Data Analysis Guide

Version 11.1, Document Number: HERSCHEL-HSC-DOC-1199
07 December 2016

Data Analysis Guide

Table of Contents

Preface xxvi
1. Conventions used in this manual xxvi
1. Data input/output 1
1.1. Components of an observation 1
1.2. Typical workflow 3
1.3. How data are stored on your disk 4
1.3.1. Managing storages and pools 5
1.4. Getting observations from the Herschel Science Archive 7
1.4.1. Logging into the HSA 8
1.4.2. Finding observations in the HSA 10
1.4.3. Inspecting the query results of an observation 11
1.4.4. Finding observation IDs outside the HUI 12
1.4.5. Downloading one entire observation 13
1.4.6. Browsing an observation in the HSA with known OBSID 14
1.4.7. Downloading multiple observations 17
1.5. Loading observations downloaded from the HSA into HIPE 19
1.6. Managing your HSA downloads 21
1.6.1. Advanced configuration 23
1.7. Retrieving an observation from disk 24
1.8. Customising the Product Browser results 27
1.9. How to use the Quick Analysis perspective 28
1.10. Saving data (products and observations) to disk 30
1.11. Migrating pools across *incompatible versions* of HIPE 32
1.12. Exporting an observation to a colleague 34
1.13. Retrieving products from disk 35
1.14. Removing data from disk 35
1.15. On-demand reprocessing of observations 35
1.16. Exchanging data with FITS files 36
1.16.1. Saving a product to a FITS file 36
1.16.2. Retrieving a Herschel product from a FITS file 38
1.16.3. Translation of Herschel metadata to FITS keywords 39
1.16.4. Structure of Herschel products when saved as FITS 41
1.16.5. Troubleshooting FITS import/export 46
1.16.6. Importing a non-Herschel FITS file into HIPE 46
1.16.7. Importing a Herschel FITS file into external applications 48
1.17. Working with the VO (External Tools) 52
1.17.1. Sending products from HIPE to external tools 52
1.17.2. Sending products from external tools to HIPE 54
1.17.3. Opening VO Tables from HIPE 54
1.17.4. Writing tables to files in VO-table XML format 55
2. Saving data as text files 56
2.1. Considerations and concepts for working with text files 56
2.2. Worked example: Saving a Spectrum product as a text file 58
2.3. Worked example: Saving a SourceListProduct as a text file 60
2.4. Worked example: Reading a Spitzer spectrum into a table dataset 62
2.5. Worked example: Reading a VizieR catalogue into a table dataset 64
2.6. Reading a comma-separated-value (CSV) file into a table dataset 67
2.7. Reading a space-separated file into a table dataset 68
2.8. Reading an IPAC, SExtractor or Topcat file into a table dataset 70
2.9. Reading a generic ASCII table file into a table dataset 71
2.10. Writing a table dataset to a comma-separated-values (CSV) file 71
2.11. Writing a table dataset into a space-separated-value file 73
2.12. Writing a spectrum to an ASCII table file 74
2.13. Writing a table dataset to a generic ASCII table file 79
2.14. Reading column names from a file 80

2.15. Defining which lines to ignore when reading a file ... 80
2.16. Specifying the data types when reading a file ... 82
2.17. Specifying how data values are separated when reading a file ... 83
2.18. Saving and loading a configuration for reading from file ... 84
2.19. Adding a header to an ASCII table file ... 85
2.20. Adding table dataset metadata to an ASCII table file ... 85
2.21. Defining a custom prefix for commented lines ... 86
2.22. Choosing how to separate data values ... 86
2.23. Saving and loading options for writing to file ... 86
2.24. Parsers, formatters and templates ... 87
2.25. Creating and configuring table templates ... 88
2.26. Creating and configuring parsers for reading in data ... 89
2.27. Creating and configuring formatters for writing data ... 90
2.28. Regular expressions ... 92
3. Plotting ... 94
3.1. Getting started ... 94
3.2. Creating a plot ... 94
3.3. Customising title and subtitle ... 95
3.4. Managing layers ... 97
3.5. Showing and customising a legend ... 100
3.6. Customising plot properties ... 100
3.6.1. Command line equivalents ... 101
3.7. Setting margins ... 102
3.8. Saving and printing ... 102
3.9. Setting line and symbol styles ... 102
3.10. Customising axes ... 104
3.11. Drawing grid lines ... 110
3.12. Managing annotations ... 110
3.13. Drawing filled areas ... 113
3.13.1. Drawing filled areas between curves ... 114
3.14. Drawing a straight line ... 115
3.14.1. Drawing an arbitrarily-positioned straight line ... 116
3.15. Customising auxiliary axes ... 116
3.16. Changing the thickness of axes ... 118
3.17. Adding error bars ... 118
3.18. Switching to histogram mode ... 119
3.19. Adding subplots ... 120
3.20. Embedding monochromatic images in plots ... 121
3.21. Embedding RGB images in plots ... 122
3.22. Inserting math and special symbols ... 124
3.23. Creating a plot in batch mode ... 125
3.24. Drawing multiple plots per window ... 125
3.25. Colours in plots ... 126
3.26. Methods for colours, fonts and visibility ... 126
3.27. Invisible plots ... 127
3.28. Getting mouse coordinates on plots ... 128
3.29. More on plot methods ... 128
3.30. Worked example: Plot with an image ... 129
3.31. Worked example: Initial plot of this chapter ... 138
3.32. Worked example: Multi-panel plot ... 139
3.33. Worked example: Error bars ... 141
3.34. Worked example: Auxiliary axes ... 142
3.35. Worked example: Histograms ... 143
3.36. Worked example: Styles ... 146
3.37. Worked example: Two plots in one ... 149
3.38. Worked example: Coloured band ... 153
3.39. Worked example: Plot with PACS and SPIRE spectra ... 155
3.40. The TablePlotter ... 156

3.40.1. Invoking TablePlotter ... 157
3.40.2. Layout of the TablePlotter ... 158
3.40.3. Controls and functions ... 158
3.41. The Over Plotter ... 164
3.41.1. Invoke Over Plotter ... 164
3.41.2. Layout of Over Plotter ... 164
3.41.3. Controls and Functions ... 166
3.42. The Power Spectrum Generator ... 168
4. Working with images ... 170
4.1. Summary ... 170
4.2. Running image manipulation and analysis tasks ... 170
4.3. Importing and exporting images ... 171
4.3.1. Importing ... 171
4.3.2. Exporting ... 172
4.4. Viewing an image ... 173
4.5. Measuring angular distances ... 174
4.6. Creating masks ... 175
4.7. Viewing metadata and array data associated to an image ... 175
4.8. Saving an image ... 176
4.9. `SimpleImage` editing ... 177
4.10. Manipulating the axes (cropping, rotating, scaling...) ... 181
4.11. Manipulating fluxes ... 183
4.11.1. Image arithmetics ... 183
4.11.2. Smoothing images ... 187
4.11.3. Converting image units ... 187
4.11.4. Convolving images ... 188
4.12. Flagging saturated pixels ... 188
4.13. Getting cut levels ... 188
4.14. Combining images (stitching, RGB) ... 189
4.14.1. Stitching ... 189
4.14.2. Creating RGB images ... 189
4.15. Defining and using the World Coordinates System (WCS) ... 190
4.16. Creating intensity profiles ... 192
4.17. Creating contour plots ... 194
4.18. Creating histograms ... 196
4.18.1. Histograms via the command line ... 197
4.19. Finding and extracting sources ... 203
4.20. Fitting sources ... 210
4.21. Aperture photometry ... 211
4.21.1. Centroiding ... 211
4.21.2. Units and aperture photometry ... 212
4.21.3. Point sources ... 212
4.22. Comparing PSFs to point source profiles ... 221
4.22.1. Setting up and getting the data ... 223
4.22.2. Rotate the PSF and match it to the astronomical source ... 224
4.22.3. EEF Curves ... 228
4.22.4. Measuring the sky background scatter on PACS and SPIRE maps ... 230
4.22.5. Fitting the PACS PSF (for SPIRE it will be similar) ... 240
5. Spectral analysis ... 243
5.1. Summary ... 243
5.2. Spectra in HIPE ... 243
5.3. How to display spectra ... 243
5.3.1. Showing and Hiding spectra ... 244
5.3.2. Overplotting spectra ... 245
5.3.3. Viewing multiple plots ... 246
5.3.4. Zooming and Panning ... 247
5.3.5. Changing Display Axes ... 247
5.3.6. Changing Plot Properties ... 248

5.3.7. Viewing large datasets ... 249
5.3.8. Filtering and sorting what is viewed ... 251
5.3.9. Viewing Flags/masks and plot information ... 251
5.3.10. Viewing SpectralLineLists ... 252
5.3.11. Printing and saving ... 252
5.3.12. Plotting from the command line ... 252
5.4. Working on Spectra ... 252
5.4.1. Using the Spectrum Toolbox ... 253
5.4.2. Spectral Selection: extraction, and flagging ... 253
5.4.3. Spectrum Arithmetics ... 255
5.4.4. Spectral Averaging and Statistics ... 255
5.4.5. Spectral Manipulation: resampling, smoothing, replacing, gridding, stitching, and folding ... 257
5.4.6. Spectral Unit Conversion ... 257
5.4.7. Finding the integral under a line ... 257
5.4.8. Weight/error and flag/mask propagation ... 258
5.5. Dealing with baseline issues ... 259
5.5.1. General Standing Wave Removal Tool ... 260
5.5.2. Baseline Smoothing and Line Masking Tool ... 265
6. Spectral analysis for cubes ... 269
6.1. Summary ... 269
6.2. Cubes and the Spectrum Explorer ... 270
6.3. A message about cube coordinates and the WCS ... 271
6.4. A message about errors, weights, flags ... 272
6.5. A quick cube viewer: the Standard Cube Viewer ... 273
6.6. Using the Spectrum Explorer to look at cubes ... 274
6.6.1. Opening the Spectrum Explorer on a cube ... 274
6.6.2. Showing and hiding cube spectra; clearing stubborn spectra ... 276
6.6.3. Zooming and panning ... 277
6.6.4. Real-time spectrum display: preview panel ... 277
6.6.5. (Over)plotting spectra from multiple cubes ... 277
6.6.6. Linking the display of spectra from multiple cubes ... 278
6.6.7. A grid layout of the spectra in a cube ... 278
6.6.8. Viewing in subplots (multiple spectrum plots) ... 279
6.6.9. Standalone plot panel ... 280
6.6.10. Changing display axes ... 280
6.6.11. Changing plot properties and behaviour ... 280
6.6.12. A table of the plot—mouse interactions ... 282
6.6.13. Changing your Spectrum Explorer preferences ... 283
6.6.14. Viewing plot information ... 283
6.6.15. Viewing datapoint flags ... 283
6.6.16. Printing and saving ... 283
6.6.17. Creating a new variable from a plotted spectrum ... 284
6.6.18. A meta data list: and how to relate spaxel coordinates to index coordinates .. 284
6.6.19. Filtering what is viewed: not useful for cubes ... 285
6.6.20. Plotting from the command line ... 285
6.7. Working on cubes: the Spectrum and Cube Toolboxes ... 285
6.7.1. How to open the Toolboxes; getting extra help ... 286
6.7.2. Defining the input, looking at the output ... 286
6.7.3. Spectrum extraction and cube cropping ... 290
6.7.4. Spectrum arithmetics ... 293
6.7.5. Spectrum averaging/summing and statistics ... 293
6.7.6. Spectrum manipulation: resampling, smoothing, replacing, gridding, stitching, and folding ... 296
6.7.7. Spectrum flagging ... 298
6.7.8. Spectrum wave unit conversion ... 300
6.7.9. Weight/error and flag propagation ... 301
6.7.10. Making 2d flux maps from cubes ... 301

6.7.11. Velocity maps 303
6.7.12. Position-velocity maps 305
6.7.13. Removing the continuum from cubes 305
6.7.14. Dealing with baseline issues 306
6.7.15. Exporting to ASCII or FITS 306
6.7.16. Converting units for Cube Toolbox flux maps 306
6.8. Combining the PACS and SPIRE full SED for point sources 307
7. Spectral Fitting 309
7.1. Spectrum fitting 309
7.1.1. Using the Spectrum Fitter GUI: an overview 309
7.1.2. Using the Spectrum Fitter (command-line fitting): an overview 314
7.1.3. Fitting tips 315
7.2. Worked Example: Fitting a polynomial to the baseline/continuum 316
7.2.1. Worked Example: Fitting a polynomial to the baseline/continuum in the command line 322
7.3. Worked Example: Fitting a polynomial to a spectral cube (or any multi-spectrum dataset) 323
7.3.1. Worked Example: Fitting a polynomial to a spectral cube (or any multi-spectrum dataset) in the command line 325
7.4. Worked Example: Fitting Gaussians and a polynomial to a spectrum 326
7.4.1. Worked Example: Fitting Gaussians and a polynomial to a spectrum in the command line 331
7.5. Worked Example: Fitting multiple lines (Gaussians) and a Polynomial baseline to a cube and making maps of the results 333
7.5.1. With the GUI 333
7.5.2. On the command line 338
7.5.3. Making 2d maps from the fit results 339
7.6. Adding and Initialising Models 341
7.7. Configuring the Spectrum Fitter GUI to automatically apply a fit upon opening 342
7.8. Setting weights 342
7.9. Setting limits to model parameters 344
7.10. Fixing model parameters 345
7.11. Modifying Models 345
7.12. Applying a fit 346
7.13. Inspecting fit parameter results 346
7.14. Deleting models and excluding models from a fit 348
7.15. Resetting and restarting fitting 348
7.16. Saving a script 349
7.17. Saving the residual and models 350
7.18. Saving a SpectralLineList 353
7.19. Obtaining a line integral 353
7.20. Using Saved models 354
7.21. Automatic fitting of multiple datasets 355
7.22. Continuing work on the residual outside of the Spectrum Fitter GUI 358
7.23. Using the Combo Model 358
7.24. Models available to the fitter 359
7.25. How to add your own model 360
7.26. Selecting the best fitter engine 362
7.27. NaNs and the Spectrum Fitter 362
7.28. Making images from fitting results to cubes: the ParameterCube 362
7.28.1. After fitting with the MultiFitter tab of the Spectrum Fitter GUI 362
7.28.2. After fitting with the MultiFitter on the command line 363
7.28.3. Manipulating the images taken from the ParameterCube. 364
7.29. Calculating uncertainty and error after fitting 365
7.29.1. Introduction to errors or fitting and confidence 365
7.29.2. Practical information for getting the fitting errors in HIPE 367
7.29.3. Advanced practical information 367
7.30. Troubleshooting and limitations of the fitter 385

8. Unit Conversion 388
8.1. Units in HIPE 388
8.2. Built-in units and how to define new ones 389
8.2.1. Defining new units 393
8.3. How to convert data products units 393
8.3.1. Worked Example: Converting the units of an instance of a subclass of Spectrum1d 394
8.3.2. Worked Example: Converting the units of an instance of a subclass of Spectrum2d 394
8.3.3. Worked Example: Converting the units of an instance of a subclass of SimpleCube 395
8.3.4. Worked Example: Converting the units of an instance of SimpleImage 396
Index 397

List of Figures

1.1. Typical workflow for downloading, reprocessing and saving an observation. 3
1.2. Pools and storages 5
1.3. Preferences for data access 6
1.4. Workflows for retrieving observation data from the Herschel Science Archive 7
1.5. Logging in to the Herschel Science Archive 8
1.6. Accessing the Herschel Science Archive 9
1.7. The HSA view 9
1.8. The HUI Search tab. 10
1.9. HSA query result, compact view. 11
1.10. HSA query result, expanded view. 11
1.11. Information summary for an observation returned as a query result. 12
1.12. The Observing Log webpage. 12
1.13. Downloading an observation from the Herschel Science Archive. 13
1.14. Selecting which part of an observation to browse. 15
1.15. Product loaded into HIPE from the HSA. 15
1.16. The shopping basket of data to retrieve from the HSA 18
1.17. The shopping basket of data to retrieve from the HSA 18
1.18. A Herschel observation ready to be loaded into HIPE. 20
1.19. The My HSA preferences dialogue window. 22
1.20. A detail of the *Indexed Datasets* table showing the *Requires* and *Is required by* columns. 23
1.21. The *Advanced* tab of the My HSA preferences dialogue window. 24
1.22. The Product Browser. 25
1.23. Main window of Quick Analysis. 29
1.24. Quick Analysis - Browse. 30
1.25. Quick Analysis - Observation. 30
1.26. The Save Products tool. 31
1.27. Product export from HIPE into standard Herschel directory structure. 34
1.28. FITS save task dialogue window. 37
1.29. FITS read task dialogue window. 39
1.30. Structure of a FITS file produced from a `SimpleImage`. 42
1.31. Structure of a FITS file produced from a `SpectralSimpleCube` from a PACS observation. The two columns of the `ImageIndex` binary table extension are shown. 42
1.32. Structure of a FITS file produced from a `SpectralSimpleCube` from a SPIRE observation. The two columns of the `ImageIndex` binary table extension are shown. 43
1.33. Structure of a FITS file produced from a `SpectralSimpleCube` from a HIFI observation. 43
1.34. Structure of a FITS file produced from a `PacsRebinnedCube`. The two columns of the `ImageIndex` binary table extension are shown. 44
1.35. Structure of a FITS file produced from a `HifiTimelineProduct`. This product cannot be saved directly as a FITS file, but the summary table and each `DatasetWrapper` can. The dashed gray lines show the contents of each FITS file. 44
1.36. Structure of a FITS file produced from a `SpectrometerPointSourceSpectrum`. The five table columns are shown for the `SSWD4` extension. They are the same for the `SLWC3` extension. 45
1.37. Structure of the `History` extension of a FITS file created from a Herschel product. Column names for each of the three binary table extensions are shown. 46
1.38. The SAMP Hub Monitor window. 54
2.1. Excerpt from the output of exportSpectrumToAscii 59
2.2. The TableDataset resulting from running the example script 59
2.3. The initial SourceListProduct generated by sourceExtractorSussextractor. 61
2.4. Excerpt from the text file written from the SourceListProduct. 61
2.5. The reconstituted SourceListProduct with data read in from the text file. 62
2.6. Excerpt from the a Spitzer spectrum product. 62
2.7. The table dataset resulting from running the example script. 64

2.8. Excerpt from the Planck Early Cold Cores Catalogue. 65
2.9. Excerpt from ReadMe file for Planck Early Cold Cores Catalogue. 65
2.10. The table dataset resulting from running the example script. 66
2.11. A table dataset imported from a CSV file. 68
2.12. A table dataset imported from a space-separated-value file. 69
2.13. A simple table dataset. 72
2.14. A simple table dataset. 73
2.15. Input spectrum for the exportSpectrumToAscii task. 75
3.1. Some of the features of HIPE plots. If you are reading the HTML version of this manual, click on any of the blue labels to jump straight to the relevant section. 94
3.2. The *Property Panel* window. 101
3.3. Available filling closure types. 113
3.4. The filling patterns produced by the three `LineHatchPaint` objects shown in the previous example. 114
3.5. The filled area between the *sinc(x)* curves. 115
3.6. Plot with a customised auxiliary axis. 117
3.7. The same data set plotted as LINECHART (default) and HISTOGRAM. 119
3.8. Plot with embedded subplot. 120
3.9. RGB image in a plot. 124
3.10. Using special characters for labels. 124
3.11. Classes involved in plot operations. 129
3.12. The result of the commented plot example presented in this section. 130
3.13. The result of the plot example created with the simplified version of the script. 136
3.14. A plot with four panels. 139
3.15. A plot with horizontal and vertical error bars. 141
3.16. A plot with three layers and customised auxiliary axes. 142
3.17. A plot with three panels, each containing superimposed histograms. 144
3.18. A plot using several styles and colours for lines and plot symbols. 146
3.19. A plot made of two independent plots. 150
3.20. A plot with different symbol styles, error bars and a coloured horizontal band. 153
3.21. The result of the commented plot example presented in this section. 155
3.22. Viewers available for a table dataset in the product viewer, among them TablePlotter and OverPlotter. 157
3.23. Layout of the TablePlotter GUI. 158
3.24. The plot with selected (blue) and hidden (red crosses) data points. 161
3.25. Extract Selected Data from Multi Columns to a New DataSet. 162
3.26. Simple overlay plots of different columns plotted against the same X-axis are created by marking the Overlay field. 163
3.27. Preferences: Complex data can be displayed in four different ways as shown in this properties menu. 163
3.28. The main panel of Over Plotter is very similar to that of the Table Plotter. New features include the Layer Controls panel and the synchronization buttons. This Over Plotter is in *All Layers* mode. 165
3.29. This Over Plotter is in "Single Layer" mode. The primary layer is displayed in its selected colour and the secondary layer is displayed in green. All other layers are displayed in grey colour. 166
3.30. This Over Plotter is in "Single Layer" mode. The primary layer is displayed in its selected colour and the secondary layer is displayed in green. All other layers are displayed in grey colour. These are the same layers as in the previous figure, but after selecting Layer 1 to become prime. 166
3.31. A complex example for illustration. The Over Potter is in "Single Layer" mode. The primary layer is displayed in blue with large symbols and connected by a line. The Y-axis is set to logarithmic mode. The secondary layer is displayed in green with large filled diamonds. The third layer is displayed in grey colour. 168
3.32. A signal timeline displayed in Table Plotter that the Power Spectrum generator can be applied to. 168
3.33. Main view of the Power Spectrum Generator. 169
3.34. Displaying the newly created power spectra in the Table Plotter. 169

4.1. Finding variable types in the *Variables* view. 171
4.2. Viewing an image in HIPE. 173
4.3. Measuring angular distances on an image. 175
4.4. Opening the Dataset Viewer from the Outline view. 176
4.5. Colour map window. 177
4.6. Cut level selection window. 178
4.7. The annotation toolbox. 178
4.8. Adding annotations to a Display. 180
4.9. Jython code appearing in the annotation toolbox. 181
4.10. Example image transformation dialogue window. Rotating an image using the "rotate" task. Several interpolation options are available. 183
4.11. Example image arithmetic dialogue window. 187
4.12. The createRgbImage task dialogue window. 190
4.13. The intensity profile below the image. 192
4.14. Dialogue window for `automaticContour` . 195
4.15. Circle histogram area selection and parameter selection. 196
4.16. Display of the histogram task results, held in the histogram output dataset. 197
4.17. List of parameters for the two source extraction tasks. 204
4.18. The list of sources shown in the Product Viewer, with the internal dataset highlighted. 205
4.19. Opening the SkyMask toolbox. 208
4.20. Drawing a region of interest on the image. 209
4.21. Aperture photometry with an annular sky aperture as displayed in HIPE. 213
4.22. Aperture photometry results plot and tables. Note that n.a. stands for "not applicable" and typically occurs when units are not assigned to the image. 213
5.1. The Spectrum Explorer for a single spectrum. 244
5.2. The new layout properties panel. 248
5.3. Filters on attributes. 251
5.4. fitFringe task UI. 261
5.5. Plot with standing waves. 262
5.6. Overplot of fitFringe results. 262
6.1. HIFI cube: data arrays 273
6.2. The Standard Cube viewer: zoom to fit is indicated 274
6.3. The Spectrum Explorer with a cube loaded 275
6.4. The cube comparison buttons 278
6.5. Mosaic/raster view 278
6.6. The Subplot menu. 279
6.7. changing layer properties 282
6.8. The arrays in a cube. 283
6.9. Tab arrangement in the Cube Explorer. 287
6.10. Combination of PACS and SPIRE spectra. 308
7.1. Selecting one spectrum to fit with the Spectrum Fitter GUI. Here a pixel near the centre of the cube (highlighted with a green box) is displayed in the top left Spectrum Explorer and the second subband of a HIFI WBS spectrum is displayed in the bottom right Spectrum Explorer. 310
7.2. The SFG accessed via the Spectrum Explorer, and showing a Polynomial fitted to one pixel from a SPIRE cube. The labelled Spectrum Explorer and Fitter GUI panes are described in the text below. 311
7.3. The working area of the SFG. 312
7.4. The MultiFit_Parms output. 314
7.5. Plot one spectrum (a spaxel/pixel) and open the Spectrum Fitter GUI. 317
7.6. Add a Polynomial model using *addModel* and press *Accept* to fit. 318
7.7. Reset the Spectrum Fitter GU to start work on the original spectrum again.*addModel* and press *Accept* to fit. 318
7.8. Set weights by opening the *Weights* tab and drawing a range on the spectrum. 319
7.9. After setting weights go back to the *Models* tab and re-initialise the fit. 320
7.10. Setting the weights to zero at the line edges still did not produce a satisfactory fit. 320
7.11. Set weights to one either side of the line in the *Weights* tab before reinitialising the Polynomial fit in the *Models* tab. 321
7.12. A script, the models and the residual can be saved in the *Export* tab. 321

7.13. The contents of MultiFit_Parms ... 325
7.14. Plot one spectrum (a spaxel/pixel) and open the Spectrum Fitter GUI. ... 327
7.15. The cross-hair indicating the layer the cube is displayed at may be obtrusive, you can move it to the edge of the spectrum in the Spectrum Explorer *Data Selection Panel*. ... 328
7.16. Add a Polynomial model using *addModel* in the *Models* tab. ... 328
7.17. Add a Gaussian model using *addModel* and set the position and amplitude of the peak by clicking on the spectrum. ... 329
7.18. Use *addModel* to add another Gaussian. ... 329
7.19. Add another Gaussian for the absorption, you will need to zoom out in order to set the amplitude of the line below zero. Press *Accept* to fit. ... 330
7.20. The total model and Polynomial fit are plotted over the original spectrum, while the Gaussian models appear below with the residual. ... 330
7.21. A script and the model parameters (and also the residual) can be saved from the *Export* tab. ... 331
7.22. Plot one spectrum (a spaxel/pixel) and open the Spectrum Fitter GUI. ... 334
7.23. Set weights by opening the *Weights* tab and drawing a range on the spectrum. ... 334
7.24. Add a Polynomial model using *addModel* and press *Accept* to fit. ... 335
7.25. Add a Gaussian model using *addModel* and set the position and amplitude of the peak by clicking on the parameter boxes ("Amplitude" or "X-Position") to the right and then on the spectrum. ... 335
7.26. To fit an entire cube, use the MultiFit tab ... 336
7.27. The contents of MultiFit_Parms ... 337
7.28. Load models. ... 355
7.29. The MultiFit_Parms output ... 357
7.30. Linear model fitting. ... 369
7.31. Non-linear model fitting. ... 371
7.32. Non-linear model fitting. ... 373
7.33. Non-linear model fitting. ... 376
7.34. Mixed sine model fitting. ... 377
7.35. Combined model fitting. ... 381
7.36. Combined model fitting. ... 385

List of Tables

2.1. Regular expressions. 92
3.1. Plot line styles 103
3.2. Symbol codes and images. 103
7.1. SpectrumFitter models 359
7.2. Spectrum fit model types and their use. 360
8.1. Built-in units. 389

List of Examples

1. Printing "hello" to the console. xxvi
2. How to retrieve an observation given an observation ID. xxvi
3. Setting a title plot. xxvi
4. Setting properties using Jython syntax. xxvii
1.1. Getting an observation from the HSA and saving it locally. 3
1.2. Creating a variable from an observation previously saved to disk. 3
1.3. How to save a product to a local pool, in the background. 4
1.4. Getting an observation from the HSA given the observation ID. 4
1.5. Saving an observation to disk from the HSA given the observation ID. 13
1.6. Getting an observation from the HSA given an observation ID. 14
1.7. Setting on the connected status of the MyHSA pool. 14
1.8. Setting off the connected status of the MyHSA pool. 14
1.9. Retrieving an observation from the HSA given the observation ID. 16
1.10. Browse all the versions of an observation (part 1). 16
1.11. Browse all the versions of an observation (part 2a). 17
1.12. Browse all the versions of an observation (part 2b). 17
1.13. Browse all the versions of an observation (part 3). 17
1.14. Downloading multiple observations from an array of obs ids. 18
1.15. Retrieving several observations from the HSA as tar.gz and opening them in HIPE. 19
1.16. Retrieve an observation from the HSA given the observation ID. 19
1.17. Load an observation from disk into a new variable. 20
1.18. Retrieving an observation from the HSA as a tar.gz and opening it in HIPE. 21
1.19. Load an observation from disk, specifying both path and observation ID. 21
1.20. Perform a query on a local store using the result of another query. 25
1.21. Retrieve an observation given the observation ID. 26
1.22. Load an observation from disk specifying both the observation ID and the pool directory. 27
1.23. Load an observation from disk specifying the observation ID, the local pool name and the pool location. 27
1.24. Several examples using the getObservation task. 27
1.25. Save an observation to disk specifying the pool name and a tag. 32
1.26. How to rebuild the index of a local pool. 33
1.27. Second option for rebuilding the index of a local pool. 33
1.28. How to get the current workind directory for the Jython interpreter. 37
1.29. Creating a new empty data product and writing it to disk as a FITS file. 37
1.30. Creating a SimpleImage with random data and saving it to disk as a FITS file, reading it back afterwards. 38
1.31. Using a dataset as a wrapper to store an array in a FITS file. 38
1.32. Load a product from a FITS file. 39
1.33. Setting a metadata property to a StringParameter value. 40
1.34. Create FITS file from random data and read it back. 40
1.35. Printing a FineTime formatted string to the console. 45
1.36. Importing a non-Herschel FITS file with the simpleFitsReader task. 47
1.37. Importing non-Herschel FITS files using specific image import tasks. 47
1.38. Importing non-Herschel FITS files using specific spectral import tasks. 47
1.39. Complete example to convert a Spectrum1d class to a CLASS FITS file. 49
1.40. Converting a VO table in XML format to TableDataset. 54
1.41. Writing a TableDataset from an observation to an XML-based VO file. 55
1.42. Writing a synthetic TableDataset to an XML-based VO file. 55
2.1. Creating a TableDataset with a Column made up of array data. 57
2.2. Read a numeric array from a file and loop over its values. 57
2.3. Read a numeric array from a file and tokenise its values in a loop. 57
2.4. Script to export a SpectralSimpleCube to ASCII, and read back into a TableDataset 58
2.5. Script to generate a SourceListProduct, write it to a text file, and read it back into HIPE 60
2.6. Worked example for reading in a Spitzer spectrum in "IPAC table" format 62

2.7. Complete script for reading in the Planck Early Cold Cores Catalogue. 65
2.8. A standard comma-separated-value (CSV) file with a four-line header. 67
2.9. A standard comma-separated-value (CSV) file with only column titles specified. 67
2.10. Reading a table from a file, specifying its tabular format as CSV. 68
2.11. A space-separated-value file of the type that can be imported into HIPE with default options. 68
2.12. A space-separated-value file with only column titles specified. 69
2.13. Read a table from an ASCII file, specifying that its values are space-separated. 69
2.14. Read a table from an ASCII file, specifying the input format as IPAC. 70
2.15. Reading a table from an ASCII file, without specifying any input format. 71
2.16. Write a table to an ASCII file. 72
2.17. Creating a formatter that separates values using a single space character. 73
2.18. Creating a space-separated formatter and using it to write a table as an ASCII file. 74
2.19. Writing spectrum data to an ASCII file. 76
2.20. Writing spectrum data to an ASCII file without including metadata. 76
2.21. Writing spectrum data to an ASCII file, including flags. 77
2.22. Writing spectrum data to an ASCII file, including weights. 77
2.23. Writing spectrum data to an ASCII file, concatenating the spectral segments. 77
2.24. Writing spectrum data to an ASCII file, specifying a selection of spectral indices. 78
2.25. Writing spectrum data to an ASCII file, with a literal array of spectral indices. 78
2.26. Writing spectrum data to an ASCII file, specifying a selection of spectral segments. 79
2.27. Writing spectrum data to an ASCII file, specifying a literal array of spectral segments. 79
2.28. Creating a space-delimited formatter. 79
2.29. Writing spectrum data to an ASCII file, specifying a space-delimited formatter. 79
2.30. Writing a table to disk. 80
2.31. Reading a table from a file, specifying the ADVANCED type of table for parsing. 80
2.32. Reading a table from a file, specifying the ADVANCED table type for parsing and skipping header lines. 81
2.33. Reading a table from an ASCII file, with ADVANCED type and character ignore options for the parser. 81
2.34. Reading a table from an ASCII file, with ADVANCED type and an ignore pattern that trims spaces at the beginning. 82
2.35. Reading a table from an ASCII file, with ADVANCED type and guessing all value types. 83
2.36. Reading a table from an ASCII file, with ADVANCED type and parsing all values as doubles. 83
2.37. Reading a table from an ASCII file, with ADVANCED type and a custom table template. 83
2.38. Reading a table from an ASCII file, with ADVANCED type and assuming 18 character-wide columns and space separators. 84
2.39. Reading a table from an ASCII file, with ADVANCED type and providing a fully customised parser. 84
2.40. Reading a table from an ASCII file while writing the parsing configuration to a file for reuse. 84
2.41. Reading a table from an ASCII file, specifying a parsing configuration file. 85
2.42. Writing a table to an ASCII file without a header. 85
2.43. Writing a table to an ASCII file including metadata. 86
2.44. Writing a table to an ASCII file including metadata with a custom prefix. 86
2.45. Writing a table to an ASCII file with a custom formatter. 86
2.46. Writing a table to an ASCII file saving the writing configuration to another file. 87
2.47. Writing a table to an ASCII file, specifying a previously-saved writing configuration file. 87
2.48. Creating a TableTemplate with 3 columns. 88
2.49. Customising a TableTemplate with column names, types, units and descriptions. 88
2.50. Setting column descriptions for a partial set of columns. 88
2.51. Creating a unit variable and checking if it is built in the system.s 88
2.52. Creating and customising a TableTemplate in one step. 88
2.53. Creating a comma-separated CSV parser. 89

2.54. Creating a CSV parser with a dollar sign for quoting values. 89
2.55. Creating a parser that specifies the widths of the columns. 89
2.56. Creating a parser based on a regular expression. 89
2.57. Creating a CSV parser that ignores line starting with a specific string. 90
2.58. Creating a fixed width parser that skips a number of header lines. 90
2.59. Create a parser based on a regular expression that trims the lines of the file. 90
2.60. Creating a CSV formatter that specifies a space character as a delimiter. 91
2.61. Creating a CSV formatter with the default options. 91
2.62. Creating a CSV formatter that uses a tab as a delimiter. 91
2.63. Creating a fixed width formatter. 91
2.64. Creating a CSV formatter that includes a header. 91
2.65. Creating a fixed width formatter with commented metadata. 91
2.66. Creating a CSV formatter with a custom comment prefix. 91
2.67. Creating a CSV formatter that includes a header, delimited with spaces and with comments with a custom prefix. 92
2.68. Creating a fixed width formatter. 92
3.1. Creating a double array and populating with a range of values. 94
3.2. Creating a plot and adding layers to it in two steps. 95
3.3. Creating a plot and adding a layer to it in one step. 95
3.4. Setting the dimensions of the plot. 95
3.5. Creating a plot with initial dimensions. 95
3.6. Creating a plot with initial dimensions and auto-adjust. 95
3.7. Setting title and subtitle text in a plot. 95
3.8. Setting the visibility for a plot's title and subtitle. 95
3.9. Sets the text to be displayed. 96
3.10. Sets the horizontal alignment. Possible values are LEFT, CENTER and RIGHT. 96
3.11. Sets the vertical alignment. Possible values are MIDDLE, TOP and BOTTOM. 96
3.12. Sets the position of the title. Possible values are BOTTOMCENTER, BOTTOMLEFT, BOTTOMRIGHT, TOPCENTER, TOPLEFT, TOPRIGHT and CUSTOMIZED. If set to CUSTOMIZED, the title position is controlled by the `setLocation` method. 96
3.13. Sets the x and y location of the title, automatically switching the position to CUSTOMIZED. 97
3.14. Sets the x position of the title. 97
3.15. Sets the x and y position of the title. Equivalent to `setLocation` 97
3.16. How to create an additional layer in a plot. 97
3.17. Changes the name (and thus the legend) of the layer. 98
3.18. Sets the line style of the layer. Possible values are NONE, SOLID, MARKED, DASHED and MARK_DASHED. You can also use the numbers 0, 1, 2, 3 and 4. 98
3.19. Sets the size of the layer symbols, in points. 98
3.20. Sets the shape of the symbol. See Table 3.2 for the names and numbers of available symbols. 98
3.21. Sets the line thickness, in points. Only for line plots. 98
3.22. Sets the style of the layer. The input parameter is an instance of the `Style` class. For more information on creating styles see Section 3.9 98
3.23. Adds a point to the layer. 99
3.24. Adds a set of points to the layer. 99
3.25. Waits for mouse click and returns the coordinates of the pointer. Returns a `double[]` 99
3.26. Like the previous method, but this one does the job for `n` successive clicks. Returns a `double[][]` , that is, an array of double arrays. Each array holds the coordinates of a mouse click. 99
3.27. The difference with respect to the previous two methods is that this time the coordinates of the layer point closer to the mouse pointer are returned. Returns a `double[]` 99
3.28. Like the previous method, but this one does the job for `n` successive clicks. Returns a `double[][]` , that is, an array of double arrays. Each array holds the coordinates of the data point closest to each mouse click. 99
3.29. Returns an `int` representing the index of the current layer inside the `PlotXY` 99
3.30. Sets whether the layer is shown in the legend. Getter method `isInLegend` available. 100
3.31. Customising the appearance of the different plot symbols. 104

3.32. Adding a customised range and title to a plot axis. 104
3.33. If flag is true, adjusts the range of the specified axis so that all data points will be shown. 105
3.34. Sets the range of the axis. The lower and upper limit are passed as separate double parameters. Note that there is no "Jython style" example because lists and tuples in Jython use the same syntax. See the row just below for the example. 105
3.35. Set the range of the specified axis to values between lower and upper. Note that instead of two arguments for the lower and upper limits, there is one array argument containing both values. 105
3.36. Show grid lines for the specified axis if flag is true, hide the grid lines otherwise. 105
3.37. Sets the axis type. Available types are LINEAR , LOG , DATE , RIGHT_ASCENSION and DECLINATION 106
3.38. Gets the axis orientation, either HORIZONTAL or VERTICAL . Setter method *not* available. 106
3.39. Sets the axis to a linear scale. Equivalent to setType(Axis.LINEAR) 106
3.40. Sets the axis to a logarithmic scale. Equivalent to setType(Axis.LOG) 106
3.41. Sets whether values on the axis are displayed in inverted order (for instance, right to left for the abscissa). 106
3.42. Sets the position of the axis with respect to the plot. Possible values are TOP or BOTTOM for abscissa axis and LEFT or RIGHT for ordinate axis. Get method available. 106
3.43. Sets the text to be displayed. 106
3.44. Sets the physical height of the major ticks. 107
3.45. Sets the interval between major ticks, in axis units. 107
3.46. Sets the side of the axis on which the ticks are drawn. Possible values are INWARD , OUTWARD and BOTH 107
3.47. Sets the number of minor ticks displayed between two major ticks. 107
3.48. Sets whether the number of ticks on the axis is adjusted automatically to avoid overlapping labels. Getter method isAutoAdjustNumber available. 107
3.49. Sets whether grid lines are displayed for major ticks. Getter method isGridLines available. 107
3.50. Sets the colour of labels. 108
3.51. Sets the font of labels. 108
3.52. Sets the physical size of labels. 108
3.53. Sets the interval (in ticks) between successive labels. For example, a value of two displays a label on every other tick. 108
3.54. Sets the orientation of the labels (0 for horizontal, 1 for vertical). 108
3.55. Replaces the current labels with the values in an array of String objects. 108
3.56. Sets the side of the axis on which the labels are drawn. Possible values are INWARD and OUTWARD 109
3.57. Sets the x axis to the specified Axis instance. The x axis will be reinstantiated with its default settings plus whatever is indicated in the Axis instance. So any prior manipulations of the axis are lost. 109
3.58. Sets the range of the x axis. 109
3.59. Sets the title of the x axis. 109
3.60. Sets the type of the x axis. Available types are LINEAR and LOG. 109
3.61. Sets the x and y values, passed as elements of an "array of arrays" of size two. Get method available. Note that there is no setYx method! 109
3.62. Sets the x and y values, passed as two separate arrays. Note that there is no setYx method! 110
3.63. Sets the ordinate values. Get method available. Note there is a getX method but not a setX method. 110
3.64. Removes the x axis and uses the given axis as a shared x axis. 110
3.65. Creates an empty annotation. 111
3.66. Creates an annotation with the given text. 111
3.67. Creates an annotation with the given position and text. 111
3.68. Sets the position angle, in degrees, counterclockwise. 111
3.69. Sets the horizontal alignment. 111
3.70. Sets the vertical alignment. 111

3.71. Sets the x position. 111
3.72. Sets the x and y position. 111
3.73. Sets the text of the annotation. 112
3.74. Gets the unique id of the annotation. No setter available. 112
3.75. Adds an Annotation object to the layer. 112
3.76. Adds several Annotation objects to the layer. The input Annotations are passed as an array. 112
3.77. Sets an annotation to a given id, replacing what was there before. 112
3.78. Replaces all the annotations with the ones provided in the array. 112
3.79. Retrieves one annotation from the layer. 112
3.80. Retrieves all the annotations from the layer. The annotations are returned as an array. 112
3.81. Removes the annotation with the specified id. 113
3.82. Removes all the annotations. 113
3.83. Filling the areas between the curves. 114
3.84. Drawing an arbitrarily-positioned straight line using LayerXY. 116
3.85. Appends a low and high error value of x. 118
3.86. Appends a set of low and high error values of x. 118
3.87. Sets low and high error values of x. 119
3.88. Sets the low and high error values of x. 119
3.89. Returns an array of Ordered1dData with length equal to 2. 119
3.90. How to add and customise a subplot inside a main plot. 120
3.91. How to manually combine three bands to create an RGB image, respecting WCS information. 122
3.92. Batching several plots to improve speed. 125
3.93. Distributing layers inside a main plot. 125
3.94. Sets whether the component is visible. 127
3.95. Sets the foreground colour of the component. 127
3.96. Sets the font of the component. You can specify a font by giving a name, style and point size. Available font styles are PLAIN , BOLD and ITALIC . You can also use the numbers 0, 1 and 2, respectively. 127
3.97. Sets the name of the font of the component. 127
3.98. Sets the size of the font of the component. 127
3.99. Sets the style of the font of the component. Possible values are PLAIN , BOLD and ITALIC . You can also use the numbers 0, 1 and 2, respectively. 127
3.100. Creating a plot that is not drawn on the screen. 128
3.101. Complete example that demonstrates the use of the PlotXY class. 130
3.102. Version of the example above using the Display class. 136
3.103. Plotting the figure that appears at the beginning of this chapter. 138
3.104. Distributing multiple plots using panels. 139
3.105. Plotting horizontal and vertical error bars. 141
3.106. Adding multiple layers and customised auxiliary axes to a plot. 142
3.107. Using several panels with histograms. 144
3.108. Plotting with customised line and plot styles. 146
3.109. Plotting two independent subplots. 150
3.110. How to add error bars and customised horizontal bars to a plot. 153
3.111. Complete example using PACS and SPIRE data of AFGL 2688. 155
4.1. Printing the type of a variable. 171
4.2. Setting the image value of a SimpleImage object. 171
4.3. Set the wavelength value of an image. 172
4.4. Set the wavelength value of an image, including units. 172
4.5. Setting the units of an image directly. 172
4.6. Setting the exposure of an image. 172
4.7. Setting the error and coverage datsets of an image. 172
4.8. Constructing a Display object from an image. 173
4.9. Saving the current view of a Display object. 176
4.10. Saving an image to disk without blocking the GUI. 177
4.11. Adding annotations and setting formatting options in a Display object. 180
4.12. Filling shapes with oversized lines in a Display. 180

4.13. Opening the annotation toolbox programmatically. 181
4.14. Opening the colour and cut level dialogues programmatically. 181
4.15. Clamping an image with explicit limits. 181
4.16. Cropping an image specifying the rectangle dimensions. 182
4.17. Regridding an image and getting the flux change factor. 182
4.18. Rotating an image without specifying an interpolation method. 182
4.19. Rotating an image with a specific interpolation method. 182
4.20. Scaling an image both with and without custom interpolation. 182
4.21. Translating an image using two different sets of coordinates. 183
4.22. Transpose an image with a horizontal flip. 183
4.23. Adding images and also scalars to images. 184
4.24. Subtracting images and also scalars from images. 184
4.25. Multiplying images and also images by scalars. 184
4.26. Changing the pixel size of an image while ensuring flux conservation. 184
4.27. Dividing images and also images by scalars. 185
4.28. Integer division of images and image by scalar. 185
4.29. Applying the absolute value to the intensity values of an image. 185
4.30. Rounding the intensity values of an image. 185
4.31. Rounding the intensity values of an image to the largest previous integer. 185
4.32. Rouding the intensity values of an image to the smallest following integer. 185
4.33. Raising the intensity values of an image to the n-th power. 185
4.34. Squaring the intensity values with an specific task. 186
4.35. Taking the square root of the intensity values of an image. 186
4.36. How to obtain the natural logarithm of the intensity values of an image. 186
4.37. How to obtain the base 10 logarithm of the intensity values of an image. 186
4.38. How to obtain the base N logarithm of all intensity values of an image. 186
4.39. How to obtain the exponential of the intensity values of an image. 186
4.40. Raising 10 to the intensity value for all image intensity values. 186
4.41. Raising N to the intensity value for all image intensity values. 186
4.42. Smoothing an image using four different algorithms. 187
4.43. Converting image units with a specific task. 187
4.44. Convolving an image with a specific kernel. 188
4.45. Flagging pixels whose value exceeds the limit provided. 188
4.46. Getting the minimum and maximum values for a certain cut-off percentage. 188
4.47. Mosaicking images with the help of an intermediate array. 189
4.48. Creating an RGB image with specific weights for each channel. 189
4.49. Creating an RGB image with cut levels for each channel. 189
4.50. Creating an RGB image with weights, overall cut level and new WCS information. 190
4.51. Printing the WCS information of an image. 190
4.52. Creating WCS coordinate data. 191
4.53. Creating WCS coordinate data with parameters. 191
4.54. Creating a profile plot of an image. 193
4.55. Retrieving the pixel coordinates of the beginning of the line. 193
4.56. Retrieving the sky coordinates of the beginning of the line. 193
4.57. Converting the profile plot to a Double1d. 193
4.58. Getting the unit of the profile plot. 193
4.59. Creating the pixel profile plot of a synthetic image. 194
4.60. Creating an automatic line contour specifying distribution and plotting details. 194
4.61. Creating a manual line contour providing contour level values. 194
4.62. Creating a contour containing one single contour level. 194
4.63. Plotting automatically-generated contours on top of an image. 194
4.64. Creating a histogram from an image. 198
4.65. Getting the number of bins from the histogram. 199
4.66. Getting the lower cut level of the histogram. 199
4.67. Converting the histogram to a table dataset. 199
4.68. Getting the values of the bins as a Double1d array. 199
4.69. Getting the count for each histogram bin as a Double1d. 200
4.70. Get the unit of the histogram values. 200

4.71. Getting the centre pixel coordinates of the circle histogram. 200
4.72. Getting the centre sky coordinates of the circle histogram. 200
4.73. Getting the radius of the circle histogram in pixels. 200
4.74. Getting the radius of the circle histogram in arcseconds. 200
4.75. Getting the centre pixel coordinates for the ellipse histogram. 201
4.76. Getting the centre sky coordinates for the ellipse histogram. 201
4.77. Getting the width of the ellipse histogram in pixels. 201
4.78. Getting the width of the ellipse histogram in arcseconds. 201
4.79. Getting the upper left corner pixel coordinates of a rectangle histogram. 201
4.80. Getting the upper left corner sky coordinates of a rectangle histogram. 202
4.81. Getting the width in pixels of a rectangle histogram. 202
4.82. Getting the width in arcseconds of a rectangle histogram. 202
4.83. Getting the edges of a polygon histogram. 202
4.84. Getting the vertices of a polygon histogram as a table dataset. 202
4.85. Getting the vertices of a polygon histogram as a Double2D array. 203
4.86. Getting the edges of a polygon histogram in sky coordinates. 203
4.87. Creating a residual image subtracting the sources for the original image. 205
4.88. Creating an image containing only the sources of the original image. 205
4.89. Extracting the results of the task from the output array. 205
4.90. Removing a row from a sources list. 206
4.91. Plotting the source list along with the image with the help of the Display class. 206
4.92. Customising the size of the position circles when plotting sources with Display. 206
4.93. Making the position circles of plotted sources proportional to the flux intensity. 207
4.94. Plotting fully customised position circles by passing Color and Float1d objects. 207
4.95. Plotting position circles that take the sizes as sky coordinates by passing a Wcs object. 207
4.96. Creating a rectangular region of interest (SkyMask) using sky coordinates in decimal degrees. 207
4.97. Creating a circular region of interest (SkyMask) using sky coordinates in decimal degrees. 207
4.98. Creating a region of interest from a bidimensional array of booleans. 208
4.99. Applying the boolean OR operation between SkyMasks. 208
4.100. Inverting a region of interest. 208
4.101. Joining the areas of three different SkyMasks. 208
4.102. Masking an image with a complex, boolean SkyMask. 209
4.103. Retrieving the source list dataset from the results output list of a source extraction task. .. 209
4.104. Creating a SourceListProduct from a source list table dataset. 210
4.105. Creating a SourceListProduct from a source list dataset. 210
4.106. Fitting sources in an image. 211
4.107. Performing annular sky aperture photometry on a PACS map. 214
4.108. Performing annular sky aperture photometry on a SPIRE map. 214
4.109. Getting the centre pixel coordinates for the target. 215
4.110. Getting the centre sky coordinates for the target. 215
4.111. Getting the target radius in pixels. 215
4.112. Getting the target radius in arcseconds. 215
4.113. Getting the outer target radius in pixels. 215
4.114. Getting the inner radius of the sky estimation annulus in arcseconds. 216
4.115. Getting the name of the algorithm used by the aperture photometry task. 216
4.116. Checking if the aperture photometry task considers fractional or entire pixels. 216
4.117. Getting the results of the aperture photometry task as a table dataset. 216
4.118. Getting the results of the aperture photometry task as a Double2d table. 216
4.119. Getting the total flux (target+sky). 217
4.120. Getting the total number of pixels (target+sky). 217
4.121. Getting the flux intensity averaged by the total pixels (target+sky). 217
4.122. Getting the curve of growth for the results of aperture photometry task. 217
4.123. Getting the growth radius for the results of the aperture photometry task. 217
4.124. Getting the growth flux column of the results of the aperture photometry task. 217
4.125. Getting the intensity plot as a table dataset. 218
4.126. Getting the sky intensity radius of the intensity plot. 218

4.127. Getting the sky intensity values from the intensity plot as a Double1d. 218
4.128. Performing rectangular sky aperture photometry. 219
4.129. Getting the width in pixels of the rectangular aperture. 219
4.130. Getting the width in arcseconds of the rectangular aperture. 220
4.131. Getting the upper left corner of the rectangular aperture in pixel coordinates. 220
4.132. Getting the upper left corner of the rectangular aperture in sky coordinates. 220
4.133. Performing fixed sky aperture photometry. 221
4.134. Running the aperture photometry correction task for point sources. 221
4.135. Importing the required classes for the PSF-point source comparison example. 223
4.136. Reading the data from a PACS calibration observation (PSF-point source comparison). .. 224
4.137. Retrieving the beam profile from the latest SPIRE calibration (PSF-point source comparison). 224
4.138. Retrieving the necessary properties for PACS and SPIRE (PSF-point source comparison). 224
4.139. Creating auxiliary variables with the obsid and band (PSF-point source comparison). 224
4.140. Rotating the PSF and matching to the source (PSF-point source comparison). 225
4.141. Updating the Wcs information (PSF-point source comparison). 225
4.142. Matching the WCS of the beam and the source (PSF-point source comparison). 225
4.143. Retrieving the RA and declination in decimal degrees (PSF-point source comparison). ... 225
4.144. Retrieving the X, Y coordinates in pixels (PSF-point source comparison). 225
4.145. Setting the coordinates to the centre of the map (PSF-point source comparison). 226
4.146. Finding the exact centre of the PSF (PSF-point source comparison). 226
4.147. Find the exact centre of the point source (PSF-point source comparison). 226
4.148. Matching the WCS of the PSF to the coordinates of the source (PSF-point source comparison). 227
4.149. Defining the coordinates for building the EEF curve. 228
4.150. Creating an array of apertures. 228
4.151. Removing the first aperture from the array. 229
4.152. Creating a variable for storing two arrays of fluxes. 229
4.153. Performing the point source aperture photometry for every aperture in the array. 229
4.154. Performing the PSF aperture photometry for every aperture in the array. 229
4.155. Scaling the fluxes of the PSF and point source to peak (min aperture). 229
4.156. Defining apertures for the beam model. 229
4.157. Scaling the fluxes of the PSF and point source to median. 230
4.158. Plotting the results of this comparative aperture photometry. 230
4.159. Measuring the FWHM of the point source using a cubic spline interpolator. 230
4.160. Measuring the FWHM of the PSF using a cubic spline interpolator. 230
4.161. Comparing both values for FWHM. 230
4.162. Measuring the sky background scatter of beam and astro source. 231
4.163. Defining recommended values for circular photometry. 231
4.164. Performing the aperture photometry for all the small spot apertures. 231
4.165. Removing outliers from the sample. 231
4.166. Performing PSF comparison for PACS point sources. 232
4.167. Performing PSF comparison for SPIRE point sources. 236
4.168. Setting the camera and aperture variables. 240
4.169. Defining scaling factors for the data. 240
4.170. Setting up recommended aperture photometry values for all observation ids. 241
4.171. Scaling and subtracting the PSF. 241
4.172. Performing the aperture photometry of the residual image, computing the EEF and printing every result. 241
5.1. Opening the Spectrum Explorer from a script. 244
5.2. Opening the Spectrum Explorer forcing all spectra to be plotted 245
5.3. Adding a new spectrum to a plot. 245
5.4. Usage of SpectrumPlot variables (overplotting spectra). 246
5.5. Adding spectra using the overloaded method add() 255
5.6. Selecting point spectra and segments. 258
5.7. Selecting point spectra and segments (second variation). 258
5.8. Importing the Integrator class 258

5.9. Integrating over a set of ranges 258
5.10. Integration over a set of ranges with masking. 258
5.11. Integration over a set of ranges with masking and removing up to n levels of background 258
5.12. Fitting the fringes. 264
5.13. Smoothing the background baseline. 266
6.1. Creating an SpectralSimpleCube object. 271
6.2. Printing a cube dimensions 271
6.3. Printing all the spaxel coordinates that make up this cube. 271
6.4. Getting different coordinates from the WCS information of a cube 272
6.5. Opening the Spectrum Explorer and plotting a cube. 274
6.6. Opening the Spectrum Explorer and plotting all spectra from a cube. 274
6.7. Opening the Spectrum Explorer and plotting all spectra from a cube 276
6.8. Adding cube data to a Spectrum Explorer instance 277
6.9. Creating a plotting variable for later use with splot 284
6.10. Creating an array for selecting a particular spectrum from a cube 285
6.11. Creating a selection array with spaxel coordinates. 289
6.12. Creating a spectrum selection array. 289
6.13. Creating a deep copy of a cube. 293
6.14. Creating a wavelength/frequency grid for resampling data. 297
6.15. Using resampling tasks with cubes. 298
6.16. Declaring a Short3d array as flags for a cube. 299
6.17. Creating a mask using list slicing. 300
6.18. Creating start and end point array for integration. 302
6.19. Getting a layer from the images dataset of a cube. 302
6.20. Inspecting the contents of the images dataset of a cube. 303
6.21. Defining some Double1d arrays for range selection. 305
6.22. Converting the units of the flux maps generated by the Cube Toolbox 306
6.23. Converting the units of a map represented as a SimpleImage. 307
7.1. Creating a new instance of the Spectrum Fitter with and without a plot window. 314
7.2. Creating a Spectrum Fitter specifying the particular spectra by segment number(s). 315
7.3. Creating a Spectrum Fitter specifying the spectra by cube coordinates. 315
7.4. Creating a new instance of the Spectrum Fitter specifying both segment number and the display of the plot window. 315
7.5. Retrieving an observation whose data will be used for a Polynomial interpolation. 316
7.6. Extracting a product from the observation that will be used for Polynomial interpolation. ... 316
7.7. Script automatically (some manual changes added for the sake of clarity) generated by Spectrum Fitter (example 1). 322
7.8. Script automatically generated by Spectrum Fitter (example 1), and modified for this example. 325
7.9. Multifitting a cube using exported models. 326
7.10. Retrieving an observation from the HSA to use its data for multimodel fitting. 327
7.11. Script automatically generated by the Spectrum Fitter (example 3), slightly modified for this example. 331
7.12. Retrieving an observation from the HSA to use it in a multimodel fitting through the GUI. 333
7.13. Extracting a product to use the data for multimodel fitting through the GUI. 333
7.14. Example script fitting a spectrum with multiple models as exported by HIPE. 338
7.15. Multifitting a cube with a set of exported models. 339
7.16. Extracting the parameters from the results of the multifitting. 339
7.17. Manually setting the unit of the velocity and intensity maps 340
7.18. Worked example of a manual conversion of all data in a cube from Jy·u to W/m while creating flux maps. 340
7.19. Worked example of a manual conversion of all data in a cube from W/(m2 Hz Sr) to W/m while creating flux maps. 341
7.20. Adding a new model to an instance of the Spectrum Fitter 342
7.21. Setting weighted regions for the fitting using the setMask method. 343
7.22. Setting "binary" weights to effectively mask out ranges of the spectrum from the fit. 343

7.23. Setting a mask array that weights every point in the x-axis. 343
7.24. Setting limits for MultiFitting. 344
7.25. Setting limits for a particular model parameter in the MultiFitter. 344
7.26. Setting limits for all the parameters of a model in the MultiFitter (generic). 344
7.27. Setting limits for all the parameters of a model in the MultiFitter (with values). 344
7.28. Fixing the value of a model parameter during fitting. 345
7.29. Printing the model information, including expected parameters. 346
7.30. Running the fit after setting parameters and (optional) limits and masks. 346
7.31. Printing the fitted parameters of a model. 347
7.32. Creating a table of fitted parameters. 347
7.33. Removing one or all models from an instance of the Spectrum Fitter. 348
7.34. Mostly complete example on iterative fitting (using the residual for the next step). 349
7.35. Exporting a fitting script with the same actions performed in the GUI 350
7.36. Saving a residual data as an ASCII file. 350
7.37. Retrieving the residual in the same format as the input was. 351
7.38. Exporting the (total) model details as an ASCII file. 351
7.39. Exporting model details as an ASCII file. 351
7.40. Getting the model or total model in the same format as the input data. 351
7.41. Saving the fit parameters to an ASCII file. 352
7.42. Saving the model fit parameters as an XML file. 352
7.43. Getting the model parameters as a TableDataset. 352
7.44. Saving the model fit parameters as a SpectralLineList for use with other spectra. 353
7.45. Getting the integral of the model line. 354
7.46. Computing the integrated flux after MultiFitting a spectral cube. 354
7.47. Printing several data in MultiFit_Parms. 357
7.48. MultiFitting a SpectrumContainer with a set of previously exported models. 357
7.49. Using ComboModels with a specific relationship between fit parameters. 358
7.50. Setting the parameters of the added model. 359
7.51. Setting the parameters of a multi model fitting or a MultiFitting. 359
7.52. Creating a non-linear model for use with the Spectrum Fitter. 361
7.53. Adding a custom model, previously exported as a Jython file. 361
7.54. Selecting the fitting algorithm to use. 362
7.55. Extracting images from the ParameterCube after MultiFitting a cube 363
7.56. Retrieving the ParameterCube from the MultiFitter results. 363
7.57. Getting the peak flux image from the ParameterCube. 363
7.58. Converting the peak flux image of the ParameterCube into an image of the integrated flux, for a PACS cube. 364
7.59. Manual conversion of map units from Jy·u to W/m 365
7.60. Manual conversion of map units from W/(m2 Hz Sr) to W/m 365
7.61. Getting the cube of the total fitted model, and parameters and errors datasets. 365
7.62. Getting the Chi-squared on the command line 367
7.63. Simple polynomial fitting with error calculation using MonteCarloError. 368
7.64. Arctan fit with evidence estimation using a prior. 370
7.65. Fitting with PadeModel and evidence estimation using a prior range. 371
7.66. Fitting data with a non-linear SineModel, using a Hessian matrix as confidence; then fitting with a mixed SineModel and estimating error with MonteCarloError. 373
7.67. Fitting data with a combined model of Gaussian, Polynomial and Sine models. 378
7.68. Fitting data with a combined model of Gaussian, Polynomial and Sine models. 382
8.1. Getting the product stored in the results of a MultiFitting. 388
8.2. Creating an empty matrix that can hold the frequency values of the fitted products. 388
8.3. Filling the matrix with the fitting results data. 388
8.4. Defining constants required for the conversion. 388
8.5. Converting the whole matrix values taking advantage of the capabilities of HIPE arrays. 388
8.6. Creating a new image object to hold the velocity values. 388
8.7. Manually setting the units of the output velocity map. 388
8.8. How to convert an instance of SpireSpectrum1d. 394
8.9. How to convert an instance of HrsSpectrumDataset. 395
8.10. How to convert an instance of SpectralSimpleCube. 395

8.11. How to convert an instance of SimpleImage. 396

List of Procedures

3.1. Useful methods of the `PlotTitle` class. See Section 3.29 for the conventions used in this table. 96
3.2. Miscellaneous setters of the `LayerXY` class. See Section 3.29 for the conventions used in this table. 98
3.3. Other methods of the `LayerXY` class. See Section 3.29 for the conventions used in this table. 99
3.4. Useful methods of the `Axis` class. See Section 3.29 for the conventions used in this table. 105
3.5. Useful methods of the `AxisTitle` class. See Section 3.29 for the conventions used in this table. 106
3.6. Some methods of the `AxisTick` class. See Section 3.29 for the conventions used in this table. 107
3.7. Some methods of the `AxisTickLabel` class. See Section 3.29 for the conventions used in this table. 108
3.8. Axis-related methods of the `LayerXY` class. See Section 3.29 for the conventions used in this table. 109
3.9. Methods of the `Annotation` class. See Section 3.29 for the conventions used in this table. 111
3.10. Methods of the `PlotXY` class for handling annotations. See Section 3.29 for the conventions used in this table. 112
3.11. Methods for handling error bars in layers. See Section 3.29 for the conventions used in this table. 118
3.12. Common methods to customise colours, fonts and visibility. See Section 3.29 for the conventions used in this table. 127
4.1. Available methods for `profilePixel` and `profileSky` 193
4.2. Available methods for the output of histogram tasks. 199
4.3. Available methods for the output of the `circleHistogram` task. 200
4.4. Available methods for the output of the `ellipseHistogram` task. 201
4.5. Available methods for the output of the `rectangleHistogram` tasks. 201
4.6. Available methods for the output of the `polygonHistogram` task. 202
4.7. Available methods for the output of the `annularSkyAperturePhotometry` task. 215
4.8. Available methods for the output of the `rectangularSkyAperturePhotometry` task. 219

Preface

This document describes all the data analysis and visualization tools available in HIPE:

- **Data input/output tools.** Chapter 1
- **ASCII table data import/export tools.** Chapter 2
- **Plotting tools.** Chapter 3
- **Image analysis tools.** Chapter 4
- **Spectral analysis tools.** Chapter 5
- **Spectral analysis tools for cubes.** Chapter 6
- **Spectral fitting tools.** Chapter 7

If you are interested in knowing more about HIPE and its features, see the *HIPE Owner's Guide*.

If you are interested in knowing more about the scripting language and data types used by HIPE, see the *Scripting Guide*.

Note

Since HIPE is a multi-platform software, screenshots in this manual come from different operating systems. Do not worry if the look and feel on your system is different from what you see in this manual: all the relevant features are system-independent.

1. Conventions used in this manual

Interactive code examples. These are code examples where you are supposed to issue commands in the *Console* view of HIPE. These examples never show the `HIPE>` prompt that appears in the *Console* view. The output by HIPE is preceded by a hash mark `#`. The hash mark does not appear when you try the example in HIPE.

```
print "Hello"
# Hello
```

Example 1. Printing "hello" to the console.

Variable names. The names of variables in code examples are shown in a different typeface than the rest of the code, to indicate that they are not fixed keywords but can be changed. In the following example, you can use a different name instead of *myObs*, while you must type `getObservation` and `obsid` exactly as shown:

```
myObs = getObservation(obsid=1342183046)
```

Example 2. How to retrieve an observation given an observation ID.

Java-style and Jython-style method calling. With most objects you interact with via the command line in HIPE, you can get and set values via methods whose names begin by `get` and `set`:

```
myPlot.setTitleText("Plot title")
print myPlot.getTitleText()
# Plot title
```

Example 3. Setting a title plot.

Jython offers a simplified syntax for these cases:

```
myPlot.titleText = "Plot title"
print myPlot.titleText
# Plot title
```

Example 4. Setting properties using Jython syntax.

This manual uses the simplified syntax, but you are free to use either style in your scripts. Note that the simplified syntax exists only for `get` and `set` methods.

Chapter 1. Data input/output

This chapter and the next describe how to import data into HIPE from a variety of sources and how to export data from HIPE to a variety of destinations.

There are four main topics related to data input/output in HIPE. These correspond to the four icons you see when clicking on *Access Data* from the *Welcome* HIPE view:

Data Access

The Data Access perspective provides the means to query and download products from the Herschel Science Archive into HIPE, as well as export them for external use.

Import FITS Files

FITS files may be loaded into the session directly, so both its contents and header (meta-data) can be accessed. You can read FITS files generated with Herschel software as well as standard FITS files.

Product Browser

The Product Access Layer (PAL) provides the means to access and store data products from and to a variety of storage locations, be it remotely (and cached) or locally.
The Product Browser allows to query and retrieve data from PAL storages.

Import ASCII Tables

Tables of data written in text files can be read and imported into the session as what is called "Table Datasets" in Herschel terminology.
The exact columns and separators that you expect for your data can be specified in the task dialog for reading the table.

- **Data Access:** this perspective provides all the tools to get your data from the Herschel Science Archive (HSA). See Section 1.4 for more information.
- **Access Data Products:** an infrastructure to help you store, query and retrieve Herschel data on your computer. See Section 1.7 for more information.
- **Import FITS files:** save your data to FITS and import external FITS files. HIPE will do its best to determine what's inside the file and act accordingly. See Section 1.16.2 for importing Herschel data and Section 1.16 for importing non-Herschel data.
- **Import text tables:** save and read data as text-only files in a variety of formats. See Chapter 2 for more information.

You can also exchange data with applications compatible with Virtual Observatory standards, such as SAOImage DS9 and Topcat. See Section 1.17 for more information.

1.1. Components of an observation

Herschel observations (more formally, *observation contexts*) contain many data products, grouped in several *contexts* . Contexts are special products that contain references to other products, allowing data to be organized in a very natural, tree-like structure. Moreover, these references can point not only to products loaded into a HIPE session, but also to products still on your local disk, or even stored in the Herschel Science Archive. This feature allows HIPE to easily keep track of all the various pieces of data associated with processing of an observation, and to load these automatically when needed, regardless of where they are stored. For more information on contexts, see the *Scripting Guide* : Section 2.8.5 in *Scripting Guide* .

The following is the structure of a typical observation, common to all instruments and all observing modes:

History: Contains the automatically generated script of actions performed on your data, a history of the tasks applied to the data, and the parameters belonging to those tasks.

Auxiliary context: All Herschel non-science spacecraft data required directly or indirectly in the processing and analysis of the scientific data.

Browse image product: A thumbnail image associated with an observation. This is the thumbnail you see when you browse observations in the Herschel Science Archive, or when you open an observation with the Observation Viewer in HIPE.

Browse product: A representative product from an observation, for example an image or a spectrum. Useful for quick browsing and inspection of observations.

Calibration context: The parameters that characterise the behaviour of the satellite and the instruments. Used for reprocessing data.

Level-0 context: Raw data, minimally manipulated.

Level-0.5 context: Data processed to an intermediate point adequate for inspection.

Level-1 context: Detector readouts calibrated and converted to physical units, in principle instrument and observatory independent.

Level-2 context: Scientific analysis can be performed. These data products are in theory at a publishable quality level and should be suitable for Virtual Observatory access. However, please consult the TWiki pages set up by HIFI, PACS and SPIRE for more information on the quality of these products.

Level-2.5 context:

- For PACS, level 2.5 products are photometric maps (`SimpleImage`) produced with MadMap and the high-pass-filter pipeline, combining the scan and cross-scan AORs.
- For SPIRE, level 2.5 products are destriped maps combinining from the scan and cross-scan AORs from parallel mode.
- For HIFI, level 2.5 contents depend on the AOT. Broadly there are three kinds of HIFI AOTs: single point observations, spectral scan observations and mapping observations. The first two of these will produce one or more single spectra while the last will produce a spectral cube.

Level-3 context:

- For the PACS photometer, level 3 products are photometric maps (`SimpleImage`) produced by the high-pass-filter pipeline, combining all overlapping images from a given program.
- For the SPIRE photometer, level 3 products are combined maps of any overlapping areas from the same proposal using scan mode AOTs. The maps are mosaics of the level-2 maps that have been calibrated for extended emission and have the Planck zero-level correction applied.

Observation log context: A log of actions performed on the Products in the observation context.

Quality context: Issues flagged by the pipelines that indicate possible issues with the quality of the data or pipelining. An empty quality report indicates no problems in processing.

Trend analysis context — Products useful for tracking systematic changes in instrument response over time.

1.2. Typical workflow

This section describes the typical workflow involved with downloading, reprocessing and saving a single observation.

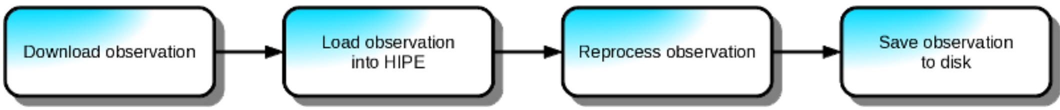


Figure 1.1. Typical workflow for downloading, reprocessing and saving an observation.

Download observation.

Download your observation from the Herschel Science Archive. You can download a tar file from the HSA User Interface, or get an observation directly into HIPE with the following command, provided you know the obsid already:

```
obs = getObservation(obsid=1342231345, useHsa=True,
 save=True)
```

Example 1.1. Getting an observation from the HSA and saving it locally.

See Section 1.4.5 for more details.

Warning

The example above downloads an entire observation and saves it to disk. This could be expensive both in time (depending on network conditions) and disk space (as there are observations that take several gigabytes).

Load observation into HIPE.

Needed only if you downloaded a tar file from the Herschel Science Archive in the previous step. Extract the contents of the tar file into any directory and load the observation into HIPE with the following command, where `/path/to/dir` is the path to the directory containing the observation:

```
myObs = getObservation(path="/path/to/dir")
```

Example 1.2. Creating a variable from an observation previously saved to disk.

See Section 1.5 for more details.

Reprocess observation.

This step depends on what kind of observation you have downloaded. Each of the Herschel instruments' data reduction guides includes a *Launch Pad* with quick-look information about how to reprocess your data.

- HIFI Launch Pad
- PACS Launch Pad for photometry and spectroscopy .
- SPIRE Launch Pad

Save observation to disk

Once you have reprocessed your observation, you can save it to disk with the following command:

```
bg("saveProduct(product=myObs, pool='myPool',
 tag='My reprocessed data')")
```

Example 1.3. How to save a product to a local pool, in the background.

Note that you will not overwrite the original observation downloaded from the Herschel Archive. See Section 1.10 for more details.

In subsequent HIPE sessions, you can open the original observation or the reprocessed versions with the following command, provided you know the obsid:

```
obs = getObservation(obsid=1342231345)
```

Example 1.4. Getting an observation from the HSA given the observation ID.

HIPE will search for the observation in the following locations (see Section 1.3 for more information on where your data are stored):

- A pool named like the obsid in your local pool directory (by default `.hcss/lstore` in your home directory).
- Any other pool in your local pool directory.
- The *MyHSA* pool.

See Section 1.7 for more information on retrieving observations from disk.

1.3. How data are stored on your disk

This section describes where HIPE keeps Herschel data on your disk. Note that you do not need to know where a certain data product or observation is stored, because you can use HIPE to search through all the Herschel data stored on your computer. See Section 1.7 for more information.

Data from the Herschel Science Archive. Observations you download from the Herschel Science Archive are stored in tar files. Once you unpack an observation and load it into HIPE, the contents are indexed and referenced by a special data pool called *MyHSA* .

If you get your data from the Herschel Science Archive via the **getObservation** command, with the *`useHsa=True`* and *`save=True`* options, the data are stored directly in the MyHSA pool. See Section 1.4.5 for more details.

You can set the directory where the MyHSA pool is stored by choosing *Edit* → *Preferences* and clicking *MyHSA* in the left-hand side list. The repository is kept by default in a `MyHSA` directory inside the `.hcss` directory.

Note that MyHSA contains only *unmodified* data downloaded from the Archive. It is like your local copy of the Herschel Science Archive with the observations of interest to you. Reprocessed data are stored elsewhere.

Warning

Do not change the contents of the `MyHSA` directory manually. Always handle your data through HIPE, as described in this chapter.

Reprocessed observations. Observations and data products you save after reprocessing are stored into *pools* , which are grouped into *storages* . A *pool* is a repository you can use to save, load and search

observations and data products. A *storage* groups one or more pools. Every pool must be *registered* to a storage. A common situation is a storage containing a single pool. If a storage contains more than one pool, only the first registered pool is accessible for writing.

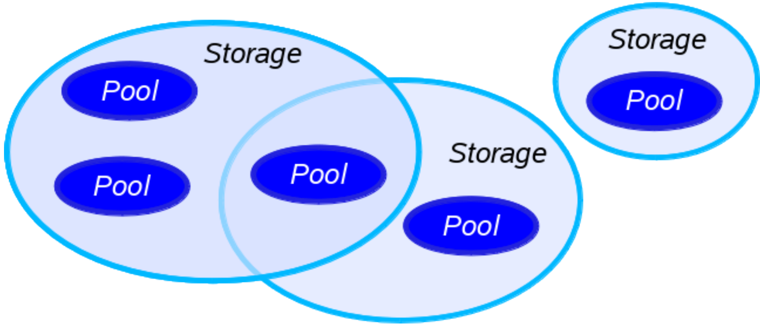


Figure 1.2. Pools and storages. All pools must be registered to a storage. A pool can be registered to more than one storage. A common situation is a storage containing a single pool.

There are many types of pools, for handling local and remote data. The *local pool* , or *local store* , is probably the one you will use most often. As the name suggests, this pool is held locally on your system, usually in a `.hcss/lstore` directory under your home directory. Although products are stored as FITS files, you should use the graphical tools provided by HIPE and described in this chapter (see for instance Section 1.7) rather than manipulating the files directly.

Note

Local pools are also called local stores for historical reasons, but they are *pools* , not *storages* .

See Section 1.3.1 for more information on how to manage pools and storages on your system.

This chapter contains all you need to know to use pools and storages in most situations. If you want to delve deeper and learn how to manage pools and storages from the command line, see the *Scripting Guide* : Chapter 7 in *Scripting Guide*.

1.3.1. Managing storages and pools

Storages and *pools* are the two tools with which you can store and retrieve data on your computer (see Section 1.3 for more details). With the *Storages & Pools* panel in the *Preferences* dialogue window (see Figure 1.3) you can create, delete and associate storages and pools.

To open this window, choose *Edit → Preferences* or press **Alt+Enter** , then go to *Data Access* > *Storages & Pools* .

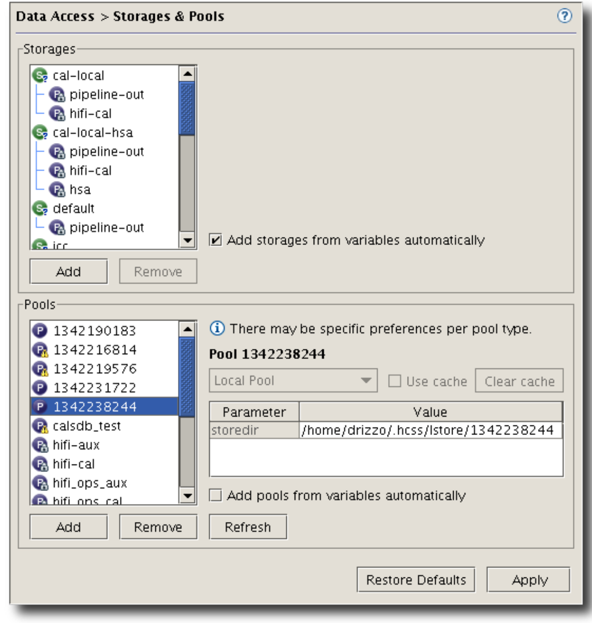


Figure 1.3. Preferences for data access

Within this window you can accomplish the following tasks:

- **Creating and deleting pools.** In the *Pools* area, click *Add* , enter the pool name and click *OK* . You can choose what kind of pool to create from the drop-down list. If you are unsure, or if you just want to store data on your local disk, leave the *Local Pool* default.

  For pools fetching remote data (Versant Database Pool and HTTP Pool) you can cache data locally by ticking the *Use cache* checkbox.

  You can set other properties of the new pool (for instance, the directory of a Local Pool) in the parameters table.

  Click *Apply* when you are done.

  To delete a pool, select it from the list and click *Remove* .

  To refresh the list of pools, click *Refresh* . This is useful if, for instance, you copy a pool into your local pool directory while HIPE is running.

- **Creating and deleting storages.** Use the *Add* and *Remove* buttons in the *Storages* area, in the same way as with pools.

- **Registering pools to storages.** Select a pool in the *Pools* area and drag it to a storage in the *Storages* area.

By ticking the *Add pools/storages from variables automatically* checkboxes, any new pools or storages you create elsewhere in HIPE (for instance, from the command line) will appear automatically in this dialogue window.

Note that this does not work the other way around: pools and storages created in this dialogue window will not appear as variables in the *Variables* view.

Warning

Do not remove pools from disk while you have variables in HIPE referring to data in those pools. Since HIPE does not always keep all the contents of a variable in memory, you may not be able to save the variable contents to disk again.

1.4. Getting observations from the Herschel Science Archive

This section outlines the steps for getting your data out of the Herschel Science Archive, and serves as a map for the rest of the subsections. Four common workflows are outlined in the following flowchart. See the text after the flowchart for links to the relevant documentation.

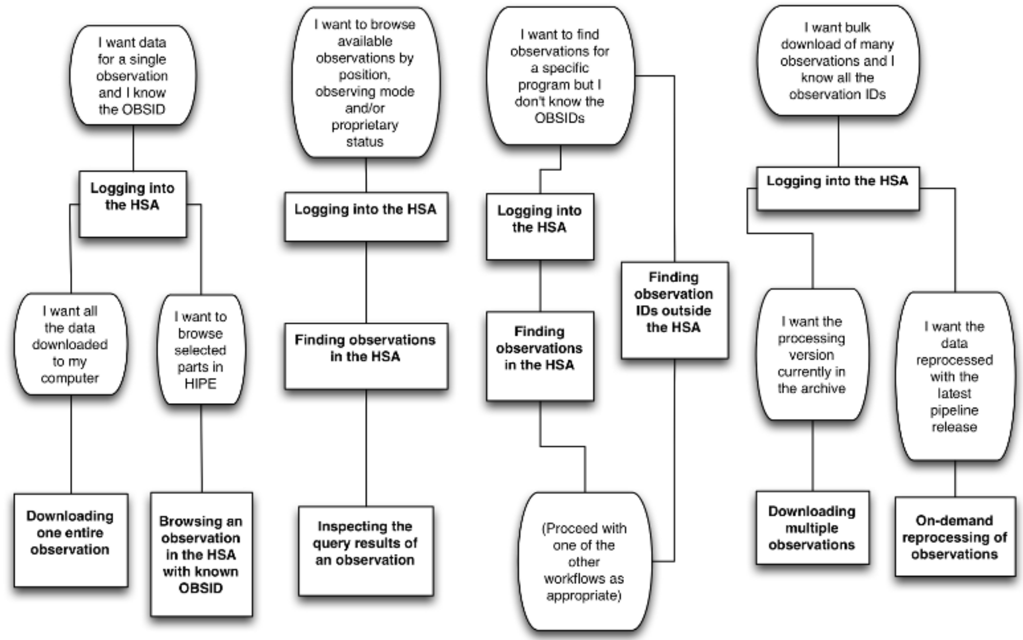


Figure 1.4. Workflows for retrieving observation data from the Herschel Science Archive

- **Workflow A: You want data for a specific observation and you know the OBSID.**

  1. Log into the HSA as shown in Section 1.4.1 .
  2. If you want all data for the observation to download to your computer, see Section 1.4.5 .
  3. If you want to browse parts of the observation in HIPE, see Section 1.4.6 .

- **Workflow B: You want to browse the archive by position, observing mode and/or proprietary status.**

  1. Log into the HSA as shown in Section 1.4.1 .
  2. Search for data in the HSA: Section 1.4.2 .
  3. See Section 1.4.3 for an explanation of the information that is returned

- **Workflow C: You want a list of observations for a specific program but you don't know the OBSIDs.**

  a. Using the HSA User Interface: First, log into the HSA as shown in Section 1.4.1 . Then, query the archive following the instructions in Section 1.4.2 .

  b. Using the online observing log: follow the instructions in Section 1.4.4 .

  With the list of OBSIDs in hand, you may proceed with any of the other workflows to download or browse the data.

- **Workflow D: You have a list of multiple OBSIDs and you want to download the data for all of them.**

  1. Log into the HSA as shown in Section 1.4.1 .
  2. Download the data, requesting either the processing version currently in the archive, or reprocessing using the current pipeline release:

     a. To download the data currently in the archive, see Section 1.4.7 .

     b. Using the online observing log: follow the instructions in Section 1.15 .

1.4.1. Logging into the HSA

To easily access data in the archive, you must first ensure that you are logged into the HSA inside HIPE. Typically this must be done only once.

To check your login status, you can inspect the bottom status bar of HIPE. If you are not logged in, the text "HSA Log-in" will be visible as shown in Figure 1.5 . Click on this part of the sidebar and enter your Herschel username and password. Tick the *Remember me* checkbox if you want HIPE to store your credentials securely—then you won't have to re-enter your login and password in future HIPE sessions.

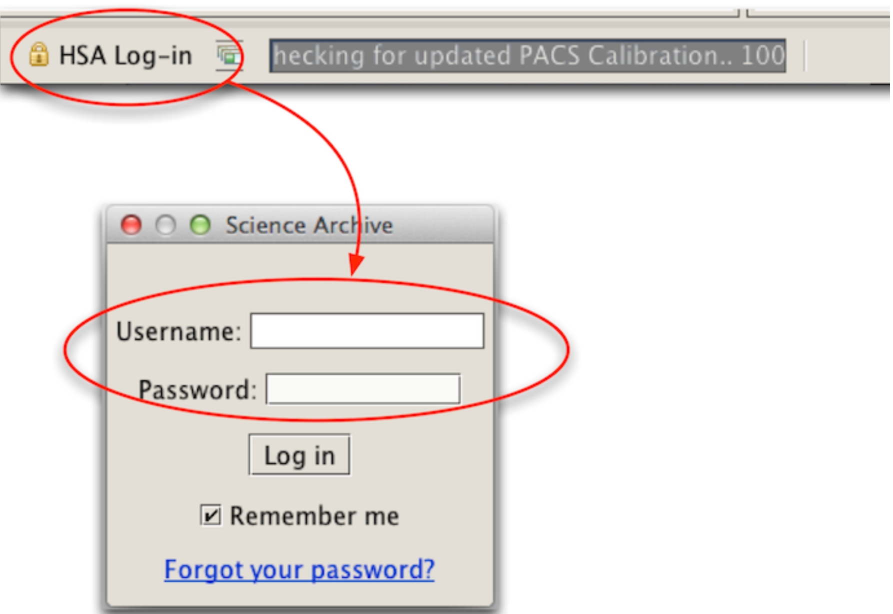


Figure 1.5. Logging in to the Herschel Science Archive

Once you have logged into the archive in HIPE, you can move to the Welcome page (see below).

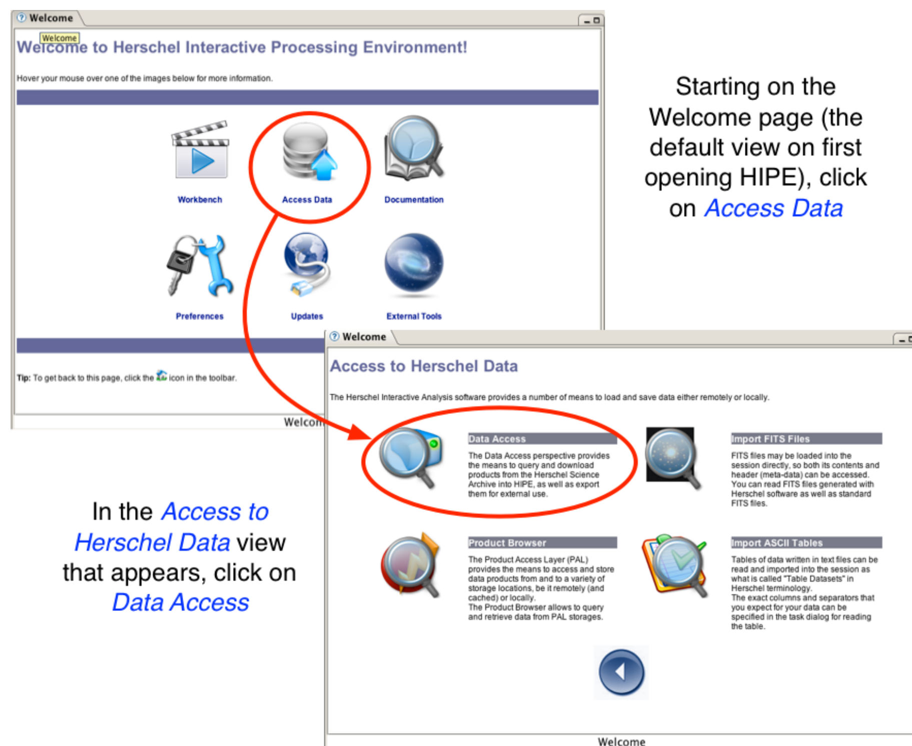


Figure 1.6. Accessing the Herschel Science Archive

Starting on the *Welcome* page (the default view on first opening HIPE), click on *Access Data*. In the *Access to Herschel Data* view that appears, click *Data Access* (see Figure 1.6). The *Data Access* perspective appears. From it you can use the *Herschel Science Archive* view to launch the archive interface.

Tip

If you cannot see the *Welcome* page, click the icon at the top right corner of the HIPE main window. Also, in the same toolbar to the left, there is an always-visible icon that instantly launches the HSA interface.

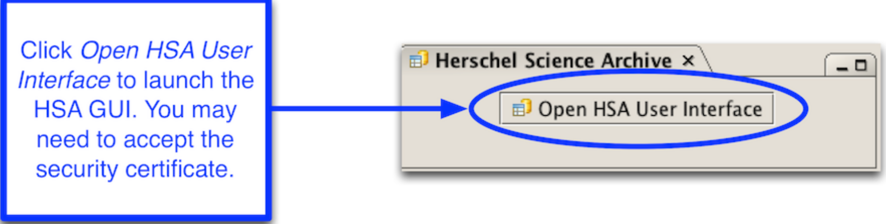


Figure 1.7. The HSA view

In the *Herschel Science Archive* view, click *Open HSA User Interface* to launch the HSA application. You may need to accept the security certificate (see Figure 1.7).

Tip

If you cannot find the *Open HSA User Interface* button in the HIPE window, go to the top-level menu item *Window → Show View → Data Access → Herschel Science Archive* .

Only authorised users can access data covered by proprietary rights. The same rule applies to the viewable quick-look products of observations, as well as to proposal-related files. They can only be viewed by the logged-in observation owner.

1.4.2. Finding observations in the HSA

Using the HUI: The Herschel Science Archive User Interface (HUI) opens on the *Search* tab: see Figure 1.8.

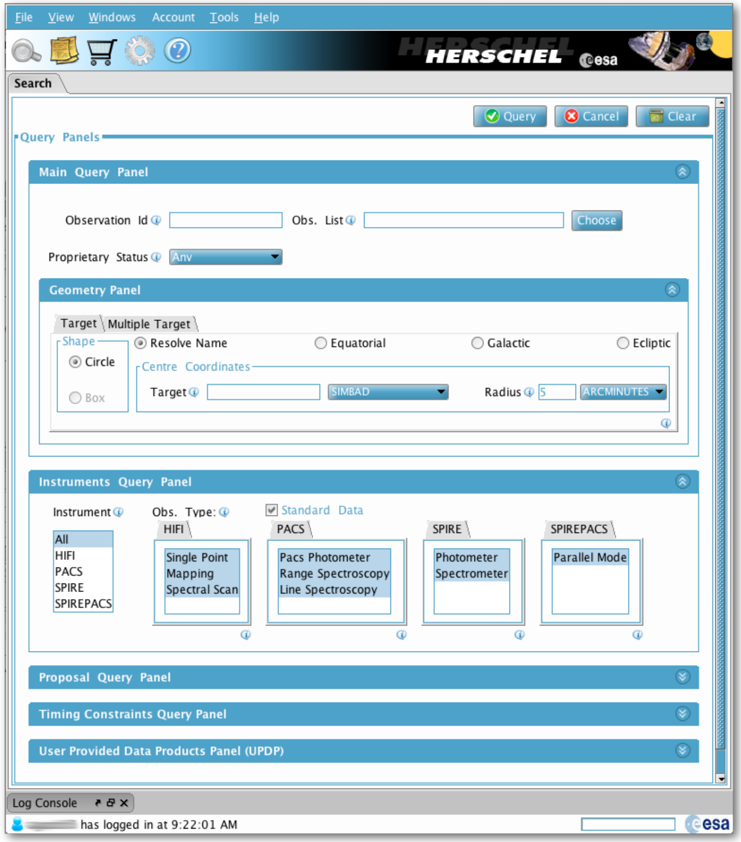


Figure 1.8. The HUI Search tab.

Most items in the HUI have small speech bubble icons next to them. Click a speech bubble icon to obtain help on the relative item.

To define your query, set all the relevant fields in the *Search* tab. You can search for observations by position, instrument mode, proprietary status, observer, or program identifier. Click *Query* to run your search. A new tab opens (see Figure 1.9) with the observations matching the query.

Click the icon to switch between a compact (Figure 1.9) and expanded (Figure 1.10) view of the results. See Section 1.4.3 for an explanation of the summary information.

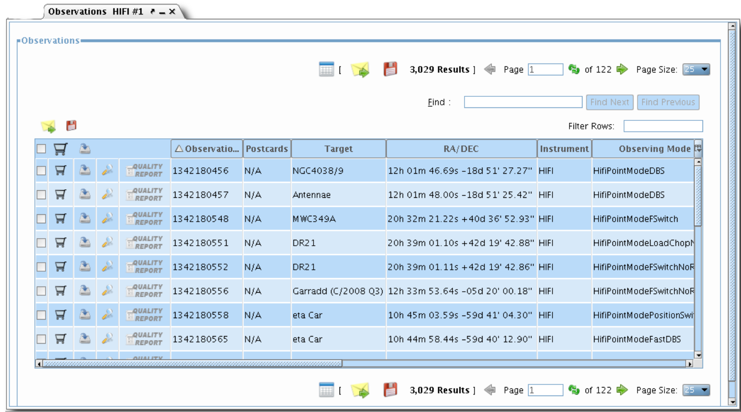


Figure 1.9. HSA query result, compact view.

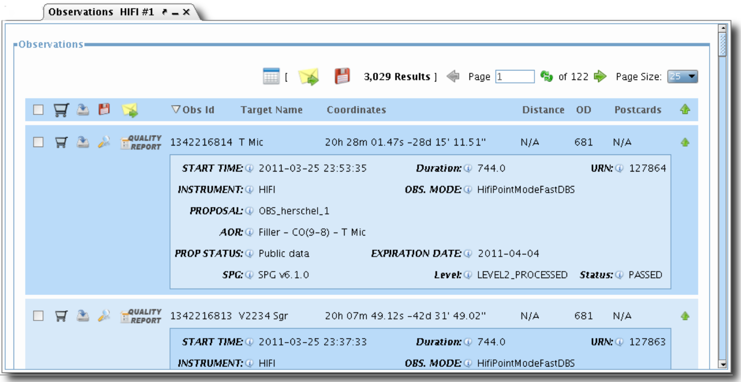


Figure 1.10. HSA query result, expanded view.

Sending query results to HIPE. You can send all or some of the query results to HIPE as a table dataset.

- To send all results, click the yellow envelope icon at the top of the query results tab. Choose *Send ALL as VOTable → HIPE* .
- To send some results:
  - Select the results you want to send by ticking the checkbox next to each entry.
  - Click the smaller yellow envelop icon just above the results table. Choose *Send selected Metadata To → HIPE* .

1.4.3. Inspecting the query results of an observation

A query made with the HUI returns a table with one row per observation, as shown in Figure 1.9 . Click the magnifying lens icon to display additional information, including an enlarged version of the observation thumbnail image, if available.

Information about the quality of science data in an observation is held in a special product called *Quality Control Report* . It includes the assessment of the execution of the observation by the spacecraft and the instruments, the evaluation of the success of the data processing, the outcome of the systematic inspection of the final product and, if required, the instrument specialist and community support astronomer analysis.

In the query results page of the HSA interface, every observation displays the status of the quality control process and a *Quality Report* button to access the Quality Report summary.

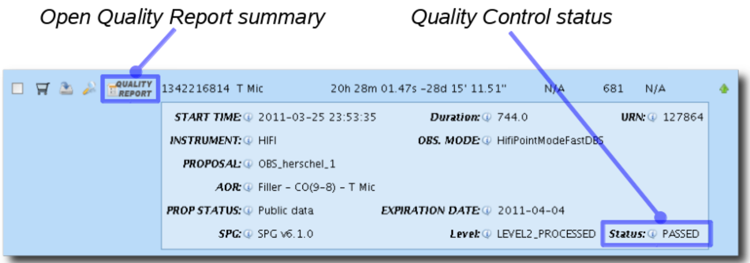


Figure 1.11. Information summary for an observation returned as a query result.

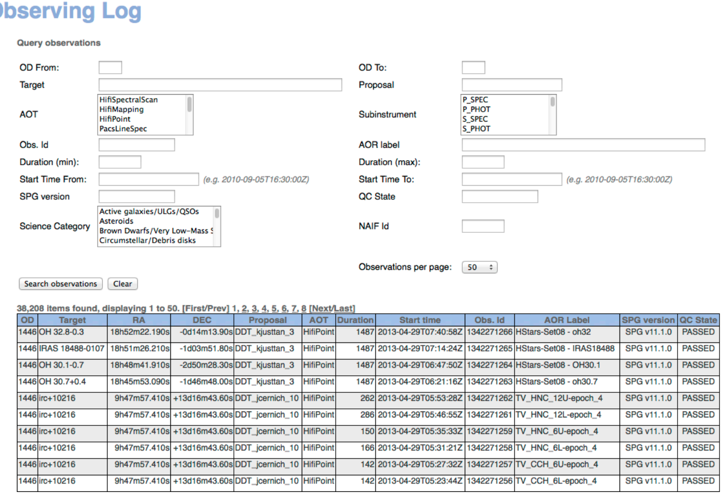
The Quality Report summary lists all the quality information relevant to you. This summary is only generated once the quality control cycle of an observation has been completed, which can take some time.

For observations still under quality control assessment, the *Status* field is empty, and the *Quality Report* button is greyed out (see for example Figure 1.9).

Once the quality control assessment of an observation is complete, the Quality Control status box displays its outcome as one of *PASSED* , *FAILED* or *PENDING* (see for example Figure 1.11). Further explanations and known caveats about the outcome are included in the *Comments* section of the Quality Report summary. The *PENDING* flag is used when the quality control cycle has finished but there are still actions to be taken. Typically this involves the reprocessing of an observation.

1.4.4. Finding observation IDs outside the HUI

The Observing Log. The most common alternative to the HUI for finding observation IDs, or program/proposal IDs, is to consult the Herschel Observing Log. By default this webpage shows the most recent downlinked observations (the last one was Obs. Id 1342271266 on the 29th of April of 2013), as shown in Figure 1.12. You can also use the form at the top of the page to query by any of the column headings: operational day (OD), target name, proposal ID, observing mode, observation ID, AOR label, AOR duration, start time, processing (SPG) version, or quality control state. You can click the column headings themselves to sort the values for that column, sorting on all items found and not just those displayed on the current page.


Observing Log

Query observations

OD From:
OD To:
Target
Proposal
AOT: HifiSpectralScan, HifiMapping, HifiPoint, PacsLineSpec
Subinstrument: P_SPEC, P_PHOT, S_SPEC, S_PHOT
Obs. Id
AOR label
Duration (min):
Duration (max):
Start Time From: (e.g. 2010-09-05T16:30:00Z)
Start Time To: (e.g. 2010-09-05T16:30:00Z)
SPG version
QC State
Science Category: Active galaxies/ULGs/QSOs, Asteroids, Brown Dwarfs/Very Low-Mass S, Circumstellar/Debris disks
NAIF Id
Observations per page: 50

Search observations | Clear

38,208 items found, displaying 1 to 50. [First/Prev] 1, 2, 3, 4, 5, 6, 7, 8 [Next/Last]

OD	Target	RA	DEC	Proposal	AOT	Duration	Start time	Obs. Id	AOR Label	SPG version	QC State
1446	OH 32.8-0.3	18h52m22.190s	-0d14m13.90s	DDT_kjusttan_3	HifiPoint	1487	2013-04-29T07:40:58Z	1342271266	HStars-Set08 - oh32	SPG v11.1.0	PASSED
1446	IRAS 18488-0107	18h51m26.210s	-1d03m51.80s	DDT_kjusttan_3	HifiPoint	1487	2013-04-29T07:14:24Z	1342271265	HStars-Set08 - IRAS18488	SPG v11.1.0	PASSED
1446	OH 30.1-0.7	18h48m41.910s	-2d50m28.30s	DDT_kjusttan_3	HifiPoint	1487	2013-04-29T06:47:50Z	1342271264	HStars-Set08 - OH30.1	SPG v11.1.0	PASSED
1446	OH 30.7+0.4	18h45m53.090s	-1d46m48.00s	DDT_kjusttan_3	HifiPoint	1487	2013-04-29T06:21:16Z	1342271263	HStars-Set08 - oh30.7	SPG v11.1.0	PASSED
1446	irc+10216	9h47m57.410s	+13d16m43.60s	DDT_jcernich_10	HifiPoint	262	2013-04-29T05:53:28Z	1342271262	TV_HNC_12U-epoch_4	SPG v11.1.0	PASSED
1446	irc+10216	9h47m57.410s	+13d16m43.60s	DDT_jcernich_10	HifiPoint	286	2013-04-29T05:46:55Z	1342271261	TV_HNC_12L-epoch_4	SPG v11.1.0	PASSED
1446	irc+10216	9h47m57.410s	+13d16m43.60s	DDT_jcernich_10	HifiPoint	150	2013-04-29T05:35:33Z	1342271259	TV_HNC_6U-epoch_4	SPG v11.1.0	PASSED
1446	irc+10216	9h47m57.410s	+13d16m43.60s	DDT_jcernich_10	HifiPoint	166	2013-04-29T05:31:21Z	1342271258	TV_HNC_6L-epoch_4	SPG v11.1.0	PASSED
1446	irc+10216	9h47m57.410s	+13d16m43.60s	DDT_jcernich_10	HifiPoint	142	2013-04-29T05:27:32Z	1342271257	TV_CCH_6U-epoch_4	SPG v11.1.0	PASSED
1446	irc+10216	9h47m57.410s	+13d16m43.60s	DDT_jcernich_10	HifiPoint	142	2013-04-29T05:23:44Z	1342271256	TV_CCH_6L-epoch_4	SPG v11.1.0	PASSED

Figure 1.12. The Observing Log webpage.

The Observing Log form does not allow querying by coordinates or proprietary status.

Using IRSA. In your web browser, navigate to the IRSA homepage at irsa.ipac.caltech.edu. Type a target name into the *Search* box and click *Search* . Scroll to the bottom of the search results page, and find the table "Archive Availability". Look for lines containing `Mission = Herschel`, such

as Herschel PACS Data. Click on the *Go* button in the last column (*Explore Data*). This takes you to a results page similar to the results tab in the HUI, where you can browse postcards and locate the observation id of interest.

Using Vizier. In your web browser, navigate to the Vizier catalog of Herschel observations . In the *Simple Target* tab, enter the source name. In the *Constraints Table* be sure to tick the *ObsId* entry under the *Show* column. Then click on *Submit* . As for the HUI and IRSA, browse the postcards to find the observation id of interest.

1.4.5. Downloading one entire observation

This section explains how to download all the data for a single observation from the Herschel Science Archive to your local disk.

GUI Method: Using the HUI

Prerequisites. You must have logged into the HSA, as described in Section 1.4.1 , and searched for one or more observations, as described in Section 1.4.2 .

In the query results tab of the HUI, click the download icon to access a *Retrieve Products* menu, from which you can select what parts of the observation to download. Choose *All* to download the entire observation. Once you have made your choice, you are prompted to save the tar file with the observation.

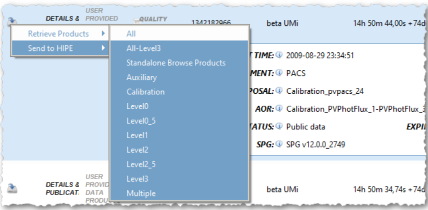


Figure 1.13. Downloading an observation from the Herschel Science Archive.

Using this method is only recommended for individual observations. To download many observations at once, see Section 1.4.7 .

Where are my data? Your data are contained in a tar file on your disk. HIPE is not yet aware of the data.

Where do I go from here? Once you have downloaded your observation from the Archive, you must load it into HIPE, as explained in Section 1.5 .

Command-line method: Using *getObservation*

Prerequisites. You must be logged into the HSA as described in Section 1.4.1 .

Use the `getObservation` command to download an observation from the Herschel Science Archive and save it to disk. You need to know the obsid of your observation:

```
myObs = getObservation(obsid=1342231345, useHsa=True, save=True)
```

Example 1.5. Saving an observation to disk from the HSA given the observation ID.

Warning

When retrieving SPIRE/PACS parallel mode observations, you must specify the `instrument` parameter with a value of either `PACS` or `SPIRE`, as shown by the following examples:

```
myObs = getObservation(obsid=1342183046, instrument='PACS',
 useHsa=True, save=True)
```

```
myObs = getObservation(obsid=1342183046, instrument='SPIRE',
 useHsa=True, save=True)
```

Where are my data? Your observation has been saved to the My HSA repository on your disk. Do not access the files directly. You can access your observation in HIPE at any time with the Product Browser, as explained in Section 1.7 , or with this command, which will look for all copies of an observation with a given obsid:

```
myObs = getObservation(obsid=1342183046)
```

Example 1.6. Getting an observation from the HSA given an observation ID.

For more information on managing your My HSA repository, see Section 1.6 .

Note

In some cases you may want to configure `getObservation` so that it doesn't look for and download observations from the HSA. There is a setting within Preferences in *Data Access > My HSA* called *Save data on-demand* that controls this behaviour. In a script, you can set it on and off using the following commands:

```
# Turn it on
MyHSAPool.getInstance().setConnection(herschel.ia.pal.pool.hsa.MyHSAConnection.ON)
```

Example 1.7. Setting on the connected status of the MyHSA pool.

```
# Turn it off
MyHSAPool.getInstance().setConnection(herschel.ia.pal.pool.hsa.MyHSAConnection.OFF
```

Example 1.8. Setting off the connected status of the MyHSA pool.

Where do I go from here? Your observation is now referenced by the `myObs` variable. You can now work on your data and then save your modified data to disk as explained in Section 1.10 .

In future HIPE sessions, you can retrieve the original observation or the modified one as explained in Section 1.7 .

1.4.6. Browsing an observation in the HSA with known OBSID

With HIPE you can browse the contents of observations stored in the HSA without having to save them first. What is sent to HIPE are just *references* to data products, not the products themselves.

GUI Method: Using the HUI

Prerequisites. You must have logged into the HSA, as described in Section 1.4.1 , and searched for one or more observations, as described in Section 1.4.2 .

In the query result tab, click the download icon to access a *Send to HIPE* menu. With this you can choose whether to browse the whole observation or just a portion of it.

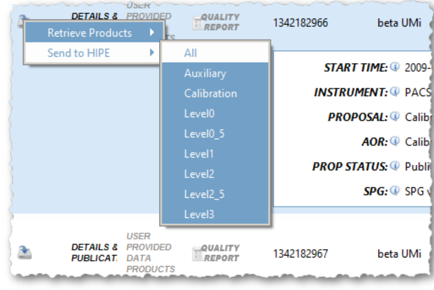


Figure 1.14. Selecting which part of an observation to browse.

Select an option and, if the connection between the two applications (HUI and HIPE) is well established, a pop-up window appears with the message *VOTable sent successfully to external application* . Data starts to load into HIPE automatically.

If the option *All* was selected, a variable called `obsid_xxxxxxxxxx` is created in HIPE, with an actual observation number. Other options are also stored in different variables as illustrated in Figure 1.15 .

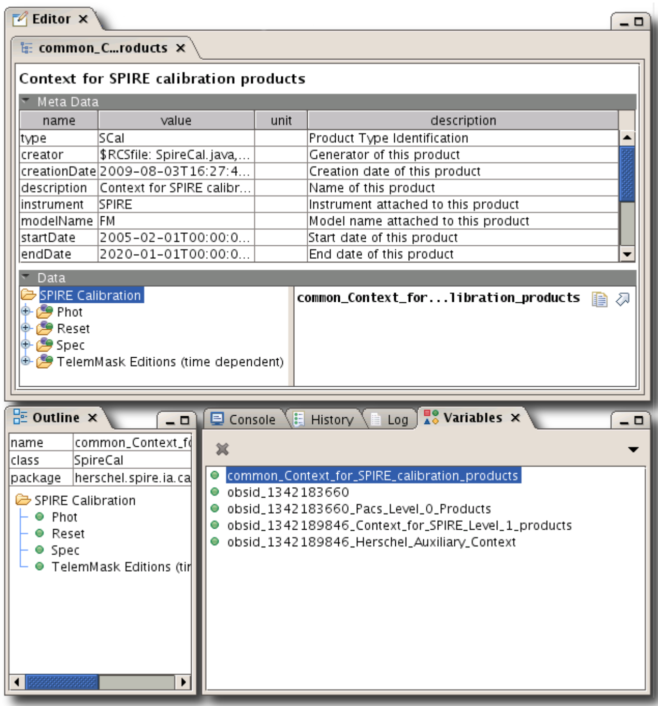


Figure 1.15. Product loaded into HIPE from the HSA.

Where are my data? Your data are not stored on your machine, but read on-demand from the HSA. Note also that for this, *the Internet connection must be kept open* throughout the session.

Where do I go from here? Once you have finished browsing the data, you can save them to disk as described in Section 1.10 .

Command-line method: Using *getObservation*

Prerequisites. You must be logged into the HSA as described in Section 1.4.1 .

Use the `getObservation` command to browse an observation from the Herschel Science Archive. You must know the OBSID of the observation:

```
myObs = getObservation(obsid = 1342231345, useHsa = True)
```

Example 1.9. Retrieving an observation from the HSA given the observation ID.

Warning

When retrieving SPIRE/PACS parallel mode observations, you must specify the *instrument* parameter with a value of either PACS or SPIRE, as shown by the following examples:

```
myObs = getObservation(obsid=1342183046, instrument='PACS',
 useHsa=True, save=True)
```

```
myObs = getObservation(obsid=1342183046, instrument='SPIRE',
 useHsa=True, save=True)
```

Double click on the `myObs` variable in the *Variables* view to open the Observation Viewer. You can also print out portions of the observation context via commands like `print myObs.level2` , but in general the Observation Viewer will be much more convenient for browsing.

Where are my data? Your data are not being accessed from storage on your machine, but are read on-demand from the HSA. Note also that for this to work, *the Internet connection must be kept open* throughout the session. If either the save=True option is passed to getObservation, or the "Save data on-demand" option is turned on as detailed in Section 1.6.1 , data are saved into your MyHSA location.

Where do I go from here? Once you have finished browsing the data, you can save them to disk as described in Section 1.10 .

1.4.6.1. Multiple versions of the same observation

There are two additional cases when, given an observation ID, there are multiple instances of the observation in the HSA or in your computer. They are identified by two metadata fields, one of which changes its meaning between these two different locations.

- `spgVersion` is always the version of HCSS that was used to generate the product as part of the Standard Product Generation (SPG) pipeline. Its format is `M.m.p` where `M` is major version, `m` is minor version and `p` is patch version like in `11.1.0`.
- `version` can be two things, identified by an integer number, depending on the location (even if, conceptually, they are the same thing):
  - In the HSA: it is the number of bulk reprocessings performed on the data.
  - In your computer: it is the number of changes followed by a save to the same pool that you have made to the data.

Using the command line, you can browse all versions in a very powerful way:

```
obs = getObservation(obsid = 1342197792, useHsa = True, allVersions = True)
```

Example 1.10. Browse all the versions of an observation (part 1).

If you haven't downloaded this observation before, all the versions displayed are the ones available in the HSA, as in this example. The output of the command can be seen below:

```
More than one observation found. Please, refine your query

The following parameters:
urn = null
obsid = 1342197792
```

```
od = -1
instrument = null
tag = null
spgVersion = null
creationDate = null
version = -1

Produces more than one result:
Pool id | Obsid | Tag | Version | Track id | Urn | Creator
hsa | 1342197792 |  | 8 | OBS_12508@wn25.n1grid.lan_1_16441247557132381 |
 urn:hsa:herschel.ia.obs.ObservationContext:429666 | SPG v11.1.0 |
hsa | 1342197792 |  | 9 | OBS_12508@wn25.n1grid.lan_1_16441247557132381 |
 urn:hsa:herschel.ia.obs.ObservationContext:526876 | SPG v12.1.0 |
hsa | 1342197792 | OBS:P:0050005020 | 10 |
 OBS_12508@wn25.n1grid.lan_1_16441247557132381 |
 urn:hsa:herschel.ia.obs.ObservationContext:620498 | SPG v13.0.0 |

To retrieve the observation please provide a version, urn, or a SPG version number
 like '11.1.0'
```

This output text advises you that `spgVersion` is a substring of the metadata field `creator` present in the archive. This way, `SPG v11.1.0` becomes `11.1.0` for the purposes of retrieving the observation. If you now type:

```
obs = getObservation(obsid = 1342197792, useHsa = True, spgVersion="11.1.0")
```

Example 1.11. Browse all the versions of an observation (part 2a).

Or:

```
obs = getObservation(obsid = 1342197792, useHsa = True, version = 8)
```

Example 1.12. Browse all the versions of an observation (part 2b).

Which in this case is the same, you will download the desired observation version to disk.

Note

When downloading observations to your computer and, as stated above, the `version` number is reset to zero (0) to represent, from that moment on, user changes, so if you want to retrieve a specific SPG version from a local pool, you are required to use `spgVersion`. To load from disk the previously downloaded observation, you should type:

```
obs = getObservation(obsid = 1342197792, useHsa = False,
 spgVersion="11.1.0")
```

Example 1.13. Browse all the versions of an observation (part 3).

1.4.7. Downloading multiple observations

GUI Method: Using the HUI

Prerequisites. You must have logged into the HSA, as described in Section 1.4.1 , and searched for one or more observations, as described in Section 1.4.2 .

