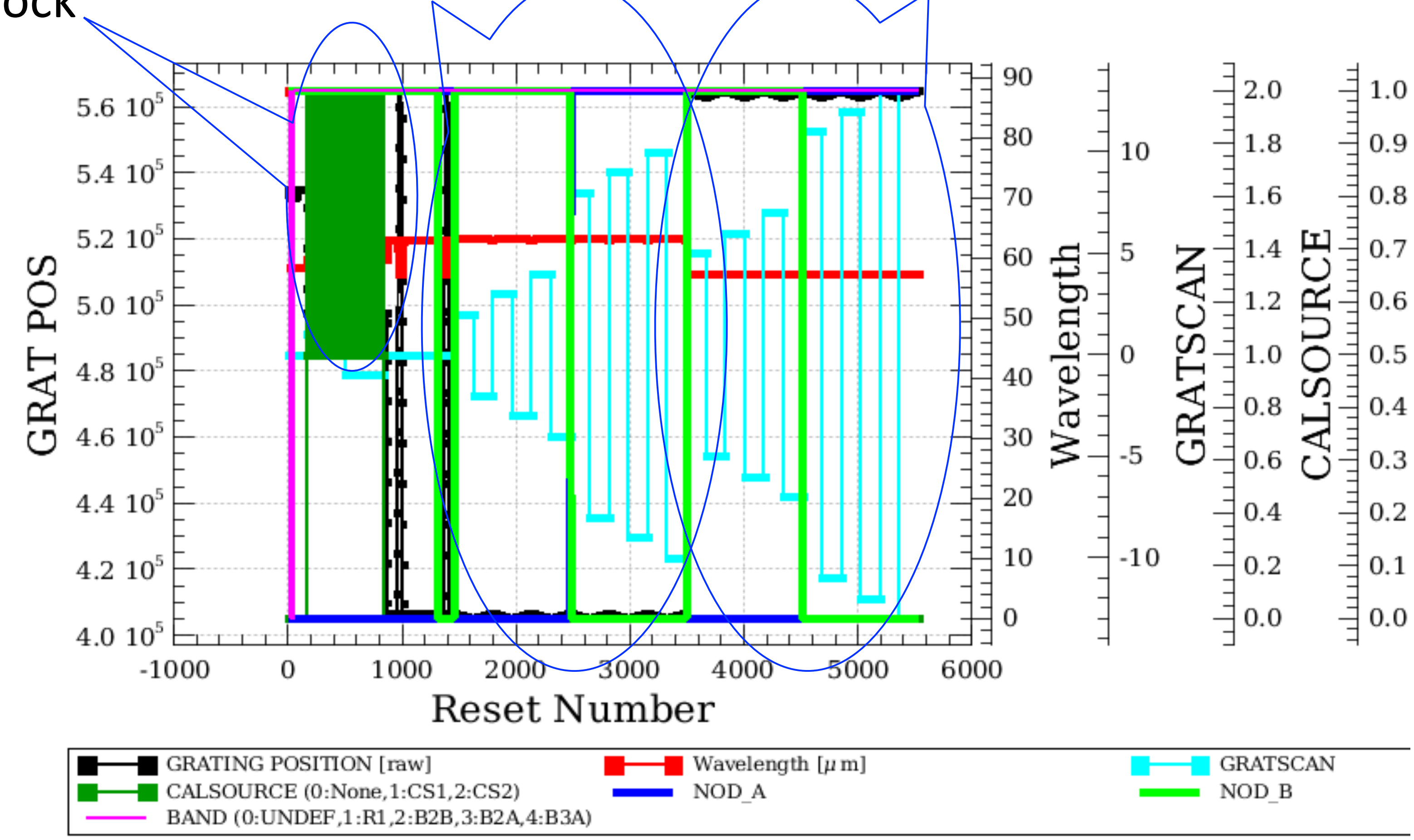
nhsc.ipac.caltech.edu/helpdesk

Check: after slicing

Cal Block

Line 1: OI 63

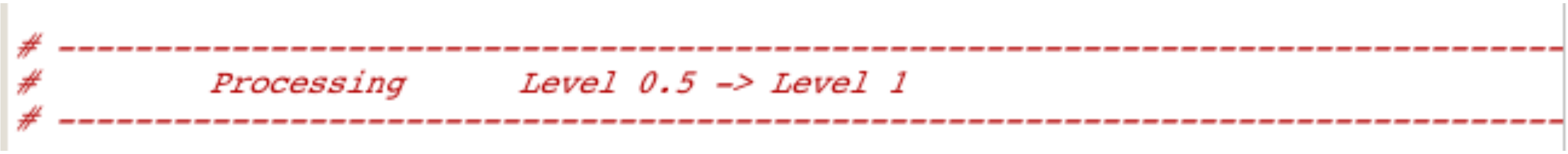
Line 2: NIII 57



There are four slices (calibration, nod A and B for the 1st line, nod A and B for the 2nd line).

Continue …

With remaining Level 0 to 0.5 processing steps as outlined in slide 17. Step through with the green arrow.



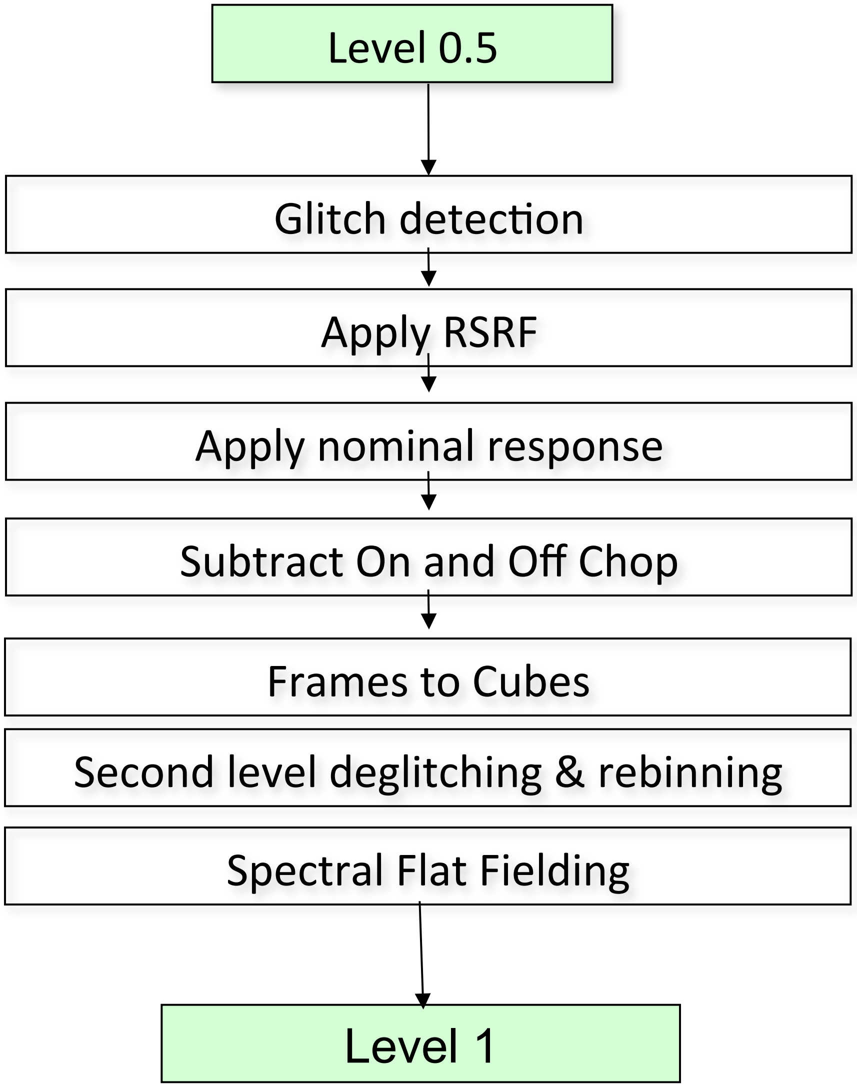
```
# -----------------------------------------------------------------------
#        Processing      Level 0.5 -> Level 1
# -----------------------------------------------------------------------
```