You can download multiple observations at once via FTP with the *Shopping Basket* .

To select data for retrieval, tick the checkbox to the left of each record of the observations list, and then click the shopping basket icon at the top of the table. Alternatively, add one observation at a time by clicking the shopping basket icon in the observation's table row. A *Shopping Basket Overview* window appears with the observations you have added.

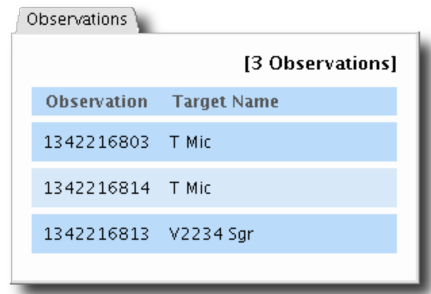


Figure 1.16. The shopping basket of data to retrieve from the HSA

Once you have added all the observations you wanted to the shopping basket, click the large shopping basket icon on the HUI toolbar to open the *Shopping Basket* tab.

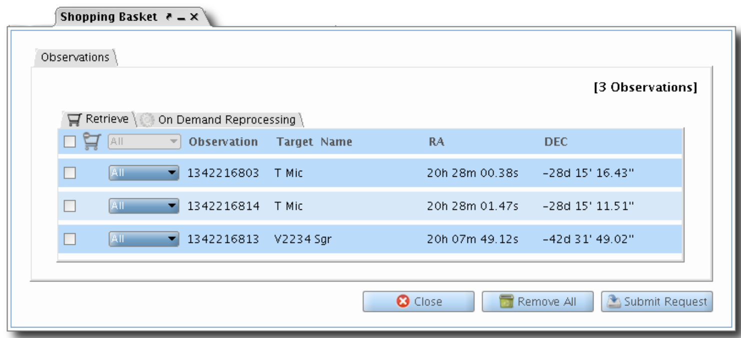


Figure 1.17. The shopping basket of data to retrieve from the HSA

You can remove one or more observations from the shopping basket by ticking the checkboxes on the corresponding rows and then clicking the *Remove All Selected* icon in the top row of the table. You can also add more observations at any time by going back to the query results tab.

Once you are happy with the contents of the shopping basket, click the *Submit Request* button. After a short while you receive an email message with the FTP location where your data are stored. You can then download the data as a tar file.

Where are my data? Your data are contained in a tar file on your disk. HIPE is not yet aware of the data.

Where do I go from here? Once you have downloaded your observations from the archive, you must load them into HIPE, as explained in Section 1.5 .

Command-line method: Using *getObservation*

Prerequisites. You must be logged into the HSA as described in Section 1.4.1 .

You can use the `getObservation` command within a loop to download multiple observations from the Herschel Science Archive and save them to disk. You need to know the obsid of all the observations:

```
myObsids = [1342242595, 1342239349]

for i in myObsids:
  print "Downloading obsid " + `i`
  myObs = getObservation(obsid=i, useHsa=True, save=True)
```

Example 1.14. Downloading multiple observations from an array of obs ids.

It is also possible to do this with tarred observations retrieved using the HAIO

```
from java.nio.file import Files
import urllib

# Create temporary directory for TAR.GZ download
tempdir = Files.createTempDirectory("hipe")
observationIds = [1342231052, 1342231345]

# Output dictionary of observations (empty for now)
observations = {}

# Do the loop
for obs in observationIds:
 # Download a TAR.GZ observation using the HAIO and builtin urllib
 strObsPath = str(tempdir)+"/obs.tar.gz"

 urllib.urlretrieve("http://archives.esac.esa.int/hsa/aio/jsp/product.jsp?"+\
   "OBSID="+str(obs)+"&PRODUCT_LEVEL=Auxiliary&COMPRESSION=TARGZ&"+\
   "PROTOCOL=HTTP", strObsPath)
 # Decompress the file
 outputDir = str(tempdir)+"/obs"+str(obs)
 decompress(strObsPath, outputDir)

 # One more step as there is another directory with a random name within outputDir
 randomDir = os.listdir(outputDir)[0]
 finalPath = outputDir+"/"+randomDir

 # Open the tarred observation from disk
 observations[obs] = getObservation(path = finalPath)
```

Example 1.15. Retrieving several observations from the HSA as tar.gz and opening them in HIPE.

Warning

When retrieving SPIRE/PACS parallel mode observations, you must specify the *instrument* parameter with a value of either PACS or SPIRE, as shown by the following examples:

```
myObs = getObservation(obsid=1342183046, instrument='PACS',
 useHsa=True, save=True)
```

```
myObs = getObservation(obsid=1342183046, instrument='SPIRE',
 useHsa=True, save=True)
```

Where are my data? Your observations have been saved to the My HSA repository on your disk. Do not access the files directly. You can access your observations in HIPE at any time with the Product Browser, as explained in Section 1.7 , or with this command, which will look for all copies of an observation with a given obsid:

```
myObs = getObservation(obsid=1342183046)
```

Example 1.16. Retrieve an observation from the HSA given the observation ID.

For more information on managing your My HSA repository, see Section 1.6 .

Where do I go from here? The last observation downloaded is now referenced by the myObs variable. You can find and load the other observations via the Product Browser as explained in Section 1.7 . Then you can work on your data and save the modified data to disk as explained in Section 1.10 .

In future HIPE sessions, you can retrieve the original observations or the modified ones as explained in Section 1.7 .

1.5. Loading observations downloaded from the HSA into HIPE

Prerequisites. You have already downloaded one or more observations as tar files from the Herschel Science Archive as described in Section 1.4.5 and Section 1.4.7 . Alternatively, you have received on-demand reprocessed data as explained in Section 1.15 .

You need to load the observations into HIPE before you can view and reprocess the data. You need to do it only once for each observation you download from the Herschel Science Archive.

Note

If you downloaded your observation via the command line using the **getObservation** command with the *save=True* parameter, you do not need to follow these steps. Your observation has already been loaded into HIPE. See Section 1.7 for information on how to find your observation in HIPE.

GUI method: using the Navigator view

1. Uncompress the tar file you downloaded from the Herschel Science Archive. One directory is created for each observation contained in the tar file.

Warning

Some compression applications (especially in non-UNIX operating systems, i.e. Windows) need to convert the line endings of the text files inside the TAR archive from the UNIX standard (using only `LF` characters). For a common utility that has problems with this, please see this Known Issue describing the issue (and a workaround). You can also use other software such as 7-Zip, or the `decompress` task in HIPE: Section 1.104 in *HCSS User's Reference Manual*

2. In the *Navigator* view of HIPE, open the directory that was created when you uncompressed the tar file. You will see an item with a Saturn icon, as shown in the following image.

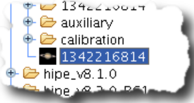


Figure 1.18. A Herschel observation ready to be loaded into HIPE.

3. Double click on the item with the Saturn icon. HIPE loads and opens the observation.

Where are my data? Your data are still in the directory created from uncompressing the tar file. The data has been indexed in the My HSA repository, but HIPE has not copied the data elsewhere, which is why you must not delete this directory. You must also not modify the data files directly. For more information on managing your downloads from the Herschel Science Archive, including how to delete unwanted data, see Section 1.6 .

Where do I go from here? You can now start working on your observation. Once you are finished, to save your modified data see Section 1.10 . To load the original or modified observation in future HIPE sessions, see Section 1.7 .

Command-line method: using getObservation

1. Uncompress the tar file you downloaded from the Herschel Science Archive. One directory is created for each observation contained in the tar file.

2. Issue the following command in the *Console* view of HIPE, where *`/path/to/dir`* is the path to the directory containing the observation (for example, `/home/joe/joe141729940`):

```
myObs = getObservation(path="/path/to/dir")
```

Example 1.17. Load an observation from disk into a new variable.

For a complete example downloading the observation in `tar.gz` format from the Herschel Science Archive using Herschel Archive InterOperability subsystem (HAIO), see below:

```
from java.nio.file import Files

# Create temporary directory for TAR.GZ download
tempdir = Files.createTempDirectory("hipe")
obs = 1342231052
# Download a TAR.GZ observation using the HAIO and builtin urllib
strObsPath = str(tempdir)+"/obs.tar.gz"
import urllib
urllib.urlretrieve("http://archives.esac.esa.int/hsa/aio/jsp/product.jsp?"+\
  "OBSID=1342231052&PRODUCT_LEVEL=Auxiliary&COMPRESSION=TARGZ&"+\
  "PROTOCOL=HTTP", strObsPath)
# Decompress the file
outputDir = str(tempdir)+"/obs"
decompress(strObsPath, outputDir)

# One more step as there is another directory with a random name within
 outputDir
randomDir = os.listdir(outputDir)[0]
finalPath = outputDir+"/"+randomDir

# Open the tarred observation from disk
obs = getObservation(path = finalPath)
```

Example 1.18. Retrieving an observation from the HSA as a tar.gz and opening it in HIPE.

HIPE loads the observation, assigning it to variable *myObs* .

If the directory contains multiple observations, you must specify the obsid of the observation you want to load:

```
myObs = getObservation(path="/path/to/dir", obsid=1342183046)
```

Example 1.19. Load an observation from disk, specifying both path and observation ID.

Where are my data? Your data are still in the directory created from uncompressing the tar file. The data has been indexed in the My HSA repository, but HIPE has not copied the data elsewhere, which is why you must not delete this directory. You must also not modify the data files directly. For more information on managing your downloads from the Herschel Science Archive, including how to delete unwanted data, see Section 1.6 .

Where do I go from here? You can now start working on your observation. Once you are finished, to save your modified data see Section 1.10 . To load the original or modified observation in future HIPE sessions, see Section 1.7 .

1.6. Managing your HSA downloads

You can manage your downloads from the Herschel Science Archive via the My HSA preferences dialogue window. To open it, choose *Edit → Preferences* and click *My HSA* in the left-hand side list.

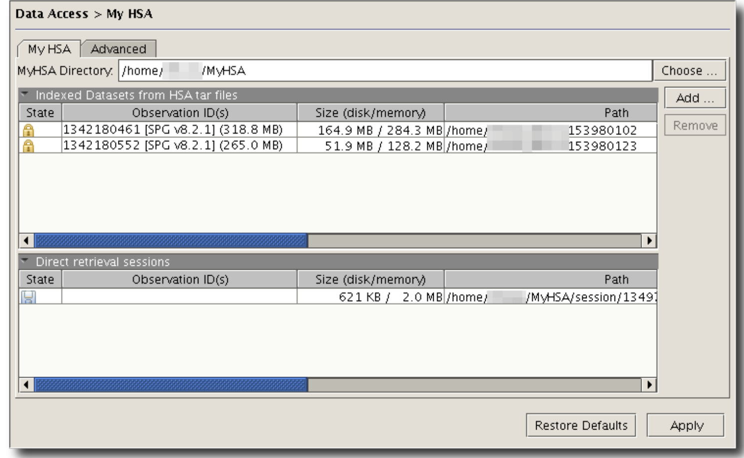


Figure 1.19. The My HSA preferences dialogue window.

The two tables. The *Indexed Datasets* table lists observations you have loaded directly from the HSA User Interface (HUI) in tar form, as explained in Section 1.5 . The *Direct retrieval sessions* table lists observations you have browsed or downloaded using the Product Browser, as explained in Section 1.7 , or the `getObservation` command, as explained in Section 1.4.5 and Section 1.4.6 .

In the *My HSA* dialogue window you can perform the following tasks:

- **Change the location where configuration information and downloads from the Herschel Science Archive are stored. .** Write a new path in the *Directory* text field, or click *Choose* and navigate to the new directory.
- **Load downloaded observations into HIPE.** Click *Add* and navigate to a directory created from unpacking a tar file downloaded from the HUI. Select the `.xml` file in the directory and click *Open* . HIPE loads the observation. A new row appears in the *Indexed Datasets* table.

  This way of loading observations is equivalent to the one described in Section 1.5 .
- **Remove downloaded observations from disk.** Select one or more rows from the *Indexed Datasets* and *Direct retrieval sessions* tables and click *Remove* . A dialogue window appears asking for confirmation. The default is to delete the *configuration* , so that HIPE is no longer aware of the data, but the data themselves are kept on disk. You can tick the *Remove FTP products as well* checkbox to delete the data.

  Note that removing one observation could affect the integrity of other observations. See the warning at the end of this section for more information.

The state icons. The state of the data in each row of the tables is indicated by the following icons:

- — The data have been fully downloaded and indexed. No new data are being added.
- — New data are being added. For example, you are still browsing the corresponding observation in HIPE.
- — HIPE has detected an error. Usually this means that you have moved or removed the observation from the original location when you unpacked the tar file provided by the HUI. Note that this does not mean that the MyHSA database got corrupted. You just need to import the data again and remove the old link using the *Remove* button. Alternatively, move the observation to its original location to restore the appropriate links.

The table columns. Some of the columns of the two tables are described below:

- **Path.** Gives the directory where the tar downloaded from the HUI was unpacked, or the location (session) inside MyHSA where a direct download of data from HIPE was stored.
- **Size.** Gives the actual size of your observation, both the physical size on disk and the size that it occupies in memory. Note that the values can be lower than the size given in the observation ID column. This can be due to two reasons: either you have downloaded the observation only partially, or some of the files were not indexed or downloaded again into MyHSA because they were already in a different MyHSA location. The latter case can occur, for instance, with auxiliary files of two observations taken in the same OD, or with calibration files of two observations performed with the same instrument. In this way MyHSA avoids duplication of files. *This means that observations are linked among them.* Such links are shown in the two columns *Requires* and *Is required by* .
- **Requires.** Gives the location in MyHSA where part of the data reside, when different from the actual location of the observation.
- **Is required by.** Means that some files of this observation are required by a different one.

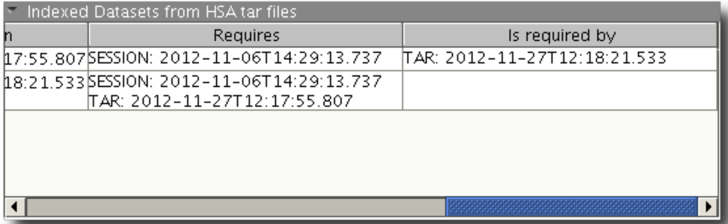


Figure 1.20. A detail of the *Indexed Datasets* table showing the *Requires* and *Is required by* columns.

Warning

The *Requires* and *Is required by* columns show that some observations depend on each other inside MyHSA. Therefore, the removal of one observation can impact one or more different observations. Again, this does not mean that the MyHSA database got corrupted. You just need to import the removed observation again and remove the old link using the *Remove* button, or to move the observation to its original location, to establish the appropriate links again. Then click the *Update Links* button in the *Advanced* tab as explained in Section 1.6.1 .

Note

Data indexed or downloaded with a version of HIPE before 10.0 will not show real values for these columns. Recovering this information from previously downloaded observations would impact HIPE performance.

Warning

If you find large amounts of data in the *Direct retrieval sessions* that you don't recall downloading, it may be that you have set the "Save data on-demand" option. See Section 1.6.1 . Alternatively, you may have been using `getObservation` with `useHsa=True` and `save=True` (though this is less likely as `save=False` is the default).

1.6.1. Advanced configuration

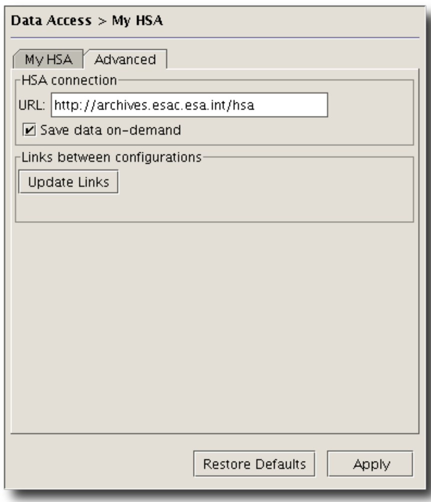


Figure 1.21. The *Advanced* tab of the My HSA preferences dialogue window.

In the *Advanced* tab of the *My HSA* dialogue window you can perform the following tasks:

- **Update links between configurations.** Click the *Update Links* button to update the information in the *Requires* and *Is required by* columns in the tables of the *My HSA* tab. It is useful to click this button each time you remove data from the main *My HSA* tab, since HIPE does not update the links automatically for performance reasons.
- **Download online data automatically when you browse them. .** Tick the *Save data on-demand* to enable automatic data download (the default is "unticked"). This applies to two cases:
  - You query the Herschel Science Archive via the Product Browser using the *HSA* data source (see Section 1.7 for more details), or `getObservation` with `useHsa=True` (see Section 1.4.6). When you browse any of the results, the data you view are downloaded automatically.
  - You download a partial observation from the Herschel Science Archive by selecting something other than *Retrieve Products* → *All* in the HUI (see Section 1.4.5 for more details). If you then load the partial observation into HIPE and try to view missing data, these are downloaded automatically. Note that automatic download works only if HIPE is connected to the Herschel Science Archive through the Product Browser, by selecting the *On-line* radio button in the *Data Source* panel. See Section 1.7 for more details.

Warning

Do not change the address in the *URL* text field. This is the address of the Herschel Science Archive service.

1.7. Retrieving an observation from disk

Prerequisites. You must have downloaded one or more observations from the Herschel Science Archive (see Section 1.4) and loaded them into HIPE (see Section 1.5). Optionally you may have already reprocessed your data and saved the results to disk (see Section 1.10).

GUI Method: Using the Product Browser

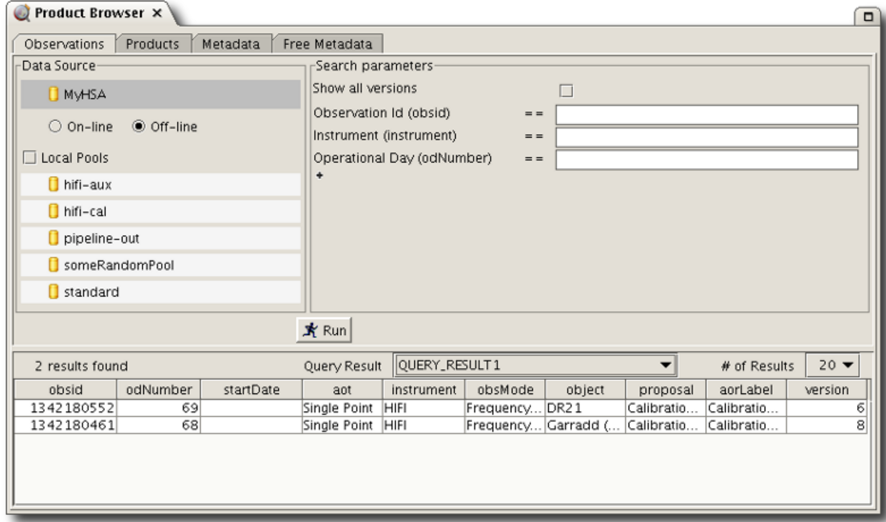


Figure 1.22. The Product Browser.

Follow these steps to find observations on your disk:

1. Open the Product Browser perspective by clicking the icon on the HIPE toolbar, or by choosing *Window → Show Perspectives → Product Browser perspective* .

2. Select the data sources you want to query in the *Data Source* panel (see Figure 1.22). Those of interest to you are the following:

   - *MyHSA, On-line* : the Herschel Science Archive. This is an alternative way of browsing the HSA.
   - *MyHSA, Off-line* : the observations you have downloaded from the Herschel Science Archive and loaded into HIPE.
   - *Local Pools* : the data pools on your disk where you save Herschel data after you have reprocessed them.

3. Set your query parameters in the *Observations* tab. Typically you will want to indicate an observation ID. You can add more query parameters by clicking on the small plus icon at the bottom of the list in the *Search parameters* panel. Then you can select a metadata value, an operator (such as ==) and a value.

4. Click *Run* to execute the query.

   - **Result.** Your result is shown in the table below the query parameters area and stored in a variable called `QUERY_RESULT` , for the first query, or `QUERY_RESULT_x` for subsequent queries, where `x` is a number. If no result is found then no `QUERY_RESULT` is produced.

     For expert users: the result variable may be used as argument in a `ProductStorage.select()` statement.

     ```
     results=storage.select(MetaQuery(...),QUERY_RESULT_1)
     ```

     Example 1.20. Perform a query on a local store using the result of another query.

   - **Versions.** Versions are created whenever saved data are modified and stored again. If you ticked the *Show all versions* checkbox, you will see all versions of your data, instead of just the latest one.

   - **Tags.** Tags are keywords or phrases you can associate to a product, to better describe and remember its contents. For example, you could assign to a product the tag "to be completed" to

remember that you have not finished processing it. When defining tags, you are free to use the keywords and phrases that work best for you.

If the data was saved with a tag assigned to it, you will see the tag displayed in the *tag* column. If you do not see a *tag* column, right click on any column header and choose *Select layout* → *Standard Table Layout* . See Section 1.10 for how to add tags when saving observations to disk.

5. Review the results.

   - Select a row to further inspect it in the *Outline* view. This also creates a variable called `selected`.
   - Double click a row to create a variable in the *Variables* view. This is not the same as the `selected` variable, whose contents change according to the selected row.
   - Right click on the table to do the following on the selected rows:
     - Create variables in the *Variables* view (same as double click).
     - Remove the data from the pool. You can also remove an entire observation, including all child products. Removing does not work for data in the *MyHSA* area.
     - Export the data to FITS.

   See Section 1.8 for how to customise the layout of the result area.

6. Inspect selected results in the *Outline* view.

   - Double click an item to open it with the default viewer. Be aware that HIPE may have to load the item first, which could be a time and memory consuming operation for large products.
   - Right click on an item and choose *Open With* to open it with a viewer of your choice in the *Editor* view.

Tip

The Product Browser does not find data I know to be present. Check the following:

- Make sure that the location of your data is selected in the *Data Source* panel. If in doubt, select more data locations.
- Make sure you are searching for the right kind of data. For example, you may be in the *Observations* tab, thus searching for entire observations when instead you want to look for a data product inside an observation. If so, switch to the *Products* tab and make sure that the *Product type* field has the correct value. If in doubt, choose `herschel.ia.dataset.Product` to search for any data product.

Command-line method: getObservation

With `getObservation` you must keep track of, and provide to the task, the OBSID, and possibly also the name and location of the pool where you saved your data, if these are not the defaults.

- If your data were saved with the default name (the OBSID) and location (your `lstore` directory), they can be loaded by specifying the OBSID:

```
myObs = getObservation(obsid=1342231345)
```

Example 1.21. Retrieve an observation given the observation ID.

HIPE will search for the observation in the following locations (see Section 1.3 for more information on where your data are stored):

  - A pool named like the obsid in your local pool directory (by default `.hcss/lstore` in your home directory).
  - Any other pool in your local pool directory.
  - The *MyHSA* pool.
- If your data were saved in a non-default pool location, this must be specified:

```
myObs = getObservation(obsid=1342183046, poolLocation=myDirectory)
```

Example 1.22. Load an observation from disk specifying both the observation ID and the pool directory.

HIPE will look in the non-default location for a pool with the same name as the obsid.

- If your data were saved in a non-default pool location and with a non-default pool name (different from the obsid), both must be specified:

```
myObs = getObservation(obsid=1342183046, poolName=myPool,
 poolLocation=myDirectory)
```

Example 1.23. Load an observation from disk specifying the observation ID, the local pool name and the pool location.

HIPE will look in the non-default location for a pool with the specified name.

Here are some further examples:

```
# Most useful task parameters
myObs = getObservation(obsid=<int|string> [,poolName=<string>]
      [,poolLocation=<string>] [,useHsa=<boolean>] [,save=<boolean>]
      [,tag=<string>])

# Most common uses:

# Get your data from [HOME]/.hcss/lstore/134211111
myObs = getObservation(obsid=134211111)

# Get your data from [HOME]/.hcss/lstore/MyFirstDataset
myObs = getObservation(obsid=134211111, poolName="MyFirstDataSet")

# Get your data from /BigDisc/PACS/MyFirstDataSet
myObs = getObservation(obsid=134211111, poolLocation="/BigDisc/PACS/",
 poolName="MyFirstDataSet")

# Get your data using a tag
myObs = getObservation(obsid=134211111, tag="Reprocessed version 2")

# Get your data from the MyHSA pool
myObs = getObservation(obsid=134211111, poolName="MyHSA")
```

Example 1.24. Several examples using the getObservation task.

1.8. Customising the Product Browser results

You can customise the result area of the Product Browser perspective (Figure 1.22) in the following ways:

- Click on a column title to sort the column in ascending or descending order. You can sort up to three columns. Double click a column to reset sorting.
- Drag and drop a column title to move the column.

- Right click on a column title to show a menu with additional options:
  - Select one of the two predefined layouts. The layouts differ by the type, size and position of visible column.
  - Save the current layout with a custom name. The new layout will then be available from the *Select Layout* submenu. You can also overwrite one of the predefined layouts, although this is not recommended.
  - Add a column before or after the current column.
  - Remove the current column from the layout.

1.9. How to use the Quick Analysis perspective

GUI method

To perform a search with the Quick Analysis tool, select any of the search tabs present at the top of the interface and fill the required inputs in the following fashion:

- *Search by Target:* give the **Target** name, which will be resolved by SIMBAD or NED and, if wanted, the search **Radius** around the Target. If not specified, the search Radius is 10 arcsec. Note that any observation within the given Radius will also be retrieved.
- *Search by RA/Dec:* In this case, the required data are **Right Ascension** (hh:mm:ss), **Declination** (dd:mm:ss) and a search **Radius** in arcsec like in the above case.
- *Search by Obs ID:* The only data required is the **Observation identifier**.
- *Search by OpDay:* all observations taken in the given **Operational Day** will be returned.

To execute any of the searches, either press the Search button on the right represented by magnifying glasses or press the **Enter** key in any of the text fields. Please note that only the active tab data will be taken into account when performing the search.

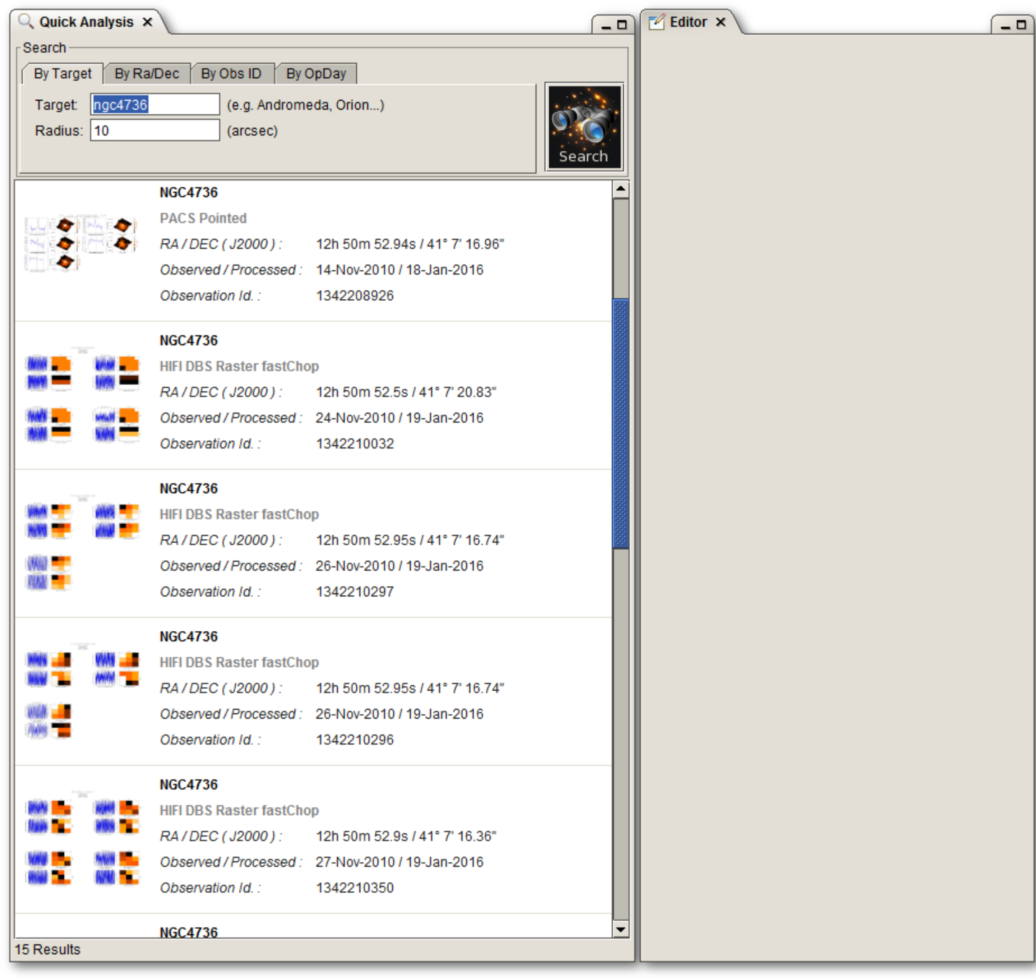


Figure 1.23. Main window of the Quick Analysis perspective with the results of a search by target name.

Upon execution of the search (see the figure above), the results found will be displayed just below the search panels. The data loaded are Browse Products, which are a particular subset of the final science products generated by the pipeline (see the Data Product Overview (please open this link in a new page or tab) web page for details on the products served per sub-instrument). Selecting any of the results will open the most appropriate analysis tool for the corresponding type of data:

- For cubes and spectrum datasets, the **Spectrum Explorer** will be opened. For more information, see Chapter 5.
- For maps, an instance of the `Display` class will be created with all the tools required to analyse the image. For more information, see Section 4.1.

It is also possible to work on the data using these standard tools without any further action, if wished.

Additionally, you can right-click on the result to be able to choose between opening the Browse Product or an Observation Context. This choice can be saved by click on *Edit* → *Preferences* , navigating to *General* > *Appearance* > *Quick Analysis* and set as default the desired behaviour.

Note the difference in the contents of the Data navigator panel (left of the spectrum plot) between opening only the Browse Product:

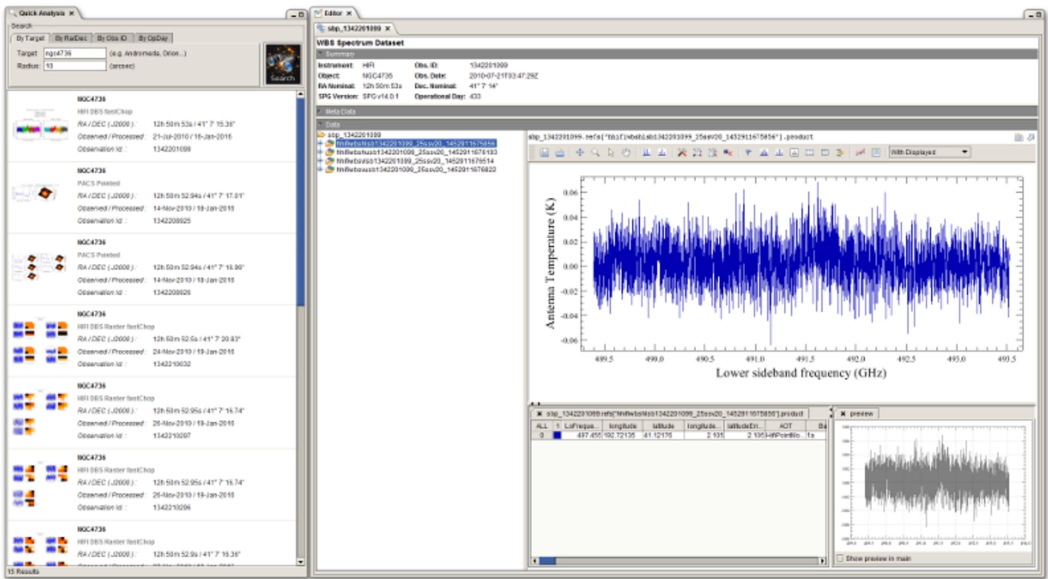


Figure 1.24. A detailed view of the result of a search using Quick Analysis (Browse Products).

Or opening the whole Observation Context and focusing on the product:

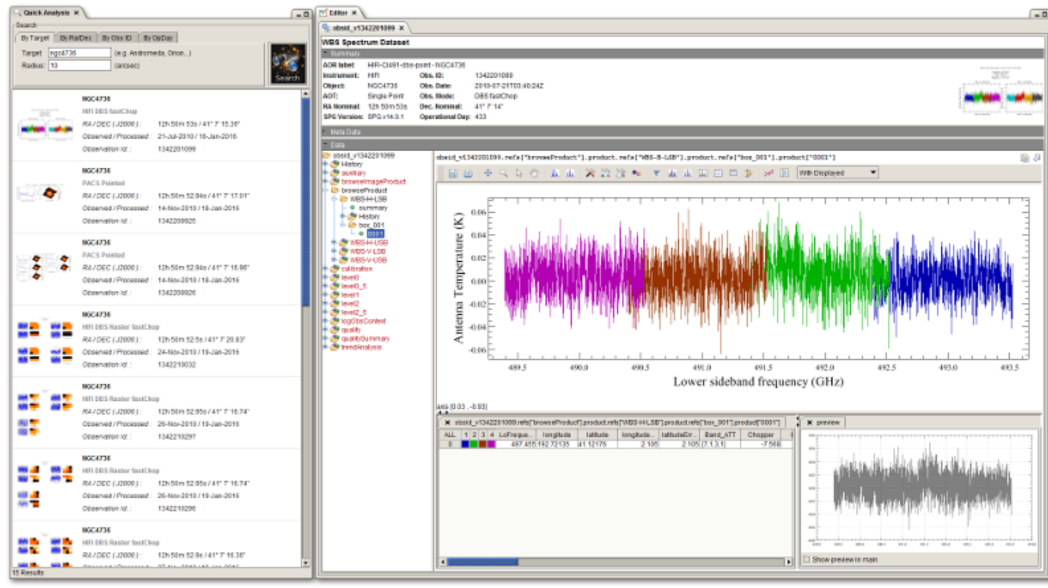


Figure 1.25. A detailed view of the result of a search using Quick Analysis (Observation Context mode).

Command-line method

Note

While using the Quick Analysis perspective, all GUI interactions are echoed to the Console. The Console view is not included in this perspective, but you can open it manually by clicking on *Window* → *Show View* → *Workbench* → *Console* .

1.10. Saving data (products and observations) to disk

Prerequisites. You have made changes on an observation, for instance by reprocessing data with a pipeline script. You now want to save your reprocessed data. As you will see, the procedure is the same whether you want to save an entire observation or a smaller piece of data *that is still a product*.

GUI method

To save an observation or product to disk, follow these steps:

1. Right click on the data variable name in the *Variables* view and choose *Send to* → *Local pool* .

   The *Save Products Tool* opens in the *Editor* view (see Figure 1.26). Your observation variable name appears in the *Products* column.

2. Assign one or more *tags* to the observation you want to save. To assign a tag to a product, double click the corresponding cell in the *Tags* column, write the tag and press **Enter** .

   Tags are keywords or phrases you can associate to a product, to better describe and remember its contents. For example, you could assign to a product the tag "To be completed" to remember that you have not finished processing it. When defining tags, you are free to use the keywords and phrases that work best for you. Tags are especially useful to recognise different versions of the same observation, which by definition have the same obsid.

3. Select a pool from the *Select Pool* drop-down list, or write a pool name. If a pool with that name does not exist, it is created.

4. Make sure that the observations and products you want to save are selected and click *Save* to store them into the pool.

   When you press *Save* there is no success or failure message. You can check the *Console* view, where the corresponding command has been echoed, to make sure that the data have been saved correctly.

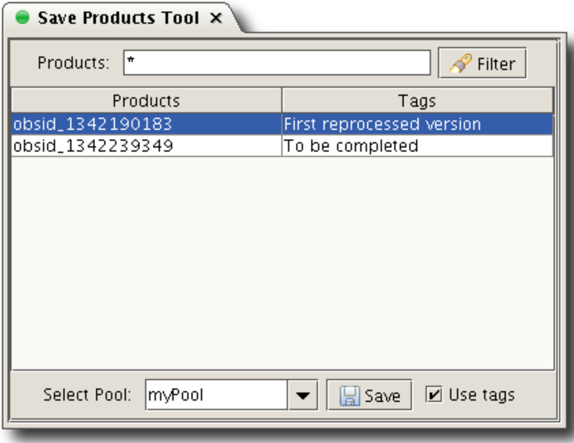


Figure 1.26. The Save Products tool.

Command-line method

To save products (including observations) to disk, use the **saveProduct** command. You must specify the variable you want to save, the pool you want to save it to and the tag you want to associate to it. Adding a tag is not compulsory but strongly recommended.

Tags are keywords or phrases you can associate to a product, to better describe and remember its contents. For example, you could assign to a product the tag "To be completed" to remember that you

have not finished processing it. When defining tags, you are free to use the keywords and phrases that work best for you. Tags are especially useful to recognise different versions of the same observation, which by definition have the same obsid.

```
bg("saveProduct(product=myObs, pool='myPool', tag='My reprocessed data')")
```

Example 1.25. Save an observation to disk specifying the pool name and a tag.

If a pool with the name you specify does not exist, it is created.

Note

If you use the `saveObservation` in *HCSS User's Reference Manual* command, note these differences with respect to `saveProduct` :

- `saveObservation` does not support tags.
- `saveObservation` does not store the calibration tree when used with observations.

Where are my data? Your data are now in a *local pool* , typically in a subdirectory of `.hcss/lstore` in your home directory. Do not touch the files directly, but keep working on them from HIPE. If you modify the files outside HIPE, you may corrupt the pool structure.

Where do I go from here? Now that your data are saved, you have these possibilities:

- If you quit HIPE and start a new session at a later date, you will want to load the data you have saved. See Section 1.7 for more information.
- After saving data on your local disk, you may want to share them with a colleague. See Section 1.12 for information on how to export an observation for sharing with other HIPE users.

1.11. Migrating pools across *incompatible versions* of HIPE

HIPE 10 developer versions (builds between 2069 and 2674) and HIPE 11 or later pools are incompatible with earlier versions of HIPE (including all public releases of HIPE 10).

In order to use (where "use" means any process involving reading, scanning, or checking of the local pools by HIPE and not just explicitly opening the pool from, for example, the Product Browser) these existing pools with an incompatible HIPE version, you should rebuild the index of the pool. Given that it is not guaranteed that newer data will work with older versions of the software, this guide will focus only on making the pools useable in the following cases:

- **Case 1:** Using pools that were created/modified in a HIPE *10* version *prior to 2069*, with a HIPE *10* version *between 2069 and 2674*.
- **Case 2:** Using pools that were created/modified in a HIPE *10* version *between 2069 and 2674*, with a HIPE *10* version *after 2674*.
- **Case 3:** Using pools that were created/modified in a HIPE *10* version *other than the 2069-2674 range*, with HIPE *11 or later*.
- **Case 4:** Using pools with an index rebuilt for a HIPE *10* version *other than the range 2069-2674*, with HIPE *11 or later*.

Prerequisite: A pool created with a HIPE 10 version prior to 2069.

Case 1:

- *Pool you want to open:* A pool like the one detailed in the prerequisite.
- *Error message:* There is no error displayed and the pool is completely usable in HIPE 10 versions 2069-2674.

Case 2:

- *Pool you want to open:* A pool that is like the prerequisite or has been processed in a scenario like Case 1.
- *Error message:* Querying the pool using the Product Browser returns nothing (no error message). Opening the observation from the Console using getObservation returns the following error message:

```
#herschel.ia.task.TaskException: Error processing getObservation task: Index Version
 not compatible.
Expected : 4 Existing: 6.Pool pool_name requires upgrading before you can use it with
 this software.
In order to do so, you need to run pool_name.rebuildIndex() to upgrade.Depending on the
 size of the
pool this process can take a long time, please be patient!More information can be found
 in the Data
Analysis Guide, section 1.2.2.1 (Update of index format for local stores).
```

- *Notes about the error message:*
  - `pool_name` is the name of the pool you are trying to load.
  - `pool_name.rebuildIndex()` is not the proper command to rebuild the pool. See below for the appropriate commands.
  - The section of the Data Analysis Guide referenced is not correct. This is Section 1.11 .
- *Workaround:*

  Any of the following commands will rebuild the index of the pool:

  - This ensures we are rebuilding the specified pool using a static method from the class `LocalStoreFactory`:

    ```
    LocalStoreFactory.getStore(pool_name).rebuildIndex()
    ```

    Example 1.26. How to rebuild the index of a local pool.

  - This rebuilds the pool using a `ProductStorage` instance named `store` :

    ```
    store.writablePool.rebuildIndex()
    ```

    Example 1.27. Second option for rebuilding the index of a local pool.

Case 3:

- *Pool you want to open:* A pool that is like the prerequisite.
- *Error message:* There is no error displayed and the pool is completely usable in HIPE 11 or later.

Case 4:

- *Pool you want to open:* A pool which index was rebuilt following Case 2 instructions.
- *Error message:* There is no error displayed and the pool is completely usable in HIPE 11 or later.

That's it! You should now be able to read your pool. If you find you cannot, contact the Helpdesk.

Tip

Note that the index files will not be backed up while you run rebuildIndex(), unless you invoke `Configuration.setProperty("hcss.ia.pal.pool.lstore.index.backup", "true")` first.

See the HIPE Owner's Guide: Section 4 in *HIPE Owner's Guide* .

1.12. Exporting an observation to a colleague

Prerequisites. You have reprocessed one or more observations and saved them to disk as explained in Section 1.10 .

You can pack data held in local pools on your disk so that you can send them to other HIPE users. Follow these steps:

1. Choose *Window → Show View → Data Access → Export Herschel data from HIPE* .

   The *Export Herschel data from HIPE* view opens.

2. From the *Input Pool* drop-down list select the pool that contains the data you want to export.

   a. **If you want to export the whole pool.** Click *Export pool* and enter the name of the zip file the data will be exported to.

   b. **If you want to export one or more observations from the pool. .**

      i. Click *Show Contents* . The full observations contained in the pool appear in the *Observations* area.

      ii. Select the observations you want to export from the list.

      iii. Choose the export data format from the drop-down list at the bottom of the view. You can choose among compressed tar (recommended), uncompressed tar and loose files (unpacked directory).

      iv. Click *Export to HSA hierarchical structure* and enter the name of the file the observations will be exported to.

Note

What is meant by *HSA hierarchical structure* is the same hierarchical directory structure of the data downloaded from the HSA. The format of individual files is FITS.

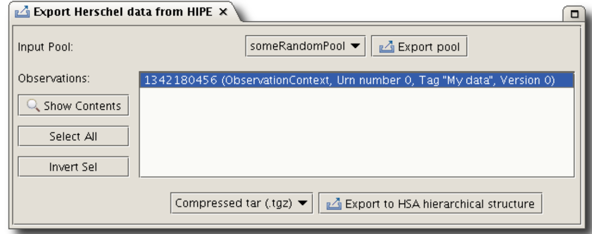


Figure 1.27. Product export from HIPE into standard Herschel directory structure.

This view makes use of the `exportObservation` task, documented in the *User's Reference Manual* : Section 1.131 in *HCSS User's Reference Manual*

Where do I go from here? Now you can send the zip or tar file you produced to other users. This is how they can load your data into their HIPE installation:

- **If you exported your pool by clicking *Export pool* ..** Other users can extract the zip file into their local pool directory (typically `.hcss/lstore` in their home directory) and find your data via the Product Browser as explained in Section 1.7 .
- **If you exported one or more observations by clicking *Export to HSA Data Format* ..** Other users can uncompress the tar files and load the observations into HIPE as described in Section 1.5 .

1.13. Retrieving products from disk

You can use the Product Browser perspective to find data products inside observations on your disk. The procedure is the same as for finding whole observations, as described in Section 1.7 , with only these differences:

- Rather than setting your search parameters in the *Observations* tab alone, you will want to use the *Products* , *Metadata* and *Free Metadata* tabs as well. For example, you may indicate an observation ID and select a particular type of product from the *Product type* drop-down list in the *Products* tab.
- If you do not see the product type you want in the *Product type* drop-down list in the *Products* tab, try clicking the icon to reload the product types.

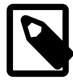
Tip

The Product Browser does not find data I know to be present. Check the following:

- Make sure that the location of your data is selected in the *Data Source* panel. If in doubt, select more data locations.
- Make sure you are searching for the right kind of data. For example, you may be in the *Observations* tab, thus searching for entire observations when instead you want to look for a data product inside an observation. If so, switch to the *Products* tab and make sure that the *Product type* field has the correct value. If in doubt, choose `herschel.ia.dataset.Product` to search for any data product.

1.14. Removing data from disk

Removing reprocessed data. You can remove data products, up to full observations, from the result area of the Product Browser (see Figure 1.22). Select one or more rows, then right click and choose *Remove product from storage/pool* . Click *Yes* in the confirmation window to remove the data.



Note

Removing a product context, such as an observation context, does not remove just the context itself (the container) but also its child products (the contents).

Removing data downloaded from the HSA. Original data downloaded from the HSA cannot be removed in the Product Browser. Instead, choose *Edit* → *Preferences* and click *My HSA* in the left-hand side list. The *My HSA* panel opens. Here you can select data you loaded from tar files downloaded from the HSA (*Indexed Datasets from HSA tar files*) or online data you browsed from the Product Browser (*Direct retrieval sessions*). Click *Remove* to delete the selected data items.

Warning

Do not remove data from disk while you have variables in HIPE referring to those data. Since HIPE does not always keep all the contents of a variable in memory, you may not be able to save the variable contents to disk again.

1.15. On-demand reprocessing of observations

You can reprocess data "on demand" to have the most updated data products.

The HSA contains data processed with different versions of the processing pipeline and calibrations. Although all the data are periodically bulk-reprocessed with certain pipeline versions and the same calibration files, a certain degree of inhomogeneity is unavoidable, since the HSA is the science archive of an operational mission, whose content is being continuously upgraded.

To reprocess your data with the latest operational version of the pipeline and calibration files, you can submit a request for *on-demand reprocessing* . The selected observations are processed at the Herschel Science Centre and the results are provided to you through FTP. Only the same processing profile which is used by the standard processing is available in on-demand mode.

You can request on-demand reprocessing by following these steps:

1. Log into the HSA and open the HSA User Interface as described in Section 1.4.1 .
2. Search for the observations you want to reprocess, as described in Section 1.4.2 .
3. Select the observations for which you would like to request on-demand reprocessing, by ticking the checkbox next to the observation record.
4. Click the shopping cart icon in the toolbar of the HSA User Interface.

   The *Shopping Basket* tab opens.
5. Click the *On Demand Reprocessing* tab.
6. Click *Submit Request* .

To monitor the status of your on-demand jobs, choose *Windows → On Demand Monitor* in the HSA User Interface.

Once the reprocessing has been completed, you receive an email notifying the availability of processed data for FTP retrieval.

The new products are delivered in a zipped file. Unzip this file in your favourite directory and load the observations into HIPE by using the *Navigator* view or the `getObservation` task, as explained in Section 1.5 .

Since the products generated in on-demand mode are not stored in the HSA, they cannot be indexed under MyHSA. Instead they are indexed in your local store (see Section 1.3) under a pool called by the same name as the zip file provided by the HSA (`<instrument>_<obsid>` , for instance `s_1342216878`). Therefore, once the observation has been indexed for the first time, subsequent recoveries of it from your disk should be done with the `getObservation` command or the Product Browser as explained in Section 1.7 .

Note

On-demand reprocessing is intended to be used for a limited set of observations. This is because this functionality makes use of the same operational system at the Herschel Science Centre which is used for the daily processing of Herschel data and for bulk reprocessing. These processes always have priority over any on-demand reprocessing request.

1.16. Exchanging data with FITS files

1.16.1. Saving a product to a FITS file

You can save any kind of Herschel data to FITS files, as long as it is of type `Product` or a dataset such as a `TableDataset`. All the raw and reduced data coming from the Herschel Science Archive

are either products or datasets. Note that you cannot save arrays such as `Double1d` (for example, single columns extracted from a table dataset). In that case, see the end of this section for how to wrap arrays into datasets.

To save a product or dataset as FITS file, follow these steps:

1. Select the product or dataset in the *Variables* view.
2. Right click on the variable name and choose *Send to* → *FITS file*.

   The `simpleFitsWriter` task dialogue window opens in the *Editor* view, as shown in Figure 1.28.
3. Write the name of the FITS file in the *file* field. Alternatively, click the folder icon to browse to a different directory.
4. Optionally, tick the *Ask before overwriting* checkbox to be warned if you are about to overwrite an existing file.
5. Optionally, choose a compression method from the *compression* drop-down list. You can choose between *ZIP* and *GZIP*.
6. Press *Accept* to save the product or dataset to file.

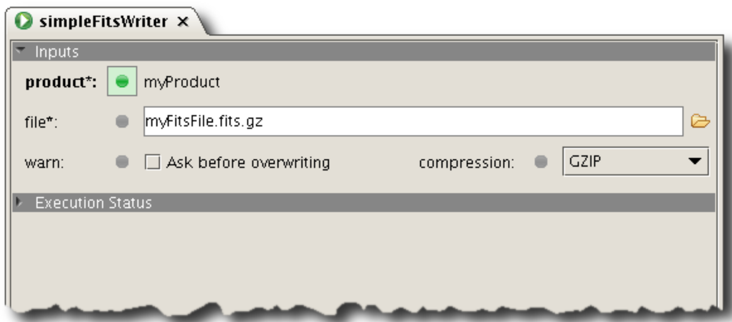


Figure 1.28. FITS save task dialogue window.

Note

- You are responsible for adding the `.fits` extension to the file name, plus any additional extension, such as `.gz`, if you choose a compression method. If you fail to do so, other applications such as ds9 may not handle the file correctly.
- Unless you choose a different directory, FITS files are saved in the directory HIPE was started from. To locate this directory, issue these commands in the *Console* view:

```
import os
      print os.getcwd()
```

Example 1.28. How to get the current workind directory for the Jython interpreter.

From the command line

You can write a product or dataset to a FITS file with the simpleFitsWriter in *HCSS User's Reference Manual* task. Follow the link to access the corresponding entry in the *User's Reference Manual*.

```
myProduct = Product()  # Empty data product
simpleFitsWriter(myProduct, "myProduct.fits")
```

Example 1.29. Creating a new empty data product and writing it to disk as a FITS file.

Files are saved in the directory from which you started HIPE, unless you provide a different path with the file name.

The following commands create an image and save it as a multi-extension FITS file:

```
# Saving an image composed of random data to disk
myImage = SimpleImage(description="An image",image=Double2d(50,100), \
          error=Double2d(50,100),exposure=Double2d(50,100))
simpleFitsWriter(myImage, "myImage.fits")
# Reading back the file created
path = "myImage.fits"
myImage = importImage(filename = path)
```

Example 1.30. Creating a SimpleImage with random data and saving it to disk as a FITS file, reading it back afterwards.

Warning

The above code generates a FITS file with the value 50 assigned to the `NAXIS2` keyword and 100 assigned to `NAXIS1`. In other words, the image size is 50 pixels along the *y* axis and 100 pixels along the *x* axis. The coordinate values are displayed in this order (*y*, *x*) in the Image Viewer. For an explanation of why the *y* size is specified *before* the *x* size, see the *Scripting Guide* : Section 2.2.5 in *Scripting Guide*.

If you get a `SignatureException` error when trying to save a variable to FITS, it probably means that your variables are not a product or dataset, but a simple array, such as a `Double1d`. To save it to FITS file you have to manually wrap a table dataset around it:

```
myArray = Double1d(10, 10.0)  # Array, cannot be saved to FITS
myTable = TableDataset()
myTable["myArray"] = Column(myArray)  # Putting array into dataset
simpleFitsWriter(myTable, "myTable.fits")
```

Example 1.31. Using a dataset as a wrapper to store an array in a FITS file.

1.16.2. Retrieving a Herschel product from a FITS file

To load a Herschel product stored in a FITS file (or any other standard FITS file), do either of the following:

- Double click on the FITS file in the *Navigator* view.
- Choose *File* → *Open File* , select the FITS file and click *Open*.

The tasks used by HIPE to load FITS files are `fitsReader` and `simpleFitsReader`. The `fitsReader` task (see Figure 1.29) tries to guess the file contents by looking at the `XTENSION` keyword, and puts the contents in a variable of the appropriate type.

Note

This procedure is also valid for high-level reduced data from ISO, XMM-Newton, ALMA and SOFIA. For files that aren't correctly imported this way, please see Section 1.16.6

If `fitsReader` does not recognise the file contents, it defaults to the `simpleFitsReader` task. This task is optimised to read data from FITS files as packaged by HIPE. If the file is not a HIPE FITS product, the contents are put in unformatted arrays. You can choose how to read the file or let the software choose.

To run `fitsReader` or `simpleFitsReader` from HIPE, go to the *Tasks* view, select the *All* tasks folder and scroll down to `fitsReader` or `simpleFitsReader`. Double click on the task name

to open its dialogue window. Insert the input file name and click the Accept button to run the task and read in the FITS file.

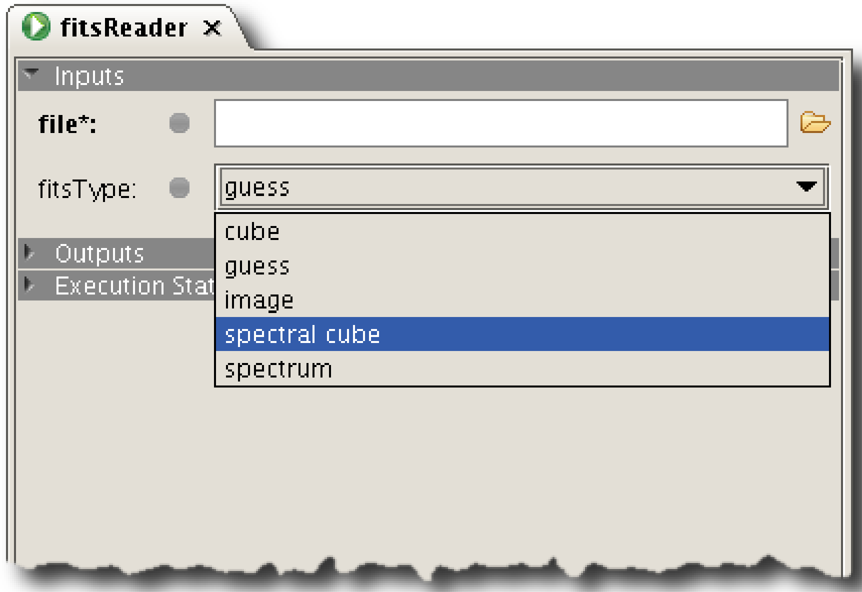


Figure 1.29. FITS read task dialogue window.

From the command line

You can read data from a FITS file into HIPE with the fitsReader in *HCSS User's Reference Manual* task. Follow the link to access the corresponding entry in the *User's Reference Manual*.

```
myProduct = fitsReader("myProduct.fits")  # Load a product from FITS
```

Example 1.32. Load a product from a FITS file.

1.16.3. Translation of Herschel metadata to FITS keywords

Long, mixed-case parameter names, defined in the metadata of your product, are converted to a FITS compliant notation. This notation dictates that parameter names must be uppercase, with a maximum length of eight characters.

HIPE uses the following lookup dictionaries to convert well-known FITS parameter names into a convenient and human-readable name:

- Common keywords widely used within the astronomical community, which are taken from HEASARC.
- Standard FITS keywords.
- HCSS keywords containing keywords that are not defined in the above dictionaries.
- Instrument-specific dictionaries:
  - The HIFI-specific dictionary is included in the HCSS dictionary and maintained as a part of it. See above.

- The PACS-specific dictionary is located here.

For example the following metadata is transformed into a known FITS keyword:

```
product.meta["softwareTaskName"]=StringParameter("FooBar")
```

Example 1.33. Setting a metadata property to a StringParameter value.

The result in the FITS product header is the following:

```
HIERARCH key.PROGRAM='softwareTaskName'
   PROGRAM = 'FooBar '
```

A full demonstration is available in the following example. The script creates a product with several nested datasets, stores it into a FITS file, and then retrieves it again.

```
# First we will get some unit definitions for our example
from herschel.share.unit import *
from java.lang.Math import PI

# Construction of a product (only for demonstration purposes)
points = 50
x = Double1d.range(points)
x *= 2*PI/points
eV = Energy.ELECTRON_VOLTS
# Create an array dataset that will eventually be exported
s = ArrayDataset(data = x, description = "range of real values", \
unit = eV)
degK = Temperature.KELVIN
# Provide some metadata for it (header information)
s.meta["temperature"] = LongParameter(long=293,\
description="room temperature", unit = degK)

# We can store the array in a FITS file
# after making it a Product
p = Product(description="FITS demonstration",creator="You")
# Add some meta data
p.meta["sampleKeyword"]=StringParameter("First FITS file")
p.meta["observationInstrumentMode"]=StringParameter("UnitTest")
# Add the array of data to the product
p["myArray"] = s
# Store in FITS file
fits = FitsArchive()
fits.save("sdemo.fits", p)

# And restore it
scopy = fits.load("sdemo.fits")

# Create a TableDataset for export
t = TableDataset(description = "This is a table")
t["x"] = Column(x)
t["sin"] = Column(data=SIN(x),description="sin(x)")

# And a composite dataset with an array and a table in it
c = CompositeDataset(description="Composite with three datasets!")
c.meta["exposeTime"] = DoubleParameter(double=10,description="duration")
c["childArray"] = s
c["childTable"] = t
c["childNest"] = CompositeDataset("Empty child, just to prove nesting")

# And finally, a product that has the composite dataset,
# TableDataset and array dataset.
p = Product(description="FITS demonstration",creator="demo.py")
p.creator = "You?"
p.modelName = "demonstration"
p.meta["sampleKeyword"] = \
```

```
StringParameter("Example keyword not in FITS dictionaries")
p.meta["observationInstrumentMode"] = StringParameter("UnitTest")
p["myArray"] = s
p["myTable"] = t
p["myNest"] = c

# Save our product ...
fits.save("demo.fits",p)
# ... load it back into a new variable, n,...
n = fits.load("demo.fits")
# ... and show it!
print n
print n["myArray"]
print n["myNest"]
print n["myNest"]["childNest"]

# We can also get information on the metadata/keywords
print n.meta
# And look at a specific piece of metadata
print n.meta["startDate"]
```

Example 1.34. Create FITS file from random data and read it back.

1.16.4. Structure of Herschel products when saved as FITS

This section describes the structure of FITS files created from typical Herschel product types appearing in Level 2 data.

All FITS files described here, when produced from Herschel observation products, also have a `History` extension with three child extensions: `HistoryScript`, `HistoryTasks` and `HistoryParameters`. These are explained separately in Section 1.16.4.9.

How to export data products from HIPE to other astronomical software is described in Section 1.16.7. If you have successfully exported Herschel data to other software, you are encouraged to contribute information to this page. Click the icon in the toolbar of the HIPE Help System to get in touch with us.

1.16.4.1. General information

World Coordinate System. WCS information is held in the main header of the FITS file and in the `image` extension, for those products that have an `image` dataset.

Measurement units. Information on measurement units is held in the header of each FITS extension. Look for the `QTTY_____` **and** `BUNIT` keywords, unless stated otherwise in the following sections.

1.16.4.2. SimpleImage

A FITS file from a `SimpleImage` shows *at least* three image extensions called `image`, `error` and `coverage`. These have the same size and contain the flux, error and coverage information of the original image, respectively. Usually there is also a `History` extension and more could be created by the pipelines. To check which extensions are present in the product, you can use `print` *`mySimpleImage`* and check the contents of the `datasets` attribute.

If a WCS is present in the original image, the WCS keywords appear in the FITS file for each array-like extension such as `coverage`, `error`.

For the most current information about the structure of products and their datasets, you can check the Product Definitions Document and, for PACS products, the PACS Products Explained document.

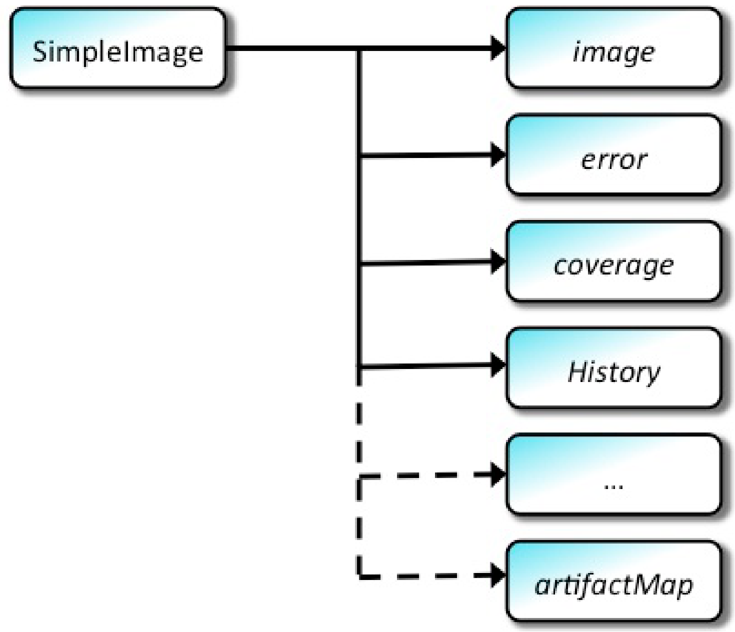


Figure 1.30. Structure of a FITS file produced from a `SimpleImage`.

1.16.4.3. SpectralSimpleCube for PACS

A `SpectralSimpleCube` has two three-dimensional datasets, `image` and `coverage`, and one table dataset `ImageIndex`, with two columns relating each cube layer to its wavelength. The `LayerCount` column contains the layer index (starting from zero) and the `DepthIndex` column contains the corresponding wavelength.

These datasets are translated to two image and one binary extension in the FITS file, with the same names. The wavelength measurement unit is held in the header of the `ImageIndex` extension, under the `TUNIT1` keyword.

If a WCS is present in the original image, this is kept in the FITS file.

Tip

A PACS projected cube is a `SpectralSimpleCube`.

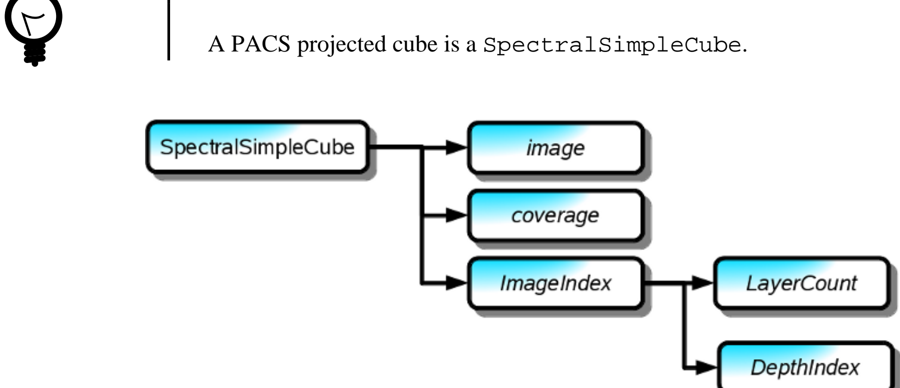


Figure 1.31. Structure of a FITS file produced from a `SpectralSimpleCube` from a PACS observation. The two columns of the `ImageIndex` binary table extension are shown.

1.16.4.4. SpectralSimpleCube for SPIRE

The structure of this product, and corresponding FITS files, for SPIRE observation is mostly the same as for PACS observations, as described in Section 1.16.4.3. The only difference is the addition of

two more three-dimensional datasets, `error` and `flag` , converted to two image extensions in the FITS file.

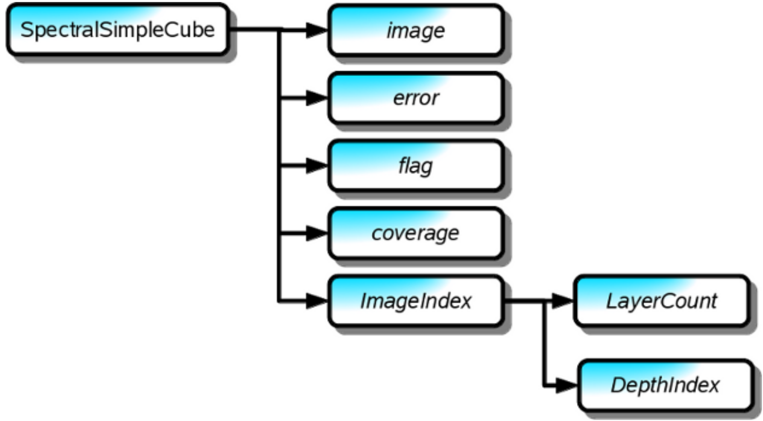


Figure 1.32. Structure of a FITS file produced from a `SpectralSimpleCube` from a SPIRE observation. The two columns of the `ImageIndex` binary table extension are shown.

1.16.4.5. SpectralSimpleCube for HIFI

You can find `SpectralSimpleCube` objects in Level 2.5 HIFI data. These cubes are made of three three-dimensional datasets, called `image`, `weight` and `flag`. These are converted to three image extensions in the FITS file.

Unlike cubes from PACS and SPIRE observations, there is no `ImageIndex` dataset relating cube layers to their wavelength (frequency for HIFI). Instead, you can look at the `image` dataset metadata, where parameters `crpix3`, `crval3`, `ctype3` and so on define the reference layer, unit and scale of the frequency axis.

These keywords are translated to the header of the `image` extension in the FITS file.

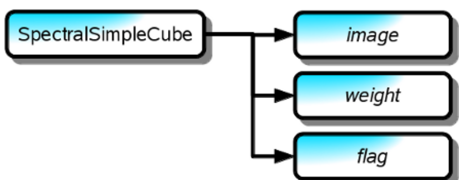


Figure 1.33. Structure of a FITS file produced from a `SpectralSimpleCube` from a HIFI observation.

1.16.4.6. PacsRebinnedCube

A `PacsRebinnedCube` derives from a `SpectralSimpleCube` and adds more components to it. The exported FITS file is correspondingly more complicated.

Cube data is held in six three-dimensional image extensions, called `image`, `ra`, `dec`, `stddev`, `exposure` and `flag`. The `ra` and `dec` extensions hold the coordinates of each pixel in degrees (the measurement unit is shown in the extension header).

The `ImageIndex` extension relates each cube slice to its wavelength, in the same way as with a `SpectralSimpleCube`. The `waveGrid` extension contains the same wavelength information as the `ImageIndex` extension, but without the `LayerCount` column.

The contents of the `qualityControl` extension can be ignored.

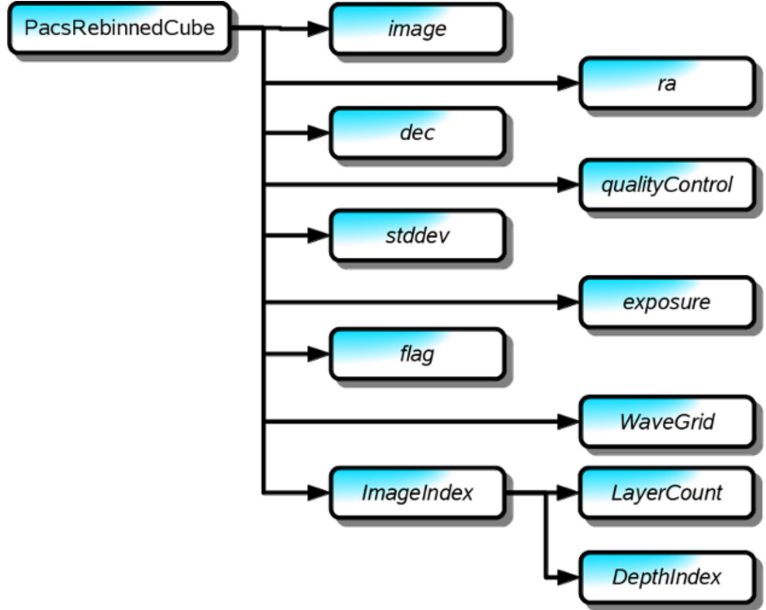


Figure 1.34. Structure of a FITS file produced from a `PacsRebinnedCube`. The two columns of the `ImageIndex` binary table extension are shown.

1.16.4.7. HifiTimelineProduct

The `HifiTimelineProduct` is a *product context* (a container with references to other products), which means that it cannot be saved as FITS file from HIPE.

Inside a `HifiTimelineProduct` there are a summary table and one or more `DatasetWrapper` products (one per building block) containing a number of `SpectrumDataset` objects.

The summary table and each `DatasetWrapper` can be separately saved as FITS files, but note that these FITS files will not have the *History* extension.

The FITS file of a summary table has one binary extension called *wrapped*, which reproduces the original table.

The FITS file of a `DatasetWrapper` has one binary extension per spectrum. These are called `0001`, `0002` and so on. **These extensions contain the actual spectra.** Each extension is a table with one row and as many columns as the parameters describing the spectrum. Each table cell may contain a single value (like *longitude* and *obs time*) or an array of values (like *flux* and *lsbfrequency*).

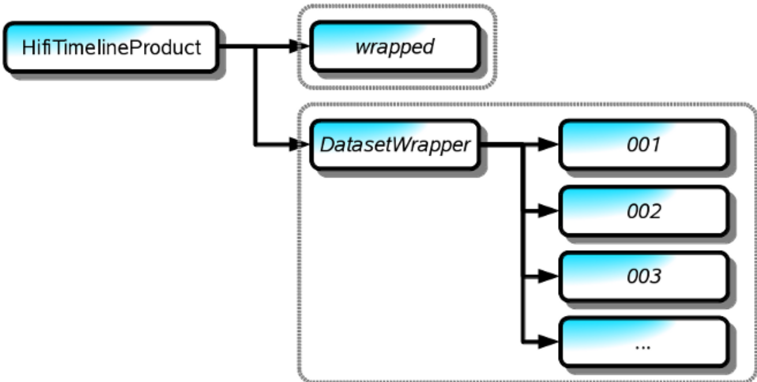


Figure 1.35. Structure of a FITS file produced from a `HifiTimelineProduct`. This product cannot be saved directly as a FITS file, but the summary table and each `DatasetWrapper` can. The dashed gray lines show the contents of each FITS file.

1.16.4.8. SpectrometerPointSourceSpectrum

This SPIRE product has one extension called `0000`, with two children extensions called `SSWD4` and `SLWD4`. These correspond to the centre bolometers of the short and long wavelength spectrometer arrays, respectively. Each extension is a table with five columns: `wave` (wavelength), `flux`, `error`, `mask` (zero unless mask flags have been applied) and `numScans`. Each row is a data point of the spectrum.

The `numScans` column is not present for data processed with SPG versions prior to 9.1.0.

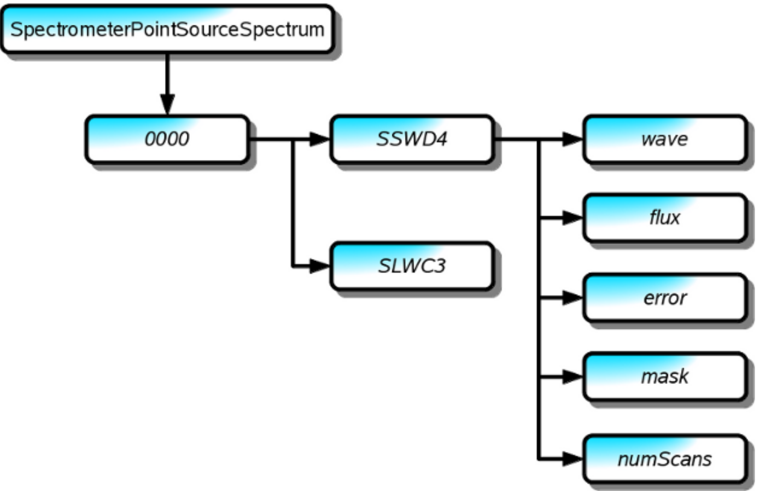


Figure 1.36. Structure of a FITS file produced from a `SpectrometerPointSourceSpectrum`. The five table columns are shown for the `SSWD4` extension. They are the same for the `SLWC3` extension.

1.16.4.9. The History extension

The `History` extension is part of all the FITS files generated from HIPE products, including those described in the previous sections. It contains the following child extensions, all in binary table format:

- **HistoryScript.** This table contains a Jython script with all the operations performed on the data that resulted in this data product. The table as a single column, and each row corresponds to a line of the script.
- **HistoryTasks.** This table shows the names of all the tasks used in data processing, and the corresponding HIPE version and build number. The execution date and time is also shown in the `ExecDate` column. The format is `FineTime`, that is, the number of microseconds since 1st January 1958. To convert a value to a more convenient format, you can use a command like the following in the *Console* view of HIPE:

```
print FineTime(1693341725238000)
```

  Example 1.35. Printing a FineTime formatted string to the console.

- **HistoryParameters.** This table lists all the task parameters used during data processing, with their type, value and whether the value used was the default (column `IsDefault`). Note that, for parameters, of type `PRODUCT`, the value is usually an expression like `hash2798118624`. This is a unique value identifying the particular data product that was used.

  With the `TaskID` column you can find the task a given parameter was used in, by comparing the value with those in the `ID` column of the `HistoryTasks` column.

For more information about history in products, see the *Scripting Guide*: Section 2.8.7 in *Scripting Guide*.

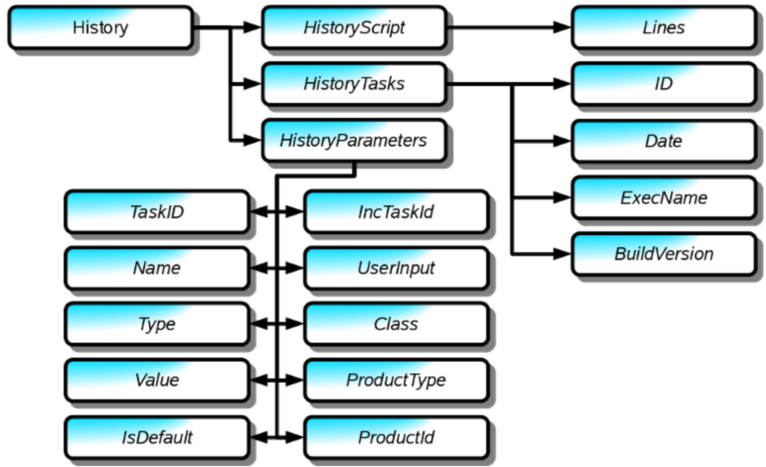


Figure 1.37. Structure of the `History` extension of a FITS file created from a Herschel product. Column names for each of the three binary table extensions are shown.

1.16.5. Troubleshooting FITS import/export

For more information see the FITS IO general documentation.

Problems opening FITS files created by HIPE

If you export a FITS file from HIPE and modify it with an external program, HIPE may not be able to import it anymore. If this happens, follow these steps:

1. Open the FITS file with a FITS editing program such as **fv**.
2. Delete the `HCSS_____` keyword from the header of *all extensions*.
3. Save the file.

HIPE should now be able to read the file.

FITS header character limit

A FITS header card is limited to 80 characters. StringParameters and FITS card descriptions longer than the allocated length are distributed over multiple lines. An `&` character at the end of a line means that the text continues on the next line. The keyword `CONTINUE` is used for the lines after the first one.

Opening multi-extension FITS files in DS9

When FITS files with multiple extensions are opened as cubes in DS9, the application crashes. One alternative is to open the different extensions in separate frames, for which you need at least version 6 of DS9. Version 6 or higher of DS9 does not crash on cubes, and it correctly opens only the relevant extensions.

1.16.6. Importing a non-Herschel FITS file into HIPE

There are several options for importing data from a non-Herschel FITS file and making it fit into a Herschel product or dataset. These options go from a relatively simple class for reading/writing FITS files which is included along with Java (`FitsArchive`) to a HIPE task that tries to smartly guess the most similar Herschel product to create and load the FITS file data into (`fitsReader`). Some of them have been previously described in this section, but they will be mentioned again as part of a workflow that will allow you to select the most appropriate mechanism for importing some exotic FITS file.

1. The `FitsArchive` class can only be used via script and you can find examples in both the URM entry in *HCSS User's Reference Manual* and in this section in Example 1.39 (writing FITS only). The usual output for this class is simply a dataset that holds the data encapsulated in a generic `Product`.

2. The `simpleFitsReader` task has been explained before in Section 1.16.2. To use the task through the GUI, you can:

   - Double click on the FITS file in the *Navigator* view.

   - Choose *File* → *Open File* , select the FITS file and click *Open* .

   When using the GUI, if you select `guess` (see Figure 1.29) as the `fitsType` input parameter, the task that will be called internally is `fitsReader` instead. This option tries to guess the file contents by looking at the `XTENSION` keyword, and puts the contents in a variable of the appropriate type. If `fitsReader` does not recognise the file contents, it defaults to the `simpleFitsReader` task. This task is optimised to read data from FITS files as packaged by HIPE. If the file is not a HIPE FITS product, the usual output is a set of unformatted arrays. To run `fitsReader` or `simpleFitsReader` from HIPE, go to the *Tasks* view, select the *All* tasks folder and scroll down to `fitsReader` or `simpleFitsReader`. Double click on the task name to open its dialogue window. Insert the input file name and click the *Accept* button to run the task and read in the FITS file. Finally, if you want to script those tasks, either have a look at their URM entries (simpleFitsReader in *HCSS User's Reference Manual* or fitsReader in *HCSS User's Reference Manual*) or try this example copied from the URM:

```
filepath = "path_to_file/filename"
readertype = SimpleFitsReaderTask.ReaderType.STANDARD
product=simpleFitsReader(file=filepath, reader=readertype)
```

   Example 1.36. Importing a non-Herschel FITS file with the simpleFitsReader task.

   Remember that the `fitsReader` task is the same but it only requires one parameter (therefore using guessing implicitly): `file`.

3. If you know the HIPE type that most closely matches the FITS file data, you can use any of the `import` * tasks. These are the current import tasks:

   - `importCube` in *HCSS User's Reference Manual*

   - `importSpectralCube` in *HCSS User's Reference Manual*

   - `importImage` in *HCSS User's Reference Manual*

   - `importRgbImage` in *HCSS User's Reference Manual*

   The image importing tasks only have one parameter, a string containing the path to the file and return the image as an output parameter. For example:

```
filePath = "myNonHerschel.fits"
myImage = importImage(filename = filePath)
```

   Example 1.37. Importing non-Herschel FITS files using specific image import tasks.

   The cube importing tasks have two parameters, an input/output parameter that takes a previously defined variable and sets its content to the data in the FITS file and a string containing the path to the file. For example:

```
filePath = "myNonHerschel.fits"
myCube = SpectralSimpleCube()
myCube = importSpectralCube(spectralcube = myCube, filename = filePath)
```

   Example 1.38. Importing non-Herschel FITS files using specific spectral import tasks.

1.16.6.1. Using data from other missions and observatories

HIPE is able to load FITS files. Being an open format, many missions and observatories offer their products as FITS files. In particular, HIPE can read the FITS files that the Common Astronomy Software Applications (CASA) suite of tools produces. This software is used to process the raw data (unreadable by HIPE) obtained from ALMA interferometer and generate the final products (readable by HIPE in the form of a standard FITS file).

1.16.7. Importing a Herschel FITS file into external applications

This section describes how to import FITS files of Herschel products into some popular data analysis applications.

1.16.7.1. IDL

Importing images. See the following code:

```
IDL> im = mrdfits('/path/image.fits',1)
% Compiled module: FXMOVE.
% Compiled module: MRD_HREAD.
% Compiled module: FXPAR.
% Compiled module: GETTOK.
% Compiled module: VALID_NUM.
% Compiled module: MRD_SKIP.
MRDFITS: Image array (2012,2009)  Type=Real*8
% Compiled module: SWAP_ENDIAN_INPLACE.
IDL> tv,im
```

Importing spectra. See the following code:

```
IDL> spec = mrdfits('/path/spectrum.fits',2)
% Compiled module: MATCH.
% Compiled module: MRD_STRUCT.
MRDFITS: Binary table.  4 columns by  2061 rows.
IDL> help,spec,/struc
** Structure <15e03af4>, 4 tags, length=28, data length=28, refs=1:
  WAVE            DOUBLE           31.200000
  FLUX            DOUBLE           8.2931329
  ERROR           DOUBLE           3.4131544
  MASK            LONG                     0
IDL> plot,spec.wave,spec.flux
```

Importing cubes. See the following code:

```
IDL> cube = mrdfits('/path/cube.fits',2)
MRDFITS: Image array (16,18,374)  Type=Real*8
IDL> help,cube
CUBE            DOUBLE    = Array[16, 18, 374]
IDL> plot,cube[8,8,*]
```

In the case of PACS projected cubes, the structure of the FITS file is that described in Section 1.16.4.6 .

```
IDL> FITS_HELP,'path/cubeName.fits'
XTENSION  EXTNAME         EXTVER EXTLEVEL BITPIX GCOUNT  PCOUNT NAXIS  NAXIS*
0                                           32      0       0     0
1 IMAGE    image                           -64      1       0     3  39 x 39 x 29
2 IMAGE    coverage                        -64      1       0     3  39 x 39 x 29
3 BINTABLE ImageIndex                        8      1       0     2  12 x 29
4 IMAGE    History                          32      1       0     0
5 BINTABLE HistoryScript                     8      1       0     2  80 x 7
6 BINTABLE HistoryTasks                      8      1       0     2  35 x 1
7 BINTABLE HistoryParameters                 8      1       0     2  103 x 12
IDL> image = mrdfits('path/cubeName', 'image', hd) ; the header contains the image's
 WCS
IDL> imageIndex = mrdfits('path/cubeName','ImageIndex')
```

```
IDL> wave = imageIndex.depthindex  ; cube's wavescale
```

1.16.7.2. CLASS

You can read the FITS files produced with the `hiClass` task in HIPE on HIFI data with the following commands:

```
file out MyHIFISpectra.hifi mul
fits read MyHIFISpectra.fits
#
# Now you have a CLASS file named MyHIFISpectra.hifi (you can use whatever
# you want as an extension) you can access like you always do in CLASS:
#
file in MyHIFISpectra.hifi
find
get first
set unit f i
device image white
plot
```

For PACS data or any `Spectrum1d` product, run this script in HIPE:

```
Spectrum1d to CLASS FITS conversion
Written by C. Borys April 15, 2010
cborys@ipac.caltech.edu
Inspired greatly by HICLASS, written originally by Bertrand Delforge
and now maintained by Damien Rabois.
The core code was taken directly from that package.

NOTE: this code is specific for HIFI, and even then may lack
some of the keywords CLASS looks for.  The script is relatively
easy to tweak however.

Out of the box, this should work on the spectrum1d that
is output from HIFI's deconvolution task.  Indeed that was
the driver for this task in the first place.

'''

from herschel.ia.io.fits.dictionary import AbstractFitsDictionary
from herschel.share.fltdyn.time import FineTime
from java.util import Date
from herschel.share.unit import Frequency

# Define keyword dictionary
# The following class is stolen directly from HICLASS
class MyFitsDictionary(AbstractFitsDictionary):
"""Dictionary to use with FitsArchive to get proper keywords.

Because HCSS can use metadata parameters with fancy names and FITS
is stuck with keywords of 8 uppercase ASCII characters, a
dictionary is needed to convert the meta data parameter names into
FITS keywords.

The present class defines a dictionary for which the HCSS name and
the FITS name of a parameter are identical.  This allows you to
populate a HCSS dataset using the keywords you will want to see
appear in the FITS file.  And they will be used.

When instanciating this dictionary, feed to the constructor a
product created by HiClass.  It will be scanned and all the meta
data parameters found in its datasets will be added to the
dictionary.

Say you want to export a product p:
>>> dico = MyFitsDictionary(p)
>>> archive = FitsArchive()
>>> archive.rules.append(dico)
>>> archive.save(sFileName, p)

"""
```

```
def __init__(self, p):
    """p: HiClass product."""
    AbstractFitsDictionary.__init__(self)
    self._addKeysForProduct(p)
def _addKeysForProduct(self, prod):
    map(self._addKeysForDataset, map(prod.get, prod.keySet()))
def _addKeysForDataset(self, ds):
    for meta_name in ds.meta.keySet():
        self.set(meta_name, meta_name)

# This routine checks for a meta data parameter and if it doesn't exist,
# sets a default.
def checkForMeta(spectrum,metaName,metaDefault) :
if spectrum.meta.containsKey(metaName):
 mdata = spectrum.meta[metaName]
else :
 class_name= metaDefault.__class__.__name__
 print metaName, "tag not found in spectrum. Setting to default"
 if class_name.endswith('Float') :
   mdata = DoubleParameter(Double(metaDefault))
 elif class_name.endswith('Double') :
   mdata = DoubleParameter(metaDefault)
 elif class_name.endswith('String') :
   mdata = StringParameter(metaDefault)
 elif class_name.endswith('Long') :
   mdata = LongParameter(metaDefault)
return mdata

# the main routine:
def spectrum1dToClass(spectrum,fitsfn):
# ensure that the spectrum is a 1d.
class_name = spectrum.__class__.__name__
if class_name.endswith('Spectrum1d'):
 print "Converting input spectrum"
else :
 print "Input is not a Spectrum1d, exiting."
 return -1

p=Product(description = 'Herschel HIFI', \
        instrument = 'HIFI', \
        creator = 'spectrum1dToClass')
p.type = 'Class formatted fits file'

sFlux=spectrum.getFlux()
sWave=spectrum.getWave()*1e6 # converts to Hz, assumes data is in MHz
n_channels=sFlux.length()

# compute frequency parameters
# this computes a scale, but needs a lot of error checking
# -assumes data has no NANs and is ordered, etc.
# works fine for decon output but may crash on types of 1d.
sIndex = Double1d.range(len(sWave))
fitter = Fitter(sIndex, PolynomialModel(1)) # Degree 1: y = ax+b.
result = fitter.fit(sWave)
freqSpacing=result[1]
freqStart=result[0]
# Irregularity. not used for now.
diff = (sIndex * freqSpacing + freqStart)-sWave
irregularity = STDDEV(diff)



blankingvalue=-1000

# here is where we set all the meta data CLASS fits needs.
meta= MetaData()
#
# Axis dimensions.
# -----------------
meta['MAXIS' ] = LongParameter(4, "Number of axis")
meta['MAXIS1'] = LongParameter(n_channels, "Max nb of channels in spectrum")
meta['MAXIS2'] = LongParameter(1, "Position coordinate 1 scale")
```

```
meta['MAXIS3'] = LongParameter(1, "Position coordinate 2 scale")
meta['MAXIS4'] = LongParameter(1, "Stokes parameters")
#
# Axis 1: Frequency.
# -------------------
# CLASS understands FREQ, FREQUENCY, LAMBDA, WAVELENGTH.
meta['CTYPE1'] = StringParameter("FREQ", "Frequency scale parameters")
meta['CRVAL1'] = DoubleParameter(freqStart, \
                 "Frequency offset @ reference channel")
meta['CDELT1'] = DoubleParameter(freqSpacing, \
                "Freq step, fres, channel width.")
meta['CRPIX1'] = LongParameter(0, "Number of the reference channel")
#
# Axis 2: Right ascention.
# -------------------------
# CLASS understands RA--, RA ; DEC-, DEC ; GLON ; GLAT ; TIME, UT.
# For the projection system, CLASS understands
#     : P_NONE      = 0 ! Unprojected data
# -TAN: P_GNOMONIC  = 1 ! Radial Tangent plane
# -SIN: P_ORTHO     = 2 ! Dixon Tangent plane
# -ARC: P_AZIMUTHAL = 3 ! Schmidt Tangent plane
# -STG: P_STEREO    = 4 ! Stereographic
# lamb: P_LAMBERT   = 5 ! Lambert equal area
# -ATF: P_AITOFF    = 6 ! Aitoff equal area
# -GLS: P_RADIO     = 7 ! Classic Single dish radio mapping
# Read Representations of celestial coordinates in FITS
# Authors: Mark R. Calabretta, Eric W. Greisen
# (Submitted on 19 Jul 2002)
# arXiv:astro-ph/0207413v1
# http://arxiv.org/abs/astro-ph/0207413
#
# Be careful:
# RA becomes  RA---GLS with three hyphens
# Dec becomes DEC--GLS with two hyphens.
# That's why we can also write 'RA--' and 'DEC-'
# for 'RA' and 'DEC': it's just easier to add the
# projection code after that.
#
proj = '-GLS'
#

meta['CTYPE2'] = StringParameter('RA--' + proj)
meta['CRVAL2'] = checkForMeta(spectrum,"raNominal",0.0)
meta['CDELT2'] = DoubleParameter(0.0)
meta['CRPIX2'] = DoubleParameter(0.0)
#
# Axis 3: Declination.
# ---------------------
meta['CTYPE3'] = StringParameter('DEC-' + proj)
meta['CRVAL3'] = checkForMeta(spectrum,"decNominal",0.0)
meta['CDELT3'] = DoubleParameter(0.0)
meta['CRPIX3'] = DoubleParameter(0.0)
#
# Axis 4: Stokes.
# -----------------
meta['CTYPE4'] = StringParameter('STOKES')
meta['CRVAL4'] = DoubleParameter(1.0)
meta['CDELT4'] = DoubleParameter(0.0)
meta['CRPIX4'] = DoubleParameter(0.0)
#
# Misc. information.
# -------------------
meta['EQUINOX']  = checkForMeta(spectrum,"equinox",0.0)
meta['BLANK']    = LongParameter(blankingvalue, "Marker of invalid channels")
meta['DATE-RED'] = DateParameter(FineTime(Date()),"Creation date of this file")
meta['PVEL-LSR'] = DoubleParameter(0.0,"source velocity")
meta['PVELTYPE'] = StringParameter('radio','source velocity type')
meta['TELESCOP'] = StringParameter('Herschel-HIFI-WBS','source of data')
meta['SCAN']     = LongParameter(1)
meta['SUBSCAN']  = LongParameter(1)
meta['OBJECT']   = checkForMeta(spectrum,"object",'Unknown object')
meta['MOLECULE'] = StringParameter('Unknown molecule','Molecule name')
```

```
meta['LINE']     = StringParameter('Unknown line','Line name')
meta['EXPOSURE'] = checkForMeta(spectrum,"exposure",0.0)
meta['TSYS']     = checkForMeta(spectrum,"Tsys",0.0)
meta['RESTFREQ'] = DoubleParameter(freqStart,'')
meta['IMAGFREQ'] = DoubleParameter(freqStart,'')
meta['BEAMEFF']  = checkForMeta(spectrum,"beff",1.0)
meta['PRESSURE'] = DoubleParameter(0.0,'Atmospheric pressure')
meta['TOUTSIDE'] = DoubleParameter(0.0,'Atmospheric temperature')

# convert the 1d flux into a 2d array for CLASS.
sArray=Double2d(1,sFlux.length())
sArray[0,:]=sFlux
# format the data into a table dataset, and tag it with our metadata
sData=TableDataset()
sData["DATA"]=Column(data=sArray,description="The spectrum",unit=Frequency.HERTZ)
sData.meta=meta
# insert the data into our product, and convert metadata keywords into FITS
 compliant text.
p["data"]=sData
keyDictionary=MyFitsDictionary(p)
for meta_name in meta.keySet():
 keyDictionary.set(meta_name, meta_name)

# save the output.
fits=FitsArchive()
fits.rules.append(keyDictionary)
fits.save(fitsfn, p)

# example usage:
# spectrum1dToClass(mySpectrum1d,'myClassOutput.fits')
```

Example 1.39. Complete example to convert a Spectrum1d class to a CLASS FITS file.

1.16.7.3. SAOImage DS9

Choose *File → Open* to open FITS files of Herschel images and cubes.

Note also that you can exchange data between HIPE and SAOImage DS9 via the Virtual Observatory SAMP protocol. See Section 1.17 for more information.

1.17. Working with the VO (External Tools)

The Virtual Observatory is a set of technologies allowing, among other things, the integration of different data analysis applications. You can view and manipulate data in one application (such as HIPE), send it to another application with the click of a button, view and manipulate the data there, and send it back to the original application. HIPE supports this using SAMP, *Simple Application Message Protocol* .

SAMP works using a *message hub* , a very light-weight piece of software that coordinates data exchange among VO-aware applications.

All ESA archives are VO-aware already, but access to VO-aware archives in HIPE is not available yet. Aladin or VOSpec (please open this link in a new tab or window) already provide an interface to many data sources, such as the ESA archives, including ISO. So it is possible to access the ESA archives by retrieving the data using Aladin and sending it to HIPE from there.

1.17.1. Sending products from HIPE to external tools

To send a data product to a VO-enabled application, follow these steps:

1. If the application is not one of *SAOImage DS9* , *Topcat* , *Aladin* or *VOSpec* , start it manually. If the application is one of those listed, HIPE will start it automatically when sending the data.

You can start some VO-enabled applications directly from HIPE. Click the *External Tools* icon in the HIPE Welcome page to display the list. Click an icon to launch the corresponding application.

2. Select the product in the *Variables* view.

3. Do either of the following:

    - From *Tools → Interoperability → Send Data to* , select the application to which you want to send your product.

    - Right click on the product in the *Variables* view and from the *Send to* menu choose the application to which you want to send your product.

The product appears in the chosen application.

To return the data to HIPE, send them from the other application. Refer to the documentation of the external application for instructions.

Troubleshooting.

- Data exchange is possible only if there is an overlap between the VO interfaces supported by HIPE and the other application. If the applications have no supported interface in common, no data can be exchanged. This is indicated by the external application name being greyed out in the *Send Data to* menu.

- There is no VO protocol to exchange whole data cubes between application. You must instead extract and send single images or spectra.

- If the application is not listed in the *Send Data to* menu at all, make sure that the application is connected to SAMP.

- VOSpec may be listed twice in the menu as *VOSpec* and *VOSpec (2, not supported)* . This is due to a bug in version 6.5.p2 of VOSpec. Use the *VOSpec* entry and ignore the other one.

- If HIPE cannot find *SAOImage DS9* on your system, choose *Edit → Preferences* and go to *External Tools* , where you can specify where the application is installed.

- HIPE connects to the VO automatically at startup. If the icon at the bottom right corner of HIPE is white ( ) instead of yellow ( ) it means that HIPE has disconnected for some reason. Choose *Tools → Interoperability → Connect to the VO* to connect again.

- When you select an application from the *Send Data to* menu and that application is downloaded via Java WebStart, you may need to send the data a second time after the application has started.

- If none of the above points solves your problem, you may have found a bug in the software. Please raise a Helpdesk ticket.

Choosing *Tools → Interoperability → SAMP Hub Status* opens the SAMP Hub Monitor:

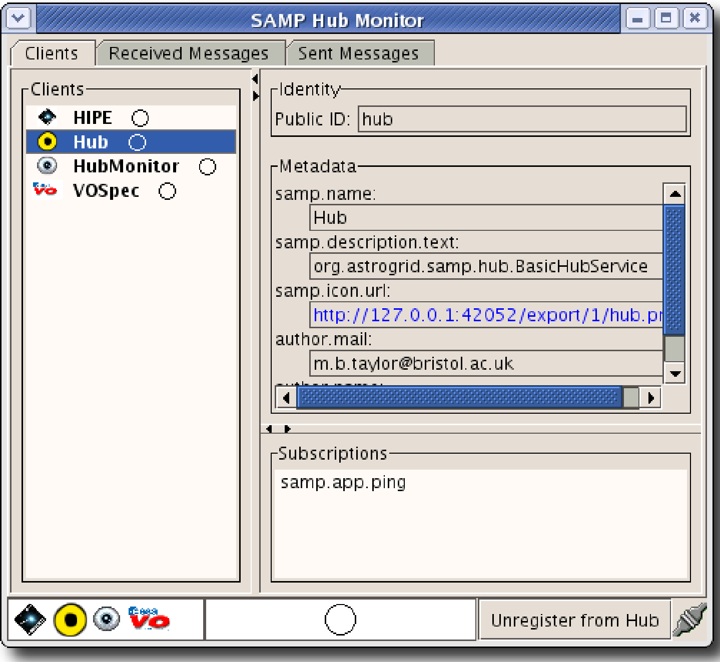


Figure 1.38. The SAMP Hub Monitor window.

Here you can find information about the client applications connected to the hub and the messages sent and received by each application. You should not have to look at this window other than for debugging purposes.

1.17.2. Sending products from external tools to HIPE

To return the data to HIPE, send them from the other application. Refer to the documentation of the external application for instructions.

Caveats. Sending data from external applications to HIPE has the following limitations:

- It is not possible to send multiple planes at once from Aladin to HIPE.
- It is not possible to send HIFI OTF maps back to HIPE from Aladin.
- When sending a table dataset to Topcat, units are not treated correctly. Degrees and arcminutes are converted to radians, while other units are ignored.

1.17.3. Opening VO Tables from HIPE

You can open VO table data in XML form without resorting to SAMP interoperability. This type of VO Table file is returned from the execution of TAP queries on services like VizieR or Simbad, or using tools like Topcat and Aladin. It can be identified by the root tag `<VOTABLE>` or the XML declaration `<?xml` at the beginning of the file. To do that, you make use of the task `voToTable`. This task takes two parameters, `data` is the path to the VO table XML file and `name` is the *mandatory* name for the output `TableDataset`. The `path` parameter can be either an absolute path or relative to HIPE installation and the XML file could even be packaged in a JAR file.

```
table1 = voToTable(data='votable.xml', name='Dubhe')
```

Example 1.40. Converting a VO table in XML format to TableDataset.

The output is a regular instance of `TableDataset` containing all the info from the VO table XML file.

1.17.4. Writing tables to files in VO-table XML format

To write a `TableDataset` instance to a file in VO-table XML format, there is a task called `tableToVo` which takes as arguments the file name and the `TableDataset` to write. You can use the following two examples as reference:

```
# Taking a dataset from an auxiliary product - happens to be a EventsLogDataset in
 this case.
obs = getObservation(obsid=1342265972, useHsa=True)
data = obs.refs["auxiliary"].product.refs["EventsLogProduct"].product["16_2"]
from tempfile import NamedTemporaryFile
outfile = NamedTemporaryFile(delete=False)
# The task requires a specific file extension (.xml, .vot or .votable)
outfile.name += ".xml"
tableToVo(file=outfile.name, table=data)
```

Example 1.41. Writing a TableDataset from an observation to an XML-based VO file.

```
# Constructing a simple TableDataset from scratch
t = TableDataset()
t["First"] = Column(Double1d(10))
t["Second"] = Column(Double1d(10))

from tempfile import NamedTemporaryFile
outfile = NamedTemporaryFile(delete=False)
# The task requires a specific file extension (.xml, .vot or .votable)
outfile.name += ".xml"
tableToVo(file=outfile.name, table=t)
```

Example 1.42. Writing a synthetic TableDataset to an XML-based VO file.

Chapter 2. Saving data as text files

This chapter covers the reading and writing of tabular data from text (ASCII) files. The first section lays out some considerations for users working with text files, and explains some of the concepts and terms that apply to handling ASCII data in HIPE. This is followed by several sections of "worked examples" with data formats you may typically encounter. The remaining sections each address a particular task that you may want to accomplish.

You can choose to first go to the "worked examples" as a quick start to working with the ASCII I/O tasks in HIPE, or to jump to individual task-based sections of interest, or to read straight through the chapter from beginning to end.

2.1. Considerations and concepts for working with text files

Points to consider about ASCII I/O.

- **The best ASCII format to use with HIPE is CSV.** HIPE is prepared to automatically open comma-separated values (CSV) files with the `.csv` file extension, when double-clicking them in the *Navigator* view. Files with delimiter characters usually require less effort to parse than blank space separated files, which can require fine-tuning and configuration of the parser class. See Section 2.14 and the sections after that one.
- **FITS files are often a better exchange format.** Products in HIPE are easily exported to FITS files, which are easily read back into HIPE with metadata and history preserved. There is no general way to save a data product or a product context to text files, aside from the Spectrum products. If you must save a product to file, save it into FITS format. See Section 1.16.1 for more details.
- **The ASCII I/O tasks work, in general, with table datasets.** Table datasets are by far the most common data structure for Herschel data. Any Herschel data product is ultimately a collection of table datasets. There are dedicated HIPE tasks to exchange data in table dataset form with text files. See the next sections for details.

  For more information on table datasets, see the *Scripting Guide*: Section 2.4 in *Scripting Guide*.

  Tip

  You can save table datasets directly to FITS format. This is the recommended way to save table datasets to file. See Section 1.16.1 for more details.
- **The ASCII I/O tasks are tools that often require manual configuration.** Aside from a few automatically-supported formats, the tasks require some setup in order to handle all cases of data in text files. To set up all column information in a table dataset such as name, unit, type and description, typically you will have to perform some configuration on the command-line.

  The ASCII I/O tasks do *not* automatically detect the format of the data in a text file, with the exception of certain `.csv` (comma-separated-values) and `.tbl` (space-separated) files.

  Tip

  In the *Navigator* view of HIPE, you can double-click on files ending in `.csv` or `.tbl`, to read these in as, respectively, comma-separated-value or space-separated tables.
- **Spectra have their own dedicated task for writing to text files.** There is a dedicated `exportSpectrumToAscii` task for exporting spectra to text files. This task accepts as input all the most common data types describing spectra in HIPE, including `Spectrum1d`, `Spectrum2d` and `SpectralSimpleCube`. An example of using the `exportSpectrumToAscii` task is given in Section 2.2. For more information on the `exportSpectrumToAscii` task, see Section 2.12.

- **Jython in HIPE includes a rich set of functionality for handling text files.** There are different ways to exchange data with text files, depending on the type of data you want to exchange:

  - **Jython lists, tuples and dictionaries.** You can write these data structures to file using Jython commands, as explained in the *Scripting Guide*: Section 1.25 in *Scripting Guide*.

    Note that Herschel data is never distributed as plain Jython data structure, so it is unlikely you will have to write them to file.

    For more information on lists, dictionaries and tuples, see the *Scripting Guide*: Section 1.10 in *Scripting Guide*.

  - **Numeric arrays, such as Double1d.** You can wrap 1-dimensional numeric arrays into a table dataset and write the table dataset to file, as explained later in this chapter. Assuming you have a `Double1d` array called `myArray`, this is how you create a table dataset containing it:

```
myTableDataset = TableDataset()
myTableDataset["myColumn"] = Column(myArray)
```

    Example 2.1. Creating a TableDataset with a Column made up of array data.

    Numeric arrays may be written to a file using the **print** statement. Consider two `Double1d` arrays named `wavelength` and `flux` with equal lengths:

```
wavelength = Double1d(5, 1.0)
flux = Double1d(5, 1.0)
fh = open('myspectrum.txt','w')
for i in range(len(wavelength)):
    print >> fh, '%13.6f %13.6f'% (wavelength[i], flux[i])
fh.close()
```

    Example 2.2. Read a numeric array from a file and loop over its values.

    For more information about formatting strings and printing to file, see the *Scripting Guide*, Section 1.23 in *Scripting Guide* and Section 1.8 in *Scripting Guide* respectively.

    You can read back the values as follows:

```
fh = open('myspectrum.txt')
lines = fh.readlines()
wave = Double1d()
fl = Double1d()
for line in lines:
    lsplit = line.split()
    wave.append(float(lsplit[0]))
    fl.append(float(lsplit[1]))
fh.close()
```

    Example 2.3. Read a numeric array from a file and tokenise its values in a loop.

    For more information on Numeric arrays, see the *Scripting Guide*: Section 2.2 in *Scripting Guide*.

Concepts in working with the ASCII I/O tasks. There are several concepts and terms that you need to know to work with the full functionality of the ASCII I/O tasks.

- **Parsers.** A *parser* defines rules to read a text file into HIPE. See Section 2.26 for the available types of parser, their features and how to configure them.

- **Formatters.** A *formatter* defines rules to write data from HIPE into a text file. See Section 2.27 for the available types of formatter, their features and how to configure them.

- **Table templates.** A *table template* describes the data to be read from, or written to, a text file. It defines the number of columns in the file, their name, the type and description of the data. While the parser defines general formatting rules, such as the character used to separate data values, the

table template describes the data themselves. See Section 2.25 for how to create and configure a table template.

- **Configuration files.** You can use a *configuration file* to store a particular configuration of the tasks for reading and writing text files. You can then load the configuration file for subsequent executions of the task. See Section 2.18 and Section 2.23 for instructions.
- **Delimiters.** A *delimiter* is a character that denotes a boundary between fields in a text file. The most common delimiter is a comma. For more information on specifying delimiters, see Section 2.17.
- **Regular expressions.** A *regular expression* is a concise and flexible means to match strings of text, such as particular characters or patterns of characters. Regular expressions are used to specify which lines of a file to skip, as discussed in Section 2.15, and with the `RegexParser` for specifying the delimiter between data fields (for example, to specify multiple spaces or tabs). A discussion of regular expressions is outside the scope of this manual, but Section 2.28 contains a few examples.

2.2. Worked example: Saving a Spectrum product as a text file

Use the `exportSpectrumToAscii` task for exporting spectra to text files. This task accepts as input all the most common data types describing spectra in HIPE, including `Spectrum1d`, `Spectrum2d` and `SpectralSimpleCube`. This section explains how to output a `SpectralSimpleCube` to the default comma-separated-value format, and how to read it back into a table dataset.

Warning

The `exportSpectrumToAscii` task does not write out all of the columns of the spectrum -it only writes the standard `wave` , `flux` and `weight` columns. This is particularly important for SPIRE and PACS data because the `error` column is not written out. However, the errors can be written as weights, if weight is set to `True` in the task, where the weights are calculated as `1/error^2`. To ensure that all the data are written to file, please use the `asciiTableWriter` via a script or the GUI, e.g. by right clicking the variable name in the Variables view and choosing *Send To -> Text file*.

For more information on the `exportSpectrumToAscii` task, see Section 2.12.

Command-line script. The script in Example 2.4 retrieves a HIFI cube from the archive, exports it to an ASCII file in your home directory, and then reads it back into a table dataset.

```
# Worked example for exporting a Spectrum object to the default
# ascii format

# Retrieve a cube from the archive
obs = getObservation(obsid=1342210097, useHsa=True)
HifiCube=obs.level2_5.refs["cubesContext"].product.refs["cubesContext_WBS-H-USB"]. \
  product.refs["cube_WBS_H_USB_1"].product

print HifiCube.class
# prints <type 'herschel.ia.dataset.spectrum.SpectralSimpleCube'>

# Set up the output file
from herschel.share.io import FileUtil
myFile = java.io.File(FileUtil.USER_HOME,'HifiCube.txt')
# The previous line creates the text file in your home directory. If you
# want to create the file in another directory, you can write the full
# path plus the file name within quotes, for example:
# myFile = java.io.File('/my/custom/path/HifiCube.txt')

# Export the Spectrum object to an ASCII file
# myFile.absolutePath returns the directory path you chose
# for the file, so you do not need to write it again.
```

```
exportSpectrumToAscii(ds=HifiCube, file=myFile.absolutePath)

# Now read it back in
HifiCubeTxt = asciiTableReader(file=myFile.absolutePath,\
    ignoreWarn=False)
```

Example 2.4. Script to export a SpectralSimpleCube to ASCII, and read back into a TableDataset

Output text file format. The script in Example 2.4 outputs the cube into a `HifiCube.txt` in your home directory. An excerpt of the output file is shown in Figure 2.1.

```
# Meta data {
#  type={description="Product Type Identification", string="herschel.ia.dataset.spectrum.SpectralSimpleCube"}
#  creator={description="Generator of this product", string="SPG v6.1.0"}
#  creationDate={description="Creation date of this product", time="2011-06-03T18:03:31.630000Z"}
#  description={description="Name of this product", string="HIFI cube product"}
#  instrument={description="", string="HIFI"}
#  modelName={description="Model name attached to this product", string="FLIGHT"}
#  startDate={description="Start date of this product", time="2011-06-03T18:03:31.630000Z"}
#  endDate={description="End date of this product", time="2011-06-03T18:03:31.630000Z"}
#  formatVersion={description="Version of the product format", string="0.1"}
#  wavedescription={description="Description of WaveColumn", string="Upper sideband frequency"}
#  wavename={description="Actual name of the WaveColumn", string="usbfrequency"}
#  waveunit={description="Units of the WaveColumn", string="GHz"}
#  sideband={description="upper or lower subband", string="usb"}
#  obsid={description="Observation id", long=1342210097, unit=null}
#  apid={description="Apid", long=1030, unit=null}
#  band={description="Active band", string="1b"}

#  nodCycleNum={description="", long=0, unit=null}
#  nyquistSampling={description="Map is Nyquist sampled", boolean=false}
#  loFrequency={description="The LO frequency of the source phase", double=571.57425, unit=GHz [1.0E9 Hz]}
#  version={description="Version of the product", string="1.0"}
# } MetaData
usbfrequency_0,flux_0,usbfrequency_1,flux_1,usbfrequency_2,flux_2,usbfrequency_3,flux_3,usbfrequency_4,flux_4,usbfre
Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Doub
GHz,K,GHz,K,GHz,K,GHz,K,GHz,K,GHz,K,GHz,K,GHz,K,GHz,K,GHz,K,GHz,K,GHz,K,GHz,K,GHz,K,GHz,K,GHz,K,Gl
usbfrequency,Cube,usbfrequency,Cube,usbfrequency,Cube,usbfrequency,Cube,usbfrequency,Cube,usbfreq
575.462,0.0411261859126850 1,575.462,0.10520306580112145,575.462,-0.1502299661978008,575.462,-0.050482533342963744,5
575.4625,0.025595185850949952,575.4625,0.07780529651840655,575.4625,-0.09669915832484992,575.4625,-0.031018764915009
575.463,-0.03342912024189695,575.463,0.07681287608907848,575.463,-0.0571473950687211,575.463,-0.0255590103491692,575
575.4635,-0.01299159526105395,575.4635,0.08016574995398647,575.4635,-0.0775304795479722,575.4635,-0.013916408912834
575.4639999999999,0.04983664554632733,575.4639999999999,0.06951254002702104,575.4639999999999,-0.06807212931402987,57
575.4645,0.08542963690987823,575.4645,0.05009852793523031 4,575.4645,-0.04115307966373852 5,575.4645,-0.027360306553
575.465,0.07360169324169391,575.465,0.03148626605180956,575.465,-0.05795629580359649,575.465,-0.04333819637611109,57
```

Figure 2.1. Excerpt from the output of exportSpectrumToAscii

Each "metadata" or "comment" line begins with a # character. The rest of the file is in the 'HIPE-standard' comma-separated-value format with a header of four lines.

Re-read table in HIPE. The last line of the script in Example 2.4 reads the exported spectrum into a table dataset named `HifiCubeTxt`. A view of this table dataset as rendered by the Dataset Viewer in HIPE is shown in Figure 2.2. Note that the imported object is not a `SpectralSimpleCube`. To recover the original cube, it would be necessary to output it as a FITS file and then to import it to HIPE. Note also that the metadata are not recovered by `asciiTableReader`.