Step 4

Run the 0.5 → 1 pipeline

Glitch detection, chop differentiation, RSRF, flat

Level 0.5 → 1



Level 0.5

Glitch detection

Apply RSRF

Apply nominal response

Subtract On and Off Chop

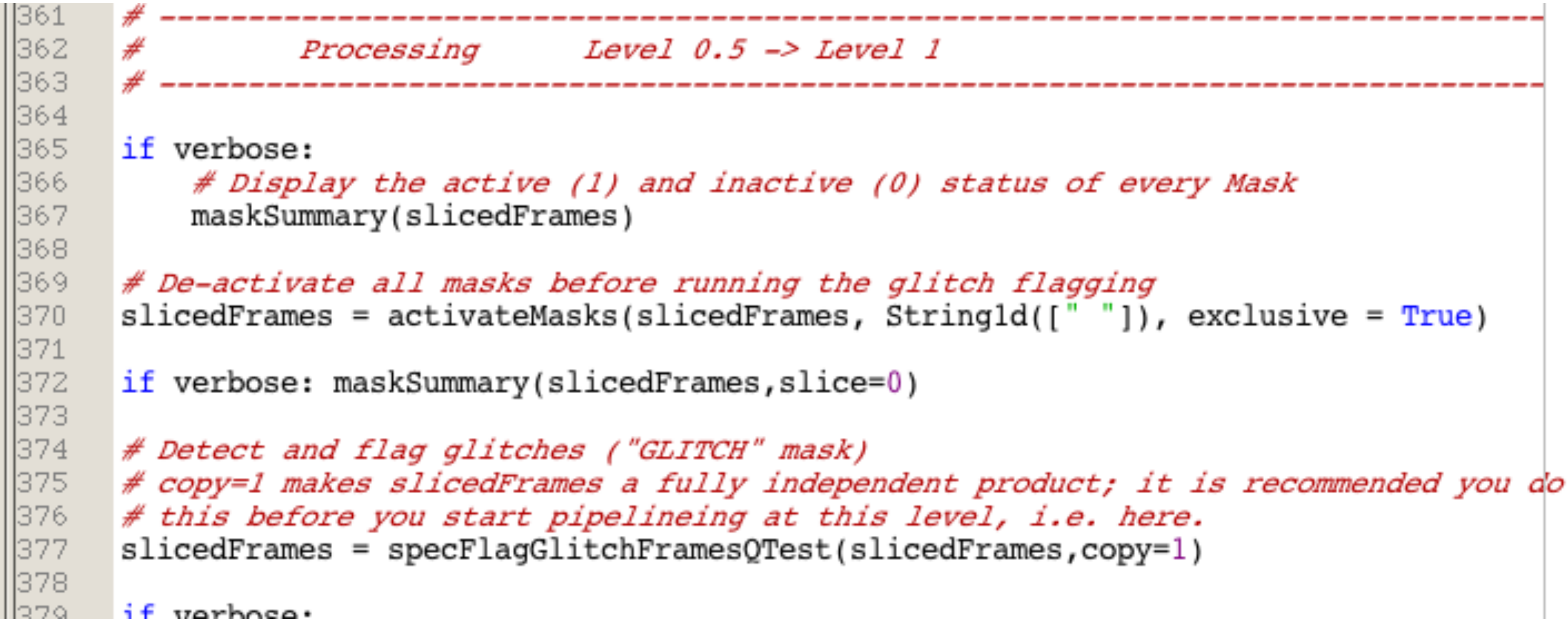
Frames to Cubes

Second level deglitching & rebinning

Spectral Flat Fielding

Level 1

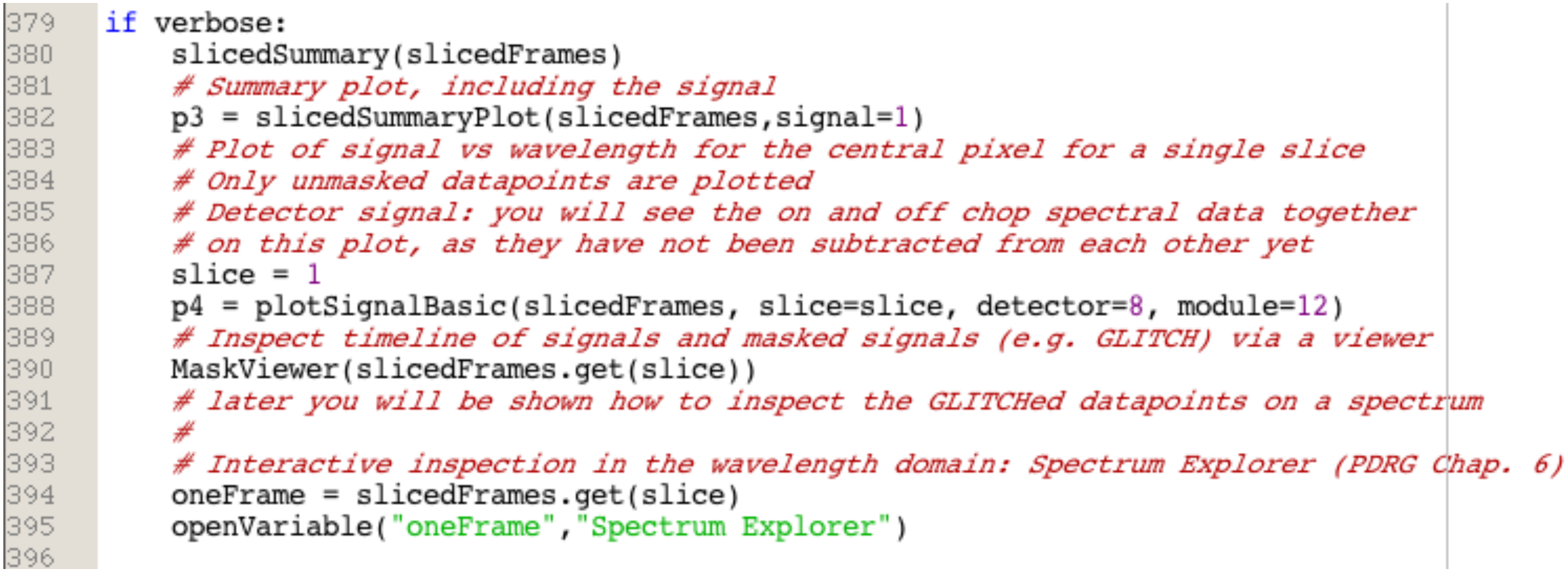
Step through glitch detection



```
# ------------------------------------------------------------------------------
#         Processing      Level 0.5 -> Level 1
# ------------------------------------------------------------------------------

if verbose:
    # Display the active (1) and inactive (0) status of every Mask
    maskSummary(slicedFrames)

# De-activate all masks before running the glitch flagging
slicedFrames = activateMasks(slicedFrames, String1d([" "]), exclusive = True)

if verbose: maskSummary(slicedFrames,slice=0)

# Detect and flag glitches ("GLITCH" mask)
# copy=1 makes slicedFrames a fully independent product; it is recommended you do
# this before you start pipelineing at this level, i.e. here.
slicedFrames = specFlagGlitchFramesQTest(slicedFrames,copy=1)

if verbose:
```

Diagnostics

With verbose=1 (earlier in the script) several diagnostic plots and print out (watch your console) will appear after these lines …



```
if verbose:
    slicedSummary(slicedFrames)
    # Summary plot, including the signal
    p3 = slicedSummaryPlot(slicedFrames,signal=1)
    # Plot of signal vs wavelength for the central pixel for a single slice
    # Only unmasked datapoints are plotted
    # Detector signal: you will see the on and off chop spectral data together
    # on this plot, as they have not been subtracted from each other yet
    slice = 1
    p4 = plotSignalBasic(slicedFrames, slice=slice, detector=8, module=12)
    # Inspect timeline of signals and masked signals (e.g. GLITCH) via a viewer
    MaskViewer(slicedFrames.get(slice))
    # later you will be shown how to inspect the GLITCHed datapoints on a spectrum
    #
    # Interactive inspection in the wavelength domain: Spectrum Explorer (PDRG Chap. 6)
    oneFrame = slicedFrames.get(slice)
    openVariable("oneFrame","Spectrum Explorer")
```

Check: Glitch detection

You can check manually the points flagged as glitches or masked for other reasons using the *maskviewer*