Meta Data

None

Table Data

Index	Show or hide the table data	flux_0 [K]	usbfrequency_1 [GHz]	flux_1 [K]	usbfrequency_2 [GHz]	
0	575.462	0.04112618591268501	575.462	0.10520306580112145	575.462	-0.1502
1	575.4625	0.025595185850949952	575.4625	0.07780529651840655	575.4625	-0.0966
2	575.463	-0.03342912024189695	575.463	0.07681287608907848	575.463	-0.0571
3	575.4635	-0.01299159526105395	575.4635	0.08016574995398647	575.4635	-0.0775
4	575.4639999999999	0.04983664554632733	575.4639999999999	0.06951254002702104	575.4639999999999	-0.0680
5	575.4645	0.08542963690987823	575.4645	0.050098527935230314	575.4645	-0.0411
6	575.465	0.07360169324169391	575.465	0.03148626605180956	575.465	-0.0579
7	575.4655	0.07724118843196827	575.4655	0.014592034109124533	575.4655	-0.1317
8	575.466	0.05371095614577734	575.466	0.04331268471028663	575.466	-0.0925
9	575.4665	0.08764682758578542	575.4665	0.08714788449540875	575.4665	-0.0256
10	575.467	0.06291357322857152	575.467	0.054768505827582345	575.467	-0.0568
11	575.4675	0.01619720456258012	575.4675	0.01717077990540213	575.4675	-0.1083
12	575.468	0.0084481180273505	575.468	0.0207430928669927	575.468	-0.1168
13	575.4685	0.05696703224742	575.4685	0.06630567363005221	575.4685	-0.0918
14	575.4689999999999	0.1258823471360696	575.4689999999999	0.11276205194654178	575.4689999999999	-0.0817
15	575.4695	0.14390811630226816	575.4695	0.09644839882185358	575.4695	-0.0865

Figure 2.2. The TableDataset resulting from running the example script

2.3. Worked example: Saving a SourceList-Product as a text file

The `SourceListProduct` is a catalogue of sources extracted from PACS or SPIRE maps by the source extraction tasks in HIPE. This section describes how to write a `SourceListProduct` to HIPE, then how to read it back in as a `TableDataset`. This example further shows how to reformat the `TableDataset` back into the form of a `SourceListProduct`.

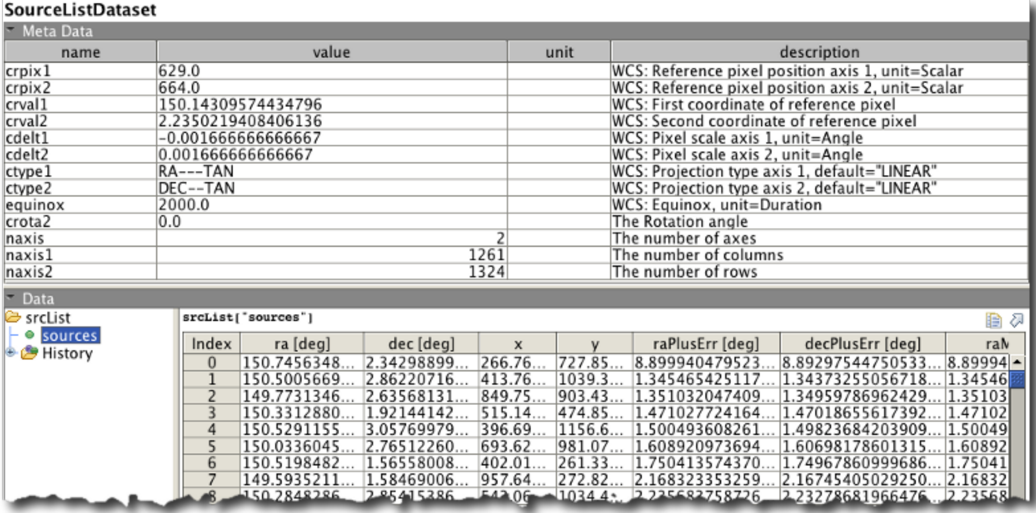
Parts of this example can be used to reformat an external catalogue into a `SourceListProduct`. You can use such a product as input to the HIPE source extractors, or drag and drop it onto a displayed image in HIPE to mark the source positions.

Command-line script. The script in Example 2.5 will generate a `SourceListProduct` from a map retrieved from the archive, write it to a text file in a temporary directory, read the table back into HIPE and reformat it as a `SourceListProduct`.

```
# Worked example for outputting a SourceListProduct and
#  reading it back into HIPE

# Retrieve a SPIRE map from the archive
obs = getObservation(obsid=1342222849, useHsa=True)
myMap = obs.level2.refs['psrcPSW'].product

# Run sourceExtractorSussextractor to create source list
srcList = sourceExtractorSussextractor(image=myMap,\
    detThreshold=5.0, fwhm=17.5)

# Get the temporary directory
import os
temp_dir = Configuration.getProperty('var.hcss.workdir')

# Show that the SourceListProduct can be written to FITS
#  Note this preserves the metadata in the SourceListProduct
simpleFitsWriter(product=srcList, file=\
    os.path.join(temp_dir,'srcList.fits'))

# Output the "sources" table to ASCII
#  Note: in Outline view, you can right-click on the
#  "sources" TableDataset and then Send to -> Text file
asciiTableWriter(table=srcList["sources"], \
    file=os.path.join(temp_dir,'srcList.txt'))

# Read the text file back into HIPE
newSrcTable = asciiTableReader(file=\
    os.path.join(temp_dir,'srcList.txt'), \
    ignoreWarn=False)

# Make the TableDataset into a sourceListDataset
newSourceTable = SourceListDataset(newSrcTable)

# Insert the SourceListDataset into a SourceListProduct
#    and update the wcs
newSourceList = SourceListProduct(newSourceTable)
newSourceList.setWcs(myMap.wcs)
```

Example 2.5. Script to generate a SourceListProduct, write it to a text file, and read it back into HIPE

The initial SourceListProduct. The script in Example 2.5 generates a `SourceListProduct` named `srcList`. The format of this list is shown in Figure 2.3.



SourceListDataset

Meta Data

name	value	unit	description
crpix1	629.0		WCS: Reference pixel position axis 1, unit=Scalar
crpix2	664.0		WCS: Reference pixel position axis 2, unit=Scalar
crval1	150.14309574434796		WCS: First coordinate of reference pixel
crval2	2.2350219408406136		WCS: Second coordinate of reference pixel
cdelt1	-0.001666666666667		WCS: Pixel scale axis 1, unit=Angle
cdelt2	0.001666666666667		WCS: Pixel scale axis 2, unit=Angle
ctype1	RA---TAN		WCS: Projection type axis 1, default="LINEAR"
ctype2	DEC--TAN		WCS: Projection type axis 2, default="LINEAR"
equinox	2000.0		WCS: Equinox, unit=Duration
crota2	0.0		The Rotation angle
naxis	2		The number of axes
naxis1	1261		The number of columns
naxis2	1324		The number of rows

Data

srcList
sources
History

srcList["sources"]

Index	ra [deg]	dec [deg]	x	y	raPlusErr [deg]	decPlusErr [deg]	raM
0	150.7456348...	2.34298899...	266.76...	727.85...	8.899940479523...	8.89297544750533...	8.89994
1	150.5005669...	2.86220716...	413.76...	1039.3...	1.345465425117...	1.34373255056718...	1.34546
2	149.7731346...	2.63568131...	849.75...	903.43...	1.351032047409...	1.34959786962429...	1.35103
3	150.3312880...	1.92144142...	515.14...	474.85...	1.471027724164...	1.47018655617392...	1.47102
4	150.5291155...	3.05769979...	396.69...	1156.6...	1.500493608261...	1.49823684203909...	1.50049
5	150.0336045...	2.76512260...	693.62...	981.07...	1.608920973694...	1.60698178601315...	1.60892
6	150.5198482...	1.56558008...	402.01...	261.33...	1.750413574370...	1.74967860999686...	1.75041
7	149.5935211...	1.58469006...	957.64...	272.82...	2.168323353259...	2.16745405029250...	2.16832

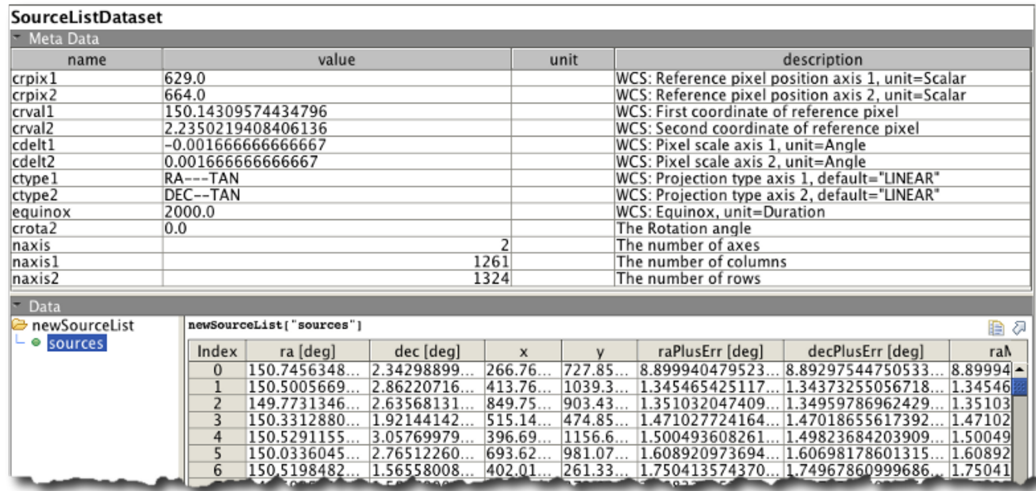
Figure 2.3. The initial SourceListProduct generated by sourceExtractorSussextractor.

Output text file format. The middle lines in the script in Example 2.5 output the `SourceList-Product` into a temporary directory, as a FITS file and as a text file. An excerpt of the text file is shown in Figure 2.4. This default format is comma-separated values with a four-line header as described in Section 2.10.

```
ra,dec,x,y,raPlusErr,decPlusErr,raMinusErr,decMinusErr,xPlusErr,yPlusErr,xMinusErr,yMinusErr,flux,fluxPlusErr,fluxMinusErr,background,bgPlusErr,bgM
Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double,Double
deg,deg,,,deg,deg,deg,deg,,,,,mJy,mJy,mJy,Jy/beam,Jy/beam,Jy/beam,
,,,,,,,,,,,,,,,,,,
```

Figure 2.4. Excerpt from the text file written from the SourceListProduct.

Re-read table in HIPE. The last section of the script in Example 2.5 reads the exported source list into a table dataset named `newSrcTable`. Note that the imported object is not a `SourceListProduct`. The rest of the script shows how to convert the imported table dataset into a `SourceListProduct`. A view of this table dataset as rendered by the Dataset Viewer in HIPE is shown in Figure 2.5.



SourceListDataset

Meta Data

name	value	unit	description
crpix1	629.0		WCS: Reference pixel position axis 1, unit=Scalar
crpix2	664.0		WCS: Reference pixel position axis 2, unit=Scalar
crval1	150.14309574434796		WCS: First coordinate of reference pixel
crval2	2.2350219408406136		WCS: Second coordinate of reference pixel
cdelt1	-0.001666666666667		WCS: Pixel scale axis 1, unit=Angle
cdelt2	0.001666666666667		WCS: Pixel scale axis 2, unit=Angle
ctype1	RA---TAN		WCS: Projection type axis 1, default="LINEAR"
ctype2	DEC--TAN		WCS: Projection type axis 2, default="LINEAR"
equinox	2000.0		WCS: Equinox, unit=Duration
crota2	0.0		The Rotation angle
naxis	2		The number of axes
naxis1	1261		The number of columns
naxis2	1324		The number of rows

Data

newSourceList
sources

newSourceList["sources"]

Index	ra [deg]	dec [deg]	x	y	raPlusErr [deg]	decPlusErr [deg]	raM
0	150.7456348...	2.34298899...	266.76...	727.85...	8.899940479523...	8.89297544750533...	8.89994
1	150.5005669...	2.86220716...	413.76...	1039.3...	1.345465425117...	1.34373255056718...	1.34546
2	149.7731346...	2.63568131...	849.75...	903.43...	1.351032047409...	1.34959786962429...	1.35103
3	150.3312880...	1.92144142...	515.14...	474.85...	1.471027724164...	1.47018655617392...	1.47102
4	150.5291155...	3.05769979...	396.69...	1156.6...	1.500493608261...	1.49823684203909...	1.50049
5	150.0336045...	2.76512260...	693.62...	981.07...	1.608920973694...	1.60698178601315...	1.60892
6	150.5198482...	1.56558008...	402.01...	261.33...	1.750413574370...	1.74967860999686...	1.75041

Figure 2.5. The reconstituted SourceListProduct with data read in from the text file.

2.4. Worked example: Reading a Spitzer spectrum into a table dataset

The Spitzer Heritage Archive serves a variety of data products in an ASCII format known commonly as *IPAC Table Format*. In this section, the procedure to read in a Spitzer spectrum is shown, using the `RegexParser` and a table template.

Note

HIPE can now read files in the IPAC Table Format automatically. See Section 2.8 for more information. This example is left as an illustration of advanced usage of parsers and table templates.

Input file format. Spitzer Spectra ASCII table in fixed format. An excerpt is shown in Figure 2.6.

```
\char HISTORY PROCESS 2010/10/02 02:34:03
\char HISTORY irs_tune v3.4  (nl_and_params v1.9)
\char HISTORY INFILE = extract.tbl
\char HISTORY INFILE = cal/fluxcon.tbl
\char HISTORY UP_DOWN_MODE = tune_down
\char HISTORY APPLY = 3,
\char HISTORY NAMELIST_FILE = cdf/irs_tune.nl
|order  |wavelength   |flux_density   |error       |bit-flag |
|int    |real         |real           |real        |int      |
|       |             |Jy             |Jy          |         |
 1       7.45515       0.346469        NaN          12288
 1       7.51564       0.391154        0.003242     0
 1       7.57612       0.425596        0.003127     0
 1       7.63660       0.407751        0.002902     0
 1       7.69709       0.402711        0.002706     0
 1       7.75757       0.436215        0.002651     0
 1       7.81805       0.467682        0.002622     0
 1       7.87854       0.487097        0.002598     0
 1       7.93902       0.495642        0.002537     0
 1       7.99951       0.507413        0.002473     0
 1       8.05999       0.518590        0.002417     0
```

Figure 2.6. Excerpt from the a Spitzer spectrum product.

Each metadata or comment line begins with a backslash. The header information is delimited by the pipe symbol. The rest of the table can be treated as a space-separated file.

Command-line script. The following script downloads the online catalogue, writes it to a temporary file, sets up the template and the `RegexParser`, reads in the table, and deletes the input file.

```
# Worked example for reading in a Spitzer spectrum in "IPAC table" format
```

```
# Following is the contents of the table, which is excerpted
#  from a Spitzer spectrum 'SPITZER_S0_20925696_0005_5_E7559452_bksub.tbl'
contents= (
"\\ char HISTORY PROCESS 2010/10/02 02:34:03\n"
"\\ char HISTORY irs_tune v3.4  (nl_and_params v1.9)\n"
"\\ char HISTORY INFILE = extract.tbl\n"
"\\ char HISTORY INFILE = cal/fluxcon.tbl\n"
"\\ char HISTORY UP_DOWN_MODE = tune_down\n"
"\\ char HISTORY APPLY = 3,\n"
"\\ char HISTORY NAMELIST_FILE = cdf/irs_tune.nl\n"
"|order  |wavelength  |flux_density  |error      |bit-flag |\n"
"|int    |real        |real          |real       |int      |\n"
"|       |            |Jy            |Jy         |         |\n"
" 1       7.45515      0.346469       NaN        12288     \n"
" 1       7.51564      0.391154       0.003242    0         \n"
" 1       7.57612      0.425596       0.003127    0         \n"
" 1       7.63660      0.407751       0.002902    0         \n"
" 1       7.69709      0.402711       0.002706    0         \n"
" 1       7.75757      0.436215       0.002651    0         \n"
" 1       7.81805      0.467682       0.002622    0         \n"
" 1       7.87854      0.487097       0.002598    0         \n"
" 1       7.93902      0.495642       0.002537    0         \n"
" 1       7.99951      0.507413       0.002473    0         \n"
" 1       8.05999      0.518590       0.002417    0         \n"
" 1       8.12047      0.526655       0.002387    0         \n"
" 1       8.18096      0.536147       0.002327    0         \n"
" 1       8.24144      0.545094       0.002325    0         \n"
" 1       8.30192      0.547389       0.002331    0         \n"
" 1       8.36241      0.548689       0.002258    0         \n"
" 1       8.42289      0.546367       0.002166    0         \n"
" 1       8.48337      0.538944       0.002116    0         \n"
" 1       8.54386      0.526541       0.002110    0         \n"
" 1       8.60434      0.508472       0.002094    0         \n"
" 1       8.66483      0.486871       0.002067    0         \n"
" 1       8.72531      0.461781       0.001984    0         \n"
" 1       8.78579      0.437327       0.001933    0         \n"
" 1       8.84628      0.408632       0.001860    0         \n"
" 1       8.90676      0.381725       0.001845    0         \n"
" 1       8.96724      0.356162       0.001866    0         \n"
" 1       9.02773      0.326461       0.001852    0         \n"
" 1       9.08821      0.298850       0.001872    0         \n"
" 1       9.14869      0.268248       0.001751    0         \n"
" 1       9.20918      0.238048       0.001703    0         \n"
" 1       9.26966      0.208400       0.001683    0         \n"
" 1       9.33015      0.178549       0.001605    0         \n"
" 1       9.39063      0.150669       0.001579    0         \n"
" 1       9.45111      0.123782       0.001466    0         \n"
" 1       9.51160      0.105198       0.001351    0         \n"
" 1       9.57208      0.081897       0.001237    0         \n"
" 1       9.63256      0.066417       0.001172    0         \n"
" 1       9.69305      0.054389       0.001019    0         \n"
" 1       9.75353      0.043911       NaN        12288     \n"
" 1       9.81401      0.034835       NaN        12288     \n"
" 1       9.87450      0.029328       0.001027    0         \n"
" 1       9.93498      0.025912       0.000797    0         \n"
" 1       9.99547      0.021534       0.000759    0         \n"
)


# The next three lines merely set up the example to be read in
from herschel.share.io import FileUtil
myFile = java.io.File(FileUtil.TEMP_DIR,\
    'SPITZER_S0_20925696_0005_5_E7559452_bksub.tbl')
FileUtil.saveTextFile(myFile, contents)

# Set up the parser to ignore (1) lines beginning with backslash;
#  (2) lines beginning with the pipe symbol; (3) blank lines.
#  Set the delimiter to be one or more spaces.
myParser=RegexParser(ignore='^\\\\|^\||^\\s*$',delimiter='\\s+')

# Check the units that we will feed to the Table Template
```

```
# NOTE: Some HIPE core-only installation need this import
from herschel.share.unit import Unit
print Unit.parse("microns").isKnown()
# returns False

print Unit.parse("MICROMETERS").isKnown()
# returns True

print Unit.parse("Jy").isKnown()
# returns True

# Set up the Table Template
myTemplate = TableTemplate(5, \
  names = ["Order", "Wavelength", "Flux", "Error", "Bit-flag"], \
  types = ["Integer", "Double", "Double", "Double", "Integer"], \
  units = ["", "MICROMETERS", "Jy", "Jy", ""], \
  descriptions = ["Spectral order", "Wavelength", \
    "Flux density", "Error in flux density", "Flags"])

# Read in the table
table = asciiTableReader(file=myFile.absolutePath, \
      template = myTemplate,
      parser=myParser)

# Clean up by deleting the file
myFile.delete()
```

Example 2.6. Worked example for reading in a Spitzer spectrum in "IPAC table" format

Resulting table in HIPE. The result of running the previous script is a table dataset named `table`. A view of this table dataset as rendered by the Dataset Viewer in HIPE is shown in Figure 2.7.

Meta Data

None

Table Data

Index	Order	Wavelength [µm]	Flux [Jy]	Error [Jy]	Bit-flag
0	1	7.45515	0.346469	NaN	12288
1	1	7.51564	0.391154	0.003242	0
2	1	7.57612	0.425596	0.003127	0
3	1	7.6366	0.407751	0.002902	0
4	1	7.69709	0.402711	0.002706	0
5	1	7.75757	0.436215	0.002651	0
6	1	7.81805	0.467682	0.002622	0
7	1	7.87854	0.487097	0.002598	0
8	1	7.93902	0.495642	0.002537	0
9	1	7.99951	0.507413	0.002473	0
10	1	8.05999	0.51859	0.002417	0
11	1	8.12047	0.526655	0.002387	0
12	1	8.18096	0.536147	0.002327	0
13	1	8.24144	0.545094	0.002325	0
14	[illegible]	[illegible]	0.547389	0.002331	0

Figure 2.7. The table dataset resulting from running the example script.

2.5. Worked example: Reading a VizieR catalogue into a table dataset

The VizieR archive server (CDS, Strasbourg) hosts a number of astronomical catalogues in text form. This section shows the procedure to read in the ASCII format of the Planck Early Cold Cores Catalogue, using the `FixedWidthParser` and a table template.

Tip

The VizieR archive server can also output catalogues in FITS format, which are trivially read into HIPE.

Input file format. The Planck Early Cores Catalogue in text form is a large ASCII table in fixed format. An excerpt is shown in Figure 2.8.

```
PLCKECC G000.04-18.93  25.5  0.0439 -18.9380 286.7951 -37.2353    9555    40211    96489    21167    587    2105    6020    1706 13
PLCKECC G000.37-19.51 141.3  0.3735 -19.5108 287.6043 -37.1385   30830   113074   263666    25686   1011    4030   10095    2804 11
PLCKECC G000.48+11.39  69.5  0.4834  11.3961 256.1800 -22.2100   16042    66306   195003    30323   1203    4688   12793    4688 11
PLCKECC G000.50+00.00  42.1  0.5054   0.0000 266.7046 -28.5045  680888  3986022 14745516 11119693  69694  372951 1355505 4944452 13
PLCKECC G000.65-00.01 235.8  0.6592  -0.0186 266.8135 -28.3826 2314982 14194582 44525696 22178628 131463  778589 2799775 5314142 12
PLCKECC G000.92-20.16  52.9  0.9229 -20.1651 288.5931 -36.8710   14255    58385   155853    36750    816    3166    8462    2682 13
PLCKECC G001.23+09.91  15.0  1.2305   9.9158 257.9445 -22.4685    7117    25281    78119    23326    742    2914    7362    5174 13
PLCKECC G001.27+03.78  16.6  1.2744   3.7889 263.5491 -25.8457   13847    55186   114805       15   1879    7520   18858   11573  7
PLCKECC G001.31-20.48  55.4  1.3184 -20.4833 289.1209 -36.6260   11118    46278   117583    15751    767    2759    7189    2237 11
PLCKECC G001.38+20.94 119.4  1.3843  20.9420 248.6587 -15.7780   41198   164403   369258    82869    949    3814    9253    3291 13
PLCKECC G001.64-00.07  36.4  1.6479  -0.0746 267.4453 -27.5645  223312  1320999  3602984  2080002  33336  155494  546489  434191 13
PLCKECC G001.84+16.58 119.1  1.8457  16.5877 252.5540 -18.0729   31290   123324   271631    37698    774    2609    6956    4528 12
PLCKECC G001.95+09.78  46.1  1.9556   9.7833 258.5025 -21.9576   22952    87608   215217    42266   2291    8710   23634    6736 13
PLCKECC G003.07+09.95  68.0  3.0762   9.9537 259.0226 -20.9508   25645    98573   282199    70510   1082    4435   11325    4754 13
PLCKECC G003.14+08.19  18.5  3.1421   8.1971 260.6267 -21.8781   12710    52269   131823    36958    896    3825    9057    6117 13
PLCKECC G003.27+10.42  43.6  3.2739  10.4274 258.7214 -20.5230    7491    31144    81031    21162    261     911    2552    2501 13
PLCKECC G003.73+16.39  51.3  3.7354  16.3931 253.8409 -16.7265    8153    34374    75218    -5168   1449    5604   14933    1597  6
PLCKECC G003.73+18.30 112.6  3.7354  18.3082 252.2281 -15.6006   30061   107682   271852    55423   1588    6186   15815    7626 13
```

Figure 2.8. Excerpt from the Planck Early Cold Cores Catalogue.

The format is explained in a separate ReadMe file, shown in Figure 2.9.

```
Byte-by-byte Description of file: ecc.dat
--------------------------------------------------------------------------------
   Bytes Format Units   Label     Explanations
--------------------------------------------------------------------------------
   1-  7  A7     ---     ---       [PLCKECC] Planck Early Cold Cores Catalog
   9- 21  A13    ---     PLCKECC   PLCKECC name (GLLL.ll+BB.bb) (NAME)
  23- 27  F5.1   ---     SNR       Detection signal-to-noise ratio (SNR)
  29- 36  F8.4   deg     GLon      Galactic longitude  based on bandmerge algorithm
  38- 45  F8.4   deg     GLat      Galactic latitude  based on bandmerge algorithm
  47- 54  F8.4   deg     RAdeg     Right ascension (J2000) (RA)
  56- 63  F8.4   deg     DEdeg     Declination (J2000) (DEC)
  65- 73  I9     mJy     S353      Aperture flux density at 353GHz (APFLUX353)
  75- 83  I9     mJy     S545      Aperture flux density at 545GHz (APFLUX545)
  85- 93  I9     mJy     S857      Aperture flux density at 857GHz (APFLUX857)
  95-103  I9     mJy     S3000     Aperture flux density at 3000GHz (APFLUX3000)
 105-113  I9     mJy   e_S353      Uncertainty ({sigma}) on S353 (APFLUX353_ERR)
 115-123  I9     mJy   e_S545      Uncertainty ({sigma}) on S545 (APFLUX545_ERR)
 125-133  I9     mJy   e_S857      Uncertainty ({sigma}) on S857 (APFLUX857_ERR)
 135-143  I9     mJy   e_S3000     Uncertainty ({sigma}) on S3000 (APFLUX3000_ERR)
 145-150  F6.3   K       T         Temperature from greybody fit (TEMPERATURE) (1)
 152-157  F6.3   ---     beta      Emissivity index from greybody fit (BETA) (1)
 159-167  I9     mJy     gS857     Flux density at 857 GHz from greybody fit (S857)
 169-174  F6.3   K     e_T         Uncertainty ({sigma}) on T (TEMPERATURE_ERR)
```

Figure 2.9. Excerpt from ReadMe file for Planck Early Cold Cores Catalogue.

Command-line script. The sizes of the columns must be specified to the `FixedWidthParser`, and the `TableTemplate` must be set up, following the information in Figure 2.8.

The script in Example 2.7 downloads the online catalog, writes it to a temporary file, sets up the template and the `FixedWidthParser`, reads in the table, and deletes the input file.

```
# Worked example for Planck Early Cold Cores Catalogue

# Setup: grab the catalog (ecc.dat)

# The following five commented lines download a file from the Internet.
# If you have a file already on your hard disk, or if you download
# the file manually, load the file locally (see below).
```

```
# from herschel.share.io import FileUtil
# catFile = java.io.File('/path/to/ecc.dat')
# catUrl='http://cdsarc.u-strasbg.fr/ftp/cats/VIII/88/ecc.dat'
# destFile = java.io.File(FileUtil.TEMP_DIR,'ecc.dat')
# FileUtil.copyToFile(java.net.URL(catUrl),catFile)

# Load "ecc.dat" as a local resource (catFile)

# catFile = java.io.File('/path/to/ecc.dat')

# Set up the Table Template
myTemplate = TableTemplate(30, \
  names = ["cat", "name", "SNR", "GLon", "GLat", "RAdeg", "DEdeg",\
       "S353", "S545", "S857", "S3000", "e_S353", "e_S545", "e_S857", \
       "e_S3000","T", "beta", "gS857", "e_T", "e_beta", "e_gS857",\
       "fit", "T.c", "beta.c", "a.c", "b.c", "e_T.c", "e_beta.c",\
       "e_a.c", "e_b.c"], \
  types = 2*["String"] + 5*["Float"] + 8*["Integer"] + 2*["Float"] + \
       ["Integer", "Float", "Float", "Integer", "Integer"] + 8*["Float"], \
  units = 3*[""] + 4*["deg"] + 8*["mJy"] + ["K", "", "mJy", "K", "", "mJy", \
               "mJy^2", "K", "", "arcmin", "arcmin", "K", "", \
               "arcmin", "arcmin"])
# Descriptions could be added, but are omitted for brevity's sake


# Set up the parser
myParser=FixedWidthParser(sizes=[7, 14, 6] + 4*[9] + 8*[10] + 2*[7] + \
          [10, 7, 7, 10, 7, 7, 7, 6, 6, 7, 7, 6, 6])

# Read in the table
table = asciiTableReader(file=catFile.absolutePath, \
      template = myTemplate, parser=myParser)
# Here we used destFile.absolutePath to indicate the absolute path of
# the file we downloaded from the Internet at the beginning of the
# script.
# Most likely you have already a file on your hard disk, in which case
# you can just pass the path as a string to the file parameter:
# table = asciiTableReader(file="/home/user/myFile.dat", \
#       template = myTemplate, parser=myParser, delimiter=None)
```

Example 2.7. Complete script for reading in the Planck Early Cold Cores Catalogue.

Resulting table in HIPE. The result of running the script in Example 2.7 is a table dataset named `table`. An excerpt of this table dataset, displayed with the Dataset Viewer in HIPE, is shown in Figure 2.10.

Meta Data

None

Table Data

Index	cat	name	SNR	GLon [deg]	GLat [deg]	RAdeg [deg]	DEdeg [deg]	S353 [mJy]	S545 [mJy
0	PLCKECC	G000.04-18.93	25.5	0.0439	-18.938	286.7951	-37.2353	9555	402
1	PLCKECC	G000.37-19.51	141.3	0.3735	-19.5108	287.6043	-37.1385	30830	1130
2	PLCKECC	G000.48+11.39	69.5	0.4834	11.3961	256.18	-22.21	16042	663
3	PLCKECC	G000.50+00.00	42.1	0.5054	0.0	266.7046	-28.5045	680888	39860
4	PLCKECC	G000.65-00.01	235.8	0.6592	-0.0186	266.8135	-28.3826	2314982	141945
5	PLCKECC	G000.92-20.16	52.9	0.9229	-20.1651	288.5931	-36.871	14255	583
6	PLCKECC	G001.23+09.91	15.0	1.2305	9.9158	257.9445	-22.4685	7117	252
7	PLCKECC	G001.27+03.78	16.6	1.2744	3.7889	263.5491	-25.8457	13847	551
8	PLCKECC	G001.31-20.48	55.4	1.3184	-20.4833	289.1209	-36.626	11118	462
9	PLCKECC	G001.38+20.94	119.4	1.3843	20.942	248.6587	-15.778	41198	1644
10	PLCKECC	G001.64-00.07	36.4	1.6479	-0.0746	267.4453	-27.5645	223312	13209
11	PLCKECC	G001.84+16.58	119.1	1.8457	16.5877	252.554	-18.0729	31290	1233
12	PLCKECC	G001.95+09.78	46.1	1.9556	9.7833	258.5025	-21.9576	22952	876
13	PLCKECC	G003.07+09.95	68.0	3.0762	9.9537	259.0226	-20.9508	25645	985
14	PLCKECC	G003.14+08.19	18.5	3.1421	8.1971	260.6267	-21.8781	12710	522
15	PLCKECC	G003.27+10.42	43.6	3.2739	10.4274	258.7214	-20.523	7491	311
16	PLCKECC	G003.73+16.39	51.3	3.7354	16.3931	253.8409	-16.7265	8153	343
17	PLCKECC	G003.73+18.30	112.6	3.7354	18.3082	252.2281	-15.6006	30061	1076
18	PLCKECC	G004.02+16.64	33.0	4.021	16.646	253.7937	-16.3565	13979	477
19	PLCKECC	G004.15+35.77	175.8	4.1528	35.7772	238.3744	-4.6479	31490	1253

Figure 2.10. The table dataset resulting from running the example script.

2.6. Reading a comma-separated-value (CSV) file into a table dataset

This section explains how to read tabular data from a text file where data elements are separated by a **single comma**, like the following example:

```
First,Second,Third
Double,Integer,String
W,cm,
Power,Length,
# This line is a comment
1.2,3,One
4.3,7,Two
5.5,4,Three
```

Example 2.8. A standard comma-separated-value (CSV) file with a four-line header.

The above example shows an optional four-line *header* that gives the following additional information about the data:

- **First line.** Column names. Optional.
- **Second line.** Column data types. Mandatory.
- **Third line.** Column measurement units. Optional.
- **Fourth line.** Column description. Optional.

You must always include all the four header lines, but you can omit items on some lines (except the data types line) by writing just the data separator (a comma) without any data. In the previous example, only the first two values for measurement unit and description have been filled. You have to fill all values only for the second line, that of the data types. Note that, even if you choose not to fill any value of a header line, you still have to include that line. A header line with no filled values will be just a series of commas.

If you want to specify just the column names, you do not need to include a full header. You can list the column names on the first line. The line must begin with the # character, the same used for comments:

```
# First,Second,Third
# This line is a comment
1.2,3,One
4.3,7,Two
5.5,4,Three
```

Example 2.9. A standard comma-separated-value (CSV) file with only column titles specified.

In the graphical interface. Follow these steps:

1. Double click on the `asciiTableReader` task in the *Tasks* view. The task dialogue window opens in the *Editor* view.
2. Enter the path to the text file in the *file* field, or click the folder icon to the right of the field and navigate to the text file.
3. Select `CSV` from the *tableType* drop-down list.
4. Click *Accept*. The file is imported into a table dataset. For more information on table datasets, see the *Scripting Guide*: Section 2.4 in *Scripting Guide*.

Tip

If the file has extension `.csv`, you can import it by double clicking on it in the *Navigator* view of HIPE.

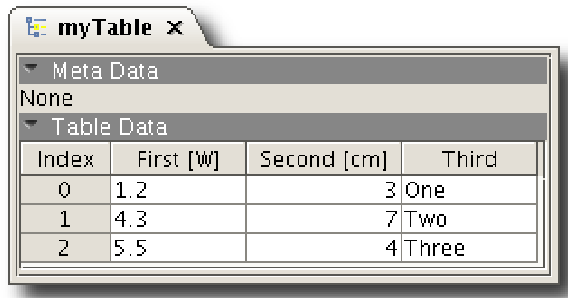
On the command line. Issue the following command in the *Console* view of HIPE, assuming that `myTable` is the name of your new table dataset and `/path/to/myTable.txt` is the path to the file you want to import:

```
myTable = asciiTableReader(file='/path/to/myTable.txt', tableType='CSV')
```

Example 2.10. Reading a table from a file, specifying its tabular format as CSV.

For a full list of task parameters of the `asciiTableReader` task, see the *User's Reference Manual*: Section 1.27 in *HCSS User's Reference Manual*.

Resulting table dataset. The file shown in Example 2.8 results in the following table dataset:



Index	First [W]	Second [cm]	Third
0	1.2	3	One
1	4.3	7	Two
2	5.5	4	Three

Figure 2.11. A table dataset imported from a CSV file.

- All lines beginning with # are ignored.
- Column names are as specified in the header. If one or more column names are missing in the header, they are replaced with `Column0`, `Column1`, `Column2` and so on.
- If a line of data has fewer elements than the others, missing data elements are represented by empty cells in the output table dataset.

2.7. Reading a space-separated file into a table dataset

This section explains how to read tabular data from a text file where data elements are separated by a **one or more spaces**, like the following example:

```
First Second   Third
Double Integer  String
W   cm []
Power Length []
# This is a comment
1.2  3 One
4.3 7 Two
5.5 4 Three
```

Example 2.11. A space-separated-value file of the type that can be imported into HIPE with default options.

The above example shows an optional four-line *header* that gives the following additional information about the data:

- **First line.** Column names. Optional.
- **Second line.** Column data types. Mandatory.
- **Third line.** Column measurement units. Optional.
- **Fourth line.** Column description. Optional.

You must always include all the four header lines, but you can omit items on some lines (except the data types line) by writing a set of empty square brackets `[]` instead of the item. In the previous example, only the first two values for measurement unit and description have been filled. You have to fill all values only for the second line, that of the data types. Note that, even if you choose not to fill any value of a header line, you still have to include that line. A header line with no filled values will be just a series of empty sets of square brackets.

If you want to specify just the column names, you do not need to include a full header. You can list the column names on the first line. The line must begin with the # character, the same used for comments:

```
# First Second Third
# This line is a comment
1.2 3 One
4.3 7 Two
5.5 4 Three
```

Example 2.12. A space-separated-value file with only column titles specified.

In the graphical interface. Follow these steps:

1. Double click the `asciiTableReader` task in the *Tasks* view. The task dialogue window opens in the *Editor* view.
2. Enter the path to the text file name in the *file* field, or click the folder icon to the right of the field and navigate to the text file.
3. Select `SPACES` from the *tableType* drop-down list.
4. Click *Accept*. The file is imported into a table dataset. For more information on table datasets, see the *Scripting Guide*: Section 2.4 in *Scripting Guide*.

Tip

If the file has extension `.tbl`, you can import it by double clicking on it in the *Navigator* view.

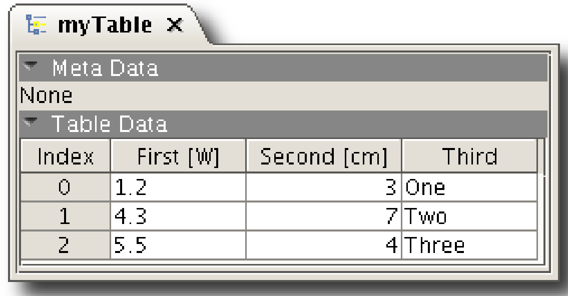
On the command line. Issue the following command in the *Console* view of HIPE, assuming that *`myTable`* is the name of your new table dataset and *`/path/to/myTable.txt`* is the path to the file you want to import:

```
myTable = asciiTableReader(file='/path/to/myTable.txt', tableType='SPACES')
```

Example 2.13. Read a table from an ASCII file, specifying that its values are space-separated.

For a full list of task parameters of the `asciiTableReader` task, see the *User's Reference Manual*: Section 1.27 in *HCSS User's Reference Manual*.

Resulting table dataset. The file shown in Example 2.11 results in the following table dataset:



myTable

Meta Data

None

Table Data

Index	First [W]	Second [cm]	Third
0	1.2	3	One
1	4.3	7	Two
2	5.5	4	Three

Figure 2.12. A table dataset imported from a space-separated-value file.

- All lines beginning with # are ignored.
- Column names are as specified in the header. If one or more column names are missing in the header, they are replaced with `c0`, `c1`, `c2` and so on.
- Enough columns are created to include all the values in the first line. If other lines have fewer elements, the resulting table dataset has empty cells.

2.8. Reading an IPAC, SExtractor or Topcat file into a table dataset

This section explains how to read text files obtained from one of the following sources:

- Catalogues from **IPAC/IRSA** (InfraRed Science Archive).
- Source lists from **SExtractor**.
- **Space-separated** tables.

In the graphical interface. Follow these steps:

1. Double click the `asciiTableReader` task in the *Tasks* view. The task dialogue window opens in the *Editor* view.
2. Enter the path to the text file name in the *file* field, or click the folder icon to the right of the field and navigate to the text file.
3. Select one of the following options from the *tableType* drop-down list:
   - `IPAC` for IPAC/IRSA files.
   - `SEXTRACTOR` for SExtractor files.
   - `SPACES` for space-separated files
4. Click *Accept*. The file is imported into a table dataset. For more information on table datasets, see the *Scripting Guide*: Section 2.4 in *Scripting Guide*.

On the command line. Issue the following command in the *Console* view of HIPE, assuming that *`myTable`* is the name of your new table dataset and *`/path/to/myTable.txt`* is the path to the file you want to import:

```
myTable = asciiTableReader(file='/path/to/myTable.txt', tableType='IPAC')
```

Example 2.14. Read a table from an ASCII file, specifying the input format as IPAC.

The previous command works for an IPAC/IRSA file. For a SExtractor or space-separated file, replace `IPAC` with `SEXTRACTOR` or `SPACES`, respectively.

For a full list of task parameters of the `asciiTableReader` task, see the *User's Reference Manual*: Section 1.27 in *HCSS User's Reference Manual*.

Known issues. While reading space-separated or SExtractor files you may encounter the following issues:

- Some SExtractor headers are not read successfully, due to an issue with the column position of units in the file header. Workaround: read the file with the *`tableType`* parameter set to *SPACES*. You can then fill any missing information by hand.

- Some space-separated files are not read successfully, due to an issue with quoted cells with spaces. Workaround when using Topcat: export the data from Topcat in FITS format, then read the FITS file back into HIPE. For information about reading FITS files into HIPE, see Section 1.16.6.

2.9. Reading a generic ASCII table file into a table dataset

Read this section if the type of file you want to read does not correspond to those described in Section 2.6, Section 2.7, or in the worked examples at the beginning of this chapter.

In the graphical interface. Follow these steps:

1. Double click on the `asciiTableReader` task in the *Tasks* view. The task dialogue window opens in the *Editor* view.
2. Enter the text file name in the *file* field, or click the folder icon to the right of the field and navigate to the text file.
3. Change the *tableType* value to `ADVANCED`.
4. Click on the *Advanced* tab.
5. Change the options in the *Advanced* tab according to your needs (see *Customisation options* in this section). In particular, you *must* change the value of the *columnType* parameter to something other than *GUESS_NONE*.
6. Click *Accept*. HIPE imports the file contents into a table dataset.

On the command line. Assuming that `/path/to/myFile.txt` is the path to the file you want to import, and `myTable` the name of the table dataset you want to create, start with this basic command:

```
myTable = asciiTableReader(file='/path/to/myFile.txt')
```

Example 2.15. Reading a table from an ASCII file, without specifying any input format.

Add options to the command according to your needs (see *Customisation options* in this section).

Customisation options

- **Column names are provided with the file.** See Section 2.14 for how to import them.
- **You want HIPE to skip a number of lines at the top of the file.** See Section 2.15.
- **You want HIPE to ignore lines starting with certain characters.** See Section 2.15.
- **You want HIPE to delete all white space at the beginning and end of each line.** See Section 2.15.
- **You want to specify the data types of the columns in the files.** See Section 2.16.
- **You want to specify how data values are separated in your file.** See Section 2.17.
- **You want to save the configuration of the task to use it again, or load a previously saved configuration.** See Section 2.18.

2.10. Writing a table dataset to a comma-separated-values (CSV) file

This section shows how to save a table dataset like the one shown in Figure 2.13 to a comma-separated-value text file.

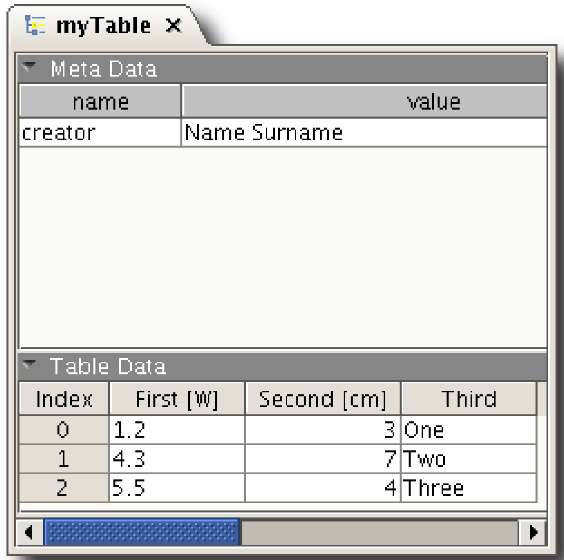


Figure 2.13. A simple table dataset.

In the graphical interface. Follow these steps:

1. In the *Variables* view, right click on the variable name corresponding to the table dataset you want to save, and choose *Send to* → *Text file*.

   The `asciiTableWriter` task dialogue window opens in the *Console* view.

2. In the *file* text field, write the name of the text file you want to save the table dataset to.

3. Click *Accept*. The table dataset is saved to file. Moreover, HIPE writes the corresponding command to the *Console* view.

On the command line. Issue the following command in the *Console* view of HIPE, assuming that *`myTable`* is your table dataset variable and *`/path/to/myTable.txt`* is the full path to the file you want to export the dataset to:

```
asciiTableWriter(table=myTable, file='/path/to/myTable.txt')
```

Example 2.16. Write a table to an ASCII file.

For a full list of parameters of the `asciiTableWriter` task, see the *User's Reference Manual*: Section 1.28 in *HCSS User's Reference Manual*.

Resulting file. The table dataset shown in Figure 2.13 is saved in the following file:

```
First,Second,Third
Double,Integer,String
W,cm,
Power,Length,
1.2,3,One
4.3,7,Two
5.5,4,Three
```

- Data values are separated by commas.

- A four-line header is added to the top of the text file. The four lines show the names, data types, units of measurements and descriptions of each column. In case the unit of measurement or the description of a column is not set, the corresponding space is left blank, as for the third column in the example.
- No metadata is written to the file. The `creator` metadata parameter in Figure 2.13 does not appear in the output file.
- HIPE gives you a warning if you are about to overwrite an existing file.

2.11. Writing a table dataset into a space-separated-value file

This section shows how to save a table dataset like the one shown in Figure 2.14 to a space-separated-value text file.

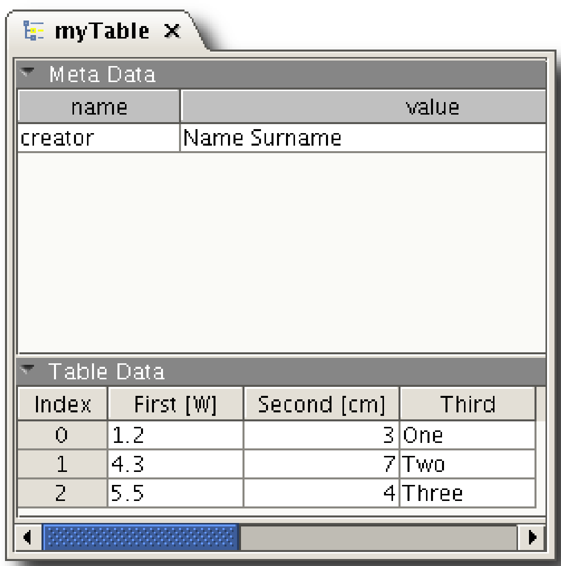


Figure 2.14. A simple table dataset.

In the graphical interface. Follow these steps:

1. In the *Variables* view, right click on the variable name corresponding to the table dataset you want to save, and choose *Send to* → *Text file*.

   The `asciiTableWriter` task dialogue window opens in the *Console* view.

2. In the *file* text field, write the name of the text file you want to save the table dataset to.
3. In the *Console* view, create a formatter with the following command:

```
formatter = CsvFormatter(delimiter=' ')
```

   Example 2.17. Creating a formatter that separates values using a single space character.

4. Drag the `formatter` variable from the *Variables* view to the grey circle next to the *formatter* label.
5. Click *Accept*. The table dataset is saved to file. Moreover, HIPE writes the corresponding command to the *Console* view.

On the command line. Issue the following commands in the *Console* view of HIPE, assuming that `myTable` is your table dataset variable and `/path/to/myTable.txt` is the full path to the file you want to export the dataset to:

```
formatter = CsvFormatter(delimiter=' ')
asciiTableWriter(table=myTable, file='/path/to/myTable.txt', formatter=formatter)
```

Example 2.18. Creating a space-separated formatter and using it to write a table as an ASCII file.

For a full list of parameters of the `asciiTableWriter` task, see the *User's Reference Manual*: Section 1.28 in *HCSS User's Reference Manual*.

Resulting file. The table dataset shown in Figure 2.13 is saved in the following file:

```
First Second Third
Double Integer String
W cm
Power Length
1.2 3 One
4.3 7 Two
5.5 4 Three
```

- Data values are separated by spaces.
- A four-line header is added to the top of the text file. The four lines show the names, data types, units of measurements and descriptions of each column. In case the unit of measurement or the description of a column is not set, the corresponding space is left blank, as for the third column in the example.
- No metadata is written to the file. The `creator` metadata parameter in Figure 2.13 does not appear in the output file.
- HIPE gives you a warning if you are about to overwrite an existing file.

2.12. Writing a spectrum to an ASCII table file

Use the `exportSpectrumToAscii` in *HCSS User's Reference Manual* task to export a spectrum from HIPE to a table in a text file. You can use this task on any spectrum implementing the `SpectrumContainer` interface, which in practice means all the main types of spectra used in HIPE. You can also use this task on spectral cubes, to export one or more spectra extracted from the cube. See Section 5.2 for more information on spectra in HIPE. See Section 6.2 for more information on spectral cubes in HIPE.

Warning

See the warning in Section 2.2 for a known limitation regarding the `error` column when using `exportSpectrumToAscii`. This is important for SPIRE and PACS data.

This section first describes how to use the task from the graphical interface and from the command line. Then it lists the available customisations.

The task is available from the Spectrum Toolbox accessed via the Spectrum Explorer. This is probably the quickest way to access it while you work on your spectra or cubes. You can also open the task from the *Tasks* view; the dialogue window is the same, but there are some differences in behaviour when selecting spectra or spectral segments.

For more information on the Spectrum Explorer, see Section 5.3 for spectra and Section 6.6 for spectral cubes.

Input spectrum. The following HIFI spectrum (class `WbsSpectrumDataset`), made of four segments, is used as an example of dataset you can export to text file with this task.

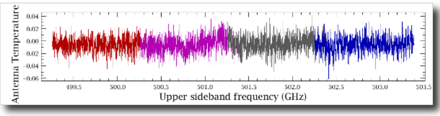


Figure 2.15. Input spectrum for the exportSpectrumToAscii task.

Running from the spectrum/cube toolbox. Follow these steps:

1. Right click on the spectrum or cube in the *Variables* view and choose *Open with → Spectrum Explorer*.

   The spectrum or cube opens in the Spectrum Explorer, in a new tab within the *Editor* view.

2. In the Spectrum Explorer toolbar, click the toolbox icon .

   The toolbox tab appears in the upper right area.

3. From the drop-down list at the top of the toolbox tab, select *ExportSpectrumToAscii*.

   The task dialogue window appears inside the toolbox tab.

4. To save a whole spectrum or cube, drag the corresponding variable from the *Variables* view to the grey circle next to the *ds* parameter. The *ds* parameter is the first in the list.

   The circle becomes green. If the circle becomes red, the variable is not of the correct type.

   - To save just a few segments from a spectrum, see *Exporting a selection of segments from the input spectrum* under *Customising task output* later in this section.
   - To save just a few spectra from a cube, see *Exporting a selection of spectra from the input dataset* under *Customising task output* later in this section.

5. In the *file* text field, write the name of the text file you want to save the selected spectra to.

6. Click *Accept*.

   HIPE saves the selected spectra to the specified text file. If you did not specify a full path, HIPE saves the file in the directory it was started from.

Running from the *Tasks* view. Follow these steps:

1. In the *Tasks* view, open the *All* folder by double clicking on it. Type the first letters of the `exportSpectrumToAscii` task name to jump close to its position in the list. Double click on the task name. The task will also appear under *Applicable* if you have first highlighted a spectrum dataset in the *Variables* view.

   The task dialogue window opens in the *Editor* view.

2. From the *Variables* view, drag the variable corresponding to your spectrum or spectral cube to the gray circle next to the *ds* input parameter in the task dialogue window. The *ds* parameter is the first in the list.

   The circle becomes green. If the circle becomes red, the variable is not of the correct type.

   - To save just a few segments from a spectrum, see *Exporting a selection of segments from the input spectrum* under *Customising task output* later in this section (grep for "Customising").
   - To save just a few spectra from a cube, see *Exporting a selection of spectra from the input dataset* under *Customising task output* later in this section.

3. In the *file* text field, write the name of the text file you want to save the spectrum or cube to.

4. Click *Accept*.

   HIPE saves the spectrum or cube to the specified text file. If you did not specify a full path, HIPE saves the file in the directory it was started from.

Running on the command line. Issue the following commands in the *Console* view of HIPE, assuming that `mySpectrum` is your spectrum variable and `/path/to/mySpectrum.txt` is the full path to the file you want to export the spectrum to:

```
exportSpectrumToAscii(ds=mySpectrum, file='/path/to/mySpectrum.txt')
```

Example 2.19. Writing spectrum data to an ASCII file.

Resulting file. A portion of the resulting file is shown below:

```
# Meta data {
#  type={description="Product Type Identification",
 string="herschel.ia.dataset.Product( ...
#  creator={description="Generator of this product", string="SPG v8.2.1"}
...
usbfrequency_segm1_0,usbfrequency_segm2_0,usbfrequency_segm3_0,
  usbfrequency_segm4_0,flux_segm1_0,flux_segm2_0,flux_segm3_0,flux_segm4_0
Double,Double,Double,Double,Double,Double,Double,Double
GHz,GHz,GHz,GHz,K,K,K,K
Upper sideband frequency,Upper sideband frequency,Upper sideband frequency,
   Upper sideband frequency,Antenna Temperature,Antenna Temperature,
   Antenna Temperature,Antenna Temperature
499.256,500.2565,501.254,502.2535,1.0687267838317963E-4,0.003435710413794657,NaN,
   -0.03445117278045858
...
```

- The metadata associated with the spectrum are listed at the top of the file. Each metadata element is on a different line, preceded by a # character.
- Following the metadata list are four lines of header with the following information about the data columns:
  - Names
  - Data types (all `Double` in the previous example)
  - Units (GHz and K in the previous example)
  - Descriptions (*Upper sideband frequency* and *Antenna Temperature* in the previous example)
- Data values and items in the header are separated by commas.

Customising task output

Omitting metadata from the output file.

- **In the graphical interface.** Untick the *meta* checkbox.
- **On the command line.** Add the *`meta=False`* parameter to the command:

  ```
  exportSpectrumToAscii(ds=mySpectrum, file='/path/to/mySpectrum.txt',\
  meta=False)
  ```

  Example 2.20. Writing spectrum data to an ASCII file without including metadata.

Including flags in the output file.

- **In the graphical interface.** Tick the *flags* checkbox.

- **On the command line.** Add the `flags=True` parameter to the command:

```
exportSpectrumToAscii(ds=mySpectrum, file='/path/to/mySpectrum.txt',\
flags=True)
```

Example 2.21. Writing spectrum data to an ASCII file, including flags.

Including weights in the output file.

- **In the graphical interface.** Tick the *weights* checkbox.
- **On the command line.** Add the `weights=True` parameter to the command:

```
exportSpectrumToAscii(ds=mySpectrum, file='/path/to/mySpectrum.txt',\
weights=True)
```

Example 2.22. Writing spectrum data to an ASCII file, including weights.

Writing data of different segments in the same column. This option is useful only for HIFI data. By default, data from different spectral segments are written to separate columns. For instance, the example spectrum used earlier in this section has four segments, and the resulting file has eight columns, corresponding to frequency and antenna temperature for each segment.

You can choose to merge the same type of data from each segment into a single column.

- **In the graphical interface.** Tick the *concat* checkbox.
- **On the command line.** Add the `concat=True` parameter to the command:

```
exportSpectrumToAscii(ds=mySpectrum, file='/path/to/mySpectrum.txt',\
concat=True)
```

Example 2.23. Writing spectrum data to an ASCII file, concatenating the spectral segments.

Exporting a selection of spectra from the input dataset. You can choose to save to file only a few spectra from your input dataset, for instance just a few spaxels from a spectral cube.

Whether you run the task from the command line or have opened the dialogue window from the *Tasks* view, you must create a variable representing the spectra you want to include, which you will then input into the task.

- If you opened the task from the Spectrum Explorer (specifically the Spectrum Toolbox) you can chose to identify the spectra to export using the Spectrum Explorer's standard spectrum selection method, that is, clicking on the spectrum row or spaxel in the Data Selection panel of the Spectrum Explorer. See Section 6.6.2 if working with a cube, and Section 5.3.1 otherwise.
- If you opened the task from the *Tasks* view, or are running from the command line, you need to create a selection array to identify the indices of the spectra you wish to export. For anything but cubes this is simply a matter of typing a list of the spectrum order (0,1,3,5,...) you want:

```
mySelection = [5, 12, 24]
```

For a cube, where spaxel coordinates are more natural (that is, 0,0 rather than 0), you will still need to identify the spectrum indices that correspond to the spaxels you want, since the task does not accept spaxel coordinates. You can do this with the Spectrum Explorer:

1. In the *Data Selection Panel*, click on the spaxels you want to select, if looking at a cube, or the row, if looking at a non-cube multi-spectrum dataset.

   HIPE plots the corresponding spectra in the *Spectrum Panel*. Make sure only the spectra you want to select are displayed.

2. Click the icon in the toolbar.

   The *DataTree* tab opens in the *Data Selection Panel*. Use this to identify the indices of the spectra you have selected; particularly useful if you are working on a cube, since the task does not accept spaxel coordinates.

3. Go to the *DataTree* tab.The spaxels you have selected are marked by coloured squares. The numeric index of each spectrum is shown in the *variable* column.

4. You can now create the selection array, using the following command in the *Console* view:

```
mySelection = [5, 12, 24]
```

After creating a selection, follow these steps:

- **In the graphical interface.** Drag the selection array from the *Variables* view to the grey circle next to the *selection* parameter.

- **On the command line.** Add the `selection` parameter to the command, giving your selection array variable as value:

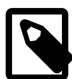
```
exportSpectrumToAscii(ds=myCube, file='/path/to/myCube.txt',\
selection=mySelection)
```

Example 2.24. Writing spectrum data to an ASCII file, specifying a selection of spectral indices.

You can also create the selection array directly, without creating a variable first:

```
exportSpectrumToAscii(ds=myCube, file='/path/to/myCube.txt',\
selection=[5, 12, 14])
```

Example 2.25. Writing spectrum data to an ASCII file, with a literal array of spectral indices.



And remember that running the task from the spectrum/cube toolbox,

you only need to select spectra in the *Data Selection Panel*: see links given above.

Exporting a selection of segments from the input spectrum. This option is useful only for HIFI data. By default HIPE exports all the segments in a spectrum to the text file. You can choose which segments to export by passing an array with their indices to the task. Note that cubes, including HIFI cubes, contain only one segment per spaxel.

If you run the task from the command line or have opened the dialogue window from the *Tasks* view, you must create a variable representing the segments you want to include. If you have opened the task from the Spectrum Explorer you can chose to identify the spectra to export using the spectrum selection method of the Spectrum Explorer (i.e. click to display): see Section 5.3.1. You can also use the Spectrum Explorer to determine the indices corresponding to the segments you want to save: in the *Data Selection Panel*, the column corresponding to each segment shows the index number. You can then create the segments array with the following command:

```
mySegments = [1, 3]
```

Then,

- **In the graphical interface.** Drag the segments array from the *Variables* view to the grey circle next to the *segments* parameter.

- **On the command line.** Add the `segments` parameter to the command, giving your segments array variable as value:

```
exportSpectrumToAscii(ds=mySpectrum,
file='/path/to/mySpectrum.txt',\
segments=mySegments)
```

Example 2.26. Writing spectrum data to an ASCII file, specifying a selection of spectral segments.

You can also create the segments array directly, without creating a variable first:

```
exportSpectrumToAscii(ds=mySpectrum,
file='/path/to/mySpectrum.txt',\
segments=[1, 3])
```

Example 2.27. Writing spectrum data to an ASCII file, specifying a literal array of spectral segments.

- **And if running the task from the Spectrum Explorer,.** then your selection is done using the methods described in the link given above i.e. simply click to display the spectra you want to include.

Changing the data delimiter. By default, data values in the output file are separated by a comma. To change the data delimiter, first you have to create a *formatter*, and that is then input into the task. For example, the following is a formatter that changes the data delimiter to a space:

```
myFormatter = CsvFormatter(delimiter=' ')
```

Example 2.28. Creating a space-delimited formatter.

To use another character as delimiter, give it as value of the `delimiter` property, surrounded by single or double quotes.

- **In the graphical interface.** Drag the formatter variable from the *Variables* view to the grey circle next to the *formatter* parameter. This will work for the graphical interface accessed via *Tasks* or via the Spectrum Explorer.
- **On the command line.** Add the `formatter` parameter to the command:

```
exportSpectrumToAscii(ds=mySpectrum, file='/path/to/mySpectrum.txt',\
formatter=myFormatter)
```

Example 2.29. Writing spectrum data to an ASCII file, specifying a space-delimited formatter.

2.13. Writing a table dataset to a generic ASCII table file

Read this section if the type of file you want to write does not correspond to those described in Section 2.10, Section 2.11 or Section 2.12.

In the graphical interface. Follow these steps:

1. In the *Variables* view, right click on the variable name corresponding to the table dataset you want to save, and choose *Send to* → *Text file*.

   The `asciiTableWriter` task dialogue window opens in the *Console* view.

2. In the *file* text field, write the name of the text file you want to save the table dataset to.
3. Change the other options according to your needs (see *Customisation options* in this section).
4. Click *Accept*. The table dataset is saved to file. Moreover, HIPE writes the corresponding command to the *Console* view.

On the command line. Assuming that *myTable* is your table dataset variable and */path/to/myTable.txt* is the full path to the file you want to export the dataset to, start with this basic command in the *Console* view of HIPE:

```
asciiTableWriter(table=myTable, file='/path/to/myTable.txt')
```

Example 2.30. Writing a table to disk.

Add options to the command according to your needs (see *Customisation options* in this section).

Customisation options

- **You want to add a header to the file.** See Section 2.19.
- **You want to add the table dataset metadata to the file.** See Section 2.20.
- **You want to define a custom prefix to denote commented lines.** See Section 2.21.
- **You want to specify how to separate data values.** See Section 2.22.
- **You want to save the configuration of the task to use it again, or to load a previously saved configuration.** See Section 2.23.

2.14. Reading column names from a file

Column names in a file must be on the first line and separated by the same character or set of characters separating data values. For example, if data values are separated by commas, column names must be separated by commas as well.

If you are reading a file with a four-line header, as explained in Section 2.6 or Section 2.7, column names are read automatically. Column names are read also if listed on the first line of the file and preceded by the # character, as explained in the same sections. In other cases, follow these steps.

In the graphical interface. In the *Advanced* tab of the `asciiTableReader` task dialogue window, tick the *parseNames* checkbox to read the column names.

On the command line. Add the `parseNames=True` parameter to the `asciiTableReader` task command in order to read the column names, as in the following example:

```
myTable = asciiTableReader(file='myFile.txt', tableType='ADVANCED', columnType=18,
 parseNames=True)
```

Example 2.31. Reading a table from a file, specifying the ADVANCED type of table for parsing.

Warning

This option is ignored if the *tableType* parameter is set to `CSV` or `SPACES`:

In the graphical interface: In the `asciiTableReader` task dialogue window, set the *tableType* drop-down list to *ADVANCED*. In the *Advanced* tab, set the *columnType* parameter to something other than *GUESS_NONE*. See Section 2.16 for more details.

On the command line: Add the `tableType='ADVANCED'` parameter to the `asciiTableReader` task command, and the `columnType` parameter with an appropriate value. See Section 2.16 for more details.

2.15. Defining which lines to ignore when reading a file

When reading data from a text file you can ignore lines, or portion of lines, in three ways:

- Ignoring a certain number of lines at the beginning of the file.
- Ignoring lines beginning with a certain character, or set of characters.
- Ignoring white space at the beginning and end of each line.

Ignoring lines at the beginning of the file

In the graphical interface. In the *Advanced* tab of the `asciiTableReader` task dialogue window, enter the number of lines to be ignored in the *`parserSkip`* text field.

On the command line. Add the `parserSkip` parameter to the `asciiTableReader` task command, as in the following example:

```
myTable = asciiTableReader(file='myFile.txt', tableType='ADVANCED', columnType=18,
 parserSkip=3)
```

Example 2.32. Reading a table from a file, specifying the ADVANCED table type for parsing and skipping header lines.

Warning

This option is ignored if the *`tableType`* parameter is set to `CSV` or `SPACES`:

In the graphical interface: In the `asciiTableReader` task dialogue window, set the *tableType* drop-down list to *ADVANCED*. In the *Advanced* tab, set the *columnType* parameter to something other than *GUESS_NONE*. See Section 2.16 for more details.

On the command line: Add the `tableType='ADVANCED'` parameter to the `asciiTableReader` task command, and the `columnType` parameter with an appropriate value. See Section 2.16 for more details.

Ignoring lines beginning with a certain character, or set of characters.

In the graphical interface. In the *Advanced* tab of the `asciiTableReader` task dialogue window, choose one option from the *ignorePattern* drop-down list, or enter your own characters in the text field.

Hover your mouse pointer on each of the options in the command line to see a tooltip explaining what it does. The three available options ignore lines beginning with `#`, empty lines or both.

If you want to ignore lines beginning by `//`, for example, enter `//` in the *ignorePattern* text field.

If you want HIPE to write a warning each time it ignores a line, set the *ignoreWarn* parameter to *True*.

On the command line. Add the `ignorePattern` parameter to the `asciiTableReader` task command, as in the following example:

```
myTable = asciiTableReader(file='myFile.txt', tableType='ADVANCED', columnType=18,
 ignorePattern='//')
```

Example 2.33. Reading a table from an ASCII file, with ADVANCED type and character ignore options for the parser.

The previous command causes HIPE to ignore all lines beginning with `//` when reading the file.

If you want HIPE to write a warning each time it ignores a line, add the *`ignoreWarn=True`* to the `asciiTableReader` task command.

Warning

This option is ignored if the *`tableType`* parameter is set to `CSV` or `SPACES`:

In the graphical interface: In the `asciiTableReader` task dialogue window, set the *tableType* drop-down list to *ADVANCED*. In the *Advanced* tab, set the *columnType* parameter to something other than *GUESS_NONE*. See Section 2.16 for more details.

On the command line: Add the `tableType='ADVANCED'` parameter to the `asciiTableReader` task command, and the `columnType` parameter with an appropriate value. See Section 2.16 for more details.

Ignoring white space at the beginning and end of each line.

This option is useful when the data values are separated by spaces, and additional spaces at the beginning and end of lines could confuse HIPE. To do this, you should specify an `ignorePattern` (see above) that removes or ignores whitespaces.

```
myTable = asciiTableReader(file='myFile.txt', tableType='ADVANCED', columnType=18,
 ignorePattern='^\s*')
```

Example 2.34. Reading a table from an ASCII file, with ADVANCED type and an ignore pattern that trims spaces at the beginning.

Warning

This option is ignored if the *`tableType`* parameter is set to `CSV` or `SPACES`:

In the graphical interface: In the `asciiTableReader` task dialogue window, set the *tableType* drop-down list to *ADVANCED*. In the *Advanced* tab, set the *columnType* parameter to something other than *GUESS_NONE*. See Section 2.16 for more details.

On the command line: Add the `tableType='ADVANCED'` parameter to the `asciiTableReader` task command, and the `columnType` parameter with an appropriate value. See Section 2.16 for more details.

2.16. Specifying the data types when reading a file

You can let HIPE guess the types of data values in a file, you can specify a single data type or you can specify multiple data types via a *table template*.

In the graphical interface. Make the following changes in the *Advanced* tab of the `asciiTableReader` task dialogue window.

- **To let HIPE guess the data types:** From the *columnType* drop-down list choose *GUESS_TRY* (HIPE guess based on the first 100 lines of the file) or *GUESS_ALL* (HIPE guess based on the whole file).
- **To select a single type for all data values:** From the *columnType* drop-down list choose *ALL_STRING*, *ALL_BOOLEAN* or one of the other similar options.
- **To define a different data value for each column of the file:** From the *columnType* drop-down list choose *GUESS_NONE*. Create a table template as described in Section 2.25. Drag the variable representing the template from the *Variables* view to the grey circle next to the *template* label.

On the command line. Make the following changes to the command calling the `asciiTableReader` task. Note that you have to issue the command `from herschel.ia.io.ascii`

`import AsciiParser` (once for each HIPE session is enough) for the `asciiTableReader` call to work.

- **To let HIPE guess the data types:** Set the *columnType* parameter to *AsciiParser.GUESS_TRY* (HIPE guess based on the first 100 lines of the file) or *AsciiParser.GUESS_ALL* (HIPE guess based on the whole file), as in the following example:

```
from herschel.ia.io.ascii import AsciiParser
myTable = asciiTableReader(file='myFile.txt', tableType='ADVANCED',
 columnType=AsciiParser.GUESS_ALL)
```

Example 2.35. Reading a table from an ASCII file, with ADVANCED type and guessing all value types.

- **To select a single type for all data values:** Set the *columnType* parameter to *AsciiParser.ALL_BOOLEAN*, *AsciiParser.ALL_BYTE* or one of the other similar options, as in the following example:

```
from herschel.ia.io.ascii import AsciiParser
myTable = asciiTableReader(file='myFile.txt', tableType='ADVANCED',
 columnType=AsciiParser.ALL_DOUBLE)
```

Example 2.36. Reading a table from an ASCII file, with ADVANCED type and parsing all values as doubles.

- **To define a different data value for each column of the file:** Create a table template as described in Section 2.25. Assuming that *myTableTemplate* is the variable representing your template, add the *template=myTableTemplate* parameter to the `asciiTableReader` task command, as in the following example:

```
myTable = asciiTableReader(file='myFile.txt', tableType='ADVANCED',
 template=myTableTemplate)
```

Example 2.37. Reading a table from an ASCII file, with ADVANCED type and a custom table template.

Warning

This option is ignored if the *tableType* parameter is set to `CSV` or `SPACES`:

In the graphical interface: In the `asciiTableReader` task dialogue window, set the *tableType* drop-down list to *ADVANCED*.

On the command line: Add the `tableType='ADVANCED'` parameter to the `asciiTableReader` task command.

2.17. Specifying how data values are separated when reading a file

Data values in the ASCII table file you are trying to read could be separated by commas, spaces or other characters. Columns could have fixed or variable width. You can tell HIPE in detail how your data values are separated.

In the graphical interface. In the *Advanced* tab of the `asciiTableReader` task dialogue window, set the *delimiter* parameter to the characters separating data values in your file. You can write them directly in the text field or choose one of the available options:

- **Comma.** Data values are separated by a single comma.
- **\s+.** Data values are separated by one or more spaces.
- **\t+.** Data values are separated by one or more tab characters.

If your data values are organised in fixed-width columns, or are separated in more complicated ways you cannot express with the *delimiter* parameter, you can create a *parser* as described in Section 2.26. Then drag the variable representing your parser from the *Variables* view to the grey circle next to the *parser* parameter in the *Advanced* tab of the `asciiTableReader` task dialogue window.

On the command line. Add the `delimiter` parameter to the `asciiTableReader` task command, as shown in the following example:

```
myTable = asciiTableReader(file='/path/to/myFile.txt', tableType='ADVANCED',
 columnType=18, delimiter=' ')
```

Example 2.38. Reading a table from an ASCII file, with ADVANCED type and assuming 18 character-wide columns and space separators.

If your data values are organised in fixed-width columns, or are separated in more complicated ways you cannot express with the `delimiter` parameter, you can create a *parser* as described in Section 2.26. Assuming that `myParser` is the name of the parser you have created, add it to the `asciiTableReader` task command, as shown in the following example:

```
myTable = asciiTableReader(file='/path/to/myFile.txt', tableType='ADVANCED',
 columnType=18, parser=myParser)
```

Example 2.39. Reading a table from an ASCII file, with ADVANCED type and providing a fully customised parser.

Warning

This option is ignored if the `tableType` parameter is set to `CSV` or `SPACES`:

In the graphical interface: In the `asciiTableReader` task dialogue window, set the *tableType* drop-down list to *ADVANCED*. In the *Advanced* tab, set the *columnType* parameter to something other than *GUESS_NONE*. See Section 2.16 for more details.

On the command line: Add the `tableType='ADVANCED'` parameter to the `asciiTableReader` task command, and the `columnType` parameter with an appropriate value. See Section 2.16 for more details.

2.18. Saving and loading a configuration for reading from file

You can save all the configuration options you set when reading a file, so that you can load them in one step when loading other files with the same formatting.

Saving a configuration file

Follow these instructions to save a configuration file when you read an ASCII table file.

In the graphical interface. In the *Advanced* tab of the `asciiTableReader` task dialogue window, write a file name in the *writeConfigFile* text field, optionally with a full path. Alternatively, click the folder icon to the right of the text field to navigate to the directory where you want to save the file.

On the command line. Assuming that `myConfig.conf` is the name you want to give to your configuration file, add the `writeConfigFile=myConfig.conf` parameter to the `asciiTableReader` task command, as in the following example:

```
myTable = asciiTableReader(file='myFile.txt', tableType='ADVANCED',
 writeConfigFile='myConfig.conf')
```

Example 2.40. Reading a table from an ASCII file while writing the parsing configuration to a file for reuse.

You can specify a full path instead of just the file name.

The configuration file is in binary format and cannot be modified with a text editor. You can leave the file name without extension or give it any extension you like, such as `.conf`.

If you specify the file name without a path, the file is saved in the directory from where HIPE was started. For ease of retrieval you should always specify a full path.

Warning

This option is ignored if the *`tableType`* parameter is set to `CSV` or `SPACES`:

In the graphical interface: In the `asciiTableReader` task dialogue window, set the *tableType* drop-down list to *ADVANCED*. In the *Advanced* tab, set the *columnType* parameter to something other than *GUESS_NONE*. See Section 2.16 for more details.

On the command line: Add the `tableType='ADVANCED'` parameter to the `asciiTableReader` task command, and the `columnType` parameter with an appropriate value. See Section 2.16 for more details.

Loading a configuration file

Follow these instructions to use the options in a configuration file when reading an ASCII table file.

In the graphical interface. In the *Advanced* tab of the `asciiTableReader` task dialogue window, write the configuration file path and name in the *readConfigFile* text field. Alternatively, click the folder icon to the right of the text field to navigate to the file.

On the command line. Assuming that *`/path/to/myConfig.conf`* is the full path and name of your configuration file, add the *`writeConfigFile=/path/to/myConfig.conf`* parameter to the `asciiTableReader` task command, as in the following example:

```
myTable = asciiTableReader(file='myFile.txt', tableType='ADVANCED',
 readConfigFile='/path/to/myConfig.conf')
```

Example 2.41. Reading a table from an ASCII file, specifying a parsing configuration file.

2.19. Adding a header to an ASCII table file

In the graphical interface. In the `asciiTableWriter` task dialogue window, set the *writeHeader* parameter to *True*.

On the command line. Adding a header is the default, so you do not need to add anything to the `asciiTableWriter` task command. To add no header, add the *`writeHeader=False`* parameter to the `asciiTableWriter` task command, as in the following example:

```
asciiTableWriter(table=myTable, file='/path/to/myFile.txt', writeHeader=False)
```

Example 2.42. Writing a table to an ASCII file without a header.

2.20. Adding table dataset metadata to an ASCII table file

In the graphical interface. In the `asciiTableWriter` task dialogue window, set the *writeMetadata* parameter to *True*.

On the command line. Add the *`writeMetadata=True`* parameter to the `asciiTableWriter` task command, as in the following example:

```
asciiTableWriter(table=myTable, file='/path/to/myFile.txt', writeMetadata=True)
```

Example 2.43. Writing a table to an ASCII file including metadata.

Metadata are written to the ASCII table file as commented lines. Commented lines begin by default with the # character. To define a custom character, or series of characters, see Section 2.21.

2.21. Defining a custom prefix for commented lines

Commented lines contain information that can be read by humans but is ignored by HIPE or other software when reading back the ASCII table file. For example, if you write to file the metadata of a table dataset, as described in Section 2.20, these are written as comments.

The default character defining commented lines is `#`. You may want to define a different one if you want to read your ASCII table file with software other than HIPE, following different conventions.

In the graphical interface. In the `asciiTableWriter` task dialogue window, write the new character, or series of characters, in the *metadataPrefix* text field.

On the command line. Add the *`metadataPrefix`* parameter to the `asciiTableWriter` task command, with the prefix character of series of characters. In the following example the prefix for commented lines is redefined as `//`:

```
asciiTableWriter(table=myTable, file='/path/to/myFile.txt', \
writeMetadata=True, metadataPrefix='//')
```

Example 2.44. Writing a table to an ASCII file including metadata with a custom prefix.

2.22. Choosing how to separate data values

By default, data values in ASCII table files written with the `asciiTableWriter` are separated by commas. You can define a different character to separate data values, or you can define a fixed width for the data columns in the file. In case of fixed-width columns, data values are always separated by spaces.

In the graphical interface. Create a *formatter* as described in Section 2.27. Drag the variables representing the formatter from the *Variables* view to the grey circle next to the *formatter* label in the `asciiTableWriter` task graphical interface.

On the command line. Create a *formatter* as described in Section 2.27. Add the *`formatter`* parameter to the `asciiTableWriter` task command. In the following example, *`myFormatter`* is the variable corresponding to the formatter:

```
asciiTableWriter(table=myTable, file='/path/to/myFile.txt', formatter=myFormatter)
```

Example 2.45. Writing a table to an ASCII file with a custom formatter.

Warning

If you set any of the parameters affecting the *formatter*, such as *writeHeader*, these will override whatever is specified in the formatter.

2.23. Saving and loading options for writing to file

You can save all the configuration options you set when writing a file, so that you can load them in one step when writing other files with the same formatting.

Saving a configuration file

Follow these instructions to save a configuration file when you write an ASCII table file.

In the graphical interface. In the `asciiTableWriter` task dialogue window, write a file name in the *writeConfigFile* text field, optionally with a full path. Alternatively, click the folder icon to the right of the text field to navigate to the directory where you want to save the file.

On the command line. Assuming that *`myConfig.conf`* is the name you want to give to your configuration file, add the *`writeConfigFile='myConfig.conf'`* parameter to the `asciiTableWriter` task command, as in the following example:

```
asciiTableWriter(table=myTable, file='myFile.txt', writeConfigFile='myConfig.conf')
```

Example 2.46. Writing a table to an ASCII file saving the writing configuration to another file.

You can specify a full path instead of just the file name.

The configuration file is in binary format and cannot be modified with a text editor. You can leave the file name without extension or give it any extension you like, such as `.conf`.

If you specify the file name without a path, the file is saved in the directory from where HIPE was started. For ease of retrieval you should always specify a full path.

Loading a configuration file

Follow these instructions to use the options in a configuration file when reading an ASCII table file.

In the graphical interface. In the `asciiTableWriter` task dialogue window, write the configuration file path and name in the *readConfigFile* text field. Alternatively, click the folder icon to the right of the text field to navigate to the file.

On the command line. Assuming that *`/path/to/myConfig.conf`* is the full path and name of your configuration file, add the *`readConfigFile='/path/to/myConfig.conf'`* parameter to the `asciiTableWriter` task command, as in the following example:

```
myTable = asciiTableWriter(table = myTable, file='myFile.txt', readConfigFile='/
path/to/myConfig.conf')
```

Example 2.47. Writing a table to an ASCII file, specifying a previously-saved writing configuration file.

2.24. Parsers, formatters and templates

Parsers, formatters and templates are three tools you use to write data to text files and read data back from text files:

- **Parsers.** A *parser* defines rules to read a text file into HIPE. See Section 2.26 for the available types of parser, their features and how to configure them.
- **Formatters.** A *formatter* defines rules to write data from HIPE into a text file. See Section 2.27 for the available types of formatter, their features and how to configure them.
- **Table templates.** A *table template* describes the data to be read from, or written to, a text file. It defines the number of columns in the file, their name, the type and description of the data. While the parser defines general formatting rules, such as the character used to separate data values, the table template describes the data themselves. See Section 2.25 for how to create and configure a table template.

2.25. Creating and configuring table templates

A table template describes the number of columns in a text file, their titles, and the types, units and descriptions of the data they contain.

Creating a table template. To create a table template you have to specify at least the number of columns in the table. The following example creates a table template with three columns:

```
myTableTemplate = TableTemplate(3)
```

Example 2.48. Creating a TableTemplate with 3 columns.

Customising a table template. The following example shows how to customise a table template by providing column names and data types, units and descriptions. In each case you provide a list with the same number of elements as the number of columns in the table.

```
# Setting column names
myTableTemplate.names = ["Frequency", "Flux", "Error"]
# Setting data types
myTableTemplate.types = ["Double", "Double", "Double"]
# Setting data units
myTableTemplate.units = ["GHz", "mJy", "mJy"]
# Setting data descriptions
myTableTemplate.descriptions = ["Spectrum frequency", \
    "Spectrum flux", "Error on flux"]
```

Example 2.49. Customising a TableTemplate with column names, types, units and descriptions.

You can leave some of the values blank by passing empty strings:

```
myTableTemplate.descriptions = ["Spectrum frequency", "", ""]
```

Example 2.50. Setting column descriptions for a partial set of columns.

Note that no check is done on the data types and units you provide. If you input an invalid name, you will likely encounter problems when reading data into HIPE using the table template, or when using the imported data.

- For data types, use one of the following: `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` or `String`. For more information on these variable types and their ranges, see the *Scripting Guide*: Section 1.5.2 in *Scripting Guide*.
- For data units, use a name or abbreviation that HIPE can recognise. You can check whether HIPE recognises a unit name like this:

  ```
  from herschel.share.unit import *
  myUnit = Unit.parse("eV")
  print myUnit.isKnown()
  ```

  Example 2.51. Creating a unit variable and checking if it is built in the system.s

  If the result is `False`, HIPE cannot recognise the unit.

  For more information on measurement units, see the *Scripting Guide*: Section 2.6 in *Scripting Guide*.

Creating and configuring a table template in one step. You can also create and configure a table template in a single step, as shown by the following example:

```
template = TableTemplate(3, \
  names = ["Frequency", "Flux", "Error"], \
  types = ["Double", "Double", "Double"], \
```

```
units = ["GHz", "mJy", "mJy"], \
descriptions = ["Spectrum frequency", \
  "Spectrum flux", "Error on flux"])
```

Example 2.52. Creating and customising a TableTemplate in one step.

You can include all the additional parameters (*names*, *types* and so on) or just some of them.

2.26. Creating and configuring parsers for reading in data

A *parser* describes how each line of an ASCII file is broken up into table cell data, as well as which lines are ignored altogether. You use a parser only for *reading* an ASCII file into a table dataset, not for writing a table dataset to file. With a parser you can customise the following:

- How data values are separated in the file: what characters act as data delimiters, and whether data columns have fixed width or not.
- Whether to ignore lines beginning with a certain character or series of characters. These lines are said to be *commented out*.
- Whether to skip a number of lines at the beginning of the file.
- Whether to ignore white space at the beginning and end of each line.

Defining how data values are separated. There are three main cases, each corresponding to a different type of parser: `CsvParser`, `FixedWidthParser` and `RegexParser`.

- **Data values are separated by a single character.** Create your parser like in the following example, where the delimiter character is a comma:

```
myParser = CsvParser(delimiter=',')
```

Example 2.53. Creating a comma-separated CSV parser.

If the same character used as data separator also appears as part of some data value, HIPE will mistakenly interpret it as two data values. For example, if the data separator is a comma and a data value is `35,7`, HIPE will interpret it as two data values, `35` and `7`. You can surround the data value with double quotes, as in `"35,7"`, to tell HIPE to interpret it as a single value. To use a different character to "quote" data values, add the *quote* parameter:

```
myParser = CsvParser(delimiter=',', quote='$')
```

Example 2.54. Creating a CSV parser with a dollar sign for quoting values.

In this case a quoted data value would look like `$35,7$`.

- **Data values are organised in fixed-width columns.** Create your parser like in the following example, which defines three columns of three, seven and five characters:

```
myParser = FixedWidthParser(sizes=[3, 7, 5])
```

Example 2.55. Creating a parser that specifies the widths of the columns.

- **Data values are separated by more than one character, and column sizes are not fixed.** Create your parser like in the following example:

```
myParser = RegexParser(delimiter='\s+')
```

Example 2.56. Creating a parser based on a regular expression.

The value of the *delimiter* parameter is a *regular expression*. Regular expressions are a powerful way to describe complicated patterns of characters. You can find more information on regular expressions in Section 2.28. Here are a couple of examples:

- The expression \s+ means "one or more spaces".
- The expression \t+ means "one or more tab characters".

Defining the prefix used for commented lines. The default prefix for commented lines is #. To change it, create a parser with the *ignore* parameter defined. The following example defines // as prefix for commented lines:

```
myParser = CsvParser(ignore='//')
```

Example 2.57. Creating a CSV parser that ignores line starting with a specific string.

This parameter is accepted by CsvParser, FixedWidthParser and RegexParser.

Defining the number of lines to ignore at the beginning of the file. This option is useful if the file has a header with information you want HIPE to ignore. To define the number of lines to ignore, create a parser with the *skip* parameter defined. The following parser will ignore the first five lines of the file it reads:

```
myParser = FixedWidthParser(sizes=[7, 9], skip=5)
```

Example 2.58. Creating a fixed width parser that skips a number of header lines.

This parameter is accepted by CsvParser, FixedWidthParser and RegexParser.

Defining whether to ignore white space at the beginning and end of lines. This option is useful when data values are separated by spaces, and additional spaces at the beginning and end of lines could confuse the parser. Create a parser with the *trim=True* parameter:

```
myParser = RegexParser(delimiter='\s+', trim=True)
```

Example 2.59. Create a parser based on a regular expression that trims the lines of the file.

This parameter is accepted by CsvParser, FixedWidthParser and RegexParser.

2.27. Creating and configuring formatters for writing data

A table formatter controls how a table dataset is written to an ASCII file. With a table formatter you can customise the following:

- The character separating data values from each other.
- Whether columns in the file have a variable or fixed width.
- Whether to add a four-line header to the top of the file, with information on column names, data types, units and descriptions.
- Whether to add commented lines with the metadata of the table dataset.
- What character, or series of characters, to use for marking commented lines.

Defining the character separating data values. Creating a formatter like in the following example:

```
myFormatter = CsvFormatter(delimiter = ' ')
```

Example 2.60. Creating a CSV formatter that specifies a space character as a delimiter.

The previous example defines a single space as the character separating data values (the `delimiter` parameter). Note that you cannot specify more than one character. If you create a formatter without setting the `delimiter` parameter, a comma is used as default separating character:

```
myFormatter = CsvFormatter()
```

Example 2.61. Creating a CSV formatter with the default options.

Tip

To define a tab character as delimiter, use the following syntax

```
myFormatter = CsvFormatter(delimiter='\t')
```

Example 2.62. Creating a CSV formatter that uses a tab as a delimiter.

Defining whether columns have fixed with. To define columns with a fixed with you can create a `FixedWidthFormatter` instead of the `CsvFormatter` used in the previous example:

```
myFormatter = FixedWidthFormatter(sizes=[4,7,12])
```

Example 2.63. Creating a fixed width formatter.

The previous example defines three columns with a width of four, seven and twelve characters, respectively.

When columns are of fixed width, data values are always separated by spaces. To define columns of variable width, create a `CsvFormatter` instead of a `FixedWidthFormatter`.

Adding a four-line header to the file. The four lines show the names, data types, units of measurements and descriptions of each column. In case the unit of measurement or the description of a column is not set, the corresponding space is left blank. Create the formatter with the parameter *`header=True`*:

```
myFormatter = CsvFormatter(header=True)
```

Example 2.64. Creating a CSV formatter that includes a header.

This parameter is accepted by `CsvFormatter` and `FixedWidthFormatter`.

Adding commented lines with table dataset metadata. Create the formatter with the parameter *`commented=True`*:

```
myFormatter = FixedWidthFormatter(commented=True)
```

Example 2.65. Creating a fixed width formatter with commented metadata.

This parameter is accepted by `CsvFormatter` and `FixedWidthFormatter`.

Defining the prefix used for commented lines. The default prefix for commented lines is `#`. To change it, create a formatter with the *`commentPrefix`* parameter defined. The following example defines `//` as prefix for commented lines:

```
myFormatter = CsvFormatter(commentPrefix='//')
```

Example 2.66. Creating a CSV formatter with a custom comment prefix.

This parameter is accepted by `CsvFormatter` and `FixedWidthFormatter`.

Examples. The following examples combine all the features described in this section.

Define a formatter for writing a file with a four-line header, data values separated by spaces and metadata on commented lines prefixed by `$$$`:

```
myFormatter = CsvFormatter(header=True, delimiter=' ', commented=True,
 commentPrefix='$$$ ')
```

Example 2.67. Creating a CSV formatter that includes a header, delimited with spaces and with comments with a custom prefix.

Define a formatter for writing a file with a four-line header and four columns of five, seven, five and ten characters of width:

```
myFormatter = FixedWidthFormatter(header=True, sizes=[5,7,5,10])
```

Example 2.68. Creating a fixed width formatter.

2.28. Regular expressions

A *regular expression* is a concise and flexible means to match strings of text, such as particular characters or patterns of characters. Regular expressions are used to specify which lines of a file to skip, as discussed in Section 2.15, and with the `RegexParser` for specifying the delimiter between data fields (for example, to specify multiple spaces or tabs).

A detailed treatment of regular expressions is beyond the scope of this manual. The following table gives some examples to cover the cases you are most likely to need.

Table 2.1. Regular expressions.

Expression	Description
`\s`	Matches one whitespace character.
`\d`	Matches a digit.
`\w`	Matches an alphanumeric character, or an underscore character.
`[a-z]`	Matches a lowercase letter.
`[^x]`	Matches any character except *x*. For example, `[^\d]` matches anything except a digit.
x?	Matches zero or one occurrence of *x*.
*x**	Matches zero or more occurrences of *x*.
x+	Matches one or more occurrences of *x*.
x{*n*}	Matches exactly *n* occurrences of *x*.
x{*n*,}	Matches at least *n* occurrences of *x*.
x{*n*, *m*}	Matches at least *n* but no more than *m* occurrences of *x*.
^*x*	Matches *x* at the beginning of a string.
x$	Matches *x* at the end of a string.
|	*OR* operator. For example, `\d|\s` matches a digit *or* a whitespace character.

If you want to match a character that has a special meaning in regular expressions, put a backslash before it. For example, `\$` represents a dollar character, not the end of a string.

The following example shows some practical applications:

```
# Matches a line starting with one or more spaces followed by three hash characters.
^\s+#{3}
# Matches two uppercase letters separated by a comma.
```

```
[A-Z],[A-Z]
# Matches a colon followed by three or more characters different from a digit.
:[^\d]{3,}
# Matches a completely empty line, filled at most with whitespace characters.
^\s*$
# Matches an empty line, or a line starting with zero or more spaces
# followed by a hash character.
^\s*$|^\s*#
```

Chapter 3. Plotting

3.1. Getting started

This chapter provides several examples that show you how to create and customise plots from the command line and from the HIPE graphical interface.

Figure 3.1 summarises the main features of the plot packages. Additional features not shown in this image are, for example, plotting histograms and using custom axis types (logarithmic, right ascension and declination, and so on).

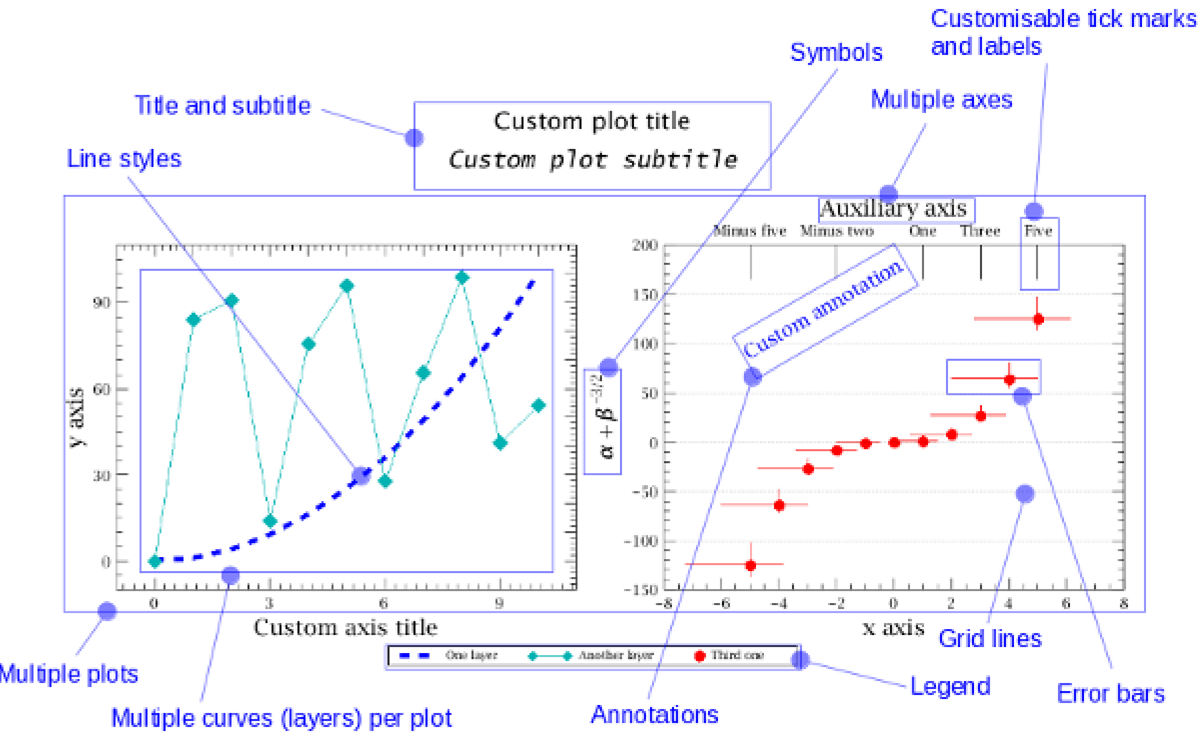


Figure 3.1. Some of the features of HIPE plots. If you are reading the HTML version of this manual, click on any of the blue labels to jump straight to the relevant section.

See Section 3.30 for the script used to generate this plot.

3.2. Creating a plot

Creating plots is done mainly from the command line in the *Console* view or via scripts in the *Editor* view. Data for plots must be of type `Numeric1d`, which includes one-dimensional arrays of numbers of any type (`Int1d` , `Float1d` or `Double1d`). The following two lines create two input arrays, containing the x and y coordinates of the data points to be plotted:

```
x = Double1d.range(11) # Creates array with values from 0.0 to 10.0
y = x*x
```

Example 3.1. Creating a double array and populating with a range of values.

Of course in real cases you will mostly use arrays with data from your observations rather than creating them from scratch.

The following example creates an empty plot and adds a *layer* of data. Each set of data in a plot corresponds to a layer.

```
myPlot = PlotXY()
myLayer = LayerXY(x,y)
myPlot.addLayer(myLayer)
```

Example 3.2. Creating a plot and adding layers to it in two steps.

You can also create the plot and the layer in one step:

```
myPlot = PlotXY(x, y)
```

Example 3.3. Creating a plot and adding a layer to it in one step.

Warning

For performance reasons, plots do not keep a copy of the data used to generate them. This means that you should not change the values after adding them to a plot. Otherwise, when the plot updates for any reason, for example due to a change of the window size, the plot will be corrupted.

The plot engine uses separate threads of execution to prevent HIPE from becoming unresponsive when large data sets are plotted. This means that, when a Jython script is running in batch mode, the rendering of the plot may finish after all or part of the script has run.

Resizing the plot. You can resize the window with the mouse or you can specify the desired window size once you have added layers to the plot. In the following example, the plot *myPlot* is resized to a width of 400 pixels and a height of 300 pixels:

```
myPlot.width = 400
myPlot.height = 300
```

Example 3.4. Setting the dimensions of the plot.

Tip

Why does my plot become huge after I add a layer? This happens when you create an empty plot and specify the `width` and `height` parameters without setting the `autoAdjustWindowSize` property to `0`, for example by issuing:

```
plot = PlotXY(width=600, height=400)
```

Example 3.5. Creating a plot with initial dimensions.

instead of:

```
plot = PlotXY(width=600, height=400, autoAdjustWindowSize=0)
```

Example 3.6. Creating a plot with initial dimensions and auto-adjust.

To avoid this problem, change the plot size only after having added all the layers.

3.3. Customising title and subtitle

Change the plot title and subtitle as in the following example:

```
myPlot.titleText = "Example plot"
myPlot.subtitleText = "Example subtitle"
```

Example 3.7. Setting title and subtitle text in a plot.

If you don't want to have plot title and subtitle you can switch them off:

```
myPlot.title.visible = 0
```

```
myPlot.subtitle.visible = 0
```

Example 3.8. Setting the visibility for a plot's title and subtitle.

Set these values to anything other than zero to switch title and subtitle back on.

About the `Label` and `Title` classes

There are multiple levels of a plot that can have a label or title, so it is important to distinguish between:

- Plot title and subtitle, both implemented in class `PlotTitle`.
- Note that instances of `SubPlot` don't have title nor subtitle.
- Layer name, displayed in the legend, is a field of class `LayerXY`.
- Axis labels or titles, implemented in class `AxisTitle` to avoid confusion with axis tick labels (below).
- Axis tick labels, implemented in class `AxisTickLabel`.
- Other plot labels, which are called Annotations throughout the documentation and code, so they should not be confused with any other type of label or title.

Methods. The title of a plot is modelled by the `PlotTitle` class. For more information see the developer's documentation for PlotTitle .

Procedure 3.1. Useful methods of the `PlotTitle` class. See Section 3.29 for the conventions used in this table.

1. **`setText(String text )`**

   ```
   # Java style
   myTitle.setText("A title")
   # Jython style
   myTitle.text = "A title"
   ```

   Example 3.9. Sets the text to be displayed.

2. **`setHalign(PComponentEngine.HAlign hAlign )`**

   ```
   from herschel.ia.gui.plot import PlotTitle
   # Java style
   myTitle.setHalign(PlotTitle.LEFT)
   # Jython style
   myTitle.halign = PlotTitle.LEFT
   ```

   Example 3.10. Sets the horizontal alignment. Possible values are LEFT, CENTER and RIGHT.

3. **`setValign(PComponentEngine.VAlign vAlign )`**

   ```
   from herschel.ia.gui.plot import PlotTitle
   # Java style
   myTitle.setValign(PlotTitle.MIDDLE)
   # Jython style
   myTitle.valign = PlotTitle.MIDDLE
   ```

   Example 3.11. Sets the vertical alignment. Possible values are MIDDLE, TOP and BOTTOM.

4. **`setPosition (PlotTitleEngine.Position position)`**

   ```
   from herschel.ia.gui.plot import PlotTitle
   # Java style
   myTitle.setPosition(PlotTitle.BOTTOMLEFT)
   # Jython style
   ```

```
myTitle.position = PlotTitle.BOTTOMLEFT
```

Example 3.12. Sets the position of the title. Possible values are BOTTOMCENTER, BOTTOMLEFT, BOTTOMRIGHT, TOPCENTER, TOPLEFT, TOPRIGHT and CUSTOMIZED. If set to CUSTOMIZED, the title position is controlled by the `setLocation` method.

5. **`setLocation(double x , double y )`**

```
myTitle.setLocation(5.5, 12.0)
```

Example 3.13. Sets the x and y location of the title, automatically switching the position to CUSTOMIZED.

6. **`setX(double x )`**

```
# Java style
myTitle.setX(12.5)
# Jython style
myTitle.x = 12.5
```

Example 3.14. Sets the x position of the title.

7. **`setXy(double[] xy )`**

```
# Java style
myTitle.setXy([12.5, 7.0])
# Jython style
myTitle.xy = [12.5, 7.0]
```

Example 3.15. Sets the x and y position of the title. Equivalent to `setLocation` .

Note

The `setVAlign` method sets the position of the title *within the title area* , not within the entire plot. In other words, `setVAlign(PlotTitle.BOTTOM)` will not put the title at the bottom of the plot. To achieve that effect use the `setPosition` method.

3.4. Managing layers

You can create and add other layers on top of the first one:

```
# Create the dataset
x1 = 10.0*Double1d.range(11)/10.0 - 5.0
y1 = x1**3.0
# Create and add the layer
myLayer2 = LayerXY(x1,y1)
myPlot.addLayer(myLayer2)
```

Example 3.16. How to create an additional layer in a plot.

The axis ranges are adjusted automatically and the new layer is given a different colour.

Accessing layers. You can access the layers of a plot as you would with the elements of an array:

```
firstLayer = myPlot[0]
```

Renaming layers. Use the following command:

```
myLayer.name = "New name"
```

Removing layers. To remove one, some or all layers from a plot do the following:

```
myPlot.removeLayer(0)  # Remove the first layer
myPlot.removeLayers([0, 1])  # Remove the first and second layer
```

```
myPlot.clearLayers()  # Remove all layers
```

Methods. Some of the methods that work on layers are listed in the tables below. For more information see the developer's documentation for the LayerXY class .

Procedure 3.2. Miscellaneous setters of the `LayerXY` class. See Section 3.29 for the conventions used in this table.

1. **`setName(text)`**

```
# Java style
myLayer.setName("A layer")
# Jython style
myLayer.name = "A layer"
```

Example 3.17. Changes the name (and thus the legend) of the layer.

2. **`setLine(Style line)`**

```
# Java style
myLayer.setLine(Style.DASHED)  # Name
myLayer.setLine(3)  # Number
# Jython style
myLayer.line = Style.DASHED  # Name
myLayer.line = 3  # Number
```

Example 3.18. Sets the line style of the layer. Possible values are NONE, SOLID, MARKED, DASHED and MARK_DASHED. You can also use the numbers 0, 1, 2, 3 and 4.

3. **`setSymbolSize(double size)`**

```
# Java style
myLayer.setSymbolSize(5.0)
# Jython style
myLayer.symbolSize = 5.0
```

Example 3.19. Sets the size of the layer symbols, in points.

4. **`setSymbolShape(SymbolShape shape)`**

```
# Java style
myLayer.setSymbolShape(SymbolShape.FTRIANGLE)  # Name
myLayer.setSymbolShape(15)  # Number
# Jython style
myLayer.symbolShape = SymbolShape.FTRIANGLE  # Name
myLayer.symbolShape = 15  # Number
```

Example 3.20. Sets the shape of the symbol. See Table 3.2 for the names and numbers of available symbols.

5. **`setStroke(float stroke)`**

```
# Java style
myLayer.setStroke(5.0)
# Jython style
myLayer.stroke = 5.0
```

Example 3.21. Sets the line thickness, in points. Only for line plots.

6. **`setStyle(Style style)`**

```
myStyle = Style(line = Style.NONE, symbol = Style.FSQUARE, symbolSize=3.5, \
color = java.awt.Color.blue)
# Java style
myLayer.setStyle(myStyle)
# Jython style
```

```
myLayer.style = myStyle
```

Example 3.22. Sets the style of the layer. The input parameter is an instance of the `Style` class. For more information on creating styles see Section 3.9 .

Procedure 3.3. Other methods of the `LayerXY` class. See Section 3.29 for the conventions used in this table.

1. **`addPoint(double x , double y)`**

   ```
   myLayer.addPoint(1.5, 2.8)
   ```

   Example 3.23. Adds a point to the layer.

2. **`addPoints(Ordered1dData x , Ordered1dData y)`**

   ```
   pointsX = Double1d([5.5, 6.0, 6.5])
   pointsY = Double1d([2.8, 3.0, 3.1])
   myLayer.addPoints(pointsX, pointsY)
   ```

   Example 3.24. Adds a set of points to the layer.

3. **`getCoords()`**

   ```
   # Java style
   print myLayer.getCoords()
   # Jython style
   print myLayer.coords
   ```

   Example 3.25. Waits for mouse click and returns the coordinates of the pointer. Returns a `double[]` .

4. **`getCoords(int n)`**

   ```
   print myLayer.getCoords(2)
   ```

   Example 3.26. Like the previous method, but this one does the job for `n` successive clicks. Returns a `double[][]` , that is, an array of double arrays. Each array holds the coordinates of a mouse click.

5. **`getDataCoords()`**

   ```
   # Java style
   print myLayer.getDataCoords()
   # Jython style
   print myLayer.dataCoords
   ```

   Example 3.27. The difference with respect to the previous two methods is that this time the coordinates of the layer point closer to the mouse pointer are returned. Returns a `double[]` .

6. **`getDataCoords(int n)`**

   ```
   print myLayer.getDataCoords(2)
   ```

   Example 3.28. Like the previous method, but this one does the job for `n` successive clicks. Returns a `double[][]` , that is, an array of double arrays. Each array holds the coordinates of the data point closest to each mouse click.

7. **`getPglid()`**

   ```
   # Java style
   print myLayer.getPglid()
   # Jython style
   print myLayer.pglid
   ```

   Example 3.29. Returns an `int` representing the index of the current layer inside the `PlotXY` .

8. **`setInLegend(boolean)`**

```
# Java style
print myLayer.setInLegend(True)
# Jython style
print myLayer.inLegend = True
```

Example 3.30. Sets whether the layer is shown in the legend. Getter method `isInLegend` available.

3.5. Showing and customising a legend

You can show or hide a legend showing the name of layers next to the symbol and line styles used to plot the corresponding data:

```
myPlot.legend.visible = 1  # Show the legend
myPlot.legend.visible = 0  # Hide the legend
```

Changing the legend name for a layer. In the following example, the data of layer *`myLayer`* is shown in the legend under the label *My data* :

```
myLayer.name = "My data"
```

Removing a layer from the legend. In the following example, the layer *`myLayer`* is removed from the legend:

```
myLayer.inLegend = 0
```

Showing or hiding the legend border. Use the following commands:

```
myPlot.legend.borderVisible = 1  # Show the border
myPlot.legend.borderVisible = 0  # Hide the border
```

Setting the number of columns. You can set the number of columns in which layers are organised in the legend, or let HIPE decide:

```
myPlot.legend.columns = 2  # Two columns
myPlot.legend.autoColumns = 1  # Automatic
```

Changing the position of the legend. To move the legend to another position, hover on it so that the mouse pointer turns to a four-arrows icon. Then click and drag. From the command line, use one of the following commands:

```
myPlot.legend.x = 3.0
myPlot.legend.y = 1.5
myPlot.legend.xy = [3.0, 1.5]
myPlot.legend.setLocation(3.0, 1.5)

from herschel.ia.gui.plot.renderer.PlotLegendEngine import Position
myPlot.legend.position = Position.TOPCENTER
```

Numeric values are in physical coordinates, not in plot coordinates. This means that the legend will not change position if you change the zoom level of the plot.

Possible values for `Position` are `BOTTOMCENTER` , `BOTTOMLEFT` , `BOTTOMRIGHT` , `LEFTBOTTOM` , `LEFTMIDDLE` , `LEFTTOP` , `RIGHTBOTTOM` , `RIGHTMIDDLE` , `RIGHTTOP` , `TOPCENTER` , `TOPLEFT` and `TOPRIGHT` .

Methods. The plot legend is modelled by the `PlotLegend` class. This class shares all the positioning methods listed in Procedure 3.1 . For more information see the developer's documentation .

3.6. Customising plot properties

Once you have created a plot, you can customise it by changing the properties of its components.

The most common way to do so is via the command line, and the rest of this chapter describes in detail all the available commands to change the appearance of your plot.

If you prefer to change properties via a graphical interface, you can use *Property Panel* window. To open this window, right-click on the plot and choose *Properties* from the context menu. Alternatively, invoke the `props()` method on your plot:

```
myPlot.props()
```

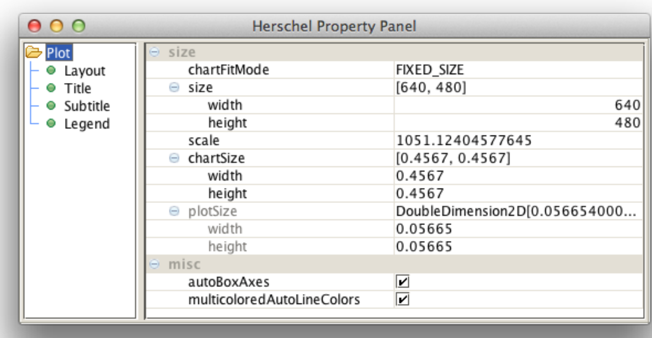


Figure 3.2. The *Property Panel* window.

The previous image shows the properties window for the plot created in Section 3.2 . The left-hand panel shows a tree structure with all the plot elements. As you add more elements (for example, additional layers) the tree is automatically updated. Click on any tree element to display the related properties in the right-hand panel.

Depending on the type of property, you can make changes by editing a text field, choosing a value from a drop-down list, ticking a checkbox or accessing an additional dialogue window. Most changes are immediately reflected in the plot. When you change the value of a text field, you have to press **Enter** or click elsewhere in the properties window for the change to appear.

3.6.1. Command line equivalents

When you change a plot property from the *Property Panel* window, HIPE writes the corresponding command to the *Console* view. For example, if you change the title of the plot, you will see an output like this in the *Console* view:

```
logger.info("myPlot.getTitle().setText(u'Another title')")
```

This is not the plot command, however. Instead, this command writes the plot command to the session log. If you switch to the *Log* view, you will see something like the following:

```
05 Nov 2012 15:51:41.788 INFO: myPlot.getTitle().setText(u'Another title')
```

After the `INFO:` label you have a command you can copy and paste to a script.

Tip

The `u` just before the `'Another title'` string makes it a *Unicode* string. This is optional unless your string includes symbols outside the ASCII character set.

Note that the command used in the previous example could be made more compact as follows:

```
myPlot.title.text = 'Another title'
```

These two styles are entirely equivalent and only a matter of preference. For more information see Section 3.29 .

3.7. Setting margins

Margins determine the space between the plot and the edges of the plot window. Margins are set automatically by HIPE, but you can change them, for example if you need more space to insert annotations outside the plot area.

The following example shows how to change the top margin of a plot called *myPlot* :

```
myPlot.getSubPlot(0).margin.autoMarginTop = False
myPlot.getSubPlot(0).margin.marginTop = 0.7
```

HIPE automatically resizes the plot window as margin sizes are changed.

You first have to set the `autoMarginTop` property to `False` . Set it back to `True` to reset the margin to its default value.

You can set the margins on the other three sides of the plot by using `Bottom` , `Right` and `Left` instead of `Top` .

Note that you do not operate on the plot directly, but get a *subplot* via the `getSubPlot` method. If you are working with a single plot, then you have one subplot with index 0. If you are creating multiple plots in the same window, as explained in Section 3.24 , you have several subplots and can set the margins of each one separately.

3.8. Saving and printing

To save a plot, right-click on it and choose *Save as* from the context menu. You can save the plot in PNG, EPS, PDF and JPEG formats. You can also do it via the command line:

```
myPlot.saveAsJPG("myfile.jpg") # JPEG format
myPlot.saveAsEPS("myfile.eps") # Encapsulated PS
myPlot.saveAsPNG("myfile.png") # PNG format
myPlot.saveAsPDF("myfile.pdf") # PDF format
```

To print a plot, right-click on it and choose either of these options from the context menu:

- *Print* sends data in vector format to the printer. This guarantees a high-quality result, but could take a long time for plots with many data points.
- *Print image* converts the plot to a bitmap image before sending it to the printer. This lowers the quality of the result, but guarantees fast printing even with a very high number of data points.

3.9. Setting line and symbol styles

To change the line style for a layer use this command:

```
myLayer.line = Style.NONE
```

You can also use a number instead of the style name:

```
myLayer.line = 0
```

The following table shows the available styles and corresponding numbers:

Table 3.1. Plot line styles

Name	Number	Description
Style.NONE	0	Symbols only
Style.SOLID	1	Solid line, no symbols
Style.MARKED	2	Symbols connected by solid line
Style.DASHED	3	Dashed line
Style.MARK_DASHED	4	Symbols connected by dashed line

To set the stroke thickness of a line use this command:

```
myLayer.style.stroke = 5.0
```

To change the plotting symbol and its size use this command:

```
myLayer.symbol = Style.FSQUARE
myLayer.symbolSize = 10
```

The available symbols are listed in the following table. A grey frame is drawn around the smaller symbols (dot and small arrows) to help gauge their size and position with respect to the data point. The frame is not part of the symbols.

Table 3.2. Symbol codes and images.

Name	Number	Image	Name	Number	Image
DOT	1		FSQUARE	16	
VCROSS	2		FDIAMOND	17	
DCROSS	3		FOCTAGON	18	
VDCROSS	4		UARROW	19	
CIRCLE	5		DARROW	20	
TRIANGLE	6		RARROW	21	
UTRIANGLE	7		LARROW	22	
SQUARE	8		DARROW_LARGE	23	
SQUARE_CROSS	9		UARROW_TRIAN-GLE	24	
DIAMOND	10		DARROW_TRIAN-GLE	25	
DIAMOND_CROSS	11		UARROW_TAIL	26	

Name	Number	Image	Name	Number	Image
OCTAGON	12		DARROW_TAIL	27	
STAR	13		RARROW_TAIL	28	
FCIRCLE	14		LARROW_TAIL	29	
FTRIANGLE	15				

Note

You can use either the code or the numeric value for the symbol. That is, `symbol = Style.FSQUARE` is equivalent to `symbol = 16`.

You can change shapes and sizes of symbols, line patterns and colours with a single command by using the `Style` class.

```
x = Double1d(range(100))
y = x*x
myPlot = PlotXY(x, y)
myPlot.style = Style(line = Style.NONE, symbol = Style.FSQUARE, symbolSize=3.5, \
 color = java.awt.Color.blue)
```

Example 3.31. Customising the appearance of the different plot symbols.

In the above example the style is automatically applied to the first layer of the plot. For plots with multiple layers, you can apply the style to one specific layer:

```
y2 = x*2
# Adding a layer
myPlot.addLayer(LayerXY(x, y2))
# Applying a style to the second layer of the plot.
# This style only changes the line pattern and thickness.
myPlot[1].style = Style(line = Style.DASHED, stroke = 0.5)
```

3.10. Customising axes

To initialise a plot with zeroed data and a sample axis, so you can work with the examples below, the following two lines are enough. Take into account that some of the examples require importing additional classes and are thus not self-contained.

```
# Initialise the plot and the x-axis
myPlot=LayerXY(Double1d(0),Double1d(0))
myPlot.xaxis=Axis(range=[5e13,3e17], titleText="radius (cm)")
```

Example 3.32. Adding a customised range and title to a plot axis.

Changing the axis labels:

```
myPlot.xaxis.titleText = "X-values"
myPlot.yaxis.titleText = "Y-values"
```

Changing the axis ranges:

```
myPlot.xaxis.range = [-2.0,2.0]
myPlot.yaxis.range = [-10.0,10.0]
```

To go back to the auto range:

```
myPlot.xaxis.autoRange = 1
myPlot.yaxis.autoRange = 1
```

Changing the spacing of the tick marks:

```
myPlot.xaxis.tick.interval = 3.0
myPlot.yaxis.tick.interval = 30.0
```

Changing the number of minor ticks between each couple of major ticks:

```
myPlot.xaxis.tick.minorNumber = 4
myPlot.yaxis.tick.minorNumber = 5
```

Changing the axis from linear to logarithmic:

```
myPlot.xaxis.setAxisType(AxisType.LOG)
myPlot.xaxis.setAxisType(AxisType.LINEAR)
```

Note

Any negative values are ignored when converting an axis to logarithmic. Negative values are plotted again when returning to linear.

Methods. Some methods that work on axes are listed in the tables below. For a complete reference see the developer's documentation for the Axis class .

Procedure 3.4. Useful methods of the `Axis` class. See Section 3.29 for the conventions used in this table.

1. **`setAutoRange(boolean flag)`**

   ```
   # Java style
   myAxis.setAutoRange(True)
   # Jython style
   myAxis.autoRange = True
   ```

   Example 3.33. If `flag` is true, adjusts the range of the specified axis so that all data points will be shown.

2. **`setRange(double low, double high)`**

   ```
   myAxis.setRange(10.0, 30.0)
   ```

   Example 3.34. Sets the range of the axis. The lower and upper limit are passed as separate `double` parameters. Note that there is no "Jython style" example because lists and tuples in Jython use the same syntax. See the row just below for the example.

3. **`setRange([lower, upper])`**

   ```
   # Java style
   myAxis.setRange([10.0, 30.0])
   # Jython style
   myAxis.range = [10.0, 30.0]
   ```

   Example 3.35. Set the range of the specified axis to values between lower and upper. Note that instead of two arguments for the lower and upper limits, there is one array argument containing both values.

4. **`setGridLines(boolean flag)`**

   ```
   # Java style
   myAxis.setGridLines(True)
   # Jython style
   myAxis.getTick().setGridLines(True)
   ```

   Example 3.36. Show grid lines for the specified axis if `flag` is true, hide the grid lines otherwise.

5. **`setType(AxisConstants.Type type)`**

```
# Java style
myAxis.setType(Axis.LOG)
# Jython style
myAxis.type = Axis.LOG
```

Example 3.37. Sets the axis type. Available types are `LINEAR` , `LOG` , `DATE` , `RIGHT_ASCENSION` and `DECLINATION` .

6. **`getOrientation()`**

```
# Java style
print myAxis.getOrientation()
# Jython style
print myAxis.orientation
```

Example 3.38. Gets the axis orientation, either `HORIZONTAL` or `VERTICAL` . Setter method *not* available.

7. **`setLinear()`**

```
myAxis.setLinear()
```

Example 3.39. Sets the axis to a linear scale. Equivalent to `setType(Axis.LINEAR)` .

8. **`setLog()`**

```
myAxis.setLog()
```

Example 3.40. Sets the axis to a logarithmic scale. Equivalent to `setType(Axis.LOG)` .

9. **`setInverted(boolean flag)`**

```
# Java style
myAxis.setInverted(True)
# Jython style
myAxis.inverted = True
```

Example 3.41. Sets whether values on the axis are displayed in inverted order (for instance, right to left for the abscissa).

10. **`setPosition (AxisConstants.Position position)`**

```
# Java style
myAxis.setPosition(Axis.BOTTOM)
# Jython style
myAxis.position = Axis.BOTTOM
```

Example 3.42. Sets the position of the axis with respect to the plot. Possible values are `TOP` or `BOTTOM` for abscissa axis and `LEFT` or `RIGHT` for ordinate axis. Get method available.

The title of an axis is modelled by the `AxisTitle` class. For more information see the developer's documentation for AxisTitle .

Procedure 3.5. Useful methods of the `AxisTitle` class. See Section 3.29 for the conventions used in this table.

- **`setText(String text)`**

```
# Java style
myTitle.setText("A title")
# Jython style
```

```
myTitle.text = "A title"
```

Example 3.43. Sets the text to be displayed.

Ticks are modelled by the `AxisTick` class. Usually you access the ticks from an `Axis` object:

```
myTicks = myAxis.getTick()  # Java style
myTicks = myAxis.tick  # Jython style
```

For more information see the developer's documentation of the AxisTick class .

Procedure 3.6. Some methods of the `AxisTick` class. See Section 3.29 for the conventions used in this table.

1. **`setHeight(double size)`**

```
# Java style
myTicks.setHeight(2.0)
# Jython style
myTicks.height = 2.0
```

Example 3.44. Sets the physical height of the major ticks.

2. **`setInterval(double interval)`**

```
# Java style
myTicks.setInterval(0.3)
# Jython style
myTicks.interval = 0.3
```

Example 3.45. Sets the interval between major ticks, in axis units.

3. **`setSide(AxisTickSide side)`**

```
# Java style
myTicks.setSide(AxisTickSide.BOTH)
# Jython style
myTicks.side = AxisTickSide.BOTH
```

Example 3.46. Sets the side of the axis on which the ticks are drawn. Possible values are `INWARD` , `OUTWARD` and `BOTH` .

4. **`setMinorNumber(int number)`**

```
# Java style
myTicks.setMinorNumber(3)
# Jython style
myTicks.minorNumber = 3
```

Example 3.47. Sets the number of minor ticks displayed between two major ticks.

5. **`setAutoAdjustNumber(boolean flag)`**

```
# Java style
myTicks.setAutoAdjustNumber(True)
# Jython style
myTicks.autoAdjustNumber = True
```

Example 3.48. Sets whether the number of ticks on the axis is adjusted automatically to avoid overlapping labels. Getter method `isAutoAdjustNumber` available.

6. **`setGridLines(boolean flag)`**

```
# Java style
myTicks.setGridLines(True)
# Jython style
```

```
myTicks.gridLines = True
```

Example 3.49. Sets whether grid lines are displayed for major ticks. Getter method `isGridLines` available.

Tick labels are modelled by the `AxisTickLabel` class. Usually you access the tick labels from an `AxisTick` object:

```
myTickLabels = myTicks.getLabel()  # Java style
myTickLabels = myTicks.label   # Jython style
```

For more information see the developer's documentation of the AxisTickLabel class .

Procedure 3.7. Some methods of the `AxisTickLabel` class. See Section 3.29 for the conventions used in this table.

1. **`setColor(java.awt.Color colour)`**

   ```
   # Java style
   myTickLabels.setColor(java.awt.Color.green)
   # Jython style
   myTickLabels.color = java.awt.Color.green
   ```

   Example 3.50. Sets the colour of labels.

2. **`setFont(java.awt.Font font)`**

   ```
   myFont = java.awt.Font("Arial", java.awt.Font.PLAIN, 15)
   # Java style
   myTickLabels.setFont(myFont)
   # Jython style
   myTickLabels.font = myFont
   ```

   Example 3.51. Sets the font of labels.

3. **`setFontSize(double size)`**

   ```
   # Java style
   myTickLabels.setFontSize(12.0)
   # Jython style
   myTickLabels.fontSize = 12.0
   ```

   Example 3.52. Sets the physical size of labels.

4. **`setInterval(int value)`**

   ```
   # Java style
   myTickLabels.setInterval(2)
   # Jython style
   myTickLabels.interval = 2
   ```

   Example 3.53. Sets the interval (in ticks) between successive labels. For example, a value of two displays a label on every other tick.

5. **`setOrientation(int value)`**

   ```
   myTickLabels.setOrientation(1)
   ```

   Example 3.54. Sets the orientation of the labels (0 for horizontal, 1 for vertical).

6. **`setFixedStrings(String[] labels)`**

   ```
   # Java style
   myTickLabels.setFixedStrings(["One", "Two", "Three"])
   # Jython style
   ```

```
myTickLabels.fixedStrings = ["One", "Two", "Three"]
```

Example 3.55. Replaces the current labels with the values in an array of `String` objects.

7. **`setSide(AxisLabelSide size)`**

```
from herschel.ia.gui.plot.renderer import AxisLabelSide
# Java style
myTickLabels.setSide(AxisLabelSide.INWARD)
# Jython style
myTickLabels.side = AxisLabelSide.INWARD
```

Example 3.56. Sets the side of the axis on which the labels are drawn. Possible values are `INWARD` and `OUTWARD`.

The `LayerXY` class also has methods related to axes. These are listed in the following table. All the methods listed in the table can equally be applied to the y-axis by replacing `X` with `Y`.

Procedure 3.8. Axis-related methods of the `LayerXY` class. See Section 3.29 for the conventions used in this table.

1. **`setXaxis(Axis axis)`**

```
# Java style
myLayer.setXaxis(myAxis)
# Jython style
myLayer.xaxis = myAxis
```

Example 3.57. Sets the x axis to the specified `Axis` instance. The x axis will be reinstantiated with its default settings plus whatever is indicated in the `Axis` instance. So any prior manipulations of the axis are lost.

2. **`setXrange(double[] range)`**

```
# Java style
myLayer.setXrange([-2.0, 10.5])
# Jython style
myLayer.xrange = [-2.0, 10.5]
```

Example 3.58. Sets the range of the x axis.

3. **`setXtitle(String title)`**

```
# Java style
myLayer.setXtitle("A title")
# Jython style
myLayer.xtitle = "A title"
```

Example 3.59. Sets the title of the x axis.

4. **`setXtype(AxisConstants.Type type)`**

```
# Import statement not needed if using numeric values
from herschel.ia.gui.plot.renderer.axtype import AxisType
# Java style
myLayer.setXAxisType(AxisType.LOG)  # Name
# Jython style
myLayer.setXAxisType(AxisType.LOG)  # Name
```

Example 3.60. Sets the type of the x axis. Available types are LINEAR and LOG.

5. **`setXy(Ordered1dData[] xy)`**

```
# New X values are 1.0, 2.0, 3.0. New Y values are 2.5, 2.8, 3.1.
# Java style
myLayer.setXy([Double1d([1.0, 2.0, 3.0]), Double1d([2.5, 2.8, 3.1])])
```

```
# Jython style
myLayer.xy = [Double1d([1.0, 2.0, 3.0]), Double1d([2.5, 2.8, 3.1])]
```

Example 3.61. Sets the x and y values, passed as elements of an "array of arrays" of size two. Get method available. Note that there is no `setYx` method!

6. **`setXy(Ordered1dData x, Ordered1dData y)`**

```
# New X values are 1.0, 2.0, 3.0. New Y values are 2.5, 2.8, 3.1.
myLayer.setXy(Double1d([1.0, 2.0, 3.0]), Double1d([2.5, 2.8, 3.1]))
```

Example 3.62. Sets the x and y values, passed as two separate arrays. Note that there is no `setYx` method!

7. **`setY(Ordered1dData y)`**

```
# New Y values are 2.5, 2.8, 3.1.
# Java style
myLayer.setY(Double1d([2.5, 2.8, 3.1]))
# Jython style
myLayer.y = Double1d([2.5, 2.8, 3.1])
```

Example 3.63. Sets the ordinate values. Get method available. Note there is a `getX` method but not a `setX` method.

8. **`shareXaxis(Axis axis)`**

```
myLayer.shareXaxis(myAxis)
```

Example 3.64. Removes the x axis and uses the given axis as a shared x axis.

3.11. Drawing grid lines

Enable grid lines as in the following example:

```
myPlot.xaxis.tick.gridLines = 1
myPlot.yaxis.tick.gridLines = 1
```

Set the `gridLines` value to zero to disable grid lines.

Grid lines are drawn at major tick marks.

3.12. Managing annotations

The following example shows how to add, modify and remove annotations:

```
# Add an annotation at x = 6.5, y = -10, with text "My text".
# The coordinates correspond to the lower left corner of the annotation.
myPlot.addAnnotation(Annotation(6.5,-10,"My text",color=java.awt.Color.GREEN))
# List all the annotations of the plot (just one in this case).
print myPlot.getAnnotations()
# array(herschel.ia.gui.plot.Annotation, [Annotation 0])
# Every annotation is identified by an index, starting from zero.
# Print the text of an annotation given its index.
print myPlot.getAnnotation(0).text  # My text
# Modify an annotation
myAnnotation = myPlot.getAnnotation(0)
myAnnotation.text = "Another text"  # Change text
myAnnotation.x = 7.3  # Change coordinates
myAnnotation.y = -5
# Delete an annotation given its index.
myPlot.removeAnnotation(0)
```

Methods. The following table lists methods of the `Annotation` class. For more information see the developer's documentation .

Procedure 3.9. Methods of the `Annotation` class. See Section 3.29 for the conventions used in this table.

1. **`Annotation()`**

   ```
   myAnnotation = Annotation()
   ```

   Example 3.65. Creates an empty annotation.

2. **`Annotation(String text)`**

   ```
   myAnnotation = Annotation("My text")
   ```

   Example 3.66. Creates an annotation with the given text.

3. **`Annotation(double x , double y , String text)`**

   ```
   myAnnotation = Annotation(1.5, 10.0, "My text")
   ```

   Example 3.67. Creates an annotation with the given position and text.

4. **`setAngle(double angle)`**

   ```
   # Java style
   myAnnotation.setAngle(40.0)
   # Jython style
   myAnnotation.angle = 40.0
   ```

   Example 3.68. Sets the position angle, in degrees, counterclockwise.

5. **`setHalign(PComponentEngine.HAlign hAlign)`**

   ```
   from herschel.ia.gui.plot.renderer import PComponentEngine
   # Java style
   myAnnotation.setHalign(PComponentEngine.HAlign.LEFT)
   # Jython style
   myAnnotation.halign = PComponentEngine.HAlign.LEFT
   # Accepted values: LEFT, RIGHT and CENTER.
   ```

   Example 3.69. Sets the horizontal alignment.

6. **`setValign(PComponentEngine.VAlign vAlign)`**

   ```
   from herschel.ia.gui.plot.renderer import PComponentEngine
   # Java style
   myAnnotation.setValign(PComponentEngine.VAlign.BOTTOM)
   # Jython style
   myAnnotation.valign = PComponentEngine.VAlign.BOTTOM
   # Accepted values: TOP, BOTTOM and MIDDLE.
   ```

   Example 3.70. Sets the vertical alignment.

7. **`setX(double x)`**

   ```
   # Java style
   myAnnotation.setX(12.5)
   # Jython style
   myAnnotation.x = 12.5
   ```

   Example 3.71. Sets the x position.

8. **`setXy(double x , double y)`**

   ```
   myAnnotation.setXy(12.5, -3.5)
   ```

   Example 3.72. Sets the x and y position.

9. **`setText(String text)`**

```
# Java style
myAnnotation.setText("Alternative text")
# Jython style
myAnnotation.text = "Alternative text"
```

Example 3.73. Sets the text of the annotation.

10. **int `getId()`**

```
# Java style
print myAnnotation.getId()
# Jython style
print myAnnotation.id
```

Example 3.74. Gets the unique id of the annotation. No setter available.

The `PlotXY` class also has methods for handling annotations. These are listed in the following table.

Procedure 3.10. Methods of the `PlotXY` class for handling annotations. See Section 3.29 for the conventions used in this table.

1. **`addAnnotation(Annotation annotation)`**

```
myAnnotation = Annotation(6.5,-10,"My text",color=java.awt.Color.GREEN)
myPlot.addAnnotation(myAnnotation)
```

Example 3.75. Adds an Annotation object to the layer.

2. **`addAnnotations(Annotation[] annotations)`**

```
firstAnnotation = Annotation(6.5,-10,"My text",color=java.awt.Color.GREEN)
secondAnnotation = Annotation(3.5,-5,"Another text",color=java.awt.Color.BLUE)
myPlot.addAnnotations([firstAnnotation, secondAnnotation])
```

Example 3.76. Adds several Annotation objects to the layer. The input Annotations are passed as an array.

3. **`setAnnotation(int id , Annotation annotation)`**

```
myAnnotation = Annotation(6.5,-10,"My text",color=java.awt.Color.GREEN)
myPlot.setAnnotation(1, myAnnotation)
```

Example 3.77. Sets an annotation to a given id, replacing what was there before.

4. **`setAnnotations(Annotation[] annotations)`**

```
firstAnnotation = Annotation(6.5,-10,"My text",color=java.awt.Color.GREEN)
secondAnnotation = Annotation(3.5,-5,"Another text",color=java.awt.Color.BLUE)
myPlot.setAnnotations([firstAnnotation, secondAnnotation])
```

Example 3.78. Replaces all the annotations with the ones provided in the array.

5. **`getAnnotation(int i)`**

```
myAnnotation = myPlot.getAnnotation(1)
```

Example 3.79. Retrieves one annotation from the layer.

6. **`getAnnotations()`**

```
# Java style
myAnnotations = myPlot.getAnnotations()
# Jython style
```

```
myAnnotations = myPlot.annotations
```

Example 3.80. Retrieves all the annotations from the layer. The annotations are returned as an array.

7. **`removeAnnotation(int id)`**

```
myPlot.removeAnnotation(1)
```

Example 3.81. Removes the annotation with the specified id.

8. **`clearAnnotations()`**

```
myPlot.clearAnnotations()
```

Example 3.82. Removes all the annotations.

3.13. Drawing filled areas

To enable area filling for a layer, use the following command, assuming that `layer` is a variable representing your layer:

```
myLayer.style.fillEnabled = 1
```

Set this property to zero to disable area filling.

Fill closure. You can determine the area to be filled by setting the *fill closure type* :

```
from herschel.ia.gui.plot.renderer import FillClosureType
myLayer.style.fillClosureType = FillClosureType.TOP
```

There are five options available: `SELF` , `TOP` , `BOTTOM` , `LEFT` and `RIGHT` .

The `SELF` option is the default. The filled area is defined by the plot curve and a straight line connecting the first and last points of the curve.

The effects of the five options are shown in the following image:

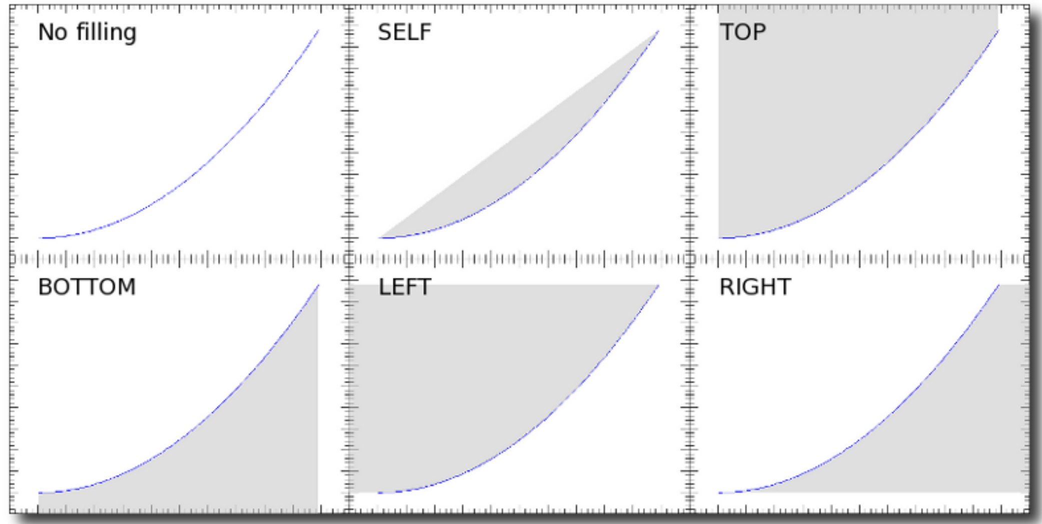


Figure 3.3. Available filling closure types.

Fill colour. The area is filled with a light grey colour by default. To change the colour, issue the following command:

```
myLayer.style.fillPaint = java.awt.Color.YELLOW
```

For more information on the `java.awt.Color` class see Section 3.25 .

Hatching. You can use a hatched pattern for filling by defining a `LineHatchPaint` object:

```
paint = LineHatchPaint(55)  # Lines tilted 55 degrees
myLayer.style.fillPaint = paint
```

There are various ways to create a `LineHatchPaint` object:

```
LineHatchPaint(angle)
LineHatchPaint(width, angle, spacing)
LineHatchPaint(color, width, dash, angle, spacing)
```

Angles are measured in degrees, widths and spacings in pixels. The *colour* parameter must be of type `java.awt.Color` , while the *dash* parameter is an array of lengths in pixels describing the dash pattern. The following examples clarifies the syntax:

```
LineHatchPaint(55)
LineHatchPaint(3, 75, 5)
LineHatchPaint(java.awt.Color.RED, 1, [5, 3, 1], 55, 5)
```

The results are shown in the following figure:

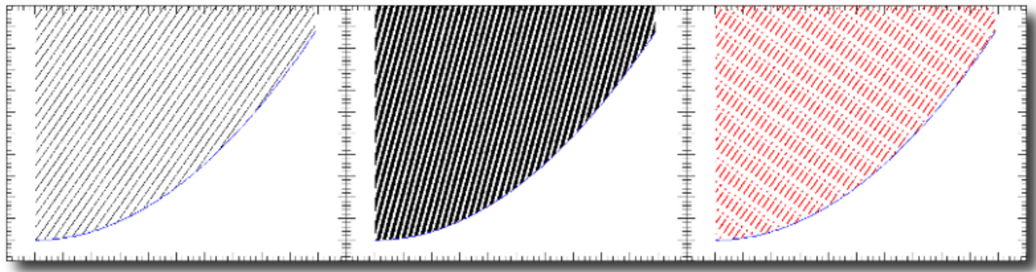


Figure 3.4. The filling patterns produced by the three `LineHatchPaint` objects shown in the previous example.

3.13.1. Drawing filled areas between curves

To fill areas between curves, the required code is more complex. Following you can find a complete example that fills the area between the curves *sinc(x)+0.5* and *sinc(x)-0.5* with a gray colour. It also strokes the curve *sinc(x)* in blue with a brush 3 pixels wide.

```
from herschel.ia.gui.plot.renderer import FillClosureType
from java.awt import Color

x = -4*Math.PI + 8.0*Math.PI*Float1d.range(101)/100.0
y = SINC(x)
yerr = 0.5
yUp = y + yerr
yDo = y - yerr

plot = PlotXY()

# fill area
ax=x.copy().append(REVERSE(x))
ay=yUp.copy().append(REVERSE(yDo))
# there is no way to omit lines and symbols together, so we set symbols to a very
 small size
la=LayerXY(ax,ay,line=Style.NONE, symbolSize=0.01)
la.style.fillClosureType=FillClosureType.SELF
la.style.fillPaint=Color(234,234,234)
la.style.fillEnabled=1
plot.addLayer(la)
l1 = LayerXY(x,y)
l1.setStroke(3)
l1up = LayerXY(x,yUp)
```

```
l1up.setColor(java.awt.Color.GRAY)
l1do = LayerXY(x,yDo)
l1do.setColor(java.awt.Color.GRAY)
plot.addLayer(l1)
plot.addLayer(l1up)
plot.addLayer(l1do)
```

Example 3.83. Filling the areas between the curves.

The result of this code is displayed in the following figure:

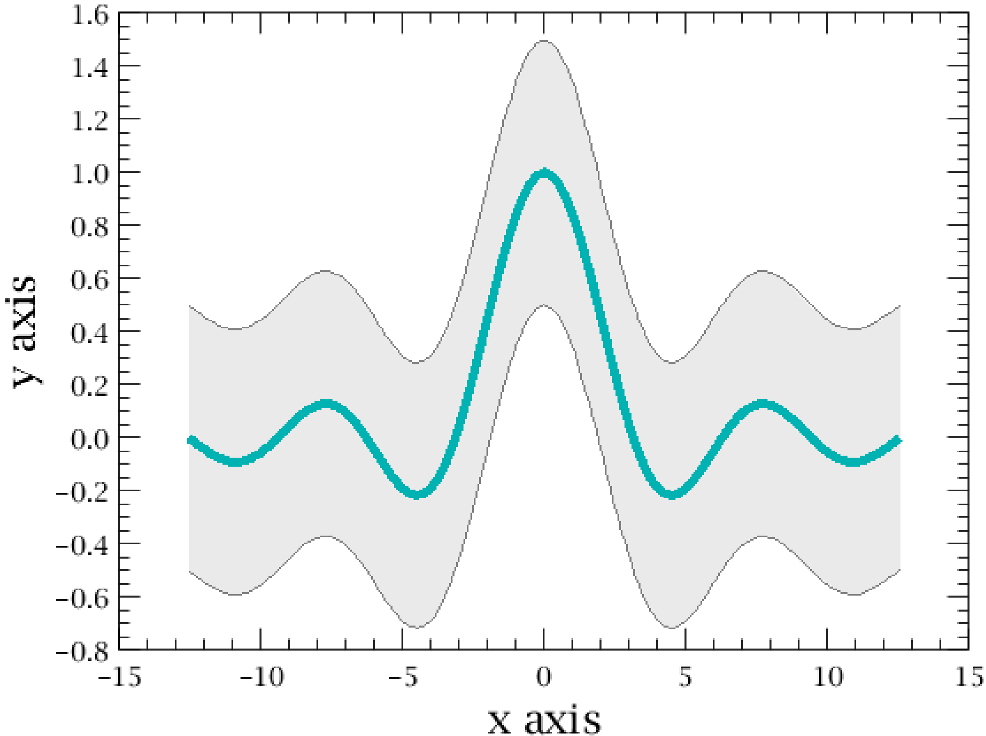


Figure 3.5. The filled area between the *sinc(x)* curves.

3.14. Drawing a straight line

Use the `LineAnnotation` class to add a horizontal or vertical line to a layer of your plot, as shown by the following example:

```
# Creating two horizontal lines at Y = 10 and 12
hline_a = LineAnnotation (LineAnnotation.YLINE, 10)
hline_b = LineAnnotation (LineAnnotation.YLINE, 12)
# Creating a vertical line at X = 15.5
vline = LineAnnotation (LineAnnotation.XLINE, 15.5)
# Adding all three lines to a layer
myLayer.addLineAnnotation(hline_a)
myLayer.addLineAnnotation(hline_b)
myLayer.addLineAnnotation(vline)
```

Each line is identified by an integer number, starting from zero. In the above example, *`hline_a`* is identified by 0, *`hline_b`* by 1 and *`vline`* by 2. You need to refer to these numbers if you want to modify or delete a line, as shown in the following examples.

`LineAnnotation` objects accept all the methods for colours, fonts and visibility described in Section 3.26 . In addition, you can change the line width and dash pattern as shown by the following example:

```
# Changing line width to 2.0 in plot units
hline_a.lineWidth = 2.0
# Applying changes (hline_a is number 0)
myLayer.setLineAnnotation(1, hline_a)
# Setting dashes of length 2.0 with gaps of length 1.0
hline_b.dashArray = [2.0, 1.0]
# Applying changes (hline_b is number 1)
myLayer.setLineAnnotation(0, hline_b)
```

Use `removeLineAnnotation` to delete a line:

```
# Removing vline, identified by number 2
myLayer.removeLineAnnotation(2)
```

3.14.1. Drawing an arbitrarily-positioned straight line

`LineAnnotation` objects can only be plotted vertically or horizontally. To arbitrarily position a straight line you need to employ the `stroke` property of `LayerXY` . See below for a minimal example.

```
from java.awt.Color import *

x = Double1d.range(11)
y = x*x
myPlot = PlotXY()
myPlot.autoBoxAxes = 1
myLayer = LayerXY(x, y)
myPlot.addLayer(myLayer, 0, 0)
x1_begin = 2.0
y1_begin = 2.0
x1_end = 6.0
y1_end = 8.0
l1 =
 LayerXY(Double1d([x1_begin,x1_end]),Double1d([y1_begin,y1_end]),color=java.awt.Color.RED,stroke
 = 2)
myPlot.addLayer(l1,0,0)
x2_begin = 7.0
y2_begin = 60.0
x2_end = 9.0
y2_end = 75.0
l2 =
 LayerXY(Double1d([x2_begin,x2_end]),Double1d([y2_begin,y2_end]),color=java.awt.Color.BLUE,stroke
 = 2)
myPlot.addLayer(l2,0,0)
```

Example 3.84. Drawing an arbitrarily-positioned straight line using LayerXY.

3.15. Customising auxiliary axes

Auxiliary axes are those that appear opposite the plot main axes. The X axis at the top of the plot and the Y axis on the right side of the plot are auxiliary axes.

You can get hold of an auxiliary axis with the `getAuxAxis` method. The following example shows how to set different units and ticks for the auxiliary axes. The main X axis is a wavelength measured in micrometers and the top X axis is changed to show the wavenumber (1/wavelength) in cm $^{-1}$.

```
# STEP 1 - Create a plot
x = 100.0 + 6*Double1d.range(100)
y = x*x
myPlot = PlotXY()
myLayer = LayerXY(x,y)
myPlot.addLayer(myLayer)
myPlot.xaxis.titleText = "Wavelength ($\\mu$m)"
myPlot.yaxis.titleText = "F$_\\lambda$ (Jy)"

# STEP 2
```

```
# Get the auxiliary axis (top X).
xaux = myPlot.xaxis.getAuxAxis(0)
# Make it free so that ticks do not have to be identical to the main axis.
xaux.tickIdentical = 0
# Remove the autoadjustment of the ticks.
xaux.tick.autoAdjustNumber = 0
# Make the axis title visible and set the title text.
xaux.title.visible = 1
xaux.titleText = "Wavenumber (cm$^{-1}$)"

# STEP 3
# Get the top axis labels and make them visible.
xauxlab = xaux.tick.label
xauxlab.visible = 1
# Set the new values for major and minor ticks.
# Major ticks: 10, 20, 30, 40, 50.
vals = Double1d(range(10,51,10))
# Minor ticks: 15, 25, 35, 45.
valsMinor = Double1d(range(15,50,10))
# Convert the wavenumbers to wavelength in microns.
wl_vals = 1.0e4/vals
wl_valsMinor = 1.04/valsMinor
xaux.tick.setFixedValues(wl_vals, wl_valsMinor)
# Create strings from values to act as tick labels.
svals = ['%.1f'%v for v in vals]
xauxlab.fixedStrings = svals
```

This is the resulting plot:

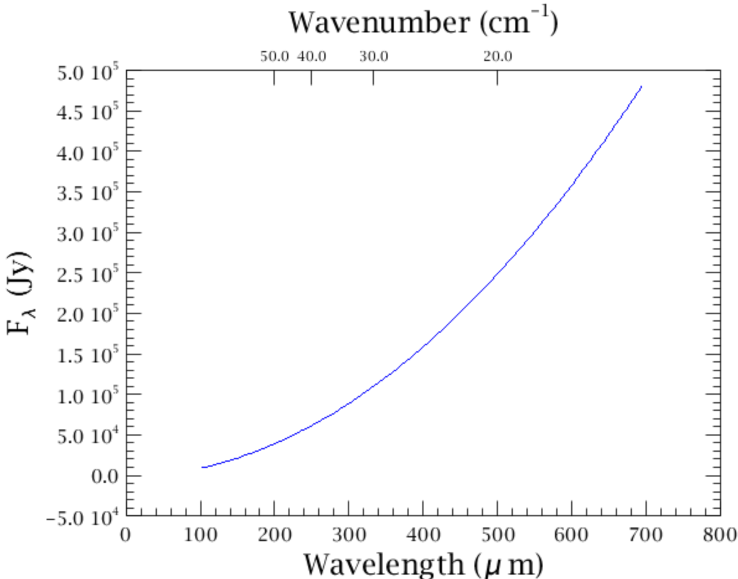


Figure 3.6. Plot with a customised auxiliary axis.

There is an easy way to construct the auxiliary axis if it is reciprocal to the main axis. Continuing from the end of step one, you can show the frequency in GHz, which is c/wavelength, where c is the speed of light in vacuum:

```
c = 299792458.0 # m/s
xaux = ReciprocalAuxAxis(1.0e-3*c)
myPlot.getXaxis().removeAuxAxis(0)
myPlot.getXaxis().addAuxAxis(xaux)
xaux.setTitleText("Frequency (GHz)")
xaux.getTick().setMinorNumber(4)
```

This way the ticks are automatically adjusted. The argument of `ReciprocalAuxAxis` is the factor used to convert from the main axis with its unit to the auxiliary axis and its unit. That is: 1 GHz = (1.0e-3*c)/µm.

3.16. Changing the thickness of axes

The following example shows how to change the thickness of the X axis of a plot called `myPlot`. The first two commands set the thickness of the main X axis and of its ticks to three pixels. The third and fourth commands do the same for the auxiliary X axis.

```
myPlot.xaxis.setLineWidth(3)
myPlot.xaxis.getTick().setLineWidth(3)
myPlot.xaxis.getAuxAxis(0).setLineWidth(3)
myPlot.xaxis.getAuxAxis(0).getTick().setLineWidth(3)
```

You can do the same for the main and auxiliary Y axes by replacing `xaxis` with `yaxis`.

3.17. Adding error bars

To add vertical and/or horizontal error bars, you first need to create arrays containing the error values, as shown by the following example:

```
# Creating arrays with error values
xerr = SQRT(x)
yerrup = SQRT(y)
yerrlow = SQRT(y) / 2.0

# Setting error bars
myLayer.errorX = [xerr, xerr]  # Setting the left and right error values
myLayer.errorY = [yerrlow, yerrup]  # Setting the lower and upper error values
```

Note how, in the previous example, the error values at each side of the data points are the same for horizontal error bars but different for vertical error bars.

The arrays containing the error values do not need to be the same size as the arrays containing the data points. If there are more errors values than data points, the surplus error values are ignored. If there are fewer error values than data points, no error is set for the surplus data points.

In the latter case, you can add error values to the ones already set:

```
# Creating dummy additional error values
xerraddleft = Double1d([0.3, 0.8])
xerraddright = Double1d([0.7, 0.4])
yerradd = Double1d([0.7, 0.2])

# Appending error bars
myLayer.appendErrorX(xerraddleft, xerraddright)
myLayer.appendErrorY(yerradd, yerradd)
```

Procedure 3.11. Methods for handling error bars in layers. See Section 3.29 for the conventions used in this table.

1. **`appendErrorX(double low, double high)`**

   ```
   myLayer.appendErrorX(1.2, 0.6)
   ```

   Example 3.85. Appends a low and high error value of x.

2. **`appendErrorX(Ordered1dData low, Ordered1dData high)`**

   ```
   lowErrors = Double1d([1.2, 1.4, 0.9])
   highErrors = Double1d([0.6, 1.1, 0.4])
   myLayer.appendErrorX(lowErrors, highErrors)
   ```

   Example 3.86. Appends a set of low and high error values of x.

3. **setErrorX(Ordered1dData [] error)**

```
errors = Double1d([1.2, 1.4, 0.9])
# Java style
myLayer.setErrorX(errors)
# Jython style
myLayer.errorX = errors
```

Example 3.87. Sets low and high error values of x.

4. **setErrorX(Ordered1dData low, Ordered1dData high)**

```
lowErrors = Double1d([1.2, 1.4, 0.9])
highErrors = Double1d([0.6, 1.1, 0.4])
myLayer.setErrorX(lowErrors, highErrors)
```

Example 3.88. Sets the low and high error values of x.

5. **getErrorX()**

```
# Java style
print myLayer.getErrorX()
# Jython style
print myLayer.errorX
```

Example 3.89. Returns an array of Ordered1dData with length equal to 2.

3.18. Switching to histogram mode

The following example shows how to change the *myLayer* layer to histogram mode. You need to be in MARKED or SOLID line style for this mode to work:

```
myLayer.line = Style.MARKED
myLayer.style.chartType = Style.HISTOGRAM
```

Three chart types are available:

- `HISTOGRAM` : the data point is in the middle of the histogram horizontal bar.
- `HISTOGRAM_EDGE` : the data point is on the edge of the histogram horizontal bar.
- `LINECHART` : the data points are connected with lines (default setting).

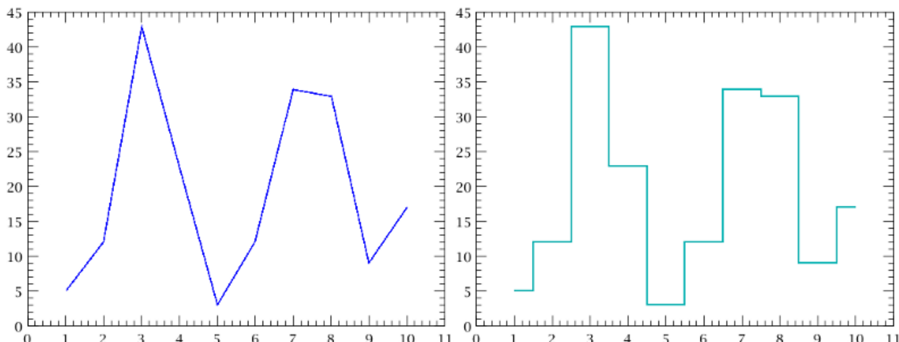


Figure 3.7. The same data set plotted as LINECHART (default) and HISTOGRAM.

To produce data sets for histogram plots you can use the `Histogram` and `BinCentres` functions of the HIPE Numeric library. For more information and examples see the corresponding entries in the *User's Reference Manual* :

- Section 1.184 in *HCSS User's Reference Manual*

- Section 1.37 in *HCSS User's Reference Manual*

3.19. Adding subplots

You can draw a smaller plot inside your main plot, as shown by the following example. The example also shows how to change the axis ranges of the subplot:

```
# Setting up dummy data
n = 10000
x = (Double1d.range(n)+1)/(n/100)
y = SIN(x)/x

# Creating main plot
myPlot = PlotXY()
# You have to follow this additional step for a plot to accept subplots.
myPlot.setLayout(PlotOverlayLayout())

myLayer0 = LayerXY(x,y)
myPlot.addLayer(myLayer0)

# Creating and adding subplot
# Here you define the area occupied by the subplot. The four numbers are
# the distances of the subplot boundaries from the top, left, bottom and right
# axes of the main plot, respectively. The units are the lengths of the main
# plot axes. In this example, the top boundary of the subplot is separated
# from the top axis of the main plot by 0.05 times the length of the
# vertical axis of the main plot. Run the example with different values
# to get a feel for the effects of these values.
mySubplot = SubPlot(SubPlotBoundsConstraints(0.05, 0.4, 0.5, 0.07))
myLayer1 = LayerXY(x, y)
# You add layers to a subplot in the same way as to a plot.
mySubplot.addLayer(myLayer1)
myPlot.addSubPlot(mySubplot)

# Modifying axis ranges
# You cannot access axes directly from the subplot. You have to get them
# through a layer. This is why this line and the following one use baseLayerXY
# to get the first layer in the subplot. Then you can set the range property to
# change the axis range.
mySubplot.baseLayerXY.xaxis.range = [0, 100]
mySubplot.baseLayerXY.yaxis.range = [-0.2, 0.2]
```

Example 3.90. How to add and customise a subplot inside a main plot.

The following figure shows the plot created by the previous example:

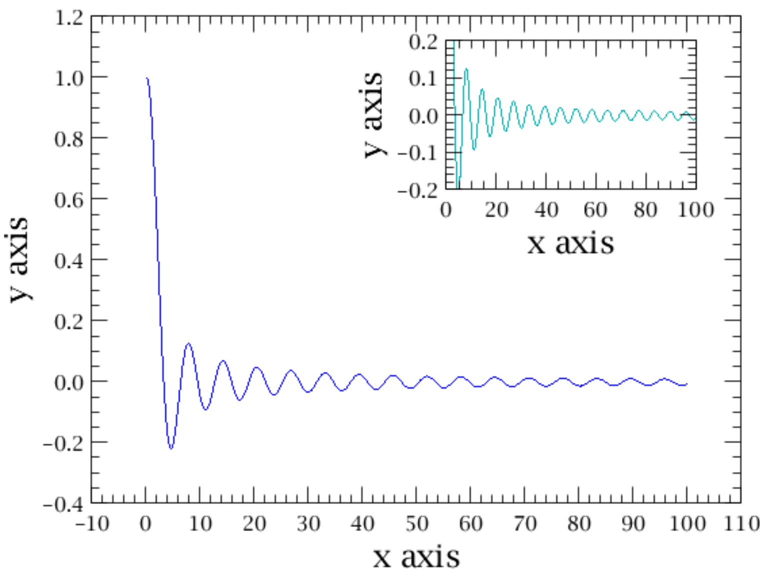


Figure 3.8. Plot with embedded subplot.

If you open the properties panel of the plot (see Section 3.6) you will see a *Subplot 1* entry, from which you can change the properties of the subplot.

Since subplots use layers just like main plots, all the methods described in Section 3.4 also apply to subplots. Subplots also accept all the methods for colours, fonts and visibility described in Section 3.26 .

3.20. Embedding monochromatic images in plots

There are several steps required in order to add an image to a plot. First, you should not use a `SimpleImage` but the `image` dataset (or extension) inside a `SimpleImage`. This is a two-dimensional array like a `Float2d`:

```
plotImage = myImage.image  # myImage is a SimpleImage
myLayer = LayerImage(myImage)  # Gives an error
myLayer = LayerImage(plotImage)  # Correct
```

As the WCS data stored in the `SimpleImage` object is not yet in the layer along with the dataset, you should now transfer it. Please note that if the source data do not use a cylindrical projection you should manually correct it (see the warning below):

```
wcs = myImage.wcs  # Extracting the WCS from the SimpleImage
crpix1 = wcs.crpix1
crpix2 = wcs.crpix2

crval1 = wcs.crval1
crval2 = wcs.crval2

cdelt1 = wcs.cdelt1
cdelt2 = wcs.cdelt2

naxis1 = wcs.naxis1
naxis2 = wcs.naxis2

cosd = Math.cos(Math.toRadians(crval2))

# Set the origin and the scale of the axes so that they coincide with the WCS.
myLayer.xcdelt = cdelt1/cosd
myLayer.ycdelt = cdelt2

myLayer.xcrpix = crpix1
myLayer.ycrpix = crpix2

myLayer.xcrval = crval1
myLayer.ycrval = crval2
```

Warning

`LayerImage` assumes a cylindrical projection, where moving up/down changes only the declination, and moving left/right changes only the right ascension. This is not the case in a real map which should have a gnomonic projection ("TAN"). For small images, the gnomonic projection is approximately cylindrical.

`LayerImage` assumes the image is small, and that a cylindrical approximation is valid. This causes differences in the world coordinates of pixels far from the reference pixel in large maps

The following example shows a workaround to ensure the projection is correct at the target position:

```
# Convert source coordinates (raNominal, decNominal) to pixel
# coordinates using the input map:
xNominal = inputMap.wcs.getPixelCoordinates(raNominal, decNominal)
[1]
```

```
yNominal = inputMap.wcs.getPixelCoordinates(raNominal, decNominal)
[0]

# Set up the LayerImage:
myPlot = PlotXY()
myPlot.addLayer(LayerImage(inputMap.image))

# Set the origin and the scale of the axes so they coincide with
# the WCS of the input image, amending the centre with the new
# reference coordinates
myPlot[0].xcdelt = inputMap.wcs.cdelt1/
COS(decNominal*Math.PI/180.)
myPlot[0].ycdelt = inputMap.wcs.cdelt2
myPlot[0].xcrpix = xNominal+1
myPlot[0].ycrpix = yNominal+1
myPlot[0].xcrval = raNominal
myPlot[0].ycrval = decNominal
```

The following example shows how to set the colour and intensity tables:

```
# Set the colour table
myLayer.colorTable = "Ramp"
# Print available values for the colour table
print myLayer.style.getColorTableNames()
# Set the intensity table
myLayer.intensityTable = "Negative"
# Print available values for the intensity table
print myLayer.style.getIntensityTableNames()
```

Finally, add the `LayerImage` object with `image` and `wcs` information to a plot:

```
myLayer = LayerImage(plotImage)
myPlot.addLayer(myLayer)
```

See also Section 3.30 for an example script in which an image from a Herschel observation is used to create a plot.

3.21. Embedding RGB images in plots

To add an RGB image to a plot, add a `LayerRgbImage` layer, like you would do with a normal `LayerXY` :

```
myLayer = LayerRgbImage(redImage, greenImage, blueImage)
myPlot.addLayer(myLayer)
```

Each of the three images you have to pass to a `LayerRgbImage` is not a `SimpleImage` but a two-dimensional Numeric array such as `Float2d` . You can extract the three components of an `Rgb-SimpleImage` in the correct format as follows:

```
red = myRgbImage.redByteImage
green = myRgbImage.greenByteImage
blue = myRgbImage.blueByteImage
```

The following example creates a plot with an RGB image obtained from public Herschel archive data.

The script connects to the Herschel Science Archive to retrieve data, so you must be connected to the Internet and logged in. For more information on logging in to the Herschel Science Archive, see Section 1.4.1 .

```
from java.awt import Color
from java.lang import Math
from herschel.ia.gui.plot import LayerRgbImage
```

```
# Get a public SPIRE observation
myobs = getObservation(obsid=1342183475, useHsa=True)

# Extract the PSW (250µm band) map
map = myobs.browseProduct

# Extract the image array and the WCS
red = map["red"].data
green = map["green"].data
blue = map["blue"].data

wcs = map.wcs

# Extract some information from the WCS
cdelt1 = wcs.cdelt1
cdelt2 = wcs.cdelt2
crpix1 = wcs.crpix1
crpix2 = wcs.crpix2
crval1 = wcs.crval1
crval2 = wcs.crval2
naxis1 = wcs.naxis1
naxis2 = wcs.naxis2
cosd = Math.cos(Math.toRadians(crval2))

# Create the layer with the image
layIma = LayerRgbImage(red, green, blue)

# Set the origin and the scale of the axes
# so that they coincide with the WCS.
# Note that you cannot rotate the image.
layIma.xcdelt = cdelt1/cosd
layIma.ycdelt = cdelt2

layIma.xcrpix = crpix1
layIma.ycrpix = crpix2

layIma.xcrval = crval1
layIma.ycrval = crval2

# Create a plot
myPlot = PlotXY()

# Add the image layer to the plot
myPlot.addLayer(layIma)

# Change the axis type to have ticks in degrees, min, sec
myPlot.xaxis.type = Axis.RIGHT_ASCENSION
myPlot.yaxis.type = Axis.DECLINATION
myPlot.xaxis.titleText = "Right Ascension (J2000)"
myPlot.yaxis.titleText = "Declination (J2000)"

# Set the axes ranges so that the image fills
# the plotting area
myPlot.xrange = [crval1-(crpix1-0.5)*cdelt1/cosd,\
       crval1-(crpix1-naxis1-0.5)*cdelt1/cosd]
myPlot.yrange = [crval2-(crpix2-0.5)*cdelt2,\
       crval2-(crpix2-naxis2-0.5)*cdelt2]

# Change the size of the plotting area so that image pixels
# are as on the sky
myPlot.setPlotSize(4.0,4.0*(naxis2*-cdelt2)/(naxis1*cdelt1))
```

Example 3.91. How to manually combine three bands to create an RGB image, respecting WCS information.

The following image shows the plot produced by the script:

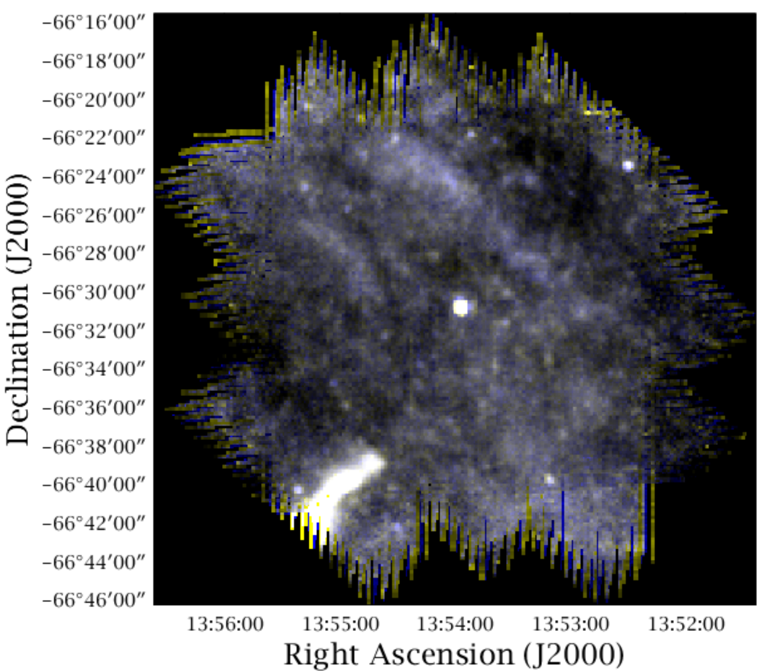


Figure 3.9. RGB image in a plot.

3.22. Inserting math and special symbols

You can use TeX-like formatting of strings in plots. In particular, entering math mode using a `$` symbol it is possible to insert Greek characters, for instance using `\\alpha` or `\\beta` . Superscripts are preceded by the ^ symbol and subscripts by the _ symbol. For example, the following two lines set the labels of the two axes:

```
myPlot.xaxis.title.text="$A_{1.3}^{b-3/2}$"
myPlot.yaxis.title.text="$\\alpha_{1.3}^{\\beta-3/2}$"
```

The following figure shows the resulting plot:

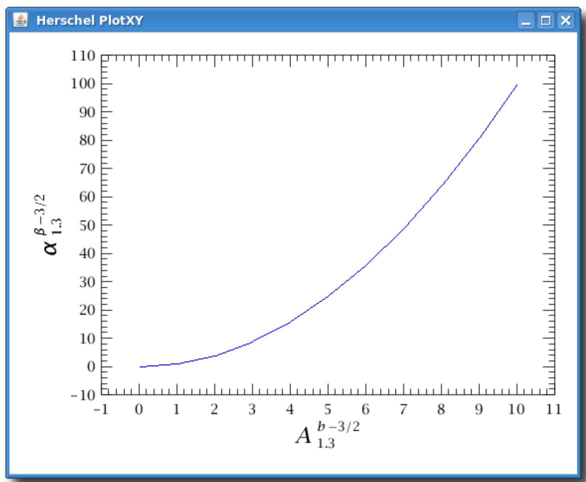


Figure 3.10. Using special characters for labels.

Note that it is necessary to use "\\" to escape the "\" symbol *from the command line* . A single backslash should be used in the *Property Panel* window instead.

Warning

Not all special symbols are available. If the symbol is not available it will be treated as normal text by the interpreter. For example, `$\\Alpha$` will be rendered as *\Alpha* .

The following special symbols are available:

- All the lower-case Greek letters.
- The following upper-case Greek letters: `\Gamma`, `\Delta`, `\Theta`, `\Lambda`, `\Xi`, `\Pi`, `\Sigma`, `\Upsilon`, `\Phi`, `\Psi`, `\Omega` .
- The `\angstrom` and `\micro` symbols.

To insert other symbols you can use the Unicode escape sequence `\uxxxx` , where `xxxx` is the hexadecimal code of the symbol. For example, `\u2299` corresponds to the *circle dot operator* , which can also be used as symbol for the Sun.

For a list of Unicode sequences see for example http://www.utf8-chartable.de/ .

3.23. Creating a plot in batch mode

HIPE tries to batch automatically plot statements to obtain the best performance. There is nothing to be configured or specified to take advantage of this. If you have a script like the one below, written for older versions of HIPE, you should remove the `.batch` assignments to avoid warning messages like this:

```
WARNING: herschel.ia.gui.plot.PlotXY.setBatch is deprecated. This method has no
 effect. Please remove it.
```

```
x = Int1d([1,2,3])
x1 = x2 = x3 = y = y1 = y2 = y3 = x
myPlot = PlotXY()
myPlot.batch = 1
myLayer1 = LayerXY(x,y)
myLayer2 = LayerXY(x1,y1)
myLayer3 = LayerXY(x2,y2)
myLayer4 = LayerXY(x3,y3)
myPlot.addLayer(myLayer1, 0, 0)
myPlot.addLayer(myLayer2, 0, 1)
myPlot.addLayer(myLayer3, 1, 0)
myPlot.addLayer(myLayer4, 1, 1)
myPlot.batch = 0
del (x, y, x1, y1, x2, y2, x3, y3)
```

Example 3.92. Batching several plots to improve speed.

3.24. Drawing multiple plots per window

When you add layers to a plot, you can specify their position on a grid. The following example places four layers onto a 2x2 grid (running indices from 0, 0 to 1, 1).

```
x = Int1d([1,2,3])
x1 = y = y1 = x
myPlot = PlotXY()
myLayer1 = LayerXY(x,y)
myLayer2 = LayerXY(x1,y1)
myLayer3 = LayerXY(x1,y1/5.0)
myLayer4 = LayerXY(x1/5.0,y1)
myPlot.addLayer(myLayer1, 0, 0) # top left
myPlot.addLayer(myLayer2, 0, 1) # bottom left
myPlot.addLayer(myLayer3, 1, 0) # top right
myPlot.addLayer(myLayer4, 1, 1) # bottom right
del (x, y, x1, y1)
```

Example 3.93. Distributing layers inside a main plot.

You can change the properties of layers via the *Property Panel* window or via the command line. See Section 3.4 for details.

See also Section 3.32 for an example of creating a plot with multiple panels.

3.25. Colours in plots

You can set colours for all the main plot components (layers, axes, titles and so on). You specify colours as objects of type `java.awt.Color` (note the US spelling of `Color`).

You can set a colour when you create a component or afterwards:

```
# At creation time
myLayer = LayerXY(x, y, color=java.awt.Color.orange)
# Setting on existing component
myLayer.color = java.awt.Color.red
```

The following thirteen predefined colours are available:

- `black`
- `blue`
- `cyan`
- `darkGray`
- `gray`
- `green`
- `lightGray`
- `magenta`
- `orange`
- `pink`
- `red`
- `white`
- `yellow`

You can get any other colour by specifying the red, green and blue values in ranges from 0 to 255:

```
myLayer.color = java.awt.Color(0,250,20)
```

You can add an `import` statement to avoid writing the `java.awt.` prefix every time:

```
# Writing the prefix
myLayer.color = java.awt.Color.red
# Importing the Color class
from java.awt import Color
# Prefix no longer needed
myLayer.color = Color.red
```

3.26. Methods for colours, fonts and visibility

The methods in the following table apply to all main components of a plot: the plot itself, layers, axes, titles, annotations and subplots. Ticks (`AxisTick` class) only implement `setVisible` , although tick labels (`AxisTickLabel` class) implement all these methods.

Procedure 3.12. Common methods to customise colours, fonts and visibility. See Section 3.29 for the conventions used in this table.

1. **`setVisible(boolean visible)`**

```
# Java style
myAxis.setVisible(True)
# Jython style
myAxis.visible = True
```

Example 3.94. Sets whether the component is visible.

2. **`setColor(Color color)`**

```
# Java style
myAnnotation.setColor(java.awt.Color.red)
# Jython style
myAnnotation.color = java.awt.Color.red
```

Example 3.95. Sets the foreground colour of the component.

3. **`setFont(Font font)`**

```
myFont = java.awt.Font("Arial", java.awt.Font.PLAIN, 15)
# Java style
myLabel.setFont(myFont)
# Jython style
myLabel.font = myFont
```

Example 3.96. Sets the font of the component. You can specify a font by giving a name, style and point size. Available font styles are PLAIN , BOLD and ITALIC . You can also use the numbers 0, 1 and 2, respectively.

4. **`setFontName(String name)`**

```
# Java style
myTitle.setFontName("Courier")
# Jython style
myTitle.fontName = "Courier"
```

Example 3.97. Sets the name of the font of the component.

5. **`setFontSize(float size)`**

```
# Java style
myAnnotation.setFontSize(12)
# Jython style
myAnnotation.fontSize = 12
```

Example 3.98. Sets the size of the font of the component.

6. **`setFontStyle(int style)`**

```
# Java style
myAnnotation.setFontStyle(java.awt.Font.BOLD)  # Name
myAnnotation.setFontStyle(1)  # Number
# Jython style
myAnnotation.fontStyle = java.awt.Font.BOLD  # Name
myAnnotation.fontStyle = 1
```

Example 3.99. Sets the style of the font of the component. Possible values are PLAIN , BOLD and ITALIC . You can also use the numbers 0, 1 and 2, respectively.

3.27. Invisible plots

You can create a plot without showing it on screen. This is useful if you want to show the plot at a later time, or if you just want to save it to file.

Create the plot with a single command, setting the `visible` attribute to zero:

```
# Create plot data
x = Double1d(range(100))/10.0
y = x*x
# Create the plot
myPlot = PlotXY(x, y, titleText = "Invisible plot", visible = 0)
```

Example 3.100. Creating a plot that is not drawn on the screen.

The plot window will briefly flash and then disappear. To make the plot visible use the following command:

```
myPlot.visible = 1
```

Note

If you create an *empty* invisible plot, the plot window may be shown even if the plot itself remains invisible. To make the window disappear, set the `visible` attribute to 1 and then back to 0.

3.28. Getting mouse coordinates on plots

You can obtain the coordinates of mouse clicks on your plots, in units of the x and y axes. Use the `getCoords` method to obtain the exact coordinates of mouse clicks:

```
coords = myPlot.getCoords(3)
```

The above example gets the coordinates of the next three clicks on plot `myPlot` and stores them in the *`coords`* variable. Note that you have to do the mouse clicks before you can issue other commands in the *Console* view of HIPE.

You can then store the x and y coordinates of the mouse clicks into two arrays:

```
xcoords = Double1d(coords[0])
ycoords = Double1d(coords[1])
```

The `getDataCoords` method gets the coordinates of the data point nearest to the mouse click, rather than the coordinates of the mouse click itself:

```
coords = myPlot.getDataCoords(3)
```

You can then obtain two arrays of x and y coordinates as with the `getCoords` method.

3.29. More on plot methods

Given the size and complexity of the plot package, not all the available commands are described in this chapter. For a complete list please refer to the related Javadoc documentation for the `herschel.ia.gui.plot` package.

The tables in this chapter follow these conventions:

- When a method with " `X` " in its name is listed, there is also a method with " `Y` ", doing the same thing for the Y axis, *unless specified otherwise* . For example, there is a `setYtitle` method in addition to `setXtitle` .
- Methods whose name begins with `set` are called *setters* and are used to set a value. For every setter there is usually a *getter* , a method whose name begins with `get` and whose job is to retrieve a value. The tables only list setters; for every setter it is implicit that a getter exists, *unless specified*

otherwise . A getter is called without input parameters and its return value is of the same type as the input parameter of the corresponding setter. For example, the `setXaxis(Axis axis)` setter has a corresponding `getXaxis()` getter returning an object of class `Axis` .

- When a setter method takes a boolean variable as argument (that is, a variable with only two possible values, `True` or `False`), the corresponding getter begins by `is` rather than `get` . For instance, `setVisible` and `isVisible` , not `getVisible` .
- Getter and setter Java methods have a simplified syntax in Jython. This syntax was used in all the examples in this chapter and is shown by the following example:

```
myLayer.setColor(java.awt.Color.RED)  # Java style for setter
myLayer.color = java.awt.Color.RED  # Simplified Jython style for setter
myColor = myLayer.getColor()  # Java style for getter
myColor = myLayer.color  # Simplified Jython style for getter
```

  Commands written in Java or Jython style have the same effect. Which style you choose is a matter of personal preference.

- The name of a method can offer useful clues about its behaviour. For example, the method `setSomething` will *replace* the preexisting Something, while `appendSomething` or `addSomething` will *add* SomethingElse to the existing Something.
- Methods with the same name as a class are called *constructors* and are used to create an object from that class. For example, using the `PlotXY()` constructor to create a *myPlot* object from the `PlotXY` class:

```
myPlot = PlotXY()
```

For more information on classes and methods see the *Scripting Guide* : Section 1.29 in *Scripting Guide*
.

The following image shows the classes you are most likely to use when designing plots on the command line.

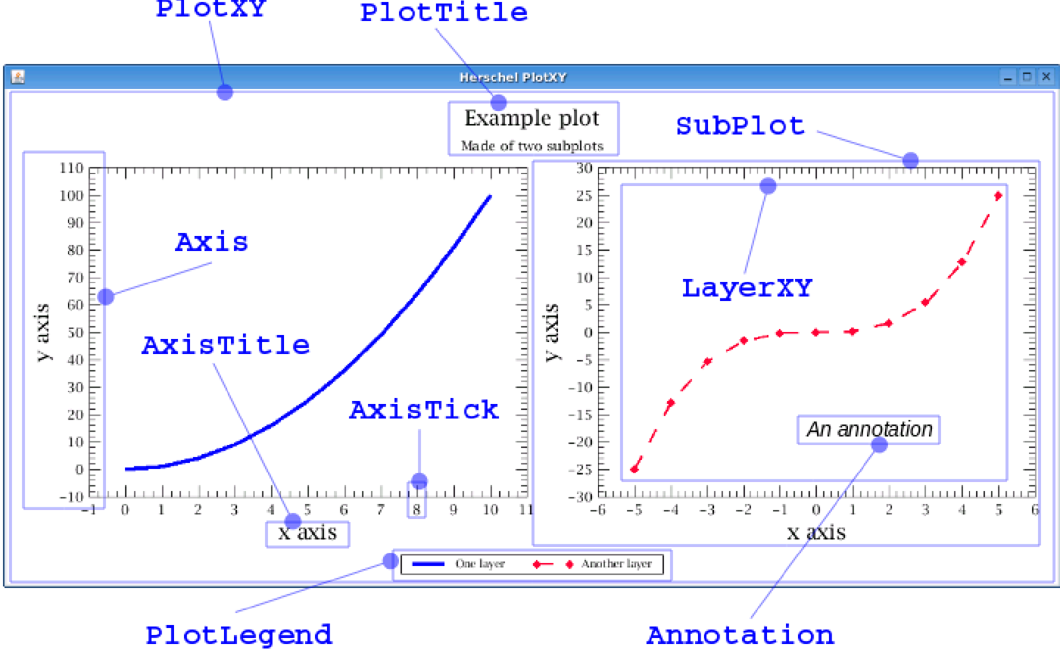
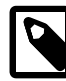


Figure 3.11. Classes involved in plot operations.

3.30. Worked example: Plot with an image

The following is a commented example of how to produce a plot of an image taken from a public Herschel observation. Many features of the plot package are illustrated, including advanced axes customisation, annotations, subplots and a technique to add contour levels.



Note

For a simplified script that makes use of the `Display` class from the herschel.ia.gui.image package, please see below this script.

The script connects to the Herschel Science Archive to retrieve data, so you must be connected to the Internet and logged in. For more information on logging in to the Herschel Science Archive, see Section 1.4.1 .

The following image shows the plot produced by the script:

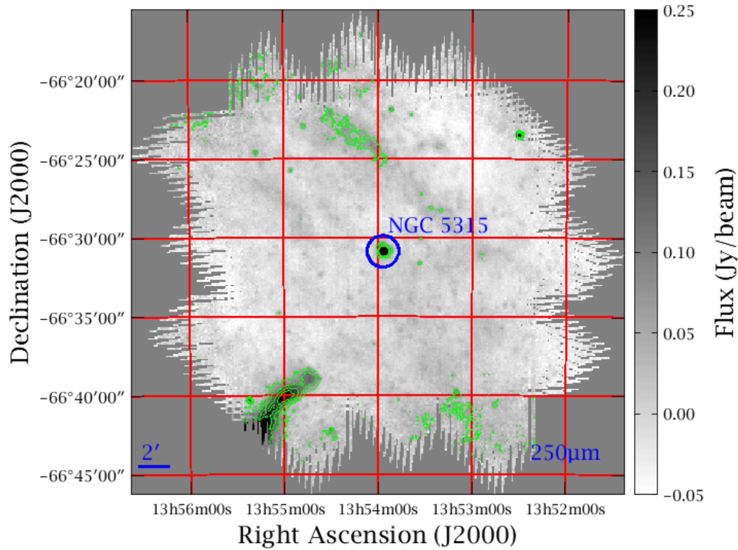


Figure 3.12. The result of the commented plot example presented in this section.

```
#
# Advanced example script to show how to display images in PlotXY.
#
# This script shows how to produce a "publication-ready" figure of a Herschel map.
# Run in HIPE 8 or newer.
#
# Author: Pasquale Panuzzo, CEA Saclay Irfu/SAp
#         pasquale.panuzzo@cea.fr
#
# Version: 9 June 2011
#

# Import some useful classes
from java.awt import Color
from java.lang import Math
from herschel.share.fltdyn.math import SexagesimalFormatter
from herschel.share.fltdyn.math.SexagesimalFormatter import Mode

# Get a public SPIRE observation
myobs=getObservation(1342183475L, useHsa=True)

# Extract the PSW (250 μm band) map
map=myobs.level2.refs["psrcPSW"].product

# Extract the image data
image=map.image

# Create the layer with the image
layIma=LayerImage(image)

# Create a PlotXY object
```

```
myPlot=PlotXY()

# Add the image layer to the plot
myPlot.addLayer(layIma)

#### Coordinates ####

# The image will be plotted in a reference system with origin in the lower-left
# corner of the image pixel [0,0] and with pixel size of 1x1.
#
# It is possible to change the position of the image respect to the PlotXY axes
# and change the pixel size, so that the PlotXY axes will show the
# Right Ascension and the Declination coordinates.
#
# The LayerImage provides methods to set the position of the image and the
# pixel size with a system similar to the FITS WCS.
#
# Please note:
# 1) PlotXY CANNOT rotate the image, so if you need to plot a map that
#    is not aligned with the North on RA and Dec axis, you will need to rotate it
#    before plotting it. In this script we assume that the map is aligned with
#    the North.
#
# 2) The PlotXY axis system assumes that coordinates are a linear
#    transformation of pixel coordinates. So coordinates on the plotted axes
#    are not fully correct in some projections. They are correct only for small
#    angles near the projection reference.

# Extract the WCS of the map and put some WCS info into variables
wcs=map.wcs

crpix1=wcs.crpix1
crpix2=wcs.crpix2

crval1=wcs.crval1
crval2=wcs.crval2

cdelt1=wcs.cdelt1
cdelt2=wcs.cdelt2

naxis1=wcs.naxis1
naxis2=wcs.naxis2

# cos(Dec)
cosd=Math.cos(Math.toRadians(crval2))

# Set the origin and the scale of the axes so that they coincide with the WCS.
myPlot[0].xcdelt=cdelt1/cosd # note the cos(Dec)!!!
myPlot[0].ycdelt=cdelt2

myPlot[0].xcrpix=crpix1
myPlot[0].ycrpix=crpix2

myPlot[0].xcrval=crval1
myPlot[0].ycrval=crval2

# Change the axis type so that we have ticks in degrees/hours, min, sec
# and the RA growing toward the left.
myPlot.xaxis.type=Axis.RIGHT_ASCENSION
myPlot.yaxis.type=Axis.DECLINATION
myPlot.xaxis.titleText="Right Ascension (J2000)"
myPlot.yaxis.titleText="Declination (J2000)"
# Adjust ticks to be nicer
myPlot.xaxis.tick.autoAdjustNumber=0
myPlot.xaxis.tick.number=5
myPlot.xaxis.tick.minorNumber=3
myPlot.xaxis.getAuxAxis(0).tick.autoAdjustNumber=0
myPlot.xaxis.getAuxAxis(0).tick.number=5
myPlot.xaxis.getAuxAxis(0).tick.minorNumber=3
myPlot.yaxis.tick.autoAdjustNumber=0
myPlot.yaxis.tick.number=5
myPlot.yaxis.tick.minorNumber=3
```

```
myPlot.yaxis.getAuxAxis(0).tick.autoAdjustNumber=0
myPlot.yaxis.getAuxAxis(0).tick.number=5
myPlot.yaxis.getAuxAxis(0).tick.minorNumber=3

# Set the axes ranges so that the image fills completely the plotting area
xrange=[crval1-(crpix1-0.5)*cdelt1/cosd,\
          crval1-(crpix1-naxis1-0.5)*cdelt1/cosd]
myPlot.xrange=xrange
yrange=[crval2-(crpix2-0.5)*cdelt2,\
          crval2-(crpix2-naxis2-0.5)*cdelt2]
myPlot.yrange=yrange

# Change the size of the plotting area so that proportions are as on the sky
myPlot.setPlotSize(4.0,4.0*(naxis2*-cdelt2)/(naxis1*cdelt1))

#### Colours and intensity manipulation ####

# Set the colour table to have a grey image and the intensity table to have
# sources in black and empty sky in white
myPlot[0].colorTable="Ramp"
myPlot[0].intensityTable="Negative"

# Set the intensity range
highCut=0.25
lowCut=-0.05
myPlot[0].highCut=highCut
myPlot[0].lowCut=lowCut

#### Coordinate grid ####

# We can draw a coordinate grid on the image. PlotXY doesn't provide a built-in
# way to generate a coordinate grid, so we need to compute in the script the
# positions of a number of meridians and parallels and draw them as LayerXY.

# Parallels every 5', meridians every 1'
deltaDec=5.0/60.0
deltaRa=1.0*15.0/60.0

# Compute the nearest parallel and meridian to the projection center
decCenter=(Math.round(crval2/deltaDec))*deltaDec
raCenter=(Math.round(crval1/deltaRa))*deltaRa

# Estimate how many parallels and meridians shall be drawn on each side
ndec=Integer(Math.round((yrange[1]-yrange[0])/deltaDec)).intValue()
ndec=1+ndec/2
nra=Integer(Math.round((xrange[0]-xrange[1])/deltaRa)).intValue()
nra=1+nra/2

# Draw parallels
dd=10
nn=(2*nra)*dd+1
for i in range(2*ndec+1):
    # Coordinates of parallels in the sky coordinates
    raPara=raCenter+(Float1d.range(nn)-nra*dd)*deltaRa/dd
    decPara=Float1d(nn)+decCenter+(i-ndec)*deltaDec
    # Coordinates of parallels in the pixels coordinates
    xpixPara=Float1d(nn)
    ypixPara=Float1d(nn)
    for j in range(nn):
        ypixPara[j],xpixPara[j]=wcs.getPixelCoordinates(raPara[j],decPara[j])
    pass
    # Coordinates of parallels in the plot axes coordinates
    xplotPara=(xpixPara-crpix1+1.0)*cdelt1/cosd+crval1
    yplotPara=(ypixPara-crpix2+1.0)*cdelt2+crval2
    layPara=LayerXY(xplotPara,yplotPara,color=Color.red,stroke=1)
    myPlot.addLayer(layPara)
pass

# Draw meridians
nn=(2*ndec)*dd+1
for i in range(2*nra+1):
    raMeri=Float1d(nn)+raCenter+(i-nra)*deltaRa
```

```
    decMeri=decCenter+(Float1d.range(nn)-ndec*dd)*deltaDec/dd
    xpixMeri=Float1d(nn)
    ypixMeri=Float1d(nn)
    for j in range(nn):
        ypixMeri[j],xpixMeri[j]=wcs.getPixelCoordinates(raMeri[j],decMeri[j])
    pass
    xplotMeri=(xpixMeri-crpix1+1.0)*cdelt1/cosd+crval1
    yplotMeri=(ypixMeri-crpix2+1.0)*cdelt2+crval2
    layMeri=LayerXY(xplotMeri,yplotMeri,color=Color.red,stroke=1)
    myPlot.addLayer(layMeri)
pass

# We can note now that meridians and parallels don't cross the axes exactly
# at the ticks positions. That's because axes are linear with respect to pixels,
# while sky coordinates are not (with the exception of some projections).
#
# We want now put ticks to coincide with meridians and parallels. To do this
# we need to compute the plot coordinates where meridians and parallels cross
# the plot axes; we will impose these positions as tick locations and we will
# set the correct labels.

# Setting up formatters for Right Ascension and Declination
format_ra = SexagesimalFormatter(Mode.RA_HMS_LOWER)
format_ra.decimals = 0
format_dec = SexagesimalFormatter(Mode.DEC_DMS_SYMBOL)
format_dec.decimals = 0

# Compute again the location of parallels and find where they cross the Y axes
nn=(2*nra)*dd+1
ycrossPara0=Double1d(2*ndec+1,Float.NaN)
ycrossPara1=Double1d(2*ndec+1,Float.NaN)
decCross=String1d(2*ndec+1)
for i in range(2*ndec+1):
    raPara=raCenter+(Float1d.range(nn)-nra*dd)*deltaRa/dd
    decPara=Float1d(nn)+decCenter+(i-ndec)*deltaDec
    decCross[i]=format_dec.formatDegrees(decCenter+(i-ndec)*deltaDec)
    xpixPara=Float1d(nn)
    ypixPara=Float1d(nn)
    for j in range(nn):
        ypixPara[j],xpixPara[j]=wcs.getPixelCoordinates(raPara[j],decPara[j])
    pass
    xplotPara=(xpixPara-crpix1+1.0)*cdelt1/cosd+crval1
    yplotPara=(ypixPara-crpix2+1.0)*cdelt2+crval2
    xplotPara0=xplotPara-xrange[0]
    xplotPara1=xplotPara-xrange[1]
    for j in range(nn-1):
        if xplotPara0[j]*xplotPara0[j+1] <= 0.0:
            ycrossPara0[i]=(yplotPara[j]*xplotPara0[j+1]-yplotPara[j
+1]*xplotPara0[j])/ \
                           (xplotPara0[j+1]-xplotPara0[j])
        if xplotPara1[j]*xplotPara1[j+1] <= 0.0:
            ycrossPara1[i]=(yplotPara[j]*xplotPara1[j+1]-yplotPara[j
+1]*xplotPara1[j])/ \
                           (xplotPara1[j+1]-xplotPara1[j])
pass


iii0=ycrossPara0.where(IS_FINITE(ycrossPara0))
iii1=ycrossPara0.where(IS_FINITE(ycrossPara1))
myPlot.yaxis.tick.setFixedValues(ycrossPara0[iii0])
myPlot.yaxis.tick.label.fixedStrings=decCross[iii0].toArray()
myPlot.yaxis.getAuxAxis(0).tick.setFixedValues(ycrossPara1[iii1])

# Remove minor ticks
myPlot.yaxis.tick.minorNumber=0
myPlot.yaxis.getAuxAxis(0).tick.minorNumber=0


# Compute the location of meridians and find where they cross the X axes
nn=(2*ndec)*dd+1
xcrossMeri0=Double1d(2*nra+1,Double.NaN)
xcrossMeri1=Double1d(2*nra+1,Double.NaN)
```

```
raCross=String1d(2*nra+1)
for i in range(2*nra+1):
    raMeri=Float1d(nn)+raCenter+(i-nra)*deltaRa
    decMeri=decCenter+(Float1d.range(nn)-ndec*dd)*deltaDec/dd
    raCross[i]=format_ra.formatDegrees(raCenter+(i-nra)*deltaRa)
    xpixMeri=Float1d(nn)
    ypixMeri=Float1d(nn)
    for j in range(nn):
        ypixMeri[j],xpixMeri[j]=wcs.getPixelCoordinates(raMeri[j],decMeri[j])
    pass
    xplotMeri=(xpixMeri-crpix1+1.0)*cdelt1/cosd+crval1
    yplotMeri=(ypixMeri-crpix2+1.0)*cdelt2+crval2
    yplotMeri0=(yplotMeri-yrange[0])
    yplotMeri1=(yplotMeri-yrange[1])
    for j in range(nn-1):
        if yplotMeri0[j]*yplotMeri0[j+1] <= 0.0:
            xcrossMeri0[i]=(xplotMeri[j]*yplotMeri0[j+1]-xplotMeri[j
+1]*yplotMeri0[j])/ \
                           (yplotMeri0[j+1]-yplotMeri0[j])
        if yplotMeri1[j]*yplotMeri1[j+1] <= 0.0:
            xcrossMeri1[i]=(xplotMeri[j]*yplotMeri1[j+1]-xplotMeri[j
+1]*yplotMeri1[j])/ \
                           (yplotMeri1[j+1]-yplotMeri1[j])
pass

iii0=xcrossMeri0.where(IS_FINITE(xcrossMeri0))
iii1=xcrossMeri0.where(IS_FINITE(xcrossMeri1))
myPlot.xaxis.tick.setFixedValues(xcrossMeri0[iii0])
myPlot.xaxis.tick.label.fixedStrings=raCross[iii0].toArray()
myPlot.xaxis.getAuxAxis(0).tick.setFixedValues(xcrossMeri1[iii1])
myPlot.xaxis.tick.minorNumber=0
myPlot.xaxis.getAuxAxis(0).tick.minorNumber=0

#### Contours #####

# We want to draw level contours. We don't have (yet) a specialized layer for
# contours, so we will need to plot each contour segment.

# Generate the contours
contours = automaticContour(image=map,levels=4,min=0.05,max=0.2,distribution=0)

# Plot the contours as
keys=contours.keySet()
for key in keys:
   if key.startswith("Contour"):
      cont=contours[key]
      keysc=cont.keySet()
      for keyc in keysc:
          x=(cont[keyc].data[:,1]-crpix1+1.0)*cdelt1/cosd+crval1
          y=(cont[keyc].data[:,0]-crpix2+1.0)*cdelt2+crval2
          myPlot.addLayer(LayerXY(x,y,color=Color.green))
pass

#### Annotations #####

# Let's plot a circle around the observed source. This will be done with a
# classical LayerXY

# Radius of the circle (60")
radius=60.0/3600.
phase=Float1d.range(101)*2.0*Math.PI/100.0

# Position of the source as entered in HSPOT
raNom=map.meta["raNominal"].value
decNom=map.meta["decNominal"].value

# Coordinates of the circle
xx=radius*COS(phase)/cosd+raNom
yy=radius*SIN(phase)+decNom

# Add the circle
myPlot.addLayer(LayerXY(xx,yy,color=Color.blue,stroke=2))
```

```
# And let's put an annotation with the name of the observed source
ann=Annotation(raNom-0.015,decNom+0.015,map.meta["object"].value.upper())
ann.fontSize=12
ann.color=Color.blue
myPlot.addAnnotation(ann)

# We want also to put a line to show the angle scale
xx2=Float1d([0,-120./3600/cosd])+myPlot.xrange[1]-0.02
yy2=Float1d(2)+myPlot.yrange[0]+0.03
myPlot.addLayer(LayerXY(xx2,yy2,color=Color.blue,stroke=2))
ann=Annotation(myPlot.xrange[1]-0.027,myPlot.yrange[0]+0.032,"2"+unichr(0x2032))
ann.fontSize=12
ann.color=Color.blue
myPlot.addAnnotation(ann)

# Finally add the wavelength
annText="%3.0f"%map.meta["wavelength"].value
annText=annText+map.meta["wavelength"].unit.dialogName
ann=Annotation(myPlot.xrange[0]+0.25,myPlot.yrange[0]+0.032,annText)
ann.fontSize=12
ann.color=Color.blue
myPlot.addAnnotation(ann)


#### Colour bar ####

# We now want to create a colour bar on the right side of the plot.
# The colour bar is just another image in a subplot.

# Create an overlay layout so that we can plot the colour bar
layout=PlotOverlayLayout(marginRight=1.0)
myPlot.setLayout(layout)

# The colour bar is just a subplot showing an image,
# create the SubPlot where we put the colour bar
# the numbers here define the position of the bar respect to the main plot
spBar = SubPlot(SubPlotBoundsConstraints(0.0, 1.02, 0.0, -0.07))

# Here we construct the colour bar image data
lenBar=2561
barIma=Float2d(lenBar,1)
barIma[:,0]=(Float1d.range(lenBar)*(highCut-lowCut)/lenBar)+lowCut
# Construct a LayerImage with it
layBar = LayerImage(barIma)
layBar.colorTable=myPlot[0].colorTable
layBar.intensityTable=myPlot[0].intensityTable
layBar.highCut=highCut
layBar.lowCut=lowCut
layBar.ycdelt=(highCut-lowCut)/lenBar
layBar.ycrval=lowCut
layBar.ycrpix=+0.5
layBar.yrange=[lowCut,highCut]
layBar.xrange=[0,1]
# Add the layer to the subplot
spBar.addLayer(layBar)

# Put the colour bar on the plot
myPlot.addSubPlot(spBar)

# Adjust the axes characteristics
# The X axis
xaxis=spBar.baseLayer.xaxis
xaxis.titleText=""
xaxis.tick.label.visible=0
xaxis.tick.autoAdjustNumber=0
xaxis.tick.minorNumber=0
xaxis.tick.height=0.0
xaxis.getAuxAxis(0).tick.height=0.0
# The Y axis
yaxis=spBar.baseLayer.yaxis
yaxis.titleText=""
```

```
yaxis.tick.label.visible=0
yaxis.tick.autoAdjustNumber=0
yaxis.tick.number=5
yaxis.tick.minorNumber=0
yaxis.tick.height=0.04
yaxis.getAuxAxis(0).tick.label.visible=1
yaxis.getAuxAxis(0).titleText="Flux (Jy/beam)"
yaxis.getAuxAxis(0).title.visible=1
yaxis.getAuxAxis(0).tick.height=0.04
yaxis.getAuxAxis(0).tick.number=5
yaxis.getAuxAxis(0).tick.minorNumber=0

##### Save ####

# Finally, save the figure to PDF
myPlot.saveAsPDF(map.meta["object"].value.upper()+".pdf")

# End of the example.
```

Example 3.101. Complete example that demonstrates the use of the PlotXY class.

The previous script is intended as an exhaustive tutorial for the `PlotXY` class. The utility class `Display` has many features built-in, so the resulting scripts are much shorter. This is the same image, with the following minor differences:

- The grid cannot be aligned to whole minutes.
- The labels of the y axis cannot be formatted to sexagesimal syntax.

This is due to limitations in the `ImageAxis` component of the `Display` class. You can find the corresponding script below the image.

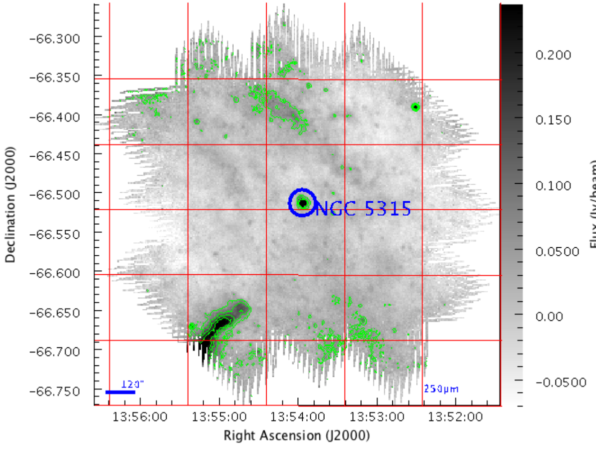


Figure 3.13. The result of the plot example created with the simplified version of the script.

```
from java.awt import Color

# Get a public SPIRE observation
myobs = getObservation(1342183475L, useHsa=True)

# Extract the PSW (250 um band) imgmap
imgmap = myobs.level2.refs["psrcPSW"].product

# Extract the WCS of the imgmap
wcs = imgmap.wcs

# Display the imgmap
d = Display(imgmap)

# Set to grayscale and reverse intensity
d.setColortable("Ramp", "Negative")
# Set cut levels
d.setCutLevelsMin(0.05)
d.setCutLevelsMax(0.25)
```

```
d.setCutLevelsPercentage(99.5)

# Dimensions of the image in pixels
imgWidth = d.getImage().getWidth()
imgHeight = d.getImage().getHeight()

# Generate the contours
contours = automaticContour(image = imgmap,levels = 4, min = 0.05, max = 0.2,
 distribution = 0)

# Plot the contours as
keys = contours.keySet()
for key in keys:
    if key.startswith("Contour"):
        cont = contours[key]
        keysc = cont.keySet()
        for keyc in keysc:
           d.addContour(contours[key][keyc], Color.green)
pass

# Draw the parallels every 5'
originWcs = wcs.getWorldCoordinates(0, 0)
# Calculate delta on the right edge of the image (y-axis)
rightBottomWcs = wcs.getWorldCoordinates(0, imgWidth)
dispRightEdge = rightBottomWcs[1]-originWcs[1]
# Calculate delta on the top edge of the image (x-axis)
topLeftWcs = wcs.getWorldCoordinates(imgHeight, 0)
dispTopEdge = originWcs[0]-topLeftWcs[0]
xDelta = wcs.cdelt1
pixelDispX = dispTopEdge/10*xDelta
pixelDispOne = 1/10./xDelta
maxLinesX = int(-imgWidth/pixelDispOne)

# Draw the meridians from right to left
for col in range(maxLinesX+1):
    d.addLine(0, (imgWidth-1)+col*pixelDispOne, imgHeight,\
   (imgWidth-1)+pixelDispX+col*pixelDispOne, 1.0, Color.red)

# Draw the parallels from bottom up
yDelta = wcs.cdelt2
pixelDispY = dispRightEdge/yDelta
pixelDispFive = 1/12./yDelta
maxLinesY = int(imgHeight/pixelDispFive)

for row in range(maxLinesY):
    d.addLine(0+row*pixelDispFive, 0, pixelDispY+row*pixelDispFive,\
    imgWidth, 1., Color.red)

# Add annotations (on top of the grid)
# Radius of the annotation circle (10 pixels)
radius = 10.

# Position of the source as entered in HSPOT
raNom = imgmap.meta["raNominal"].value
decNom = imgmap.meta["decNominal"].value

# Translate to row/column format
center = d.getPixelCoordinates(raNom, decNom)

# Draw the circle
d.addEllipse(center[0], center[1], radius*2.0, radius*2.0, 4.0,\
java.awt.Color(0, 0, 255))

# Put an annotation with the name of the observed source
d.setAnnotationFontColor(Color.blue)
d.addAnnotation(imgmap.meta["object"].value, center[0]-10., center[1]+10.)
d.setAnnotationFont(center[0]-10., center[1]+10., 24)

# Annotate the angle scale with a line below to illustrate the scale
d.addArcSecs(120, 10, 10, 4, Color.blue)

# Annotate the wavelength
```

```
annText = "%3.0f"%imgmap.meta["wavelength"].value
annText = annText+imgmap.meta["wavelength"].unit.dialogName
d.addAnnotation(annText, 10, imgWidth-60)
d.setAnnotationFont(10, imgWidth-60, 12)

# Get the axes
leftAxis = d.getLeftaxis()
bottomAxis = d.getBottomaxis()
rightAxis = d.getRightaxis()
# Add X axis
bottomAxis.enable()
bottomAxis.setWorldCoordinates(True)
bottomAxis.setLabel("Right Ascension (J2000)")
# Add Y axis
leftAxis.enable()
leftAxis.setWorldCoordinates(True)
leftAxis.setDecimalDegrees(True)
leftAxis.setLabel("Declination (J2000)")
# Add colour table
rightAxis.showColorTable(True)
rightAxis.setLabel("Flux (Jy/beam)")
```

Example 3.102. Version of the example above using the Display class.

3.31. Worked example: Initial plot of this chapter

The following script reproduces the plot in Section 3.1 .

```
x = Double1d.range(11) # Creates array with values from 0.0 to 10.0
y = x*x
x1 = x
y1 = ABS(SIN(x1))*100
x2 = 10.0*Double1d.range(11)/10.0 - 5.0
y2 = x2**3.0
x2err = SQRT(ABS(x2))
y2err = SQRT(ABS(y2))
myPlot = PlotXY()
myPlot.autoBoxAxes = 1
# Create the dataset
myLayer = LayerXY(x, y)
myLayer.name = "One layer"
myLayer1 = LayerXY(x1, y1)
myLayer1.name = "Another layer"
myLayer2 = LayerXY(x2, y2)
myLayer2.name = "Third one"
myPlot.addLayer(myLayer, 0, 0) # top left
myPlot.addLayer(myLayer1, 0, 0) # top right
myPlot.addLayer(myLayer2, 1, 0) # top right
# Title and subtitle
myPlot.titleText = "Custom plot title"
myPlot.title.font = java.awt.Font("Arial", java.awt.Font.PLAIN, 15)
myPlot.subtitleText = "Custom plot subtitle"
myPlot.subtitle.font = java.awt.Font("Courier", java.awt.Font.ITALIC, 15)
# Legend
myPlot.legend.visible = 1
# Ticks
myPlot.xaxis.tick.interval = 3.0
myPlot.yaxis.tick.interval = 30.0
myPlot.xaxis.tick.minorNumber = 10
myPlot.yaxis.tick.minorNumber = 5
# Error bars
myLayer2.errorX = [x2err, x2err/2]  # Setting the upper and lower error limits
myLayer2.errorY = [y2err, y2err*2]
# Axes
myPlot.xaxis.titleText = "Custom axis title"
myPlot[2].yaxis.title.text = "$\\alpha + \\beta^{-3/2}$"
# Symbols and line styles
myPlot.line = Style.DASHED
```

```
myPlot.style.stroke = 3
myPlot[1].line = Style.MARKED
myPlot[1].symbol = Style.FDIAMOND
myPlot[1].symbolSize = 10
myPlot[2].color = java.awt.Color.RED
myPlot[2].line = Style.NONE
myPlot[2].symbol = Style.FCIRCLE
# Grid lines
myPlot[2].yaxis.tick.gridLines = 1
# Annotations
myPlot[2].addAnnotation(Annotation(-5,80,"Custom
 annotation",color=java.awt.Color.BLUE))
myPlot[2].getAnnotation(0).fontSize = 12
myPlot[2].getAnnotation(0).angle = 30
# Auxiliary axis
xaux = myPlot[2].xaxis.getAuxAxis(0)
xaux.setTickIdentical(False)
xaux.tick.autoAdjustNumber = 0
xaux.title.visible = 1
xaux.titleText = "Auxiliary axis"
xauxlab = xaux.tick.label
xauxlab.visible = 1
vals = Double1d([-5.0, -2.0, 1.0, 3.0, 5.0]) # These are the wavenumbers we want to
 show
xaux.tick.setFixedValues(vals)
# String values to each label
svals = ["Minus five","Minus two","One","Three","Five"]
xauxlab.fixedStrings = svals
xaux.tick.height = 0.3
```

Example 3.103. Plotting the figure that appears at the beginning of this chapter.

3.32. Worked example: Multi-panel plot

The following script creates a plot made of six panels, as shown in the following image:

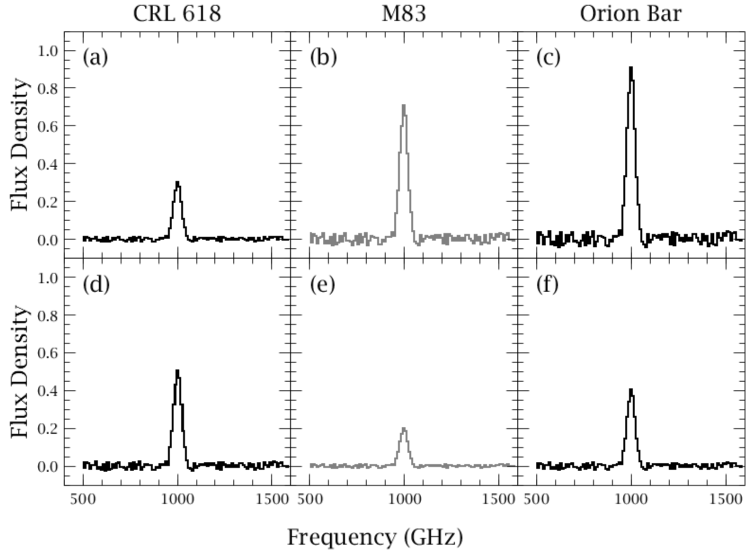


Figure 3.14. A plot with four panels.

```
from java.awt import Color
# A plot with different panels

# Set up some dummy data to plot
freq = Double1d(range(500, 1600, 10))
line = EXP(-(freq-1000)**2/30.0**2) + RandomUniform(0.1)(freq)-0.05
lineStrength = [0.5, 0.2, 0.4, 0.3, 0.7, 0.9]

# Initialise the plot
p = PlotXY()
```

```
# Specify the plot dimensions on the screen
p.plotSize=(2,2)
# Set the positions for the subplots in a grid layout
topLeft     = SubPlot(SubPlotGridConstraints(0,0))
bottomLeft  = SubPlot(SubPlotGridConstraints(0,1))
topMid      = SubPlot(SubPlotGridConstraints(1,0))
bottomMid   = SubPlot(SubPlotGridConstraints(1,1))
topRight    = SubPlot(SubPlotGridConstraints(2,0))
bottomRight = SubPlot(SubPlotGridConstraints(2,1))
# Fill the sub-plots
bottomLeft.addLayer(LayerXY(freq, lineStrength[0]*line, color=Color.BLACK, \
   stroke=1.5, chartType=Style.HISTOGRAM))
bottomMid.addLayer(LayerXY(freq, lineStrength[1]*line, color=Color.GRAY, \
   stroke=1, chartType=Style.HISTOGRAM))
bottomRight.addLayer(LayerXY(freq, lineStrength[2]*line, color=Color.BLACK, \
   stroke=1, chartType=Style.HISTOGRAM))
topLeft.addLayer(LayerXY(freq, lineStrength[3]*line, color=Color.BLACK, \
   stroke=1, chartType=Style.HISTOGRAM))
topMid.addLayer(LayerXY(freq, lineStrength[4]*line, color=Color.GRAY, \
   stroke=1, chartType=Style.HISTOGRAM))
topRight.addLayer(LayerXY(freq, lineStrength[5]*line, color=Color.BLACK, \
   stroke=1, chartType=Style.HISTOGRAM))
#
# Set the tick label and ranges for each subplot, and add it to the main plot
for subP in [topLeft, topRight, bottomLeft, bottomRight, topMid, bottomMid]:
   # Remove all axes labels to start with (add needed ones later)
   subP.baseLayerXY.xaxis.tick.labelVisible = 0
   subP.baseLayerXY.xaxis.title.visible = 0
   subP.baseLayerXY.yaxis.tick.labelVisible = 0
   subP.baseLayerXY.yaxis.title.visible = 0
   # Set axis ranges to be the same for all sub-plots
   subP.baseLayerXY.xaxis.range = [400.0, 1600.0]
   subP.baseLayerXY.yaxis.range = [-0.1, 1.1]
   # Set the ticks for the xaxis to be at nice intervals
   subP.baseLayerXY.xaxis.tick.setFixedValues(Double1d([500, 1000, 1500]), \
      Double1d(range(400, 1600, 100)))
   p.addSubPlot(subP)
# Set the gap between the sub-plots to be zero (i.e. plots touching each other)
p.gridLayout.setGap(0, 0)

# Set the titles for the left hand axes of left hand plots
# and make the tick labels visible on these axes
topLeft.baseLayerXY.yaxis.title.visible = 1
topLeft.baseLayerXY.yaxis.titleText = "Flux Density"
topLeft.baseLayerXY.yaxis.tick.labelVisible = 1
bottomLeft.baseLayerXY.yaxis.title.visible = 1
bottomLeft.baseLayerXY.yaxis.titleText = "Flux Density"
bottomLeft.baseLayerXY.yaxis.tick.labelVisible = 1
# Make the tick labels visible on the lower xaxis
bottomLeft.baseLayerXY.xaxis.tick.labelVisible = 1
bottomMid.baseLayerXY.xaxis.tick.labelVisible = 1
bottomRight.baseLayerXY.xaxis.tick.labelVisible = 1
# Use the upper xaxis for a title for each column
topLeft.baseLayerXY.xaxis.getAuxAxis(0).titleText = "CRL 618"
topLeft.baseLayerXY.xaxis.getAuxAxis(0).title.visible = 1
topMid.baseLayerXY.xaxis.getAuxAxis(0).titleText = "M83"
topMid.baseLayerXY.xaxis.getAuxAxis(0).title.visible = 1
topRight.baseLayerXY.xaxis.getAuxAxis(0).titleText = "Orion Bar"
topRight.baseLayerXY.xaxis.getAuxAxis(0).title.visible = 1
# Set some annotations
topLeft.baseLayerXY.setAnnotation(1, Annotation(500, 0.9, "(a)", fontSize=14))
topMid.baseLayerXY.setAnnotation(1, Annotation(500, 0.9, "(b)", fontSize=14))
topRight.baseLayerXY.setAnnotation(1, Annotation(500, 0.9, "(c)", fontSize=14))
bottomLeft.baseLayerXY.setAnnotation(1, Annotation(500, 0.9, "(d)", fontSize=14))
bottomMid.baseLayerXY.setAnnotation(1, Annotation(500, 0.9, "(e)", fontSize=14))
bottomRight.baseLayerXY.setAnnotation(1, Annotation(500, 0.9, "(f)", fontSize=14))
# Use the plot subtitle to contain a single label for all 3 x-axes
p.subtitle.text = "Frequency (GHz)"
p.subtitle.fontSize = p.title.fontSize
from herschel.ia.gui.plot import PlotTitle
```

```
p.subtitle.position = PlotTitle.BOTTOMCENTER
```

Example 3.104. Distributing multiple plots using panels.

3.33. Worked example: Error bars

The example in this section creates a plot with horizontal and vertical error bars. Figure 3 in J. A. Rodón et al., A&A 518, L80 (2010) .

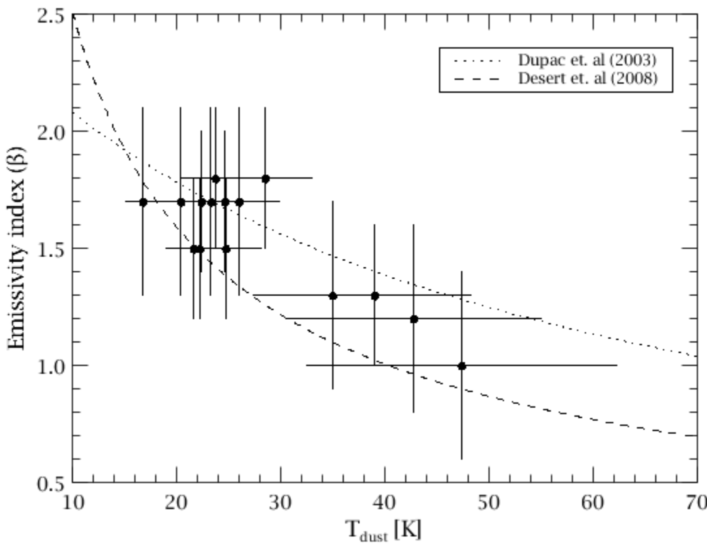


Figure 3.15. A plot with horizontal and vertical error bars.

```
from java.awt.geom import Point2D
from java.awt import Color
from herschel.ia.gui.plot import PlotLegend
from herschel.ia.gui.plot.renderer.axtype import AxisType

linex=(Double1d.range(601)/10).add(10)
lineAy=1/(0.4+8e-3*linex)
lineBy=11.5*pow(linex,-0.66)

# Dashed line A
dal=LayerXY(linex,lineAy)
dal.style=Style(line=3, color=Color.BLACK, dashArray=[1,3])
dal.name="Dupac et. al (2003)"
dal.xaxis=Axis(range=[10,70], titleText="$\\mathrm{T_{dust}}$ [K]")
dal.xaxis.setAxisType(AxisType.LINEAR)
dal.xaxis.title.fontSize=10
dal.yaxis=Axis(range=[0.5, 2.5], titleText="Emissivity index ($\\mathrm{\\beta}$)")
dal.yaxis.setAxisType(AxisType.LINEAR)
dal.yaxis.title.fontSize=10
dal.xaxis.tick.interval=10
dal.yaxis.tick.interval=0.5

# Dashed line B
dbl=LayerXY(linex,lineBy)
dbl.style=Style(line=3, color=Color.BLACK, dashArray=[4,4])
dbl.name="Desert et. al (2008)"

# Dots
x=Double1d([24.7200,42.7600,39.0100,47.3600,34.9700,16.7400,20.3900,22.3700, \
  23.3100,21.6100,22.2100,23.7200,24.6200,25.9800,28.5000])
xe=Double1d([3.40000,12.2300,9.27000,14.8800,7.61000,1.67000,2.30000,2.75000, \
  3.07000,2.58000,2.70000,3.16000,3.39000,3.85000,4.55000])
y=Double1d([1.50000,1.20000,1.30000,1.00000,1.30000,1.70000,1.70000,1.70000, \
  1.70000,1.50000,1.50000,1.80000,1.70000,1.70000,1.80000])
ye=Double1d([0.300000,0.400000,0.300000,0.400000,0.400000,0.400000,0.400000, \
```

```
  0.300000,0.400000,0.300000,0.300000,0.300000,0.300000,0.400000,0.300000])
l=LayerXY(x,y)
l.errorX=[xe,xe]
l.errorY=[ye,ye]
l.style=Style(line=0, color=Color.BLACK, symbolShape=SymbolShape.FCIRCLE,
 symbolSize=4)
l.inLegend=0

p=PlotXY()
p.addLayer(dal)
p.addLayer(dbl)
p.addLayer(l)

p.legend.visible=1
p.legend.columns=1
p.legend.position=PlotLegend.CUSTOMIZED
p.legend.setLocation(3.0,3.1)
p.legend.halign=PlotLegend.LEFT
p.legend.valign=PlotLegend.BOTTOM
```

Example 3.105. Plotting horizontal and vertical error bars.

3.34. Worked example: Auxiliary axes

The example in this section creates a plot with three layers and customised auxiliary axes.

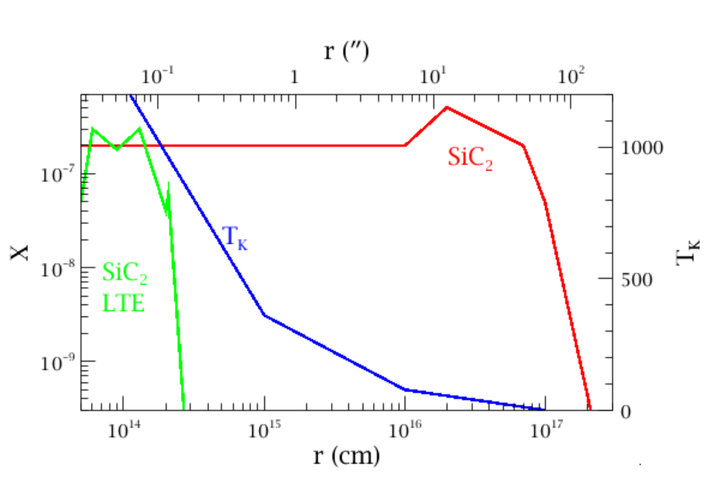


Figure 3.16. A plot with three layers and customised auxiliary axes.

```
from java.awt.geom import Point2D
from java.awt import Color
from herschel.ia.gui.plot import PlotLegend
from herschel.ia.gui.plot.renderer.axtype import AxisType

# SiC2 and X_bottom and Y left axis
xb=LayerXY(Double1d(0),Double1d(0))
xb.xaxis=Axis(range=[5e13,3e17], titleText="r (cm)")
xb.xaxis.setAxisType(AxisType.LOG)
xb.xaxis.title.fontSize=12
xb.xaxis.tick.label.format="%.0m"
xb.yaxis=Axis(range=[3e-10,7e-7], titleText="X")
xb.yaxis.setAxisType(AxisType.LOG)
xb.yaxis.title.fontSize=12
xb.yaxis.tick.label.format="%.0m"

# SiC2 values and annotation
sicx=Double1d([5e13, 1e16, 2e16, 7e16, 1e17, 2.1e17])
sicy=Double1d([2e-7, 2e-7, 5e-7, 2e-7, 5e-8, 3e-10])
sic=LayerXY(sicx,sicy)
sic.style=Style(line=1, stroke = 1.5, color=Color.RED)
sic.addAnnotation(Annotation(2e16, 1e-7, "SiC$\\mathrm{_2}$", fontSize=12,
 color=Color.RED))
```

```
sic.yaxis=xb.yaxis

# Plot X_bottom and Y left axis
p=PlotXY()
p.autoBoxAxes=0
p.setPlotSize(3.7,2.2)
p.addLayer(xb)

# Plot SiC2
p.addLayer(sic)

# SiC2 LTE values
ltex=Double1d([5e13, 6e13, 9e13, 1.3e14, 2e14, 2.1e14, 2.7e14])
ltey=Double1d([5e-8, 3e-7, 1.8e-7, 3e-7, 4e-8, 6e-8, 3e-10])
ltel=LayerXY(ltex,ltey)
ltel.style=Style(line=1, stroke = 1.5, color=Color.GREEN)
ltel.addAnnotation(Annotation(7e13, 3e-9, "SiC$\\mathrm{_2}$\nLTE", fontSize=12,
 color=Color.GREEN))
ltel.yaxis=xb.yaxis

# Plot SiC2 LTE
p.addLayer(ltel)

# Tk and X_bottom and Y right axis
yr=LayerXY(Double1d(0),Double1d(0))
yr.yaxis=Axis(range=[0,1200], position=Axis.RIGHT, titleText="$\\mathrm{T_K}$")
yr.yaxis.setAxisType(AxisType.LINEAR)
yr.yaxis.title.fontSize=12
yr.yaxis.tick.label.format="%.0f"
yr.yaxis.tick.interval=500
yr.yaxis.tick.minorNumber=4

# Plot Y right axis
p.addLayer(yr)

# Tk values
tkx=Double1d([1.1e14, 1e15, 1e16, 1e17])
tky=Double1d([1200, 360, 80, 0])
tkl=LayerXY(tkx,tky)
tkl.style=Style(line=1, stroke = 1.5, color=Color.BLUE)
tkl.addAnnotation(Annotation(5e14, 600, "T$\\mathrm{_K}$", fontSize=12,
 color=Color.BLUE))
tkl.yaxis=yr.yaxis

# Plot TK
p.addLayer(tkl)

# X_top axis
xt=LayerXY(Double1d(0),Double1d(0))
xt.xaxis=Axis(range=[2.8e-2,2E2], titleText="r (\u2033)")
xt.xaxis.setAxisType(AxisType.LOG)
xt.xaxis.title.fontSize=12
xt.xaxis.tick.label.format="%.0m"

# Plot X_top axis
p.addLayer(xt)

p.saveAsPDF("test-1.pdf")
p.saveAsEPS("test-1.eps")
p.saveAsPNG("test-1.png")
p.saveAsJPG("test-1.jpg")
# You can do all this from the property panel menu.
```

Example 3.106. Adding multiple layers and customised auxiliary axes to a plot.

3.35. Worked example: Histograms

The example in this section creates a plot with three panels, each containing superimposed histograms. Figure 1 in L. Shao et al., A&A 518, L26 (2010) .