Select a pixel by clicking on it

Select a mask

Select a frame

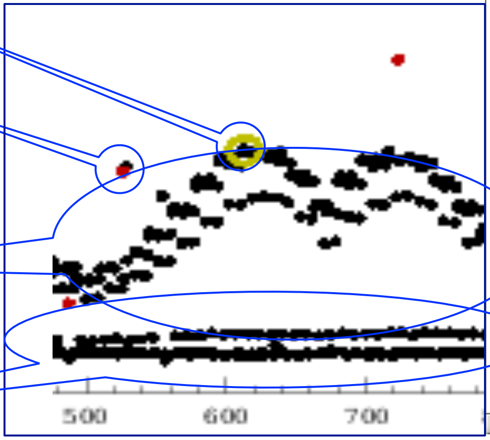
Do first

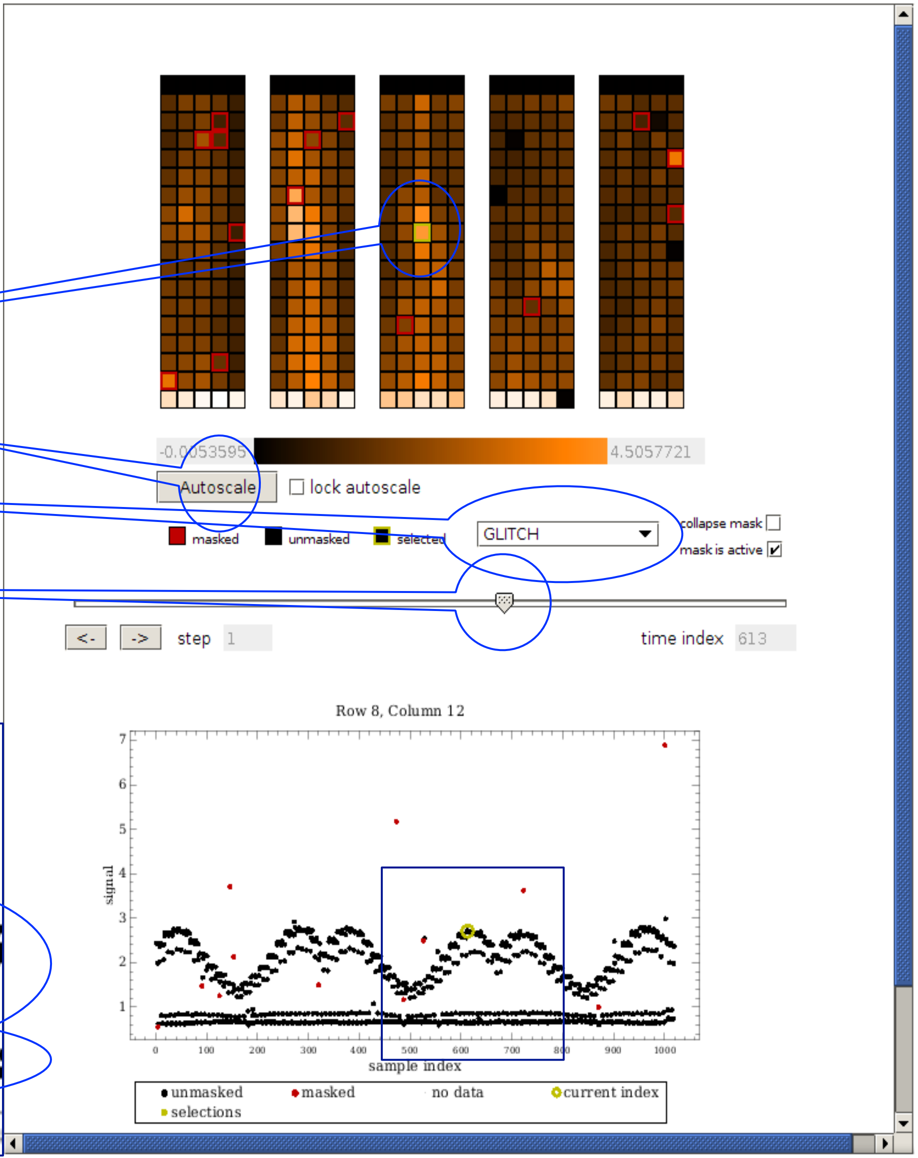
Current frame

Masked glitch

ON signal

OFF signal

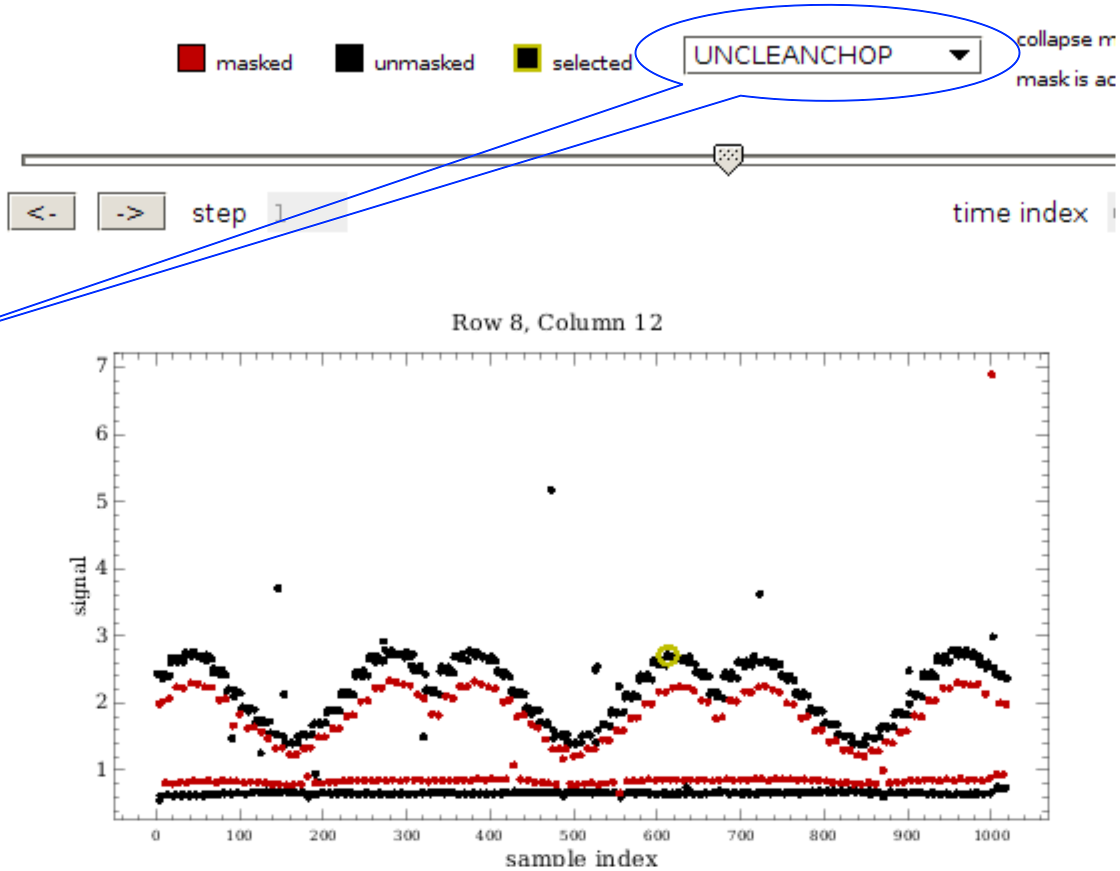




More masks

It is possible to explore other masks

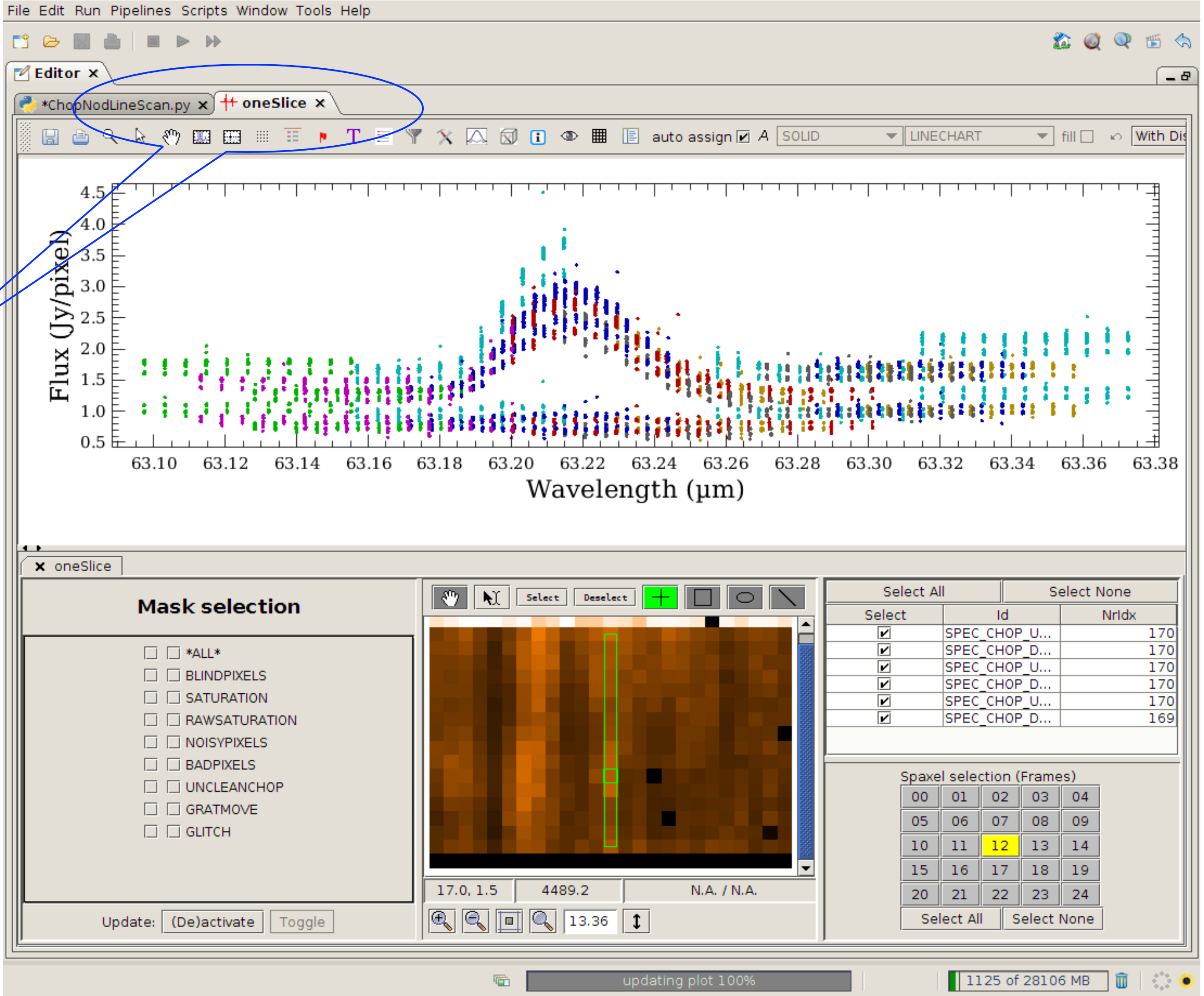
Select unclean chop



In this case, it is clear why there is a second group of points for the ON and OFF positions. These corresponds to signals obtained when the chopper was not yet in the correct position.

Look for this new tab in your editor window.

Adjust screen to match this view.



Mask selection

- ☐ ☐ *ALL*
- ☐ ☐ BLINDPIXELS
- ☐ ☐ SATURATION
- ☐ ☐ RAWSATURATION
- ☐ ☐ NOISYPIXELS