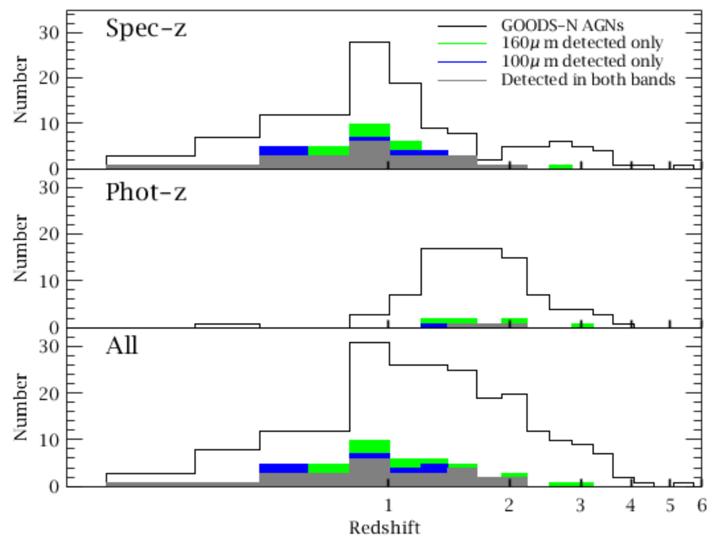


Figure 3.17. A plot with three panels, each containing superimposed histograms.

```
from java.awt import Color
from herschel.ia.gui.plot.renderer.axtype import AxisType

p=PlotXY()
p.plotSize=(4,1)
p.gridLayout.vgap=0

z_grid=Double1d([0.00000,0.100000,0.200000,0.330000,0.480000,0.630000, \
  0.800000,1.00000,1.20000,1.40000,1.65000,1.90000,2.20000,2.50000,2.85000, \
  3.20000,3.60000,4.05000,4.55000,5.10000,5.70000])
p1_all= Double1d([0,0,3,7,12,12,28,19,9,8,2,5,5,6,5,4,1,1,0,1,0])
p1_both=Double1d([0,0,1,1, 3, 3, 6, 3,3,3,1,1,0,0,0,0,0,0,0,0,0])
p1_100 =Double1d([0,0,1,1, 5, 3, 7, 4,4,3,1,1,0,0,0,0,0,0,0,0,0])
p1_160 =Double1d([0,0,1,1, 5, 5,10, 6,4,3,1,1,0,1,0,0,0,0,0,0,0])

p1=SubPlot()
p.addSubPlot(p1)

p1L_all=LayerXY(z_grid, p1_all, name="GOODS-N AGNs",color=Color.BLACK)
p1L_all.style=Style(chartType=Style.HISTOGRAM_EDGE)
p1L_all.xaxis=Axis(range=[0.16,6])
p1L_all.xaxis.setAxisType(AxisType.LOG)
p1L_all.yaxis=Axis(range=[0,35])
p1L_all.xaxis.tick.setFixedValues(Double1d([0,1,2,3,4,5,6]))
p1L_all.xaxis.getTick().setLineWidth(1)
p1L_all.xaxis.getTick().setHeight(0.05)
#p1L_all.xaxis.tick.visible=0
p1L_all.xaxis.tick.label.visible=0
p1L_all.xaxis.title.visible=0
p1L_all.yaxis.tick.label.fontSize=8
p1L_all.yaxis.title.text="Number"
p1L_all.yaxis.title.fontSize=8
p1.addLayer(p1L_all)
p1L_all.xaxis.auxAxes[0].tick.visible=0

p1L_160=LayerXY(z_grid, p1_160, name="160$\\micro$m detected
 only",color=Color.GREEN)
p1L_160.style=Style(chartType=Style.HISTOGRAM_EDGE, fillEnabled=1,
 fillPaint=Color.GREEN)
p1.addLayer(p1L_160)

p1L_100=LayerXY(z_grid, p1_100, name="100$\\micro$m detected only",color=Color.BLUE)
p1L_100.style=Style(chartType=Style.HISTOGRAM_EDGE, fillEnabled=1,
 fillPaint=Color.BLUE)
p1.addLayer(p1L_100)
```

```
p1L_both=LayerXY(z_grid, p1_both, name="Detected in both bands",color=Color.GRAY)
p1L_both.style=Style(chartType=Style.HISTOGRAM_EDGE, fillEnabled=1,
 fillPaint=Color.GRAY)
p1.addLayer(p1L_both)

p2_all =Double1d([0,0,0,1,0,0,3,7,17,17,17,15,7,4,4,3,1,0,0,0,0])
p2_both=Double1d([0,0,0,0,0,0,0,0, 0, 1, 1, 1,0,0,0,0,0,0,0,0,0])
p2_100 =Double1d([0,0,0,0,0,0,0,0, 1, 1, 1, 1,0,0,0,0,0,0,0,0,0])
p2_160 =Double1d([0,0,0,0,0,0,0,0, 2, 2, 1, 2,0,0,1,0,0,0,0,0,0])

p2=SubPlot(SubPlotGridConstraints(0,1))
p.addSubPlot(p2)

p2L_all=LayerXY(z_grid, p2_all,color=Color.BLACK)
p2L_all.style=Style(chartType=Style.HISTOGRAM_EDGE)
p2L_all.inLegend=0
p2L_all.xaxis=Axis(range=[0.16,6])
p2L_all.xaxis.setAxisType(AxisType.LOG)
p2L_all.yaxis=Axis(range=[0,34])
p2L_all.xaxis.tick.setFixedValues(Double1d([0,1,2,3,4,5,6]))
p2L_all.xaxis.getTick().setLineWidth(1)
p2L_all.xaxis.getTick().setHeight(0.05)
p2L_all.xaxis.tick.label.visible=0
p2L_all.xaxis.title.visible=0
p2L_all.yaxis.tick.label.fontSize=8
p2L_all.yaxis.title.text="Number"
p2L_all.yaxis.title.fontSize=8
p2.addLayer(p2L_all)
p2L_all.xaxis.auxAxes[0].tick.visible=0

p2L_160=LayerXY(z_grid, p2_160,color=Color.GREEN)
p2L_160.style=Style(chartType=Style.HISTOGRAM_EDGE, fillEnabled=1,
 fillPaint=Color.GREEN)
p2L_160.inLegend=0
p2.addLayer(p2L_160)

p2L_100=LayerXY(z_grid, p2_100,color=Color.BLUE)
p2L_100.style=Style(chartType=Style.HISTOGRAM_EDGE, fillEnabled=1,
 fillPaint=Color.BLUE)
p2L_100.inLegend=0
p2.addLayer(p2L_100)

p2L_both=LayerXY(z_grid, p2_both)
p2L_both.style=Style(chartType=Style.HISTOGRAM_EDGE, fillEnabled=1,
 fillPaint=Color.GRAY)
p2L_both.inLegend=0
p2.addLayer(p2L_both)

p3_all =Double1d([0,0,3,8,12,12,31,26,26,25,19,20,12,10,9,7,2,1,0,1,0])
p3_both=Double1d([0,0,1,1, 3, 3, 6, 3, 3, 4, 2, 2, 0, 0,0,0,0,0,0,0,0])
p3_100 =Double1d([0,0,1,1, 5, 3, 7, 4, 5, 4, 2, 2, 0, 0,0,0,0,0,0,0,0])
p3_160 =Double1d([0,0,1,1, 5, 5,10, 6, 6, 5, 2, 3, 0, 1,1,0,0,0,0,0,0])

p3=SubPlot(SubPlotGridConstraints(0,2))
p.addSubPlot(p3)

p3L_all=LayerXY(z_grid, p3_all,color=Color.BLACK)
p3L_all.style=Style(chartType=Style.HISTOGRAM_EDGE)
p3L_all.inLegend=0
p3L_all.xaxis=Axis(range=[0.16,6])
p3L_all.xaxis.setAxisType(AxisType.LOG)
p3L_all.xaxis.tick.setFixedValues(Double1d([0,1,2,3,4,5,6]))
p3L_all.xaxis.getTick().setLineWidth(1)
p3L_all.xaxis.getTick().setHeight(0.05)
p3L_all.yaxis=Axis(range=[0,34])
p3L_all.xaxis.tick.visible=1
p3L_all.xaxis.tick.label.fontSize=8
p3L_all.xaxis.title.text="Redshift"
p3L_all.xaxis.title.fontSize=8
p3L_all.yaxis.tick.label.fontSize=8
p3L_all.yaxis.title.text="Number"
p3L_all.yaxis.title.fontSize=8
```

```
p3.addLayer(p3L_all)
p3L_all.xaxis.auxAxes[0].tick.visible=0

p3L_160=LayerXY(z_grid, p3_160,color=Color.GREEN)
p3L_160.style=Style(chartType=Style.HISTOGRAM_EDGE, fillEnabled=1,
 fillPaint=Color.GREEN)
p3L_160.inLegend=0
p3.addLayer(p3L_160)

p3L_100=LayerXY(z_grid, p3_100,color=Color.BLUE)
p3L_100.style=Style(chartType=Style.HISTOGRAM_EDGE, fillEnabled=1,
 fillPaint=Color.BLUE)
p3L_100.inLegend=0
p3.addLayer(p3L_100)

p3L_both=LayerXY(z_grid, p3_both,color=Color.GRAY)
p3L_both.style=Style(chartType=Style.HISTOGRAM_EDGE, fillEnabled=1,
 fillPaint=Color.GRAY)
p3L_both.inLegend=0
p3.addLayer(p3L_both)

# Legend
p.legend.visible=1
p.legend.columns=1
p.legend.position=PlotLegend.CUSTOMIZED
p.legend.setLocation(2.8,3.0)
p.legend.halign=PlotLegend.LEFT
p.legend.valign=PlotLegend.BOTTOM
p.legend.borderVisible = False
p.addAnnotation(Annotation(0.2,28,"Spec-z",color=Color.BLACK,fontSize=11))
p.addAnnotation(Annotation(0.2,-8,"Phot-z",color=Color.BLACK,fontSize=11))
p.addAnnotation(Annotation(0.2,-42,"All",color=Color.BLACK,fontSize=11))
```

Example 3.107. Using several panels with histograms.

3.36. Worked example: Styles

The example in this section creates a plot using several styles and colours for lines and plot symbols. Figure 7 in C. Gruppioni et al., A&A 518, L27 (2010) .

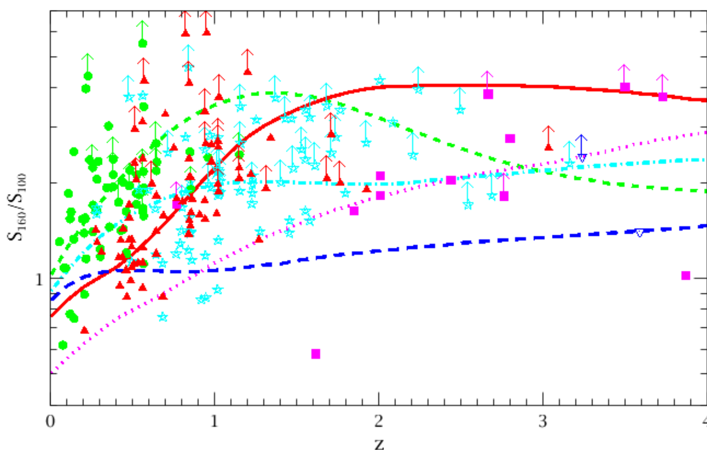


Figure 3.18. A plot using several styles and colours for lines and plot symbols.

```
from java.awt import Color
from herschel.ia.gui.plot.renderer.axtype import AxisType

x=Double1d([0.00000,0.100000,0.200000,0.300000,0.400000,0.500000,0.600000, \
  0.700000,0.800000,0.900000,1.00000,1.10000,1.20000,1.30000,1.40000,1.50000, \
  1.60000,1.70000,1.80000,1.90000,2.00000,2.10000,2.20000,2.30000,2.40000, \
  2.50000,2.60000,2.70000,2.80000,2.90000,3.00000,3.10000,3.20000,3.30000, \
  3.40000,3.50000,3.60000,3.70000,3.80000,3.90000,4.00000,4.10000,4.20000, \
```

```
  4.30000,4.40000,4.50000,4.60000,4.70000,4.80000,4.90000])

# GAL
galy=Double1d([1.03222,1.25094,1.49081,1.74462,2.01059,2.28914,2.57141,2.84729, \
  3.10365,3.33465,3.53097,3.68335,3.78715,3.83992,3.84221,3.79946,3.71873,3.61091, \
  3.48488,3.34745,3.20485,3.06281,2.92494,2.79436,2.67321,2.56291,2.46347,2.37459, \
  2.29566,2.22629,2.16592,2.11373,2.06849,2.02944,1.99586,1.96725,1.94287,1.92241, \
  1.90535,1.89116,1.87969,1.87064,1.86369,1.85858,1.85505,1.85300,1.85217,1.85245, \
  1.85373,1.85584])
gall=LayerXY(x,galy)
gall.style=Style(line=3, color=Color.GREEN, stroke=2, dashArray=[4,4])
gall.xaxis=Axis(range=[0,4], titleText="z")
gall.xaxis.setAxisType(AxisType.LINEAR)
gall.xaxis.title.fontSize=10
gall.xaxis.tick.interval=1
gall.xaxis.tick.minorNumber=9
gall.yaxis=Axis(range=[0.4,7], titleText="$\\mathrm{S_160/S_100}$")
gall.yaxis.setAxisType(AxisType.LOG)
gall.yaxis.title.fontSize=10

# STARB
starby=Double1d([0.912616,1.06472,1.21351,1.35387,1.48253,1.59905,1.70259,1.79352, \
  1.86953,1.93058,1.97602,2.00531,2.02072,2.02515,2.02118,2.01191,2.00092,1.99176, \
  1.98598,1.98427,1.98757,1.99676,2.01133,2.02998,2.05105,2.07340,2.09615,2.11933, \
  2.14282,2.16651,2.19010,2.21347,2.23635,2.25868,2.28044,2.30127,2.32095,2.33888, \
  2.35458,2.36749,2.37679,2.38218,2.38364,2.38127,2.37430,2.36275,2.34753,2.33303, \
  2.31973,2.30573])
starbl=LayerXY(x,starby)
starbl.style=Style(line=3, color=Color.CYAN, stroke=2, dashArray=[1,2,4,2])

# COMP
compy=Double1d([0.501706,0.562159,0.620910,0.686251,0.746177,0.796718,0.848579, \
  0.907463,0.973662,1.04519,1.11828,1.19289,1.26902,1.34396,1.41677,1.48817, \
  1.55816,1.62519,1.68898,1.74984,1.80912,1.86740,1.92393,1.97873,2.03165,2.08255, \
  2.13060,2.17504,2.21671,2.25848,2.30272,2.35204,2.40766,2.46871,2.53245,2.59683, \
  2.66079,2.72436,2.78559,2.84441,2.89868,2.94720,2.98878,3.02225,3.04582,3.05892, \
  3.06194,3.05405,3.03615,3.00939])
compl=LayerXY(x,compy)
compl.style=Style(line=3, color=Color.MAGENTA, stroke=2, dashArray=[1,3])

# AGN2
agn2y=Double1d([0.760343,0.857363,0.938720,1.00954,1.09040,1.19802,1.33700,1.50992,
 \
  1.71970,1.96892,2.25033,2.53001,2.79015,3.02753,3.24030,3.42664,3.58853,3.72476, \
  3.83835,3.92658,3.99002,4.03012,4.05141,4.06144,4.06581,4.06756,4.06678,4.06253, \
  4.05450,4.04236,4.02581,4.00460,3.97868,3.94816,3.91312,3.87413,3.83128,3.78502, \
  3.73568,3.68375,3.62957,3.57363,3.51634,3.45818,3.39824,3.33743,3.27520,3.21520, \
  3.15798,3.10293])
agn2L=LayerXY(x,agn2y)
agn2L.style=Style(line=1, color=Color.RED, stroke=2)

# AGN1
agn1y=Double1d([0.857343,0.951230,1.01367,1.04761,1.06091,1.06239,1.05883,1.05450, \
  1.05286,1.05524,1.06168,1.07179,1.08492,1.10048,1.11786,1.13652,1.15561,1.17397, \
  1.19142,1.20795,1.22359,1.23840,1.25238,1.26561,1.27814,1.29005,1.30133,1.31218, \
  1.32266,1.33314,1.34376,1.35463,1.36581,1.37737,1.38929,1.40153,1.41405,1.42675, \
  1.43960,1.45255,1.46554,1.47853,1.49147,1.50432,1.51704,1.52957,1.54183,1.55370, \
  1.56498,1.57554])
agn1L=LayerXY(x,agn1y)
agn1L.style=Style(line=3, color=Color.BLUE, stroke=2, dashArray=[6,4])

# GAL Dots
galdx=Double1d([0.458000,0.253000,0.210000,0.189000,0.278000,0.120000,0.437000, \
  0.200000,0.0500000,0.233000,0.0790000,0.519000,0.136000,0.299000,0.254000, \
  0.211000,0.438000,0.556000,0.136000,0.337000,0.562000,0.348000,0.114000, \
  0.224000,0.377000,0.0870000,0.456000,0.0700000,0.299000,0.286000,0.520000, \
  0.139000,0.278000,0.954000,0.207000,0.561000,0.638000,0.114000,0.845000, \
  0.202000,0.517000,0.478000,0.105000,1.14600,0.377000,0.560000,0.642000, \
  0.253000,0.639000,0.560000,0.557000,0.562000,0.377000,0.206000,0.561000, \
  0.457000,0.476000,0.848000,0.423000,0.534000,0.354000,0.410000,0.559000])
galdy=Double1d([1.19983,1.40916,3.97851,1.54511,1.64740,1.62750,2.01484,1.39483, \
  1.31159,2.37443,1.08603,1.71949,1.33610,1.17804,1.71285,1.15963,1.43642, \
```

```
  5.52177,2.52000,1.94232,3.49881,3.03669,1.54654,4.35791,1.95578,1.08363, \
  1.63426,0.619898,1.24222,2.34942,2.12971,0.751623,2.06348,2.41457,1.54825, \
  1.93624,2.67497,0.776319,2.51536,0.894211,1.86535,2.19948,1.85952,2.46762, \
  2.38978,1.88730,2.33775,2.24698,2.19522,2.18212,1.67340,1.58139,2.13363, \
  1.55927,1.11779,1.21026,1.96498,1.92929,1.42285,1.84178,1.46110,1.74453,1.62121])
galdyel=Double1d(63)
galdyeh=Double1d([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,Double.POSITIVE_INFINITY, \
  0,0,0,0,0,Double.POSITIVE_INFINITY,0,0,0,0,0,0,0,0,0,0,0,0,0, \
  Double.POSITIVE_INFINITY,0,0,0,0,0,0,Double.POSITIVE_INFINITY, \
  Double.POSITIVE_INFINITY,0,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY, \
  Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,0,0,Double.POSITIVE_INFINITY, \
  0,0,0,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,0, \
  Double.POSITIVE_INFINITY,0,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY])
galdL=LayerXY(galdx,galdy)
galdL.errorY=[galdyel,galdyeh]
galdL.style=Style(line=0, color=Color.GREEN, symbolShape=SymbolShape.FCIRCLE,
 symbolSize=5)

# STARB Stars
starbdx=Double1d([1.22400,0.276000,0.792000,0.971000,0.638000,2.07800,0.276000, \
  0.837000,2.00000,1.14800,0.678000,1.44900,1.76000,1.27000,0.965000,1.01300, \
  1.52300,4.42800,0.556000,0.817000,1.24800,1.54800,0.590000,0.534000,0.634000, \
  1.01600,0.835000,0.937000,0.839000,2.23500,1.46500,0.846000,0.472000,1.54800, \
  1.36300,1.01200,1.67800,1.15200,2.49000,1.79000,0.761000,1.57400,0.821000, \
  1.42400,0.835000,0.845000,1.73200,0.914000,0.678000,1.22600,1.91700,1.15200, \
  0.486000,1.52500,0.935000,0.711000,1.70500,2.20300,1.22300,1.54800,3.15700, \
  0.784000,1.60400,1.47300,1.01300,1.02100,0.855000,1.44900,0.850000,1.02900, \
  1.40000,0.940000,0.959000,1.22400,1.01700,1.57400,1.03100,1.02200,1.01600, \
  2.68200,2.53800,0.796000])
starbdy=Double1d([1.86449,1.67325,1.41945,1.93981,1.19289,3.11257,1.60635, \
  1.40437,4.23058,1.89618,0.761027,1.39534,3.40534,1.41488,2.83191,2.78433, \
  2.56756,3.24429,3.78861,2.80002,3.17892,1.63983,1.90686,1.91320,1.66525, \
  1.32813,4.64642,1.18533,1.55998,3.95584,3.19948,1.39701,3.73524,3.40942, \
  3.72253,0.927855,3.51171,3.48882,3.41758,1.86501,1.20016,3.23265,1.88261, \
  3.19712,1.23492,2.02955,2.80465,0.860281,1.12129,2.11778,2.68806,2.67497, \
  1.13344,2.55834,2.55175,2.50633,2.49022,2.45448,1.72822,2.37990,2.30865, \
  1.28932,2.28265,2.23398,2.21146,2.17891,1.14953,1.52855,1.40540,2.03310, \
  2.01353,0.874178,1.99085,1.98116,1.89042,1.52757,1.87413,1.86122,1.83856, \
  1.83197,1.71521,1.68282])
starbdyel=Double1d(82)
starbdyeh=Double1d([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \
  Double.POSITIVE_INFINITY,0,0,Double.POSITIVE_INFINITY,0,0, \
  Double.POSITIVE_INFINITY,0,Double.POSITIVE_INFINITY,0,Double.POSITIVE_INFINITY, \
  Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,0,0,Double.POSITIVE_INFINITY, \
  0,Double.POSITIVE_INFINITY,0,0,Double.POSITIVE_INFINITY,0,0,0,0, \
  Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,0,Double.POSITIVE_INFINITY, \
  Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY, \
  Double.POSITIVE_INFINITY,0,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,0, \
  Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY, \
  Double.POSITIVE_INFINITY,0,0,0,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,
 \
  0,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,0, \
  Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY, \
  Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY])
starbdL=LayerXY(starbdx,starbdy)
starbdL.errorY=[starbdyel,starbdyeh]
starbdL.style=Style(line=0, color=Color.CYAN, symbolShape=SymbolShape.STAR,
 symbolSize=6)

# COMP F-Squares
compdx=Double1d([2.00200,2.42000,2.00500,2.79400,3.49300,2.66000,3.72200, \
  1.84300,3.86500,2.43400,2.75600,1.61000,0.764000])
compdy=Double1d([2.11174,7.80762,1.83755,2.77715,4.02086,3.82929,3.75145, \
  1.64104,1.02414,2.04871,1.82877,0.580657,1.71854])
compdyel=Double1d(13)
compdyeh=Double1d([0,Double.POSITIVE_INFINITY,0,0,Double.POSITIVE_INFINITY, \
  Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,0,0,0,Double.POSITIVE_INFINITY,
 \
  0,Double.POSITIVE_INFINITY])
compdL=LayerXY(compdx,compdy)
compdL.errorY=[compdyel,compdyeh]
```

```
compdL.style=Style(line=0, color=Color.MAGENTA, symbolShape=SymbolShape.FSQUARE,
 symbolSize=5)

# AGN2 F-TRIANGLE
agn2dx=Double1d([0.279000,0.473000,0.410000,0.640000,0.423000,0.489000, \
  0.433000,0.638000,0.639000,0.507000,0.858000,0.475000,0.306000,0.555000, \
  0.458000,0.438000,0.934000,0.903000,0.946000,0.817000,0.694000,0.975000, \
  1.21500,0.799000,0.460000,0.557000,0.529000,0.489000,1.19500,0.849000, \
  0.202000,0.566000,0.840000,0.851000,1.01400,0.475000,0.559000,1.02100, \
  1.33600,0.841000,0.935000,0.271000,0.557000,0.839000,1.00700,0.679000, \
  0.840000,0.489000,0.847000,0.763000,0.508000,1.92000,1.70500,0.746000, \
  0.683000,1.26400,0.975000,0.936000,1.01400,0.940000,0.836000,1.01600, \
  3.02700,0.417000,0.502000,0.556000,1.14500,1.14400,1.67800,0.517000, \
  1.75900,0.612000,1.30700,1.01800,0.454000,0.484000])
agn2dy=Double1d([1.30293,1.32434,1.16939,1.83265,1.32471,1.26253,1.17360, \
  1.41605,1.20282,2.31398,1.78550,1.56257,1.21428,3.12925,1.06307,1.44278, \
  1.76653,1.59544,5.95989,5.93360,3.30248,2.34596,2.13678,1.75307,0.878780, \
  1.12357,1.12512,1.08773,4.48744,2.08833,0.688018,4.21850,4.14075,1.61897, \
  1.98866,1.71554,1.25268,3.74834,2.78586,2.09353,3.37877,1.42903,2.39244, \
  2.39662,1.60694,0.881049,1.53038,1.33242,1.46893,1.79275,2.96701,1.91567, \
  2.85022,2.58988,1.58679,1.33070,2.65534,1.54625,2.72044,2.66857,1.38688, \
  2.60687,2.59083,0.953587,1.11573,0.937821,2.24057,2.07835,2.07835,2.05569, \
  2.00705,1.99746,1.93249,1.89042,1.06601,0.988098])
agn2dyel=Double1d(76)
agn2dyeh=Double1d([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,Double.POSITIVE_INFINITY, \
  Double.POSITIVE_INFINITY,0,0,0,0,0,0,0,0,Double.POSITIVE_INFINITY,0,0, \
  Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,0,0,0,0, \
  Double.POSITIVE_INFINITY,0,0,Double.POSITIVE_INFINITY,0,0,0,0,0,0,0,0,0,0, \
  Double.POSITIVE_INFINITY,0,Double.POSITIVE_INFINITY,0,0,0,0,0,0, \
  Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,0,Double.POSITIVE_INFINITY, \
  Double.POSITIVE_INFINITY,0,0,0,Double.POSITIVE_INFINITY, \
  Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY, \
  Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY, \
  Double.POSITIVE_INFINITY,0,0])
agn2dL=LayerXY(agn2dx,agn2dy)
agn2dL.errorY=[agn2dyel,agn2dyeh]
agn2dL.style=Style(line=0, color=Color.RED, symbolShape=SymbolShape.FTRIANGLE,
 symbolSize=6)

# AGN1 U-TRIANGLE
agn1dx=Double1d([4.16400,3.23300,3.58300])
agn1dy=Double1d([0.608247,2.42214,1.41423])
agn1dyel=Double1d(3)
agn1dyeh=Double1d([0,Double.POSITIVE_INFINITY,0])
agn1dL=LayerXY(agn1dx,agn1dy)
agn1dL.errorY=[agn1dyel,agn1dyeh]
agn1dL.style=Style(line=0, color=Color.BLUE, symbolShape=SymbolShape.UTRIANGLE,
 symbolSize=6)

p=PlotXY()
p.setPlotSize(5.0,3.0)
p.addLayer(gall)
p.addLayer(starbl)
p.addLayer(compl)
p.addLayer(agn2L)
p.addLayer(agn1L)
p.addLayer(galdL)
p.addLayer(starbdL)
p.addLayer(compdL)
p.addLayer(agn2dL)
p.addLayer(agn1dL)
```

Example 3.108. Plotting with customised line and plot styles.

3.37. Worked example: Two plots in one

The example in this section creates a plot made of two independent plots. Figure 2 in M. Baes et al., A&A 518, L53 (2010) .

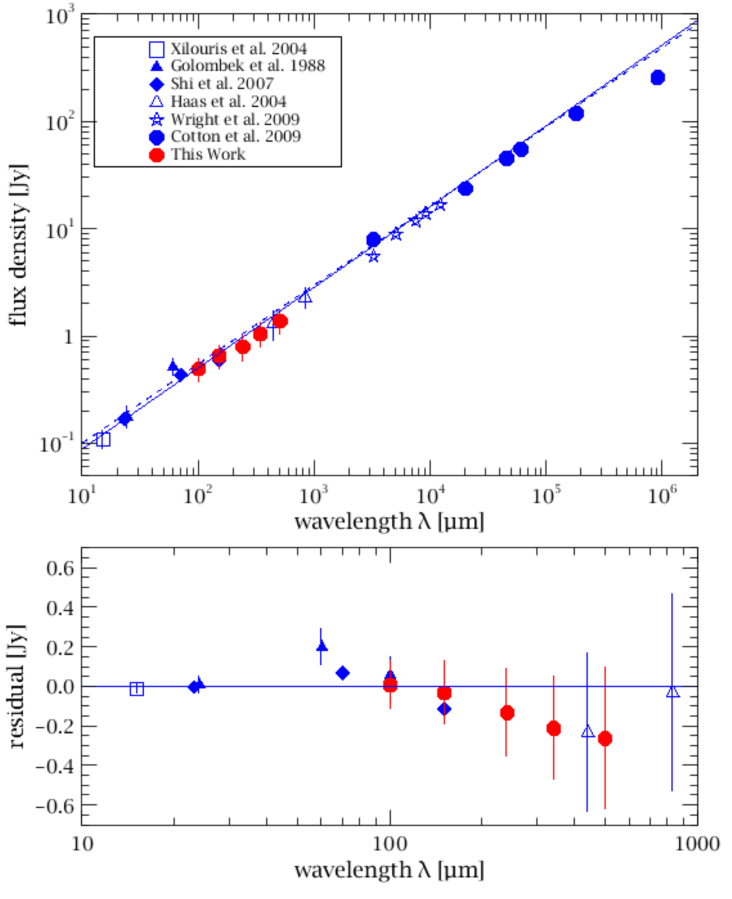


Figure 3.19. A plot made of two independent plots.

```
from java.awt.geom import Point2D
from java.awt import Color
from herschel.ia.gui.plot import PlotLegend
from herschel.ia.gui.plot.renderer.axtype import AxisType

# SubPlot Main
sll=LayerXY(Double1d(0), Double1d(0))
sll.xaxis=Axis(range=[10,2000000], titleText="wavelength $\\mathrm{\\lambda}$ [$\
\mathrm{\\micro}$m]")
sll.xaxis.setAxisType(AxisType.LOG)
sll.xaxis.title.fontSize=10
sll.yaxis=Axis(range=[0.05,1000], titleText="flux density [Jy]")
sll.yaxis.setAxisType(AxisType.LOG)
sll.yaxis.title.fontSize=10
sll.inLegend=0
sp0=SubPlot()
sp0.addLayer(sll)
p=PlotXY()
p.addSubPlot(sp0)
p.legend.visible=1

# Xilouris et al. 2004
xx=Double1d([15])
xy=Double1d([0.11])
xyel=Double1d([0.02])
xyeh=Double1d([0.02])
xl=LayerXY(xx,xy)
xl.errorY=[xyel,xyeh]
xl.style=Style(line=0, color=Color.BLUE, symbolShape=SymbolShape.SQUARE,
 symbolSize=6)
xl.name="Xilouris et al. 2004"
sp0.addLayer(xl)
```

```
# Golombek et al. 1988
gx=Double1d([24,60,100])
gy=Double1d([0.18,0.52,0.52])
gyel=Double1d([0.04,0.09,0.09])
gyeh=Double1d([0.04,0.09,0.09])
gl=LayerXY(gx,gy)
gl.errorY=[gyel,gyeh]
gl.style=Style(line=0, color=Color.BLUE, symbolShape=SymbolShape.FTRIANGLE,
 symbolSize=7)
gl.name="Golombek et al. 1988"
sp0.addLayer(gl)

#Shi et al. 2007
sx=Double1d([23,70,150])
sy=Double1d([0.17,0.44,0.6])
sl=LayerXY(sx,sy)
sl.style=Style(line=0, color=Color.BLUE, symbolShape=SymbolShape.FDIAMOND,
 symbolSize=7)
sl.name="Shi et al. 2007"
sp0.addLayer(sl)

# Haas et al. 2004
hx=Double1d([440,830])
hy=Double1d([1.3,2.3])
hyel=Double1d([0.4,0.5])
hyeh=Double1d([0.4,0.5])
hl=LayerXY(hx,hy)
hl.errorY=[hyel,hyeh]
hl.style=Style(line=0, color=Color.BLUE, symbolShape=SymbolShape.TRIANGLE,
 symbolSize=7)
hl.name="Haas et al. 2004"
sp0.addLayer(hl)

# Wright et al. 2009
wx=Double1d([3200,5000,7400,9000,12000])
wy=Double1d([5.6,9,12,14,17])
wl=LayerXY(wx,wy)
wl.style=Style(line=0, color=Color.BLUE, symbolShape=SymbolShape.STAR, symbolSize=7)
wl.name="Wright et al. 2009"
sp0.addLayer(wl)

# Cotton et al. 2009
cx=Double1d([3200,20000,45000,60000,180000,900000])
cy=Double1d([8,24,46,56,120,260])
cl=LayerXY(cx,cy)
cl.style=Style(line=0, color=Color.BLUE, symbolShape=SymbolShape.FOCTAGON,
 symbolSize=7)
cl.name="Cotton et al. 2009"
sp0.addLayer(cl)

# This work
tx=Double1d([100,150,240,340,500])
ty=Double1d([0.5,0.66,0.8,1.06,1.4])
tyel=Double1d([0.12,0.16,0.22,0.26,0.36])
tyeh=Double1d([0.12,0.16,0.22,0.26,0.36])
tl=LayerXY(tx,ty)
tl.errorY=[tyel,tyeh]
tl.style=Style(line=0, color=Color.RED, symbolShape=SymbolShape.FOCTAGON,
 symbolSize=7)
tl.name="This Work"
sp0.addLayer(tl)

# Fit solid line
slx=Double1d([10, 2000000])
sly=Double1d([0.09, 900])
sl1=LayerXY(slx,sly)
sl1.style=Style(line=1, color=Color.BLUE)
sl1.inLegend=0
sp0.addLayer(sl1)

# Fit dashed line
dlx=Double1d([10, 2000000])
```

```
dly=Double1d([0.1, 820])
dll=LayerXY(dlx,dly)
dll.style=Style(line=3, color=Color.BLUE, dashArray=[3,3])
dll.inLegend=0
sp0.addLayer(dll)

# Display legends
p.legend.visible=1
p.legend.columns=1
p.legend.position=PlotLegend.CUSTOMIZED
p.legend.halign=PlotLegend.LEFT
p.legend.valign=PlotLegend.BOTTOM
p.legend.setLocation(0.8,4.85)
# p.legend.halign=PlotLegend.RIGHT
# p.legend.valign=PlotLegend.TOP

# subplot residual
spr=SubPlot(SubPlotGridConstraints(0,1,1,0.6))
slrl=LayerXY(Double1d(0), Double1d(0))
slrl.inLegend=0
slrl.xaxis=Axis(range=[10,1000], titleText="wavelength $\\mathrm{\\lambda}$ [$\
\mathrm{\\micro}$m]")
slrl.xaxis.setAxisType(AxisType.LOG)
slrl.xaxis.title.fontSize=10
slrl.yaxis=Axis(range=[-0.7, 0.7], titleText="residual [Jy]")
slrl.yaxis.setAxisType(AxisType.LINEAR)
slrl.yaxis.title.fontSize=10
spr.addLayer(slrl)
p.addSubPlot(spr)

# Xilouris et al. 2004 R
xry=Double1d([-0.01])
xrl=LayerXY(xx,xry)
xrl.errorY=[xyel,xyeh]
xrl.style=Style(line=0, color=Color.BLUE, symbolShape=SymbolShape.SQUARE,
 symbolSize=6)
xrl.inLegend=0
xrl.name="Xilouris et al. 2004 R"
spr.addLayer(xrl)

# Golombek et al. 1988 R
gry=Double1d([0.01,0.2,0.06])
grl=LayerXY(gx,gry)
grl.errorY=[gyel,gyeh]
grl.style=Style(line=0, color=Color.BLUE, symbolShape=SymbolShape.FTRIANGLE,
 symbolSize=7)
grl.inLegend=0
grl.name="Golombek et al. 1988 R"
spr.addLayer(grl)

# Shi et al. 2007 R
sry=Double1d([0.0,0.07,-0.11])
srl=LayerXY(sx,sry)
srl.style=Style(line=0, color=Color.BLUE, symbolShape=SymbolShape.FDIAMOND,
 symbolSize=7)
srl.inLegend=0
srl.name="Shi et al. 2007 R"
spr.addLayer(srl)

# Haas et al. 2004
hry=Double1d([-0.23,-0.03])
hrl=LayerXY(hx,hry)
hrl.errorY=[hyel,hyeh]
hrl.style=Style(line=0, color=Color.BLUE, symbolShape=SymbolShape.TRIANGLE,
 symbolSize=7)
hrl.inLegend=0
hrl.name="Haas et al. 2004"
spr.addLayer(hrl)

# This work
trry=Double1d([0.01,-0.03,-0.13,-0.21,-0.26])
trl=LayerXY(tx,trry)
```

```
trl.errorY=[tyel,tyeh]
trl.style=Style(line=0, color=Color.RED, symbolShape=SymbolShape.FOCTAGON,
 symbolSize=7)
trl.inLegend=0
trl.name="This Work"
spr.addLayer(trl)


# Fit solid line
slrx=Double1d([10,1000])
slry=Double1d([0,0])
slrl=LayerXY(slrx,slry)
slrl.style=Style(line=1, color=Color.BLUE)
slrl.inLegend=0
spr.addLayer(slrl)
```

Example 3.109. Plotting two independent subplots.

3.38. Worked example: Coloured band

The example in this section creates a plot with different symbol styles, error bars and a coloured horizontal band. Figure 4 in T.D. Rawle et al., A&A 518, L14 (2010) .

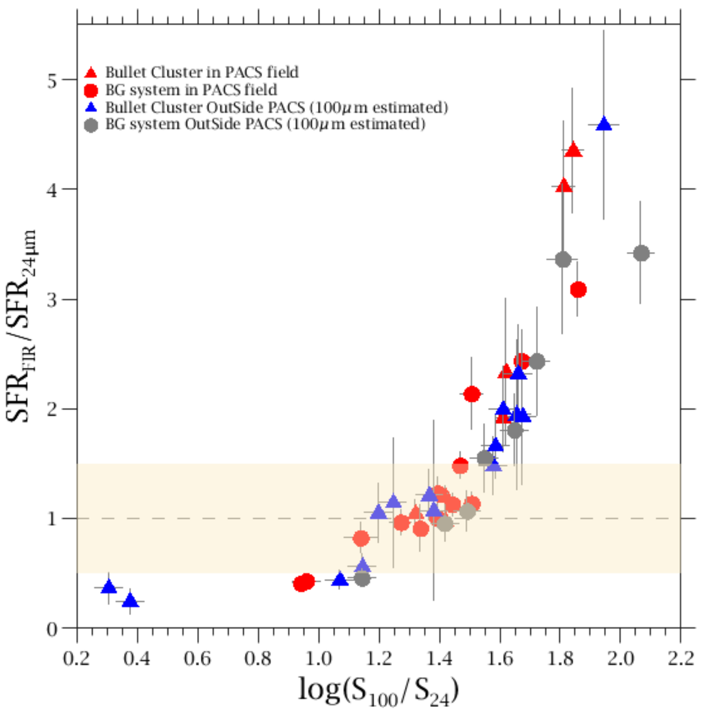


Figure 3.20. A plot with different symbol styles, error bars and a coloured horizontal band.

```
from java.awt import Color
from herschel.ia.gui.plot.renderer import AxisTickSide
from herschel.ia.gui.plot.renderer.RectAnnotationEngine import Type
from herschel.ia.gui.plot.renderer.axtype import AxisType

hd=LayerXY(Double1d([0.2,2.2]),Double1d([1,1]))
hd.style=Style(line=3,color=Color.GRAY,dashArray=[6,6])
hd.inLegend=0
hd.xaxis=Axis(range=[0.2,2.2])
hd.xaxis.tick.side=AxisTickSide.OUTWARD
hd.yaxis=Axis(range=[0,5.5])
hd.yaxis.tick.side=AxisTickSide.OUTWARD
hd.yaxis.tick.interval=1
hd.yaxis.tick.minorNumber=0
```

```
strip=RectAnnotation(0.2, 2.2, 0.5, 1.5)
strip.color=Color(251,232,189,105)
hd.addRectAnnotation(strip)

b0_1=Double1d([1.674486,1.069313,1.584537,1.654299,1.365276,1.578115,1.196919, \
  1.942886,1.144155,1.609685,0.374725,0.304246,1.245910,1.659333,1.379772])
b0_2=Double1d([0.046158,0.045599,0.044985,0.046158,0.044663,0.045469,0.044663, \
  0.049357,0.045100,0.045599,0.045342,0.045342,0.044663,0.045219,0.046013])
b0_3=Double1d([1.931079,0.439483,1.661472,1.946264,1.213043,1.480344,1.051577, \
  4.586347,0.561341,1.994514,0.245011,0.367738,1.145222,2.320756,1.071991])
b0_4=Double1d([0.618866,0.085960,0.291826,0.681112,0.230290,0.260769,0.264749, \
  0.856806,0.111786,0.390115,0.114122,0.144550,0.593335,0.443113,0.816351])
b0=LayerXY(b0_1,b0_3)
b0.errorX=[b0_2, b0_2]
b0.errorY=[b0_4, b0_4]
b0.style=Style(line=0, color=Color.GRAY, symbolShape=SymbolShape.FTRIANGLE,
 symbolSize=10, symbolColor=Color.BLUE)
b0.name="Bullet Cluster OutSide PACS (100$\\micro$m estimated)"

b1_1=Double1d([1.419899,1.619893,1.419907,1.319923,1.810285,1.841473,1.608309])
b1_2=Double1d([0.017994,0.017103,0.015887,0.017436,0.037015,0.036288,0.023305])
b1_3=Double1d([1.194547,2.334788,0.975215,1.046990,4.026402,4.353637,1.914610])
b1_4=Double1d([0.106746,0.670998,0.114467,0.120429,0.592627,0.566388,0.199998])
b1=LayerXY(b1_1,b1_3)
b1.errorX=[b1_2, b1_2]
b1.errorY=[b1_4, b1_4]
b1.style=Style(line=0, color=Color.GRAY, symbolShape=SymbolShape.FTRIANGLE,
 symbolSize=10, symbolColor=Color.RED)
b1.name="Bullet Cluster in PACS field"

bg0_1=Double1d([1.548598,2.066779,1.143054,1.648204,1.721049,1.417007, \
  1.807384,1.491405])
bg0_2=Double1d([0.045469,0.045219,0.044985,0.044873,0.045219,0.045733, \
  0.049990,0.045342])
bg0_3=Double1d([1.552600,3.422128,0.459055,1.807932,2.435197,0.955189, \
  3.364770,1.068040])
bg0_4=Double1d([0.312489,0.465107,0.080527,0.324565,0.495231,0.165624, \
  0.684698,0.183726])
bg0=LayerXY(bg0_1,bg0_3)
bg0.errorX=[bg0_2, bg0_2]
bg0.errorY=[bg0_4, bg0_4]
bg0.style=Style(line=0, color=Color.GRAY, symbolShape=SymbolShape.FCIRCLE,
 symbolSize=8, symbolColor=Color.GRAY)
bg0.name="BG system OutSide PACS (100$\\micro$m estimated)"

bg1_1=Double1d([1.391505,1.442690,1.467886,1.139075,0.959461,1.393919, \
  1.505945,1.506306,1.671487,1.336313,1.272283,0.941866,1.857896])
bg1_2=Double1d([0.014170,0.018310,0.015610,0.054709,0.046394,0.033695, \
  0.038559,0.023530,0.016676,0.022353,0.035198,0.027152,0.011652])
bg1_3=Double1d([1.005709,1.128696,1.483923,0.824007,0.430120,1.229956, \
  2.137710,1.136676,2.438485,0.910690,0.965880,0.409521,3.091505])
bg1_4=Double1d([0.091131,0.102801,0.118804,0.139412,0.058699,0.147821, \
  0.326636,0.105967,0.280889,0.211824,0.112502,0.052559,0.246040])
bg1=LayerXY(bg1_1,bg1_3)
bg1.errorX=[bg1_2, bg1_2]
bg1.errorY=[bg1_4, bg1_4]
bg1.style=Style(line=0, color=Color.GRAY, symbolShape=SymbolShape.FCIRCLE,
 symbolSize=8, symbolColor=Color.RED)
bg1.name="BG system in PACS field"

p=PlotXY()
p.setPlotSize(4,4)
p.addLayer(hd)
p.addLayer(b1)
p.addLayer(bg1)
p.addLayer(b0)
p.addLayer(bg0)

# legend
p.legend.visible=1
p.legend.borderVisible=0
```

```
p.legend.columns=1
p.legend.position=PlotLegend.CUSTOMIZED
p.legend.setLocation(0.4,4.0)
p.legend.halign=PlotLegend.LEFT
p.legend.valign=PlotLegend.BOTTOM
p.xaxis.title.text="log($\\mathrm{S_{100}}/\\mathrm{S_{24}}$)"
p.yaxis.title.text="$\\mathrm{SFR_{FIR}}/\\mathrm{SFR_{24{\\mu}m}}$"
```

Example 3.110. How to add error bars and customised horizontal bars to a plot.

3.39. Worked example: Plot with PACS and SPIRE spectra

The following is a commented example of how to produce a plot containing a PACS and SPIRE spectrum together. The example shows how to convert the data to be on the same wavescale (note that no attempt at calibrating the data to be on the same flux scale is made!) and how to manipulate spectral data so that it can be plotted. In addition the example shows how to set axes limits and titles, how to customise line styles and how to annotate and title the plot.

The script connects to the Herschel Science Archive to retrieve data, so you must be connected to the Internet and logged in. For more information on logging in to the Herschel Science Archive, see Section 1.4.1 .

The following image shows the plot produced by the script:

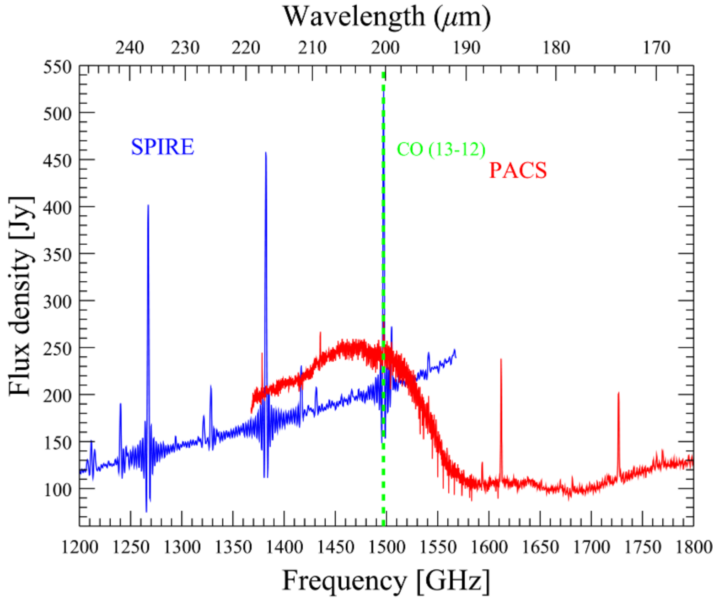


Figure 3.21. The result of the commented plot example presented in this section.

```
#   Script for spectral plot example
#
#  Plot 12CO 13-12 (1497 GHz, 200.27 micron) spectra from
#  PACS and SPIRE for AFGL 2688 together
#
#
# Get data (all public obsids)
# and extract the products that contain the CO 13-12 line
pacsobs = getObservation(1342199235, useHsa=True)
pacs_red_rebinned_cube = pacsobs.level2.getProduct("HPS3DRR").refs[0].product
```

```
#
spireobs = getObservation(1342247105, useHsa=True)
spire_spectrum = spireobs.level2.getProduct("HR_spectrum_point")["0000"]["SSWD4"]
#
#
# Extract central pixels from PACS cube
pacs_central_spectrum = extractRegionSpectrum(cube=pacs_red_rebinned_cube, \
regionType=herschel.ia.toolbox.cube.ExtractRegionSpectrumTask.Region.SINGLE_PIXEL, \
centerRow=2.0, centerCol=2.0)
#
# Convert PACS spectrum wavescale to GHz
pacs_spectrum = convertWavescale(ds=pacs_central_spectrum, to='GHz',
 overwrite=False)
#
#
# Extract wavescales and fluxes to plot against each other
spire_wave = spire_spectrum.wave
spire_flux = spire_spectrum.flux
pacs_wave = pacs_spectrum.wave
pacs_flux = pacs_spectrum.flux
#
# Create plot.
from java.awt import Color
p=PlotXY()
# Plot SPIRE data in blue
layer1  = LayerXY(spire_wave, spire_flux, color=Color.BLUE)
# Plot the PACS data in red
layer2  = LayerXY(pacs_wave, pacs_flux, color=Color.RED)
p.addLayer(layer1)
p.addLayer(layer2)
#
# Now set the axes limits and titles
p.xaxis.range = [1200.0, 1800.0]
p.yaxis.range = [60,550]
p.xtitle = "Frequency [GHz]"
p.ytitle = "Flux density [Jy]"
#
# Add top axis in wavelength in microns, GHz/c/1000 = um  ==> c/1000
c = 299792458.0 # m/s
xaux = ReciprocalAuxAxis(c/1000.0)
p.xaxis.removeAuxAxis(0)
p.xaxis.addAuxAxis(xaux)
xaux.setTitleText("Wavelength ($\\mu$m)")
xaux.getTick().setMinorNumber(4)
#
# Annotate plot
# Clear any old annotations first (helpful if you need to move annotations)
p.clearAnnotations()
# Annotations are placed using plot location (x,y)
p.addAnnotation(Annotation(1250,450,"SPIRE",color=Color.BLUE, fontSize=11))
p.addAnnotation(Annotation(1600,425,"PACS",color=Color.RED, fontSize=11))
#
# Add a line annotation at the location of the CO (13-12) line
from herschel.ia.gui.plot import LineAnnotation
line = LineAnnotation(LineAnnotation.XLINE, 1497)
line.color=Color.GREEN
line.dashArray=([3])
line.lineWidth=1.5
layer2.addLineAnnotation(line)
p.addAnnotation(Annotation(1510,450,"CO (13-12)",color=Color.GREEN, fontSize=9))
#
#
# Add a title
p.setTitleText("CO(13-12) with PACS and SPIRE")
# Now save a PDF file,
#it will be saved in the directory from which you opened HIPE
p.saveAsPDF("Fig1.pdf")
```

Example 3.111. Complete example using PACS and SPIRE data of AFGL 2688.

3.40. The TablePlotter

The *TablePlotter* utility is a tool to view and analyze table datasets organised in columns with an equal number of rows, for instance time-ordered detector signals. In addition the tool provides advanced means of interactively selecting subsets of this data and create new table datasets from these selections. The TablePlotter appears as a tab in the *Editor* view.

TablePlotter does *not* support other types of datasets.

3.40.1. Invoking TablePlotter

- **Invoking TablePlotter as a Viewer in HIPE**

  The TablePlotter works with Table Datasets and products that contain Table Datasets. For example, double clicking on a FITS binary table file in the Navigator view of HIPE will load the file into a product containing a table dataset and automatically bring up the product viewer. Right clicking on the table dataset within the product and selecting *Open With* leads to a choice of viewers and tools that can be applied (see Figure 3.22).

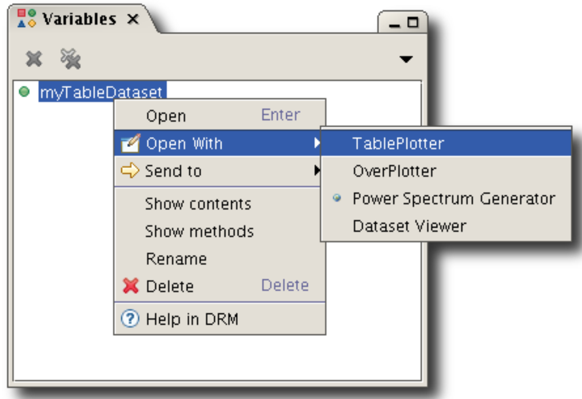


Figure 3.22. Viewers available for a table dataset in the product viewer, among them TablePlotter and OverPlotter.

  Selecting *TablePlotter* opens the table dataset in the main TablePlotter screen (see Figure 3.23).

- **Invoking TablePlotter from the command line or from a script**

  You can also invoke TablePlotter from the command line. First we need to import TablePlotter and the window manager:

```
from herschel.ia.gui.explorer.table import TablePlotter
from herschel.share.component import WindowManager
```

  Assuming tbs is a `TableDataset`, then the Table Plotter would be invoked by the following commands in a Jython script:

```
wm = WindowManager.getDefault()
wm.addWindow('test', TablePlotter(tbs).component, 1)
```

  or by the single command:

```
WindowManager.getDefault().addWindow("test", TablePlotter(tbs).component, 1)
```

  If you have a product created by reading in a FITS file containing a binary table, the first table dataset can be easily extracted with the default method. For instance, if a FITS file was read by double clicking on it in the navigator view, a product will appear as a variable. Assuming the variable name is "Myfile", the following command lines send it to TablePlotter.

```
wm = WindowManager.getDefault()
```

```
wm.addWindow("test", TablePlotter(Myfile.default).component, 1)
wm.addWindow('test', TablePlotter(TablePlotterExerciseFile["HDU_1"]).component,
 1)
```

If the product contains more than one dataset, the desired table dataset can be retrieved by its name. If you don't know the name of the dataset, a list of datasets can be obtained with the keySet method. In the following example the list of dataset names is obtained and printed, then the first dataset is chosen and displayed in TablePlotter.

```
wm = WindowManager.getDefault()
datasets = Myfile.keySet()     #Get the names of the datasets
print datasets                 #Here you see the names of the datasets within the
 product
datasetName = datasets[0]      #Choose your dataset, in this case the first with
 index 0
wm.addWindow("test", TablePlotter(Myfile[datasetName]).component, 1)
```

If invoked from the command line, the TablePlotter will appear in its own window, instead of a HIPE view.

If the name of the dataset is unknown, but its sequence number is known, the following shortcut can be used, in this case for the first dataset with index 0:

```
wm = WindowManager.getDefault()
wm.addWindow("test", TablePlotter(Myfile[Myfile.keySet()[0]]).component, 1)
```

3.40.2. Layout of the TablePlotter

When TablePlotter is invoked, it displays an X/Y-plot of the first two columns of the selected `TableDataset` (See Figure 3.23). The TablePlotter GUI contains three major components: the plot display area, the plot control panel on the right, and axis selection boxes on the bottom. Sometimes it is necessary to adjust the window size and the sizes of the sections to see all components.

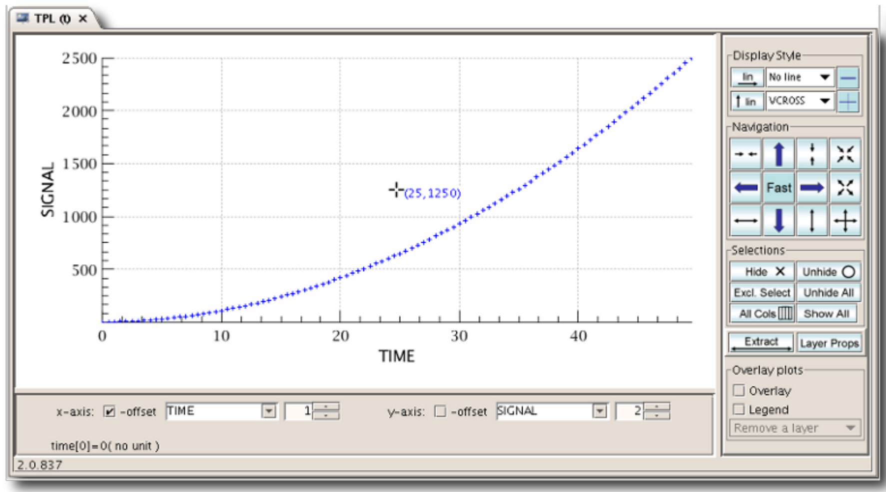


Figure 3.23. Layout of the TablePlotter GUI.

3.40.3. Controls and functions

The TablePlotter provides the following control buttons to view and analyze data.

- **X and Y- Axis Selection:**

  Under the graphics display area, two selector arrangements allow to assign columns in the table to the X and Y-axis of the plot. The elements of each selector are a combo box and a spinner.

By default the first column of the TableDataset is associated with the X-axis. The second column is initially associated with the Y-axis.

Clicking the arrow on the right of the combo box invokes a drop down menu with the displayable columns of the table dataset. Holding down the left mouse button and moving the mouse up or down scrolls through the columns if more than 8 columns are present. A column is selected by clicking on it. This list can be quite large. To help with the selection, a substring can be entered after clicking into the white name field of the combo box. Only columns with names containing this substring will be shown in the drop down menu. No distinction is made for upper or lower-case characters in this selection.

Columns can also be selected by index using the spinner, either by entering the index number directly after clicking into the index field, or by clicking on the up or down arrow buttons of the spinner. Fast forward/backward selection of columns in the spinner can be achieved by holding the left mouse button down and moving the mouse up or down.

The axis selector provides an additional "virtual" index column that allows to plot columns against the order in which they appear in the table dataset. This column only exists for convenience and is for instance not part of the extracted dataset, as shown further below.

In addition, two checkboxes named *"- offset"* allow you to subtract offsets from the data along both axes. This is useful, for example, if an axis corresponds to absolute times like TAI that start at an Epoch some time ago and bear a large offset compared to the time period covered by the data. When a checkbox is activated, the value of the subtracted offset appears below it.

- **Display style:**

  The control buttons in this section change the type of scaling of the X- and Y-axes, as well as the styles of lines and symbols used in the plot.

  lin The linear scale is selected for the X-axis. Clicking on the button will switch to logarithmic scale.

  log The linear scale is selected for the X-axis. Clicking on the button will switch to logarithmic scale.

  lin The linear scale is selected for the Y-axis. Clicking on the button will switch to logarithmic scale.

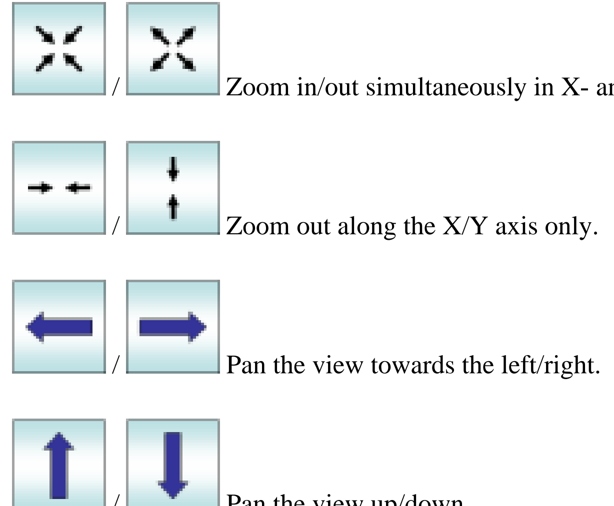
  log The linear scale is selected for the Y-axis. Clicking on the button will switch to logarithmic scale.

  The two pull-down menus select line- and symbol-styles. The selection of symbol styles is only available when the line styles are either *MARKED* , *MARK_DASHED* or *NONE* .

  / Increase/decrease symbol sizes.

- **Navigation:**

  The navigation field contains several buttons to zoom and pan within a plot. In addition the view can be controlled with the mouse pointer. Left clicking into the field, and pulling across an area with the left mouse button held down selects this area. This is called further on a hold-and-drag operation. When the mouse button is released, this area will be scaled so that it now fits the plot window (zoom-in).



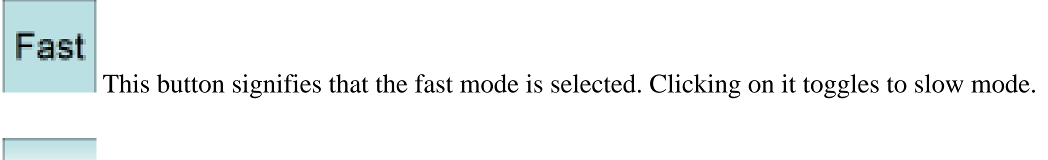
/ Zoom in/out simultaneously in X- and Y-axis.

/ Zoom out along the X/Y axis only.

/ Pan the view towards the left/right.

/ Pan the view up/down.

The size of each zooming or panning step is controlled by a toggle button at the center of the Navigation field as follows:

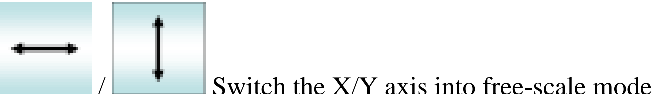


This button signifies that the fast mode is selected. Clicking on it toggles to slow mode.

Slow This button signifies that the slow mode is selected. Clicking on it toggles to the fast mode.

Layer Props This button opens the Preferences menu. The first entry in this drop-down menu opens a Properties window, where the factors can be changed that control fast and slow zooming and panning (for details see the *Preferences* section below).

This button switches into free-scale mode. It is one of the most frequently used buttons. The displayed ranges on X- and Y-axis are selected automatically to show all visible datapoints of the currently selected columns with optimal zoom parameters.



/ Switch the X/Y axis into free-scale mode.

- **Selections:**

  Table Plotter is not only a display tool for table datasets, but also a data selection tool. The selection feature can be used to hide or select a particular portion of the data points, to make use of the fast automatic scaling when scanning through many columns of data.

  The data selection feature, is also very useful for unplanned, ad-hoc, interactive data analysis tasks. Subsets of data in a table can be selected and extracted into new table datasets, that can then be subjected to other tools or tasks like the power spectrum tool. Typical applications would be for instance to manually remove glitches from a signal time stream, or to extract a specific period of a signal time stream out of a sequence of instrument configurations.

  The following buttons are relevant in this respect:

Show All This button signifies that all data points are being displayed. Deselected data points are replaced by a small red cross. The automatic scaling takes also deselected data into account. Clicking on this button switches to "Selected Only" display mode.

Sel Only This button signifies that only selected data points are being displayed. Deselected data points are not shown. The automatic scaling takes only selected data into account. Clicking on this button switches to "Show All" display mode.

Hide ✕ Clicking this button first, and then performing a drag-and-hold operation within the plot hides all selected data points within the selected rectangle. In "All Columns" mode only the X-axis range is taken into account (see below).

Unhide ○ Clicking this button first, and then performing a drag-and-hold operation within the plot selects all hidden data points within the selected rectangle. In "All Columns" mode only the X-axis range is taken into account (see below).

Excl. Select Clicking this button first, and then performing a drag-and-hold operation within the plot selects all data points within the selected rectangle and deselects everything outside. In "All Columns" mode only the X-axis range is taken into account (see below).

Unhide All This button will re-select all hidden data points.

Current Col This button signifies that selections and deselections only affect the two columns used for the plot. Clicking on this button will switch Table Plotter into "All Colum" mode.

All Cols This button signifies that selections and deselections affect all columns of the table. The selection is based on the range on the X-axis, while the selected Y-axis range is ignored. Clicking on this button will switch Table Plotter into "Current Column" mode.

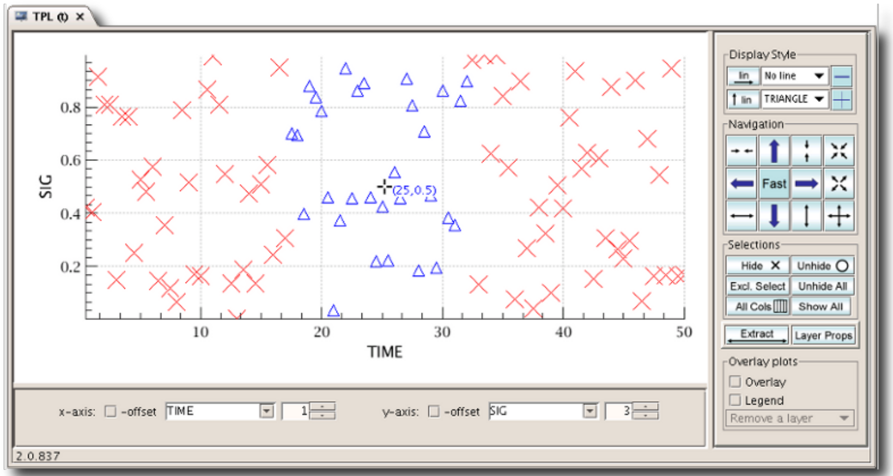


Figure 3.24. The plot with selected (blue) and hidden (red crosses) data points.

- **Printing and saving the plot:**

Right-click on the plot to display a context menu with the entries *Save as* and *Print* . You can save your plot in PDF, PNG, JPEG or EPS format.

- **Dataset Extraction:**

Besides visualisation, the Table Plotter can be handy for creating new datasets out of existing ones. Typically this is done in data analysis where a specific portion of interest is selected and saved into another dataset for subsequent analysis. The result becomes another table dataset. The extracted columns are the two being displayed while in "Current Columns" mode, or an arbitrary user selection of columns in "All Columns" mode. As a general rule, any row, where at least two columns represent a valid datapoint (X,Y), will appear in the result. Data that were "hidden" in such a row are replaced by NaNs. All other rows will be purged from the resulting table dataset.

The selection of datapoints is internally done with flags that exist for each datum. Making selections while choosing different columns for the X-axis can have sometimes results that first appear confusing, but make perfect sense in a logical way. Especially the **Exclusive Select** button and the **Unhide** button should be used with due consideration of the side effects.

This button extracts a subset of the data that remains selected after all prior selection operations. The selected data will be extracted into a new table dataset that will be fed back into the session. A name can be assigned to the new variable, which will appear in the Variables view.

If *Current Col* is selected, only the selected data points in the currently displayed column will be extracted.

If *All Cols* is selected, the selected data points in all the columns become available for extraction. After clicking *Extract*, a column selection window (see Figure 3.25) pops up, allowing to **Add** individual columns or **Add All** columns to a list. Individual columns can also be removed (**Remove**) again from the selection. The **Remove All** button allows to start over. **Up** and **Down** buttons are available to change the order of columns in the new dataset (see Figure 3.25).

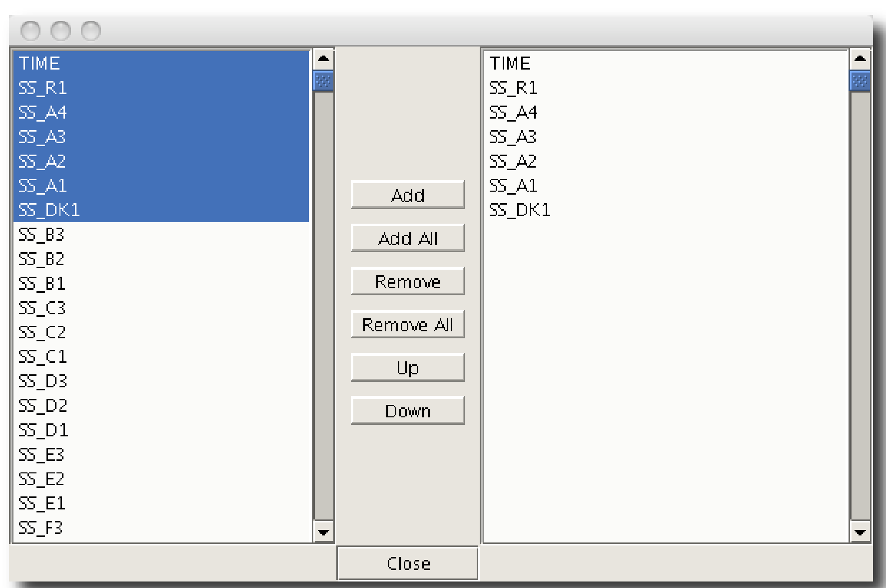


Figure 3.25. Extract Selected Data from Multi Columns to a New DataSet.

Clicking *Close* completes the extraction. You can then enter a name for the new dataset or accept the default.

The new table dataset appears as a new variable in the *Variables* view of HIPE.

- **Overlay Plots:**

  Even though the TablePlotter was primarily designed for single X-Y scattergram display, there is limited overlay capability available. For any more complex overlay plotting, the **Over Plotter** was created that is described in detail further down.

  Simple overlay plots are created by marking Overlay in the Overlay plots panel on the lower right, and selecting another column for the Y-axis. The old plot stays on display and the new X/Y-plot is overlaid with a different color. If different symbols, symbol sizes or line styles are required, they must be selected now. They can not be selected at a later stage. While Overlay is on, the Y-axis will have the same scale for all overlays and it is not possible to select another column for it. The only way to change a plot that was done earlier, is to remove the overlay in question with the **Remove a layer** drop-down menu, and selecting the column for the Y-axis again. Activating the **Legend** button shows the relation between colour and name of the overlay in a legend (see Figure 3.26).

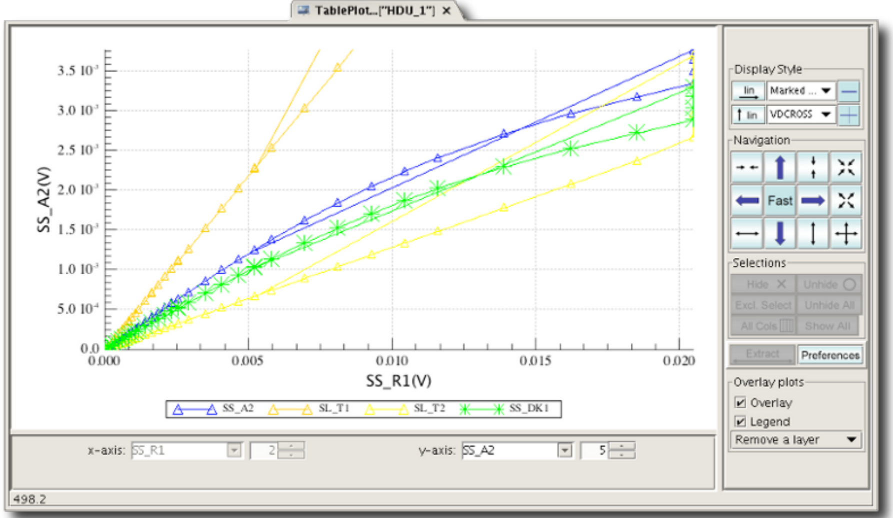


Figure 3.26. Simple overlay plots of different columns plotted against the same X-axis are created by marking the Overlay field.

- **Layer Props:**

  Layer Props This button provides a drop-down menu, giving access to the display rules for complex data and advanced layer management. See Figure 3.27 .

  The Table Plotter is able to show complex data in four different representations: the modulus, the real part, the imaginary part and the phase.

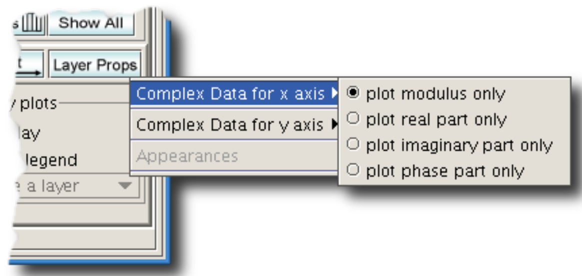


Figure 3.27. Preferences: Complex data can be displayed in four different ways as shown in this properties menu.

The selected preferences are stored in a properties file and will be remembered the next time you open Table Plotter.

- **Advanced command line control of TablePlotter**

  After invoking Table Plotter from the command line or a script, its display can be further controlled, allowing for integration of this tool into other applications that require interactive X/Y display and/or data selection. As stated before, the following imports must be performed first.

```
from herschel.ia.gui.explorer.table import TablePlotter
from herschel.share.component import WindowManager
```

  A Table Dataset tbs would be plotted as follows in a Jython script or from the command line. Note that in this case we retain the object tpl inbetween. This link enables us to access the Table Plotter and its components from the command line.

```
wm = WindowManager.getDefault()
tpl = TablePlotter(tbs)
wm.addWindow('test', tpl.component, 1)
```

  Now we should see a Table Plotter window as before, coming up detached of the HIPE window. We can now go about our business in HIPE. In case we make selections, we can get the result back into the session with the following commands.

```
extbl = tpl.activeLayerStruct.extractedTableDataset
flags = tpl.activeLayerStruct.flags
```

  The variable `extbl` now contains the resulting TableDataset after selection. It contains only rows with at least two valid entries. Deselected entries are replaced by NaNs. Sometimes however it is more convenient to just return the flags that were actually set for the original table dataset. This is done by the second line, where the flag array is saved in the variable flags. The dimensions of this flag array match those of the original table dataset tbs, but the type is a 2 dimensional Boolean array.

  The Table Plotter can also be preloaded with a flag array, which can be convenient in programmed applications.

3.41. The Over Plotter

The Over Plotter is a consequential evolution out of the Table Plotter. It can be thought of as a stack of individual Table Plotters with the same individual functionalities so that several graphs can be overlaid on top of each other with their individual scaling, panning, and data point selections. In addition, the OverPlotter provides capabilities to navigate the stack of layers in a coordinated fashion, i.e. like a stack of glued together transparencies. It further allows for synchronization of axis scales of different layers and synchronous selection of data across layers. As the basic Table Plotter functionalities apply to the single layers of Over Plotter as well, they will not be repeated here. Please refer to the applicable Table Plotter sections instead. This section will focus on all the functionalities that are specific to Over Plotter.

3.41.1. Invoke Over Plotter

To open a table dataset can be opened also in Over Plotter, right click on the corresponding variable in the *Variables* view of HIPE and choose *Open With* → *OverPlotter* from the menu (see Figure 3.22). Note that at any time there can exist only one instance of Over Plotter in a session, while Table Plotter can exist in many instances. In other words, selecting the option Table Plotter will always create a new view in HIPE, while selecting Over Plotter will create a new view for Over Plotter only once and after that send any further dataset to the same Over Plotter view as new layer.

3.41.2. Layout of Over Plotter

The Over Plotter main view looks very similar to the Table Plotter, but also shows a few important differences (see Figure 3.28). The main differences are the *Layer Controls* panel, which replaces the *Overlay Plots* panel, and the addition of four synchronization buttons. The plot area now contains obviously more graphs and a second pair of axes to the top and right sides.

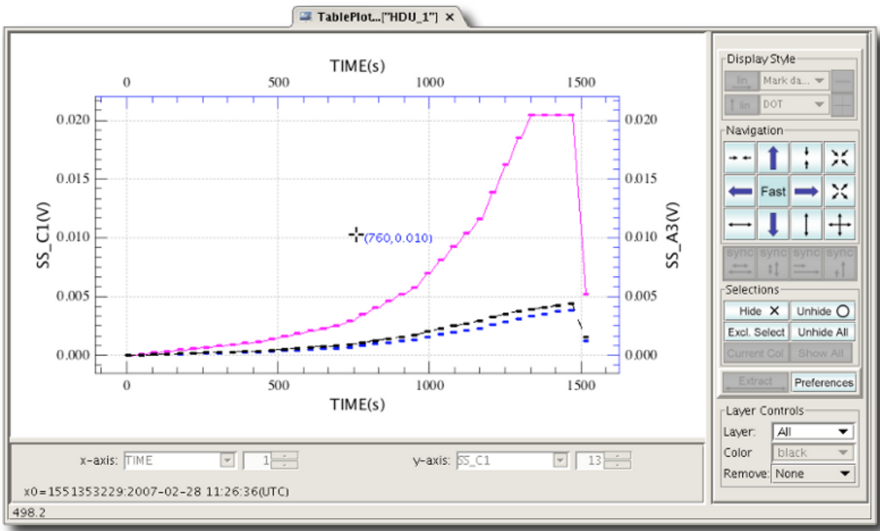


Figure 3.28. The main panel of Over Plotter is very similar to that of the Table Plotter. New features include the Layer Controls panel and the synchronization buttons. This Over Plotter is in *All Layers* mode.

The Over Plotter works in two main modes that can be chosen through the selection of layers: 1) a *Single Layer* mode and 2) an *All Layers* mode. The *Layer* drop down list shows all the available layers, that is, all the table datasets that have been sent to the Over Plotter so far. In addition, it contains an *All* entry to switch Over Plotter to *All Layers* mode.

Please note that the same dataset can be sent to Over Plotter more than once. This makes sense as one may want to overlay diagrams of different pairs of columns of the same table dataset. A limitation of the Over Plotter is that a pair of columns of two different datasets can not be combined into one diagram, as the equal number of rows of both datasets is not guaranteed. However, columns of two different datasets can easily be combined on the command line into two one table and then plotted into one diagram, provided the tables have the same length. For instance, if tbl1 and tbl2 were two related table datasets of equal length and we wanted to plot the column RA from one dataset against the column DEC from the other dataset, then we would execute 3 simple command lines like the following and then display the newly created table dataset in the Table Plotter.

```
# tbl1 and tbl2 are table datasets
tbl = TableDataset()            #create new empty table dataset
tbl['RA'] = tbl1['RA']          #add column RA
tbl['DEC'] = tbl2['DEC']        #add column DEC
#now open tbl in Table- or Over-Plotter.
```

In Figure 3.28 the Over Plotter is in *All Layers* mode and the graphs are shown in their selected colours. Only for two graphs the axes can be shown. These are called the primary and the secondary layers. The axes of the primary layer are the ones on the bottom (X-axis) and to the left (Y-axis), while the axes of the secondary layer are the ones on the top (X-axis) and to the right (Y-axis). The axes are shown in the colour of the respective layers.

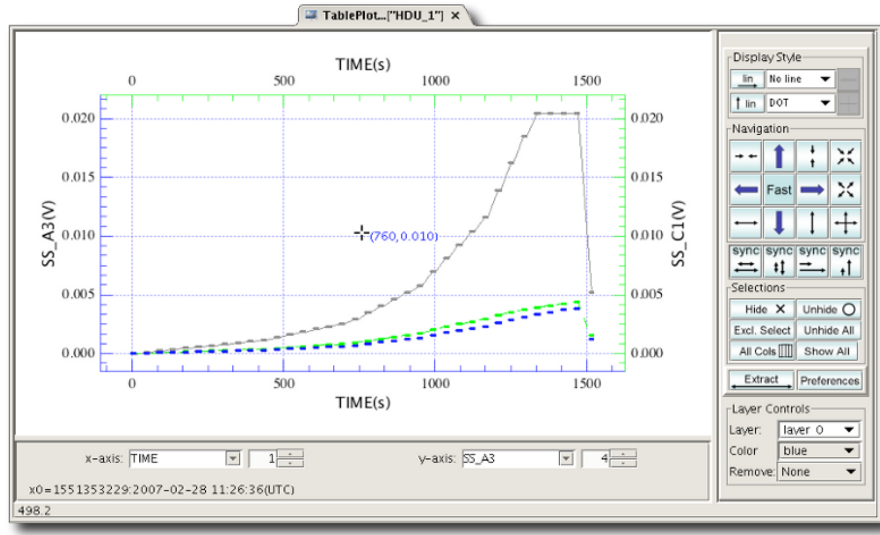


Figure 3.29. This Over Plotter is in "Single Layer" mode. The primary layer is displayed in its selected colour and the secondary layer is displayed in green. All other layers are displayed in grey colour.

In Figure 3.29 the Over Plotter is in *Single Layer* mode. In this case only the primary layer is shown in its selected colour. The secondary layer is always green and all other layers are all displayed in gray.

The assignment of primary and secondary layer is dynamic and changes when another layer is selected. Then the layer that was prime before becomes the secondary layer and will be displayed in green. The previously secondary layer changes to grey colour, unless it has been selected to be prime again, and the new prime layer is shown in its selected colour. An example is shown in Figure 3.29, where the third layer that was gray in the previous example is now chosen to be prime, and the colours change accordingly.

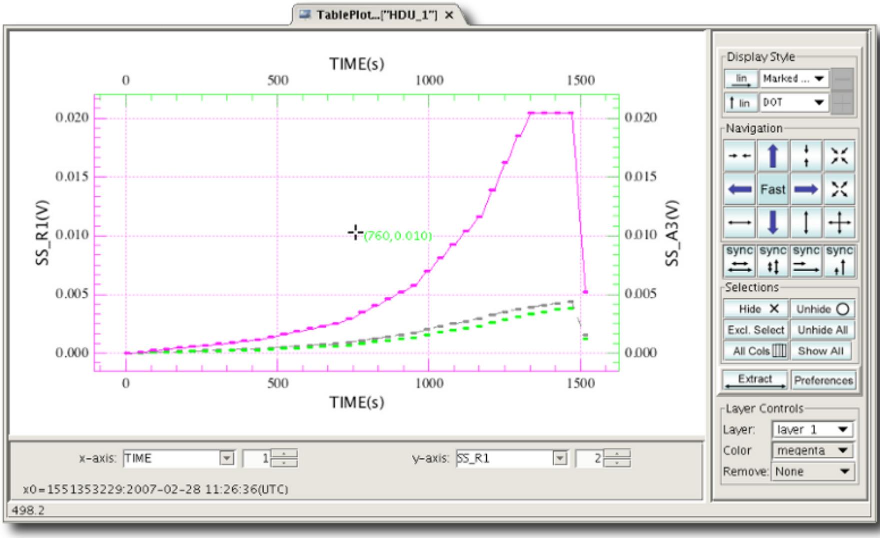


Figure 3.30. This Over Plotter is in "Single Layer" mode. The primary layer is displayed in its selected colour and the secondary layer is displayed in green. All other layers are displayed in grey colour. These are the same layers as in the previous figure, but after selecting Layer 1 to become prime.

3.41.3. Controls and Functions

Layer: layer 0 This drop down menu button shows the currently selected layer. If a single layer is selected, all actions apply to the selected layer only. Individual zooming, panning

etc. is performed in this mode. ALL indicates that all layers are selected and actions are performed on all layers simultaneously. A number of buttons are not applicable in this mode and are grayed out.

This drop down menu button allows to remove specific layers. This menu is available in any mode.

This button synchronizes the scale of the X-axis of the primary layer to the scale of that of the secondary layer, i.e. the distances between equal intervals on the X-axis display on the same scale.

This button synchronizes the scale of the Y-axis of the primary layer to the scale of that of the secondary layer, i.e. the distances between equal intervals on the Y-axis display on the same scale.

This button synchronizes the offset of the X-axis of the primary layer to the offset of the secondary layer, i.e. the primary layer is shifted in X-direction such that the values where the left Y-axis cuts the primary and secondary X-axes become the same.

This button synchronizes the offset of the Y-axis of the primary layer to the offset of the secondary layer, i.e. the primary layer is shifted in Y-direction such that the values where the bottom X-axis cuts the primary and secondary Y-axes become the same.

With all the possibilities of Table Plotter, except for the overlay function, available for each layer, many combinations are possible. In Figure 3.31 an overlay of 3 layers with different scaling and panning is shown. These are the same layers as in the previous plots, just with several display parameters changed to illustrate the possibilities. In addition the first layer (Layer 0) has a Y-log axis, and the blue circles are connected by solid lines. The second layer (Layer 1) has selected enlarged magenta filled diamonds, which are shown in green, because this is the secondary layer at this time and we are in single layer mode. The third layer (Layer 2) has selected blue enlarged triangles connected with a dashed line, which in this case is shown in gray colour, because this layer is neither primary nor secondary layer right now.

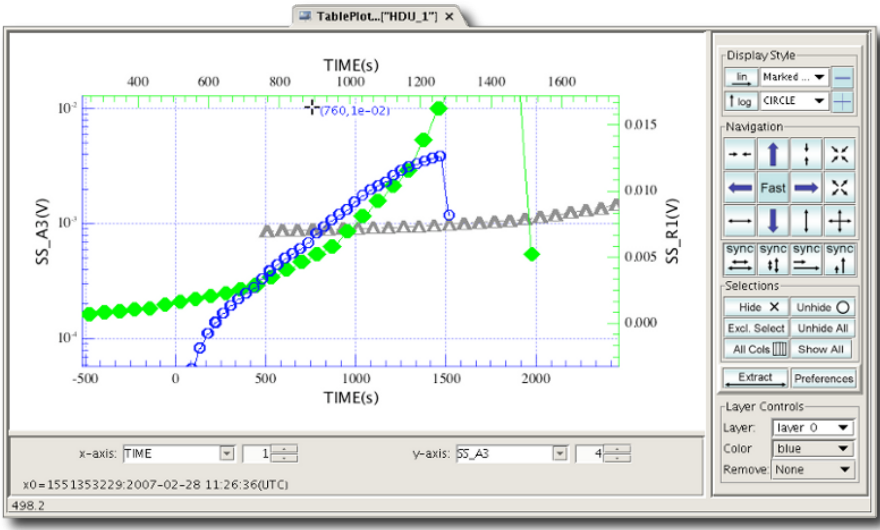


Figure 3.31. A complex example for illustration. The Over Potter is in "Single Layer" mode. The primary layer is displayed in blue with large symbols and connected by a line. The Y-axis is set to logarithmic mode. The secondary layer is displayed in green with large filled diamonds. The third layer is displayed in grey colour.

3.42. The Power Spectrum Generator

The Power Spectrum Generator computes a power spectrum for each column of a table dataset. You can access it by right-clicking on a Table dataset in the *Variables* window in HIPE and choosing *Open With → Power Spectrum Generator* .

This interface is a wrapper around a command-line tool described in the *Scripting Guide* : Section 5.4 in *Scripting Guide* . Please see that section for more details on the available options.

A time column must be selected in the main menu. The result is another table dataset, that can be displayed with the TablePlotter. An example of a signal timeline is shown in (Figure 3.32, below).

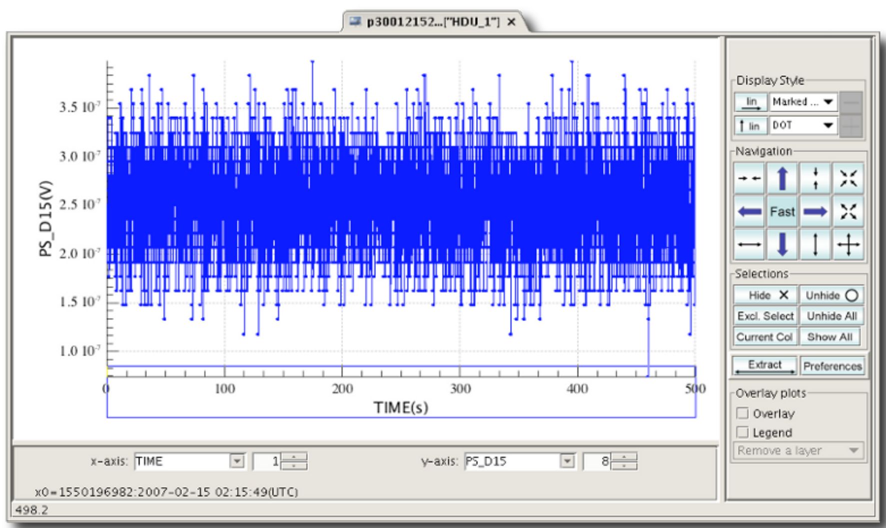


Figure 3.32. A signal timeline displayed in Table Plotter that the Power Spectrum generator can be applied to.

When the Power Spectrum Generator is invoked a menu appears. It consists of selectors for the time column in the dataset and its unit, in case that is not available or incorrect. There are two text boxes

labelled flimit and sigma , controlling the deglitcher, which can be de-activated in another selector below. The button Start FFT initiates the processing, which results in a new table datatset (see Figure 3.33 , below).

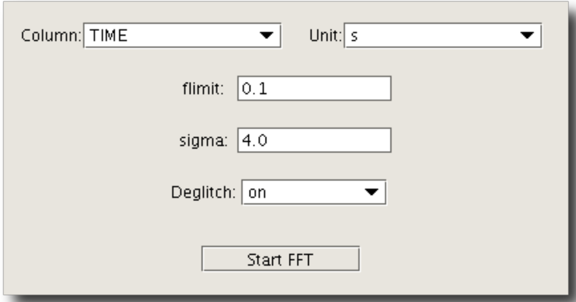


Figure 3.33. Main view of the Power Spectrum Generator.

Two text boxes are pre-filled with default values for the cut off frequency (flimit) and the deglitcher threshold (sigma). Both flimit and sigma can be changed in the menu.

After clicking the Start FFT button, and a short processing time, a widget appears that allows naming of the newly created table dataset. After pressing the OK button, the dataset is fed back into the session and appears in the Variables view of HIPE. The TablePlotter can be used to display the dataset as shown in Figure 3.34 .

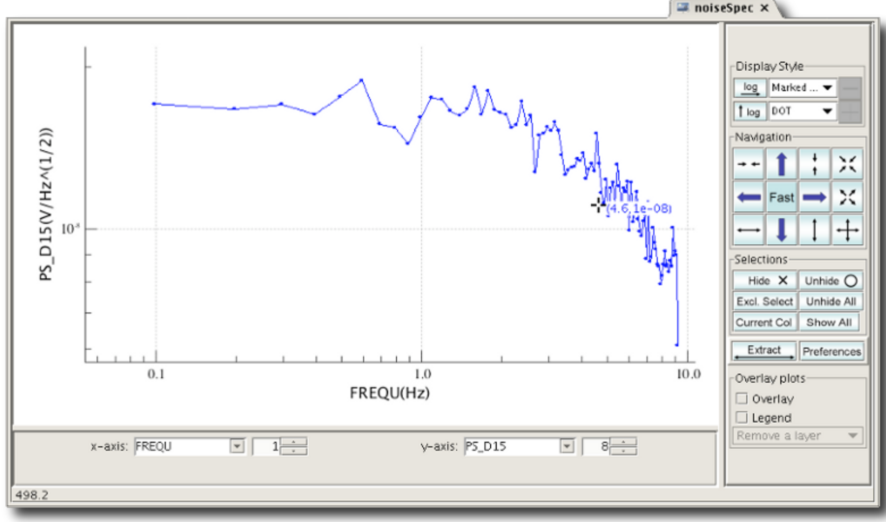


Figure 3.34. Displaying the newly created power spectra in the Table Plotter.

Chapter 4. Working with images

4.1. Summary

This chapter describes tasks for manipulating and analysing images.

Image I/O

- Importing and exporting images: Section 4.3 .

Image manipulation

- Manipulating the image axes: clipping/clamping, cropping, rotating, scaling, translating, transposing: Section 4.10 .
- Manipulating the image intensities (Image arithmetics tools): adding, subtracting, multiplying, dividing, computing logarithms, exponentials and square roots, and so on: Section 4.11.1 .
- Image smoothing: Section 4.11.2 .
- Flagging saturated pixels: Section 4.12 .
- Making histograms/obtaining cut levels: Section 4.13 .
- Stitching images together: Section 4.14 .
- Defining a World Coordinates System: Section 4.15 .

Image analysis:

- Looking at the image (image viewer/image explorer): Section 4.4 .
- Creating intensity profiles: Section 4.16 .
- Creating contour plots: Section 4.17 .
- Creating histograms of the whole image or of a region bounded by a circle, an ellipse, a rectangle or a polygon: Section 4.18 .
- Performing aperture photometry with a circular target aperture and an annular or rectangular sky aperture: Section 4.21 .
- Extracting sources: Section 4.19 .
- Fitting sources: Section 4.20 .
- Comparing PSFs to point source profiles: Section 4.22 .

For information on the types of variables accepted by these tasks, and on how to determine if a variable is of the correct type, see Section 4.2 .

4.2. Running image manipulation and analysis tasks

All the tasks described in this chapter work on variables of type `SimpleImage` in *HCSS User's Reference Manual* . These can be derived from a FITS file import, or even from an image file such as a JPEG.

If you select a variable of the right type in the *Variables* view of HIPE, or in the *Data* section of the Observation Viewer, all the image analysis tasks appear in the *Applicable* folder of the *Tasks* view. Double click on a task name to launch its graphical interface in the *Editor* view.

If the image analysis tasks do *not* appear in the *Applicable* folder of the *Tasks* view, you have probably selected a variable of the wrong type. Hovering the mouse pointer over a variable in the *Variables* view shows its type, as in the following figure:

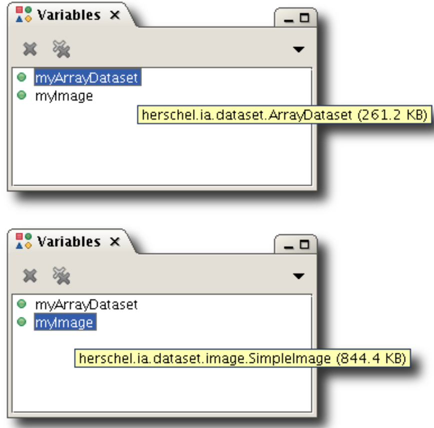


Figure 4.1. Finding variable types in the *Variables* view.

The tooltips shown in the previous figure display the full *qualified name* of the variable type, but what interests you is the last word, just before the size information. In the case shown in the previous figure, you see that one variable is of type `ArrayDataset` and the other is of type `SimpleImage` . You can apply image analysis tasks only to variables of type `SimpleImage` .

To discover the type of a component of an observation shown in the Observation Viewer, right click on it in the *Data* section and choose *Create Variable* . The corresponding variable appears in the *Variable* view, and you can look at the tooltip with the type.

You can also display a variable type via the command line in the *Console* view, via the following command:

```
print type(myVariable)
```

Example 4.1. Printing the type of a variable.

HIPE includes another image type, `RgbSimpleImage` in *HCSS User's Reference Manual* , which represents three-colour images. You cannot run the image analysis tasks described in this chapter on variables of this type. You must first extract the red, green or blue channel and save it as a `SimpleImage` variable. This is shown in the following example for the red channel of an `RgbSimpleImage` called `myRgbImage` (for creating an RGB image see Section 4.14.2):

```
red = SimpleImage()
red.setImage(myRgbImage.getRedByteImage())
```

Example 4.2. Setting the image value of a SimpleImage object.

You can now run image analysis tasks on the `red` variable.

4.3. Importing and exporting images

4.3.1. Importing

You can import an image in FITS format with a double click on the FITS file in the *Navigator* view. Alternatively, you can run the `fitsReader` in *HCSS User's Reference Manual* task. The image is assigned to a variable of type `SimpleImage` in *HCSS User's Reference Manual* . If the original image has an associated World Coordinate System, HIPE imports it into the new variable.

Once your image is imported as `SimpleImage` , you can execute all the image analysis tasks described in the following sections.

There are a few additional steps you may need to perform before being able to work on your image in HIPE. These are detailed below.

Adapting imported images

- **Setting the reference wavelength.** You can set the reference wavelength of your image with the following command:

```
myImage.setWavelength(value)
```

Example 4.3. Set the wavelength value of an image.

The default unit is microns, but you can specify an alternative unit. For instance:

```
myImage.setWavelength(value, Length.ANGSTROMS)
```

Example 4.4. Set the wavelength value of an image, including units.

For more information about units, see the *Scripting Guide* : Assigning Units in *Scripting Guide* .

- **Setting the flux unit of the image.** To set the flux unit of the image, issue the following commands:

```
myImage.setUnit(SurfaceBrightness.JANSKYS_PER_BEAM)
```

Example 4.5. Setting the units of an image directly.

For more information about units, see the *Scripting Guide* : Assigning Units in *Scripting Guide* .

- **Incorporating image components.** A `SimpleImage` product can contain several datasets. In addition to the flux values themselves, it can contain error, coverage and exposure datasets. If, for instance, you had the coverage of your external image as a separate FITS file, you would have imported it into HIPE as a separate `SimpleImage` . Assume you imported the external image to variable `myImage` and the exposure to variable `myExposure` . To include the exposure into `myImage` , use the following command:

```
myImage.exposure = myExposure.image
```

Example 4.6. Setting the exposure of an image.

To include error and coverage datasets, use these commands:

```
myImage.error = myError.image
myImage.coverage = myCoverage.image
```

Example 4.7. Setting the error and coverage datsets of an image.

Note

The size and scale of the exposure, coverage and/or error images must be the same as the flux image

4.3.2. Exporting

Once you finish working on your image in HIPE, you can export it to FITS format. Right click on the image variable name in the *Variables* view and choose *Send to → FITS file* . See Section 1.16.1 for more details on image export to FITS.

For more information on the structure of the exported FITS file, see Section 1.16.4.2 .

You can open the exported FITS files with external astronomical software like ds9 .

Tip

If the external application you are working with is VO-enabled, you can exchange data with HIPE via the SAMP interface, rather than FITS files. For more information about interoperating with the Virtual Observatory, see Section 1.17 .

4.4. Viewing an image

To display an image in HIPE, double click the image name, for instance in the *Variables* view. The standard image viewer display appears in the *Editor* view (see Figure 4.2).

Tip

If you have a large image, you may want to undock the image viewer from the *Editor* view, by clicking and dragging the viewer tab, and then enlarge it. You can obtain the same result by issuing the following command in the *Console* view:

```
Display(myImage)
```

Example 4.8. Constructing a Display object from an image.

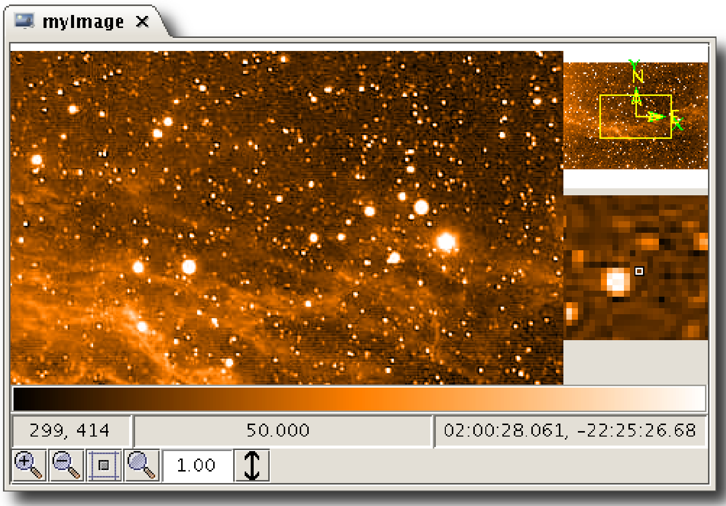


Figure 4.2. Viewing an image in HIPE.

The two smaller displays to the right of the main image display show the following:

- An overview of the full image with the area shown in the main display outlined by a rectangle. This display also shows the directions N and E on the display based on the WCS coordinates of the image, or X and Y if no WCS is present. You can change the position of the zoom/pan region by dragging it.
- A zoomed detail of the image around the mouse position.

Click and drag the mouse pointer on the gradient bar below the image to change intensity levels.

Zooming. You can change the zoom level in the following ways:

- Use your mouse scroll wheel while the mouse pointer is on the image.
- Right click on the image and choose *Zoom* → *Zoom in* or *Zoom* → *Zoom out* . Choose *Zoom* → *Zoom to Fit* to fit the entire image into the viewer.

- Use the four icons at the bottom left corner of the viewer:
  - Zoom in
  - Zoom out
  - Zoom to fit window
  - Zoom to original size
- Write a custom zoom level in the text field next to the zoom icons and press **Enter** .

Flipping the direction of the Y axis. Click the double-arrow icon in the toolbar at the bottom of the viewer, next to the zoom icons. Alternatively, right click on the image and choose *Axes → Flip Y Axis* .

Viewing image coordinates and intensity values. The three boxes below the image show the following information (left to right):

- Pixel coordinates at mouse pointer, listed as (*y, x*).
- Pixel intensity value at mouse pointer.
- WCS coordinates (if defined) at mouse pointer.

To print the X/Y and RA/Dec coordinates at the mouse pointer position to the *Console* view, right click on the image and choose *Get coordinates* .

Showing axes with coordinates. You can show axes with image coordinates at the top, bottom, left and right of the image. Right click on the image and choose, for instance, *Axes → Right Axis → Enable* . For enabled axes you can use the same submenu to display a label, change units and customise tick marks.

Advanced image visualisation: We recommend that you use SAOImage DS9 to perform advanced display techniques involving several images. The following instructions are for DS9, **not for HIPE** , and include the steps for:

- Tiling or blinking several images.
- Matching WCS coordinates.

This tool is part of the Virtual Observatory, thus allowing easy sharing of data with HIPE. Once downloaded (please open this link in a new tab or window) and installed, you can send images to it following the guidelines described in Section 4.3.2 and Section 1.17 .

- To *tile images* you just need to send images to DS9 and the software, by default, will arrange them into a grid of same sized tiles. The number of rows and columns in this grid will depend on the number of images.
- To *blink images* it is better to first match their coordinates first (see below). You should click *Frame → Blink Frames* to change from the default tiling view to blinking.
- *Matching WCS coordinates* is required to visually compare images taken with different instruments, with different resolution and/or pixel size. To match the WCS coordinates (so the same sky positions are overlapping for all images) you should click *Frame → Match → Frame → WCS* . Now all the images show the same sky region independently of their spatial resolution.

4.5. Measuring angular distances

To measure the angular distance between two points on an image with a valid WCS, follow these steps:

1. Place the mouse pointer on the first point.
2. Press and hold the **Shift** key.

3. Press and hold the left mouse button.

4. Drag the mouse pointer to the second point. A triangle appears on the image, showing the distance between the two points and the components in Right Ascension and Declination. Distances are shown in arcminutes and arcseconds in the format *mm:ss.SS* in Java and *%M:%S.%f* in Jython.

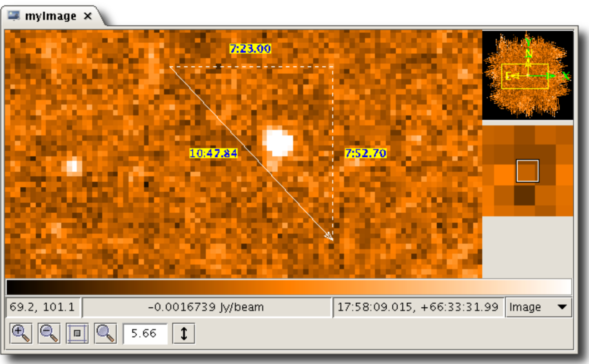


Figure 4.3. Measuring angular distances on an image.

Tip

While dragging the mouse pointer with the **Shift** and left mouse button pressed, press and hold the **Ctrl** key as well to hide the dashed lines showing the RA and Dec distance components.

4.6. Creating masks

To create a mask covering part of the image, follow these steps:

1. Right click on the image and choose *SkyMask toolbox* from the menu.

   The SkyMask toolbox opens.

2. Click on a shape button (rectangle, ellipse or polygon). Click and drag the mouse pointer on the image to draw a rectangle or an ellipse. To draw a polygon, click on the image to draw a vertex, and double click to draw the last vertex.

3. Click the scissors icon. HIPE creates a `skyMask` variable, describing a mask corresponding to the shape you drew.

Resizing shapes. To resize a shape, click inside it so that it becomes selected and blue handles appear. Click and drag a handle to resize the shape.

Moving shapes. To move a shape, click inside it and drag it to its new position.

Deleting shapes. To delete a shape, click inside it so that it becomes selected, and click the bin icon. To delete all shapes, click the red X icon.

You can use sky masks created in this way to limit source extraction to specific areas of the image. For more information see Section 4.19 .

4.7. Viewing metadata and array data associated to an image

An image can have several datasets, like a flag image dataset for flagging bad pixels (see Section 4.12 for more information). Each of these datasets has associated metadata, which has the same role as header information in a FITS file. It indicates associated flux and coordinates, plus other information such as processing history.

You have two ways to view the metadata and array data associated with an image:

- Right click on your image variable name in the *Variables* view, and choose *Product Viewer* from the *Open With* menu. The Product Viewer shows image metadata, plus all the array datasets associated with the image in the *Data* pane. For more information on the Product Viewer see the *HIPE Owner's Guide* : Section 15.1 in *HIPE Owner's Guide* .
- To display information on a single dataset in the image, right click on it either in the *Outline* view (as shown in Figure 4.4) or in the Product Viewer, and choose *Open With* → *Dataset Viewer* . The Dataset Viewer is similar to the Product Viewer, in that it shows metadata and data associated with the dataset.

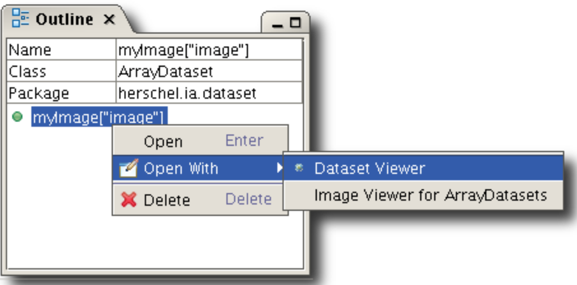


Figure 4.4. Opening the Dataset Viewer from the Outline view.

4.8. Saving an image

To save an image to file, right click on it and choose *Create screenshot* . You can choose to save the whole image or just the current view. In both cases, annotations are saved as well. You can save to four formats: JPG, PNG, PDF and PS.

For information on how to save an image to FITS, see Section 4.3.2 .

From the command line.

To save an image to file via the command line you first have to create a `Display` object, as shown by the following example:

```
d = Display(myImage)  # myImage is a SimpleImage
d.saveAsJPG("/path/to/file.jpg")  # Save whole image
d.saveCurrentViewAsJPG("/path/to/file.jpg")  # Save current view
```

Example 4.9. Saving the current view of a Display object.

The other available methods are `saveAsEPS` , `saveAsPNG` and `saveAsPDF` , and analogously `saveCurrentViewAsEPS` , `saveCurrentViewAsPNG` and `saveCurrentViewAsPDF` .

You also have the more general `save` and `saveCurrentView` methods:

- `save(" /path/to/file.jpg ")` chooses the file format based on the file extension.
- `saveCurrentView(" /path/to/file.jpg ")` is the same as `save` , but for the current view.
- `save()` shows the save dialogue window you obtain by right clicking on the image and choosing *Create screenshot* .

Note

While the methods available in the `Display` class are very useful, they block the GUI while writing to disk. This is not acceptable for scripts that save a batch of images to disk.

In cases like that, you should use the Java ImageIO package provided with the JDK which doesn't block the user interface.

```
from herschel.ia.gui.image import ImageUtil
from javax.imageio import ImageIO
from java.io import File

# Creating an image composed of random data
myImage = SimpleImage()
myImage.image=RESHAPE(Double1d.range(256*256), [256,256])

tiledImage = ImageUtil().getTiledImage(myImage)   #myImage is a
 SimpleImage
ImageIO.write(tiledImage, "png", File("image.png"))
```

Example 4.10. Saving an image to disk without blocking the GUI.

4.9. `SimpleImage` editing

This section describes simple tools to change the colours and the cut levels of an image, and to add drawings and annotations.

Note

The tools and editing techniques described in this section only apply to instances of the grayscale class `SimpleImage` and not to RGB images like `RgbSimpleImage` or any other type of image.

Editing the image colours

Right click on the image and choose *Edit colors* from the context menu to display the dialogue window in Figure 4.5 . This window allows you to change the colour map, the intensity profile and the scale algorithm. All changes are immediately reflected on the image. Click Reset to return to the default scheme (*Real* colour map, *Ramp* intensity and *Linear Scale* algorithm).

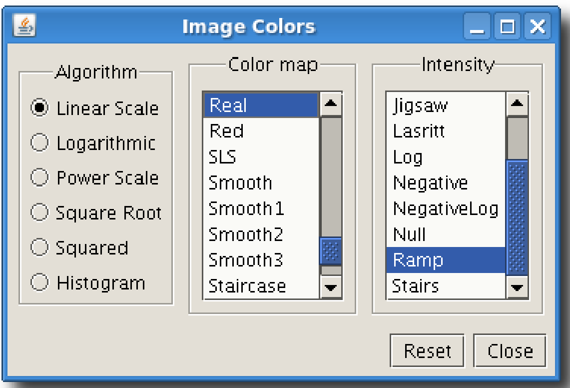


Figure 4.5. Colour map window.

Editing the cut levels

Right click on the image and choose *Edit cut levels* from the context menu to display the dialogue window in Figure 4.6 .

You can edit the cut levels in three ways:

- Click and drag the yellow arrows shown at either end of the histogram view to change the upper and lower level cutoffs.
- Enter the level values in the two text boxes.

- Click one of the *Auto Set* buttons

All changes are immediately reflected on the image and on the histogram plot. Click Reset to return to the default cut level of 99.5% of pixel values.

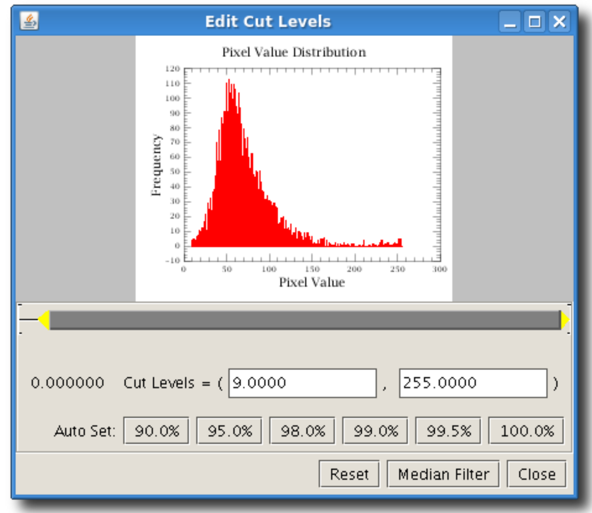


Figure 4.6. Cut level selection window.

Annotating an image

Use the annotation toolbox to add lines, shapes and text to an image. The annotation toolbox is shown in Figure 4.7 .

To open the annotation toolbox, right click on an image and choose *Annotations* → *Toolbox* .

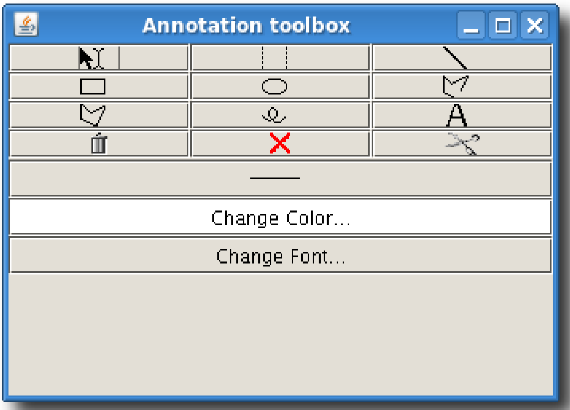


Figure 4.7. The annotation toolbox.

Note

You can use the annotation toolbox only on the image you opened it from. You cannot use it on other images you have open in HIPE.

The buttons in the annotation toolbox in Figure 4.7 have the following usage, from left to right and from top to bottom:

- Select annotation.
- Select all annotations in a region.
- Draw a line, a rectangle, an ellipse, a polyline or a polygon.

- Draw a freehand line.
- Add a text annotation.
- Remove the selected annotations.
- Remove all annotations.
- Useful only if you have one or more *closed* shapes drawn on the image. Returns the following two variables:
  - A variable called `flag` , of type `Flag` . The variable is basically a matrix of integers, with one value for each pixel of the image. Values corresponding to pixels inside the shapes are set to 1. The others are set to 0. For more information on the `Flag` variable type, see the *User's Reference Manual* : Section 1.154 in *HCSS User's Reference Manual* .
  - A variable called `bool2dMask` of type `Bool2d` . Values corresponding to pixels inside the shapes are set to `false` . The others are set to `true` .

The polygon and polyline methods enable you to select points on the image which should be used as a corner of the polygon using the mouse. Double-clicking the mouse ends the selection procedure.

The three buttons below the ones already described change the view of the annotation. From top to bottom:

- Change the thickness of the line.
- Change the colour of the annotation. The current colour of annotations corresponds to the background colour of the button.
- Change the font of text annotations.

Note

The *Select all annotations in a region* button only works when there are already annotations on the image. Pressing the button will select all the annotations which are in the selected region. This button can be used to change the colour or the line width of several annotations at once.

Displaying a scale or a compass. Right click on the image and choose *Annotations → Add scale* or *Annotations → Add compass* . The compass shows the North direction.

On the command line

You can draw figures and put text annotations on an image with the following functions of `Display` :

- *Regular text annotations* , using the `addAnnotation` , `setAnnotationFont` and `setAnnotationFontColor` methods.
- *Greek text annotations* , using the `addGreekAnnotation` , `setAnnotationFont` and `setAnnotationFontColor` methods. The `addGreekAnnotation` method converts normal characters to Greek characters ('a' becomes 'alpha', 'b' becomes 'beta' and so on).
- *Lines and shapes* , using the `addEllipse` , `addLine` , `addPolygon` , `addPolyline` and `addRectangle` methods. The `addPolygon` and `addPolyline` methods need an array of doubles as parameter. In such an array, the coordinates should be added as polygon(([x1, y1, x2, y2,...]),...).

  The sizes of these shapes are measured in *image* pixels, not in *screen* pixels. This means that shapes created with the same numbers will have the same size relative to the image, irrespective of the zoom level.

The following example shows how you can add shapes and text annotations to an image from the command line. The resulting image is shown after the example.

```
# Imports
from java.awt import Font
from java.awt import Color

myDisplay = Display(myImage)

# Placing a text annotation at position (321, 224)
myDisplay.addAnnotation("Veil nebula", 321, 224)
# Changing the font type and size of the annotations
myDisplay.setAnnotationFont(321, 224, Font("Dialog", 0, 32))
# Changing the annotation colour
myDisplay.setAnnotationFontColor(321, 224, Color(0,0,255))
# Adding an ellipse with center at (268.5,500.0), width = 38 and height = 37,
# linewidth = 3.0 and black colour
myDisplay.addEllipse(268.5, 500.0, 38.0, 37.0, 3.0, Color.green)
# Adding a Greek text annotation at position (100,500)
myDisplay.addGreekAnnotation("a = 12.34, d = +30.30", 100, 500)
# Changing the font and colour of the annotation
myDisplay.setAnnotationFont(100, 500, Font("Dialog", 0, 20))
myDisplay.setAnnotationFontColor(100, 500, Color(0,0,0))
# But white is more visible
myDisplay.setAnnotationFontColor(100, 500, Color.white)
```

Example 4.11. Adding annotations and setting formatting options in a Display object.

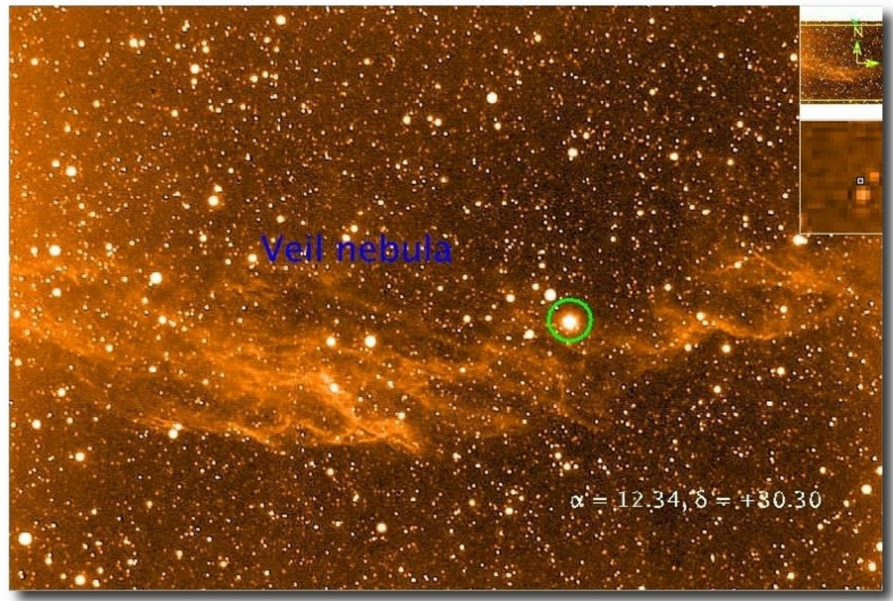


Figure 4.8. Adding annotations to a Display.