- ☐ ☐ BADPIXELS
- ☐ ☐ UNCLEANCHOP
- ☐ ☐ GRATMOVE
- ☐ ☐ GLITCH

Update: (De)activate Toggle

Select All | Select None

Select	Id	NrIdx
☑	SPEC_CHOP_U...	170
☑	SPEC_CHOP_D...	170
☑	SPEC_CHOP_U...	170
☑	SPEC_CHOP_D...	170
☑	SPEC_CHOP_U...	170
☑	SPEC_CHOP_D...	169

Spaxel selection (Frames)

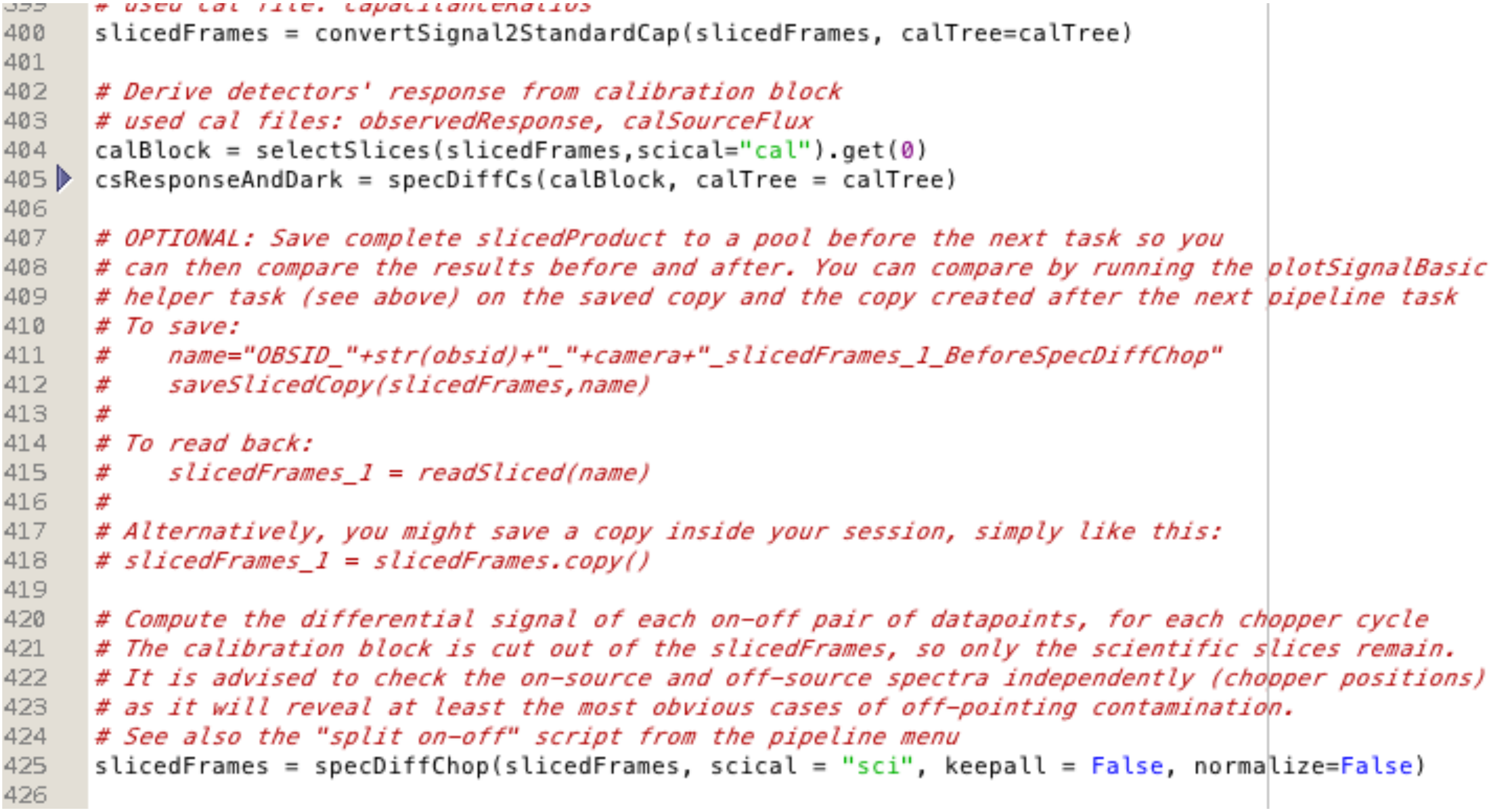
00	01	02	03	04
05	06	07	08	09
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Select All | Select None

A further inspection of your data is now possible using the Spectrum Explorer. Several options are available such as selection of pixels and different masks for the first slice.

Continue …

With remaining Level 0.5 to 1.0 processing steps as outlined in slide 28.
Step through with the green arrow.