Tip

There is no option to draw filled shapes, but you can easily simulate a filled shape by setting a suitable line width. For example, this command draws a circle with a line thickness twice as large as the radius:

```
radius = 10
myDisplay.addCircle(50.0, 50.0, radius, radius*2,
 java.awt.Color.BLACK)
```

Example 4.12. Filling shapes with oversized lines in a Display.

You can use the same trick to fill other shapes such as ellipses, rectangles and so on.

You can open the annotation toolbox via the command line as follows:

```
myDisplay.annotationToolbox()
```

Example 4.13. Opening the annotation toolbox programmatically.

You can open the dialogue windows for editing colours and cut levels in the same way as the annotation toolbox:

```
myDisplay.editColors()
myDisplay.editCutLevels()
```

Example 4.14. Opening the colour and cut level dialogues programmatically.

For more information see the entry for *Display in HCSS User's Reference Manual* in the *User's Reference Manual* . There you will find other useful methods, such as `addCompass` , to add a compass with north and east directions, and `addTenArcSecs` , to add a line ten arcseconds long.

Obtaining code for actions in the graphical interface. You can obtain the Jython code corresponding to your actions with the annotation toolbox with these two buttons at the bottom of the toolbox window:

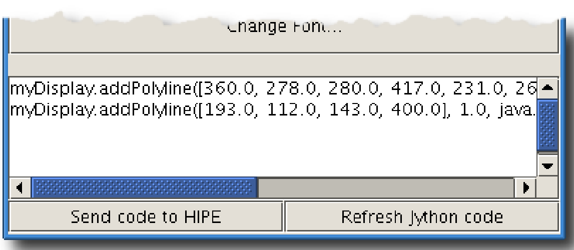


Figure 4.9. Jython code appearing in the annotation toolbox.

The code does not appear automatically as you work with the annotation toolbox. You must press the *Refresh Jython code* button to see the code corresponding to your actions. Press *Send code to HIPE* to paste the code to the *Console* view.

Note

If you change the size of a text annotation, this will not be reflected in the Jython code.

4.10. Manipulating the axes (cropping, rotating, scaling...)

See Section 4.2 for general information on how to run image analysis tasks.

With the following tasks you can perform basic transformations on your images. The links take you to the corresponding entries in the *User's Reference Manual* . All these tasks output another image.

- **Clamp in *HCSS User's Reference Manual* :** also known as *clipping* , eliminates pixel intensity values outside a given range. Intensities below the lower limit are set to that limit, and the same happens with intensities above the upper limit.

```
clamped = clamp(image = myImage, low = 11, high = 240)
```

Example 4.15. Clamping an image with explicit limits.

- **Crop in *HCSS User's Reference Manual* :** reduces the size of an image by cutting portions outside a rectangular area, defined by two boundary rows and columns. The WCS is adapted to have the same sky coordinates for the same position in the original and cropped image.

```
cropped = crop(image = myImage, row1 = 11, row2 = 55, column1 = 240, column2 =
 300)
```

Example 4.16. Cropping an image specifying the rectangle dimensions.

- **Regrid in *HCSS User's Reference Manual* :** takes a `source` image and changes its spatial resolution (grid) according to a `target` image **or** a `wcs` object created for this purpose. Output flux is proportional to the input flux by the following ratio `output_cdelt1 * output_cdelt2/input_cdelt1 * input_cdelt2` (as opposed to other tasks in this section like `scale`, whose output flux for each pixel depends on the flux of neighbouring pixels of the original image). You can retrieve the proportional factor from the output parameters of the task using the method `getFluxConservation`.

```
regridded = RegridTask(source = srcImage, target = refImage)
# Retrieving the flux conservation factor
fluxFactor = regridded.getFluxConservation()
```

Example 4.17. Regridding an image and getting the flux change factor.

- **Rotate in *HCSS User's Reference Manual* :** rotates an image by a given angle. Four interpolation methods are available:

  - **Bi-linear:** interpolates one pixel to the right and one below. Default option.
  - **Nearest neighbour:** direct pixel copying, the fastest option.
  - **Bi-cubic:** uses interpolation via a piecewise bi-cubic polynomial. This option needs the subsample precision in bits as an extra parameter. The default value is 16 bits.
  - **Bi-cubic2:** variant of bicubic interpolation that can give sharper results. This option needs the subsample precision in bits as an extra parameter. The default value is 16 bits. This option is relatively slow.

    The following example shows how to invoke the task:

    ```
    rotated = rotate(image = myImage, angle = 12.2)
    ```

    Example 4.18. Rotating an image without specifying an interpolation method.

    When using an interpolation method other than the default you need to specify it:

    ```
    rotated = rotate(image=myImage, angle=12.2,
     interpolation=rotate.INTERP_BICUBIC, subsampleBits=32)
    ```

    Example 4.19. Rotating an image with a specific interpolation method.

- **Scale in *HCSS User's Reference Manual* :** scales an image, allowing for different scaling factors in the X and Y directions. For example, a scale factor of 2 doubles the image size, while a factor of 0.25 reduces it to one quarter of the original. A negative factor also flips the image along the axis. Note that scaling does not conserve flux. The available interpolation types are as for the `rotate` task.

```
# Default interpolation
scaled = scale(image = myImage, x = 1.4, y = 0.4)
```

```
# Custom interpolation
scaled = scale(image = myImage, x = 1.4, y = 0.4, interpolation =
 scale.INTERP_BICUBIC, subsampleBits = 32)
```

Example 4.20. Scaling an image both with and without custom interpolation.

- [Translate](#) **in *HCSS User's Reference Manual* :** translates an image along a given vector in pixel or sky coordinates. Coordinates are given as decimal degrees.

```
# Pixel coordinates
translated = translate(image = myImage, x = 5, y = 7)
# Sky coordinates
translated = translate(image = myImage, ra = 0.03, dec = 0.03)
```

Example 4.21. Translating an image using two different sets of coordinates.

- [Transpose](#) **in *HCSS User's Reference Manual* :** transposes the image in one of the following ways, automatically adapting the WCS:
  - Flip vertically (flips top and bottom) and horizontally (flips from side to side).
  - Flip diagonally (bottom left to top right) and antidiagonally (top left to bottom right).
  - Rotate 90, 180 and 270 degrees (clockwise).

```
flipped = transpose(image = myImage, type = TransposeTask.FLIP_HORIZONTAL)
```

Example 4.22. Transpose an image with a horizontal flip.

Dialogue windows. All the dialogue windows for image transformations work in the same way. You drag your image from the *Variables* view to the task dialogue window and drop it onto the circle next to the *image* parameter. If your image has the right format, the circle turns green and the name of your image appears next to it. To execute the task, click the *Accept* button.

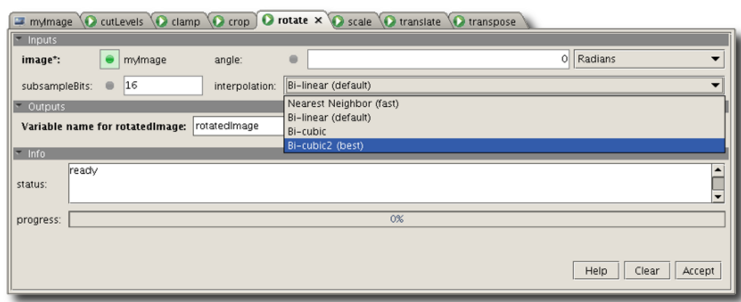


Figure 4.10. Example image transformation dialogue window. Rotating an image using the "rotate" task. Several interpolation options are available.

4.11. Manipulating fluxes

4.11.1. Image arithmetics

See [Section 4.2](#) for general information on how to run image analysis tasks.

Depending on the task you want to apply to images and the particular instrument, the flux density should be in a particular unit. Some recommendations provided by the instrument teams are:

- *SPIRE maps for aperture photometry:* The flux should be in `Jy/pixel` .

- *SPIRE maps for source extraction/detection or gaussian fitting:* The flux should in `Jy/beam` .
- *PACS maps:* The best unit to use with PACS data is `Jy/pixel` .

The following image arithmetics tasks are available. The links take you to the corresponding entries in the *User's Reference Manual* . All these tasks output another image.

- **imageAdd in *HCSS User's Reference Manual* :** adds a scalar to all the pixels of an image, or adds two images together, either pixel by pixel or based on their WCS. The addition mode is set by the *reference frame* integer parameter (0 for pixel by pixel, 1 for WCS). If the two images are to be added based on their WCS, they are regridded onto the spatial grid of the smaller image. If the two images have the same unit, the sum uses the same unit, otherwise the calculation is done as if the units were in counts.

```
# Adding another image
sum = imageAdd(image1 = myImage1, image2 = myImage2, ref = 0)
# Adding a scalar
sum = imageAdd(image1 = myImage, scalar = 5.0)
```

Example 4.23. Adding images and also scalars to images.

- **imageSubtract in *HCSS User's Reference Manual* :** subtracts a scalar from all the pixels of an image, or subtracts an image from another one, either pixel by pixel or based on their WCS. The observations made for the `imageAdd` task also apply to `imageSubtract`.

```
# Subtracting another image
difference = imageSubtract(image1 = myImage1, image2 = myImage2, ref = 0)
# Subtracting a scalar
difference = imageSubtract(image1 = myImage, scalar = 5.0)
```

Example 4.24. Subtracting images and also scalars from images.

- **imageMultiply in *HCSS User's Reference Manual* :** multiplies all the pixels of an image by a scalar, or multiplies an image by another one, either pixel by pixel or based on their WCS. The observations made for the `imageAdd` task also apply to `imageMultiply` . When multiplying two images, the unit of the result is the composite of the units of the original images.

```
# Multiplying by another image
product = imageMultiply(image1 = myImage1, image2 = myImage2, ref = 0)
# Multiplying by a scalar
product = imageMultiply(image1 = myImage1, scalar = 5.0)
```

Example 4.25. Multiplying images and also images by scalars.

You can combine `imageMultiply` and `regrid` to change the sampling of an image and then to ensure flux conservation of the output image. See below for an example on how to change the pixel size to 4.5 arcseconds if the original image flux density is in `MJy/sr` :

```
oldWcs = mapHiRes.wcs
newWcs = mapHiRes.wcs.copy()
newWcs.setCdelt1(-4.5 / 3600.) # the - conserves the standard orientation
newWcs.setCdelt2(4.5 / 3600.)
mapLowRes = imageMultiply(image1=regrid(mapHiRes, wcs=newWcs), \
                          scalar=(oldWcs.getCdelt2()/newWcs.getCdelt2())**2)
```

Example 4.26. Changing the pixel size of an image while ensuring flux conservation.

- **imageDivide in *HCSS User's Reference Manual* :** divides all the pixels of an image by a scalar, or divides an image by another one, either pixel by pixel or based on their WCS. The observations made for the `imageMultiply` task also apply to `imageDivide`.

```
# Dividing by another image
quotient = imageDivide(image1 = myImage1, image2 = myImage2, ref = 0)
# Dividing by a scalar
quotient = imageDivide(image1 = myImage1, scalar = 5.0)
```

Example 4.27. Dividing images and also images by scalars.

- **imageModulo in *HCSS User's Reference Manual* :** calculates the remainder of a division between an image and a scalar, or two images, divided either pixel by pixel or based on their WCS. The observations made for the `imageMultiply` task also apply to `imageModulo`.

```
# Dividing by another image
remainder = imageModulo(image1 = myImage1, image2 = myImage2, ref = 0)
# Dividing by a scalar
remainder = imageModulo(image1 = myImage1, scalar = 5.0)
```

Example 4.28. Integer division of images and image by scalar.

- **imageAbs in *HCSS User's Reference Manual* :** computes the absolute value of all intensity values of an image.

```
absolute = imageAbs(image = myImage)
```

Example 4.29. Applying the absolute value to the intensity values of an image.

- **imageRound in *HCSS User's Reference Manual* :** rounds all intensity values of an image to the nearest integer.

```
rounded = imageRound(image = myImage)
```

Example 4.30. Rounding the intensity values of an image.

- **imageFloor in *HCSS User's Reference Manual* :** rounds all intensity values of an image to the largest previous integer.

```
floored = imageFloor(image = myImage)
```

Example 4.31. Rounding the intensity values of an image to the largest previous integer.

- **imageCeil in *HCSS User's Reference Manual* :** rounds all intensity values of an image to the smallest following integer.

```
floored = imageFloor(image = myImage)
```

Example 4.32. Rouding the intensity values of an image to the smallest following integer.

- **imagePower in *HCSS User's Reference Manual* :** raises all intensity values of an image to the given power. The power value does not have to be an integer.

```
powered = imagePower(image = myImage, n = 2.0)
```

Example 4.33. Raising the intensity values of an image to the n-th power.

- **imageSquare in *HCSS User's Reference Manual* :** computes the square of all intensity values of an image. The same as running `imagePower` with power value 2.0.

```
square = imageSquare(image = myImage)
```

Example 4.34. Squaring the intensity values with an specific task.

- **imageSqrt in *HCSS User's Reference Manual* :** computes the square root of all intensity values of an image. The same as running `imagePower` with power value 0.5.

```
sqrt = imageSqrt(image = myImage)
```

Example 4.35. Taking the square root of the intensity values of an image.

- **imageLog in *HCSS User's Reference Manual* :** computes the natural logarithm of all intensity values of an image.

```
log = imageLog(image = myImage)
```

Example 4.36. How to obtain the natural logarithm of the intensity values of an image.

- **imageLog10 in *HCSS User's Reference Manual* :** computes the base 10 logarithm of all intensity values of an image.

```
log10 = imageLog10(image = myImage)
```

Example 4.37. How to obtain the base 10 logarithm of the intensity values of an image.

- **imageLogN in *HCSS User's Reference Manual* :** computes the base N logarithm of all intensity values of an image. The base N can be any positive real number. Negative values for N do not give an error, but produce an output image with `NaN` as intensity value for all pixels.

```
logN = imageLogN(image = myImage)
```

Example 4.38. How to obtain the base N logarithm of all intensity values of an image.

- **imageExp in *HCSS User's Reference Manual* :** computes the exponential function of all intensity values of an image.

```
exp = imageExp(image = myImage)
```

Example 4.39. How to obtain the exponential of the intensity values of an image.

- **imageExp10 in *HCSS User's Reference Manual* :** replaces all intensity values of an image with 10 raised to the intensity value.

```
exp10 = imageExp10(image = myImage)
```

Example 4.40. Raising 10 to the intensity value for all image intensity values.

- **imageExpN in *HCSS User's Reference Manual* :** replaces all intensity values of an image with N raised to the intensity value.

```
expN = imageExpN(image = myImage, n = 2.0)
```

Example 4.41. Raising N to the intensity value for all image intensity values.

Tip

When using the graphical interface of tasks requiring a second image (such as `imageAdd` and `imageSubtract`) you can add the second image by dragging it from the *Variables* view to the small circle close to the corresponding parameter. If the variable is of the correct type, the circle becomes green.

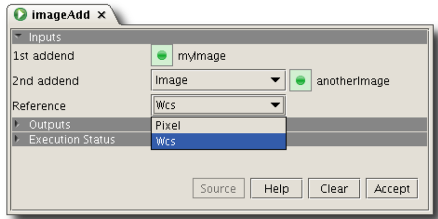


Figure 4.11. Example image arithmetic dialogue window.

4.11.2. Smoothing images

See Section 4.2 for general information on how to run image analysis tasks.

The following smoothing tasks are available in the *Tasks* view: `meanSmoothing` , `medianSmoothing` , `boxCarSmoothing` and `gaussianSmoothing` .

These tasks have a *`width`* parameter, representing the width of the filtering window, boxcar or the FWHM of the Gaussian. This parameter is called *`sigma`* for Gaussian smoothing.

The parameter *`width`* must be an odd positive integer for mean and median smoothing and a positive integer for boxcar smoothing. The parameter `sigma` must be a positive floating point number for Gaussian smoothing.

The four tasks share the similar command line syntax:

```
# Mean smoothing
smoothedMean = meanSmoothing(image = myImage, width = 3)

# Median smoothing
smoothedMedian = medianSmoothing(image = myImage, width = 3)

# Boxcar smoothing
boxCarSmoothed = boxCarSmoothing(image = myImage, width = 4)

# Gaussian smoothing
gaussianSmoothed = gaussianSmoothing(image = myImage, sigma = 2.5)
```

Example 4.42. Smoothing an image using four different algorithms.

All these tasks have a `SimpleImage` as output, with the same settings (WCS, errors, flag, exposure) as the input image.

4.11.3. Converting image units

To convert the surface brightness unit of your image to some other surface brightness unit, use the `convertImageUnit` task. The task takes the image and the new unit as input parameters and produces a converted image. For some conversions you might need to provide the beam area as an optional parameter. The following example shows how to do the conversion: assuming that the beam area is modelled by a 2D Gaussian, then the area follows equation `2*Math.PI*sigma^2` with sigma being `sigma = FWHM/SQRT(8*LOG(2.0))`. The beam area is measured in square arcseconds. Note that, in HIPE, `LOG` in *HCSS User's Reference Manual* is the natural logarithm:

```
FWHM = 18.1
myBeamArea = (Math.PI * (FWHM**2))/(4 * LOG(2.0))
converted = convertImageUnit(image = myImage, newUnit = "Jy/beam", beamArea
 = myBeamArea)
```

Example 4.43. Converting image units with a specific task.

4.11.4. Convolving images

There is often a need to bring images to a common beam size so that data taken at different wavelengths can be compared to each other. For this you need to convolve one image with the PSF of the other image. You can do so in HIPE with the `imageConvolution` task. The tasks takes an image and a convolution kernel, in the form of another image, as input parameters, and provides the convolved image as output. The following example shows how to invoke the task:

```
convolution = imageConvolution(image = myImage, kernel = myPsf)
```

Example 4.44. Convolving an image with a specific kernel.

4.12. Flagging saturated pixels

See Section 4.2 for general information on how to run image analysis tasks.

With the `flagSaturatedPixels` task you can set a cutoff value above which pixels are considered saturated.

The output is another image, called `flaggedImage` by default. It looks like a copy of the input image, except that pixels whose value lies above the cutoff value are flagged with the `SATURATED` flag type. The pixel values are not changed.

You can flag pixels via the command line as shown in the following example, with the *`value`* parameter giving the cutoff value:

```
flagged = flagSaturatedPixels(image = myImage, value = 100.0)
```

Example 4.45. Flagging pixels whose value exceeds the limit provided.

For more information see the User's Reference Manual: *FlagSaturatedPixelTask* in *HCSS User's Reference Manual*

4.13. Getting cut levels

See Section 4.2 for general information on how to run image analysis tasks.

With the `cutLevels` task you can determine the cut levels of an image, using the percentage method or applying a median filter.

In the task dialogue window, set the *Method* parameter to the method you want to use to determine the cut levels. If you select *Percent* , you can change the default percentage value in the *Percent* field.

The result is an array with two elements, the low and high cut. You will find it in the *Variables* view. Use the **print** command in the *Console* view to show the contents of the variable.

For information on determining cut levels via the command line, see the `cutLevels` task entry in the User's Reference Manual: *CutLevelsTask* in *HCSS User's Reference Manual* or check the following example.

```
# Creating an image composed of random data
```

```
i = SimpleImage()
i.image=RESHAPE(Double1d.range(256*256), [256,256])

levels = cutLevels(i, CutLevelsTask.PERCENT, 95.0)
print "Minimum value:"+str(levels[0])
print "Maximum value:"+str(levels[1])
```

Example 4.46. Getting the minimum and maximum values for a certain cut-off percentage.

4.14. Combining images (stitching, RGB)

4.14.1. Stitching

Use the `mosaic` task to create a mosaic of images. The input parameters are the following:

- A list with images you want to combine.
- Whether to do oversampling (optional, `True` by default).
- A WCS for the output mosaic (optional).

The following example shows how to combine `n` images, from `image_1` to `image_n`, into a mosaic:

```
# Import
from java.util import ArrayList

# Making an ArrayList with the images
images = ArrayList()
images.add(image_1)
...
images.add(image_n)

# Making an oversampled mosaic
mosaicOversampled1 = mosaic(images = images, oversample = 1)
mosaicOversampled2 = mosaic(images = images)

# Making a non-oversampled mosaic
mosaicNonOversampled = mosaic(images = images, oversample = 0)
```

Example 4.47. Mosaicking images with the help of an intermediate array.

The result, `mosaic`, is of type `SimpleImage`.

For more information about the parameters and usage of the `mosaic` task, see the *User's Reference Manual* : *MosaicTask* in *HCSS User's Reference Manual* .

4.14.2. Creating RGB images

You can create RGB images with the `createRgbImage` task. To run this task you need three images of type `SimpleImage` that you wish to combine. You have to define either the cut percentage or the scaling factors by which the images should be multiplied.

```
rgb = createRgbImage(red = myRedImage, green = myGreenImage, blue = myBlueImage, \
percent = 98.0, redFactor = 1.3, greenFactor = 1.0, blueFactor = 1.6)
```

Example 4.48. Creating an RGB image with specific weights for each channel.

You can also specify the cut levels for each image. In case you define the cut levels, you have to set the scaling factors to 1.0.

```
rgb = createRgbImage(red = myRedImage, green = myGreenImage, blue = myBlueImage, \
```

```
lowBlue = 0.0, highBlue = 50.0, lowGreen = 60.0, highGreen = 120.0, lowRed = 12.0,
 highRed = 160.0)
```

Example 4.49. Creating an RGB image with cut levels for each channel.

In addition you can define a WCS for the output image. See Section 4.15 for information on how to define the WCS.

```
rgb = createRgbImage(red = myRedImage, green = myGreenImage, blue = myBlueImage, \
percent = 98.0, redFactor = 1.3, greenFactor = 1.0, blueFactor = 1.6, wcs = myWcs)
```

Example 4.50. Creating an RGB image with weights, overall cut level and new WCS information.

The output of this task is an image of type `RgbSimpleImage` .

This is the dialogue window shown when running the task in graphical mode:

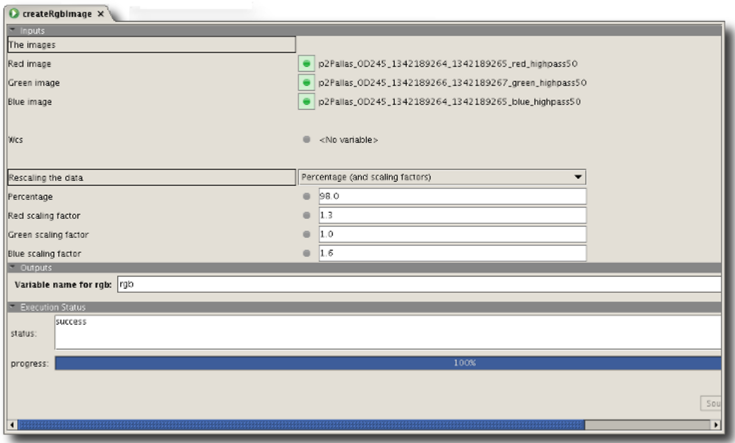


Figure 4.12. The createRgbImage task dialogue window.

For more information about the parameters and command line usage of the `createRgbImage` task, see the *User's Reference Manual* : CreateRgbImage in *HCSS User's Reference Manual* .

4.15. Defining and using the World Coordinates System (WCS)

The WCS information for an image is stored in its metadata. You can access it by using the following command:

```
print myImage.wcs
```

Example 4.51. Printing the WCS information of an image.

This results in the following output:

```
World Coordinate System
-----------------------
cunit1: Degrees
cunit2: Degrees
cdelt1: -2.777777777778E-4
cdelt2: 2.777777777777778E-4
crota2: 0.0
```

```
ctype1: RA---TAN
ctype2: DEC--TAN
naxis1: 668
naxis2: 764
crpix1: 334.0
crpix2: 382.0
crval1: 76.03702441240077
crval2: 32.7159459572391
flipy: FLIPY
equinox: 2000.0
```

You can also use the WCS Explorer by right clicking on the image in the *Variables* view and choosing *Open With* → *WCS explorer for Images* .

The following illustrates how you can create a WCS and add it to a `SimpleImage` .

```
i = SimpleImage()
i.image=RESHAPE(Double1d.range(200*300), [200,300])
# Create a fake image 200x300 pixels in size

myWcs = Wcs() # Set up the Wcs() object
myWcs.ctype1 = "LINEAR"  # Start adding things to it...
myWcs.cdelt1 = 5
myWcs.crval1 = 200
myWcs.cunit1 = "K"
myWcs.crpix1 = 0

myWcs.ctype2 = "LINEAR"
myWcs.cdelt2 = .05
myWcs.crval2 = 2.0
myWcs.cunit2 = "V"
myWcs.crpix2 = 0

i.wcs = myWcs  # Apply the set of WCS information to our image
print i.wcs # To see the WCS of the image
```

Example 4.52. Creating WCS coordinate data.

Warning

The above code generates an image with the value 200 assigned to the `NAXIS2` keyword and 300 assigned to `NAXIS1` . In other words, the image size will be 200 pixels along the *y* axis and 300 pixels along the *x* axis. The coordinate values will be displayed in the order (*y* , *x*) in the Image Viewer. For an explanation of why the *y* size comes *before* the *x* size, see the *Scripting Guide* : Array ordering in *Scripting Guide* .

The above example creates a coordinate system, with temperature and current as axes. The x-axis is LINEAR (ctype1), has the central pixel in column 0 (crpix1), has a value of 200 in the central pixel (crval1), uses steps of 5 (cdelt1) and has as unit Kelvin. The y-axis is also LINEAR (ctype2), has the central pixel in row 0 (crpix2, this is the top of the image), has a value of 2 in the central pixel (crval2), uses steps of 0.05 (cdelt2) and has as unit Volts.

Note

Rows and columns start counting from (0,0), pixels from (1,1).

Defining transformations between pixel coordinates and sky coordinates . You can do so with the `Wcs` class, using the standard WCS parameters. An example is given below. It also indicates how you can set WCS values in your WCS object:

```
wcs2 = Wcs()  # Creating a WCS.
wcs2.setCrpix1(128)
wcs2.setCrpix2(128)  # The central pixel, in this case at (128, 128).
# Setting the position of the central pixel. In this case, the
# central pixel is located at 6h46'42.387" and 0 degrees 49'45.94".
wcs2.setCrval1(101.676612741936)
```

```
wcs2.setCrval2(0.829427624677429)
# Setting the type of the axes. The first axis defines the right
# ascension and the second axis the declination, both in a gnomonic projection.
wcs2.setCtype1("RA---TAN")
wcs2.setCtype2("DEC--TAN")
# Setting the coordinate system (here the standard ICRS type) and the equinox.
wcs2.setRadesys("ICRS")
wcs2.setEquinox(2000.0)
# Creating the linear transformation matrix, which defines
# the pixel size and the rotation of the images.
wcs2.setParameter("cd1_1", -1.9064468150235E-6, "")
wcs2.setParameter("cd1_2", 3.39797311269006E-4, "")
wcs2.setParameter("cd2_1", 3.39811958581193E-4, "")
wcs2.setParameter("cd2_2", 1.580446989748E-6, "")
```

Example 4.53. Creating WCS coordinate data with parameters.

For a more in-depth discussion about creating a WCS, including more options and examples, see the *Scripting Guide* : Section 4.1 in *Scripting Guide* .

4.16. Creating intensity profiles

See Section 4.2 for general information on how to run image analysis tasks.

With the `profile` task you can draw a straight line on an image and plot the intensity along that line. This is useful to check whether there is a gradient in intensity in your image.

Click once on the image to define one end of the line. As you move the mouse, the line is updated and the corresponding profile appears below the image (see Figure 4.13). Click a second time to define the other end of the line. The resulting profile is saved into a dataset in the *Variables* view.

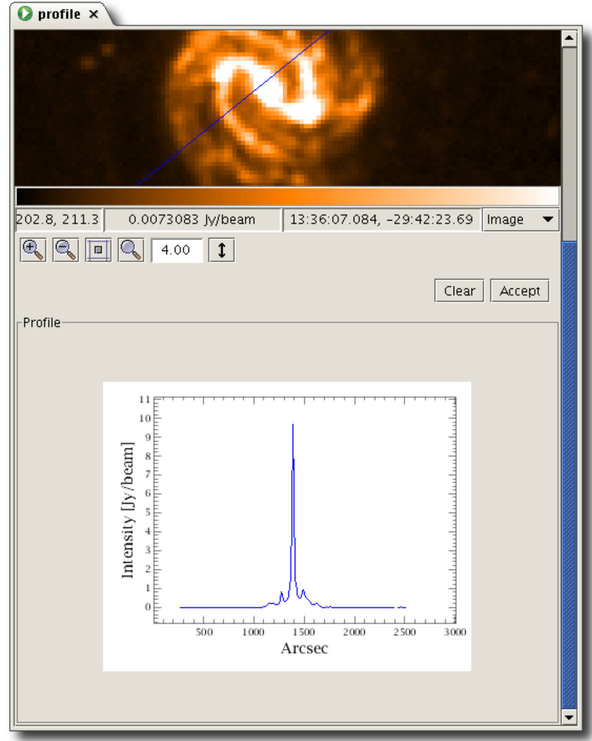


Figure 4.13. The intensity profile below the image.

You can modify the line by clicking on it and dragging the blue handles. Note that, while the plot below the image is updated in real time, the output dataset is not. You have to click the *Accept* button again to obtain a new dataset with the updated result.

Alternatively, you can click *Clear* to delete the line and draw a new one. The output dataset will appear as soon as you define the second end of the line, without having to click *Accept* .

To use the task via the command line you need to pass the following input parameters:

- The image (`image`).
- The beginning and end of the straight line, in pixels (`beginX` , `beginY` , `endX` and `endY`) or in sky coordinates in hexadecimal format as strings (`beginRA` , `beginDec` , `endRA` and `endDec`).

The following example shows the two options available to make a profile plot:

```
profilePixel = profile(image = myImage, beginX = 236.0, beginY = 378.0, \
   endX = 557.0, endY = 232.0)
profileSky = profile(image = myImage, beginRA = "02:00:15.119", \
  beginDec = "-22:24:07.16", endRA = "02:00:38.462", endDec = "-22:26:34.08")
```

Example 4.54. Creating a profile plot of an image.

You can inspect both output products (`profilePixel` and `profileSky`) with the methods listed in the following table:

Procedure 4.1. Available methods for `profilePixel` and `profileSky` .

1. `getBeginPixelCoordinates()`

   Returns a `Double1d` with the pixel coordinates of the beginning of the straight line. Also available a `getEndPixelCoordinates()` method that does the same for the end of the line.

   ```
   # Java style
   print myProfile.getBeginPixelCoordinates()
   # Jython style
   print myProfile.beginPixelCoordinates
   ```

   Example 4.55. Retrieving the pixel coordinates of the beginning of the line.

2. `getBeginSkyCoordinates()`

   Returns a `String1d` with the sky coordinates of the beginning of the straight line. Also available a `getEndSkyCoordinates()` method that does the same for the end of the line.

   ```
   # Java style
   print myProfile.getBeginSkyCoordinates()
   # Jython style
   print myProfile.beginSkyCoordinates
   ```

   Example 4.56. Retrieving the sky coordinates of the beginning of the line.

3. `getProfile()`

   Returns the intensity profile as a `Double1d` .

   ```
   # Java style
   print myProfile.getProfile()
   # Jython style
   print myProfile.profile
   ```

   Example 4.57. Converting the profile plot to a Double1d.

4. `getUnit()`

   Returns the name of the unit in which the intensity profile is expressed.

   ```
   # Java style
   print myProfile.getUnit()
   ```

```
# Jython style
print myProfile.unit
```

Example 4.58. Getting the unit of the profile plot.

For a complete example obtaining the profile plot of a fake data image (with a vertical gradient), see below:

```
# Creating an image composed of random data
i = SimpleImage()
i.image=RESHAPE(Double1d.range(256*256), [256,256])

# Creating a profile plot using pixel coordinates
profilePixel = profile(image = i, beginX = 136.0, beginY = 138.0, \
   endX = 254.0, endY = 232.0)
```

Example 4.59. Creating the pixel profile plot of a synthetic image.

4.17. Creating contour plots

See Section 4.2 for general information on how to run image analysis tasks.

A contour plot connects all image points with the same intensity, like isobars on a weather map.

You can provide a set of contours via three tasks:

- The `automaticContour` task, where you select the number of levels and the lower and upper values, and the intermediate levels are generated automatically with linear or logarithmic intervals of intensity. The parameter controlling the interval distribution is *distribution* . Possible values are 0 (linear), 1 (log) or 2 (ln).

```
from java.awt.Color import GREEN
from java.awt.Color import RED
contours = automaticContour(image = myImage, levels = 2, min = 3.7, max = 4.2, \
distribution = 1, colors = [GREEN, RED])
contours = automaticContour(image = myImage, levels = 2, min = 3.7, max = 4.2, \
distribution = 1)
```

Example 4.60. Creating an automatic line contour specifying distribution and plotting details.

- The `manualContour` task, where you to specify the values of each contour level.

```
from java.awt.Color import GREEN
from java.awt.Color import RED
contours = manualContour(image = myImage, values = Double1d([3.7,4.2]), \
colors = [GREEN, RED])
contours = manualContour(image = myImage, values = Double1d([3.7,4.2]))
```

Example 4.61. Creating a manual line contour providing contour level values.

- The `contour` task, where you specify a single contour level.

```
from java.awt.Color import GREEN
contours = contour(image = myImage, value = 3.7, color = GREEN)
contours = contour(image = myImage, value = 3.7)
```

Example 4.62. Creating a contour containing one single contour level.

This is a complete example where the task `automaticContour` (described above) is used to generate the contours and then, with the help of the `Display` class, plot them over the image.

```
from java.awt import Color

# Get a public SPIRE observation
myobs=getObservation(1342183475L, useHsa=True)
```

```
# Extract the PSW (250 um band) map
map=myobs.level2.refs["psrcPSW"].product

# Display the map
d = Display(map)

# Generate the contours
contours = automaticContour(image=map,levels=4,min=0.05,max=0.2,distribution=0)

# Plot the contours on top of the map (contour default colour is blue)
d.addImageContour(contours)
```

Example 4.63. Plotting automatically-generated contours on top of an image.

To run the task via its graphical interface, click the name of the target image in the *Variables* view. Then double click on `automaticContour` , `manualContour` or `contour` in the *Tasks* list. The corresponding dialogue window appears in the *Editor* view, as shown in the following image.

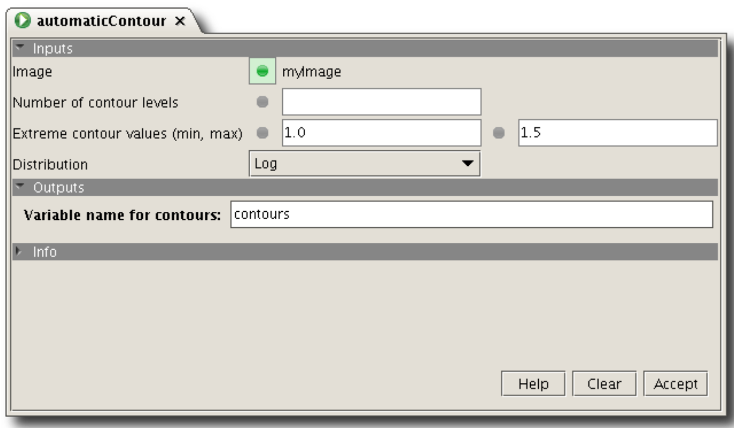


Figure 4.14. Dialogue window for `automaticContour` .

With `manualContour` , enter a contour value and press *Add* to add it to the list. Remove the last selected value or the whole list by clicking *Remove* or *Clear* , then press *Accept* .

Tip

All contours produced via the task graphical interface have the same colour. To change their colour, run the task via the command line instead.

Plotting the contour over an image. First open the image in a display, and then drag and drop the contours onto the image.

If the contours are calculated for an image with a valid WCS and dragged onto an image with a valid WCS, the plot is based on the sky coordinates. In all other cases, the pixel coordinates are used.

Deleting contours from an image. Follow these steps:

1. Right click on the image and choose *Annotations* → *Toolbox*

   The annotation toolbox opens.

2. Click the red cross icon.

   The contours and any other annotations disappear.

Tip

When displayed on an image, contour levels are like any other annotation. You can select, resize and move them.

For more information on creating contour plots via the command line, see the following entries in the *User's Reference Manual* :

- ContourTask in *HCSS User's Reference Manual*
- ManualContourTask in *HCSS User's Reference Manual*
- AutomaticContourTask in *HCSS User's Reference Manual*

4.18. Creating histograms

See Section 4.2 for general information on how to run image analysis tasks.

You can make a histogram of the values in a whole image or of a region bounded by a circle, ellipse, rectangle or polygon, with the `imageHistogram` , `circleHistogram` , `ellipseHistogram` , `rectangleHistogram` and `polygonHistogram` tasks.

With the exception of `imageHistogram` , which computes the histogram for the entire image, a new copy of the image appears in the *Editor* view when you open the task dialogue window. From this you can select the region inside which to take the histogram.

In the task dialogue window, click and drag the mouse pointer over the image to draw the region. In the case of `polygonHistogram` a single click adds a vertex, and a double click adds the final vertex.

Once you have created the region, you can move and resize it. To move the region, click and drag it. To resize the region, click once inside it, then drag the blue resize handles.

In the graphical interface, below the image you can enter the cut levels and number of bins for the histogram (see Figure 4.15). Once you click *Accept* , the following happens:

- The histogram appears in the same window. Scroll down to see it.
- The equivalent command appears in the *Console* view.
- The histogram values are placed in a dataset that appears in the *Variables* list. Double click the variable name to show more information (see Figure 4.16).

Note that changing the area *after* running the task will modify the histogram shown in the task window, but *not* the one in the dataset. You have to click on *Accept* again to produce a new dataset with the updated result.

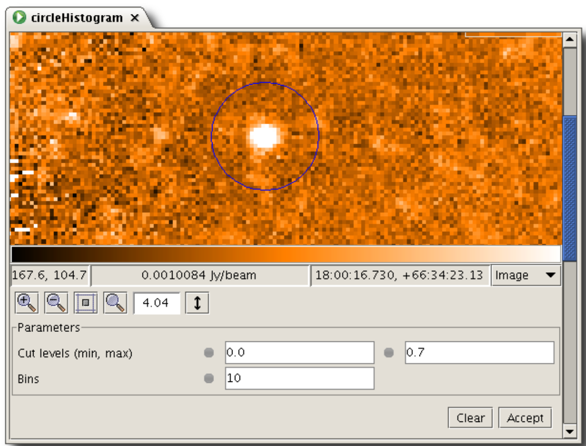


Figure 4.15. Circle histogram area selection and parameter selection.

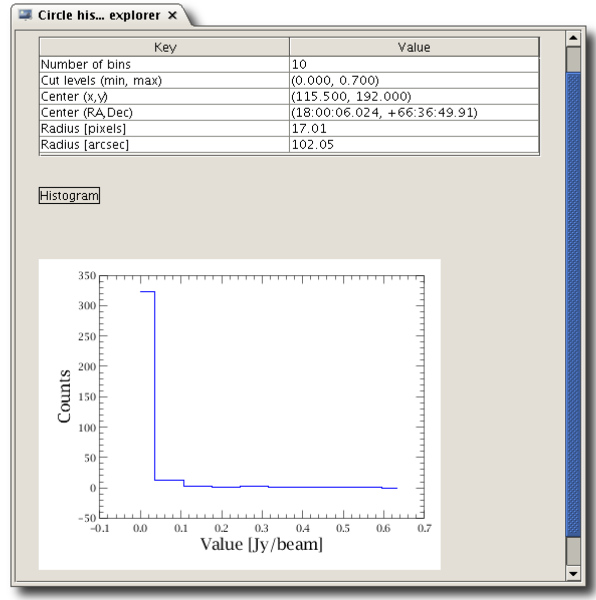


Figure 4.16. Display of the histogram task results, held in the histogram output dataset.

4.18.1. Histograms via the command line

To execute the histogram tasks via the command line, you must provide the following input parameters:

- The image (`image`).
- The cut levels (`lowCut` and `highCut`).
- The number of bins (`bins`).

To create histograms of a specific region within the image, you have to specify additional parameters:

- For a *circle* :
  - The position of the centre in pixels (`centerX` and `centerY`) or sky coordinates (`centerRa` and `centerDec`).
  - The radius in pixels (`radiusPixels`) or arcseconds (`radiusArcsec`).
- For an *ellipse* :
  - The position of the centre in pixels (`centerX` and `centerY`) or sky coordinates (`centerRa` and `centerDec`).
  - The width and height in pixels (`widthPixels` and `heightPixels`) or arcseconds (`widthArcsec` and `heightArcsec`).
- For a *rectangle* :
  - The position of the corner with the smallest coordinate values, in pixels (`minX` and `minY`) or sky coordinates (`minRa` and `minDec`).
  - The width and height in pixels (`widthPixels` and `heightPixels`) or arcseconds (`widthArcsec` and `heightArcsec`).
- For a *polygon* :

- The positions of the vertices in pixels (`edgesPixel` , stored as `x1` , `y1` , `x2` , `y2` and so on) or sky coordinates (`edgesSky` , stored as `RA1` , `Dec1` , `RA2` , `Dec2` , and so on).

To create a histogram, follow this example:

```
# Creating an image composed of random data
myImage = SimpleImage()
myImage.image=RESHAPE(Double1d.range(256*256), [256,256])

# Creating an image composed of random data and WCS information
myImage2 = SimpleImage()
myImage2.image=RESHAPE(Double1d.range(256*256), [256,256])
# Create a fake image 200x300 pixels in size

wcs2 = Wcs()  # Creating a WCS.
wcs2.setCrpix1(128)
wcs2.setCrpix2(128)  # The central pixel, in this case at (128, 128).
# Setting the position of the central pixel. In this case, the
# central pixel is located at 6h46'42.387" and 0 degrees 49'45.94".
wcs2.setCrval1(101.676612741936)
wcs2.setCrval2(0.829427624677429)
# Setting the type of the axes. The first axis defines the right
# ascension and the second axis the declination, both in a gnomonic projection.
wcs2.setCtype1("RA---TAN")
wcs2.setCtype2("DEC--TAN")
# Setting the coordinate system (here the standard ICRS type) and the equinox.
wcs2.setRadesys("ICRS")
wcs2.setEquinox(2000.0)
# Creating the linear transformation matrix, which defines
# the pixel size and the rotation of the images.
wcs2.setParameter("cd1_1", -1.9064468150235E-6, "")
wcs2.setParameter("cd1_2", 3.39797311269006E-4, "")
wcs2.setParameter("cd2_1", 3.39811958581193E-4, "")
wcs2.setParameter("cd2_2", 1.580446989748E-6, "")

myImage2.wcs = wcs2  # Applying the set of WCS information to our image

# Making a histogram of an image
histogram = imageHistogram(image = myImage, lowCut = 0.0, \
 highCut = 255.0, bins = 10)

# Making a histogram of a region bounded by a circle
circleHistogramPixel = circleHistogram(image = myImage, centerX = 417.5, \
 centerY = 240.0, radiusPixels = 217.6, lowCut = 9.0, highCut = 255.0, bins = 10)
circleHistogramSky = circleHistogram(image = myImage2, centerRa = "02:00:28.319", \
 centerDec = "-22:26:26.15", radiusArcsec = 219.3, lowCut = 9.0, \
 highCut = 255.0, bins = 10)

# Making a histogram of a region bounded by an ellipse
ellipseHistogramPixel = ellipseHistogram(image = myImage, centerX = 360.0, \
 centerY = 237.0, widthPixels = 642.0, heightPixels = 229.1, lowCut = 9.0, \
 highCut = 255.0, bins = 10)
ellipseHistogramSky = ellipseHistogram(image = myImage2, centerRa = "02:00:24.138",
 \
 centerDec = "-22:26:29.22", widthArcsec = 647.136, heightArcsec = 230.9, \
 lowCut = 9.0, highCut = 255.0, bins = 10)

# Making a histogram of a region bounded by a rectangle
rectangleHistogramPixel = rectangleHistogram(image = myImage, minX = 211.0, \
 minY = 127.0, widthPixels = 471.0, heightPixels = 175.0, lowCut = 9.0, \
 highCut = 255.0, bins = 10)
rectangleHistogramSky = rectangleHistogram(image = myImage2, minRa = "02:00:13.308",
 \
 minDec = "-22:28:20.17", heightArcsec = 474.8, widthArcsec = 176.4, \
 lowCut = 9.0, highCut = 255.0, bins = 10)

# Making a histogram of a region bounded by a polygon
pyEdgesPixel = Double1d([133.0, 206.0, 247.0, 333.0, 620.0, 233.0, 487.0, 112.01])
polygonHistogramPixel = polygonHistogram(image = myImage, \
 edgesPixel = pyEdgesPixel, lowCut = 9.0, highCut = 255.0, bins = 10)
```

```
pyEdgesSky = String1d(["05:47:51.56", "-51:04:36.74", "05:47:37.37",
 "-510:59:08.06", "05:46:39.92",
                    "-51:00:48.54"])
polygonHistogramSky = polygonHistogram(image = myImage2, \
 edgesSky = pyEdgesSky, lowCut = 9.0, highCut = 255.0, bins = 10)
```

Example 4.64. Creating a histogram from an image.

Note

For each task, all dimensions must have the same unit.

You can specify dimensions in arcseconds only if the image has a valid WCS and the pixel scaling is the same in both directions.

The following table lists some methods useful to inspect the output of the histogram tasks. You can use these methods on histograms created from regions of any shape:

Procedure 4.2. Available methods for the output of histogram tasks.

1. `getNbOfBins()`

   Returns the number of bins.

   ```
   # Java style
   print myHistogram.getNbOfBins()
   # Jython style
   print myHistogram.nbOfBins
   ```

   Example 4.65. Getting the number of bins from the histogram.

2. `getLowCut()`

   Returns the lower cut level of the histogram. A `getHighCut()` method to return the upper cut level is also available.

   ```
   # Java style
   print myHistogram.getLowCut()
   # Jython style
   print myHistogram.lowCut
   ```

   Example 4.66. Getting the lower cut level of the histogram.

3. `getHistogram()`

   Returns the entire histogram as a table dataset.

   ```
   # Java style
   myTable = myHistogram.getHistogram()
   # Jython style
   myTable = myHistogram.histogram
   ```

   Example 4.67. Converting the histogram to a table dataset.

4. `getValues()`

   Returns the values of the histogram bins as a `Double1d` .

   ```
   # Java style
   print myHistogram.getValues()
   # Jython style
   print myHistogram.values
   ```

   Example 4.68. Getting the values of the bins as a Double1d array.

5. `getFrequencies()`

Returns the occurrences of each value (the height of the histogram bars) as a `Double1d` .

```
# Java style
print myHistogram.getFrequencies()
# Jython style
print myHistogram.frequencies
```

Example 4.69. Getting the count for each histogram bin as a Double1d.

6. `getUnit()`

   Returns the name of the unit in which the bin values are expressed.

```
# Java style
print myHistogram.getUnit()
# Jython style
print myHistogram.unit
```

Example 4.70. Get the unit of the histogram values.

Additional methods are available for the tasks creating histograms from regions of specific shapes. These methods are listed in the following tables.

Procedure 4.3. Available methods for the output of the `circleHistogram` task.

1. `getCenterPixelCoordinates()`

   Returns the centre of the circle in pixel coordinates as a `Double1d` .

```
# Java style
print myHistogram.getCenterPixelCoordinates()
# Jython style
print myHistogram.centerPixelCoordinates
```

Example 4.71. Getting the centre pixel coordinates of the circle histogram.

2. `getCenterSkyCoordinates()`

   Returns the centre of the circle in sky coordinates as a `String1d` .

```
# Java style
print myHistogram.getCenterSkyCoordinates()
# Jython style
print myHistogram.centerSkyCoordinates
```

Example 4.72. Getting the centre sky coordinates of the circle histogram.

3. `getRadiusPixels()`

   Returns the radius of the circle in pixels.

```
# Java style
print myHistogram.getRadiusPixels()
# Jython style
print myHistogram.radiusPixels
```

Example 4.73. Getting the radius of the circle histogram in pixels.

4. `getRadiusArcsec()`

   Returns the radius of the circle in arcseconds.

```
# Java style
print myHistogram.getRadiusArcsec()
```

```
# Jython style
print myHistogram.radiusArcsec
```

Example 4.74. Getting the radius of the circle histogram in arcseconds.

Procedure 4.4. Available methods for the output of the `ellipseHistogram` task.

1. `getCenterPixelCoordinates()`

   Returns the centre of the ellipse in pixel coordinates as a `Double1d` .

   ```
   # Java style
   print myHistogram.getCenterPixelCoordinates()
   # Jython style
   print myHistogram.centerPixelCoordinates
   ```

   Example 4.75. Getting the centre pixel coordinates for the ellipse histogram.

2. `getCenterSkyCoordinates()`

   Returns the centre of the ellipse in sky coordinates as a `String1d` .

   ```
   # Java style
   print myHistogram.getCenterSkyCoordinates()
   # Jython style
   print myHistogram.centerSkyCoordinates
   ```

   Example 4.76. Getting the centre sky coordinates for the ellipse histogram.

3. `getWidthPixels()`

   Returns the width of the ellipse in pixels. A `getHeightPixels()` method to get the height of the ellipse is also available.

   ```
   # Java style
   print myHistogram.getWidthPixels()
   # Jython style
   print myHistogram.widthPixels
   ```

   Example 4.77. Getting the width of the ellipse histogram in pixels.

4. `getWidthArcsec()`

   Returns the width of the ellipse in arcseconds. A `getHeightArcsec()` method to get the height of the ellipse is also available.

   ```
   # Java style
   print myHistogram.getWidthArcsec()
   # Jython style
   print myHistogram.widthArcsec
   ```

   Example 4.78. Getting the width of the ellipse histogram in arcseconds.

Procedure 4.5. Available methods for the output of the `rectangleHistogram` tasks.

1. `getUpperLeftCornerPixelCoordinates()`

   Returns the pixel coordinates of the upper left corner of the rectangle as a `Double1d` . The upper left corner is the corner with lowest pixel coordinates.

   ```
   # Java style
   print myHistogram.getUpperLeftCornerPixelCoordinates()
   ```

```
# Jython style
print myHistogram.upperLeftCornerPixelCoordinates
```

Example 4.79. Getting the upper left corner pixel coordinates of a rectangle histogram.

2. `getUpperLeftCornerSkyCoordinates()`

   Returns the sky coordinates of the upper left corner of the rectangle as a `String1d` . The upper left corner is the corner with lowest pixel coordinates.

```
# Java style
print myHistogram.getUpperLeftCornerSkyCoordinates()
# Jython style
print myHistogram.upperLeftCornerSkyCoordinates
```

Example 4.80. Getting the upper left corner sky coordinates of a rectangle histogram.

3. `getWidthPixels()`

   Returns the width of the rectangle in pixels. A `getHeightPixels()` method to get the height of the rectangle is also available.

```
# Java style
print myHistogram.getWidthPixels()
# Jython style
print myHistogram.widthPixels
```

Example 4.81. Getting the width in pixels of a rectangle histogram.

4. `getWidthArcsec()`

   Returns the width of the rectangle in arcseconds. A `getHeightArcsec()` method to get the height of the rectangle is also available.

```
# Java style
print myHistogram.getWidthArcsec()
# Jython style
print myHistogram.widthArcsec
```

Example 4.82. Getting the width in arcseconds of a rectangle histogram.

Procedure 4.6. Available methods for the output of the `polygonHistogram` task.

1. `getEdges()`

   Returns the edges of the histogram as a composite dataset.

```
# Java style
myEdges = myHistogram.getEdges()
# Jython style
myEdges = myHistogram.edges
```

Example 4.83. Getting the edges of a polygon histogram.

2. `getEdgesPixelCoordinates()`

   Returns the pixel coordinates of the polygon vertices as a table dataset.

```
# Java style
myVertices = myHistogram.getEdgesPixelCoordinates()
# Jython style
myVertices = myHistogram.edgesPixelCoordinates
```

Example 4.84. Getting the vertices of a polygon histogram as a table dataset.

3. `getEdgesPixelCoordinatesDouble2d()`

   Returns the pixel coordinates of the polygon vertices as a `Double2d`.

```
# Java style
myVertices = myHistogram.getEdgesPixelCoordinatesDouble2d()
# Jython style
myVertices = myHistogram.edgesPixelCoordinatesDouble2d
```

   Example 4.85. Getting the vertices of a polygon histogram as a Double2D array.

4. `getEdgesSkyCoordinates()`

   Returns the sky coordinates of the polygon edges as a table dataset.

```
# Java style
myVertices = myHistogram.getEdgesSkyCoordinates()
# Jython style
myVertices = myHistogram.edgesSkyCoordinates
```

   Example 4.86. Getting the edges of a polygon histogram in sky coordinates.

4.19. Finding and extracting sources

See Section 4.2 for general information on how to run image analysis tasks.

HIPE includes the `sourceExtractorDaophot` and `sourceExtractorSussextractor` tasks, designed primarily for use on PACS and SPIRE maps.

- **sourceExtractorSussextractor** implements the SUSSEXtractor algorithm, described by Savage & Oliver (2007), ApJ, 661, 1339. The algorithm uses a model of a source on top of a flat background, and finds the maximum likelihood flux+background combination, all in one step, without iterations. The image is smoothed with a convolution kernel, derived from the point response function, and the resulting smoothed image is searched for peaks, which are taken to be the positions of the point sources. The intensity in the smoothed image at the position of a point source is taken as the estimate of that source's flux density. Note that `NaN` pixels near point sources in the image or error extensions of the input image may prevent `sourceExtractorSussextractor` from finding that source.

- **sourceExtractorDaophot** implements the DAOPHOT (classic) algorithm, following the `FIND` and `APER` procedures in the IDL Astronomy User's Library . The image is smoothed with the DAOPHOT convolution kernel to find the source positions, as in the `sourcExtractorSussextractor` task. Then the source flux densities are estimated using aperture photometry, as in the `aperturePhotometry` task.

For more details on the algorithms used by the two tasks, see the corresponding entries in the *User's Reference Manual* :

- sourceExtractorDaophot in *HCSS User's Reference Manual*

- sourceExtractorSussextractor in *HCSS User's Reference Manual*

The following figure shows the lists of parameters for the two tasks:

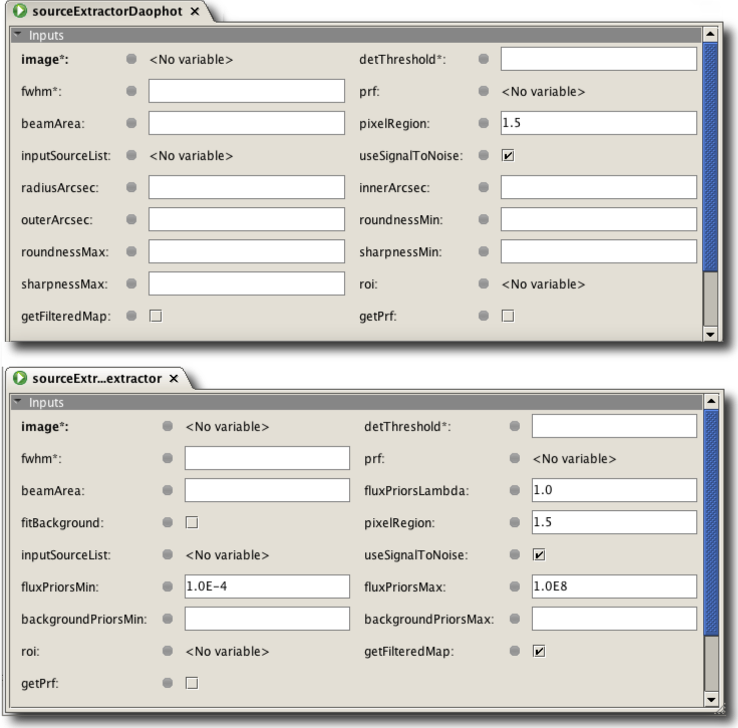


Figure 4.17. List of parameters for the two source extraction tasks.

Output. The output is of type `SourceListProduct` and is called `sourceList` by default. You can inspect it in the *Product Viewer* like any other product, as shown by the following figure. Measurement units are shown next to each column name.

Units. Note that fluxes are always given in mJy, whatever the units of the original image. The image must have units specified, and these must be mJy, Jy or MJy per pixel, sr or beam. The task will fail if it encounters units it does not understand.

Use `print` *`myImage`* `.getUnit()` to view the image units, and *`myImage`* `.setUnit("MJy/sr")` to set the units of the image to MJy/sr. Other units such as Jy/beam, Jy/pixel and mJy/pixel also work. You must set units manually only if the image has no units, or if the units are inappropriate (for example, Jy instead of Jy/beam).

Warning

Changing the units of an image as described above has no effect on the data values.

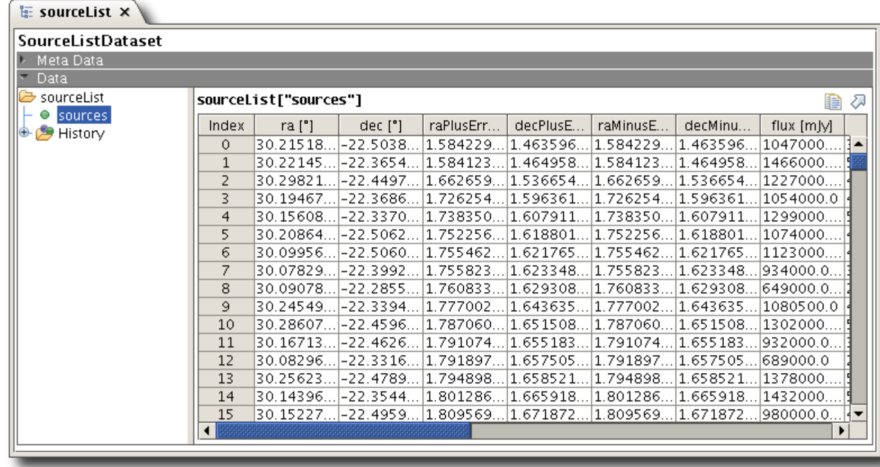


Figure 4.18. The list of sources shown in the Product Viewer, with the internal dataset highlighted.

To display the extracted sources on the image, or on any other image with the same field of view, drag and drop the `sourceList` variable on the image in a display. A circle is overlaid at the location of each source. Dragging and dropping will not work if you select the *returnPixelCoordinates* checkbox in the task graphical interface. When this option is selected, the task returns source coordinates in pixels rather than astronomical coordinates.

Creating residual and source images

Images can easily be created to assist with evaluating the quality of the extracted source list. A residual image (the original image with sources subtracted) can be created using

```
imageWithSourcesSubtracted = sourceList.subtractedFromImage(image, fwhm[, beamArea])
```

Example 4.87. Creating a residual image subtracting the sources for the original image.

A source image (an artificial image based on the source list) can be created using

```
imageOfSources = sourceList.asNewImage(image, fwhm[, beamArea])
```

Example 4.88. Creating an image containing only the sources of the original image.

In both of these cases, a Gaussian PRF (Point Response Function) is used.

Extracting and viewing additional outputs

If you check the *getPrf* or *getFilteredMap* checkbox, the output will include the *point response function* and the *filtered map* as additional images. For the SUSSEXtractor algorithm, the filtered map is equal to the input map convolved with the point response function, such that the value at each pixel gives an estimate of the flux of a source, in mJy, assuming there is a source located at the centre of that pixel. For the DAOPHOT algorithm, the filtered map gives the input map convolved with the DAOPHOT kernel.

If you select one or both of these additional outputs, the result of the task will be an *array* of products (more precisely, a Jython tuple). Double clicking on it in the *Variables* view will open a viewer. Alternatively you can extract the individual outputs with the following commands, assuming that the array is called `sourceList` as the task defaults to :

```
srcList = sourceList[0]
```

```
filteredMap = sourceList[1]
prf = sourceList[2]
```

Example 4.89. Extracting the results of the task from the output array.

If you select *getPrf* but not *getFilteredMap* , or vice versa, `result[0]` still returns the source list, while `result[1]` returns either the point response function or the filtered map, depending on what you have selected.

Specifying the positions of known sources

You can use a `SourceListProduct` as an input to the source extractor task to specify the positions of known sources. For example, it could be a source list created from another image with WCS coordinates. The task will then give the fluxes of sources at those positions. To provide the list of known sources, drag and drop a variable of type `SourceListProduct` onto the small circle next to the `inputSourceList` parameter in the task graphical interface. You can inspect the type of a variable by hovering the mouse pointer on the variable name in the *Variables* view. In this case you should see `herschel.ia.toolbox.srcext.SourceListProduct` .

The best way to create a `SourceListProduct` is to load the data from a text file. The file must have at least the `ra` and `dec` columns. For more information see *Working with source lists in text files* below.

Removing sources from the source list

To remove a source from the list use the following command, assuming your source list is called *`mySourceList`* :

```
mySourceList["sources"].removeRow(index)
```

Example 4.90. Removing a row from a sources list.

where `index` is the index of the source you want to remove. To find the index, open the source list by right clicking on the corresponding variable name in the *Variables* view and choosing *Open With → Product Viewer* . Then click on `sources` to display the table. The first column is the index.

Changing the colour and size of the source circles

When you extract sources, or drag and drop a list of sources onto an image, the circles representing the sources are green by default. To represent sources with circles of another colour you have to use the command line. Assuming that *`myImage`* and *`mySourceList`* are your image and list of sources, respectively, issue the following commands:

```
disp = Display(myImage)
disp.addPositionList(mySourceList, java.awt.Color.YELLOW)
```

Example 4.91. Plotting the source list along with the image with the help of the Display class.

This will plot yellow circles. For more information on the `java.awt.Color` class, used to specify different colours, see Section 3.25 .

To specify the sizes of the source circles, use the following command, where `disp` is still the display corresponding to `myImage` :

```
disp.addPositionList(mySourceList, sizes)
```

Example 4.92. Customising the size of the position circles when plotting sources with Display.

where *sizes* is a `Float1d` array of pixel sizes. This means that you can specify different sizes for different sources.

For example, to have the size of each circle to be proportional to the flux:

```
fluxes = mySourceList["sources"]["flux"].data
sizes = Float1d(3 + 7*fluxes/MAX(fluxes)) # Sizes will range from 3 to 10 pixels
disp.addPositionList(mySourceList, java.awt.Color.RED, sizes)
```

Example 4.93. Making the position circles of plotted sources proportional to the flux intensity.

You can also specify the colour and sizes in one step:

```
disp.addPositionList(mySourceList, color, sizes)
```

Example 4.94. Plotting fully customised position circles by passing Color and Float1d objects.

where *colour* is a colour specified by the `java.awt.Color` class (see Section 3.25).

If the image has an associated WCS, you can specify sizes in arcseconds, rather than pixels, with the `addPositionListWcs` method:

```
disp.addPositionListWcs(mySourceList, color, sizes)
```

Example 4.95. Plotting position circles that take the sizes as sky coordinates by passing a Wcs object.

In the last case you always have to specify the colour as well.

Specifying a custom point response function

By default, the point response function (PRF) is assumed to be Gaussian, with full-width half maximum in arcseconds provided by the *fwhm* parameter. Alternatively, you can specify a custom PRF via the *prf* parameter. This should be a variable of type `SimpleImage` . The image must be of odd width and height in number of pixels, with the peak at the centre, normalised such that it gives a point source flux of 1 Jy, in the units of the input map. The PRF image is assumed to have the same pixel scale of the main image, and does not need to have an associated WCS. If you input a PRF you do not need to specify the FWHM.

Extracting sources on part of the image

You can use the *roi* parameter to define a region of interest within the image and extract sources only inside that region. This can either be a `SkyMask` or a `Bool2d`. To define a rectangular region of interest between `raMin` and `raMax` and between `decMin` and `decMax`, use the following (note that this is just the prototype of the constructor, a proper example with values can be found at the end of this subsection):

```
roi = SkyMaskRectangle(raMin, raMax, decMin, decMax)
```

Example 4.96. Creating a rectangular region of interest (SkyMask) using sky coordinates in decimal degrees.

To define a circular region of interest centred on (`ra`, `dec`) and with radius 5 arcmin, use the following (again, note that this is just the prototype of the constructor, a proper example with values can be found at the end of this subsection):

```
roi = SkyMaskCircle(ra, dec, 5)
```

Example 4.97. Creating a circular region of interest (SkyMask) using sky coordinates in decimal degrees.

To define a region of interest based on a Bool2d variable, myBool2d, of the same dimensions as the image (for instance created with the image annotation toolbox, or via code in *HCSS User's Reference Manual*) use the following:

```
roi = myBool2d
```

Example 4.98. Creating a region of interest from a bidimensional array of booleans.

Variables of type SkyMask... can be combined using .or(), .and() and .xor() and inverted with .not(). For example, to define the region of interest to be the region more than 5 arcmin away from both (ra1, dec1) and (ra2, dec2), first combine two circles using .or() and then invert with .not():

```
roi = SkyMaskCircle(ra1, dec1, 5).or(SkyMaskCircle(ra2, dec2, 5)).not()
```

Example 4.99. Applying the boolean OR operation between SkyMasks.

A simpler example is the inversion of a region of interest, using .not(). This consists of the whole of the image excluding a 5 arcminutes hole centered at (ra, dec):

```
roi = SkyMaskCircle(ra, dec, 5).not()
```

Example 4.100. Inverting a region of interest.

A collection of SkyMask... variables can be combined as a SkyMaskUnion (equivalent to .or()) or as a SkyMaskIntersection (equivalent to .and()). To define the region of interest to be the union of skyMask1, skyMask2 and skyMask3, use the following:

```
roi = SkyMaskUnion([skyMask1, skyMask2, skyMask3])
```

Example 4.101. Joining the areas of three different SkyMasks.

You can manually define a SkyMask using the SkyMask toolbox. Right-click on an image to open the SkyMask toolbox, as in the following figure:

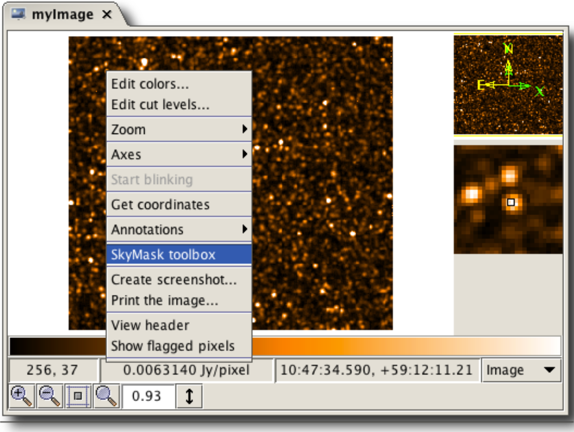


Figure 4.19. Opening the SkyMask toolbox.

Draw the regions you want to mask as shown in the following figure. Click on the scissors to created a variable called skyMask with the selected regions of the image masked. You can use this variable for the *roi* parameter in the source extractor task.

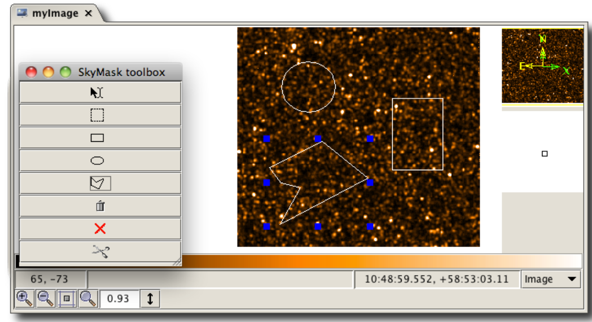


Figure 4.20. Drawing a region of interest on the image.

To visualise a `SkyMask` on an image, first display the image and then drag and drop the `SkyMask` variable onto the image. This will add a new layer to the image, set to 1.0 for those pixels of the image that are masked by the `SkyMask`. You can see a complete example of working with `SkyMasks` in images with WCS information below.

```
# Creating an image composed of random data
i = SimpleImage()
i.image=RESHAPE(Double1d.range(256*256), [256,256])
# Create a fake image 200x300 pixels in size

wcs2 = Wcs()  # Creating a WCS.
wcs2.setCrpix1(128)
wcs2.setCrpix2(128)  # The central pixel, in this case at (128, 128).
# Setting the position of the central pixel. In this case, the
# central pixel is located at 6h46'42.387" and 0 degrees 49'45.94".
wcs2.setCrval1(101.676612741936)
wcs2.setCrval2(0.829427624677429)
# Setting the type of the axes. The first axis defines the right
# ascension and the second axis the declination, both in a gnomonic projection.
wcs2.setCtype1("RA---TAN")
wcs2.setCtype2("DEC--TAN")
# Setting the coordinate system (here the standard ICRS type) and the equinox.
wcs2.setRadesys("ICRS")
wcs2.setEquinox(2000.0)
# Creating the linear transformation matrix, which defines
# the pixel size and the rotation of the images.
wcs2.setParameter("cd1_1", -1.9064468150235E-6, "")
wcs2.setParameter("cd1_2", 3.39797311269006E-4, "")
wcs2.setParameter("cd2_1", 3.39811958581193E-4, "")
wcs2.setParameter("cd2_2", 1.580446989748E-6, "")

i.wcs = wcs2  # Applying the set of WCS information to our image

# Defining two ROIs, one rectangular and another circular
roiRect = SkyMaskRectangle(101.67661273, 101.67661275, 0.82942761, 0.82942763)
roiCirc = SkyMaskCircle(wcs2.getCrval1(), wcs2.getCrval2(), 0.000002)

# Testing the boolean capabilities of the ROIs
roiBool = roiRect.or(roiCirc)

# Does this ROI mask the center of the image? True
print roiBool.masks(wcs2.getCrval1(), wcs2.getCrval2())
```

Example 4.102. Masking an image with a complex, boolean SkyMask.

Working with source lists in text files

To export the source list to a text file, run the `asciiTableWriter` task. First you have to retrieve the source list dataset from the result of the source extraction:

```
sourceListDataset = results[0]
```

Example 4.103. Retrieving the source list dataset from the results output list of a source extraction task.

Then click on `sourceListDataset` in the *Variables* view, and you will find `asciiTableWriter` among the applicable tasks.

To import a text file as a list of sources, use the `asciiTableReader` task. The result of this task is of type `TableDataset`. To obtain a `SourceListProduct`, issue the following command:

```
importedSourceList = SourceListProduct(table)
```

Example 4.104. Creating a SourceListProduct from a source list table dataset.

Note that the column names in the imported source list must match the default column names in a `SourceListDataset` (`ra`, `dec`, `flux` and so on). Column names are case insensitive.

For more information on exchanging data with text files, see Chapter 2 .

Working with source lists in FITS files

To export a list of sources of type `SourceListProduct` to a FITS file, select `simpleFitsWriter` from the applicable tasks.

To import a `SourceListProduct` stored in a FITS file, load the file with *File* → *Open File* , or double click on the file in the *Navigator* view, and HIPE will do the rest. If the FITS file does not contain a `SourceListProduct`, the data will be imported as a generic `Product`, with the source list contained in a `Dataset`. You can create a proper `SourceListProduct` with the following command, assuming that the dataset is called `HDU_1`:

```
importedSourceList = SourceListProduct(sourceList["HDU_1"])
```

Example 4.105. Creating a SourceListProduct from a source list dataset.

For more information on exchanging data with FITS files, see Section 1.16 .

Common problems

- **No error extension (sourceExtractorSussextractor only)**

  The `sourceExtractorSussextractor` task requires the input image to have an error extension, and if this is not present the task fails. The error in the pixel values should be determined as part of the map-making algorithm. However, you can add an error extension to an image, assuming the uncertainty in each pixel is 0.001, with the following command: `myImage.setError(myImage.getImage() * 0 + 0.001)`.

- **Invalid units, or units not specified**

  Both source extraction tasks require the input image to specify its units in a valid format. The task fails if it cannot recognise the units of the image as units of surface brightness. To set the units of the `SimpleImage`, `image`, to be "Jy/beam" (for example), use `image.setUnit("Jy/beam")`. Other units based on Jy, mJy, MJy, beam, pixel, sr and so on are recognised.

4.20. Fitting sources

See Section 4.2 for general information on how to run image analysis tasks.

You can fit a source with a two-dimensional Gaussian using the `sourceFitting` task. Open the task dialogue window and click and drag with the mouse on the image to enclose the source you want to fit. With the drop-down lists below the image you can then choose between circular and elongated source, and between constant and sloping background. Click *Accept* to run the task.

If the fit is successful, the result appears in the *Variables* view as a variable of type `SourceFittingProduct`, and of default name `parameters`. Double click on the variable to open the parameters table in the *Editor* view.

You can run the sourceFitter from the command line as in the following example:

```
# Getting a PACS photo observation for NGC 281
myObs = getObservation(obsid = 1342247320, useHsa = True)

# Using the level-2 blue map as the image
elongatedSrc = myObs.refs["level2"].product.refs["HPPPMAPB"].product

# Running the sourceFitting task within an appropriately big frame
parameters = sourceFitting(elongated=True, slope=False, image=elongatedSrc, \
      minX = 852.0, minY = 908.0, width=926.0, height=880.0)
```

Example 4.106. Fitting sources in an image.

The *image* parameter is the input image, *elongated* indicates whether the source is elongated or circular, *slope* indicates whether the background has a slope or is constant and *minX* , *minY* , *width* and *height* determine the box where the fitter searches for the source.

For more information on running source fitting from the command line, see the *User's Reference Manual* : SourceFittingTask in *HCSS User's Reference Manual* .

4.21. Aperture photometry

See Section 4.2 for general information on how to run image analysis tasks.

You can perform aperture photometry on an image using a circular target aperture. You can determine the appropriate sky values using either an annular or rectangular sky aperture. If there is no suitable area for determining the sky, you can also provide a fixed sky value.

You can use five algorithms to estimate the sky:

- average: average of the pixel values within the sky area.
- median: median of the pixel values within the sky area.
- mean-median: average of all the values closer to the median than a specified number of standard deviations, for example 1.5.
- synthetic mode: the mean-median algorithm is repeated in an iterative process up to ten times, or until the average does not change anymore.
- daophot: translation of the algorithm used in the IDL `daophot` package. This is the default method.

4.21.1. Centroiding

The aperture photometry tasks include an option for centroiding. Selecting this option will refine the input coordinates to those determined by a centroiding algorithm. This works as follows: a square region, with length 4 times the requested target radius, is fit with a 2D Gaussian plus a constant; the indicated target centre is the initial estimate of the source position and the indicated target radius is the initial guess at the width of the Gaussian; if the centroiding can find a fit, then the "Centroiding" keyword is set to "success" and the coordinates updated; if the algorithm finds no solution, then the new coordinates are set to those either of the brightest pixel in the square region or, where there are multiple bright pixels, the pixel closest to the initial guess, and "Centroiding" is set to "failed". If using the GUI form of the task, the circle indicating where the input position is located will move to the centroided position. The Meta datum "Centroid" is added to the output "result" of the task (this information is also displayed in the results table).

Tip

If the centroiding was successful, note that this does not necessarily mean that a good centroid was found, but rather than the centroiding algorithm could complete. Experience

shows that this centroiding does not always perform well, and if you are using the aperture photometry tasks in an automatic way (i.e. not checking the resulting coordinates on the image), it is recommended that this option is not selected. If it is necessary to refine the target coordinates (e.g. you only have an estimated position), the sourceFitting task is a good alternative for an automatic refining of point source coordinates, and the output of sourceFitting can be used as input to the photometry tasks (with the centroiding option then turned off). See Section 4.20 for an explanation of sourceFitting, see also Section 4.22 for examples.

4.21.2. Units and aperture photometry

To use an image as input of an aperture photometry task, make sure that the units are flux/pixel. The flux may be expressed in Jy or in ADU depending on the kind of image. This ensures that the end product of the photometry will be expressed in (flux/pixel)*pixel = flux units.

For example, if the image units are Jy/beam, then you need to convert first into MJy/sr, using the beamsize, then to Jy/pixel using the pixel area. Then you can perform the photometry, whose output will be in Jy.

The units of PACS maps are Jy/pixel, so no conversion is needed. For instructions on performing aperture photometry on SPIRE maps, see the SPIRE Data Reduction Guide .

4.21.3. Point sources

4.21.3.1. Annular sky aperture photometry (annularSkyAperturePhotometry)

The best place to compute the background is a circular annulus as close to the target as possible. This task determines the background in an annulus around the circular target aperture.

Via the GUI

To perform **annular aperture photometry** , run the `annularSkyAperturePhotometry` task. The image appears in a new tab within the *Editor* view. Below the image you can find the interface to enter the task options.

In the *Target center* pane, a drop-down menu gives you three ways to identify the target:

- By mouse interaction. With this option selected, click once on the image to select the target.
- By entering the X and Y pixel coordinates.
- By entering sky coordinates, if the image has a valid WCS. Use the format `"02:00:39.4"` for Right Ascension and `"-22:27:20.6"` for Declination, including the quotation marks.

The target is identified by a circle.

In the *Apertures* panel you can enter the radii for the target and sky regions. In the *Sky estimation* panel you can specify the algorithm, and whether to use entire pixels or fractional pixels.

You can reset the parameters at any time by clicking *Clear* . Click *Accept* to execute the task. The circular radii are shown on the image (see Figure 4.21).

You can display the results by double clicking on the `result` variable shown in the *Variables* view (see Figure 4.22).

Note

The error given by the aperture photometry task is not correct in the case of flux-calibrated Herschel images. It is however correct in case of CCD images where the units are in ADU. Currently the best way to determine the photometric error is to place several apertures on

the background around the source, and to measure the flux within these apertures. The standard deviation of the values gives the photometric error on the source.

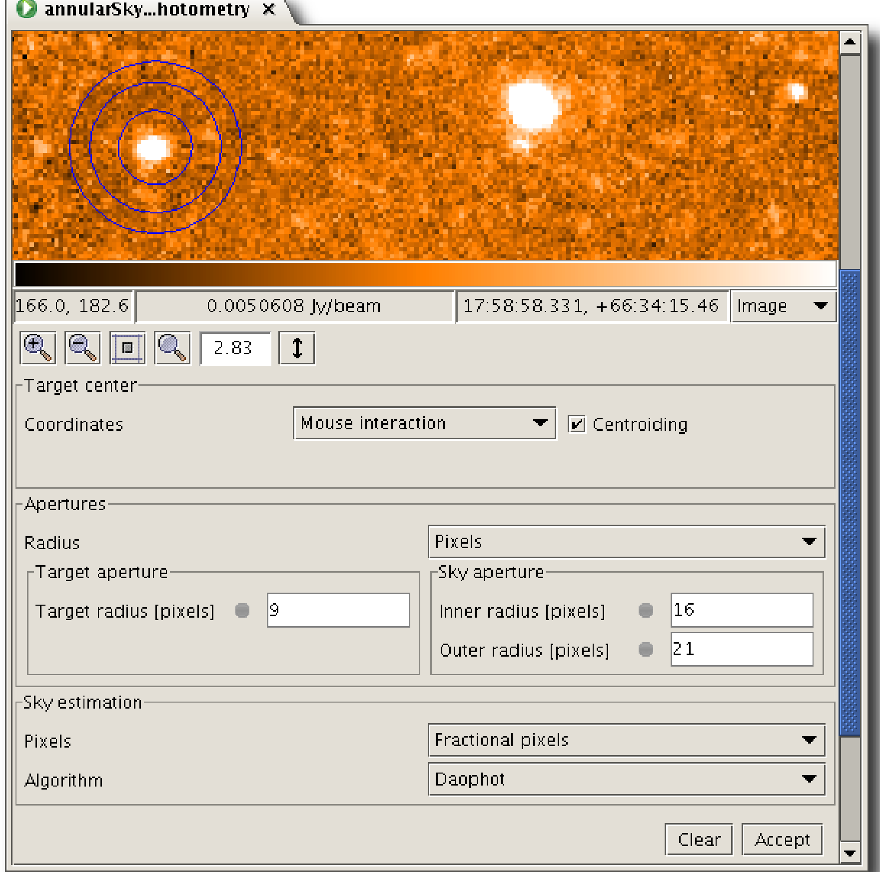


Figure 4.21. Aperture photometry with an annular sky aperture as displayed in HIPE.

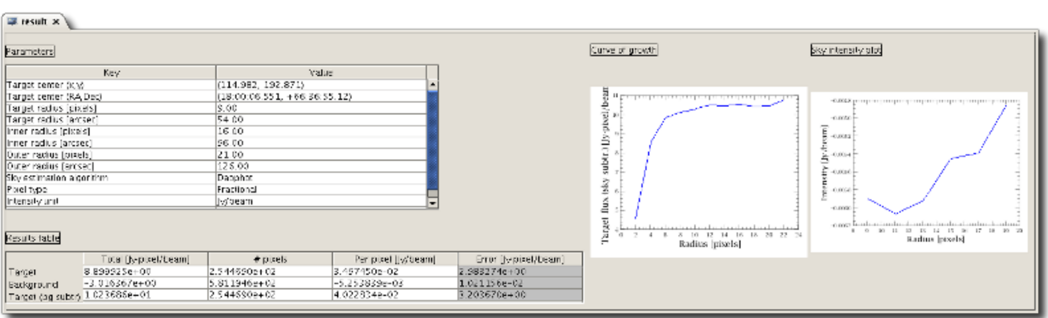


Figure 4.22. Aperture photometry results plot and tables. Note that n.a. stands for "not applicable" and typically occurs when units are not assigned to the image.

The results include two plots, useful to judge whether your choice of radii was sensible:

- A *curve of growth* , showing the target flux, without the sky, as a function of the radius.
- A *sky intensity plot* , showing the intensity per sky pixel as a function of the inner radius, the outer radius being constant.

Via the command line

To perform aperture photometry from the command line you must specify the following input parameters:

- The image (`image`).
- The position of the target centre, in pixels (`centerX` and `centerY`) or sky coordinates (`centerRa` and `centerDec`).
- The radius of the target, in pixels (`radiusPixels`) or in arcseconds (`radiusArcsec`).
- The inner and outer radii of the annular sky aperture, in pixels (`innerPixels` and `outerPixels`) or arcseconds (`innerArcsec` and `outerArcsec`).
- Whether fractional pixels are to be used (`fractional`). It can be `True` (1) or `False` (0), `True` by default.
- The sky estimation algorithm (`algorithm`). This is an integer with the following possible values: 0 for average, 1 for median, 2 for mean-median, 3 for synthetic mode and 4 for the algorithm used by Daophot.

The following examples show how to perform aperture photometry:

```
from herschel.calsdb.util import Coordinate
# Getting a point-source PACS observation
myObs = getObservation(obsid = 1342184579, useHsa = True)

# Extracting the map
pointSrcMap = myObs.refs["level2"].product.refs["HPPPMAPB"].product

# The target centre specified in pixel coordinates, the radii in pixels
# and using fractional pixels
photPixels = annularSkyAperturePhotometry(image = pointSrcMap, centerX = 149.0, \
 centerY = 160.0, radiusPixels = 5.0, innerPixels = 20.0, outerPixels = 40.0, \
 fractional = 1, algorithm = 4)

# The task requires strings for the RA and the DEC, so they must be converted
sexagesimalRA = Coordinate.ra2String(232.85505321881504)
sexagesimalDEC = Coordinate.dec2String(77.34943290224619)

# The target center specified in sky coordinates, the radii in arcseconds
# and using entire pixels
photSky = annularSkyAperturePhotometry(image = pointSrcMap, \
 centerRa = sexagesimalRA, centerDec = sexagesimalDEC, radiusArcsec = 5.04, \
 innerArcsec = 20.16, outerArcsec = 40.32, fractional = 0, algorithm = 4)
```

Example 4.107. Performing annular sky aperture photometry on a PACS map.

```
from herschel.calsdb.util import Coordinate
# Getting a point-source SPIRE observation
myObs = getObservation(obsid = 1342182472, useHsa = True)

# Extracting the map
pointSrcMap = myObs.refs["level2"].product.refs["psrcPSW"].product

# The target centre specified in pixel coordinates, the radii in pixels
# and using fractional pixels
photPixels = annularSkyAperturePhotometry(image = pointSrcMap, centerX = 98.0, \
 centerY = 114.0, radiusPixels = 5.0, innerPixels = 20.0, outerPixels = 40.0, \
 fractional = 1, algorithm = 4)

# The task requires strings for the RA and the DEC, so they must be converted
sexagesimalRA = Coordinate.ra2String(65.81568453395388)
sexagesimalDEC = Coordinate.dec2String(-1.3421057265332947)

# The target center specified in sky coordinates, the radii in arcseconds
# and using entire pixels
photSky = annularSkyAperturePhotometry(image = pointSrcMap, \
 centerRa = sexagesimalRA, centerDec = sexagesimalDEC, radiusArcsec = 5.04, \
 innerArcsec = 20.16, outerArcsec = 40.32, fractional = 0, algorithm = 4)
```

Example 4.108. Performing annular sky aperture photometry on a SPIRE map.

Note

You can specify distances in arcseconds (here `radiusArcsec` , `innerArcsec` and `outerArcsec`) only if the pixel scaling is the same in both directions (`myImage.getCdelt1() = myImage.getCdelt2()`). Moreover, the image must have a valid WCS.

You must specify all distances in the same unit, pixels or arcseconds.

You can inspect the output product with the methods listed in the following table:

Procedure 4.7. Available methods for the output of the `annularSkyAperturePhotometry` task.

1. `getTargetCenterPixelCoordinates()`

   Returns the pixel coordinates of the target centre as a `Double1d` .

   ```
   # Java style
   print myPhot.getTargetCenterPixelCoordinates()
   # Jython style
   print myPhot.targetCenterPixelCoordinates
   ```

   Example 4.109. Getting the centre pixel coordinates for the target.

2. `getTargetCenterSkyCoordinates()`

   Returns the sky coordinates of the target centre as a `String1d` .

   ```
   # Java style
   print myPhot.getTargetCenterSkyCoordinates()
   # Jython style
   print myPhot.targetCenterSkyCoordinates
   ```

   Example 4.110. Getting the centre sky coordinates for the target.

3. `getTargetRadiusPixels()`

   Returns the radius of the target aperture in pixels.

   ```
   # Java style
   print myPhot.getTargetRadiusPixels()
   # Jython style
   print myPhot.targetRadiusPixels
   ```

   Example 4.111. Getting the target radius in pixels.

4. `getTargetRadiusArcsec()`

   Returns the radius of the target aperture in arcseconds.

   ```
   # Java style
   print myPhot.getTargetRadiusArcsec()
   # Jython style
   print myPhot.targetRadiusArcsec
   ```

   Example 4.112. Getting the target radius in arcseconds.

5. `getInnerRadiusPixels()`

   Returns the inner radius of the sky estimation annulus in pixels. The `getOuterRadiusPixels()` method to return the outer radius is also available.

   ```
   # Java style
   print myPhot.getInnerRadiusPixels()
   # Jython style
   ```

```
print myPhot.innerRadiusPixels
```

Example 4.113. Getting the outer target radius in pixels.

6. `getInnerRadiusArcsec()`

   Returns the inner radius of the sky estimation annulus in arcseconds. The `getOuterRadiusArcsec()` method to return the outer radius is also available.

```
# Java style
print myPhot.getInnerRadiusArcsec()
# Jython style
print myPhot.innerRadiusArcsec
```

Example 4.114. Getting the inner radius of the sky estimation annulus in arcseconds.

7. `getAlgorithm()`

   Returns the name of the algorithm used by the task.

```
# Java style
print myPhot.getAlgorithm()
# Jython style
print myPhot.algorithm
```

Example 4.115. Getting the name of the algorithm used by the aperture photometry task.

8. `getPixels()`

   Returns the type of pixels, either `entire` or `fractional` , used by the task.

```
# Java style
print myPhot.getPixels()
# Jython style
print myPhot.pixels
```

Example 4.116. Checking if the aperture photometry task considers fractional or entire pixels.

9. `getTable()`

   Returns the results table as a table dataset.

```
# Java style
print myPhot.getTable()
# Jython style
print myPhot.table
```

Example 4.117. Getting the results of the aperture photometry task as a table dataset.

10. `getDouble2dTable()`

    Returns the results table as a `Double2d` .

```
# Java style
print myPhot.getDouble2dTable()
# Jython style
print myPhot.double2dTable
```

Example 4.118. Getting the results of the aperture photometry task as a Double2d table.

11. `getTargetPlusSkyTotal()`

    Returns the total flux of the target plus the sky. To get the same for the sky and for the target without the sky, replace `TargetPlusSky` with `Sky` or `Target` in the method name. To get the corresponding error, replace `Total` with `Error` in the method name.

```
# Java style
print myPhot.getTargetPlusSkyTotal()
# Jython style
print myPhot.targetPlusSkyTotal
```

Example 4.119. Getting the total flux (target+sky).

12. `getNbOfTargetPlusSkyPixels()`

    Returns the number of pixels in the target and sky areas. To get the same for the sky and for the target areas only, replace `TargetPlusSky` with `Sky` or `Target` in the method name.

```
# Java style
print myPhot.getNbOfTargetPlusSkyPixels()
# Jython style
print myPhot.nbOfTargetPlusSkyPixels
```

Example 4.120. Getting the total number of pixels (target+sky).

13. `getIntensityPerTargetPlusSkyPixel()`

    Returns the intensity per pixel for the target plus the sky. To get the same for the sky and for the target without the sky, replace `TargetPlusSky` with `Sky` or `Target` in the method name.

```
# Java style
print myPhot.getIntensityPerTargetPlusSkyPixel()
# Jython style
print myPhot.intensityPerTargetPlusSkyPixel
```

Example 4.121. Getting the flux intensity averaged by the total pixels (target+sky).

14. `getCurveOfGrowth()`

    Returns the curve of growth as a table dataset. The table dataset has two columns: *Growth radius* and *Growth flux* .

```
# Java style
myTable =  myPhot.getCurveOfGrowth()
# Jython style
myTable = myPhot.curveOfGrowth
```

Example 4.122. Getting the curve of growth for the results of aperture photometry task.

15. `getGrowthRadius()`

    Returns a `Double1d` with the values in the *Growth radius* column of the curve of growth table dataset returned by the `getCurveOfGrowth()` method.

```
# Java style
print myPhot.getGrowthRadius()
# Jython style
print myPhot.growthRadius
```

Example 4.123. Getting the growth radius for the results of the aperture photometry task.

16. `getGrowthFlux()`

    Returns a `Double1d` with the values in the *Growth flux* column of the curve of growth table dataset returned by the `getCurveOfGrowth()` method.

```
# Java style
print myPhot.getGrowthFlux()
# Jython style
```

```
print myPhot.growthFlux
```

Example 4.124. Getting the growth flux column of the results of the aperture photometry task.

17. `getSkyIntensityPlot()`

    Returns the curve in the sky intensity plot as a table dataset. The table dataset has two columns: *Sky radius* and *Sky intensity* .

```
# Java style
myTable = myPhot.getSkyIntensityPlot()
# Jython style
myTable = myPhot.skyIntensityPlot
```

Example 4.125. Getting the intensity plot as a table dataset.

18. `getSkyIntensityRadius()`

    Returns a `Double1d` with the values in the *Sky radius* column of the sky intensity table dataset returned by the `getSkyIntensityPlot()` method.

```
# Java style
print myPhot.getSkyIntensityRadius()
# Jython style
print myPhot.skyIntensityRadius
```

Example 4.126. Getting the sky intensity radius of the intensity plot.

19. `getSkyIntensity()`

    Returns a `Double1d` with the values in the *Sky intensity* column of the sky intensity table dataset returned by the `getSkyIntensityPlot()` method.

```
# Java style
print myPhot.getSkyIntensity()
# Jython style
print myPhot.skyIntensity
```

Example 4.127. Getting the sky intensity values from the intensity plot as a Double1d.

Note

The aperture photometry task offers a recentering option, but the results are not always accurate. If you do not know the exact coordinates of your source, use the `sourceFitter` task before running the photometry task.

4.21.3.2. Rectangular sky aperture photometry (rectangularSkyAperturePhotometry)

The immediate neighbourhood of the target is not always the best location to estimate the sky. In these cases you can take a rectangular region further away from the target. This is known as **rectangular sky aperture photometry** . You can do it in HIPE with the `rectangularSkyAperturePhotometry` task.

Via the GUI

In the same way as with annular sky aperture photometry, you can select the object with one click or give the coordinates explicitly. Click and drag your mouse pointer on the image to select a rectangular aperture. Following the calculation for the first position, you can use the same rectangular box for the sky and choose a new object with a further single click on the image.

The result product has the same structure as the annular photometry result product, except that the sky intensity plot is missing.

Via the command line

To perform aperture photometry from the command line you must specify the following input parameters:

- The image (`image`).
- The position of the target centre, in pixels (`centerX` and `centerY`) or sky coordinates (`centerRa` and `centerDec`).
- The radius of the target, in pixels (`radiusPixels`) or in arcseconds (`radiusArcsec`).
- The position of the corner of the background area with the smallest coordinate values, in pixels (`minX` and `minY`) or sky coordinates (`minRa` and `minDec`).
- The width and height of the background area in pixels (`widthPixels` and `heightPixels`) or arcseconds (`widthArcsec` and `heightArcsec`).
- Whether fractional pixels are to be used (`fractional`). It can be `True` (1) or `False` (0), `True` by default.
- The sky estimation algorithm (`algorithm`). This is an integer with the following possible values: 0 for average, 1 for median, 2 for mean-median, 3 for synthetic mode and 4 for the algorithm used by Daophot.

The following example shows how to perform rectangular sky aperture photometry:

```
# The target centre is specified in pixel coordinates, the target radius in pixels
photPixel = rectangularSkyAperturePhotometry(image = myImage, centerX = 501.0,\
 centerY = 266.0, radiusPixels = 5.0, minX = 553.0, minY = 132.0, \
 widthPixels = 120.0, heightPixels = 47.0, algorithm = 4)

# The target centre is specified in sky coordinates, the target radius in arcseconds
photSky = rectangularSkyAperturePhotometry(image = myImage, \
 centerRa = "02:00:34.388", centerDec = "-22:25:59.87", radiusArcsec = 5.04, \
 minRa = "02:00:38.179", minDec = "-22:28:14.89", widthArcsec = 120.96, \
 heightArcsec = 47.376)
```

Example 4.128. Performing rectangular sky aperture photometry.

Note

The target centre and the corner of the rectangle with smallest coordinate values must be specified in the same coordinates, either pixel or sky.

You can choose the kind of pixels and the sky estimation algorithm in the same way as for the `annularSkyAperturePhotometry` task.

To inspect the output product via the command line, you can use the same methods described in Procedure 4.7 for the `annularSkyAperturePhotometry` task, except for those referring to the annular sky aperture. Further methods exclusive to the output of `rectangularSkyAperturePhotometry` are shown in the following table:

Procedure 4.8. Available methods for the output of the `rectangularSkyAperturePhotometry` task.

1. `getWidthPixels()`

   Returns the width of the rectangle in pixels. The `getHeightPixels()` method to get the height of the rectangle is also available.

   ```
   # Java style
   print myPhot.getWidthPixels()
   # Jython style
   ```

```
print myPhot.widthPixels
```

Example 4.129. Getting the width in pixels of the rectangular aperture.

2. `getWidthArcsec()`

   Returns the width of the rectangle in arcseconds. The `getHeightArcsec()` method to get the height of the rectangle is also available.

```
# Java style
print myPhot.getWidthArcsec()
# Jython style
print myPhot.widthArcsec
```

Example 4.130. Getting the width in arcseconds of the rectangular aperture.

3. `getUpperLeftCornerPixelCoordinates()`

   Returns the pixel coordinates of the upper left corner of the rectangle as a `Double1d` . The upper left corner is the corner with lowest pixel coordinates.

```
# Java style
print myPhot.getUpperLeftCornerPixelCoordinates()
# Jython style
print myPhot.upperLeftCornerPixelCoordinates
```

Example 4.131. Getting the upper left corner of the rectangular aperture in pixel coordinates.

4. `getUpperLeftCornerSkyCoordinates()`

   Returns the sky coordinates of the upper left corner of the rectangle as a `String1d` . The upper left corner is the corner with lowest pixel coordinates.

```
# Java style
print myPhot.getUpperLeftCornerSkyCoordinates()
# Jython style
print myPhot.upperLeftCornerSkyCoordinates
```

Example 4.132. Getting the upper left corner of the rectangular aperture in sky coordinates.

4.21.3.3. Fixed sky aperture photometry (fixedSkyAperturePhotometry)

If both the annular and rectangular sky aperture methods fail to provide a meaningful estimate of the background, you can provide a fixed background value.

Via the GUI

Use `fixedSkyAperturePhotometry` to provide a **fixed sky value** . Executing the task and inspecting the results is done in the same way as for the other types of photometry.

Via the command line

To perform aperture photometry from the command line you must specify the following input parameters:

- The image (`image`).
- The position of the target centre, in pixels (`centerX` and `centerY`) or sky coordinates (`centerRa` and `centerDec`).
- The radius of the target, in pixels (`radiusPixels`) or in arcseconds (`radiusArcsec`).

- The sky intensity value (`sky`).
- Whether fractional pixels are to be used (`fractional`). It can be `True` (1) or `False` (0), `True` by default.

The following example shows how to perform fixed sky aperture photometry:

```
# The target centre for PACS observation 1342184579 is specified in pixel
 coordinates, the target
# radius in pixels
photPixels = fixedSkyAperturePhotometry(image = myImage, centerX = 159.0, \
 centerY = 150.0, radiusPixels = 5.0, sky = 48.0)
# The target centre for PACS observation 1342184579 is specified in sky coordinates,
 the target radius in arcsec
photSky = fixedSkyAperturePhotometry(image = myImage, centerRa = "15:31:24.91",\
 centerDec = "77:20:57.97"", radiusArcsec = 5.04, sky = 48.0)
```

Example 4.133. Performing fixed sky aperture photometry.

Note

The target radius can only be specified if the image has a valid WCS and the pixel scaling is the same in both directions.

4.21.3.4. Aperture correction

Aperture photometry measures the flux within a finite, relatively small aperture. The total flux however is distributed in a much larger area well outside the aperture. To account for this missing flux you need to apply a correction factor to the flux values. Such correction factors are determined through careful signal-to-noise measurements of bright celestial standards and are available as calibration files in HIPE.

For PACS. Run the aperture correction task as in the following example:

```
result_apcor = photApertureCorrectionPointSource(apphot=myResult, band="blue", \
calTree=calTree, responsivityVersion=6)
```

Example 4.134. Running the aperture photometry correction task for point sources.

The *`myResult`* variable is the output of the aperture photometry task. The *`band`* parameter determines the filter. Because the aperture correction changes when there is an update in responsivity calibration, you must specify which responsivity calibration was used when your image was processed.

You can examine the task output in the same way as the output of the aperture photometry tasks.

For more information on the `photApertureCorrectionPointSource` task, see the corresponding entry in the *User's Reference Manual* .

For SPIRE. Aperture correction is mentioned in the *SPIRE Data Reduction Guide* , Section 5.7 . Search for *aperture correction* in that section.

4.22. Comparing PSFs to point source profiles

In many cases you have to compare the PSF of individual sources with a model PSF. The model PSF can be built from the sources in the image, or taken from models or dedicated observations of the instrument teams. In this section we describe how to work with PACS and SPIRE PSFs. We give an example script to rotate and match the PACS PSFs to an observation, and then to compare the two in a few ways.

Note that the PSF of the Herschel instruments depends on the wavelength and also, in many cases, on the observing mode (such as the scan speed for PACS), as well as possibly the flux of the source. For more information, see the instrument web pages for PACS and SPIRE .

The PSF/beams can be obtained from the public Herschel TWiki for PACS and SPIRE , or from re-pipelined observations.

You can compute and compare the curve-of-growth (*EEF curves* in PACS terminology) and/or radial profiles of your source and the PSF to see whether your source is consistent with being a point source. It is unlikely that your source will exactly *match* the model PSF, but you can look for consistency. You can also overplot the PSF image on your source to compare their relative sizes. Finally you can attempt point-source fitting photometry, **but** noting that we provide no task to do this: we simply show you how to subtract the scaled PSF from the astronomical source. PSF-fitting photometry is unlikely, at this point in time, to give very accurate results, since the profile of Herschel's point-spread-function during the observations that the PSF maps were made from, is not going to be exactly the same as the profile of Herschel's point-spread-function during your observations. However, see the instrument websites to learn more, as this situation will improve with time. Rather than PSF-subtraction, done, for example, to disentangle a point+extended source, you may be better off fitting your point source with a Gaussian (e.g. sourceFitting) and subtracting that.

For PACS it is recommended that rather than taking the PSFs from the PACS public TWiki, you instead reduce the observation the PSF was taken from yourself : you can find the obsid in this report . Your reduction should use the same pipeline parameters, as far as possible, as was done for your astronomical source. This is because the way you reduce the data and create the maps has an effect on the effective PSF; the profile of a true point source will not be exactly the same if you change the parameters of some of the pipeline tasks.

Tip

A paper (Popesso et al. 2012) is available on the effect of pipelining and mapmaking on the PACS beam profile. A technical note version is on the PACS public TWiki.

Required before you follow this script:

- Your maps. You can do what we describe here on the scan and cross-scan separately or after being combined. For fast scan speed observations, however, see below.
- Your own version of the pipelined PSF observations, although for the demonstration here we take them from the public TWiki.
- For SPIRE we assume you have observations that are also using the scanmapping AOTs.
- You can do most of this on a HIPE installation with either the PACS or SPIRE component, however some of the tasks will require one or the other. You will be warned when appropriate.

You will need to do some of your own scripting. There are several ways to do anything in HIPE, so you may want to use a different method you know about. Also, if you want to run this script on several observations at once, you may want to wrap a loop around it.

What you will be doing:

- Downloading the maps from the HSA, because ideally you would re-reduce the PSF obsids for PACS yourself, instead of using the PSF files mentioned above.
- Rotating by the position angle (meta data `posAngle`). This is because the beams are not symmetric, and in their wings the profile moves with the position angle of the telescope. Hence, to match the shape of the beam to the shape of your source they should both be at the same position angle.
- Matching the WCS of the beam to that of your source, so you can compare them on their WCSs. For this you need to measure the position of the source and the beam.

- Overplotting contours of one on the greyscale display of the other.
- Creating EEF curves/radial profiles, and comparing those for the beam and the astronomical source.
- Scaling and subtracting the beam map from the astronomical map, to see at what point your point source disappears.

Note

Rotating the beam involves mathematically manipulating your data, that is, resampling the spatial plane. This will never reproduce exactly what real life would have given you. The rotation task will perform better if you have smaller map pixel sizes, although the map pixel sizes you can meaningfully adopt will depend on the depth of your data.

Note

To compare the beam profiles to your astronomical profiles, you will need to be comparing maps of the same pixel sizes as each other, for each band separately. The profile depends on the spatial sampling!

You must be extra careful when dealing with **fast scan speed PACS maps**. At the fast scan speed the PSF is distorted along the scanning direction. By *scanning direction* we mean the scan angle, which in HSPOT is called *orientation angle*. By default this is 45 and 135 for scan and cross-scan, and in the metadata of your observation is called `mapScanAngle`. For more detail see this technical report, for example Figure 1 to compare the PSF of a slow and fast scanning speed map. Because of this, if you combine the scan and cross-scan data to make a final mosaic/map you will get a sort of cross-shaped PSF (by how much depends on the strength of your source and the depth of your background: faint sources and noisy backgrounds will hide the fainter parts of the PSF). Your choices here are to work on your astronomical data scan and cross-scan maps separately (easiest option) or if you want to combine them (e.g. to get a better SNR) then you need to create a similarly distorted and combined PSF map: pipeline the PSF obsids (as recommended), copy the map twice, one rotated by the scan angle and one by the cross-scan angle, and then use the `mosaic` task to combine those maps. This will also give then a sort of cross-shaped PSF. Note that we say here to "rotate by the scan/cross-scan angle and combine". Normally one should rotate by the position angle, since - as explained in the technical report - the shape of the PSF (trilobal) rotates with the PA (i.e. it is "fixed" to the telescope). For slow and medium scan speed observations you only need to rotate the PSF map by the position angle of your observations (there is no distortion along the scanning angle), whether you work on the astronomical scan and cross-scan maps separately or combined. But for fast scan speed maps you will need to see which is more important to the profile of your point source for your observation - the scan angle or the position angle - and then decide which to rotate by. This will vary with band, as the scan-angle distortion is stronger in the blue than the red. For not-bright sources with not very deep backgrounds, the distortion along the scanning angle is probably more influential to the shape of your PSF than the trilobal shape of the beam. It is with this in mind that we say to rotate by the scan angle before mosaicking the scan+cross scan maps, or before comparing them individually. But if you deem that the trilobal beam shape is more important to match to your astronomical maps than the scanning direction distortion, then instead you only need to rotate a single PSF map by the position angle, and compare that single map to your scan, cross-scan, or combined scan+cross-scan map.

4.22.1. Setting up and getting the data

The first thing is a few imports:

```
import math
from math import *
from java.awt.Color import GREEN
from java.awt.Color import BLUE
```

Example 4.135. Importing the required classes for the PSF-point source comparison example.

Then you need to get the PSFs. For PACS you want to match the scan speed (meta data "mapScanSpeed") of your astronomical observations (10=slow, 20=medium, 60=fast) and you could try

also to match the scan angle (+63/+42/-42), although note that for all the PSFs on the PACS wiki, the position angle has been set to 0 (and unfortunately the `posAngle` meta data are not given in the maps). For that reason, in this example *regular observations from the Herschel Science Archive will be used* . Note that the beam units are Jy and they are normalised to have a total flux of 1Jy. So:

PACS

```
# For PACS, extract the PSF from calibration observation 1342186136
calObs = getObservation(obsid = 1342186136, useHsa = True)
bm = calObs.refs["level2"].product.refs["HPPPMAPB"].product
pabm = 0.0
```

Example 4.136. Reading the data from a PACS calibration observation (PSF-point source comparison).

SPIRE

```
calTree=spireCal(calTree="spire_cal")
band="PLW" # or "PSW" or "PMW"
bm=calTree.phot.getBeamProf(band)

pabm = bm.getMeta()["posAngle"].value
```

Example 4.137. Retrieving the beam profile from the latest SPIRE calibration (PSF-point source comparison).

Note

For SPIRE the units of the beams are Jy/beam, but those from the wiki and those from the calibration file have different flux - the latter have been scaled to give a peak flux of 1. Those from the calibration file may not have the units in it, but they are Jy/beam. These PSFs have not been rotated to 0 PA so you need to extract the position angle as well.

Finally you need the map of your source, which we will call "src". For example, you can use 1342184579, but note that this is a chop/nod observation and doesn't include `mapScanSpeed` nor `mapScanAngle` metadata. From your map, if a PACS observation you need your position angle, scan speed and scan angle, and for SPIRE the position angle:

```
srcObs = getObservation(obsid = 1342184579, useHsa = True)
src = srcObs.refs["level2"].product.refs["HPPPMAPB"].product
pa    = src.getMeta()["posAngle"].value
# uncomment the next two lines if your observation is not chop/nod
# speed = src.getMeta()["mapScanSpeed"].value
# sa    = src.getMeta()["mapScanAngle"].value
```

Example 4.138. Retrieving the necessary properties for PACS and SPIRE (PSF-point source comparison).

it is also worth creating a variable for the obsid and the band or filter (R/B/G" for PACS red, blue, green, and "PLW/PMW/PSW" for the SPIRE long, mid, short bands) so you can attach this information to your output:

```
obsid = 1342184579
band  = "B"
```

Example 4.139. Creating auxiliary variables with the obsid and band (PSF-point source comparison).

4.22.2. Rotate the PSF and match it to the astronomical source

First thing you need to do is to rotate the PSF/beam by the PA or, if you are working on PACS fast scan speed maps maybe by the SA (scan angle), of your observation. For PACS PSF maps taken from

the wiki you only need to rotate by the PA value (as the PA of the PSF map is 0), but for SPIRE you need to rotate by the difference between the PA of the beam and your source:

```
angle=pa-pabm-180
bmr=rotate(image=bm, angle=angle, subsampleBits=32,\
  interpolation=rotate.INTERP_BICUBIC)
# optional information
bmr.setDescription("beam rotated for "+str(obsid)+", "+band)
Display(bm)
Display(bmr)
print "WCS of rotated beam:",bmr.wcs
```

Example 4.140. Rotating the PSF and matching to the source (PSF-point source comparison).

See the documentation for `rotate` in *HCSS User's Reference Manual* to learn more about this task.

Note

The rotate task will rotate the WCS of a map. When you Display the rotated map it will have a different orientation to the original. HOWEVER, when you are rotating in order to match an angle, you need to rather rotate the **source** and keep the WCS the same. To do this, but while still using "rotate", you then need to copy the original map WCS over to the new. This is perfectly OK to do as you do not change the map pixel sizes by rotating, however this will also usually slightly shift the source position.

```
wcs=bm.getWcs()
bmr.setWcs(wcs)
bmr.setDescription("beam rotated for "+str(obsid)+", "+band\
  +" with original wcs")
print "WCS of rotated beam with old WCS imposed:",bmr.wcs
```

Example 4.141. Updating the Wcs information (PSF-point source comparison).

Now match the WCS of the beam and the astronomical source, so the sources lie at the same RA, DEC on both. For this, first you have to locate the exact source position. Here we use the sourceFitter for that:

```
xpixstep=bmr["image"].meta["cdelt1"].value*3600.
ypixstep=bmr["image"].meta["cdelt2"].value*3600.
if (xpixstep<0): xpixstep=-1*xpixstep
if (ypixstep<0): ypixstep=-1*ypixstep
```

Example 4.142. Matching the WCS of the beam and the source (PSF-point source comparison).

where xpixstep and ypixstep are the pixel sizes in arcsec. The final two if lines are needed to get rid of negative values.

Then you get the RA, Dec coordinates of your sources (for the sourceFitter to begin its search) in decimal degrees from the meta data (or you can work it out yourself):

```
RA=bmr.meta["raNominal"].value
DEC=bmr.meta["decNominal"].value
```

Example 4.143. Retrieving the RA and declination in decimal degrees (PSF-point source comparison).

...and in pixel values:

```
cxpix=bmr.wcs.getPixelCoordinates(RA,DEC)[1]
cypix=bmr.wcs.getPixelCoordinates(RA,DEC)[0]
```

Example 4.144. Retrieving the X, Y coordinates in pixels (PSF-point source comparison).

Or set the coordinates to be the centre of the map (where the source most likely is):

```
nxpix=bmr.wcs.naxis1
nypix=bmr.wcs.naxis2
cxpix=nxpix/2.
cypix=nypix/2.
```

Example 4.145. Setting the coordinates to the centre of the map (PSF-point source comparison).