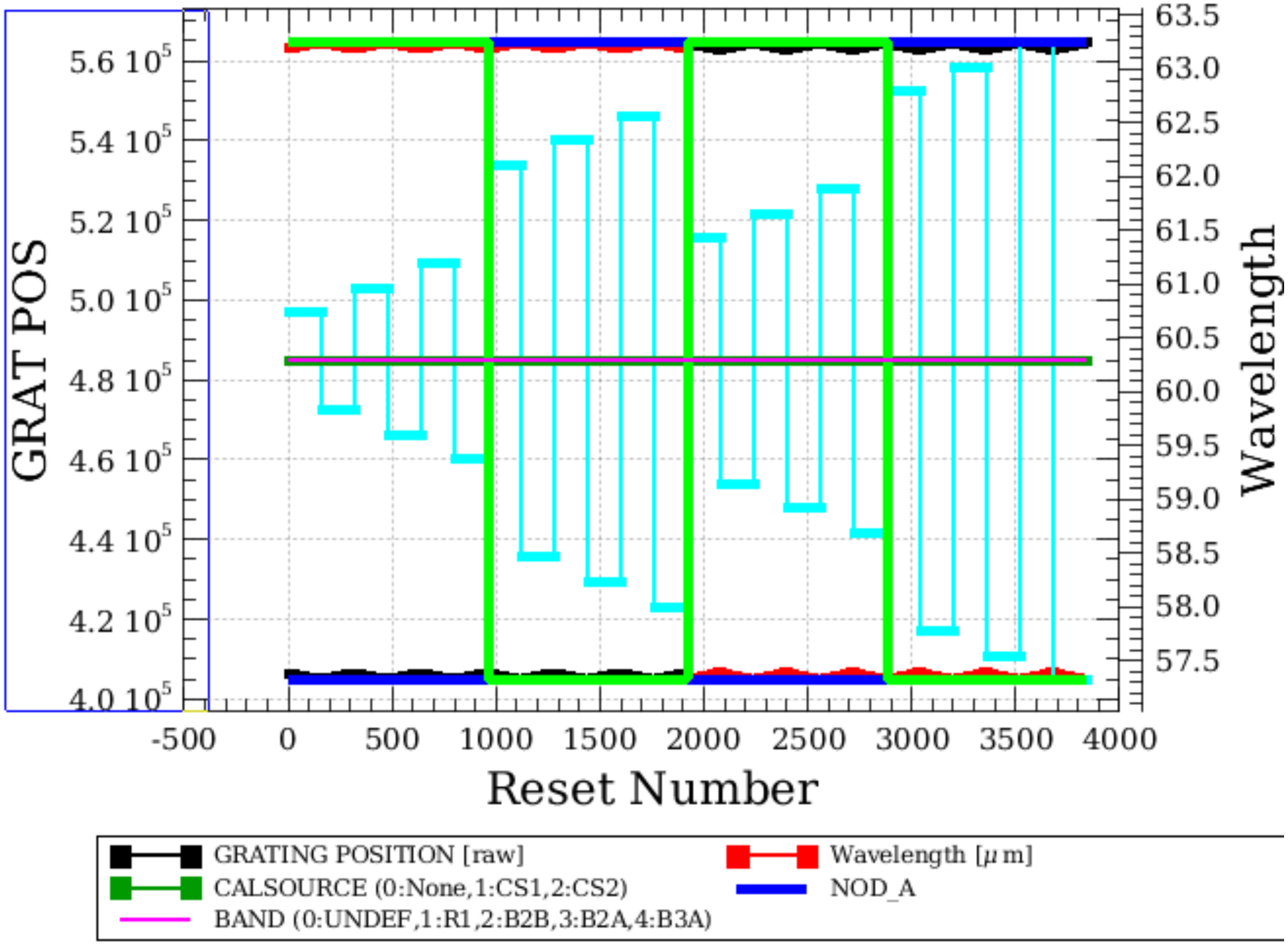


```
# used cal file: capacitanceRatios
slicedFrames = convertSignal2StandardCap(slicedFrames, calTree=calTree)

# Derive detectors' response from calibration block
# used cal files: observedResponse, calSourceFlux
calBlock = selectSlices(slicedFrames,scical="cal").get(0)
csResponseAndDark = specDiffCs(calBlock, calTree = calTree)

# OPTIONAL: Save complete slicedProduct to a pool before the next task so you
# can then compare the results before and after. You can compare by running the plotSignalBasic
# helper task (see above) on the saved copy and the copy created after the next pipeline task
# To save:
#    name="OBSID_"+str(obsid)+"_"+camera+"_slicedFrames_1_BeforeSpecDiffChop"
#    saveSlicedCopy(slicedFrames,name)
#
# To read back:
#    slicedFrames_1 = readSliced(name)
#
# Alternatively, you might save a copy inside your session, simply like this:
# slicedFrames_1 = slicedFrames.copy()

# Compute the differential signal of each on-off pair of datapoints, for each chopper cycle
# The calibration block is cut out of the slicedFrames, so only the scientific slices remain.
# It is advised to check the on-source and off-source spectra independently (chopper positions)
# as it will reveal at least the most obvious cases of off-pointing contamination.
# See also the "split on-off" script from the pipeline menu
slicedFrames = specDiffChop(slicedFrames, scical = "sci", keepall = False, normalize=False)

```

Chop differentiation

Verbose=1, shows

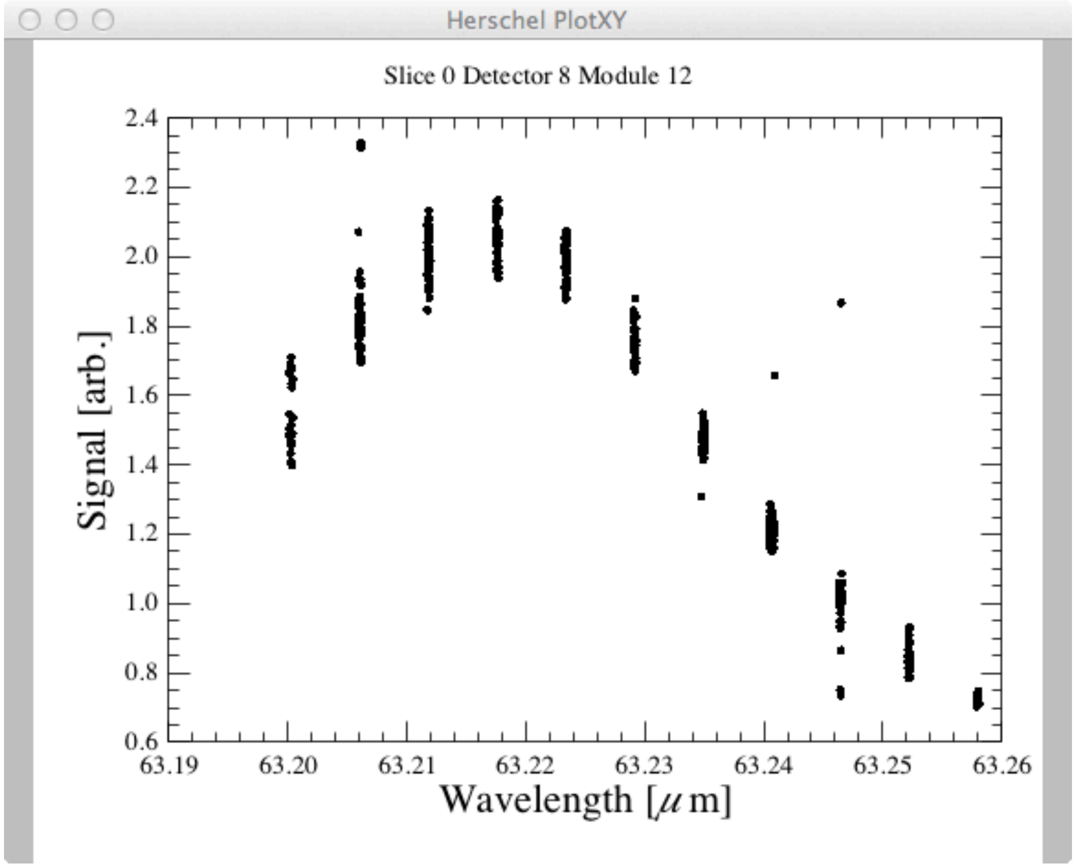
After chop differentiation, the calibration block is excluded from the data



nhsc.ipac.caltech.edu/helpdesk

Chop differentiation

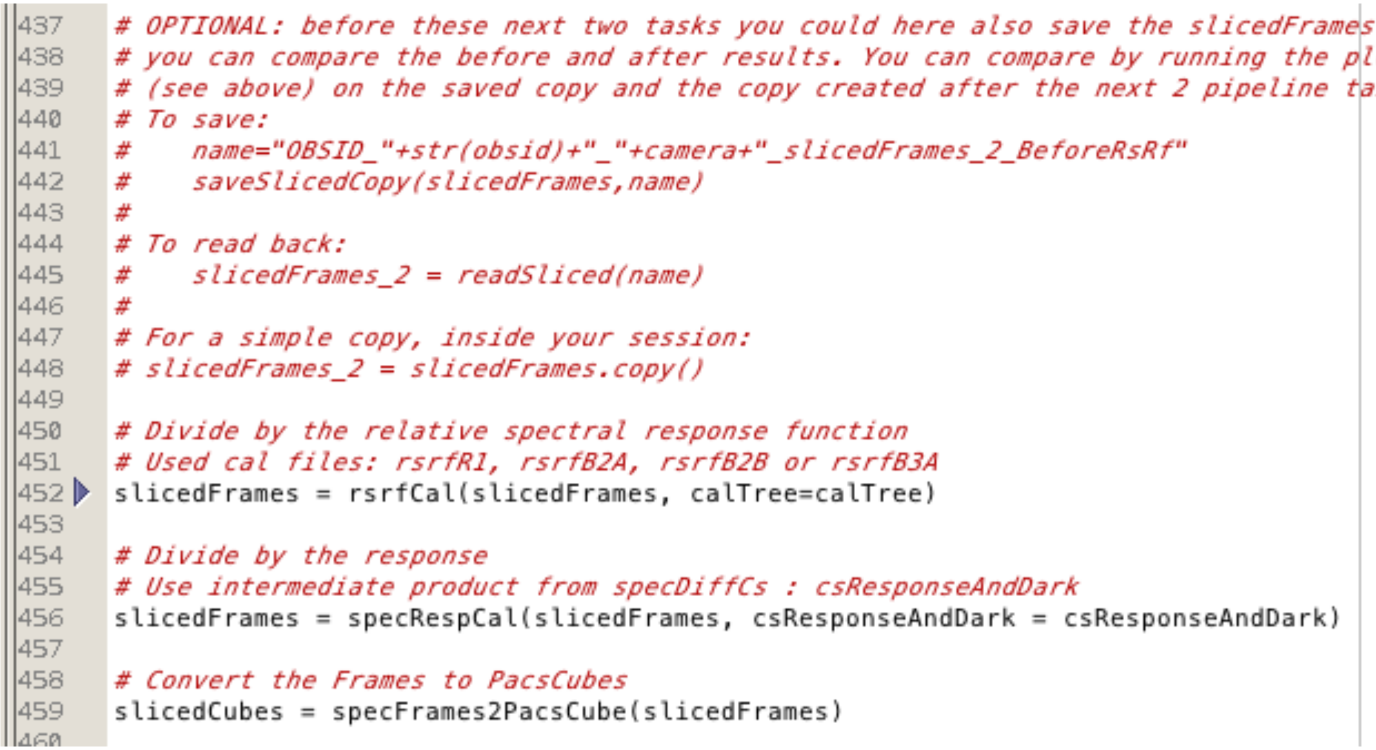
Verbose=1 shows
The data are only on the ON position (OFF being subtracted)



nhsc.ipac.caltech.edu/helpdesk

Continue …

With remaining Level 0.5 to 1.0 processing steps as outlined in slide 28.
Step through with the green arrow.

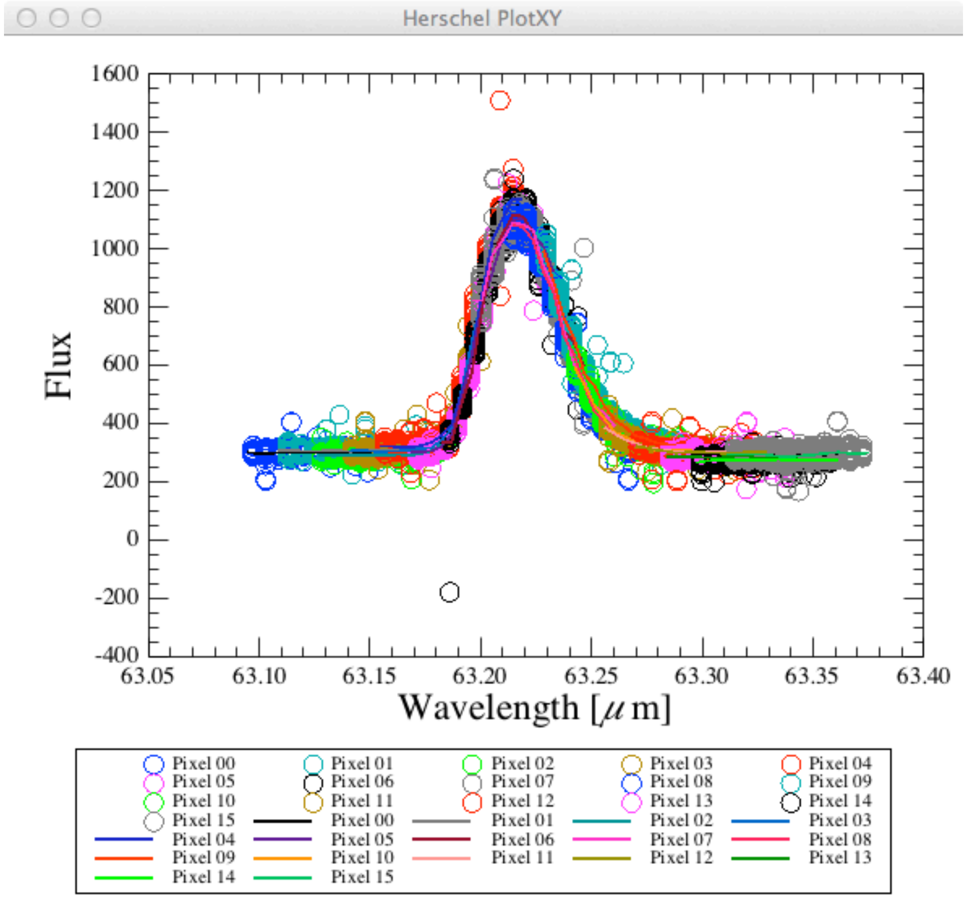


```
# OPTIONAL: before these next two tasks you could here also save the slicedFrames to pool, so
# you can compare the before and after results. You can compare by running the plotSignalBasic helper
# (see above) on the saved copy and the copy created after the next 2 pipeline tasks
# To save:
#    name="OBSID_"+str(obsid)+"_"+camera+"_slicedFrames_2_BeforeRsRf"
#    saveSlicedCopy(slicedFrames,name)
#
# To read back:
#    slicedFrames_2 = readSliced(name)
#
# For a simple copy, inside your session:
# slicedFrames_2 = slicedFrames.copy()

# Divide by the relative spectral response function
# Used cal files: rsrfR1, rsrfB2A, rsrfB2B or rsrfB3A
slicedFrames = rsrfCal(slicedFrames, calTree=calTree)

# Divide by the response
# Use intermediate product from specDiffCs : csResponseAndDark
slicedFrames = specRespCal(slicedFrames, csResponseAndDark = csResponseAndDark)

# Convert the Frames to PacsCubes
slicedCubes = specFrames2PacsCube(slicedFrames)
```

Check: RSRF and response

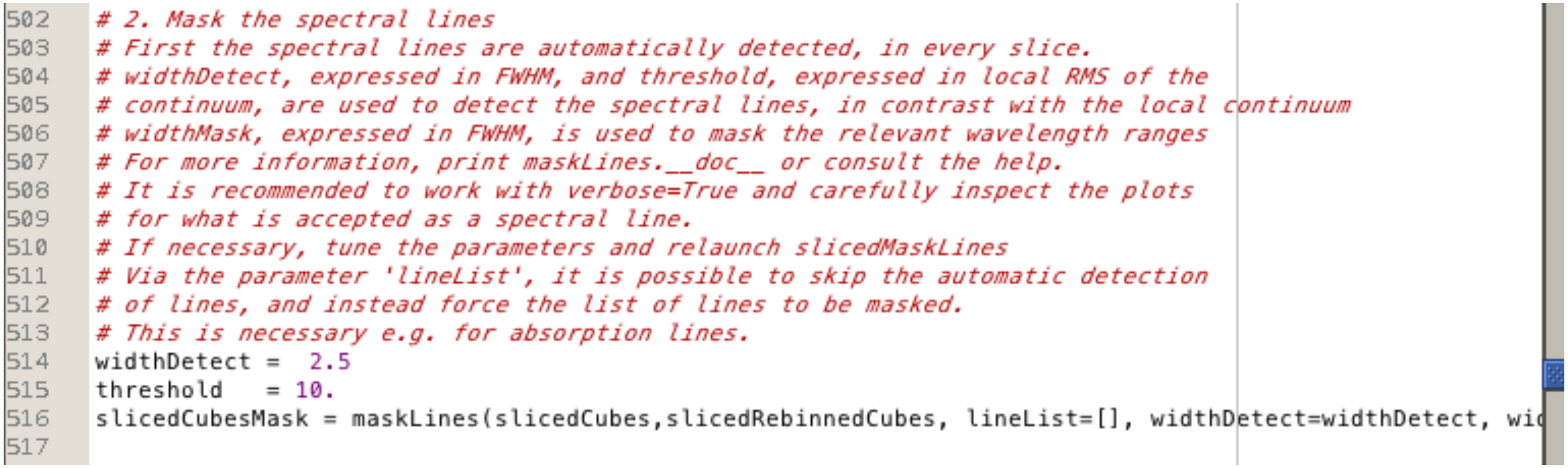
After   pb4ff = plotPixels(slicedCubes.get(........