Now you have to find the exact centre of your PSF. For that you have to first set a maximum shift to search for source in, around cy|xpix you may want to make this generous for the rotated beam map, if the source is very far off the centre. You also need to run this in a loop increasing the search radius each time. You can change the boxsize values 5,60 to something else if you like.

```
maxshift = 10.
print "   Fitting rotated beam ..."
for boxsize in range(5,60):
  try:
    minX=cxpix-boxsize/2.
    minY=cypix-boxsize/2.
    sfit   = sourceFitting(elongated=True,slope=False,image=bmr,\
            minX=minX,minY=minY,width=boxsize,height=boxsize)
    print "      boxsize "+str(boxsize)+"...success!"
    poo=1 # seems to be necessary to make this stop when it has success
  except:
    print "      boxsize "+str(boxsize)+"...failed"
    cxpixfit=-1  # or some other value to indicate failure
    cypixfit=-1  # eg maybe the centre of the map, ...
    pixfit_RA=-1
    pixfit_Dec=-1
    poo=0
  else:
    # upon success, grab the results of the sourceFitting
    cxpixfit   = sfit.getCenterX()
    cypixfit   = sfit.getCenterY()
    pixfit_RA  = sfit.getCenterRA()
    pixfit_Dec = sfit.getCenterDec()
    # very occasionally sourceFitting gives nonsence results, so:
    if ((ABS(cxpix-cxpixfit) > maxshift) or (ABS(cypix-cypixfit) > maxshift)):
      print "         ...however, source found too far from original coords"
      print"             so setting to -1,-1"
      print "            orig coords: "+str(cxpix)+", "+str(cypix)+\
            " found coords: "+str(bm_cxpixfit)+", "+str(bm_cypixfit)
      cxpixfit=-1 # or some other value to indicate failure
      cypixfit=-1 # eg maybe the centre of the map, ...
      pixfit_RA=-1
      pixfit_Dec=-1
      poo=0
    else:
      bm_cxpixfit   = sfit.getCenterX()
      bm_cypixfit   = sfit.getCenterY()
      bm_pixfit_RA  = sfit.getCenterRA()
      bm_pixfit_Dec = sfit.getCenterDec()
      sigma_x    = sfit.getSigmaXPixels()
      sigma_y    = sfit.getSigmaYPixels()
      bm_fwhm_x     = round(abs(2.*SQRT(2.*LOG(2))*sigma_x*xpixstep),1)
      bm_fwhm_y     = round(abs(2.*SQRT(2.*LOG(2))*sigma_y*ypixstep),1)
      print "Beam details:"
      print "   beam position: pixel:",bm_cxpixfit, bm_cypixfit
      print "                  coordinates:",bm_pixfit_RA, bm_pixfit_Dec
      print "   beam fwhm x,y:",bm_fwhm_x,bm_fwhm_y
      poo=1
    if (poo == 1): break
    pass
```

Example 4.146. Finding the exact centre of the PSF (PSF-point source comparison).

You need to do exactly the same for your astronomical source.

```
xpixstep=src["image"].meta["cdelt1"].value*3600. # pixel size in arcsec
```

```
ypixstep=src["image"].meta["cdelt2"].value*3600.
if (xpixstep<0): xpixstep=-1*xpixstep # avoid negative values
if (ypixstep<0): ypixstep=-1*ypixstep
# starting RA, Dec values in decimal degrees...from the meta data
# or you can work it out yourself
# decimal values are necessary
RA=src.meta["raNominal"].value
DEC=src.meta["decNominal"].value
# ...and in pixel values
cxpix=src.wcs.getPixelCoordinates(RA,DEC)[1]
cypix=src.wcs.getPixelCoordinates(RA,DEC)[0]
maxshift = 10.
print "   Fitting astronomical source ..."
for boxsize in range(5,60):
  try:
    minX=cxpix-boxsize/2.
    minY=cypix-boxsize/2.
    sfit   = sourceFitting(elongated=True,slope=False,image=src,\
            minX=minX,minY=minY,width=boxsize,height=boxsize)
    print "      boxsize "+str(boxsize)+"...success!"
    poo=1 # seems to be necessary to make this stop when it has success
  except:
    print "      boxsize "+str(boxsize)+"...failed"
    cxpixfit=-1 # or some other value to indicate failure
    cypixfit=-1 # eg maybe the centre of the map, ...
    pixfit_RA=-1
    pixfit_Dec=-1
    poo=0
  else:
    # upon success, grab the results of the sourceFitting
    cxpixfit   = sfit.getCenterX()
    cypixfit   = sfit.getCenterY()
    pixfit_RA  = sfit.getCenterRA()
    pixfit_Dec = sfit.getCenterDec()
    # very occasionally sourceFitting gives nonsence results, so:
    if ((ABS(cxpix-cxpixfit) > maxshift) or (ABS(cypix-cypixfit) > maxshift)):
      print "         ...however, source found too far from original coords,"
      print "         so setting to -1,-1"
      print "            orig coords: "+str(cxpix)+","+str(cypix)+\
            " found coords: "+str(bm_cxpixfit)+", "+str(bm_cypixfit)
      cxpixfit=-1 # or some other value to indicate failure
      cypixfit=-1 # eg maybe the centre of the map, ...
      pixfit_RA=-1
      pixfit_Dec=-1
      poo=0
    else:
      src_cxpixfit   = sfit.getCenterX()
      src_cypixfit   = sfit.getCenterY()
      src_pixfit_RA  = sfit.getCenterRA()
      src_pixfit_Dec = sfit.getCenterDec()
      sigma_x    = sfit.getSigmaXPixels()
      sigma_y    = sfit.getSigmaYPixels()
      src_fwhm_x     = round(abs(2.*SQRT(2.*LOG(2))*sigma_x*xpixstep),1)
      src_fwhm_y     = round(abs(2.*SQRT(2.*LOG(2))*sigma_y*ypixstep),1)
      print "Source details:"
      print "   source position: pixel:",src_cxpixfit, src_cypixfit
      print "                  coordinates:",src_pixfit_RA, src_pixfit_Dec
      print "   source fwhm x,y:",src_fwhm_x,src_fwhm_y
      poo=1
    if (poo == 1): break
    pass
```

Example 4.147. Find the exact centre of the point source (PSF-point source comparison).

Now move the WCS of the PSF map so that the pixel coordinates where the beam is located are the WCS coordinates of the astronomical source. Then display the astro map to check:

```
bmr.getWcs().setCrval1(src_pixfit_RA)
bmr.getWcs().setCrval2(src_pixfit_Dec)
bmr.getWcs().setCrpix1(bm_cxpixfit+1.0)
bmr.getWcs().setCrpix2(bm_cypixfit+1.0)
```

```
print "WCS of rotated beam with old WCS imposed and shifted to the astro source
 position:"\
  ,bmr.wcs
d=Display(src)
d.setTitle(str(obsid)+","+band)
d.setCutLevelsPercentage(99.0)
d.setTitle(str(obsid)+","+band+",rotated by "+str(angle))
# red circle where the astro source was found to be
d.addCircle(src_cypixfit,src_cxpixfit,2,2,java.awt.Color.red)
# If you want to add contours of the beam and the astro source to the greyscale map"
# Beam:
# ->must be as many as there are contour levels
cs=[BLUE,BLUE,BLUE,BLUE,BLUE]
# ->change the min and max according to the flux range in your map
contoursb = automaticContour(image=bmr, levels=5, min=0.1, max=0.5, distribution=0,\
   colors=cs)
d.addWcsImageContour(contoursb)
# Astro source:
#cs=[GREEN,GREEN,GREEN,GREEN,GREEN]
#contourss = automaticContour(image=src, levels=5, min=0.002, max=0.004, \
#  distribution=0, colors=cs)
#d.addWcsImageContour(contourss)
d.setZoomFactor(16)
```

Example 4.148. Matching the WCS of the PSF to the coordinates of the source (PSF-point source comparison).

4.22.3. EEF Curves

One way to get curves of growth is to use the aperture photometry task, which produces one as a result. You have the choice of annular, fixed, and rectangular aperture photometry (read the DAG or the URM entries for each of the tasks to learn more). A better way is to do aperture photometry on increasing radii and build your own EEF curve. This way you have control over the grid to build it on. The annularAperturePhotometry task will allow you to centroid on an approximate set of coordinates for the source, however this does not work as well as the coordinates you get from sourceFitting; so use those gotten using the script in the previous section. You will need to enter a size for the aperture on the source and the sky annuli. You should try to get the sky aperture radii correct, but note that they cannot be smaller than the "raper" value. If you want to build up EEF curves that are longer than the aperture from which you get the sky flux from, you need to instead use the fixedSkyAperturePhotometry task without sky subtraction, and then scale the EEF curves to their values in the sky radii range.

If you are interested in the aperture photometry, you will need to read the point source photometry guides for PACS and SPIRE , to find the aperture corrections and recommended aperture sizes.

Begin with some definitions:

```
rskyin_src  = 60
rskyout_src = 75
```

Example 4.149. Defining the coordinates for building the EEF curve.

Indicating the coordinates you want to build the EEF curve around, i.e. the RA and DEC of the source, as worked out in the previous section. These will be for source and beam: src_cxpixfit, src_cxpixfit, bm_cxpixfit, src_cypixfit.

Now build up a curve of growth. First you need to define the apertures:

```
rapers=Double1d.range(41)*2
```

Example 4.150. Creating an array of apertures.

then remove the first, as you cannot have an aperture of 0:

```
rapers=rapers[1:]
```

Example 4.151. Removing the first aperture from the array.

create a variable to store the fluxes (one per aperture, for astro source and PSF):

```
fluxes=Double2d(len(rapers),2)
```

Example 4.152. Creating a variable for storing two arrays of fluxes.

Start the photometry of your source:

```
cnt=0
for raper in (rapers):
  apphot = fixedSkyAperturePhotometry(image=src,centroid=False,\
           centerX=round(src_cxpixfit,3),centerY=round(src_cypixfit,3),\
           radiusArcsec=raper,sky=0)
  fluxes[cnt,0]=apphot["Results table"]["Total flux"].data[1]
  cnt+=1
```

Example 4.153. Performing the point source aperture photometry for every aperture in the array.

and your PSF:

```
cnt=0
for raper in (rapers):
  apphot = fixedSkyAperturePhotometry(image=bmr,centroid=False,\
           centerX=round(bm_cxpixfit,1),centerY=round(bm_cypixfit,1),\
           radiusArcsec=raper,sky=0)
  fluxes[cnt,1]=apphot["Results table"]["Total flux"].data[1]
  cnt+=1
```

Example 4.154. Performing the PSF aperture photometry for every aperture in the array.

Note

`apphot["Results table"]["Total flux"].data[1]` will get you the total flux in the aperture, sky subtracted; if you want the average flux in the aperture (which, however, will not account for any NaN or 0 flux pixels in your aperture) then you can grab `apphot["Results table"]["Intensity per pixel].data[1]` instead

Note

The apertures will have different scalings because the fluxes of your astronomical source and the PSF will be different. If you want to scale you can do so on e.g. the peak flux or e.g. the flux between certain radii. Here are two possible following ways:

scale to peak (ie at the minimum aperture):

```
fluxes[:,0]=fluxes[:,0]/fluxes[0,0]
fluxes[:,1]=fluxes[:,1]/fluxes[0,1]
```

Example 4.155. Scaling the fluxes of the PSF and point source to peak (min aperture).

you also have to define the aperture for the beam model,

```
# The aperture for the bm is assumed to be the same
rskyin_bm = 60
rskyout_bm = 75
```

Example 4.156. Defining apertures for the beam model.

scale to median in the sky area:

```
idx=rapers.where((rapers>rskyin_src)&(rapers<rskyout_src))
fluxes[:,0]=fluxes[:,0]/MEAN(fluxes[idx,0])
idx=rapers.where((rapers>rskyin_bm)&(rapers<rskyout_bm))
fluxes[:,1]=fluxes[:,1]/MEAN(fluxes[idx,1])
```

Example 4.157. Scaling the fluxes of the PSF and point source to median.

Now you can plot your results:

```
p=PlotXY(titleText="My EEF for obsid "+str(obsid)+", "+band)
p.addLayer(LayerXY(rapers,fluxes[:,0],line=1,color=java.awt.Color.black))
p.addLayer(LayerXY(rapers,fluxes[:,1],line=3,color=java.awt.Color.red))
p[1].setName("beam")
p[0].setName("astro source")
p.xaxis.title.text="arcsec"
p.yaxis.title.text="flux"
p.getLegend().setVisible(True)
```

Example 4.158. Plotting the results of this comparative aperture photometry.

then you can compute the FWHM of your EEF curve. This is just for information, it is not a 100% good measure of the spatial extent of your source (that depends on how far out you have been able to go, because the beam actually extends quite far: for PACS this is past 60".) First measure the FWHM from the astro source:

```
interpgrid = Double1d.range(78*8)/8.+2
interp = CubicSplineInterpolator(rapers,fluxes[:,0])
eef_cont=interp(interpgrid)
idx=eef_cont.where((eef_cont >=0.47) & (eef_cont <=0.53))
sey=interpgrid[idx]
fw_as=MEAN(sey)
```

Example 4.159. Measuring the FWHM of the point source using a cubic spline interpolator.

the third line is needed to pick out the FWHM.

Do the same for the PSF:

```
interp = CubicSplineInterpolator(rapers,fluxes[:,1])
eef_cont=interp(interpgrid)
idx=eef_cont.where((eef_cont >=0.47) & (eef_cont <=0.53))
sey=interpgrid[idx]
fw_bm=MEAN(sey)
```

Example 4.160. Measuring the FWHM of the PSF using a cubic spline interpolator.

and finally compare the results:

```
print "FWHM (arcsec):"
print "   astro source:",fw_as
print "     PSF source:",fw_bm
```

Example 4.161. Comparing both values for FWHM.

4.22.4. Measuring the sky background scatter on PACS and SPIRE maps

There is a way to work out the sky background value and the sky scatter value: measure the flux in circular apertures of the same aperture size you used when doing aperture photometry (using, e.g.

annularSkyAperturePhotometry) on your source. To be scriptable, we chose here apertures evenly located around a set radius away from your source.

Get the coordinates: either use src_cxpixfit, src_cypixfit, bm_cxpixfit, bm_cypixfit computed previously, or get them from the rotated beam maps, remembering that we recorded the source's position in the bmr's wcs:

```
RA=bmr.getWcs().getCrval1()
DEC=bmr.getWcs().getCrval2()
cxpixb=bmr.wcs.getPixelCoordinates(RA,DEC)[1] # pixel coordinates of the beam
cypixb=bmr.wcs.getPixelCoordinates(RA,DEC)[0]
cxpixs=src.wcs.getPixelCoordinates(RA,DEC)[1] # pixel coordinates of the astro
 source
cypixs=src.wcs.getPixelCoordinates(RA,DEC)[0]
```

Example 4.162. Measuring the sky background scatter of beam and astro source.

Define the radius about which you want to measure the sky background value and its scatter, between the apertures. You could e.g. use a value that is the sky aperture radius value recommended to do with photometry. The radius of the mini circular apertures to compute the sky value from should be the same value as used for your aperture photometry on your astronomical source:

```
rsky = 40
raper= 12
raperpix=raper/ypixstep
separation=rsky/ypixstep
```

Example 4.163. Defining recommended values for circular photometry.

Define the position of the mini apertures and display them on your source and perform the photometry:

```
ycos=COS(60.0/180.*Math.PI)
xsin=SIN(60.0/180.*Math.PI)
matrix = [[-xsin,-ycos],[-xsin,ycos],[0,-1],[0,1],[xsin,-ycos],[xsin,ycos]]
sum_bck=Double1d()
d=Display(src)
d.setTitle(str(obsid)+", "+band)
d.addCircle(cypixs,cxpixs,raperpix,1,java.awt.Color.blue) # aperture photometry
 radius
d.addCircle(cypixs,cxpixs,separation,1,java.awt.Color.red) # (aperture photometry)
 sky radius
d.setZoomFactor(12)

for spot in matrix:
  xpos=round(cxpixs+spot[0]*separation,3)
  ypos=round(cypixs+spot[1]*separation,3)
  d.addCircle(ypos,xpos,raperpix,1,java.awt.Color.white) # the sky apertures
  eaper = fixedSkyAperturePhotometry(image=src,centroid=False,\
          centerX=xpos,centerY=ypos,radiusArcsec=raper)
  sum_bck.append(eaper["Results table"]["Total flux"].data[0])
```

Example 4.164. Performing the aperture photometry for all the small spot apertures.

then clean your sample from the outliers and calculate the sky value:

```
clip=sum_bck.apply(Sigclip(nsigma=3.0,returnmode = Sigclip.RETURN_BOOL))
idx = clip.where(clip==False).toInt1d()
sum_bck_clip          = sum_bck[Selection(idx)]
avg_matrix_back       = MEAN(sum_bck_clip)
med_matrix_back       = MEDIAN(sum_bck_clip)
rms_matrix_back       = STDDEV(sum_bck_clip)
sky_bm = avg_matrix_back
print "PSF source sky values (mean: %f, median: %f, stddev: %f):"%\
  (avg_matrix_back,med_matrix_back,rms_matrix_back)
```

Example 4.165. Removing outliers from the sample.

A complete, self-contained example for PACS observation **1342184579** is given below:

```
import math
from math import *
from java.awt.Color import GREEN
from java.awt.Color import BLUE

# For PACS, extract the PSF from calibration observation 1342186136
calObs = getObservation(obsid = 1342186136, useHsa = True)
bm = calObs.refs["level2"].product.refs["HPPPMAPB"].product
pabm = 0.0

# Getting the observation with the source
srcObs = getObservation(obsid = 1342184579, useHsa = True)
src = srcObs.refs["level2"].product.refs["HPPPMAPB"].product

pa    = src.getMeta()["posAngle"].value

# Ancillary variables
obsid = 1342184579
OBSID = [obsid]
band  = "B"


angle=pa-pabm-180
bmr=rotate(image=bm, angle=angle, subsampleBits=32,\
  interpolation=rotate.INTERP_BICUBIC)
# optional information
bmr.setDescription("beam rotated for "+str(obsid)+", "+band)
Display(bm)
Display(bmr)
print "WCS of rotated beam:",bmr.wcs

xpixstep=bmr["image"].meta["cdelt1"].value*3600.
ypixstep=bmr["image"].meta["cdelt2"].value*3600.
if (xpixstep<0): xpixstep=-1*xpixstep
if (ypixstep<0): ypixstep=-1*ypixstep

RA=bmr.meta["raNominal"].value
DEC=bmr.meta["decNominal"].value

cxpix=bmr.wcs.getPixelCoordinates(RA,DEC)[1]
cypix=bmr.wcs.getPixelCoordinates(RA,DEC)[0]

nxpix=bmr.wcs.naxis1
nypix=bmr.wcs.naxis2
cxpix=nxpix/2.
cypix=nypix/2.

maxshift = 10.
print "   Fitting rotated beam ..."
for boxsize in range(5,60):
  try:
    minX=cxpix-boxsize/2.
    minY=cypix-boxsize/2.
    sfit   = sourceFitting(elongated=True,slope=False,image=bmr,\
            minX=minX,minY=minY,width=boxsize,height=boxsize)
    print "      boxsize "+str(boxsize)+"...success!"
    poo=1 # seems to be necessary to make this stop when it has success
  except:
    print "      boxsize "+str(boxsize)+"...failed"
    cxpixfit=-1  # or some other value to indicate failure
    cypixfit=-1  # eg maybe the centre of the map, ...
    pixfit_RA=-1
    pixfit_Dec=-1
    poo=0
  else:
    # upon success, grab the results of the sourceFitting
    cxpixfit   = sfit.getCenterX()
    cypixfit   = sfit.getCenterY()
    pixfit_RA  = sfit.getCenterRA()
    pixfit_Dec = sfit.getCenterDec()
```

```
    # very occasionally sourceFitting gives nonsence results, so:
    if ((ABS(cxpix-cxpixfit) > maxshift) or (ABS(cypix-cypixfit) > maxshift)):
      print "         ...however, source found too far from original coords"
      print"             so setting to -1,-1"
      print "            orig coords: "+str(cxpix)+", "+str(cypix)+\
           " found coords: "+str(bm_cxpixfit)+", "+str(bm_cypixfit)
      cxpixfit=-1 # or some other value to indicate failure
      cypixfit=-1 # eg maybe the centre of the map, ...
      pixfit_RA=-1
      pixfit_Dec=-1
      poo=0
    else:
      bm_cxpixfit   = sfit.getCenterX()
      bm_cypixfit   = sfit.getCenterY()
      bm_pixfit_RA  = sfit.getCenterRA()
      bm_pixfit_Dec = sfit.getCenterDec()
      sigma_x    = sfit.getSigmaXPixels()
      sigma_y    = sfit.getSigmaYPixels()
      bm_fwhm_x     = round(abs(2.*SQRT(2.*LOG(2))*sigma_x*xpixstep),1)
      bm_fwhm_y     = round(abs(2.*SQRT(2.*LOG(2))*sigma_y*ypixstep),1)
      print "Beam details:"
      print "   beam position: pixel:",bm_cxpixfit, bm_cypixfit
      print "                  coordinates:",bm_pixfit_RA, bm_pixfit_Dec
      print "   beam fwhm x,y:",bm_fwhm_x,bm_fwhm_y
      poo=1
    if (poo == 1): break
    pass

xpixstep=src["image"].meta["cdelt1"].value*3600. # pixel size in arcsec
ypixstep=src["image"].meta["cdelt2"].value*3600.
if (xpixstep<0): xpixstep=-1*xpixstep # avoid negative values
if (ypixstep<0): ypixstep=-1*ypixstep
# starting RA, Dec values in decimal degrees...from the meta data
# or you can work it out yourself
# decimal values are necessary
RA=src.meta["raNominal"].value
DEC=src.meta["decNominal"].value
# ...and in pixel values
cxpix=src.wcs.getPixelCoordinates(RA,DEC)[1]
cypix=src.wcs.getPixelCoordinates(RA,DEC)[0]
maxshift = 10.
print "   Fitting astronomical source ..."
for boxsize in range(5,60):
  try:
    minX=cxpix-boxsize/2.
    minY=cypix-boxsize/2.
    sfit   = sourceFitting(elongated=True,slope=False,image=src,\
            minX=minX,minY=minY,width=boxsize,height=boxsize)
    print "      boxsize "+str(boxsize)+"...success!"
    poo=1 # seems to be necessary to make this stop when it has success
  except:
    print "      boxsize "+str(boxsize)+"...failed"
    cxpixfit=-1 # or some other value to indicate failure
    cypixfit=-1 # eg maybe the centre of the map, ...
    pixfit_RA=-1
    pixfit_Dec=-1
    poo=0
  else:
    # upon success, grab the results of the sourceFitting
    cxpixfit   = sfit.getCenterX()
    cypixfit   = sfit.getCenterY()
    pixfit_RA  = sfit.getCenterRA()
    pixfit_Dec = sfit.getCenterDec()
    # very occasionally sourceFitting gives nonsence results, so:
    if ((ABS(cxpix-cxpixfit) > maxshift) or (ABS(cypix-cypixfit) > maxshift)):
      print "         ...however, source found too far from original coords,"
      print "         so setting to -1,-1"
      print "            orig coords: "+str(cxpix)+","+str(cypix)+\
           " found coords: "+str(bm_cxpixfit)+", "+str(bm_cypixfit)
      cxpixfit=-1 # or some other value to indicate failure
      cypixfit=-1 # eg maybe the centre of the map, ...
      pixfit_RA=-1
```

```
      pixfit_Dec=-1
      poo=0
    else:
      src_cxpixfit   = sfit.getCenterX()
      src_cypixfit   = sfit.getCenterY()
      src_pixfit_RA  = sfit.getCenterRA()
      src_pixfit_Dec = sfit.getCenterDec()
      sigma_x    = sfit.getSigmaXPixels()
      sigma_y    = sfit.getSigmaYPixels()
      src_fwhm_x     = round(abs(2.*SQRT(2.*LOG(2))*sigma_x*xpixstep),1)
      src_fwhm_y     = round(abs(2.*SQRT(2.*LOG(2))*sigma_y*ypixstep),1)
      print "Source details:"
      print "   source position: pixel:",src_cxpixfit, src_cypixfit
      print "                  coordinates:",src_pixfit_RA, src_pixfit_Dec
      print "   source fwhm x,y:",src_fwhm_x,src_fwhm_y
      poo=1
    if (poo == 1): break
    pass

bmr.getWcs().setCrval1(src_pixfit_RA)
bmr.getWcs().setCrval2(src_pixfit_Dec)
bmr.getWcs().setCrpix1(bm_cxpixfit+1.0)
bmr.getWcs().setCrpix2(bm_cypixfit+1.0)
print "WCS of rotated beam with old WCS imposed and shifted to the astro source
 position:"\
  ,bmr.wcs
d=Display(src)
d.setTitle(str(obsid)+","+band)
d.setCutLevelsPercentage(99.0)
d.setTitle(str(obsid)+","+band+",rotated by "+str(angle))
# red circle where the astro source was found to be
d.addCircle(src_cypixfit,src_cxpixfit,2,2,java.awt.Color.red)
# If you want to add contours of the beam and the astro source to the greyscale map"
# Beam:
# ->must be as many as there are contour levels
cs=[BLUE,BLUE,BLUE,BLUE,BLUE]
# ->change the min and max according to the flux range in your map
contoursb = automaticContour(image=bmr, levels=5, min=0.1, max=0.5, distribution=0,\
   colors=cs)
d.addWcsImageContour(contoursb)
# Astro source:
#cs=[GREEN,GREEN,GREEN,GREEN,GREEN]
#contourss = automaticContour(image=src, levels=5, min=0.002, max=0.004, \
#  distribution=0, colors=cs)
#d.addWcsImageContour(contourss)
d.setZoomFactor(16)

# Plotting the EEF curves
# Recommended aperture size
rskyin_src  = 60
rskyout_src = 75
# The aperture for the bm is not given (will use the same)
rskyin_bm = 60
rskyout_bm = 75

rapers=Double1d.range(41)*2

rapers=rapers[1:]

fluxes=Double2d(len(rapers),2)

cnt=0
for raper in (rapers):
  apphot = fixedSkyAperturePhotometry(image=src,centroid=False,\
           centerX=round(src_cxpixfit,3),centerY=round(src_cypixfit,3),\
           radiusArcsec=raper,sky=0)
  fluxes[cnt,0]=apphot["Results table"]["Total flux"].data[1]
  cnt+=1

cnt=0
for raper in (rapers):
  apphot = fixedSkyAperturePhotometry(image=bmr,centroid=False,\
```

```
          centerX=round(bm_cxpixfit,1),centerY=round(bm_cypixfit,1),\
          radiusArcsec=raper,sky=0)
  fluxes[cnt,1]=apphot["Results table"]["Total flux"].data[1]
  cnt+=1
# Scaling the fluxes to median

idx=rapers.where((rapers>rskyin_src)&(rapers<rskyout_src))
fluxes[:,0]=fluxes[:,0]/MEAN(fluxes[idx,0])
idx=rapers.where((rapers>rskyin_bm)&(rapers<rskyout_bm))
fluxes[:,1]=fluxes[:,1]/MEAN(fluxes[idx,1])



p=PlotXY(titleText="My EEF for obsid "+str(obsid)+", "+band)
p.addLayer(LayerXY(rapers,fluxes[:,0],line=1,color=java.awt.Color.black))
p.addLayer(LayerXY(rapers,fluxes[:,1],line=3,color=java.awt.Color.red))
p[1].setName("beam")
p[0].setName("astro source")
p.xaxis.title.text="arcsec"
p.yaxis.title.text="flux"
p.getLegend().setVisible(True)

# interpgrid needs to be an array of points to evaluate with the interpolator
interpgrid = Double1d.range(78*8)/8.+2
interp = CubicSplineInterpolator(rapers,fluxes[:,0])
eef_cont=interp(interpgrid)
idx=eef_cont.where((eef_cont >=0.47) & (eef_cont <=0.53))
sey=interpgrid[idx]
fw_as=MEAN(sey)

interp = CubicSplineInterpolator(rapers,fluxes[:,1])
eef_cont=interp(interpgrid)
idx=eef_cont.where((eef_cont >=0.47) & (eef_cont <=0.53))
sey=interpgrid[idx]
fw_bm=MEAN(sey)

print "FWHM (arcsec):"
print "   astro source:",fw_as
print "     PSF source:",fw_bm

# Sky background scatter
RA=bmr.getWcs().getCrval1()
DEC=bmr.getWcs().getCrval2()
cxpixb=bmr.wcs.getPixelCoordinates(RA,DEC)[1] # pixel coordinates of the beam
cypixb=bmr.wcs.getPixelCoordinates(RA,DEC)[0]
cxpixs=src.wcs.getPixelCoordinates(RA,DEC)[1] # pixel coordinates of the astro
 source
cypixs=src.wcs.getPixelCoordinates(RA,DEC)[0]

rsky = 40
raper= 12
raperpix=raper/ypixstep
separation=rsky/ypixstep

ycos=COS(60.0/180.*Math.PI)
xsin=SIN(60.0/180.*Math.PI)
matrix = [[-xsin,-ycos],[-xsin,ycos],[0,-1],[0,1],[xsin,-ycos],[xsin,ycos]]
sum_bck=Double1d()
d=Display(src)
d.setTitle(str(obsid)+", "+band)
d.addCircle(cypixs,cxpixs,raperpix,1,java.awt.Color.blue) # aperture photometry
 radius
d.addCircle(cypixs,cxpixs,separation,1,java.awt.Color.red) # (aperture photometry)
 sky radius
d.setZoomFactor(12)

for spot in matrix:
  xpos=round(cxpixs+spot[0]*separation,3)
  ypos=round(cypixs+spot[1]*separation,3)
  d.addCircle(ypos,xpos,raperpix,1,java.awt.Color.white) # the sky apertures
  eaper = fixedSkyAperturePhotometry(image=src,centroid=False,\
         centerX=xpos,centerY=ypos,radiusArcsec=raper)
```

```
  sum_bck.append(eaper["Results table"]["Total flux"].data[0])

clip=sum_bck.apply(Sigclip(nsigma=3.0,returnmode = Sigclip.RETURN_BOOL))
idx = clip.where(clip==False).toInt1d()
sum_bck_clip          = sum_bck[Selection(idx)]
avg_matrix_back       = MEAN(sum_bck_clip)
med_matrix_back       = MEDIAN(sum_bck_clip)
rms_matrix_back       = STDDEV(sum_bck_clip)
sky_bm = avg_matrix_back
print "PSF source sky values (mean: %f, median: %f, stddev: %f):"%\
  (avg_matrix_back,med_matrix_back,rms_matrix_back)
```

Example 4.166. Performing PSF comparison for PACS point sources.

You can also check an equivalent example for SPIRE, using observation **1342190662** is given below:

```
import math
from math import *
from java.awt.Color import GREEN
from java.awt.Color import BLUE

# Retrieving an older calibration tree that had the posAngle value
calTree=spireCal(calTree="spire_cal_12_0")
band="PSW" # or "PSW" or "PMW"
bm=calTree.phot.getBeamProf(band)

pabm = bm.getMeta()["posAngle"].value

# Getting the observation with the source
srcObs = getObservation(obsid = 1342190662, useHsa = True)
src = srcObs.refs["level2"].product.refs["psrcPSW"].product

pa    = src.getMeta()["posAngle"].value

# Ancillary variables
obsid = 1342190662

angle=pa-pabm-180
bmr=rotate(image=bm, angle=angle, subsampleBits=32,\
  interpolation=rotate.INTERP_BICUBIC)
# optional information
bmr.setDescription("beam rotated for "+str(obsid)+", "+band)
Display(bm)
Display(bmr)
print "WCS of rotated beam:",bmr.wcs

xpixstep=bmr["image"].meta["cdelt1"].value*3600.
ypixstep=bmr["image"].meta["cdelt2"].value*3600.
if (xpixstep<0): xpixstep=-1*xpixstep
if (ypixstep<0): ypixstep=-1*ypixstep

# These data are not contained in the calibration tree

#RA=bmr.meta["raNominal"].value
#DEC=bmr.meta["decNominal"].value

#cxpix=bmr.wcs.getPixelCoordinates(RA,DEC)[1]
#cypix=bmr.wcs.getPixelCoordinates(RA,DEC)[0]

# Let's try with the centre method

nxpix=bmr.wcs.naxis1
nypix=bmr.wcs.naxis2
cxpix=nxpix/2.
cypix=nypix/2.

maxshift = 10.
print "   Fitting rotated beam ..."
for boxsize in range(5,60):
  try:
    minX=cxpix-boxsize/2.
```

```
    minY=cypix-boxsize/2.
    sfit   = sourceFitting(elongated=True,slope=False,image=bmr,\
            minX=minX,minY=minY,width=boxsize,height=boxsize)
    print "      boxsize "+str(boxsize)+"...success!"
    poo=1 # seems to be necessary to make this stop when it has success
  except:
    print "      boxsize "+str(boxsize)+"...failed"
    cxpixfit=-1  # or some other value to indicate failure
    cypixfit=-1  # eg maybe the centre of the map, ...
    pixfit_RA=-1
    pixfit_Dec=-1
    poo=0
  else:
    # upon success, grab the results of the sourceFitting
    cxpixfit   = sfit.getCenterX()
    cypixfit   = sfit.getCenterY()
    pixfit_RA  = sfit.getCenterRA()
    pixfit_Dec = sfit.getCenterDec()
    # very occasionally sourceFitting gives nonsence results, so:
    if ((ABS(cxpix-cxpixfit) > maxshift) or (ABS(cypix-cypixfit) > maxshift)):
      print "         ...however, source found too far from original coords"
      print"             so setting to -1,-1"
      print "            orig coords: "+str(cxpix)+", "+str(cypix)+\
            " found coords: "+str(bm_cxpixfit)+", "+str(bm_cypixfit)
      cxpixfit=-1 # or some other value to indicate failure
      cypixfit=-1 # eg maybe the centre of the map, ...
      pixfit_RA=-1
      pixfit_Dec=-1
      poo=0
    else:
      bm_cxpixfit   = sfit.getCenterX()
      bm_cypixfit   = sfit.getCenterY()
      bm_pixfit_RA  = sfit.getCenterRA()
      bm_pixfit_Dec = sfit.getCenterDec()
      sigma_x    = sfit.getSigmaXPixels()
      sigma_y    = sfit.getSigmaYPixels()
      bm_fwhm_x     = round(abs(2.*SQRT(2.*LOG(2))*sigma_x*xpixstep),1)
      bm_fwhm_y     = round(abs(2.*SQRT(2.*LOG(2))*sigma_y*ypixstep),1)
      print "Beam details:"
      print "   beam position: pixel:",bm_cxpixfit, bm_cypixfit
      print "                  coordinates:",bm_pixfit_RA, bm_pixfit_Dec
      print "   beam fwhm x,y:",bm_fwhm_x,bm_fwhm_y
      poo=1
    if (poo == 1): break
    pass

xpixstep=src["image"].meta["cdelt1"].value*3600. # pixel size in arcsec
ypixstep=src["image"].meta["cdelt2"].value*3600.
if (xpixstep<0): xpixstep=-1*xpixstep # avoid negative values
if (ypixstep<0): ypixstep=-1*ypixstep
# starting RA, Dec values in decimal degrees...from the meta data
# or you can work it out yourself
# decimal values are necessary
RA=src.meta["raNominal"].value
DEC=src.meta["decNominal"].value
# ...and in pixel values
cxpix=src.wcs.getPixelCoordinates(RA,DEC)[1]
cypix=src.wcs.getPixelCoordinates(RA,DEC)[0]
maxshift = 10.
print "   Fitting astronomical source ..."
for boxsize in range(5,60):
  try:
    minX=cxpix-boxsize/2.
    minY=cypix-boxsize/2.
    sfit   = sourceFitting(elongated=True,slope=False,image=src,\
            minX=minX,minY=minY,width=boxsize,height=boxsize)
    print "      boxsize "+str(boxsize)+"...success!"
    poo=1 # seems to be necessary to make this stop when it has success
  except:
    print "      boxsize "+str(boxsize)+"...failed"
    cxpixfit=-1 # or some other value to indicate failure
    cypixfit=-1 # eg maybe the centre of the map, ...
```

```
    pixfit_RA=-1
    pixfit_Dec=-1
    poo=0
  else:
    # upon success, grab the results of the sourceFitting
    cxpixfit   = sfit.getCenterX()
    cypixfit   = sfit.getCenterY()
    pixfit_RA  = sfit.getCenterRA()
    pixfit_Dec = sfit.getCenterDec()
    # very occasionally sourceFitting gives nonsence results, so:
    if ((ABS(cxpix-cxpixfit) > maxshift) or (ABS(cypix-cypixfit) > maxshift)):
      print "         ...however, source found too far from original coords,"
      print "         so setting to -1,-1"
      print "            orig coords: "+str(cxpix)+","+str(cypix)+\
            " found coords: "+str(bm_cxpixfit)+", "+str(bm_cypixfit)
      cxpixfit=-1 # or some other value to indicate failure
      cypixfit=-1 # eg maybe the centre of the map, ...
      pixfit_RA=-1
      pixfit_Dec=-1
      poo=0
    else:
      src_cxpixfit   = sfit.getCenterX()
      src_cypixfit   = sfit.getCenterY()
      src_pixfit_RA  = sfit.getCenterRA()
      src_pixfit_Dec = sfit.getCenterDec()
      sigma_x    = sfit.getSigmaXPixels()
      sigma_y    = sfit.getSigmaYPixels()
      src_fwhm_x     = round(abs(2.*SQRT(2.*LOG(2))*sigma_x*xpixstep),1)
      src_fwhm_y     = round(abs(2.*SQRT(2.*LOG(2))*sigma_y*ypixstep),1)
      print "Source details:"
      print "   source position: pixel:",src_cxpixfit, src_cypixfit
      print "                  coordinates:",src_pixfit_RA, src_pixfit_Dec
      print "   source fwhm x,y:",src_fwhm_x,src_fwhm_y
      poo=1
    if (poo == 1): break
    pass

bmr.getWcs().setCrval1(src_pixfit_RA)
bmr.getWcs().setCrval2(src_pixfit_Dec)
bmr.getWcs().setCrpix1(bm_cxpixfit+1.0)
bmr.getWcs().setCrpix2(bm_cypixfit+1.0)
print "WCS of rotated beam with old WCS imposed and shifted to the astro source
 position:"\
  ,bmr.wcs
d=Display(src)
d.setTitle(str(obsid)+","+band)
d.setCutLevelsPercentage(99.0)
d.setTitle(str(obsid)+","+band+",rotated by "+str(angle))
# red circle where the astro source was found to be
d.addCircle(src_cypixfit,src_cxpixfit,2,2,java.awt.Color.red)
# If you want to add contours of the beam and the astro source to the greyscale map"
# Beam:
# ->must be as many as there are contour levels
cs=[BLUE,BLUE,BLUE,BLUE,BLUE]
# ->change the min and max according to the flux range in your map
contoursb = automaticContour(image=bmr, levels=5, min=0.1, max=0.5, distribution=0,\
   colors=cs)
d.addWcsImageContour(contoursb)
# Astro source:
#cs=[GREEN,GREEN,GREEN,GREEN,GREEN]
#contourss = automaticContour(image=src, levels=5, min=0.002, max=0.004, \
#  distribution=0, colors=cs)
#d.addWcsImageContour(contourss)
d.setZoomFactor(16)

# Plotting the EEF curves
# Recommended aperture size
rskyin_src  = 60
rskyout_src = 75
# The aperture for the bm is not given (will use the same)
rskyin_bm = 60
rskyout_bm = 75
```

```
rapers=Double1d.range(41)*2

rapers=rapers[1:]

fluxes=Double2d(len(rapers),2)

cnt=0
for raper in (rapers):
  apphot = fixedSkyAperturePhotometry(image=src,centroid=False,\
           centerX=round(src_cxpixfit,3),centerY=round(src_cypixfit,3),\
           radiusArcsec=raper,sky=0)
  fluxes[cnt,0]=apphot["Results table"]["Total flux"].data[1]
  cnt+=1

cnt=0
for raper in (rapers):
  apphot = fixedSkyAperturePhotometry(image=bmr,centroid=False,\
           centerX=round(bm_cxpixfit,1),centerY=round(bm_cypixfit,1),\
           radiusArcsec=raper,sky=0)
  fluxes[cnt,1]=apphot["Results table"]["Total flux"].data[1]
  cnt+=1
# Scaling the fluxes to median

idx=rapers.where((rapers>rskyin_src)&(rapers<rskyout_src))
fluxes[:,0]=fluxes[:,0]/MEAN(fluxes[idx,0])
idx=rapers.where((rapers>rskyin_bm)&(rapers<rskyout_bm))
fluxes[:,1]=fluxes[:,1]/MEAN(fluxes[idx,1])



p=PlotXY(titleText="My EEF for obsid "+str(obsid)+", "+band)
p.addLayer(LayerXY(rapers,fluxes[:,0],line=1,color=java.awt.Color.black))
p.addLayer(LayerXY(rapers,fluxes[:,1],line=3,color=java.awt.Color.red))
p[1].setName("beam")
p[0].setName("astro source")
p.xaxis.title.text="arcsec"
p.yaxis.title.text="flux"
p.getLegend().setVisible(True)

# interpgrid needs to be an array of points to evaluate with the interpolator
interpgrid = Double1d.range(78*8)/8.+2
interp = CubicSplineInterpolator(rapers,fluxes[:,0])
eef_cont=interp(interpgrid)
idx=eef_cont.where((eef_cont >=0.47) & (eef_cont <=0.53))
sey=interpgrid[idx]
fw_as=MEAN(sey)

interp = CubicSplineInterpolator(rapers,fluxes[:,1])
eef_cont=interp(interpgrid)
idx=eef_cont.where((eef_cont >=0.47) & (eef_cont <=0.53))
sey=interpgrid[idx]
fw_bm=MEAN(sey)

print "FWHM (arcsec):"
print "   astro source:",fw_as
print "     PSF source:",fw_bm

# Sky background scatter
RA=bmr.getWcs().getCrval1()
DEC=bmr.getWcs().getCrval2()
cxpixb=bmr.wcs.getPixelCoordinates(RA,DEC)[1] # pixel coordinates of the beam
cypixb=bmr.wcs.getPixelCoordinates(RA,DEC)[0]
cxpixs=src.wcs.getPixelCoordinates(RA,DEC)[1] # pixel coordinates of the astro
 source
cypixs=src.wcs.getPixelCoordinates(RA,DEC)[0]

rsky = 40
raper= 12
raperpix=raper/ypixstep
separation=rsky/ypixstep
```

```
ycos=COS(60.0/180.*Math.PI)
xsin=SIN(60.0/180.*Math.PI)
matrix = [[-xsin,-ycos],[-xsin,ycos],[0,-1],[0,1],[xsin,-ycos],[xsin,ycos]]
sum_bck=Double1d()
d=Display(src)
d.setTitle(str(obsid)+", "+band)
d.addCircle(cypixs,cxpixs,raperpix,1,java.awt.Color.blue) # aperture photometry
 radius
d.addCircle(cypixs,cxpixs,separation,1,java.awt.Color.red) # (aperture photometry)
 sky radius
d.setZoomFactor(12)

for spot in matrix:
  xpos=round(cxpixs+spot[0]*separation,3)
  ypos=round(cypixs+spot[1]*separation,3)
  d.addCircle(ypos,xpos,raperpix,1,java.awt.Color.white) # the sky apertures
  eaper = fixedSkyAperturePhotometry(image=src,centroid=False,\
          centerX=xpos,centerY=ypos,radiusArcsec=raper)
  sum_bck.append(eaper["Results table"]["Total flux"].data[0])

clip=sum_bck.apply(Sigclip(nsigma=3.0,returnmode = Sigclip.RETURN_BOOL))
idx = clip.where(clip==False).toInt1d()
sum_bck_clip          = sum_bck[Selection(idx)]
avg_matrix_back       = MEAN(sum_bck_clip)
med_matrix_back       = MEDIAN(sum_bck_clip)
rms_matrix_back       = STDDEV(sum_bck_clip)
sky_bm = avg_matrix_back
print "PSF source sky values (mean: %f, median: %f, stddev: %f)" %\
  (avg_matrix_back,med_matrix_back,rms_matrix_back)
```

Example 4.167. Performing PSF comparison for SPIRE point sources.

4.22.5. Fitting the PACS PSF (for SPIRE it will be similar)

After rotating and aligning your astronomical source and the PSF you might want to try a PSF subtraction to perform photometry. Of course you have to scale your rotated PSF so that the residual flux would be minimized. The final flux of the source is the aperture photometry flux of the PSF star (rotated version), adjusted by the scaling factor you decide gives the best residuals. Most of the work here is in deciding what the best residuals are. By rotating a beam you introduce some artifacts and so there will always be irregular residuals. To help decide we produce: maps of the residual, EEFs of the residual, and photometry of the residual. First you set up some basic variables that you will need later, such as the camera (`"blue"` lets you reuse Example 4.166) and apertures:

```
camera="blue"
rapers=Double1d.range(21)*2
rapers=rapers[1:]
```

Example 4.168. Setting the camera and aperture variables.

then the scaling factors and the increments to apply to the scaling. you can have different scaling factors for each map if you do more than one map in one script. The increments are needed to refine the scaling factor. You will set your own increments, e.g.:

```
factors=[2.3,2.3,2.4,2.4,2.5,2.5,2.6,2.6,2.7,2.7,2.8,2.8]
incr=[-0.3,-0.25,-0.2,-0.15,-0.1,-0.05,0,0.05,0.1,0.15,0.2,0.25,0.3]
```

Example 4.169. Defining scaling factors for the data.

Here we show the case when the PSF subtraction is performed on two maps, and this is the reason why the values in "factors" are repeated. In this document we need to break up the script to smaller bits in order to be able to explain what is going on. But, since it contains a couple of "for" loops, you need to pay extra attention to the indentation if you cut and paste from here.

Note

Although you can run this on all obsids at once, it is easier to digest the results if you do not. If you decide to do more than one obsid then you need to make sure that factors have exactly as many elements as the number of your obsids.

Start the cycle and set up your parameters for aperture photometry. You will need them later:

```
# Initialise these arrays with your observations
OBSID = [1342186136]
mapList = [src]
bmList = [bmr]
for obsi in range(len(OBSID)):
  map=mapList[obsi]
  bm=bmList[obsi]
  RA=bm.getWcs().getCrval1()
  DEC=bm.getWcs().getCrval2()
  print "\nDoing",OBSID[obsi]
  if (camera=="red"):
     raper=12
     rskyin1=35
     rskyin2=41
     rskyout=45
  else:
     raper=12
     rskyin1=35
     rskyin2=41
     rskyout=45
  EEFs=[]
  SCALE=[]
```

Example 4.170. Setting up recommended aperture photometry values for all observation ids.

The outer aperture of the sky depends a bit on your map's background but it should be no more than 45 (because of the size of the PSF).

In the next step we scale and subtract the PSF.

```
  for k in incr:
    eef=Double1d(len(rapers))
    SCALE.append(factors[obsi]+k) # for PlotXY
    # divide by scaling--so your photometry of PSF star should be also:
    bm_sc=imageDivide(image1=bm,scalar=factors[obsi]+k)
    resid = imageSubtract(image1=map,image2=bm_sc,ref=1)
```

Example 4.171. Scaling and subtracting the PSF.

The loop that starts here goes through all the scaling increments and fills up the SCALE array, and scales the beam by dividing the PSF image with the scaling factor, and calculates the residual image by subtracting the scaled beam from the science data. Then continue, to perform aperture photometry on the residual image, create the EEF of the residual, and finally display the results and the EEFs of the residuals for all scaling factors.

```
    cxpix=resid.wcs.getPixelCoordinates(RA,DEC)[1] # pixel coordinates of the beam
    cypix=resid.wcs.getPixelCoordinates(RA,DEC)[0]
    ypixstep=resid["image"].meta["cdelt2"].value*3600.
    if (ypixstep < 0): ypixstep=-1*ypixstep
    raperpix=raper/ypixstep # for calculating flux error from mini sky apertures
    # Aperture photometry task does not like precise fits, this is what the
 "digits"
    # bit is for
    apphot = annularSkyAperturePhotometry(image=resid,centroid=False,\
             fractional=True,algorithm=4,centerX=round(cxpix,1),\
             centerY=round(cypix,1),radiusArcsec=raper,\
             innerArcsec=rskyin1,outerArcsec=rskyout)
    flux = apphot["Results table"]["Total flux"].data[0]
    separation=rskyout/ypixstep
    ycos=COS(60.0/180.*Math.PI)
```

```
    xsin=SIN(60.0/180.*Math.PI)
    matrix = [[-xsin,-ycos],[-xsin,ycos],[0,-1],[0,1],[xsin,-ycos],[xsin,ycos]]
    sum_bck=Double1d()
    sum_bck_ac=Double1d()
    # As commented above, you can play with the precision
    digits=14
    for spot in matrix:
      xpos=round(cxpix+spot[0]*separation,digits)
      ypos=round(cypix+spot[1]*separation,digits)
      eaper = annularSkyAperturePhotometry(image=resid,centroid=False,\
              fractional=True,algorithm=4,centerX=xpos,\
              centerY=ypos,radiusArcsec=raper,\
              innerArcsec=rskyin1,outerArcsec=rskyout)
      sum_bck.append(eaper["Results table"]["Total flux"].data[1])
    pass
    clip=sum_bck.apply(Sigclip(nsigma=3.0,returnmode = Sigclip.RETURN_BOOL))
    idx = clip.where(clip==False).toInt1d()
    sum_bck_clip          = sum_bck[Selection(idx)]
    avg_matrix_back       = MEAN(sum_bck_clip)
    med_matrix_back       = MEDIAN(sum_bck_clip)
    rms_matrix_back       = STDDEV(sum_bck_clip)
    cnt=0
    for r in (rapers):
       apphot = annularSkyAperturePhotometry(image=resid,centroid=False,\
                fractional=True,algorithm=4,centerX=round(cxpix,1),\
                centerY=round(cypix,1),radiusArcsec=r,\
                innerArcsec=rskyin2,outerArcsec=rskyout)
       eef[cnt]=apphot["Results table"]["Total flux"].data[2]
       cnt+=1
    EEFs.append(eef)
    disp=Display(resid)
    disp.setCutLevelsPercentage(95.0)
    disp.setZoomFactor(6.0)
    tit="Residual for "+ str(OBSID[obsi]) + ",scaling "+str(factors[obsi]+k)
    disp.setTitle(tit)
    disp.showAxes(True)
    axes = disp.showAxes(True)
    axes[0].setWorldCoordinates(True)
    axes[1].setWorldCoordinates(True)
    disp.addCircle(cypix,cxpix,2,2,java.awt.Color.red)
    print 'Scaling, Residual flux, sky median flux, sky rms::'\
        '%5.3f %.3f, %.3f +/- %.3f
 Jy'%(factors[obsi]+k,flux,med_matrix_back,rms_matrix_back)
tit="Residual EEF curve for "+ str(OBSID[obsi])
p=PlotXY(titleText=tit)
for i in range(len(EEFs)):
    p.addLayer(LayerXY(rapers,EEFs[i],line=1))
    p[i].setName("scaling:"+str(SCALE[i]))
p.xaxis.title.text="arcsec"
p.yaxis.title.text="normalised flux"
p.getLegend().setVisible(True)
```

Example 4.172. Performing the aperture photometry of the residual image, computing the EEF and printing every result.

Chapter 5. Spectral analysis

5.1. Summary

This chapter tells you about working with spectra in HIPE. It describes:

- Some basic concepts about the spectral tools and Herschel spectral data, in Section 5.2.
- How to visualise your spectra, Section 5.3.
- How to perform spectral arithmetics, averaging, manipulation, selection, how to convert units in your spectra and how to find the integral under a line in Section 5.4.
- How to deal with problematic baselines Section 5.5.

Many of these operations can be performed on spectral cubes too, which is described in Chapter 6. To learn how to get spectra into and out of HIPE, see Chapter 1.

Spectral fitting in HIPE is described in Chapter 7.

To learn about more advanced scripting when working with spectra, see the Scripting Guide.

5.2. Spectra in HIPE

Spectra from all three spectrometers on board Herschel (and spectra from other observatories) can be viewed and interacted with using the *Spectrum Explorer* and may be modified using the *Spectrum Toolbox* in HIPE. There are some instrument specific viewers and tools, such as the SPIRE SDI Explorer and some selection tools designed for HIFI data are described in the relevant instrument manuals.

Spectra come in various flavours in the framework of HIPE, and are explained in detail in Chapter 3 of the Scripting Manual in *Scripting Guide*. HIFI spectra are `HifiSpectrumDatasets` with `WbsSpectrumDatasets` and `HrsSpectrumDatasets` used for the Wide Band and High Resolution Spectrometers, respectively. SPIRE level 1 spectra are datasets of type `SpectrometerDetectorSpectrum`, while SPIRE level 2 spectra are datasets of type `SpectrometerPointSourceSpectrum`. All PACS pipeline spectral products are spectral cubes but spectra extracted from the PACS cubes by post-pipeline tasks are `spectrum1d` or `SimpleSpectrum`. Spectra extracted from spectral cubes of data from all three instruments are `Spectrum1d`, `Spectrum2d` or `SimpleSpectrum`, depending on the task used. All of these types of spectra are known to HIPE as `SpectrumDatasets` (often "datasets" in conversation) and implement what is known as the `SpectrumContainer` interface, which allows them all to be visualised and interacted with in the same way.

`SpectralLineLists`, which contain information on the properties of spectral lines extracted from Herschel spectra, can also be visualised in the Spectrum Explorer.

5.3. How to display spectra

Spectra are viewed in HIPE with the *Spectrum Explorer*.

To view a spectrum in Spectrum Explorer, right-click on the variable name of your spectrum (in the Variables pane or in the context holding your spectra) and choose Spectrum Explorer from the "Open With" menu option.

By clicking once on a variable in an Observation Context tree you will find that the spectra contained within it may be viewed in a small Spectrum Explorer window that appears next to the Observation

Context tree. This can be a convenient way to quickly inspect many spectra. However, not all aspects of the Spectrum Explorer work in this view. You may click on the arrow to the top right of the plot to make Spectrum Explorer 'take over' the entire Editor Pane, which will give you full functionality in Spectrum Explorer, or you may open the variable as described above.

If you have a spectrum stored as a variable in HIPE, say *MySpectrum*, you can also display it in the Spectrum Explorer from the command line with:

```
myPlot = openSE(MySpectrum)
```

Example 5.1. Opening the Spectrum Explorer from a script.

The Spectrum Explorer opens in a new tab inside the *Editor* view. Note that the tab title is always *plot*, irrespective of the spectrum variable name.

The Spectrum Explorer is divided into three panels, see Figure 5.1:

- The *Spectrum Panel*, the spectra are displayed and interacted with here. A button bar is found above the plot area, which gives access to tasks and toolboxes that work with the Spectrum Explorer. Note that if you move the focus away from the Spectrum Explorer, for example by clicking on the *Console* view, the icons will disappear. You have to click on the Spectrum Explorer tab in the *Editor* view to display the icons again.
- The *Data Selection Panel*, from here you select what to plot. The contents of your dataset are listed with a row for each spectrum (labelled by index) and columns for each SpectralSegment, only HIFI data contains multiple segments. If your spectrum contains *attributes* these are also displayed here.
- The *Preview panel*, shows a quick preview of a spectrum when selecting a row in the selection Panel without taking the memory to plot it in the Spectrum Panel. This requires the preview mode to be turned on, which is the default, and can be a helpful feature for large datasets.

You can resize these panels by dragging the divider bar and you can maximise any of the panels by clicking the small black arrows on the divider bars.

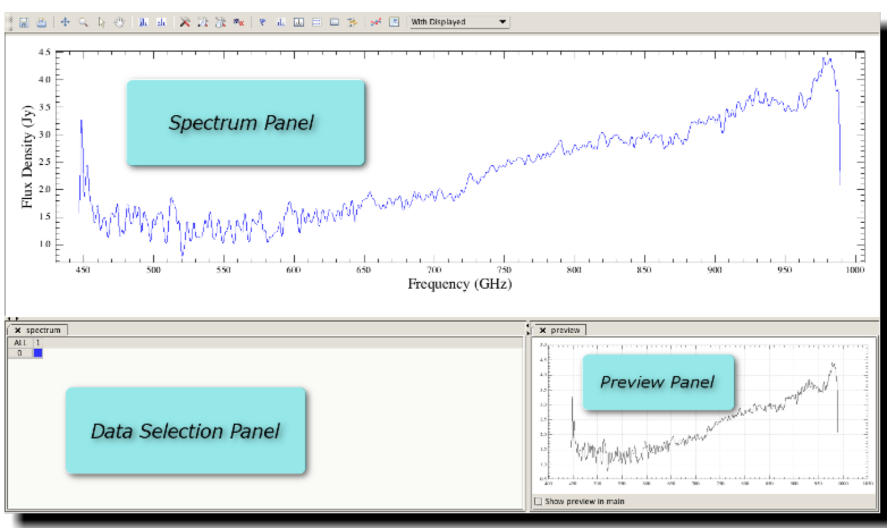


Figure 5.1. The Spectrum Explorer for a single spectrum.

The type of tab that opens depends on the product, but in all tabs you can perform the same actions. (Single-segment Spectrum1d will open in a Data Tree.)

5.3.1. Showing and Hiding spectra

The spectra displayed in the Spectrum Panel are added and removed via the Data Selection Panel:

- If your dataset contains only one spectrum it will be plotted immediately upon opening the dataset in the Spectrum Explorer. If your dataset contains multiple spectra (multiple rows in the Data Selection Panel) then no spectra will be shown initially when you open Spectrum Explorer using the "Open With" option. You can force Spectrum Explorer to plot all spectra in a dataset upon opening from the command line:

```
plot = openSE(MySpectrum, display = 1)
```

Example 5.2. Opening the Spectrum Explorer forcing all spectra to be plotted

- Spectra are displayed by clicking in the squares in the Data Selection Panel. Click on the square again to remove the spectrum from the plot.
- You can plot all the spectra in your dataset by clicking on the *All* button at the top left of the Data Selection Panel. Click on it to remove them all again.
- You can plot all of the spectra in one row by clicking on the index number (first column) of that row and you can plot all the spectra in a column by clicking on the column number. Clicking again will remove these spectra from the display.
- Data is plotted with a distinct colour for each spectrum, with the colour of the plotted spectrum matching that in its associated square in the Data Selection Panel. The colours cycle through blue, cyan, green, orange, red, magenta and grey.

You can remove all spectra from the plot by choosing the *remove from plot* option of the *With Displayed* feature at the far right of the button bar (you may need to resize the Spectrum Explorer panel to see it).

You can also remove one spectrum or a group of spectra from plot after *selecting* them (see Section 5.4.2). To select a spectrum you should enable the selection mode of the Spectrum Explorer by pressing the arrow icon at the left of the button bar then clicking on the spectrum you wish to remove. Select several spectra by clicking on all the spectra you wish to remove while in selection mode. The *With Displayed* menu will now read *With Selected* and you can choose the *remove from plot* option as above to remove the selected spectra.

5.3.2. Overplotting spectra

Spectra from different `SpectrumDatasets` can be overplotted in the Spectrum Explorer by dragging a new variable into the Spectrum Panel of an already open Spectrum Explorer.

- If a specific viewer is available for the type of data you drag into the Spectrum Explorer then this will be opened by default, for example the *Data Tree* is used for `HifiTimelineProducts`.
- From the command line you can add all the spectra in a new variable (*`MyNewSpectrum`*) to the plot with:

```
plot.add(MyNewSpectrum)
```

Example 5.3. Adding a new spectrum to a plot.

The new spectral product will open in a new tab in the Data Selection panel.

Note that displaying large amounts of data at once can take a long time. It is adviseable to be selective about what you choose to plot.

Adding a new variable to the plot adds a new layer to the plot. The new layer's wave and flux units and descriptions are compared with those already plotted. If the two sets of values are compatible (or

one set of values is not defined) then the data are all displayed in the same plot. If the values are not compatible (e.g., different units or same units but a different axis label) the the new data is displayed in a new plot (a *subplot*) in the Spectrum Panel.

5.3.3. Viewing multiple plots

It is possible to view several plots (*subplots*) in the Spectrum Panel. You can add plots either below or to the right of existing ones.

- To add a new plot below an existing one, right click beneath the plot and select *Add subplot* from the menu. An empty subplot will appear below the existing plot.
- Drag a spectrum from the Variables pane into the new subplot. The spectrum will be displayed and a new tab will be added to the Data Selection panel.
- Right-click to the right of any plot to add a new subplot to the right hand side.

Right clicking inside a subplot and on subplot axes gives you access to a subplot menu (over the usual menu items found by right clicking on a plot) with the following options:

- *Remove:* removes the subplot from the Spectrum Panel. If you remove all of the subplots you can add a new one by right clicking anywhere in the Spectrum Panel and selecting *Add subplot*.
- *Create plot variable:* creates a variable (an instance of `SpectrumPlot`) representing the plot that can be used in scripting. The first such variable created will be named `splot_0`. The following example shows how you can add spectra to a variable of this type:

```
obs = getObservation(1342249478, useHsa=True)
slwc3 = obs.level2.getProduct("HR_spectrum_point").getDefault()["SLWC3"]
sswd4 = obs.level2.getProduct("HR_spectrum_point").getDefault()["SSWD4"]
# splot_0 can be created from the Spectrum Explorer using the context menu
splot_0 = SpectrumPlot(slwc3)
splot_0.add(sswd4)
```

Example 5.4. Usage of SpectrumPlot variables (overplotting spectra).

- *Create plot copy:* creates a new `PlotXY` undocked window displaying just the selected subplot. Note that the new window lacks the Spectrum Explorer toolbar and options.
- *Active:* making a subplot active means that *any* (de-)selections you make in the Data Selection Panel will be reflected in this subplot, even if the data in the tab you are modifying 'belongs' to a different subplot. Clicking on a plot will make it active too.
- *Lock axes:* locks the axes of the subplots so that a pan, zoom in or rescale on one will be reflected on the other(s).
- *Unlock axes:* unlocks axes.
- *Align axes:* used on an axis or subplot, aligns the axes of the other subplots so that all show the same range for that axis. To undo this action, you must use the Auto range option (you can find it in the same right-click menu as the others) in each of the aligned subplots.

The option to (un)lock and align is also available from the *Axis* menu option upon right-clicking on an axis.

When the Spectrum Panel contains more than two subplots, the axis options Align and Lock allow the user to select one or more subplots where to apply the operation by pressing Ctrl + left-click for multiple selections. While doing multiple plot axes selections, the cursor will appear as a cross shape and the axes selection in the subplots will result in a blue highlight. To complete the operation, left-click on an blank area of the Spectrum Panel.

5.3.4. Zooming and Panning

- You can zoom in and out using the scroll wheel on your mouse (or with a two-finger gesture on any compatible trackpad or touchpad).
- This icon in the Spectrum Explorer button bar enables zoom mode, which is the default mode when the Spectrum Explorer is started. When this mode is enabled you can change zoom by drawing a rectangular box using the left mouse button and you can zoom back out to the original scale using Ctrl + left click (Cmd + left click on a Mac) in the plot.
- You can also zoom back out to the original scale using the *Autorange* option under the right mouse button. Autorange gives you the option to auto-scale the plot axes ignoring any flagged data (*without flags*); this can be helpful if there is a very strong artifact in your data.
- This icon in the Spectrum Explorer button bar enables panning mode. When this mode is enabled you can pan through the spectrum in the plot window by clicking the left mouse button and moving the mouse, you can pan along both axes.

5.3.5. Changing Display Axes

You can modify the displayed axes and add auxiliary axes within Spectrum Explorer by right-clicking on a *plot axis* and selecting the Axis option from the menu:

- On the left (or main) y-axis you have the option to hide the main axis (after hiding an axis this option becomes show the main axis), to invert the axis and also to show a grid, which displays grid lines at the values of the main tick points on the y-axis.
- On the lower (main) and x-axes, you have the same options as on the y-axis and also the possibility to add an auxilliary axis (*add aux axis*). Under the auxiliary axis option you can choose from wave number, wavelength (in m) and radial velocity (in km/s, RELATIVISTIC convention). The wavelength and radial velocity options offer other units from sub-menus.

Note

Some level products from the HIFI processing are computed on an *IF (Intermediate Frequency)* scale, which is an intermediate scale proper to the instrument but is not yet representative of the absolute sky frequency scale applying to the data. While the x-axes for these spectra can be displayed with other units, it is scientifically meaningless. Please consult the explanation on HIFI data below for more information.

Once an auxilliary axis has been added you will also find the options to remove and change the units of an axis. Note that these actions only change the way the axes are displayed and do not change the data at all. To change the data you should use the appropriate unit conversion tools, such as convertWavescale.

You may also find an instrument specific menu option. This is only enabled for HIFI data at the moment and allows the possibilty to display data in velocity (km/s, RADIO convention), LSB/USB (GHz), or intermediate frequency (MHz). Each of these options has other units available from sub-menus. The LSB/USB option is automatically the opposite to the frequency scale of the plotted data, i.e., LSB scale is offered for USB data.

Chapter 22 of the HIFI Data Reduction Guide includes a table of velocity conventions.

- On the right y-axis and the upper x-axis you do not have the option the show or hide the main axis but otherwise have the same options as for the main axis.

You can change the properties of the axes by right-clicking on an axis and selecting *Properties...* from the menu.

5.3.6. Changing Plot Properties

As of version 13.0 onwards, many of the buttons that affect the layout of the spectrum plot have been moved to a dedicated panel: the layout properties panel. Only the shortcuts on the main toolbar of the Spectrum Explorer have been removed, the functionality is still present and it is easier to use when displayed in its own properties panel.

- To open the layout properties panel click on the icon of the toolbar .

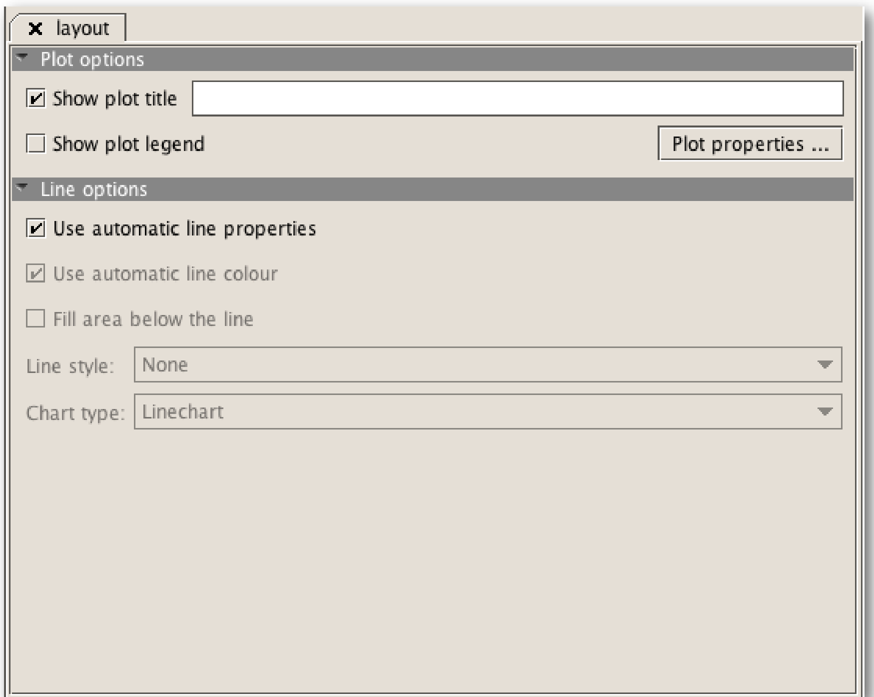


Figure 5.2. The new layout properties panel.

You can change the following properties in this panel:

- Display/hide the plot title using the checkbox *Show plot title*. This text can be changed at any moment the textbox to the right. The updated text is applied to the plot in real time.
- Display/hide the plot legend with the checkbox *Show plot legend*.
- Access finer configuration options for the plot clicking on the *Plot properties...* button.
- The line colour, line style, chart type (determined in the HIPE preferences), and line fill are automatically assigned by default. You can modify these using the lower part of the layout options panel (*Line options*), after deselecting the *Use automatic line properties* checkbox.
  - *Line (layer) colour*: Choose between automatic or manual line colour selection. To choose line (or layer) colours, first you have to deselect the checkbox *Use automatic line colour*. A coloured square button appears to the right. If you click on it, a grid of swatches will appear in a pop-up from which you can select the colour you want. Automatic mode can be enabled again selecting the checkbox as it is by default.
  - Select if the areas above and below the slope (the line between the start and the end values of the spectrum) are to be filled with a light gray colour. To do so, check the box *Fill area below the line* . More advanced options are available if you use PlotXY to plot your data.

  - *Line style*: after enabling manual configuration of the line options, you can select SOLID, MARKED (solid line with '+' symbols), DASHED, MARK_DASHED (dashed line with '+' symbols) or NONE in the combo-box to the right. Finer configuration is available (symbol shape, size and colour) using the properties panel, described below.
  - *Chart type*: select the chart syle from the drop-down menu. Choose between LINECHART (smooth line through the data points), HISTOGRAM, HISTOGRAM_EDGE (histogram with end points drawn to zero).

  You can apply these properties to *all* the subplots in the Spectrum Panel by selecting *assign layer properties* from the drop-down menu under *With Displayed* at the far right of the button bar. Note that you may have to resize the Spectrum Explorer to see this.

  Alternatively, you can apply these properties to a selection of spectra by *selecting* (see Section 5.4.2) those spectra, whereupon the menu changes to *With Selected* and then assigning the layer properties.

- To open the plot properties in the top-right panel of the Spectrum Explorer click on this icon . This dialog allows you to modify almost all the properties of the plot.

  Properties can be applied to three different plot elements: layer (line), axes, and plot. Once the property panel is open you can access the properties for a different element by clicking on it while holding down the Shift key. Each layer (line) has its own properties so you need to do this for each line you wish to modify the properties of.

  You can also open the properties panel using the 'Properties...' option seen in the pop-up menu when right clicking on a plot element. If a particular element in the context contains no changeable properties, the plot properties are displayed.

- The icon of the button displays/hides a grid in the active sub plot.

To change the default Spectrum Explorer settings, choose *Preferences* from the *Edit* HIPE menu and go to the *Spectrum Explorer* sections. There you can choose the default chart type and also make displaying grids and/or legends a default by checking those boxes.

In addition, there are subcategories for the data types that can be displayed in the Spectrum Explorer, such as `SpectralLineList`. For each data type, you can specify a custom plot title, subtitle and legend to be applied every time this data stype is displayed in Spectrum Explorer. Metadata fields and attribute fields can be filled in automatically by specifying the fields name between angular brackets, optionally with a printf-style format suffix. For example, `<longitude>%.2f` in the legend element field displays the value of the longitude attribute for each spectrum in the legend.

5.3.7. Viewing large datasets

The Spectrum Explorer will load all the data before starting to plot it and this can be a slow process for very large datsets - for example, a HIFI Spectral Scan or PACS, SPIRE EGS. It is adviseable to be selective about what you plot in the Spectrum Panel and there are several options available from the Spectrum Explorer button bar to help you to do that.

- : (de-) activates preview mode. This mode is automatically turned on and causes a preview of any row selected in the Data Selection panel to be shown in the Preview panel in the bottom right of Spectrum Explorer.

  You can use the preview to scan through your data by clicking on one row in the Data Selection panel and using the arrow keys to navigate row-by-row through the remainder. Checking the box in the bottom left of the preview panel will show the data in the main plot.

- : shows/hides the filter panel. The filter panel allows you to eliminate datasets from the Data Selection table, based on values of attributes (see the section below) in this way you can define a limited sample of data to plot.

- You can use the *selection task* to create a subset of your data to view and work with, see Section 5.4.2.

- : shows/hides the Data Tree panel. The Data Tree allows 'lazy loading', which means that a product is not loaded into HIPE until you click on it to display it. This allows you to start looking at large datasets without having to wait for the entire product to load, as was previously the case. It also means that you do not have to load all of a product even if you are only interested in looking at a few spectra contained in it, which saves memory in HIPE. One consequence of lazy loading is that you will not see any attributes for a spectrum until it is loaded.

  The Data Tree shows the hierarchical structure of *all* of the data in the Spectrum Explorer GUI in one tab. Data are shown in a collapsable tree structure that allows you plot or hide an entire branch of data in one go. The Data Tree allows you to inspect all of the data in the Spectrum Explorer and to add or remove data from the Spectrum Explorer from one location, rather than having to work with multiple tabs. However, you can open a tab for any product listed in the Data Tree by double clicking on its variable name in the second column.

  A right click anywhere in the Data Tree brings up a pop-up menu with the options to *display/hide*, *select/deselect* or *expand/collapse* one more selected rows (or all rows of none are selected). Right clicking on any cell and selecting the *copy cell contents* option copies the cell content to the clipboard, which can then be pasted elsewhere in HIPE or in other software.

  Rows are selected by clicking and are coloured yellow, a plot will be previewed in the Preview panel. Spectra selected in the plot (using the selection mode) are coloured pink.

  The columns (from left to right) show:

  - *First column*: a '+' for collapsed data containing multiple spectra, or a '-' for expanded datasets. Clicking on the symbol will expand or collapse that data tree.

  - *Second column*: displays the variable name of the data. You can rename the variable by double clicking on the variable name, the renamed variable will be added to HIPE Variables pane. Double clicking on a variable name also causes a red cross to appear next to the variable name, clicking on the red cross allows you to remove the data (and all 'sub-data' beloinging to it) from the Spectrum Explorer.

  - *Third column*: displays the colour of the line (layer), if displayed. If collapsed data are displayed in the plot the box will be grey coloured. Clicking on this box will display or hide all the spectra in this data set. On mouse-over a displayed spectrum in the plot will be temporarily highlighted.

  - *Remaining columns*: meta data in the data. As for the Data Selection panel, the columns can be reordered horizontally or sorted according to the meta data value (see Section 5.3.8).

- : opens a mosaic panel. This does not allow you to select spectra for plotting but rather shows large sets of data in a more efficient way in a new tab in the Spectrum Panel. The raster panel has three modes, which can be selected from the drop-down menu at the top right of the panel:

  - *Grid:* all of the spectra in the dataset are displayed in order from top to bottom-right as a series of postage stamps. On mouse-over a spectrum is displayed in the preview panel. You can adjust the x and y ranges of the data are viewed over using the slide bars at the top of the panel. This is the default mode the raster panel opens in.

  - *Raster:* the spectra are displayed according to their position, RA and dec values are given on the left and top axes, respectively. The x and y ranges of the data viewd can be adjusted as for the

grid view. Spectra that are from close-by sky positions may be plotted so close together that they overlap but each plot moves to the top on mouse-over and is displayed in the preview pane. You can also zoom in and out on the raster display using the mouse wheel or track pad equivalent.

- *Location:* crosses mark the positions of spectra in the dataset, with RA and dec given on the left and top axes. On mouse-over the spectrum of each point is shown in the preview panel and you can zoom in and out on the display using the mouse wheel or track pad equivalent.

You can reset the display with the reset button at the top right on the panel and return to the original scale after zooming with a right-click.

5.3.8. Filtering and sorting what is viewed

You can refine what data is shown in the Data Selection panel using the *filter panel* to hide data based on values in the data. The icon in the button bar opens and closes the filter panel. Alternatively, you can right-click on the plot and choose Dialogue → Filter to open the filter panel.

To apply a filter, click the cell in the *attribute* column to display a list of available attribute. Click on the = sign to choose a comparison operator. Finally, enter a value in the *value* column and press **Enter**. The entries in the Data Selection Panel are filtered according to your criterion, and a new line appears in case you want to define another filter.

Although the numbers reported in the Data Selection Panel are displayed to a few significant figures the actual values, found by hovering your mouse over a cell, are given to many more significant figures: the filter is sensitive to these values rather than the displayed ones. This means that you may find better results by limiting your filtering to between two ranges for a variable than giving a precise value.

Click on the green circle next to a filter to temporarily disable it; the Data Selection Panel will be immediately updated to show the results of the modified filtering. Click the red cross to remove a filter permanently.

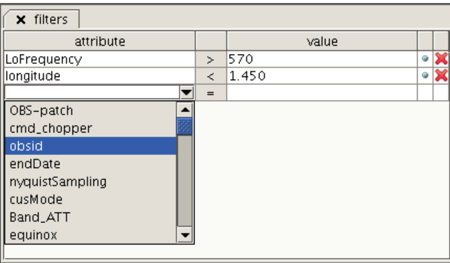


Figure 5.3. Filters on attributes.

You can sort data in the Data Selection Panel according to the values of attributes by clicking on the column header, click again to reverse the order. You can also modify which attributes are displayed: a right click on a column header will bring up an alphabetically ordered list of attributes in the data, each one preceded by a check box. Attributes that are checked will be displayed, by default all attributes are displayed. Clicking once in a check box will uncheck it and hide that column, clicking again will bring it back. Clicking in one check box and shift-clicking in another will toggle the states of all the columns in between.

Finally, you can change the order of attribute columns from left to right by directly dragging the columns around in the data selection panel, or you can drag column names up and down in the attribute list described above.

5.3.9. Viewing Flags/masks and plot information

You can visualise more detailed information about the data in your spectrum in Spectrum Explorer using two buttons in the button bar.

- : shows/hides flagged channels. If your data contains any flags (sometimes equivalently called *masks*) pressing this button once will cause a coloured block to appear over the spectrum in the region that is flagged. A different colour is used for each flag/mask in the spectrum but the colours are selected at random. Pressing the button again removes the flags from the plot.

- You can also find information about the plotted data in the lower left corner of the Spectrum Panel. On mouse-over of a plot element (a subplot, an axis or a plot layer/spectrum, the name of the context and location of the mouse cursor is displayed). When the mouse is near a spectrum a bullet the same colour as the spectrum is shown by the variable name, the channel number and any flag information is also displayed.

5.3.10. Viewing SpectralLineLists

`SpectralLineLists` contain information about spectral lines and can be overlaid on spectra to show the position of spectral lines. At the moment they are only produced in HIPE by the Spectrum Fitter, see Section 7.18.

- To view a `SpectralLineList`, open it in HIPE using the appropriate method (they can be saved as pools, FITS files or text files) and drag the `SpectralLineList` product into an already existing plot. The entries in the `SpectralLineList` will be displayed as a vertical line on the plot.

- You can also view the dataset in the `SpectralLineList` by opening the `SpectralLineList` with the *Product Viewer* and then opening the dataset contain in the with the *Dataset Viewer* product with the *Dataset Viewer*.

5.3.11. Printing and saving

Printing and saving of the plot (or all the subplots in the Spectrum Explorer) can be done via buttons on the button bar or from the menu that pops up on a right click in the plot.

- : saves the plot as a PNG, PDF, EPS or JPEG file. A pop-up appears in which you can specify the plot type, file name and file location. The same is reached by selecting *Save* from the *File* option upon right-clicking on the plot. Note that if several subplots are displayed they will all be saved together in the same file.

- : prints the plot (subplots). A pop-up allows you to set up the print job. The same is reached by selecting the *Print* from the *File* option upon right-clicking on the plot. Note that all subplots displayed in the Spectrum Panel will be printed.

5.3.12. Plotting from the command line

When making plots for publication or as interim results when running scripts it is more convenient to create plots via the command line. This is done using *PlotXY* or the command line version of the Spectrum Explorer, *splot* (a contraction of spectrum plot). The usage of these packages are described in the Chapter 3 in this manual.

5.4. Working on Spectra

Many of the tools provided in HIPE to help you reduce and analyse your data are accessible from the Spectrum Explorer. Spectral arithmetic and manipulation tools are gathered in the *Spectrum Toolbox* and are described in this section. Line fitting can be done using the *Spectrum Fitter* package, which is described in Chapter 7.

5.4.1. Using the Spectrum Toolbox

When a spectrum is viewed in Spectrum Explorer the Spectrum Toolbox can be opened by clicking on the crossed hammer and screwdriver icon in the toolbar. A task GUI will appear to the right of the spectrum and the spectrum tasks available in the toolbox can be selected from a drop-down menu. Alternatively, you can open the tasks described in this section by clicking on a spectrum in the *Variables* view and opening the *Applicable* folder in the *Tasks* view.

The tasks work on the spectrum (or spectra) that are displayed or on a selection that you make in the Spectrum Panel. In all cases the output of the task is of the same *class* as the input, for example, if you pass a `Spectrum1d` to a task, the output will also be a `Spectrum1d`.

Task execution

In HIPE 13 or later, the execution of tasks using the data displayed in the Spectrum Explorer is asynchronous, that is, the task runs in the background and the GUI remains responsive during all the process. A waiting cursor is displayed when you hover the mouse cursor over GUI elements yet to be refreshed (the Spectrum Explorer) for the duration of the task.

About tasks that create new data

For the spectrum and cube toolbox tasks that create new cubes or spectra and at the same time add them to the data selection pane, the focus will pass to the new product. This should be taken into account for additional runs of the same or other tasks, as the new product will be the new input for the task and some of the parameter entries in the task pane may change.

It is worth noting that if you open the tasks using the Spectrum Toolbox then you can open the *User Reference Manual* (URM) entry in the help for each task by clicking on the small question mark icon that is situated at the bottom right of the task GUI. If you prefer to open the tasks from the Task view then you can open the URM entry by right-clicking on the task name and selecting *Help in URM*. The URM entries for the spectral tasks, which are linked to the task name in the sections below, provide detailed descriptions and good code examples, therefore in this section we deal only with the GUI usage of the tasks.

Warning

It is better to create a variable for the spectrum you wish to work on than to open up a spectrum from the Context or Observation Viewer and work on that. Some of the spectrum tasks will not work if you operate from the Context (Observation) Viewer because they change the data they work on and HIPE will not allow you to change the contents of an `ObservationContext` in this way.

5.4.2. Spectral Selection: extraction, and flagging

All of the tasks available in the toolbox take advantage of the possibility to make selections on data while applying the task and some tasks can include or exclude ranges of data. In addition many of the tasks provide the possibility to account for flags in data. Therefore, the information in this section is relevant for all of the tasks in the spectrum toolbox.

- *Select in HCSS User's Reference Manual* : The `select` task can be used to select a subset of spectra from a dataset, the output from the task is the same as the input and can be passed to other spectrum tasks, this can be helpful when dealing with large datasets. However, the `select` task can be used by other spectrum tasks whilst running them. Here we describe how to make selections on spectra in HIPE in a general sense and then go on to discuss the usage of the `select` task.

  Selection of spectra is done with the *spectrum selection mode* of the Spectrum Explorer. Click on the arrow icon in the button bar and then click on a spectrum in the plot area to select it.

Continue clicking on other spectra to select more. A selected spectrum is indicated by dot symbols and will automatically be used by the task in the toolbox panel.

All of the spectra in a dataset can be selected by right-clicking in the plot and pressing Select → Select all and you can deselect them all with Select → Select none.

The selected spectrum can be dragged to the *Variables* view where it is stored as a new variable that can be plotted using PlotXY or splot, see Chapter 3.

The *Select* task GUI has fields for "selection", "selection_lookup" and "segments". These fields are seen on the GUI forms of all the spectral tasks that allow you to make a selection on your data. Enter indices of datasets to be selected in the "selection" field and segment numbers in the "segment" field, recalling that both start counting from 0. The "selection_lookup" field allows you to make selections on your data without needing to know the indices and make selections based on *attributes* (such as observation mode) or observation start time in the data is also possible in the command line (see examples in the URM entries). However, so far only HIFI has taken advantage of the ability to include attributes in data.

- *extract in HCSS User's Reference Manual* : Extracts data from a minimum to a maximum frequency/wavelength range for the complete set of spectra in a dataset. The spectral ranges to be extracted can be written into the boxes in the GUI but it is easier to draw a range on the spectrum using the *select range mode* of Spectrum Explorer.

  To do so, click on the *select one or more ranges* icon in the button bar. Click and drag to select ranges in the plot window (the middle mouse button can be used anytime for this as well). This will create a vertical grey bar and automatically enter the start (minimum) and end (maximum) values in the GUI. Drawing a second range adds more rows in the GUI.

  Ranges can be resized by clicking near the edge of the marker and dragging the edge of the marker to the desired position. Ranges can be removed from the GUI and the plot by right-clicking on the drawn range and selecting *Remove* or *Remove all* from the marker menu. The right-click menu also gives you the option to change the colour of the marker and of the line.

  When using the Extract GUI, clicking accept after drawing ranges will produce a new spectrum consisting only of the data in the ranges drawn on the original spectrum.

- *flagPixels in HCSS User's Reference Manual* : Flags pixels in a spectrum. A prerequisite for this task to work is that the data should contain flag values, which may not be the case for some PACS and SPIRE data. If you try to run the task and you see the error message:

```
java.lang.RuntimeException: No flag arrays included in data. Prepare the
data properly.
```

  that means you have no flag array and this task will not work on your data.

  Another pre-requisite is that the task should be able to modify the data (as flag values are overwritten); this means that you cannot use this task on a dataset that is still embedded in an `ObservationContext` tree, instead you should create a variable of the dataset and work on that.

  If no selection is made on the displayed spectrum, the task will apply new flag values to the entire spectrum. To select points (or pixels) to flag use the *select points* mode: a single click will select a point, a box drawn around a range of points will select the points in that range. Several selections can be made this way, simply continue to draw boxes around data you wish to flag. Right clicking on one set of selected points and choosing the *deselect* option under *Point selection* will remove all the selections from the plot. You can remove one selection by drawing another box around it, drawing a box while holding down the control key will extend the point selection. The `flagPixel` task will pick up any point selections in the Spectrum Explorer GUI with no need to enter anything into the task GUI.

Instead of selecting a range on the spectrum you also have the option of defining a "mask" parameter, which allows you to define regions by subband and channel number (or index number) for flagging. This is more usefully used in a script than when working with the GUI and the interested reader is referred to the URM entry for further information.

You can select the flag to use from the "flag" field. How this field is populated is instrument-dependent. The task tries to identify what instrument the data comes from and offer only the flags for that instrument. If the instrument cannot be identified then all of the flags in the build you are using will be offered. This means that if you are using a version of HIPE with only PACS installed you would only see the PACS flags but if you were using HIPE with all three instruments installed and the task could not identify what instrument your data belongs to then you will see the flags for HIFI, PACS and SPIRE.

Some flags that are set by the flagPixel task may be recognised by pipelines for some instruments, but not in all cases. Flag values that are recognised by instrument pipelines or instrument-specific data reduction tools are documented in the instrument Data Reduction Guides. There is no general 'ignore' or 'bad data' flag.

Whilst flagging data, you can optionally set the flux value of the flagged pixels to NaN by checking the "setFluxToNaN" box.

Note that it is possible to flag data from the Spectrum Panel plot without requiring recourse to the task GUI. After selecting points to be flagged using the select points mode you can right-click on the selected points and choose *flag* or *flag and remove* from the *PointSelection* menu. This approach provides lists of HIFI and SPIRE flags and a manual option with which one may assign an integer value for a flag. the *flag and remove* option is equivalent to the `setFluxToNaN` option in the `flagPixel` task.

5.4.3. Spectrum Arithmetics

- *add in HCSS User's Reference Manual, subtract in HCSS User's Reference Manual, multiply in HCSS User's Reference Manual, divide in HCSS User's Reference Manual* : Adds/subtracts/multiplies/divides a scalar value that should be entered in the *param* field to/from all spectra in the selected dataset. If your data contains multiple spectra and/or segments in *Scripting Guide* you can select a subset to work on by following the instructions in Section 5.4.2.

  By selecting the *pair-wise* option from the drop-menu, adds/subtracts/multiplies/divides groups of spectra or single spectra together. With the Pair-wise mode, if your datasets contains multiple spectra, the first spectrum of the datasets will be combined, then the second, etc. As for the scalar mode, you can select a subset of spectra to work on and specify segments.

  One note about command line usage: the tasks can also be performed using the common +, -, *, / symbols so:

```
spectrum_add_2 = add(ds = spectrum, param = 2.0)
#
# Is equivalent to
spectrum_add_2 = spectrum + 2
```

Example 5.5. Adding spectra using the overloaded method add()

5.4.4. Spectral Averaging and Statistics

- *avg in HCSS User's Reference Manual* : Averages a selection of spectra from a dataset.

  Flags and weights for individual channels/pixels can be taken into account using the variant parameter if they available in the spectrum (this will be the case for HIFI data, perhaps not for PACS or

SPIRE data). Flagged data can be ignored by the task, note that flags are specific to each instrument so you should refer to the Data Reduction Guide of your instrument to check flag values. See the URM entry for a detailed explanation of the effects of the different options for the variant parameter.

Selections can be applied before running the task, see Section 5.4.2.

In some cases the data, or your selection from the data, may fall into groups. For example, spectra taken at the same sky position, or with the same instrumental tuning. By checking the per_group box you can calculate the average for each of these groups separately.

- *pairAvg in HCSS User's Reference Manual* : Averages spectra pairwise from two input spectrum containers. The first spectrum in the first and second container are averaged, and so on. In case the size of the two containers is different, the result will contain a number of point spectra equal to the lowest size. A subset of spectra and segments, if available, within the dataset can be worked on.
- *accumulate in HCSS User's Reference Manual* : The accumulate task averages spectra to a common wave scale grid and will automatically resample data if needed. In addition, the accumulate task does not require that the spectra have the same length and will resample overlapping regions of spectra as required. The accumulate task will not average spectra that are not on a common flux scale or spectra that are not taken at the same position, within a given tolerance that you can specify.

  The task works on the all spectra that are displayed in the Spectrum Panel or on the selected spectra in the display. In addition you can use the *selection in HCSS User's Reference Manual* task to make a selection and pass that to the accumulate task by dragging it to the selection bullet from the Variables view. In addition, you may specify a range to average over by drawing that range on the plot after enabling the "Select one or more ranges" mode from the Spectrum Explorer button bar.

  You may specify how the average is done with the variant parameter in the same way as is done for the *avg in HCSS User's Reference Manual* task. You may also choose which flag to ignore, rather than ignore all flagged data. To identify the flag you must enter its integer value, rather than a flag name, check instrument guides for these.

  The task will resample the data if required but you can also specify the frequency grid to be used (the unit parameter refers to the units of the frequency grid) and the resampling method to be used. You are directed to the URM entry for details.

  Finally, you can specify the tolerance in wavelength/frequency and position to be used by the task. These are specified in the units of these values in the data.
- *statistics in HCSS User's Reference Manual* : Performs statistical operations on the datasets, always calculating the mean, rms, median and percentiles (quantiles). The quantities are automatically calculated for each segment is the data, which is typically one for PACS and SPIRE and may be more for HIFI.

  There are two modes of operation available from the drop-down menu:

  - perChannel: this mode operates per channel. Taking the mean as an example, if you had, say, ten spectra in your dataset, this mode would calculate the mean in the first data bin (or *channel*) from all ten spectra, and then for the second bin, then the third, and so on. This mode produces a Product, e.g. "stats", containing a `Spectrum1d` for each statistic, which may be plotted in Spectrum Explorer.
  - acrossChannel: this mode operates across the range of each segment in the data for each spectrum in the data. Taking the same example as above and assuming one segment, the result would be ten mean values. This mode produces a `TableDataset` that can be inspected in HIPE with the Dataset Viewer, allowing one to read off the values, and also export to text file, see Chapter 2. When using this mode you can also use sigma clipping, by supplying the clip value and a flag value to be assigned to the clipped data. Data will be clipped when it is above `clip value * sigma` (standard deviation).

You can select ranges to include or exclude by drawing a range (see Section 5.4.2) on the plot and choose whether the range is to be included or excluded from the drop-down menu found by clicking on *include ranges*.

5.4.5. Spectral Manipulation: resampling, smoothing, replacing, gridding, stitching, and folding

- *resample in HCSS User's Reference Manual* : Resamples data using a Trapezoidal or Euler box. When using a box filter you should either supply a frequency grid (*grid*) that the data should be resampled to as an array of Double1ds, or supply a fixed width (*resolution*). Note that the resolution you supply is actually twice the width that the data is resampled to. Another option is to use a Gaussian filter, which requires that you set a smoothing width in the *kernel* field. All resampling modes conserve flux.

  By default the *density* option is set to True (the box is checked) and this means that the flux data is treated as a flux density (per wavescale unit), if set to false the flux is treated as a per wavescale bin quantity (i.e. the integrated flux per bin).

- *smooth in HCSS User's Reference Manual* : Transforms the displayed (or selected) spectra via a box or gaussian (of user-selected width) smooth of the spectra in a dataset. You must supply a value for the width parameter. See the URM entry for details of this and the other task parameters.

- *replace in HCSS User's Reference Manual* : Replaces all or part of a spectrum with another. This is potentially useful if (part of) one integration among several is "bad" and could safely be replaced with the average of the others. In theory, bad parts of spectra could be replaced with NaNs in the case that analysis tools do not honour flags in the data but this is not tested.

  In order to replace one spectrum with another you should plot both and then use the selection mode (see Section 5.4.2) to select the spectrum to be replaced, *ds*, and the one to replace it with, *by*.

  To replace part of one spectrum by another you should extract (see Section 5.4.2) the part of the spectrum you wish to insert and use that as the dataset in the `by` bullet.

- *Gridding* : Used to grid data in a SpectrumContainer into spectral cubes. This is originally a HIFI task and still requires that you install the HIFI build to have access to it but is currently being adapted by SPIRE to produce spectral cubes. For descriptions on usage see the SPIRE and HIFI Data Reduction Guides.

- *stitch in HCSS User's Reference Manual* : Stitches together spectra or spectral segments. This is mostly used for HIFI data to stitch subbands. In theory, one could combine overlapping spectra from the same or different instruments into one dataset and stitch them together with this task but this has not yet been systematically tested.

- *fold in HCSS User's Reference Manual* : A HIFI-specific tool that folds frequency-switched spectra. The frequency throw is found in the meta data and is picked up automatically by the task.

5.4.6. Spectral Unit Conversion

- *convertWavescale in HCSS User's Reference Manual* : Transforms the wavescale between frequency, velocity, wavelength and wavenumber. When converting to velocity you must supply a reference frequency and the units of the reference frequency.

5.4.7. Finding the integral under a line

You can use the Spectrum Fitter, which produces the integral under the model of a fitted line as one of its outputs, see Chapter 7.

A command line tool is available with which to calculate the integral under a line. It allows to integrate over user-defined regions of a spectrum and to optionally remove a Polynomial background.

The data is expected to be in the form of a `SpectralSegment` (*segment* in the examples below) and the value reurned is a `Double`, which cannot be saved to disk unless you wrap it in a Product but can be used in scripts.

How `SpectralSegments` relate to spectra is described in the Scripting Guide. A quick example is given here showing how to extract a `SpectralSegment` from a `SpectrumDataset`, *MySpectrum*. Assume *MySpectrum* contains only one spectrum, in the Spectrum Explorer Data Selection Panel you would see only one row with one box, to get to `SpectralSegment` associated with that data:

```
# Note that PointSpectra are counted from 0 and SpectralSegments from 1!
    segment = MySpectrum.getPointSpectrum(0).getSegment(1)
```

Example 5.6. Selecting point spectra and segments.

If *MySpectrum* contained, say, five spectra and you wanted the fourth you would use:

```
# Note that PointSpectra are counted from 0 and SpectralSegments from 1!
    segment = MySpectrum.getPointSpectrum(3).getSegment(1)
```

Example 5.7. Selecting point spectra and segments (second variation).

Some PointSpectra can contain multiple `SpectralSegments`, for example, HIFI data contains one segment per subband, specify the segment number you want.

Before using the integrator you must import it:

```
from herschel.ia.toolbox.spectrum.integrator import Integrator
```

Example 5.8. Importing the Integrator class

- *Integration over a range or ranges:*

  The ranges (`[windows]` in the example) over which to integrate are formatted as [start, stop, start, stop, ...]. So for one range from a to b: [a, b], and for n ranges: [a1, b1, a2, b2, ..., an, bn]. The ranges should be given in the same units as the abscissa data.

  ```
  i = Integrator.doIntegration(segment, [windows])
  ```

  Example 5.9. Integrating over a set of ranges

- *Integration over a range or ranges with 1st order background removal:*

  The background is removed by fitting a 1st order poly in the [masks] areas. The format of [masks] is similar to the [windows] format and, similarly, masks have the same units as the data.

  ```
  i = Integrator.doIntegration(seg, [windows], [masks])
  ```

  Example 5.10. Integration over a set of ranges with masking.

- *Integration over a range or ranges with nth order background removal:*

  ```
  i = Integrator.doIntegration(seg, [windows], [masks], n)
  ```

  Example 5.11. Integration over a set of ranges with masking and removing up to n levels of background

5.4.8. Weight/error and flag/mask propagation

Weights and errors in datasets are set by the instrument pipelines and are propagated by the spectral arithmetics tasks described above. Datsets of HIFI spectra contain a weights column, while SPIRE datasets contain an error column (and may also contain a weight column but the values contained in it are calculated from the error). PACS does not currently assign any errors or weights in its datasets so PACS users should ensure they exclude weights when running any of the spectral tasks that can consider weight as a variant.

In practice, the spectral arithmetics tasks only propagate weights (w) and uses the standard weight-sigma relation to propagate errors, $w = \sigma^{-2}$

Until HIPE 9.0, weight propagation was carried out using a simplistic scheme. From HIPE 9.0 on, the propagation is done such that errors are also correctly propagated for scalar and pair-wise addition, subtraction, multiplication, division and pair-average. A weight propagation scheme that also correctly propagates errors for the remaining tasks was set in place in HIPE 10.0. The table below shows the weight propagation scheme used from HIPE 9.0 on, throughout the subscript 1 refers to the first dataset passed to the task and the subscript 2 to the second.

The terms "flags" and "masks" are used interchangeably in the Herschel project, with HIFI typically favouring the term "flag" and PACS and SPIRE preferring "mask". The table below also shows how the flags/masks are propagated by the spectral tasks.

Task	**Weight propagation scheme**	**Comment**	**Flag/mask propagation scheme**
Pair-wise add/subtract	$w = (w_1 * w_2) / (w_1 + w_2)$	If the denominator is zero then a zero weight is defined	
Scalar add/subtract	unchanged		unchanged
Pair-wise multiply/divide	$w = (u_1 * u_2) / (u_1 + u_2) * f^{-2}$	where $u_k = w_k * f_k^2$ and f is flux. If the denomimator is zero then a zero weight is defined.	
Scalar multiply/divide	$w = w / k^2$	where k is the scalar	unchanged
Pair-average	$w_{arithmetic\ mean} = 4 * (w_1 * w_2) / (w_1 + w_2)$, $w_{weighted} = (w_1 + w_2)$	If the denominator is zero then a zero weight is defined. The arithmetic mean is calculated with variant="flux", the weighted mean with variant = "flux-weight".	
Stitch			
Accumulate			
Smooth	same smoothing as chosen for the fluxes		bitwise-OR logic
Resample	same scheme as used for the fluxes		bitwise-OR logic

5.5. Dealing with baseline issues

If you are only interested in line emission rather than the continuum then you may wish to correct baseline issues before working on your data. Within HIPE you have access to a task that allows you to correct for standing waves in the baseline of data from any instrument and also a tool to allow you to mask out lines and then smooth the baseline.

5.5.1. General Standing Wave Removal Tool

5.5.1.1. Introduction to FitFringe

`FitFringe` is a general sine-wave fitting task that can be used to remove periodic signals in spectra, such as standing waves, from HIFI, PACS and SPIRE data. A description of the method and history of the code can be found in Kester et al. ("The Calibration Legacy of the ISO Mission", 2003, ESASP 481, 375). Briefly, `FitFringe` does the following:

1. A baseline for the signal to be fitted is determined by using the `SmoothBaseline` task, see Section 5.5.2. Sharp spectral features are masked out using a sigma clipping algorithm, which is also done by the `SmoothBaseline` task. You can control the baseline shape by indicating a typical period ('midcycle') that is being searched for.

2. Single sine waves are fitted to the baseline-subtracted spectrum, over a wide range of periods. Best-fitting periods are determined from local or absolute minimum Chi-square points.

3. The sine-wave amplitudes and phases are determined by solving a set of linear equations using the 'LU' matrix decomposition method.

4. The solution is subtracted from the data and the baseline is added back in.

One can fit any number of sine waves to the data. The wavelength units of the input and output spectra are always micron. Finally, this is not an instrument-specific task; however, a specific task does exist for HIFI data and this is documented in the `fitHifiFringe` chapter of the *HIFI Data Reduction Guide* .

5.5.1.2. Running FitFringe

FitFringe accepts SpectrumContainers (e.g. a SpectrumSimpleCube, or HIFI's WbsSpectrumDataset) as input:

```
swData = FitFringeData(SpectrumContainer, n, m)
```

where *n* is the n-th spectrum (technically `PointSpectrum`) in the data and *m* the m-th segment in the `SpectrumContainer`. If m is not given, all segments are selected. This format is not very user-friendly for use with cubes, Section 6.7.2 explains how to work out the point spectrum index number and provides a script to work this out from cube coordinates. Note that HIFI Spectrum Datasets contain multiple segments by default, while PACS and SPIRE data contain only one segment by default, however, PACS and SPIRE data aficionados can construct spectra with mutilple spectra too.

`FitFringe` also accepts a variable produced by `FitFringeData` as input. `FitFringeData` in turn accepts either arrays of wavelength in micron (`Double1d`), flux (`Double1d`), flags (`Int1d`), and weights (`Double1d`) as input:

```
swData = FitFringeData(myFreq,myFlux,myFlag,myWeight)
```

FitFringe can be opened from the "General" menu under "By Category" in the Tasks pane and does not appear "Applicable" on any kind of data.

FitFringe can be run on the command line and with a GUI. The latter looks as follows:

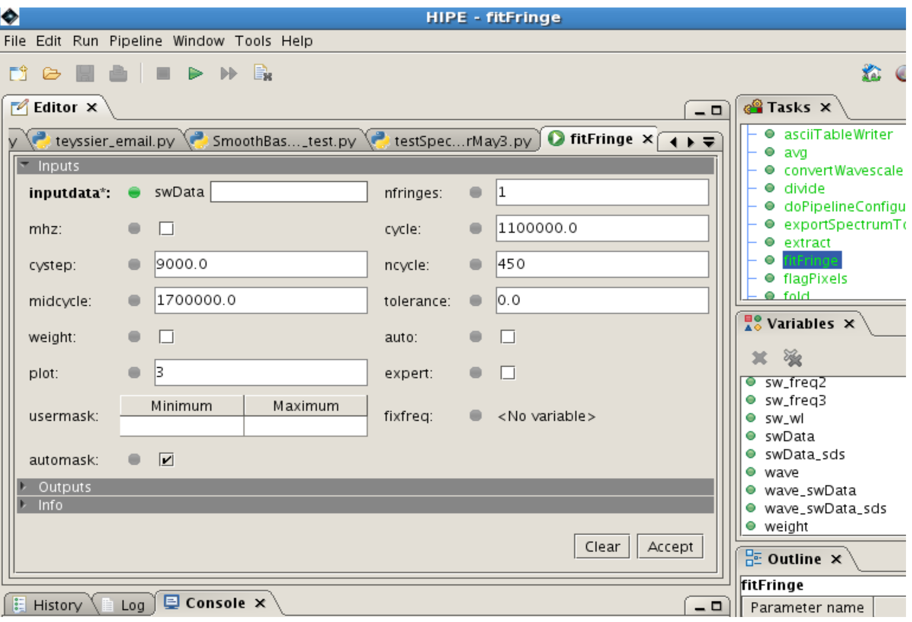


Figure 5.4. fitFringe task GUI with the parameter entry form.

Clicking on 'Accept' assumes the defaults further explained below. It is equivalent to the command line statement:

```
improvedData = fitFringe(swData)
```

In the process, two plots are created by default. The following plots were created using the script listed in the box below. The first one shows the sine wave period as a function of Chi^2. Selected dips with minimum Chi^2 are indicated with vertical red lines.

Looking at this plot helps you understand how well you have set-up the standing wave removal: if you have the right number of standing waves you will see a red line in every large dip in the Chi^2. Similarly, if you see lots of dips very compressed together and little else in the plot then you can narrow your fitting range down to that region.

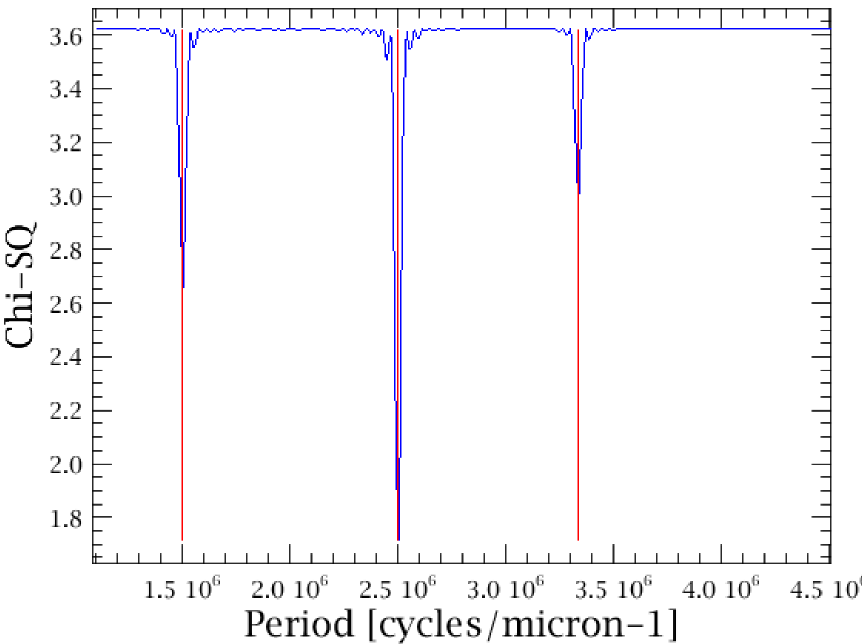


Figure 5.5. Plot with standing wave removal lines.

The second plot shows the original data, the baseline, the sine-wave subtracted data, and the mask. You can zoom in on the solution and original spectrum to investigate how well the task has done by drawing a box with the mouse.

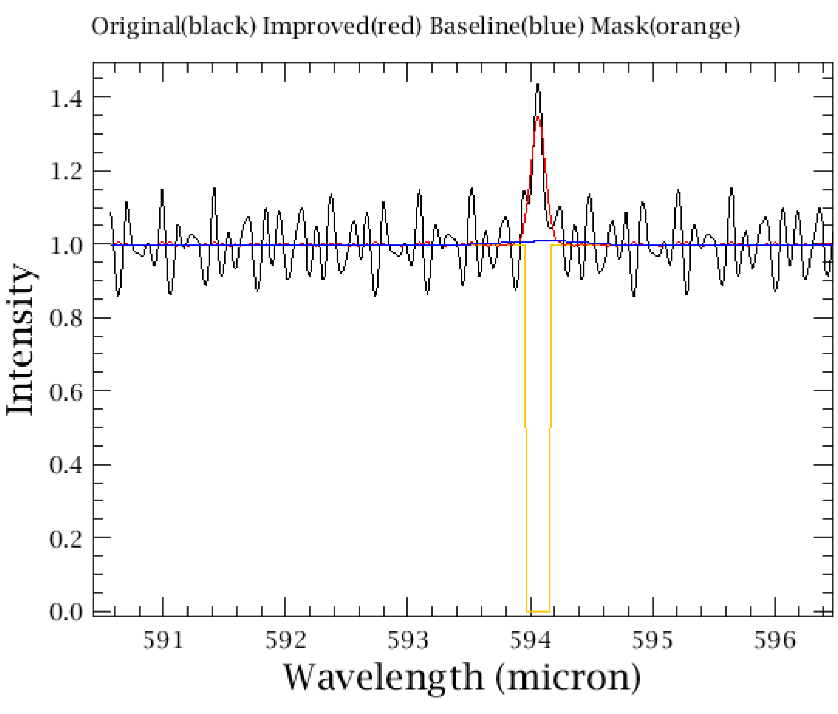


Figure 5.6. Original data with overplotted baseline, mask and subtracted data.

The output data with the sine waves subtracted can be retrieved as follows:

```
wave=improvedData.wave
```

```
flux=improvedData.flux
```

```
flag=improvedData.flag
```

```
weight=improvedData.weight
```

The applied baseline is stored in a similar way. These are `Double1d` and can be converted into a `Spectrum1d` that can be worked with a plot in the Spectrum Explorer following the method described in the *Scripting Guide*, Section 3.2.1 in *Scripting Guide*:

```
mySpectrum1d = Spectrum1d(flux, wave, weight, flag)
```

The fitted parameters are stored in a TableDataset, which contains a list of the fitted sine waves:

```
fringeNum: fringe number
cycle: period [per wavenumber in micron]
cycle_In_MHz: period [in MHz]
sinAmp:  amplitude of sine component
cosAmp:  amplitude of cosine component
chisq:   chi^2
chiRed:  total chi^2 reduction
```

The list can be viewed as

```
f=fitFringe.fringelist
```

```
print f
```

For example, the sine wave periods in MHz are retrieved as

```
print f.getColumn("cycle_In_MHz")
```

As the GUI shows, several parameters can be controlled by the user:

- *nfringes* : number of sine waves to be fitted [DEFAULT: 1].

  You require one sine wave for each periodic signal in your data. It can be hard to determine this by eye so using the number of dips seen in the Chi^2 plot is recommended.

- *mhz* : periods in the plots and in the input parameters 'midcycle', 'cycle', 'cystep', and 'fixfreq' are expressed in units of cycles per wavenumber in micron, if the 'mhz' boxed is checked then MHz are used instead. Note that regardless of the state of this keyword, the wavelength units of the input and output spectra, as well as the plots, are always in micron. The 'mhz' parameter only affects the 'midcycle', 'cycle', 'cystep', and 'fixfreq' parameters. [DEFAULT: mhz=False]

- *midcycle* : typical cycle frequency used for smoothing in order to determine the baseline [DEFAULT: 1.7E6 cycles/micron^-1 = 176 MHz]

  If your smoothed baseline shows a periodic structure to it then you need to use a longer period smoothing frequency.

- *cycle* : start of sine wave period search range [DEFAULT: 1.1E6 cycles/micron^-1=2727 MHz]

- *plot* : show results in plots [DEFAULT: a period versus Chi^2 plot and a before/after plot]

- *expert* : show more plots of intermediate steps [DEFAULT: not]

- *fixfreq* : fix periods to these values, i.e. do not search for them. Has to be same number as nfringes [DEFAULT: search for periods].

This parameter must be set by creating a Double1d variable containing the periods of the wine waves you want to fit. For example, fix two sine waves to have periods of 1.1E6 and 3E6:

`fixed = Double1d([1.1e6, 3E6])`

the variable `fixed` must then be dragged to the fixfreq bullet and you must set the number of sine waves to fit (nfringes) to 2.

- *ncycle* : number of cycles to check [DEFAULT: 450]
- *cystep* : step between cycles, i.e. resolution of the frequency space to search for standing waves [DEFAULT: 9000 cycles/micron^-1--unlikely to be modified by the user]
- *weight*: set all weights to 1 [DEFAULT: assign smaller weights to outliers]
- *automask*: automatically mask datapoints using the sigma-clip algorithm described in the `Smooth-Baseline` documentation. This mask is added to any user-defined mask ('usermask') that is provided. [DEFAULT: automask=True]
- *usermask*: mask wavelength ranges in addition to the automatically determined mask. Example: usermask=[(537.0,538.0), (539,539.5)] masks the ranges 537-538 um and 539-539.5 um. [DEFAULT: only automatically determined masks are used]
- *tolerance*: reduce Chi^2 until reduction is less than tol (0.01 == 1 percent). [not yet implemented]
- *auto*: automatically determine the maximum number of fringes needed within the noise, using Bayesian statistics. [not yet implemented]

The 'cycle', 'ncycle' and 'cystep' determine the cycle range (=range of standing wave frequencies) in which the task searches for a best fit. The search range is defined as: start = 'cycle', end = 'cycle' + 'cystep'*'ncycle'. If the best fit is outside the search range, the solution will not be very different than the input data.

However, it is more important to make a decent guess at the number of sine waves you want to fit and the typical period ('midcycle') in your data.

Converting to cycles/wavenumber from what you see in your spectrum is not intuitive. Recall that wavenumber is $\nu = 1/\lambda$. If your data are in microns you only need take the reciprocal of, say, the typical period to find the value to pass to the task. If your data are already in inverse cm, just convert to inverse microns. If your data are in MHz then use *(c/f)*1E6*, where *f* is your frequency and the factor of 1E6 is to convert from m to micron.

The best approach to take is to make some reasonable estimate from your data and modify according to how the Chi^2 plot looks.