After applying RSRF and response corrections we have a first look at the spectrum



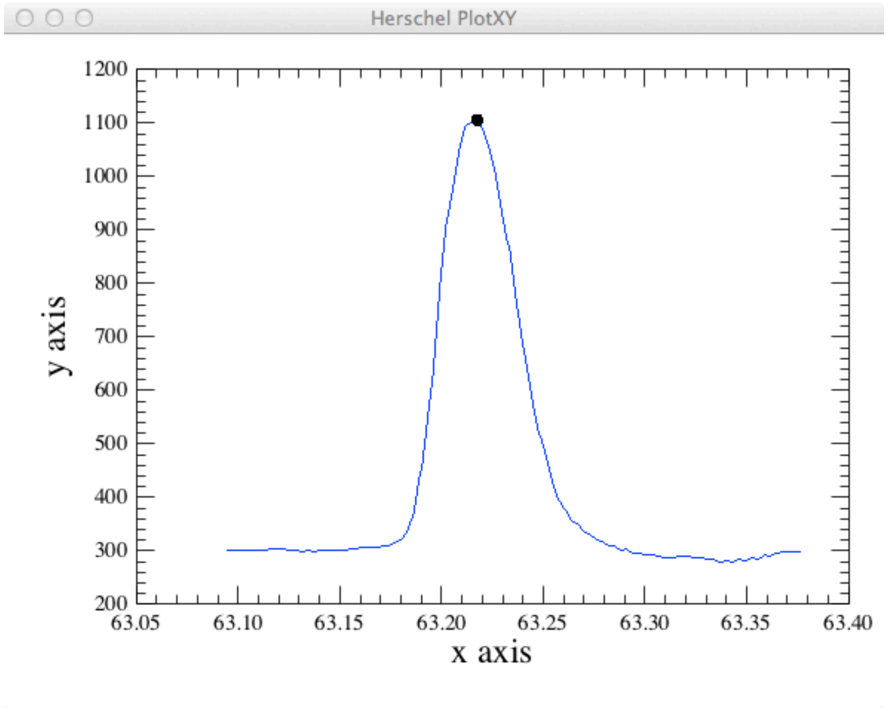
Continue …

With remaining Level 0.5 to 1.0 processing steps as outlined in slide 28.
Step through with the green arrow.



```
# 2. Mask the spectral lines
# First the spectral lines are automatically detected, in every slice.
# widthDetect, expressed in FWHM, and threshold, expressed in local RMS of the
# continuum, are used to detect the spectral lines, in contrast with the local continuum
# widthMask, expressed in FWHM, is used to mask the relevant wavelength ranges
# For more information, print maskLines.__doc__ or consult the help.
# It is recommended to work with verbose=True and carefully inspect the plots
# for what is accepted as a spectral line.
# If necessary, tune the parameters and relaunch slicedMaskLines
# Via the parameter 'lineList', it is possible to skip the automatic detection
# of lines, and instead force the list of lines to be masked.
# This is necessary e.g. for absorption lines.
widthDetect =  2.5
threshold   = 10.
slicedCubesMask = maskLines(slicedCubes,slicedRebinnedCubes, lineList=[], widthDetect=widthDetect, wid
```

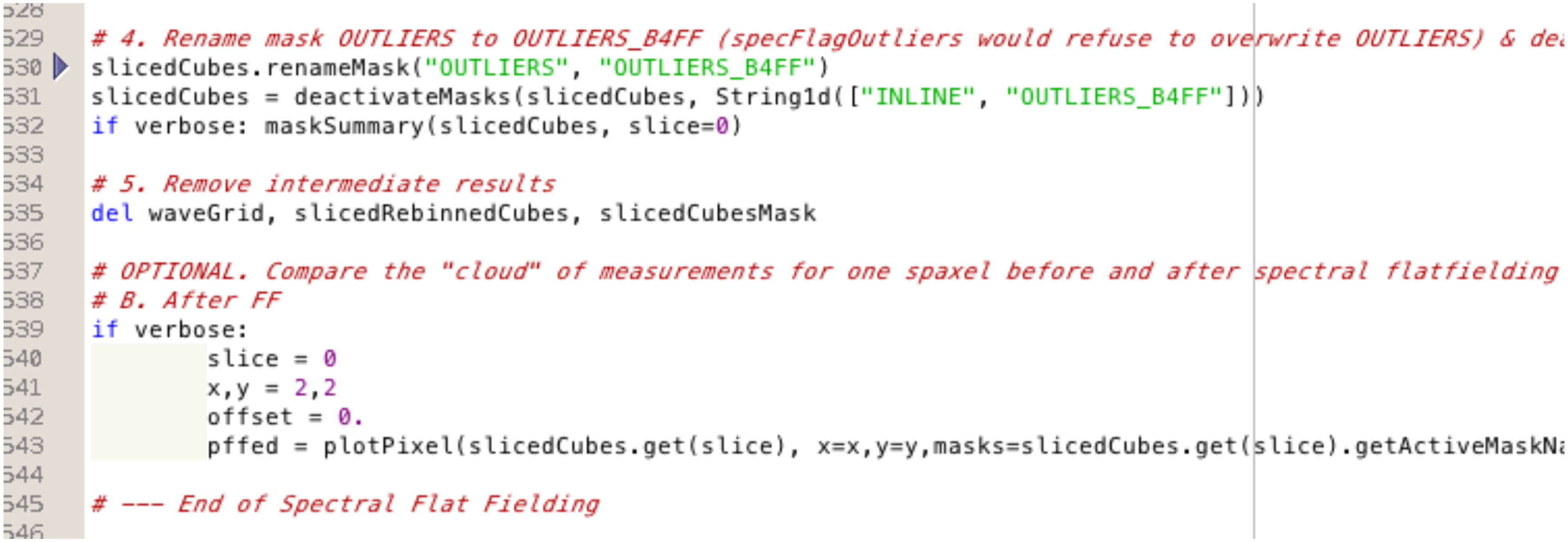
Check: Spectral FlatField



As a default, the code will search for lines in all the pixels and then mask them before computing the spectral flat field.
It is possible to give directly the list of lines to be masked via the parameter lineList = [63.227], for instance.

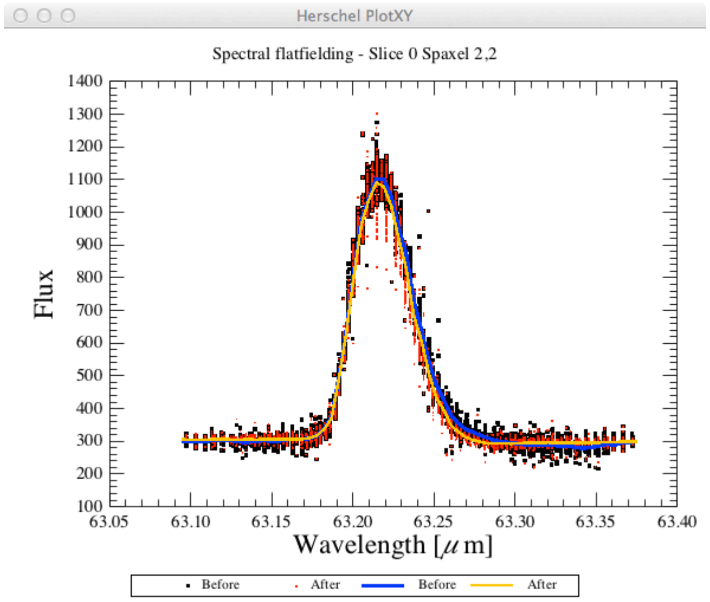
Continue …

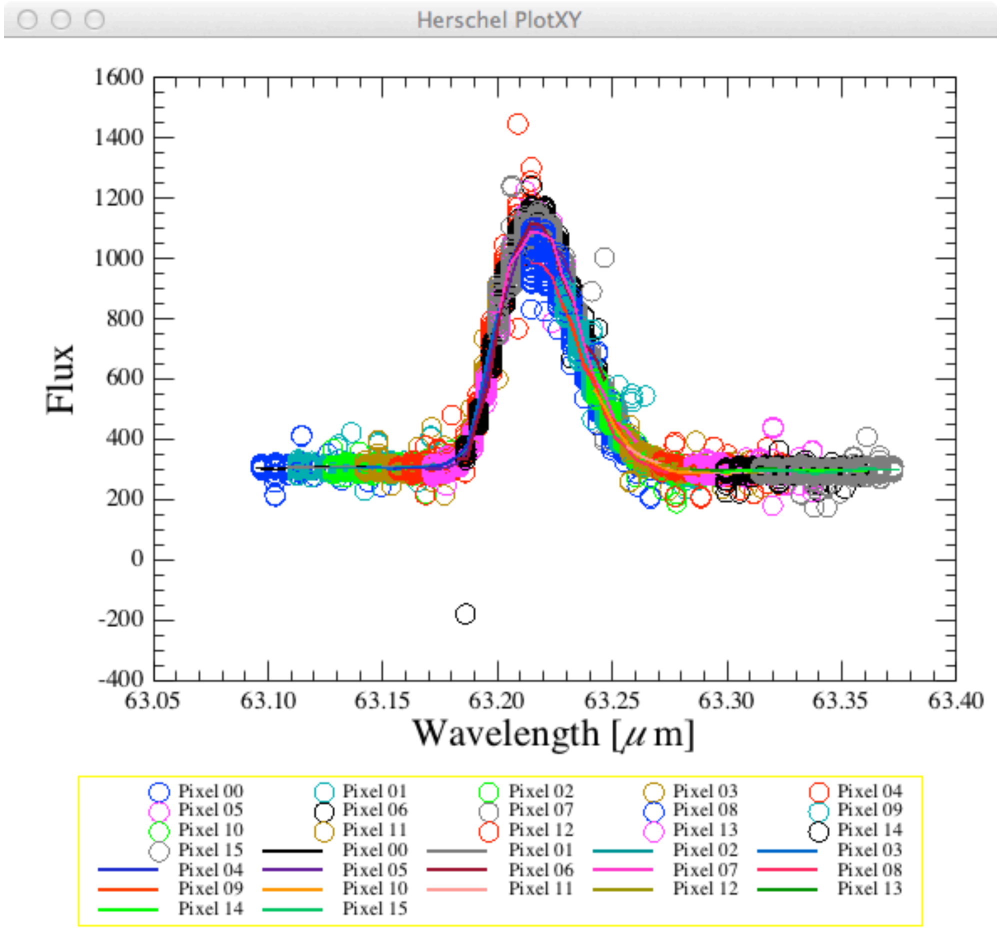
With remaining Level 0.5 to 1.0 processing steps as outlined in slide 28.
Step through with the green arrow.



```
# 4. Rename mask OUTLIERS to OUTLIERS_B4FF (specFlagOutliers would refuse to overwrite OUTLIERS) & de
slicedCubes.renameMask("OUTLIERS", "OUTLIERS_B4FF")
slicedCubes = deactivateMasks(slicedCubes, String1d(["INLINE", "OUTLIERS_B4FF"]))
if verbose: maskSummary(slicedCubes, slice=0)

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

# OPTIONAL. Compare the "cloud" of measurements for one spaxel before and after spectral flatfielding
# B. After FF
if verbose:
        slice = 0
        x,y = 2,2
        offset = 0.
        pfed = plotPixel(slicedCubes.get(slice), x=x,y=y,masks=slicedCubes.get(slice).getActiveMaskNa

# --- End of Spectral Flat Fielding
```

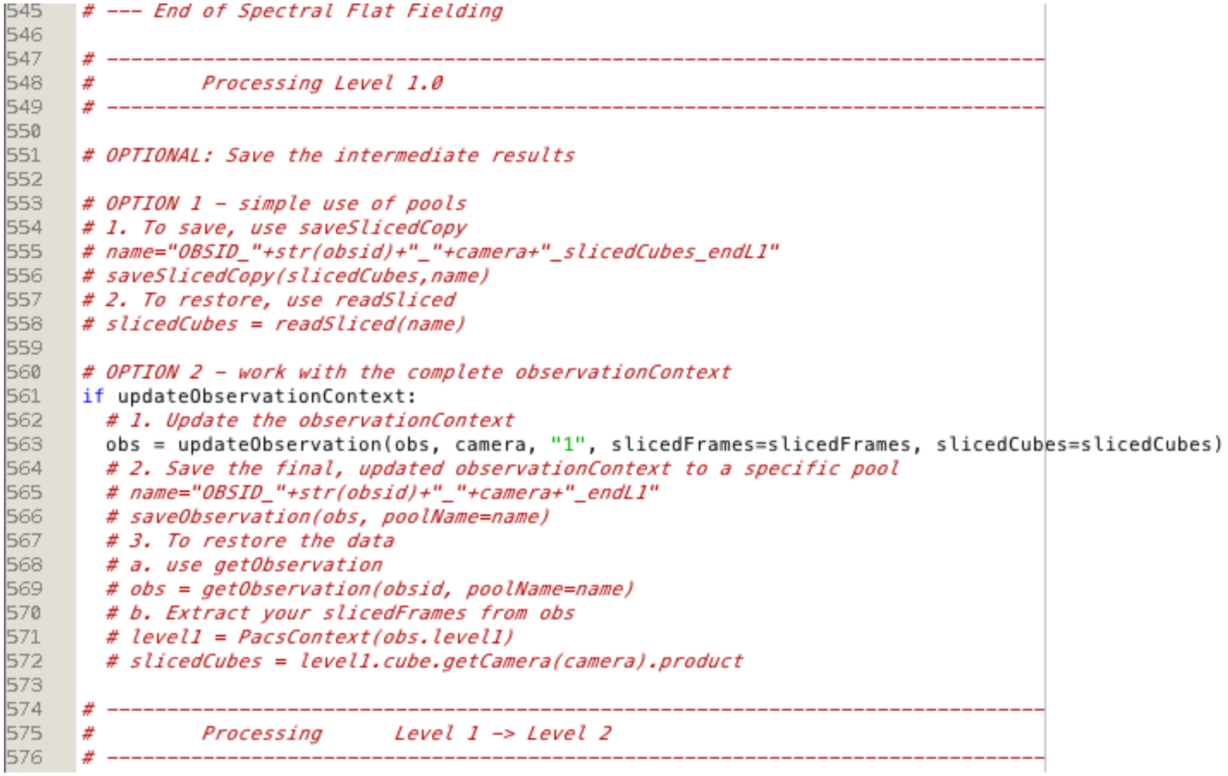
Check: Spectral FlatField





At this point, the frames are converted in calibrated cubes and we have reached level 1 !

You are ready to continue with PACS-302



```
# --- End of Spectral Flat Fielding

# ------------------------------------------------------------------------------
#         Processing Level 1.0
# ------------------------------------------------------------------------------

# OPTIONAL: Save the intermediate results

# OPTION 1 - simple use of pools
# 1. To save, use saveSlicedCopy
# name="OBSID_"+str(obsid)+"_"+camera+"_slicedCubes_endL1"
# saveSlicedCopy(slicedCubes,name)
# 2. To restore, use readSliced
# slicedCubes = readSliced(name)

# OPTION 2 - work with the complete observationContext
if updateObservationContext:
  # 1. Update the observationContext
  obs = updateObservation(obs, camera, "1", slicedFrames=slicedFrames, slicedCubes=slicedCubes)
  # 2. Save the final, updated observationContext to a specific pool
  # name="OBSID_"+str(obsid)+"_"+camera+"_endL1"
  # saveObservation(obs, poolName=name)
  # 3. To restore the data
  # a. use getObservation
  # obs = getObservation(obsid, poolName=name)
  # b. Extract your slicedFrames from obs
  # level1 = PacsContext(obs.level1)
  # slicedCubes = level1.cube.getCamera(camera).product

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