This example shows how `FitFringe` can be used. It is the script used to produce the plots shown above.

```
#frequency in GHz (FitFringe assumes the periods are
#constant in frequency space)
myFreq=Double1d.range(800)/100.+500

#flag and weights
myFlag=Int1d(800)
myWeight=Double1d(800)+1.

#sum of 90, 120, and 200 MHz standing waves
#and a Gaussian emission line
sw_freq1=90
myFlux=SIN(2*Math.PI*myFreq/(sw_freq1*1.e-3))*0.04+1.0
```

```
sw_freq2=120.
myFlux=myFlux*(1.+SIN(2*Math.PI*myFreq/(sw_freq2*1.e-3))*0.07)
sw_freq3=200.
myFlux=myFlux*(1.+SIN(2*Math.PI*myFreq/(sw_freq3*1.e-3))*0.05)
myFlux=myFlux+0.35*EXP(-0.5 * (( myFreq - 505. ) / 0.05 )**2 )

#fitFringe expects wavelength in micron
myFreq=(3.e14/(((myFreq))*1.e9))

# Make the input standingwave data
swData = FitFringeData(myFreq,myFlux,myFlag,myWeight)

# Run FitFringe
results = fitFringe(swData,nfringes=3)

#output data will be in
# results[0] : improvedData
# results[1] : baseline
# results[2] : mask
# results[3] : fringelist

# Check the fringe list (a TableDataset)
fringelist = results[3]
print fringelist
print fringelist.getColumn("cycle_In_MHz")
```

Example 5.12. Fitting the fringes.

5.5.2. Baseline Smoothing and Line Masking Tool

5.5.2.1. Introduction to SmoothBaseline

The `SmoothBaseline` task produces a smooth baseline and a mask of spectral features with no (or very little) user interaction. It works by smoothing, median filtering, and clipping the spectrum a number of times. Spectral lines are masked and any standing waves are smoothed over. Both the smooth baseline and the mask are returned. Although `SmoothBaseline` was originally developed for use with the `FitFringe` sine wave fitting routine, it can be used on its own as well, for example for automated baseline and line detection purposes.

5.5.2.2. Running SmoothBaseline

`SmoothBaseline` accepts a variable produced by `FitFringeData` as input. `FitFringeData` in turn accepts either arrays of wavelength, flux, flags, and weights or a `SpectrumContainer` (e.g. HIFI's `WbsSpectrumDataset`) as input. See the box below for examples of either case. The box also shows how `SmoothBaseline` can be run from the command line. A GUI can be opened from the "General" tasks under "By Category" in the tasks pane.

The use of this task on cubes is a little awkward because you must work spaxel/pixel by spaxel/pixel, unless you write a script to run a loop. It is also not simple to determine point spectrum index number from cube coordinates, and a script to do that can be found in Section 6.7.2.

The key input parameter is 'midcycle', which is essentially the typical scale to which the baseline is to be smoothed. Its unit is the number of cycles per wavenumber unit, where wavenumber is defined as 1/wavelength. Any structure in the spectrum that has a much longer period than 'midcycle' is considered baseline structure and will not be smoothed or masked.

After applying median filter with width 'midcycle', a boxcar smoothing with 10 times the width of midcycle is done to determine outliers larger 4 times the difference between the smoothed and input spectrum. The default boxcar value of 10 can be overruled by the user, although this is likely rarely needed.

The user can also mask spectral regions a priori, by using the 'usermask' input option.

Here is a summary of the parameters that can be controlled by the user:

- *mhz*: periods in the plots and in the input parameters 'midcycle', are by default expressed in units of cycles per inverse wavenumber in micron, unless the 'mhz' box is checked. [DEFAULT: mhz=False]
- *midcycle*: typical cycle frequency used for smoothing in order to determine the baseline [DEFAULT: 1.7E6 cycles/micron^-1=176 MHz]
- *plot*: show results in plots or not [DEFAULT: plot=True]
- *automask*: automatically mask datapoints using a sigma-clip algorithm. This mask is added to any user-defined mask ('usermask') that is provided. [DEFAULT: automask=True]
- *usermask*: mask wavelength ranges in addition to the automatically determined mask. Example: usermask=[(537.0,538.0), (539,539.5)] masks the ranges 537-538 um and 539-539.5 um. [DEFAULT: only automatically determined masks are used]
- *box*: smoothing with a box of size 'box' times the width of midcycle is done to determine outliers. [DEFAULT: box=10]

You are directed to Section 5.5.1.2 for more information about how to convert various wavelength units into cycles per wavenumber.

Below, the output from SmoothBaseline is explained. By default two plots are generated, but this can be avoided by entering 'plot=False'.

Example script:

```
#make a test spectrum

#wavelength in micron
myWave=Double1d.range(800)/100.+500

#flag and weights
myFlag=Int1d(800)
myWeight=Double1d(800)+1.

#a standing wave with a wavelength of 0.5 micron
#and a Gaussian emission line
sw_wl=0.5
myFlux=SIN(2*Math.PI*myWave/(sw_wl))*0.04+1.0
myFlux=myFlux+0.35*EXP(-0.5 * (( myWave - 505. ) / 0.05 )**2 )

# Prepare spectrum data to be processed
swData = FitFringeData(myWave,myFlux,myFlag,myWeight)

#Alternatively, if the data are available in a SpectrumContainer 'sds'
#containing N spectra ('scans') of M segments (e.g. WBS sub-bands), the
#n-th spectrum is selected as follows:
#swData = FitFringeData(sds, n)
#and its m-th segment as follows:
#swData = FitFringeData(sds, n, m)

# Run SmoothBaseline. Note that the exact value of midcyc is not
# very important, though it should be of the same order of magnitude
# as the waves in the spectrum. Here, 7.e5 cyc/micron^-1
# corresponds to waves with lengths of lambda^2/midcyc=0.35 micron
baseline = smoothBaseline(data=swData,midcycle=7.e5, plot=True)

#obtain mask of found spectral lines
mask = smoothBaseline.mask

#smooth baseline will be in
# baseline.wave
# baseline.flux
```

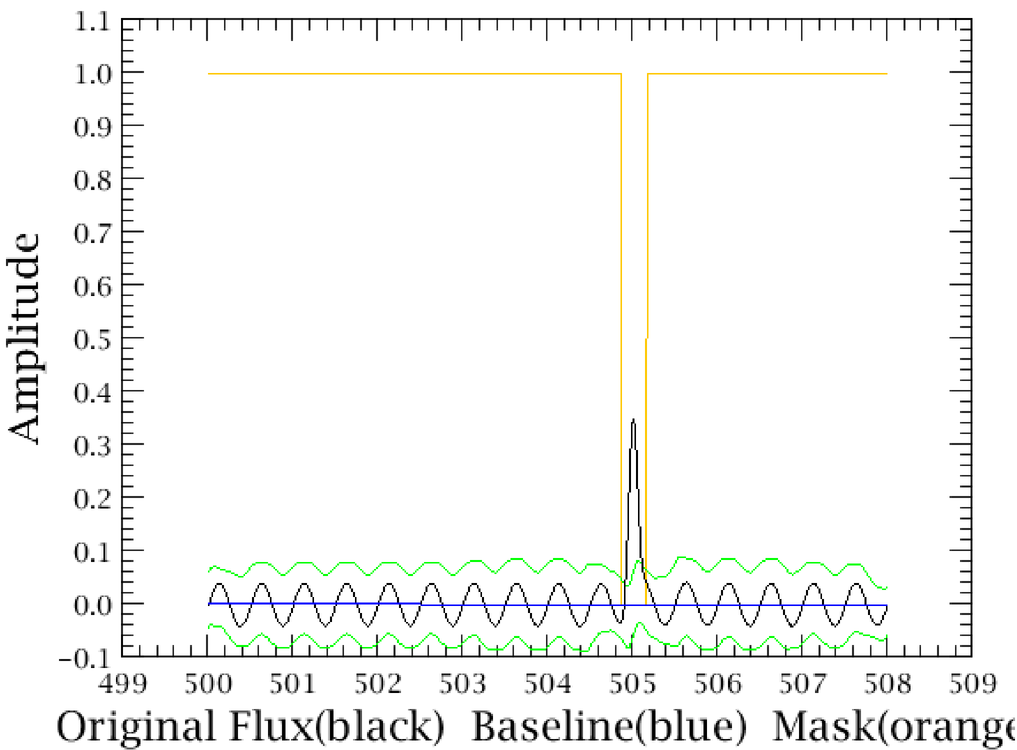
```
# baseline.flag
# baseline.weight
```

Example 5.13. Smoothing the background baseline.

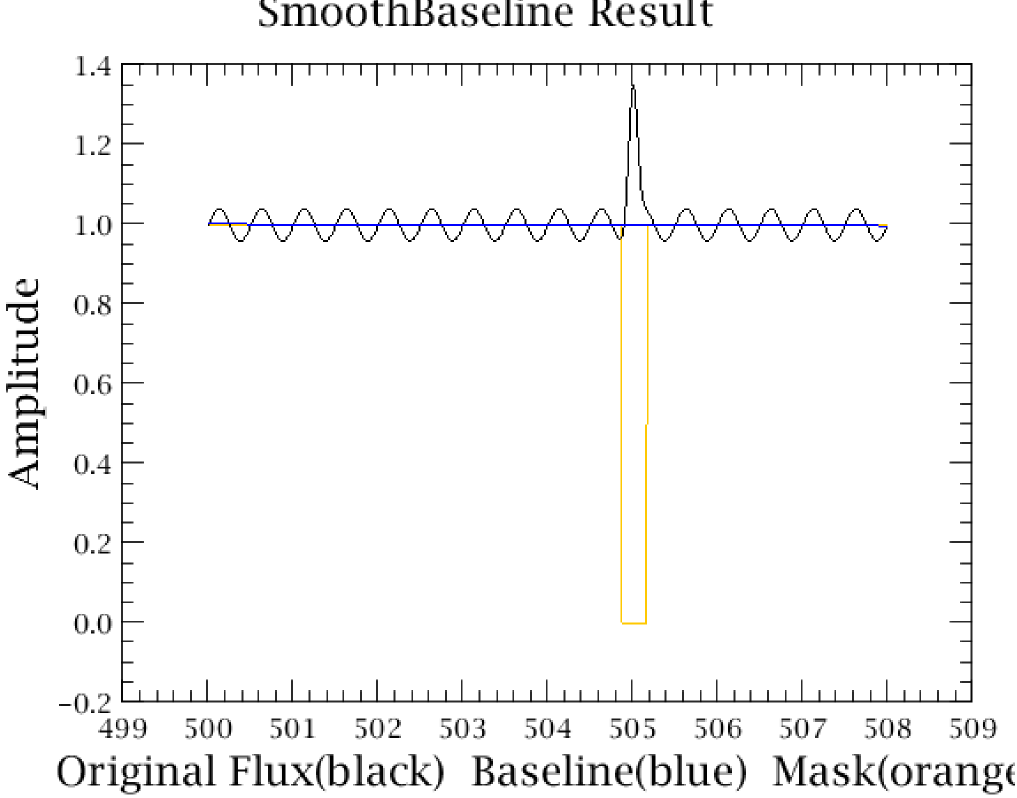
These are Double1d and can be converted into a Spectrum1d that can be worked with a plot in the Spectrum Explorer following the method described in the *Scripting Guide*, Section 3.2.1 in *Scripting Guide*:

```
mySpectrum1d = Spectrum1d(flux, wave, weight, flag)
```

The script generates the following plots:



The first plot generated by SmoothBaseline shows the initial baseline (blue) and the limits above and below which signal will be masked (green). Clearly the emission line is masked, as indicated by the orange line.



The second plot generated by SmoothBaseline shows the baseline in blue and the masked regions indicated in orange.

Chapter 6. Spectral analysis for cubes

6.1. Summary

This chapter tells you about working with spectral cubes in HIPE: exploring the cubes spectrally and spatially and running various extraction and manipulation tasks on them. The access point for viewing cube spectra is the *Spectrum Explorer*, and it is from the Spectrum Explorer that you can access the *Cube Toolbox* and the *Spectrum Toolbox* (and also the Spectrum Fitter GUI, but for this see Chapter 7).

It is the toolboxes that contain the tasks that do the "extraction and manipulation". The individual tasks can all also be run from the command line and can be called up from the *Tasks* pane of HIPE; but the advantage of using them through the Spectrum Explorer is that running tasks side-by-side, and specifying imports and looking at results, is a more comfortable process.

This chapter is organised around the things that the astronomer would do with their cubes, whether this is in the Spectrum Explorer itself, or found in one of the toolboxes.

This chapter describes:

- Some basic concepts about the spectral tools and Herschel spectral cubes: Section 6.2. Important information about cube coordinates: Section 6.3. Important information about the flag, weight, and error datasets of cubes: Section 6.4.
- The Standard Cube Viewer, which displays and can extract spatial slices of your cubes: Section 6.5.
- How to plot the spectra in your cubes with the Spectrum Explorer: Section 6.6.
- Working with the cube spectra, Section 6.7, including
  - spectral arithmetics, averaging and summing spaxels/pixels, statistics
  - manipulation (smoothing, resampling, gridding…), flagging
  - extracting along the spatial or spectral axes, converting units
  - making flux and velocity maps non-interactively: Section 6.7.10.
  - fitting and remove the continuum from a cube: Section 6.7.13.
  - exporting spectra to ASCII (and FITS): Section 6.7.15.
- Baseline fitting and smoothing: this is mainly aimed at HIFI data but can be used by all instruments. We refer you to the previous chapter, Section 5.5, to learn about this.
- How to combine full SED spectra of point sources from PACS and SPIRE cubes: Section 6.8.

The tasks of the Spectrum and Cube toolboxes will change the data within the input products, according to the action requested. Most of the task do not change the class of the product—a cube in usually means a cube out, even if only some of the spectra have been altered. Most of the tasks copy over the entire Meta data of the input product to the output product, and where appropriate new Meta data are added to the end of the Meta data list, and these should also find their way into the FITS files if you save the result to disk: parameters such as reference wavelengths, or the coordinates of the extracted ranges, etc.

To learn how to get data into and out of HIPE, see the Data I/O chapter Chapter 1.

To learn about more advanced scripting when working with spectra, see the *Scripting Guide*. You will also find there information about the different spectral classes defined in HIPE, which are mainly `Spectrum1/2d`, `SimpleSpectrum` and `SpectralSimpleCube`.

Please do read all of the preparatory sections of this chapter before using the Spectrum Explorer: Section 6.2; and additionally Section 6.3, Section 6.4, and Section 6.7 before working on your spectral cubes.

6.2. Cubes and the Spectrum Explorer

Cubes from all three spectrometers on board Herschel (and spectra from other observatories) can be viewed, interacted with and modified by the same set of tools in HIPE. There are some instrument-specific viewers and tools, but these are described in the instrument data reduction guides or other instrument-specific documentation.

The default viewer on cubes is the *Spectrum Explorer*, which you can select with a double click on your cube in the *Variables* pane of HIPE. This viewer allows you to:

1. plot and overplot the spectra from your spaxels/spatial pixels, and
2. access the *Spectrum Toolbox* ( ), the *SpectrumFitterGUI* (described in Chapter 7,), and the *Cube Toolbox* ( ).

The Spectrum Explorer is a *viewer*. It is not a toolbox in of itself. The Spectrum Explorer will allow you to view the spectra of a cube (or any other spectral product), overplot different spectra or different cubes, and make nice plots. In addition, when you are using the toolboxes the Spectrum Explorer can be used to identify spaxels, or whole spectral and spatial regions, that the toolbox tasks should work on. In most cases, the icons that allow you to do these selections are in the button bar at the top of the Spectrum Explorer.

What types of cubes can the toolboxes work with?
Spectral cubes come in various flavours in the framework of HIPE, and are explained in detail in the *Scripting Guide*. For all instruments the final cubes are of class `SpectralSimpleCube`.

- For PACS there are several Level 2 cubes: projected, drizzled, and interpolated cubes, which are of class `SpectralSimpleCube`, and the rebinned cubes, which are of a PACS-specific class —`PacsRebinnedCube`—and will only be accessible to the spectral tools if you are working on a PACS build of HIPE. (Level 0.5 and 1 PACS products will also load into the Spectrum Explorer, but working with these cubes is explained in the PACS Data Reduction Guide: Spectroscopy.)
- HIFI cubes (mapping mode observations) are found at Level 2.5; SPIRE (mapping mode observation) cubes are found at Level 2, and PACS cubes are found at Level 2 and 2.5.
- Spectra extracted from the cubes from all three instruments will be either `Spectrum1|2d` or `SimpleSpectrum`.

All of these classes of cubes and spectra are known to HIPE as "SpectrumDatasets" (often "datasets" in conversation) and implement what is known as the "SpectrumContainer" interface, which allows them all to be visualised and interacted with in the same way. Any SpectrumContainer will be accepted by the Spectrum Explorer. A common class of spectral product is the `Spectrum1d` and `Spectrum2d`. These can hold single spectra, and also multiple-spectra datasets but with the data held in rows rather than a cube arrangement.

Why should you care about the class of the product that you are working on? In most cases, if you are running a task on a cube, then the output will also be a cube even if only some of the spectra have been changed. However, some of the Spectral Toolbox tasks will return the data as `Spectrum1|2d`, even if the input is a cube. In this chapter we will inform you where this is the case.

Note

The Cube Toolbox will only work on `SpectralSimpleCubes` (and the instrument flavours of these cubes). If you have a `SimpleCube` instead, you can convert with the simple command:

```
# what is the class of a cube?
print cube.class
# create a SpectralSimpleCube from SimpleCube ("cube")
specCube = SpectralSimpleCube(cube)
```

Example 6.1. Creating an SpectralSimpleCube object.

A word about arrays within cubes. The data within the cubes—the fluxes, errors, etc—are held as arrays, except RA and Dec which may be described rather by the WCS attached to the cube. HIFI cubes also describe the frequency grid via the WCS, SPIRE and PACS use an "ImageIndex" array for their standard products, and the PACS so-called equidistant cubes use the WCS. This is nothing the user needs worry about—the various spectral tasks know where to look for the arrays they need. For more information on the weights, errors and flag arrays, read Section 6.4.

Tip

Spaxels and (spatial) pixels: mean the same thing, but HIFI uses "pixel" while PACS and SPIRE use "spaxel". These are the spatial-spectral unit of the cube, so one spaxel/pixel is one spatial element of your cube (one "square") with a full spectrum contained within it. If you change your spatial grid, e.g. by regridding the cube, the spaxels/pixels are still called spaxels/pixels, it is just that their size has changed.

6.3. A message about cube coordinates and the WCS

Cubes are three-dimensional. To know the length of a cube's dimensions, type:

```
print MyCube.dimensions
#array('i', [2267, 8, 6])
```

Example 6.2. Printing a cube dimensions

where the first number is the length of the wavelength/frequency grid, and the last two numbers are the spatial dimensions.

If you want to work on a single spaxel/pixel of a cube, especially via typed commands, you will need to know its coordinates. There are two ways that coordinates for cubes are specified: either by spaxel row,column number (e.g. 0,0; 1,2...) or by so-called "spectral index" (0,1,2,3 ...), which refers to the placement of the spectrum within the container. (You will never be asked to specify by sky coordinate.) Most tasks accept cube coordinates, but some still do not.

The spaxel (row, column) number is easy to get. Look at an image of the cube and hover your mouse over it (e.g. use the Spectrum Explorer; or by right-click selecting on your cube in the *Variables* pane, to "Open With" the *Standard Cube Viewer*). The coordinates of the spaxel under the mouse are printed at the bottom left of the display and **are given in the order (y,x), aka (row, column)**. This is also the order that tasks require for cube coordinates. Rows increment as you move your mouse upwards and columns increment as you move your mouse to the right. If your cube has the spatial dimensions (as reported with the command above) of "8,6", then the width of the cube in the image is 6 and the height is 8. The bottom-left spaxel will have coordinate 0,0 in the centre of the spaxel.

The spectral index is harder to find. One way to identify the spectral index of a spaxel/pixel are to convert between them:

```
# for a cube of size 6 wide and 8 high, with dimensions
# (wavelength, 8,6)
```

```
row=8
column=6
for r in range(row):
  for c in range(column):
    idx = (column * r) + c
    print "cube coordinate",r,c," index:",idx

# for the same cube, to go back and forth for a single coordinate
row = 8   # cube dimension
column = 6 # cube dimension
specIndex = 2
c = specIndex %column
r = specIndex/column
print r,c # spaxel coordinate
```

Example 6.3. Printing all the spaxel coordinates that make up this cube.

Another way to identify the spectral index of a particular spaxel is to use the Data tree of the Spectrum Explorer: see Section 6.6.18 for more on this task.

The spatial coordinates for the cube are stored following the **World Coordinate System** (WCS) standard (the WCS is explained in the Scripting Guide in *Scripting Guide*). The two spatial dimentions are defined by a reference spaxel/pixel, a reference value, and a delta value. The third (spectral) axis can also be specified in the WCS if the axis is sampled on an equidistant grid (as is the case for HIFI but only some SPIRE and PACS cube). In that case, it is also defined by a reference bin, a reference value and a delta value. If the spectral axis grid is not equally spaced, it can be defined as an ImageIndex, which will be a separate dataset in the cube product (and a separate extension in the FITS file when the cube is exported from HIPE; you can learn more about this in the Scripting Guide in *Scripting Guide*). The following methods exist to extract and convert coordinate values in the cube WCS object:

```
# Get the world coordinates of the bottom left spaxel (0,0) [open the
# cube in any viewer to see the "bottom left"] and the first element in
# the spectral axis:
# Ra, Dec
print myCube.wcs.getWorldCoordinates(0,0)
# wavelength/frequency (if stored in the WCS)
myCube.wcs.getWorldCoordinateZ(0)

# Get the pixel coordinates corresponding to a certain position in
# RA/Dec:
print myCube.wcs.getPixelCoordinates(83.8454, -5.416)
# wavelegth/and frequency:
print myCube.wcs.getPixelCoordinateZ(461.0407682)

# When the spectral axis is stored as an ImageIndex, it can be accessed as:
print myCube['ImageIndex']['DepthIndex'].data[0]
```

Example 6.4. Getting different coordinates from the WCS information of a cube

6.4. A message about errors, weights, flags

The cubes we deal with at Level >=2 can have some or all of image, wave, imageIndex, segment, error, weight, coverage and flag datasets attached to them (see the *Scripting Guide* for more information).

What is important for the user to know is that which arrays are present depends on which instrument's cubes you are looking at. We have already mentioned that there are some differences in the way the spectral array is held: HIFI cubes describe the frequency grid via the WCS, while most SPIRE and PACS cubes use an "ImageIndex" array. Another difference between the instruments are in the flags, weights and errors. Whether your cube has these arrays is something you can establish by looking at the cube with the Product viewer: for the cube in the figure below there is an image, weight, and flag dataset in the cube:

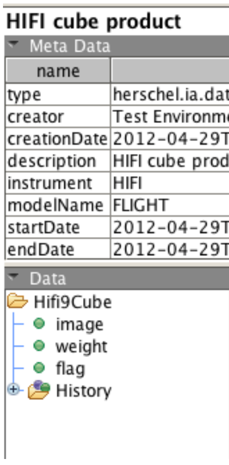


Figure 6.1. HIFI cube: data arrays

Some of the Spectrum Toolbox tasks allow you to specify that these various arrays are taken into account as they work, usually by allowing you to specify a flag or weight *variant* parameter. The default for most tasks is that all variant boxes are checked. But you need to be aware of the following:

- A flag is used to indicate individual datapoints that have problems or are bad. Flags are created by the data reduction pipelines, although users can add their own flags if they wish. You should consult the instrument data reduction guides for more information on flags and their various values. (Note that PACS and SPIRE flags are called "masks" in the pipeline).

  Flags must be contained within an array in the cube called "flag" for them to be considered by the spectral tools described in this section. Note that while the PACS cubes Level 2/2.5 cubes do have this array, it does not indicate the presence of bad datapoints: flagged datapoints are *not* carried into these final cubes when they are created by the pipeline. The "flag" simply carries information about which masks were activated when those cubes were created. It is a fruitless exercise to request the use of the flag array or run any flagging tasks on PACS cubes.

- Weights are an indication of the *relative* importance of the datapoints with respect to each other. HIFI cubes have weights, PACS cubes have weights created from the preceding errors, and SPIRE have weights for some types of cubes, or have instead/also an error dataset. Some spectral tools can consider the weights: if there is no weights or error dataset in the cube (or you do not want the weights to be considered) then deselect the weights *variant* box of the task. The weights are propagated as explained in Section 6.7.9.

- Errors, as stated above, can be found in PACS and SPIRE cubes. When a tool needs to (or is asked to) consider weights, if instead it finds an "error" array it will use that: the errors will be converted to weights as the inverse square, and then errors and weights are propagated as explained in Section 6.7.9.

6.5. A quick cube viewer: the Standard Cube Viewer

This viewer, which you can access via the right-click menu on a cube in the *Variables* pane, will allow you to look at images of each spectral slice in your cube. It looks much like the image viewer (see Section 4.4) but with a slider bar at the bottom to allow you to slide through the layers. You can also input a layer number in the text box next to the slider bar and press Enter to reach a specific layer. With a right click on the image, you can access a menu that will allow you to manipulate the image appearance, such as axes, the greyscale, printing, etc, extract out the current layer to an image, get coordinates, and more. At the bottom-right is a drop-down menu from which you can display some of the other layers in the image, e.g. coverage, error ...

Tip

If you open the display and see only a tiny dot in the middle, you need to zoom-to-fit with the third zoom icon:

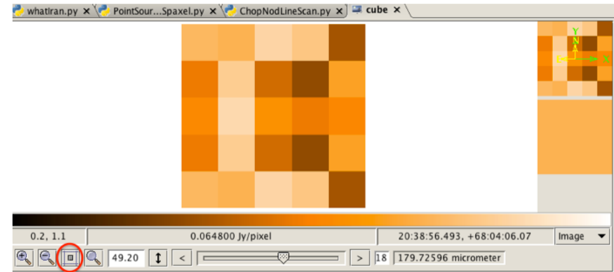


Figure 6.2. The Standard Cube viewer: zoom to fit is indicated

And if you see a blank image, try moving the scrollbar (to move along the spectral axis) and normally the cut levels are set better and the cube image is visible.

6.6. Using the Spectrum Explorer to look at cubes

6.6.1. Opening the Spectrum Explorer on a cube

Quick explanation: Double click on your cube in the *Variables* pane of HIPE and the Spectrum explorer will open in the *Editor* pane. You can drag and drop the GUI out of HIPE and so maximally size it, or you can expand it within the *Editor* pane. In the Spectrum Explorer GUI, a cube image can be seen at the bottom left (the "Data Selection panel") with a plotting pane to its right, and a larger plotting pane (the "Spectrum panel") can be seen at the top (see the figure below).

Longer additional explanation: You can also open the Spectrum Explorer from the command line with:

```
myPlot = openSE(myCube)
```

Example 6.5. Opening the Spectrum Explorer and plotting a cube.

The Spectrum Explorer opens in a new tab inside the *Editor* view. The tab title is always *plot* if opening from the command line, otherwise it is the name of the cube it was opened on.

The `openSE` command has a *`display`* parameter. When this parameter is set to `1` or `True`, the command opens a more limited version of the Spectrum Explorer, without the data selection panel, and plots all the spectra at once to a variable of type `SpectrumPlot`, called `myPlot` in the following example:

```
myPlot = openSE(myCube, display=1)
```

Example 6.6. Opening the Spectrum Explorer and plotting all spectra from a cube.

The Spectrum Explorer is divided into three panels:

- The *Spectrum panel*: the spectra are displayed and interacted with here. At the top of this is a button bar with icons that will be explained in this chapter. Many of these icons can also be found from the menu obtained with a right click inside this panel.
- The *Data Selection panel*: from here you select spaxels/pixels to plot. The selections are done from an image of the cube, and until you select a spaxel/pixel no spectrum will be displayed.
- The *Preview panel*: which gives you a real-time spectral display of the spectrum of the spaxel/pixel under the mouse.

- You can resize these panels by dragging the divider bars and you can maximise any of the panels by clicking the small black arrows on the divider bars.

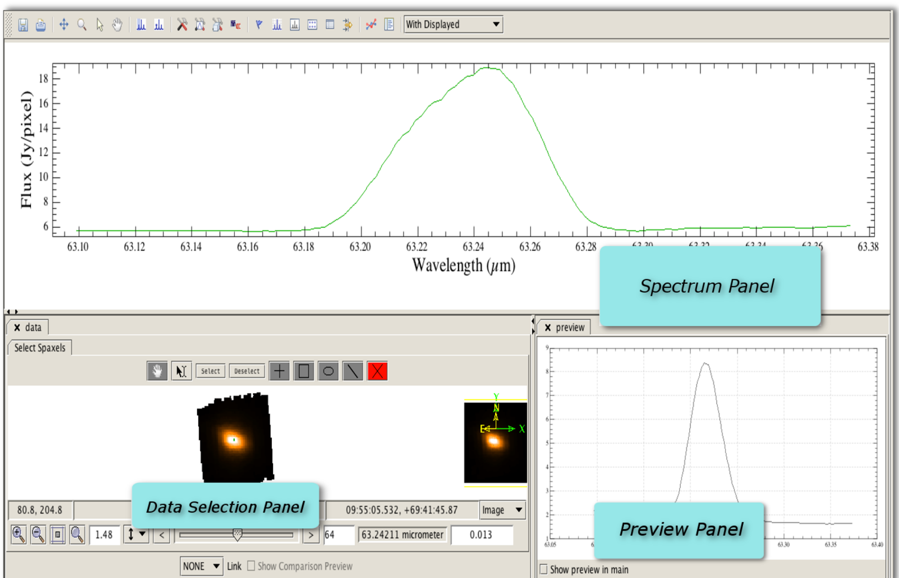


Figure 6.3. The Spectrum Explorer with a cube loaded

The red vertical line that you will see in the spectrum plot (not shown in the image above) identifies the layer, i.e. the wavelength/frequency point, that the cube image shown in the *Data Selection panel* has been built from. The array position and wavelength/frequency of this layer are listed at the bottom of the *Spectrum panel* when the mouse is over the spectrum. You can use the slidebar to scroll through the layers, and the red line will move on the plot with this scroll. If the cube image in the *Data Selection panel* is blank upon first loading into the Spectrum Explorer, a simple shift of the slidebar will usually suffice to set the cut levels more usefully.

Tip

The cube image in the *Data Selection panel* is controlled by the image viewer (Chp. 4). There are a number of HIPE preferences that can be set which may be useful, in particular the "Autoscale layers for cubes" and "Display pixels with flagged values" preferences. Working with these preference options is also possible via a right-click on the cube image.

You can adjust the displayed size of the cube image in the *Data Selection panel* with the other buttons at the bottom of the panel. Zooming in and out on the cube image is done with the magnifying lens icons, and zooming to fit and centre is done with the square icon next to those. You can also type the zoom factor directly into the box to their right.

It is often easier to work in the Spectrum Explorer if you undock it.

From inside the *Spectrum panel* and from inside the *Data Selection panel* you can access their menu of possibilities with a right click. Many of the menu items accessed from within the *Spectrum panel* are also accessible from the button bar at the top of the Spectrum Explorer GUI. The menu you get from within the *Data Selection panel* refers to manipulation of the cube image, and are the some of the same menu items found in the Standard Cube Viewer (Section 6.5). Via the drop-down menu to the bottom-right of the *Data Selection panel*, any other datasets in the cube (coverage, error...) can be displayed in the image. However, note that the *Preview panel* will always only show the spectrum plot, and moreover you can only make spaxel selections when viewing the image dataset.

Note

For some instrument builds there may be additional icons in the button bar at the top of the Spectrum Explorer or additional tabs within the GUI: these will be functions that only work on the data from that instrument, and for instructions on using those you need to read that instrument's data reduction guide.

6.6.2. Showing and hiding cube spectra; clearing stubborn spectra

Quick explanation: once you have loaded a cube into the Spectrum Explorer you can:

1. click on a **spaxel/pixel** in the cube image in the *Data Selection panel*, and its spectrum will appear in the *Spectrum panel* at the top of the GUI; click on another to also display its spectrum; the colour that the selected spaxel/pixel is highlighted in will also be the colour of its spectrum in the plot; re-click on a spaxel to deselect it.
2. to select a spaxel **shaped area** to see all the spectra of, select one of the area icons (,,) from the *Data Selection panel* and then click on the cube image to create, and then resize and move, the chosen area.
3. move or resize a **shape** after you have left it, click on the edit icon , then the shape, on the edges to resize or drag the centre to move it.
4. to remove a **shape**, click on the edit icon , then the shape and then click on the remove icon .
5. to remove everything that has been selected, click the "select" box upon which everything becomes highlighted in blue, and then chose to remove them.
6. after a removal, you can select new spaxels or areas to display by clicking the "single spaxel selection" icon or on one of the shapes again.

Longer additional explanation:

- No spectra will be shown initially when you open Spectrum Explorer. When opening from the command line, you can force Spectrum Explorer to plot all the spectra in a dataset upon opening, with:

  ```
  myPlot = openSE(myCube, display = 1)
  ```

  Example 6.7. Opening the Spectrum Explorer and plotting all spectra from a cube

  To remove the spectra you may need to use the "With Displayed" option explained below.

- in the *Data Selection panel* is for panning the cube image, useful for when you have zoomed on it and want to access a part that has fallen out of the panel display space.
- **Spectra can also be cleared** from the plot by using a drop-down menu on the very top-right of the Spectrum Explorer GUI. This will have the words "With Displayed" or "With Selected" as soon as any spectrum is displayed or has been selected (Section 6.7.2 and later here). Click on this box to access a drop-down menu, from where you can select "remove from plot":
  - This removes all the displayed or selected spectra from all the plots and subplots in the *Spectrum panel*, and *so is also a last-resort way to remove spectra* that you displayed using a task accessed via the Spectrum Explorer but for whatever reason you cannot now remove.
  - At the same time it will remove the highlight around the spaxels/pixels/shapes that you selected in the *Data Selection panel*.
- To remove individual, or sets of individual spectra by (i) "selecting them" either with a right-click on the spectrum's line or by clicking on the Select ( ) icon, and then drawing a rectangle around the spectrum or spectra you want to select, and then (ii) access the right-click menu Spectrum [Selection] → remove.

Note

The image in the *Data selection panel* is by default created from the "image" dataset. You can display other datasets, such as coverage and error (drop-down menu at bottom-right of the *Data Selection panel*). However, spaxel selection can only be done when the "image" layer is displayed.

Tip

If you don't see a spectrum it could be stuck on a previous X|Yrange. Zoom out: Section 6.6.3.

6.6.3. Zooming and panning

These actions can be accessed from the button bar at the top of the Spectrum Explorer GUI, or from menus that appears when you right-click while over a plot: *Context* and *Auto range*. *Note that when you select an action, that remains active until you select another action.*

From the icons you can:

: *zoom mode*. The default mode when the Spectrum Explorer is started. Change the horizontal and vertical plot ranges by drawing a rectangular box using the left mouse button. Ctrl + left [Cmd + left for macs] click of the mouse will zoom to fit the plot. You can also, at any time, use the mouse wheel to zoom in and out of a plot.

: *pan mode*. Pan through the spectrum in the plot window by depressing the left mouse button and dragging the mouse, doing this on either axis for 1d panning or in the middle of the plot for 2d panning.

From the "right-click inside a plot" context menu, the zoom and pan are available from the *Context* menu and from *Auto range* can you also zoom out to encompass the entire spectrum. You can choose to include or exclude flagged data points (*with flags* or *without flags*) in the automatic ranges.

6.6.4. Real-time spectrum display: preview panel

The Preview panel at the bottom right of the Spectrum Explorer GUI shows a real-time update of the spectrum of the spaxel/pixel under the mouse. If you check the small box at the bottom left of this panel, "show preview in main", you will then see the real-time update spectrum also in the main plot of the *Spectrum panel*.

6.6.5. (Over)plotting spectra from multiple cubes

Quick explanation: drag and drop a second cube or any other spectrum dataset into the *Spectrum panel* of the Spectrum Explorer, and from there you can select spaxels/pixels to see the spectra of. If the units and labels of the new cube/spectrum dataset are the same as that of the first cube, the spectra are overplotted, otherwise the spectra are instead plotted in a new subplot in the *Spectrum panel*.

Longer additional explanation:

- Dragging a non-spectrum Product from *Variables* into the Spectrum Explorer causes a Data Tree tab to open in the *Data Selection panel*: see Section 6.6.18.
- If you opened the Spectrum Explorer from the command line, you can add all the spectra in a new variable (`myNewCube`) to the plot with:

```
myPlot.add(myNewCube)
```

Example 6.8. Adding cube data to a Spectrum Explorer instance

This will at the same time plot all the spectra of that cube if you used the display=1 option when creating "myPlot".

Adding a new variable to the plot adds a new layer to the plot. The new layer's wave and flux units and descriptions are compared with those already plotted. If the two sets of values are compatible (or one set of values is not defined) then the data are all displayed in the same plot. If the values are not compatible (e.g., different units or same units but a different axis label) then the new data is displayed in a new plot (a *subplot*) in the *Spectrum panel*.

6.6.6. Linking the display of spectra from multiple cubes

If you have more than one cube loaded into the Spectrum Explorer (see the section above) then you can compare the spectra of these cube directly.

1. Load the cubes you wish to compare into the Spectrum Explorer (open one with the Spectrum Explorer and drag and drop the others to the *Spectrum panel*). If there are also cubes present you do not wish to compare, that is OK.

2. At the bottom of the Spectrum Explorer you see the following:

Figure 6.4. The cube comparison buttons

When you click on the "None" you can chose how to "link" your multiple cubes together: on the WCS coordinates ("world"), on the [absolute] spaxel coordinates ("pixel"), or not at all. You need to do this for each cube you want to compare; as the default is "None", any cube for which you do not change the "link" will not be compared.

3. To avoid asking to compare one cube on "world" and another on "pixel" (which makes no sense), whatever the last choice you made was, is applied to all the cubes you have chosen to link.

4. Now when you click on the spaxel/pixel of a cube, the spectrum from that cube, and from all the cubes you have linked, for the same pixel or world coordinate will be plotted in the *Spectrum panel* plot.

5. If you select the "Show Comparison Preview" button (see figure above) then in addition you will see the spectra of all the cubes you have linked as a real-time display in the *Spectrum panel*'s plot.

6. Unlinking (select "None") any one of the linked cubes will unlink all and also clear all spectra and spaxel selections.

6.6.7. A grid layout of the spectra in a cube

From the button bar at the top of the Spectrum Explorer (), or from the *Dialogue* menu when you right-click in the *Spectrum panel*, you can view a mosaic/raster plot of a cube:

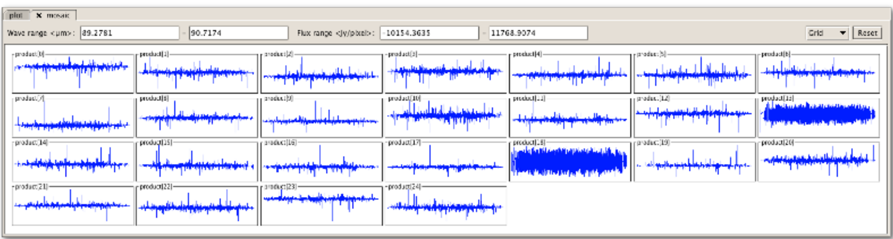


Figure 6.5. Mosaic/raster view

A second click on the icon, or deleting the grid tab, will revert back to just one tab with the Spectrum plot in it.

There are three grid layouts offered (select them from the drop-down menu at the top-right of the panel):

- *Grid*: the spectra plotted in order of their "point spectrum" number. i.e. starting from 0 at the top left and incrementing right and down (see Section 6.3 to learn how to convert between index order and cube coordinate order).
- *Location*: a cross for the central sky position of each spaxel.
- *Raster*: a layout that follows the cube's footprint, i.e. one small spectrum plot for each spaxel and in the correct relative sky locations. You can use the scroll button of your mouse to zoom in and out; for larger cubes this will be necessary.

You can adjust the wavelength and flux ranges for all the small plots using the entry boxes or range bars at the top of the panel.

6.6.8. Viewing in subplots (multiple spectrum plots)

Quick explanation: to open a new plot panel (a *subplot*) in the *Spectrum panel,* go to the location *outside of the plot axes* where you want the new plot to appear, right-click, and from the menu there select *Add subplot*. All subsequent plotting actions will appear in this plot; to activate any other plot —so plotting actions appear there instead—click in the plot you want to be active.

Longer additional explanation: When you open a new subplot, any actions you now do in the *Data Selection panel* or any spectra that will displayed by tasks running from the Spectrum Explorer, will happen in this new subplot. To select a different subplot to work with you can simply click on the subplot, or right-click over it and select *Activate* from the *Subplot* menu.

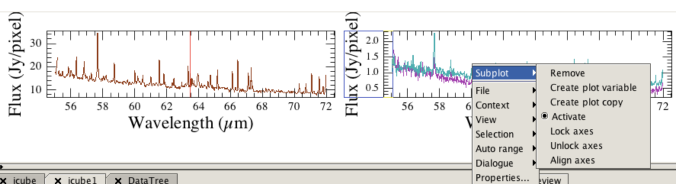


Figure 6.6. The Subplot menu.

Right-click inside any subplot to get a Subplot menu with the following:

- *Remove*: remove this subplot; if this removes all subplots (i.e. you had only one plot) you can add a new one with a right-click access to the menu from anywhere in the *Spectrum panel*.
- *Create plot variable*: explained in Section 6.6.17.
- *Create plot copy*: explained in Section 6.6.9.
- *Activate*: make this plot the active one; clicking on a plot also makes it active.
- *Lock/Unlock axes*: so that scrolling the axis of a subplot does the same on the corresponding axis of other subplots.
- *Align axes*: so that all subplots show the same portion of that axis.

When the Spectrum Panel contains more than two subplots, the options to Align and Lock axes allow the user to select one or more subplots where to apply the operation by pressing Ctrl + left-click for multiple selections. While doing multiple plot axes selections, the cursor will appear as a cross shape and the axes selection in the subplots will result in a blue highlight. To complete the operation, left-click on a blank area of the Spectrum Panel.

6.6.9. Standalone plot panel

You can create a stand-alone spectrum plot using the "Create plot copy" option from the "Subplot" menu accessed via a right-click inside the *Spectrum panel*. This can be e.g.\ to play with to create a plot to print. If you ask for this via the "Subplot" menu, all subplots are included. If the standalone window is grey, a quick resize should bring up the plots.

6.6.10. Changing display axes

When right clicking on a plot axis, you get a context menu with the following options:

- *Axis:* this opens a submenu to change various properties of the axis,
  - *lock/unlock*: if you have several subplots then you can lock axes so that scrolling the axis of a subplot does the same on the same axis of other subplots; and unlock them.
  - *hide/show*: the axis you have clicked on.
  - *align*: if you have several subplots then you can align axes so that all subplots show the same portion of that axis.
  - *add aux axis*: for the x-axis only, this will allow you to change the units of that axis (note: the display is changed, the data are not); once you have added this aux axis, from the right-click menu you can change the units or remove the aux axis.
  - *show grid*: shows grid lines for that axis direction.
  - *invert*: inverts (flips) the axis.
- *Properties:* opens a dialogue window with additional options for customising axes.

6.6.11. Changing plot properties and behaviour

The appearance of your entire plot (the data within it as well as the geography of the plot panel) can be controlled by editing its properties.

6.6.11.1. Appearance of the plot

To change the appearance of a plot, click the icon on the button bar of the Spectrum Explorer to modify plot layout properties, see below. To display or hide the plot grid, the icon should be used.

Within the layout properties panel, you can find options to:

- Display/hide the plot legend (if there is one).
- Switch between line and histogram mode (after disabling *Use automatic line properties* and changing the *Chart type*).
- Show/hide the plot title (if there is one) and update it once displayed.

This is explained with much more detail in Section 6.6.11.5.

6.6.11.2. Editing the *Spectrum panel* properties

To edit the properties of the *Spectrum panel* you should select the icon from the button bar, or chose *Properties* in the menu you get via a right-click in the plot. The properties panel will open in the

right of the *Spectrum panel* and from there you can view and modify any *Spectrum panel* properties (geography, line style, widths, colours...).

There is a separate properties panel for the three elements of the plot—the *layer* (each separate displayed spectrum is a separate layer), the *axis* (x and y), and the *plot* (the box that defines the plotting area(s)). An easy way to select these is to select the *Properties* menu item while the mouse is located on a spectrum (*layer*), on an axis, or in a blank bit of the plotting area. You can also shift between any of these elements by mouse-clicking while holding down the Shift key with the mouse located over a spectrum, axis, or plotting area, once the Properties panel is open.

You can also edit the properties *and actions* for each of these three elements from a mouse menu, as is explained in the next three subsections.

Example of changing a plot property: if you want to change the thickness of a spectrum line drawn on your plot:

1. Choose the *Properties* icon to open its panel.

2. Shift + left click a spectrum's line, i.e. you select the spectrum you want to change the properties of.

3. And in the Properties tab that opens on the right of the plot you can change the "stroke" size in the "Layer Style" part of the tab by simply clicking in the box (will say 0.5 by default), typing in your new value and pressing the Return key.

4. To change the line thickness for any other spectrum, you need to select it (each new spectrum is a new layer!) and then edits its properties.

Layer properties include line thickness, colour, etc. Axis properties includes tick marks, spacing, etc. Plot properties includes the geographic location of the plot/subplots (they get changed together).

You can only change the properties of a single layer (i.e. a single spectrum) in a single subplot at a time, i.e, that spectrum you shift-clicked on. The same applies to changing the axis properties—only the axes of the subplot you selected (via shift-select on an axis of a subplot) will have the properties changed. Plot properties get changed for all subplots displayed.

6.6.11.3. Editing the axis properties

When right-clicking on a plot axis, you get a context menu which includes the following axis properties:

- *Axis:* this opens a submenu to change various properties of the axis: hide main axis or add an auxiliary axis (with different units), show a grid, and invert. If you have multiple subplots in your *Spectrum panel* then you are also offered the chance to lock axes so that scrolling the axis of a subplot does the same on the same axis of other subplots, and unlock them. You can align axes so that all subplots show the same portion of that axis.

- *Properties:* (at the bottom of the menu) opens the properties panel.

6.6.11.4. Editing the plot properties

When right-clicking inside a plot/subplot, you get a context menu which includes the following properties options:

- *Subplot:* opens a submenu with options to remove the subplot (see Section 6.6.8), make it active (so that newly selected spectra are shown in this subplot), or generate a variable representing the plot, so that you can operate on it outside Spectrum Explorer. Note that the variable corresponds to *all* the subplots in the Spectrum Explorer. If you have multiple subplots in your *Spectrum panel* then you are also offered the chance to lock axes so that scrolling the axis of a subplot does the same on the same axis of other subplots, and unlock them. You can align axes so that all subplots show the same portion of that axis.

- *AutoRange:* enables or disables the automatic adjustment of axis ranges as spectra are added and removed. You can choose to include or exclude flagged data points (*with flags* or *without flags*) in the automatic ranges.

- *Properties:* (at the bottom of the menu) opens the properties panel.

6.6.11.5. Editing the layer properties

Each new spectrum you add to a plot is a new layer. Each layer has its own properties, and each layer's properties will be changed independently of the other. By default the layer properties are auto assigned. To change the line style, colour, and line type you should open the layout properties dialog by clicking on this icon of the button bar:

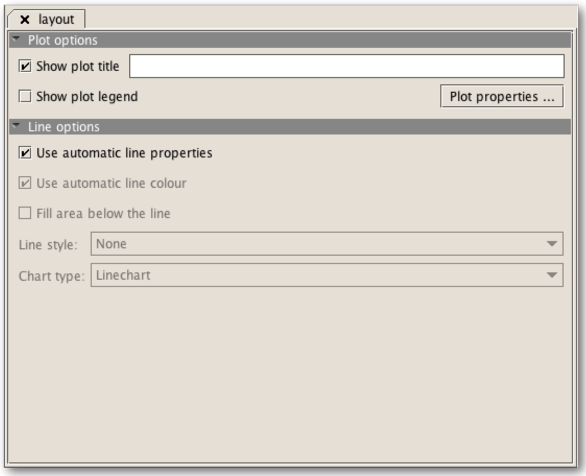


Figure 6.7. changing layer properties

Within this dialog, you can change the following properties:

- Auto or manual line style (this is toggled using the checkbox *Use automatic line properties*); when this is unchecked you can choose the line style you want from the drop-down labeled *Line style*. To control if the spectrum is to be filled, you should check the *Fill area below the line* button.

- Auto or select colour using the checkbox *Use automatic line colour*.

You can reset the changes by selecting *Use automatic line properties* again. All the next plotted spectra will be changed to your new chosen style, and will be so-affected until you either select a new style or go back to auto mode.

6.6.12. A table of the plot—mouse interactions

The Spectrum Explorer provides context-dependent plot interactions, i.e. what a mouse click or movement gives you depends on where the cursor is. The "context" is printed at the left bottom corner of the plot panel, together with the location of the mouse cursor in plot coordinates: "subplot [576.60, 5.6]" is an example. The following table provides some contexts and the mouse interaction behaviour.

Context	Click	Ctrl-click	Drag	Scroll
subplot	Set as 'active'		Zoom/Select/Pan	Zoom
axis			Pan	Zoom
spectrum	Select spectrum	Extend selection	Move spectrum to another subplot	
	Select datapoint		Extract spectrum to a new variable	

Context	Click	Ctrl-click	Drag	Scroll
			Use spectrum as task input parameter	
selection			Same as above	
marker edge			Resize marker	

6.6.13. Changing your Spectrum Explorer preferences

To change the default Spectrum Explorer settings, choose *Preferences* from the *Edit* HIPE menu and go to the *Spectrum Explorer* sections. In addition to global options, there are subcategories for a number of data types (classes). For each data type, you can specify a custom plot title, subtitle and legend, which will be triggered whenever that datatype is read in. Metadata fields and attribute fields can be filled in automatically by specifying the fields name between angular brackets, optionally with a printf-style format suffix. For example, <longitude>%.2f in the legend element field displays the value of the longitude attribute for each spectrum in the legend.

One of the preferences controls how the tabs behave, i.e. when you have multiple products loaded in the Spectrum Explorer. You can chose that the *active* tab, i.e. that which is on the top, is the one that the toolboxes (*Spectrum* and *Cube* Toolbox) preferentially take their input from. Alternatively, you can ask that if a spectrum has been chosen from a tab (and is therefore displayed in the *Spectrum panel*), that will preferentially be taken as the default input to a task.

6.6.14. Viewing plot information

You can view additional plot information by hovering the mouse cursor over the plot. The data will appear in the bottom-left corner of the main spectrum plot panel.

6.6.15. Viewing datapoint flags

: displays flagged datapoints. These will appear as a white cross and a curtain will sit over the plot at those X-axis points. Note that the flags are taken from a "flag" layer in the cube. To see if your cube has a flag dataset (and to see if any flag values are non-zero), right click on it in the Variables pane, select "Open with" and "Product viewer". The data panel there lists the arrays in the cube:

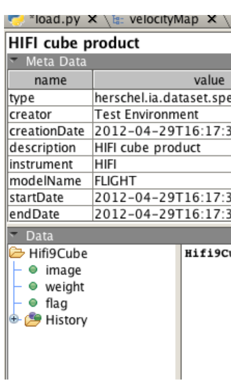


Figure 6.8. The arrays in a cube.

can also be accessed from the menu you get when you right-click on the plot and select *View*.

6.6.16. Printing and saving

These actions can be accessed from the button bar or from a menu that appears when you right-click while over a plot (*File*).

: saves the plot as a PNG, PDF, EPS or JPEG file. If several subplots are displayed, all are in the saved product.

: prints the plot. If several subplots are displayed, all are plotted.

6.6.17. Creating a new variable from a plotted spectrum

When right-clicking inside a subplot (subplots: Section 6.6.8) you get a context menu which includes *Subplot* from which you can select *Create Variable*. This will take the active plot/subplot and make a new variable, which will appear in *Variables* pane, that represents the plot. The idea is that you can work with this outside of the Spectrum Explorer.

The variable is not a spectrum, it is a plot object that you can use in further plotting on the command line. The variables created will be named splot_0|1|2... The following example shows you how to use this, e.g. to open it in a new instance of the Spectrum Explorer:

```
p = splot_0.plot
sp = splot(p)
```

Example 6.9. Creating a plotting variable for later use with splot

6.6.18. A meta data list: and how to relate spaxel coordinates to index coordinates

The icon , found in the button menu at the top or via right-click in a plot and from there in the *Dialogue* menu, allows you to see a Data Panel for your cube. It brings up a new tab in the *Data Selection panel* with a listing of information for each spaxel/pixel—what you are looking at is the Data Tree. This is presented as a table, each row of which is a new spaxel/pixel. The first row is for the entirety of the product. If you click on the "variable" column for a row you see the spectrum in the *Preview panel*, click on the square cell next to that to see the spectrum in the *Spectrum panel*, in a colour corresponding to the colour the cell turns into. Click on these elements of the first row to see the entire cube plotted in the *Preview* or *Spectrum panel*.

The information in the Data Tree comes from the meta data of the cube. However, for cubes this listing is of limited use: if you want to see the meta data you can use the Observation Viewer on the cube, and since there is only one set of information per cube, rather than per spaxel/pixel, the information in the Data Tree is exactly the same for each row. This is more useful for non-cube spectral products, and mainly for the HIFI instrument. See Section 5.3.7 to learn more about using this panel on these other data.

However, one use of the Data Tree for cubes is to help you identify the spectral index values for any particular spaxels. Spaxels can be located via their coordinates (row, column) or (dataset/spectral array) index value (0,1,2,...). Some tasks accept spaxel coordinate inputs and others only spectral index inputs. The coordinates can be found by looking at the coordinate panel at the bottom of the cube image of the Data Selection panel. The index for these coordinates can be found using the Data Tree in the following way:

1. In the *Data Selection Panel*, click on the spaxels you want to select if looking at a cube, or click on the row if looking at a non-cube multi-spectrum dataset.

   The corresponding spectra are plotted in the *Spectrum Panel*. Make sure only the spectra you want to select are there displayed.

2. Click the icon in the toolbar.

The *DataTree* tab opens in the *Data Selection Panel*.

3. Go to the *DataTree* tab. The spaxels you have selected are marked by coloured squares. The numeric index of each spectrum is shown in the *variable* column.

4. You can now create a selection array using the following command in the Console:

```
mySelection = [5, 12, 24]
```

Example 6.10. Creating an array for selecting a particular spectrum from a cube

6.6.19. Filtering what is viewed: not useful for cubes

: opens the filter panel, which appears in a new tab under the *Preview panel*; however for cubes this is not a useful. For non-cube Spectrum Containers this allows you to order spectra loaded into the Spectrum Explorer following attribute [mainly only HIFI data have attributes]. But for cubes the "listing" of spectra (in the *Data Selection panel*) is not a list but a cube image, and for an image the ordering is set by RA, Dec—no other ordering is possible. See Section 5.3.8 to learn more about using this panel on these other data.

This item can be found from the button menu of the Spectrum Explorer or by right-clicking on the plot and going to the *Dialogue* menu.

6.6.20. Plotting from the command line

When making plots for publication or as interim results when running scripts it may be more convenient to create plots via the command line. This is done using *PlotXY* or the command line version of the Spectrum Explorer, *splot* (a contraction of spectrum plot). The usage of these packages are described in the Plotting chapter in this manual.

6.7. Working on cubes: the Spectrum and Cube Toolboxes

In this section we describe the tasks of the *Spectrum* and *Cube Toolboxes* which you can call up from the Spectrum Explorer. These toolboxes include tasks to perform various spatial and spectral operations on the spectra of a cube, for example to add, average, flag, make flux and velocity maps, crop spectrally or spatially, change units and calculate statistics. The main difference between the two is that the Cube Toolbox works on cubes, while the Spectrum Toolbox will work on any spectral product, including cubes. There is some overlap between the two in what they do: the emphasis of the Cube Toolbox is on working with cubes along the spectral and spatial planes, while the Spectrum Toolbox concentrates more on mathematical tasks.

The toolboxes are opened from the icon menu at the top of the Spectrum Explorer GUI, and from within the toolbox tab that opens in the *Spectrum panel* you can select their specific tasks. The individual tasks can also be found in the *Tasks* pane of HIPE; the interaction with a task opened this way is similar to that when opening via the Spectrum Explorer, it is mainly selecting spectra or spectral ranges for the task to work on that differs.

Before continuing with this section you should read Section 6.2 to learn more about working with cubes in HIPE, Section 6.3 to learn more about working with the coordinates of the spaxels/pixel in your cube, and Section 6.4 to learn more about weights, errors, and flags.

See Section 6.6, to learn how to use the Spectrum Explorer. **Please also read the next two sections**, where we explain the behaviour of the Cube and Spectrum Toolboxes working via the Spectrum Explorer: how they interact, and how you input data and inspect the output. If this tool is not familiar to you, you will benefit greatly from reading Section 6.7.1 and Section 6.7.2.

6.7.1. How to open the Toolboxes; getting extra help

Spectrum Toolbox: can be opened from the Spectrum Explorer by clicking on the crossed hammer and screwdriver icon in the toolbar: , or from Dialogue → Spectrum Toolbox in the menu you get via a right-click in the *Spectrum panel*. A new tab will appear to the right of the *Spectrum panel*, and the tasks can be selected from a drop-down menu.

Cube Toolbox: can be opened from the Spectrum Explorer by clicking on the cube+screwdriver icon: , or from Dialogue → Cube Toolbox in the menu you get via a right-click in the *Spectrum panel*. A new tab will appear to the right of the *Spectrum panel*, and the tasks can be selected from a drop-down menu.

For both toolboxes you can also open their individual tasks by name from the *Tasks* pane of HIPE. The task dialogues will open in the *Editor* pane.

Getting help: If you open the tasks via the Spectrum Explorer then you can open the *User Reference Manual* (URM) entry for each task by clicking on the: *Cube Toolbox*: "Help" button at the bottom of the task panel; *Spectrum Toolbox*: small question mark icon at the bottom of the task panel. If you opened the task from the *Tasks* pane, the URM entry for that task can be accessed from a Help button from the task panel, or by right-clicking on the task name in the *Tasks* pane and selecting *Help in URM*. The URM entries for the spectral tasks explain the command-line use of the tasks. Here we will explain the GUI usage.

Also, the following task instructions are written assuming you have already opened the task GUI.

Warning

It is better to create a variable for the cube you wish to work on, rather than to open it from or the `ObservationContext` (or other product such as a `ListContext` as is the case for some PACS products) that the cube is contained within. Many of the tasks change the data, and HIPE will not allow you to change the contents of an `ObservationContext` in this way. So, if working from an `ObservationContext`, first extract the cube out of it, and then work on that.

6.7.2. Defining the input, looking at the output

This is a reference section that tells you how to select the spaxels/pixels that you want any task to run on.

The first tabs of the Spectrum Explorer are the ones that open when the Spectrum Explorer does, i.e. one tab in each panel.

1. More tabs may be added to the *Spectrum panel* as various toolbox tasks display results; remember that the tab that shows spectra selected from the cube is the first, left-most one called "plot".

2. More tabs can also be added to the *Data Selection panel*, either by the toolboxes (in particular the Cube toolbox) or if you drag-and-drop further cubes or spectra to the Spectrum Explorer. Tabs created for a new product loaded into the Spectrum Explorer carry the name of that product. Any tabs that open *within* each product-tab will be created by the toolbox tasks, and these carry the name of their function; remember that the only sub-tab from which spectra can be selected to display is the first, left-most one called "Select spaxels".

About tasks that create new data

For the spectrum and cube toolbox tasks that create new cubes or spectra and at the same time add them to the *Data Selection panel*, the focus may pass to the new product. This should be taken into account for additional runs of the same or other tasks, as the new product will be the new input for the task and some of the parameter entries in the task pane may change. See Section 6.6.13 to learn about setting a preference for this focus (to follow the selected tab or to follow the selected spectra).

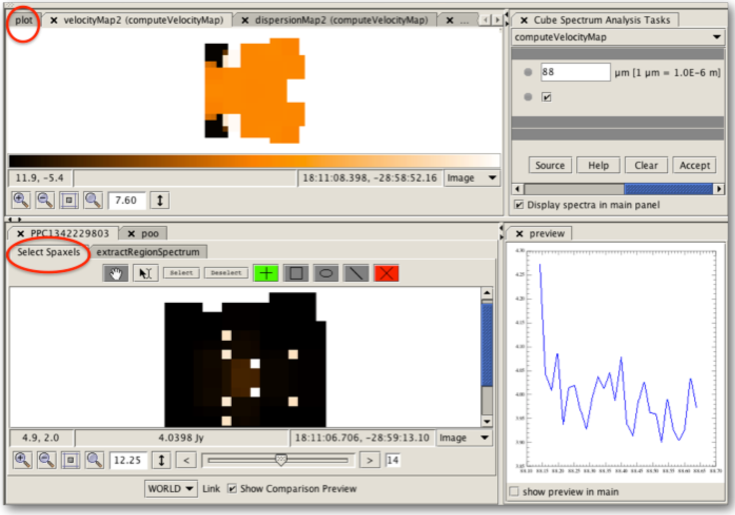


Figure 6.9. Tab arrangement: new tabs will appear in the *Data Selection* and *Spectrum panels* as tasks are run. The active tabs are "plot" (*Spectrum panel*)and "Select Spaxels" (*Data Selection panel*).

6.7.2.1. Input

The Spectrum Toolbox tasks work on one or two spectral products (e.g. cubes), or on selected spectra from the product(s). For almost all tasks, if the input is a cube, the output is also a cube—even if only some of the spectra of that cube have been changed . *The Cube Toolbox* tasks work on an entire cube—you can select out spectral or spatial ranges but the input is always the entire cube. For both toolboxes, if working from within the Spectrum Explorer, selection of spectral or spatial regions is most easily done within the *Spectrum* and *Data Selection panels*. If the task was opened from the *Tasks* pane of HIPE, then the selection is done by typing the selection list into the parameter box, or creating the selection list by typing in the *Console* and dragging it over from the *Variables* to the parameter box in the task's tab.

When using the Spectrum Explorer and the Spectrum Toolbox.

- **To run on one or more displayed spectra** simply click on the spaxels/pixels from the *Data Selection panel* to see their spectra in the *Spectrum panel*, and at the same time this will select them as input for the task. See Section 6.6.2 for more information on plotting spectra. The words "Task is applied on displayed spectrum(a)" will appear in the task panel in the "ds/ds/ds2" parameter box(es), and only these will be input to the task.

- **To run on only a few spectra of those displayed** in the plot, "select" one/some of them by clicking on the selection icon on the button bar of the Spectrum Explorer, ; and then clicking on the spectrum to be selected (on a datapoint in it) or drawing a box around (one or) several spectra to select those. The selected spectra are highlighted. As you select spectra, the words "Task is applied on selected spectrum(a)" will appear in the task panel in the "ds/ds/ds2" parameter box(es), and only these will be input to the task. Click on the same spectrum to deselect, and you can select several one after the other.

- **To select a bunch of datapoints** from one or several spectra—only the flagging task asks for this —select the icon (from the icon bar or the *Context* right-click menu) and then draw a box on the plot that encompasses your datapoint(s); the selected datapoints are highlighted. You can remove one selection by drawing another box around it, and you can extend a selection by drawing a box while depressing the Ctrl key (Cmd on a Mac).

  In the right-click Spectrum Explorer menu, under *Selection*, you can also select all or deselect all spectra displayed.

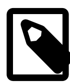
- **To select a wavelength/frequency range** first make sure that a spectrum is displayed in the *Spectrum panel* (any spectrum from your cube will do) chose the icon (from the icon bar or the *Context* right-click menu) and then draw your range on the plot by left-click and dragging the mouse from one end of the desired range to the other; the range is indicated with a grey curtain.

  Ranges can be removed by right-clicking on the drawn range and selecting "remove" or "remove all" from the *Range* menu. It is good practise to clear ranges that were drawn previously to create input to a toolbox task, before drawing any new ranges. Ranges can be resized by clicking near the edge of the marker (a resize symbol will be seen) and dragging the edge of the marker to the desired position. The right-click menu also gives you the option to change the colour of the marker and of the line.

Tip

When selecting ranges to input into a task, it is a good idea to first clear out all previous ranges, otherwise they may be carried forward to the current or a subsequent task.

- **To run a task on an entire cube** is simply a matter of activating the *Data Selection panel* tab that holds the cube you want to use. If you have selected a spectrum (from that or any tab) you may need to clear it first. See Section 6.6.13 to learn about setting a preference for this focus. To run on a cube not loaded in the Spectrum Explorer, you can drag the cube from the *Variables* pane and drop it onto the circle next to the "ds/ds1/ds2" parameter box(es) in the task panel (the circle is grey if nothing has been selected/displayed and green if there are spectra shown in the plot or if you have dragged a cube into it). You will now see the words "Task is applied on variable [name]" appear in the parameter box.



Note

Although it is possible to select spaxels/pixels to plot from two cubes open in the Spectrum Explorer, you must avoid doing this when running toolbox tasks.

When you select spectra via the Spectrum Explorer in this way, you can leave the *selection* parameter that many tasks have as a parameter empty.

In summary, most tasks will work automatically on the product in the active (top-most) tab, or in the displayed or selected spectra, and the order of preference can be set in the HIPE Preferences (Section 6.6.13).

When using the Spectrum Explorer and the Cube Toolbox.

- **To select a wavelength/frequency range** is the same as working with the Spectrum Toolbox (see above).

- **To set the cube to work on**: it is good practise to first bring to the front, in the *Data Selection panel*, the tab with the cube you want to run the Cube Toolbox on, and then open the toolbox. To run on a new cube that is open in the Spectrum Explorer, close the Cube Toolbox, bring that new cube to the front, and open the Cube Toolbox again.

- **To select spatial regions from that cube** is a function that is provided by the tasks themselves and so is explained later: but FYI the approach is to select a shaped region from a cube image that appears in a new tab of the *Data Selection panel*.

When calling Spectrum Toolbox tasks from the *Tasks* pane. You are therefore working in the tasks' own GUI, which will open in the *Editor* pane. The input cube can be dragged and dropped from the *Variables* pane into the grey circle next to "ds/ds1/ds2" in the task GUI, upon which the grey circle turns green. Replacing the input with another is done with the same action. To select spaxels/pixels from within the cube to run the task on, you need to input a jython list of coordinates:

- First identify the cube coordinates you want, e.g. by hovering the mouse over the cube selection image in the Spectrum Explorer or in the Standard Cube Viewer [right-click on the cube in the *Variables* pane to select this viewer]; the coordinates that the viewers show at the bottom left of the cube image are (row, column). For more information, see Section 6.3.

- Define a selection as a list: let's say you want (row, column) (1,2), (2,2), (4,5) then your selection variable is created by typing, in the *Console*:

```
sel=[(1,2),(2,2),(4,5)]
```

Example 6.11. Creating a selection array with spaxel coordinates.

- Some tasks allow you to type this list directly in the parameter box, for others you need to create the variable on the command line of the *Console* and then drag and drop it from *Variables* pane to the parameter box of the task. When running from the command line, you can usually define the parameter "selection" as: `selection=sel` or `selection=[(1,2),(2,2),(4,5)]`. If you want to work on the entire cube, you leave this parameter blank.
- If the task requires spectral index selection coordinates, rather than spaxel coordinates (see Section 6.3 to learn more about coordinates), you can identify the index values for the spaxels using the Data tree of the Spectrum Explorer: Section 6.6.18. In this case the list will look like this:

```
sel=[24,25,26]
```

Example 6.12. Creating a spectrum selection array.

- Spectral ranges can be input also as a jython list, and usually the values in the list are in the wavescale of the data.

When calling Cube Toolbox tasks from the *Tasks* pane. To select spectral and spatial ranges to run on:

- Spatial ranges are defined as row and column indices given as single numbers typed into the parameter box.
- Spectral ranges are defined as a listing of spectral indices or wavescale values. You simply type the numbers into the parameter box: 1 2 3 (not 1,2,3, or [1,2,3]).

A comment about some of the selection parameters that many Spectrum Toolbox tasks have. The spectra contained in a `SpectrumDataset` can contain a number of so-called Spectral Segments (see the *Scripting Guide* for more information). For example, each subband in HIFI (non-cube) spectra is a Spectral Segment—with each segment identified in the *Data Selection panel* of the Spectrum Explorer by a different box. For such products you can often choose which Spectral Segments to work on using the "segments" field in the panel of the Spectrum task you are running. You can also select on so-called attributes, which for HIFI could be the things such as the observation start time of that data frame or the position of the chopper. *However*, you should note that (i) for Level >=2 cubes each spaxel/pixel contains only one segment, and (ii) each spaxel/pixel has the same attributes so you cannot differentiate between them. Hence, when working on these cubes, you can ignore the parameters *segments*, *selection_lookup* and *attributes.*

Warning

The Spectrum Explorer will allow you to load more than one SpectrumContainer into it (e.g. two cubes). You could find yourself plotting spectra from different cubes and then trying to run a Spectrum Toolbox task on them. For most tasks this will fail, as they work with selections from a single cube only.

You will realise you are doing this if you see the "ds" parameter in the Toolbox displaying a small red cross. To correct this, clear the plotted spectra from the cube you are not interested in, or instead select only those you are interested in.

6.7.2.2. Output

All tasks send their output to the *Variables* pane of HIPE. From there you can select any viewer (including another instance of the Spectrum Explorer) to see them.

Many tasks will echo their commands to the *Console* of HIPE (this offers a quick and dirty way to see how to call a task from the command line).

Tasks that make selections or focus on parts of the input spectra following the user's request—a wavelength to take for the zero velocity, spaxel coordinates, etc—will add Meta data to the output, so you can know what the value of the selection/focus that created that output was.

- **For the Cube Toolbox**:
  - Some of the tasks send the products created to new tabs in the *Spectrum panel*, but these are displays only—you cannot perform any selections from them.
  - When you run a Cube Toolbox task twice, it will "overwrite" the previous output tabs—this is to avoid cluttering up your space with tabs. But the products are still in *Variables* and so can be loaded into the *Editor* pane to be viewed from there.
- **For the Spectrum Toolbox**:
  - Some tasks send created cubes to the *Data Selection panel* as new tabs and these are active displays—you can perform spatial selections from them.
  - Some tasks send newly created spectra to the *Spectrum panel* "Plot" tab.
  - Some tasks will send tabular output to the *Data Selection panel* as active displays.

You can delete any tab—the variable they "belong" to will not also be deleted. To remove them you simply click on the "x" on the tab. Removing the product from the *Variables* pane will also not remove the tab.

6.7.3. Spectrum extraction and cube cropping

There are four tasks to extract spectra from cubes or to crop a cube spatially or spectrally. Two—*select* and *extract*—are in the Spectrum Toolbox and the others—*extractRegionSpectrum* and *cropCube*—are found in the Cube Toolbox. Briefly, the differences between these are:

- *select* can be used to extract any spaxels from a cube, and these are then placed in a `Spectrum2d`, which is a row-stacked spectral product rather than a 3d cube.
- *extract* can do the same as select and it can also be used to crop a cube spectrally, in which case the output product is a cube.
- *extractRegionSpectrum* can be used to extract out a single spaxel from a cube and place it in a `SimpleSpectrum` (i.e. a single spectrum product); it will also produce an averaged or summed spectrum from a spatial region.
- *cropCube* can be used to crop a cube spectrally and/or spatially, and the output is a cube.

There is additionally one PACS-specific task that is offered in the Spectrum Explorer running in a PACS-version of HIPE, to extract out a single spectrum from a cube and turn it into a `SimpleSpectrum`: ConvertPacsProduct2SimpleSpectrum. This will only work on a cube of class `PacsRebinnedCube`. The spaxel coordinates must be specified in the task pane, and the slice number can always be left on 0 when used in the Spectrum Explorer. See the *PACS URM* entry for this task to learn more about using it on the command line.

6.7.3.1. *select* and *extract*: spatially and spectrally extract from a cube

From the Spectrum Toolbox you can find the "select" and "extract" tasks (see Section 6.7.1 for instructions on opening the Spectrum Toolbox). You can also select these tasks directly from the *Tasks* panel.

Note that for these tasks the "selection" is on spectral index number (0,1,2,3...: more specifically, it is the PointSpectrum number) rather than cube coordinates ([0,0], [0,1],...) so you will need to read Section 6.3 to know how to translate between the two if you run them on the command line or via the GUI.

- *select in HCSS User's Reference Manual* is to choose random spaxels/pixels to take out of the cube and put into a new, `Spectrum2d` product. Do not confuse this with "selecting" spectra to use immediately as input to another task.

  To identify the spectra that the task should select out, you need to plot them: see Section 6.7.2. Press *Accept* and the task is executed and a new product, by default called "result", will be created and added to the *Variables* pane, and a new tab containing this product will appear in the *Data Selection panel* if you are working via the Spectrum Explorer. In Section 6.7.2 you will also find information about the parameters "selection", "segments" and "selection_lookup".

- *extract in HCSS User's Reference Manual* will select out spectra and input them into a new product. You can specify a spectral and or and spatial range to extract. If you set a list of spectra to extract the output is a `Spectrum2d`, if you define only a spectral range the output is a `SpectralSimpleCube`, and if you define both the output is a `Spectrum2d`.

  To identify the spectra that the task should run on, see Section 6.7.2. Identify the wavelength/frequency range to extract over using the "select ranges" icon in the button bar (also explained in Section 6.7.2), or you can write the range value into the boxes in the GUI, pressing the Return key to add extra ranges. After selecting the input cube/spectra, press *Accept* and the task is executed and a new product, by default called result, will be created and added to the *Variables* pane.

  You can select more than one range to extract, and they do not have to be continuous. (You can open the output product in the Spectrum Explorer and inspect it by selecting from its row-listing in the *Data Selection panel*).

Tip

For both tasks, if you are running this task from the Spectrum Explorer then you have no choice over the order the spectra are selected from the cube and placed into the output product: it is in index numerical order. If you want any other order you should run the task via its own GUI or on the command line: create a selection PyList (see Section 6.7.2). Once the selection array has been created, drag and drop it from the *Variables* pane into the "selection" grey circle of the task panel.

6.7.3.2. Extracting out a single spectrum with the Cube Toolbox and changing the units

To extract out a single spectrum using the Cube Toolbox use the *extractRegionSpectrum in HCSS User's Reference Manual* task. This task also allows you to extract out an average or sum of a region but that will be explained in Section 6.7.5.2.

See Section 6.7.1 for instructions on opening the Cube Toolbox, and from there select "extractRegionSpectrum". Then do the following, in this order

1. First, select the "SINGLE_PIXEL" from the "regionType" parameter box.
2. Then, go to the *Data Selection panel* to the sub-tab "extractRegionSpectrum" there. Select the icon, and then click on the single spaxel/pixel that you want to extract out. Note that the spectrum will not appear in the Spectrum panel when you do this. The coordinates are sent to the appropriate parameter box.
3. From the "arithmetics" menu you then chose to average or sum (these are clearly the same for a single spaxel/pixel), the units of the output will be [flux]/pixel (i.e. flux in an area of a single spaxel of that cube).

4. If choosing the average option, you then also have the choice of selecting the `doAreaConversion` option. The flux is converted by the area of the region selected (in this case, one spaxel), and will be in units of [flux]/Sr, converted by:

   a. getting the area of the spaxel in square arcsec from the WCS

   b. dividing the flux by this area

   c. converting from e.g. Jy/sq_arcsec to e.g. Jy/Sr by multiplying with $(3600*180/\pi)^2$ (conversion factor $arcsec^2$/steradian).. *This conversion* assumes that the input units are [flux]/pixel *(type "print cube.getFluxUnit()" on the command line to check)—this is the case for most Herschel cubes. Your fluxes will be weirdly converted if they are already in units of flux per area* instead, e.g. Jy/beam: the "/beam" part is not converted, so your final unit ought to be e.g. Jy/beam/Sr, and will be correctly calculated for such a unit, but the unit as reported in HIPE will just be Jy/Sr.

Upon pressing *Accept* a `SimpleSpectrum` product is created and added to *Variables*. A display plot of that spectrum will appear in the Plot tab of the *Spectrum panel*.

6.7.3.3. Cropping a cube spectrally and spatially with the Cube Toolbox

To crop a rectangular region and/or to crop a spectral region of a cube, you can use the Cube Toolbox. See Section 6.7.1 for instructions on opening the Cube Toolbox. The task you want is *cropCube in HCSS User's Reference Manual.*

1. When you select this task a new sub-tab will appear in the cube's tab in the *Data Selection panel*, from where you can select the spatial range you want to crop over: select the rectangular region icon, click on the cube image, and then click on the rectangle that will have appeared there to move and resize it. The parameter boxes col|row|Min|Max will be filled with the limits you have chosen (given in spaxel coordinates rather than image coordinates: see Section 6.3).

2. If this is all that you want to do, now press *Accept*.

3. If you wish to also, or instead only, define a wavelength/frequency range, the ranges must be entered as the parameters startWave|endWave—you can enter the values in the parameter boxes manually or define the range using the "select range" Spectrum Explorer icon from the button bar: (see Section 6.7.2). You can only select one wavelength range (the first you define), and it is good practise to clear out any existing ranges first (right-click to access Range→Remove). When running this task from the command line, you can define the spectral region to extract either in wavelength/frequency space or as array index (see the URM in *HCSS User's Reference Manual* to learn more).

4. Press *Accept*, and a new cube will be created and added to the *Variables*.

Upon execution, an image of the new cube is added to the *Spectrum panel* of the Spectrum Explorer and an image of the cube appears in the *Spectrum panel* (this is just an image; to explore the new cube you need to open it with the Spectrum Explorer).

You can run the task twice in a row and define a new spatial region by moving and resizing the region box in the cube image of the *Data Selection panel*. To run a second time with a new spectral region, remove the existing region first.

You can also run this task via the Tasks menu. The GUI looks the same, the only difference here is that the spectral ranges have to be entered manually.

Warning

For PACS users of PacsRebinnedCubes: the cropCube task will not work. Instead:

- to crop spectrally you can use the PACS task `pacsExtractSpectralRange` (see the PACS URM to learn how to run this task), or the extract task explained above—

which will create a `SpectrumSimpleCube` as output but thereby losing some of the datasets of the cube; these are not important unless you wish to continue working with this cube in the PACS pipeline.

- to crop spatially is more difficult; you can use the "extract" or "select" tasks to push spaxels into a `Spectrum2d`.

6.7.4. Spectrum arithmetics

Add in HCSS User's Reference Manual, subtract in HCSS User's Reference Manual, multiply in HCSS User's Reference Manual, divide in HCSS User's Reference Manual a scalar value to/from a cube or spectra to/from a cube, or perform pair-wise operations on spectra in two cubes.

The tasks are in the Spectrum Toolbox (see Section 6.7.1 for instructions on opening the Spectrum Toolbox). You can also run them directly from the *Tasks* panel. To know how to select the spectra that the tasks should run on, see Section 6.7.2, and there is also information about the parameters "segments", "selection" and "selection_lookup".

The scalar value to add, subtract etc. is entered in the *param* field if the "mode" is Scalar. If the "mode" is Pair-wise then you will input the two spectrum products (e.g. two cubes) as "ds1" and "ds2", using the standard drag-ang-drop motion (from the *Variables* to the circle next to the parameter box in the task panel).

The following operations are possible:

- **Scalar, one entire cube**: To work on an entire cube, chose the Scalar mode and enter the scalar into the *param* field, and press *Accept*. (See below for information on "overwrite".)
- **Scalar, one cube, some spaxels only**: To work on only certain spectra in the cube, see Section 6.7.2 to learn how to select the desired spectra from the cube, either via the Spectrum Explorer or via typed commands. Then enter the scalar value in *param* field, and press *Accept*.
- **Pair-wise, two entire cubes**: the two cubes should be of the same length along all three dimensions. The spaxels/pixels are combined in a pair-wise order. (See below for information on "overwrite".)
- **Pair-wise, parts of two cubes**: this cannot be done directly from the task, i.e. you cannot specify a selection to be taken from two cubes. To do this you first need to use extract or select from the cubes (as described above) to create new `Spectrum2d` products containing the (same number of) spectra to be added in a pair-wise manner.

Warning

For all operations you have to select the "overwrite" button: the tasks will not work on cubes otherwise. Because "overwrite" will overwrite the input cube, you can make a deep copy of it before running the task:

```
cube_cp=cube.copy()
```

Example 6.13. Creating a deep copy of a cube.

After pressing *Accept* the task is executed and a new cube that is effectively a copy of the input cube, ds1, but with the selected spaxel-spectra adjusted. By default it is called result (you can change the name in the panel), and is added to the *Variables* pane.

6.7.5. Spectrum averaging/summing and statistics

These tasks are in the Spectrum and/or Cube Toolbox (see Section 6.7.1 for instructions on opening the Toolboxes). You can also run them directly from the *Tasks* panel. To know how to select the spectra that the tasks should run on, see Section 6.7.2, where there is also information about the parameters "segments", "selection" and "selection_lookup".

The averaging tasks will average together spectra, rather than average all the data in a single spectrum.

- The Spectrum Toolbox averaging is nice to use if you want to average a set of contiguous or randomly-selected spaxels. If you know their coordinates, you can average either via the *Tasks* pane or the Spectrum Explorer, but if you do not know the coordinates and want to select the spectra from a cube image, you should use the Spectrum Explorer.

  If you want to average together two entire cubes, you need to use the Spectrum Toolbox.

- The Cube Toolbox is easier to use if you want to average over a shaped area.

6.7.5.1. Averaging and statistics via the Spectrum Toolbox tasks

- *avg in HCSS User's Reference Manual*: Averages together a selection of spectra to create a `Spectrum1d`.

  Quick explanation: select the spectra to average (see Section 6.7.2) and press *Accept*, whereupon the "result" will be created.

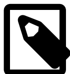
  Longer additional explanation: In Section 6.7.2 you will also find the information for parameters "Selection lookup" and "Segments". If you want to average all the spaxels/pixels in an entire cube, don't "select" but simply drag-and-drop the cube name from the *Variables* pane to the "ds" parameter.

  See the URM entry for the task to learn more about the different ways to do averaging: considering flags, and/or weights, and/or to average also the wavelength grids. The choice is set with the "variant" parameter. If you set the flag variant then flagged data can be ignored—you define the flag value to be ignored is the parameter "flagToIgnore". Datapoint weights can be considered in the averaging. If your cube has errors instead of weights, these are used instead. See Section 6.4 for information about flags, weights and errors.

  If you have NaN values in your cube then you should select the filterNaNSpectra checkbox, otherwise the NaNs will dominate the results (one NaN in a list of datapoints being averaged results in a NaN).

  The parameters "grouping" and "per group" are not useful for cubes, as there is nothing in Herschel cubes on which one can group; this is more useful for non-cube spectral data.

- *pairAvg in HCSS User's Reference Manual*: Pair-average two input spectrum containers, e.g. two cubes or a cube and a single spectrum. Pair-wise averaging means that the first spectrum in the first and second container are averaged, and so on. In case the size of the two containers is different, the result will contain a number of point spectra equal to the lowest size.



Note

The choosing of the "first" and "second" spectra to combine is based on their spectrum index, not their cube coordinates: so 0,1,2,3... rather than (0,0), (0,1) etc., although this is only important to know about if your input cubes are of different spatial sizes. To learn how to convert between the two coordinate systems, see Section 6.3.

  For cubes this task will only pair-average two entire cubes. Input the two cubes as "ds1" and "ds2" via a drag-and-drop from the *Variables* pane to the circle next to these parameters in the task's GUI, and press *Accept*. The output called "result" is created. To pair-wise average parts of cubes, you will first need to use the extract or select tasks (described above) to select the (same number of) spectra out into two new `Spectrum2d`, and then pair-wise average those.

  See the URM entry for the task to learn more about the different ways to do averaging, which consider flags, and/or weights (or errors), and/or to average also the wavelength grids. The choice is set with the "variant". See Section 6.4 for information about flags, weights and errors in general.

  Note that the input data should have the same frequency/wavelength grid, but this is not checked for. You can resample them to be the same using the *resample in HCSS User's Reference Manual* task (see Section 6.7.6).

- *accumulate in HCSS User's Reference Manual*: Accumulates (averages) spectra to a common wave-scale grid and returns a `Spectrum1d` with a single segment (i.e. it is a single spectrum). In contrast to the average task it checks whether the spectra align at the wave-scale (within a given tolerance specified by the wavescaleTolerance parameter) and automatically does a resampling if not. Furthermore, the length of the spectra need not be the same.

  For this task to work on cubes you need to set the "pointing tolerance" to a high number (so the sky offsets of the spaxels/pixels from each other is effectively ignored). To set the wavelength range(s) to include, use the range selection icon of the Spectrum Explorer (Section 6.7.2) and the wavelength ranges you chose will be entered in the "range" parameter box. Or you can type the numbers in directly yourself, particularly if running the task via its own GUI or on the command line. You can select spectra to work on using the methods explained in see Section 6.7.2. See the task's URM entry to learn more about the other parameters.

  The advantage of this task over the avg task is that the individual spectra in the input product need not have the same wavescale or length of spectra; however, as the wavescale and length of the spectra is the same for all spectra in a cube, this plays no role here.

- *statistics in HCSS User's Reference Manual*: Performs statistical operations on the datasets, calculating the mean, rms, median and percentiles (quantiles).

  Quick explanation: To select the spectra that the tasks should run on, see Section 6.7.2. After chosing the cube, and perhaps also a spaxel/pixel selection, your next choice is the mode: perChannel to work across the wavelength/frequency grid, e.g. you want to work out what the rms is for each wavelength point across an entire cube; acrossChannels to work along the spectral grid for each individual spectrum; or both. For the acrossChannels mode you can also select wavelength ranges to include or exclude: using the range selection (Section 6.7.2), upon which your chosen ranges will appear in the table in the task's GUI, or by typing the numbers directly into the table, and then choosing between "exclude" and "include" from the drop-down menu to the left of the table; this sets the task parameters "exclude" and "range", respectively (you can also set this for the perChannel mode, but it will ignore the request). Press *Accept* and the result, called stats, will be a set of `Spectrum1d` for the perChannel mode (these can then be inspected in the Spectrum Explorer) or a `TableDataset` for the acrossChannels mode.

  Longer additional explanation: In Section 6.7.2 you will also find the information for parameters "selection lookup" and "segments". With the "variant" checkboxes you can ask to include flag, weights (or error), and wavelengths in a special way in the statistical calculations: see the *URM in HCSS User's Reference Manual* entry for the task to learn more. If you set the flag variant then flagged data can be ignored—you define the flag value to be ignored as the parameter "flagToIgnore".

  The two modes of operation available from the drop-down menu:

  - perChannel: this mode operates per channel, i.e. per wavelength/frequency point. Taking the mean as an example, if you had, say, ten spectra in your dataset, this mode would calculate the mean in the first data bin (or *channel*) from all ten spectra, and then for the second bin, then the third, and so on. In other words, you are looking at all the spectra together, frequency/wavelength point by frequency/wavelength point.

  - acrossChannels: this mode operates across the wavelength/frequency range of each spectrum in the data. In other words, it works on each input spectrum separately, working on all the frequency/wavelength points in each. Taking the same example as above the result would be ten mean values. This mode produces a `TableDataset` that can be inspected in HIPE with the Dataset Viewer, allowing one to read off the values and from where you can export to a text file (see Chapter 2). When using this mode you can also use sigma clipping, by supplying the clip value and a flag value to be assigned to the clipped data. Data will be clipped when it is above `clipvalue*sigma` (standard deviation). The parameters to set are "clipvalue" and "clipflag", where the latter is a flag value you want these clipped datapoints to be given.

With this mode you can also select ranges (e.g. a spectral line) to include or exclude (chose the icon from the Spectrum Explorer button bar and draw the range(s) on the plot: see Section 6.7.3), and choose whether the range is to be included or excluded from the drop-down menu to the left of the parameter table, by clicking on *exclude/include ranges.* (You can set the ranges if you ask for "perChannel" mode also, but it will be ignored.)

Finally, the "integrated flux" box allows you to ask for the integrated flux over the spectra and region you have selected to be computed. The wavelength scale must be strictly monotonic.

6.7.5.2. Averaging and summing via the Cube Toolbox and changing the units

You can create a `SimpleSpectrum` (a single spectrum product) that is the sum or average of a spaxel region using the Cube Toolbox, using the task extractRegionSpectrum in *HCSS User's Reference Manual*. See Section 6.7.1 for instructions on opening this toolbox.

Quick explanation: chose whether you want to sum or average from the "arithmetics" menu; then chose whether you want to work on an entire cube, a single spaxel, or a shaped region ("regionType"), and if you chose for a shaped region or single spaxel, then go to the cube image in the extractRegionSpectrum tab of the *Data Selection panel* and create (and then edit) the shape/single pixel by first selecting the icon (e.g.) and then making your choice on the cube image; chose whether you want the units in [flux]/Sr (doAreaConversion clicked) or just [flux] (not clicked). Press *Accept*.

Longer additional explanation: You must first selection what type of region you want from the task tab, then select the region on the cube. The coordinates will be send to the task tab if you do it in this order.

For elliptical regions the edge spaxels have their flux scaled by the relative area of the part covered by the ellipses' contour to the total area of the spaxel.

You can also extract a single spaxel with this task: see Section 6.7.3.

From the "arithmetics" menu you can chose average or sum. From the doAreaConversion button you can chose to have the area included in the calculation—the flux is converted by the area of the region selected, and will be [unit]/Sr (per steradian) in the output spectrum. This is done as follows:

If choosing the average option, you then also have the choice of selecting the `doAreaConversion` option. The flux is converted by the area of the region selected (in this case, one spaxel), and will be in units of [flux]/Sr, converted by:

1. get the area of the spaxel in square arcsec from the WCS
2. divide the flux by this area
3. convert from e.g. Jy/sq_arcsec to e.g. Jy/Sr by multiplying with $(3600*180/\pi)^2$ (conversion factor arcsec^2/steradian).. *This conversion* assumes that the input units are [flux]/pixel *(type "print cube.getFluxUnit()" on the command line to check)—this is the case for most Herschel cubes. Your fluxes will be weirdly converted if they are already in units of flux per area* instead, e.g. Jy/beam: the "/beam" part is not converted, so your final unit ought to be e.g. Jy/beam/Sr, and will be correctly calculated for such a unit, but the unit as reported in HIPE will just be Jy/Sr.

Upon pressing *Accept* a `SimpleSpectrum` product (called "spectrum") is created and added to *Variables*. A display plot of that spectrum will also appear in a new tab in the *Spectrum panel*.

6.7.6. Spectrum manipulation: resampling, smoothing, replacing, gridding, stitching, and folding

These tasks are offered by the Spectrum Toolbox: see Section 6.7.1 for instructions on opening this toolbox. To select the spectra that the tasks should run on, see Section 6.7.2. The "overwrite" checkbox that some of these tasks have will replace the input cube with the output cube, the default is to create a new, separate output cube. For all tasks, read their *URM* entries to learn about the details of what (and how) they do.

- *resample in HCSS User's Reference Manual*: Resamples flux values of the input cube with respect to a modified wavelength/frequency grid. This task will always change all the spectra in a cube, even if you try to specify spectral selections.

  Various interpolation schemes are offered (set via the "scheme" drop-down menu). The schemes are typically a filter method in combination with an interpolation scheme and an integration scheme to assure flux conservation. The default scheme uses a box filter in combination with trapezoidal integration and linear interpolation. A second scheme consists of using a box filter in combination with Euler integration and nearest neighbour interpolation. A third scheme is based on a Gaussian filter.

  You can supply a wavelength/frequency grid that the data should be resampled to as an array of `Double1ds`, e.g.:

```
grid=Double1d([56.1, 56.2, 56.3, 56.4,...])
```

  Example 6.14. Creating a wavelength/frequency grid for resampling data.

  which you can either type directly into the "grid" parameter box, or you can create as a variable (on the *Console* command line) and drag and drop it from *Variables* to the circle next to the parameter box in the task panel. Alternatively, you can simply supply a fixed width via the "resolution" parameter, where resolution value you supply is actually twice the width that the data is resampled to: if you want to resample data to, say 0.5 km/s (so a 'width' of 0.5 km/s per bin) then you supply a resolution parameter of 1 km/s. The unit of the resolution is either set by the "unit" selection button or takes that of the input cube. Flags and weights/errors, if present, are also resampled.

  If you use a Gaussian filter, you are required to set the "kernel" (this box will appear when you set scheme to "Gaussian"). This is a smoothing filter width (the Gaussian's sigma rather than its FWHM).

  By default the *density* option is set to True (the box is checked) and this means that the flux data is treated as a flux density (per wavescale unit); if set to false the flux is treated as a per wavescale bin quantity (i.e. the integrated flux per bin).

- *smooth in HCSS User's Reference Manual*: Smooths the data *in the spectral domain* via a Box or Gaussian (of user-selected width) filter. This task will always change all the spectra in a cube, even if you try to specify spectral selections.

  The "width" parameterises the kernel functions: if the "unit" is set to "pixels", the width is rounded to the nearest integer and taken as the width of the box (for the Box smoothing) or as the standard deviation (not the FWHM) of the Gaussian kernel function (for Gaussian smoothing). Consult this task's *URM* entry to learn about the "edge" parameter, which is used to determine how the smoothing works at the edges of the spectra.

  The "variant" parameter sets whether you include weights and flags in the task. If you set the flag variant then flagged data can be ignored—you define the flag value to be ignored as the parameter flagToIgnore. Datapoint weights can be considered, giving you weighted averages for the smoothed values. If your cube has errors instead of weights, these are used instead: see Section 6.4. (See Section 6.4 for information about flags, weights (and errors) in general.)

  If you have NaN values in your cube then you should select the ignoreNaNs checkbox, otherwise the NaNs will dominate the results (one NaN in a list of datapoints gives a result of NaN).

The smoothing units you can set in the GUI are rather limited; but it is possible to smooth data on any wavescale using the command line and specifying. e.g. unit="km/s" .

- *replace in HCSS User's Reference Manual*: Replaces the data of the desired wavelength/frequency ranges in one cube with the data of the same wavelegth/frequency ranges in another cube. For example, if the first cube has a range with bad data, that for the second cube is filled with good data, you can replace the good with the bad. The input cubes must be of the same length along the two spatial dimensions, but do not have to be the same length in the spectral dimension. However, the entirety of the second cube is placed into the first cube, so the second cube should extend over the spectral range you want to replace, and no more. (You can use the extract task to extract out a spectral range: Section 6.7.3). See the *URM in HCSS User's Reference Manual* entry to learn about the mathematics of the replacing, which is set via the *mode* parameter.

- *stitch in HCSS User's Reference Manual*: Stitches together spectra or spectral segments that overlap. This is mostly used for HIFI data to stitch subbands. It is currently not useful for Herschel cubes since it will not stitch together two cubes, rather it will stick together different segments of the same cube—but Herschel cubes do not have segments. It is therefore useful only for non-cube multi-spectral products.

  The task parameters control the stitching: "variant" is to control how the stitching works (how the overlap regions are handled); "edgeTolerance" controls the task's search for overlapping points; "unit" is used to set the spectral unit that the stepSize and splitPoints are expressed in; "stepSize" is used to define a linear wavescale/frequency step the spectra are resampled to. "splitPoints" is a additional parameter that comes up if you select "splitPoints" for the "variant". Choosing this variant means you are telling the task where, in the overlapping ranges of the spectra, the points are taken to split and then join the spectra. The split points are specified as a Python list (e.g. splitPoints=[1,2,3,4]). Consult the *URM in HCSS User's Reference Manual* entry to learn more.

- *fold in HCSS User's Reference Manual*: A HIFI-specific tool that folds frequency-switched spectra (it will not work on SPIRE or PACS cubes). The frequency throw is found in the meta data and is picked up automatically by the task. If the parameter "shift" is set to True, the spectrum is shifted in frequency scale by half the throw distance so that the resulting spectrum is centred between the original and the "switched" spectrum.

See below for some command-line example of how to use these tasks with real data.

```
spireObs = getObservation(obsid = 1342227519, useHsa = True)
spireCube = spireObs.refs["browseProduct"].product.refs["HR_SSW_cube"].product
spireCube2 = spireObs.refs["browseProduct"].product.refs["HR_SLW_cube"].product
# resample
resampledCube = resample(ds = spireCube, density = True, resolution = 1.0)
# smooth
smoothedCube = smooth(ds = spireCube, filter = "box", width = 10)
# replace (requires, as per the documentation, that the two input cubes are of the
 same size)
# extract from the second cube only the part of the spectra that overlaps
# SSW: from 945 GHz to 1570 GHz
# SLW: from 450 GHz to 1015 GHz
pixnum = spireCube2.wcs.naxis3
extractedCube = extract (ds = spireCube2, minFreq = Double1d(pixnum, 950.0), maxFreq
 = Double1d(pixnum, 1000.0))
replacedSlices1 = replace(ds = spireCube, by = extractedCube, mode = "replace")
```

Example 6.15. Using resampling tasks with cubes.

6.7.7. Spectrum flagging

flagPixels in HCSS User's Reference Manual: Flags pixels in a spectrum according to a wavelength mask that the user can set. This will flag the pixels in the cube you are working on, no separate output cube is created.

Note: this task does not work on any PACS cubes and may not on SPIRE cubes either. For PACS and SPIRE, flagging during the data reduction is anyway part of their pipelines.

6.7.7.1. Pre-requisites

A pre-requisite for this task to work is that the cube should contain a flag dataset.To check for the presence of a flag dataset you can use the Product viewer on your cube (chosen via a right-click on the cube in the *Variables* pane), which will show, in its listing, all the datasets in the cube. If you run the task on a cube without the necessary flag dataset, then a standard java error message will be returned. Another prerequisite is that the task should be able to modify the data, and so you cannot open the task on a dataset that is still embedded in an `ObservationContext`.

Tip

If you want to add a flag dataset, you can try the following:

```
# set up the array (by default filled with 0)
flag = Short3d(len(cube.wave),cube.wcs.naxis2,cube.wcs.naxis1)

# Now set the values in the flag array,
# and then add the flag dataset to the cube
cube.setFlag(flag)
```

Example 6.16. Declaring a Short3d array as flags for a cube.

"Set the values in the flag array" can be done: by copying a pre-existing flag dataset, if you have one that has the flag values you exactly want; via your own scripting; or by attaching the default flag dataset and then manually editing the values via the task explained in Section 6.7.7. *Note that creating and working with flags while doing your data reduction is something that is only recommended to be done within the PACS and SPIRE Herschel data-reduction pipelines, while for HIFI this task can be used but it will be better explained in the HIFI DRG*. And as mentioned above, this task not work on PACS cubes.

6.7.7.2. Running the task

The task lies in the Spectrum Toolbox (see Section 6.7.1 for instructions on opening the Spectrum Toolbox). You can also run it directly from the *Tasks* panel.

Quick explanation: Display the spectrum/spectra you want to flag data from, and then select the datapoints to flag using the select point(s) item of the Spectrum Explorer: (see Section 6.7.2). The datapoints to flag and the spectra they belong to are selected with the same movement, so it is best to display only the spectra you want to flag datapoints of. You set the flag value/type using a parameter box/listing if using the task's own GUI: which you get depends on which instrument build of HIPE you are using (see below). If running from the Spectrum Explorer you can also select with a right-click on a selected point to select from a "Point Spectrum" menu.

Longer additional explanation: There are some differences in the way you run this task via the Spectrum Explorer or via the task's own GUI:

- **Via the Spectrum Toolbox** of the Spectrum Explorer: To define the frequency/wavelength range to flag *and at the same time* the spectra for the flagging to work on, you can use the select point(s) item of the Spectrum Explorer which can be found in its button bar (: see Section 6.7.2 to learn how to use this selection). That allows you to select either a single point (single click) or a range of points (draw a box), and at the same time all the spectra included in your selection will be included in the running of the task. Hence, it is recommend to run this task with *only* the spectra you wish to work on displayed in the plot. Several selections can be made.

  Click and/or drag with the left mouse button to select one or more spectral points. This selection then becomes the "mask" parameter. You can use the setFluxToNaN checkbox if you want the flagged datapoints also set to NaN, but this can also be set using the right-click described just next. The "flag" parameter box is provided for you to entre/chose the flag to set for these datapoints.

You can also chose flag values to set by right-clicking on the selected points to bring up a *PointSelection* menu which allows various flagging options:

- flag: you will then be asked what flag value you want these data to have.
- flag & remove: add a flag to those wavelength points and the data values are set to NaN.
- deselect.
- create variable(s): this creates a `Selection` class product that will be placed in the *Variables* pane.

As soon as you select any of the above options, the task is executed. You do not need to press the *Accept* button. If you do press the *Accept* button, the task will simply be executed again.

- **Via the task GUI**: To define the input cube ("ds") you drag and drop the cube from the *Variables* pane into the grey circle next to "ds", upon which the grey circle turns green. Replacing the input with another is done with the same action. If you want to not only set a flag for some datapoints but set the flux values to NaN, then check the setFluxToNaN box. In the "flag" parameter box you enter the value you want the flag to have or chose from the offered drop-down menu.

  To set what datapoint you want to be flagged, you need to set the parameter "mask" to the index values of those wavelengths/frequencies. This parameter is also used to define which spectra from the cube you want to have the flag set for. Defining these two is actually rather awkward: see the examples in the URM in *HCSS User's Reference Manual* entry for the task. For example, one way to define "mask" is:

```
mask={1:[1,20,667], 2:[30,690]}
```

  Example 6.17. Creating a mask using list slicing.

  for spectrum array index 1, wavelength/frequency array indices 1,20,667, and spectrum array index 2, wavelength/frequency array indices 30, 690. *Working out what these indices are is not easy, and for this reason we recommend you run the task via the Spectrum Explorer*.

For either method, if you do not specify a range but you do specify spectra to flag, then the flags will apply to all datapoints of the selected spaxels/pixels.

The values for the flags that you can set is instrument-dependent. In the "flag" box you can type in any number you like. Set your own or use a number that corresponds to flags your instrument has: consult the data reduction or instrument guides for more information.

Note

For your information (and maybe only for HIFI): flag values are calculated as 2 to the power of (n+1), where n is a bit number. The default value used is 2 to the power of 30. Flag values that are recognised by instrument pipelines or instrument-specific data reduction tools are documented in the instrument Data Reduction Guides. There is no general 'ignore' or 'bad data' flag, although a value of 1 for bad and 0 for good is a common choice, for PACS and SPIRE at least.

6.7.8. Spectrum wave unit conversion

convertWavescale in HCSS User's Reference Manual: This Spectrum Toolbox task ransforms the wavescale between frequency, velocity, wavelength and wavenumber. When converting to velocity you must supply a reference frequency and the units of the reference frequency. Note that all spectra in a cube are changed, not only any previously-selected spectra.

Running this task via the Spectrum Explorer or its own GUI is the same: since no spectral selections can be done, there is no difference in how they work. Working from the task panel directly you should

drag and drop the cube from the *Variables* pane into the grey circle next to "ds", upon which the grey circle turns green. Replacing the input with another is done with the same action. Then choose the units you want to convert to and press *Accept*.

6.7.9. Weight/error and flag propagation

Weights and errors in datasets are set by the instrument pipelines and are propagated by the spectral arithmetics tasks described above. All Herschel cubes contain a weights and flags array, SPIRE cubes may also have errors. General information on these arrays can be found in Section 6.4.

In practice, the spectral arithmetics tasks only propagate weights (w) and use the standard weight-sigma relation to propagate errors, $w = \sigma^{-2}$ to work on errors.

Until HIPE 9.0, weight propagation was carried out using a simplistic scheme. In HIPE 9.0 the propagation was done such that errors are also correctly propagated for scalar and pair-wise addition, subtraction, multiplication, division and pair-average. A weight propagation scheme that also correctly propagates errors for the remaining tasks was set in place in HIPE 10.0. The table below shows the weight propagation scheme used in HIPE 9.0 and 10.0 now, and throughout the subscript 1 refers to the first dataset passed to the task and the subscript 2 to the second. The errors are propagated with the same equations, where $w=1/e^2$.

Task	**Weight propagation scheme**	**Comment**	**Flag/mask propagation scheme**
Pair-wise add/subtract	$w = 1/w_1 + 1/w_2$	If the denominator is zero then a zero weight is returned	bitwise OR
Scalar add/subtract	unchanged		unchanged
Pair-wise multiply/divide	$w = (1/u_1 + 1/u_2) * f^{-2}$	where $u_k = w_k * f_k^{\ 2}$ and f is flux. If the resulting flux is zero then a zero weight is returned	bitwise OR
Scalar multiply/divide	$w = w / k^2$	where k is the scalar	unchanged
Pair-average	$w_{arithmetic\ mean} = 4 * (1/w_1 + 1/w_2)$, $w_{weighted} = (w_1 + w_2)$	If the denominator is zero then a zero weight is returned. The arithmetic mean is calculated with variant="flux", the weighted mean with variant = "flux-weight".	bitwise OR
Smooth	Same smoothing as chosen for the fluxes		bitwise-OR logic
Resample	same scheme as used for the fluxes		bitwise-OR logic

6.7.10. Making 2d flux maps from cubes

Flux mapping is provided by the Cube Toolbox via two tasks. In all cases the mapping is non-, or semi-interactive. Interactive fitting with the SpectrumFitterGUI is an alternative way to make such maps: for instructions see Chapter 7.

- To make direct integrated flux maps you can use *integrateSpectralMap in HCSS User's Reference Manual* which you select from the drop-down menu of the Cube Toolbox. This task adds up all the flux between 0 and the flux points of your line. Hence, if you wish to integrate the flux of a

spectral line only, you should subtract the continuum first. Removal of the continuum can be done with another Cube Toolbox task (see Section 6.7.13).

- To make flux maps using the velocity mapping task, select *computeVelocityMap in HCSS User's Reference Manual*. This velocity task also produces 2d maps of the line peak flux and the integrated line flux (everything between the line—be it absorption or emission—and 0) either by fitting the line with a Gaussian or by computing the moments. For this task to work properly, you must remove the continuum and should also crop your cube around the spectral line of interest. Removal of the continuum can be done with another Cube Toolbox task (see Section 6.7.13).

The integration approach of the Cube Toolbox is especially useful for spectral profiles that are too complex to fit with analytical functions. To find the Cube Toolbox in the Spectrum Explorer, go to the "Cube Toolbox" icon: . The toolbox will open to the right of the *Spectrum panel*.

6.7.10.1. Integrated flux maps

integrateSpectralMap in HCSS User's Reference Manual will sum up all of the flux values that are associated with each wavelength/frequency point contained within the spectral range you define. No interpolation between data-points is done. If you want a 2d map of emission/absorption line flux only, you should first remove its continuum/baseline (see Section 6.7.13), and then continue with this task on the baseline-subtracted cube.

To run the task from the Cube Toolbox you need to enter the start and end array points of your spectrum that you want to integrate over. Do this by selecting ranges (Section 6.7.2: the icon), for which you first need to have a spectrum plotted in the *Spectrum panel* (in the "plot" tab); or you can type the numbers into the box (e.g. 88.211 88.549 for startArray and 88.278 88.597 for endArray). It is good practise to remove any pre-existing ranges before defining new ones (right-click to access Range → Remove). You can also set the ranges on the command line:

```
start=Double1d([88.158,88.428])
end=Double1d([88.269,88.578])
```

Example 6.18. Creating start and end point array for integration.

and drag and drop "start" and "end" to the small circles (grey when not filled, green when filled) in the integrateSpectralMap panel and next to the "end|startOfRanges" boxes. The values will be in the units of your data (e.g. micrometres for PACS). Several spectral ranges can be input to the task at the same time.

Upon clicking *Accept*, an image will appear in a new tab in the *Spectrum panel*. Each layer of this image is a 2d map from a range you selected. Scroll through the images (if you selected more than one range) with a slide-bar at the bottom of the "images (integrateSpectralMap)" tab.

The resulting map or maps are placed into a `SimpleCube`. To extract the individual `SimpleImages` you can do one of the following:

1. Right click from inside the image in the "images (integrateSpectralMap)" tab and select the item "Extract current layer". This "extractedLayer" will then appear in the *Variables*.
2. On the command line, working on the output ("IntegratedMap" by default):

```
myimage=images.getSimpleImage(layer) # layer is a number
```

Example 6.19. Getting a layer from the images dataset of a cube.

The layer numbers begin from 0, and the images are placed in the `SimpleCube` in the order they were specified in (note that if you select the ranges via the *Spectrum panel*, then the order is always organised by wavelength value). This order can also be found by looking at the "LayerInfo" dataset of the "IntegratedMap" output (use the Product viewer on "IntegratedMap" to see this). This gives you a table of index number with the start and end of ranges information following. Or you can type:

```
print images["LayerInfo"] # see what columns are there
print images["LayerInfo"]["LayerIndex"],images["LayerInfo"]["StartOfRange"]
```

Example 6.20. Inspecting the contents of the images dataset of a cube.

The units of the data are their flux unit *times* spectral_unit (e.g. Jy·µm). See Section 6.7.16 to learn how to convert this to other units.

6.7.10.2. Gaussian line fit, and moments flux maps

You can also make flux maps—line peak and integrated line flux—using the Cube Toolbox task *computeVelocityMap in HCSS User's Reference Manual*. The velocity task uses two algorithms: fitting a Gaussian to the line, and computing the moments. Flux maps come out automatically from this task, along with the velocity maps. The way these work, and how to use the task, is explained in Section 6.7.11.

The units of the data is of the integrated flux map for this task is its flux unit *times* velocity (e.g. Jy·km/s). See Section 6.7.16 to learn how to convert this to other units.

6.7.11. Velocity maps

To make velocity maps—radial velocity and dispersion—from spectral cubes you can use a task from the Cube Toolbox. To find the Cube Toolbox in the Spectrum Explorer, go to the "Cube Toolbox" icon: . The toolbox will open to the right of the *Spectrum panel*. The task you want is called *computeVelocityMap in HCSS User's Reference Manual*. The task will also compute two flux maps: integrated line flux (everything between the line and 0 flux) and peak line flux.

Note

You can also make velocity and flux maps using the SpectrumFitterGUI, or the command-line SpectrumFitter: this is documented in worked examples of Chapter 7.

The velocity mapping task recommends:

1. that you have subtracted or divided the continuum, i.e. that it is at 0 (for emission lines) or 1 (for absorption lines),
2. there is only one spectral line of interest in your cube.

Not doing either of these will give you wrong results. To tune your cube to one spectral region of interest, you can crop it spectrally using the extract task (Section 6.7.3) of the Spectrum Toolbox or the cropCube task of the Cube Toolbox: Section 6.7.3. To learn how to remove the continuum (baseline), see Section 6.7.13, and then continue with this task on the baseline-subtracted cube.

There are two ways to compute these maps: non-interactive Gaussian fitting, or a moments method. You choose one of the two in the "velAlgorithm" parameter of the tab. You must also enter the reference wavelength (the value for zero velocity), and whether the line is emission or absorption (*isEmission* radio button). Press *Accept* to execute the task.

- *The Gaussian algorithm* will fit a single Gaussian to the brightest peak in your spectrum, using the HIPE fitter functions, for each pixel/spaxel. The names of the output products can be set in the task GUI (in "Outputs").

  Before computing the velocities the spectral grid is converted to velocity using the input reference (wavelength or frequency). The equations used in this task are given in the *URM in HCSS User's Reference Manual* entry. The task will produce the following:

  1. A velocity map: a `SimpleImage` called velocityMap by default, containing the datasets image (the velocities), chiSquared (the χ^2 of the fit), and error (the error returned by the fitter task used to fit the Gaussian).

2. A dispersion map: a `SimpleImage` called dispersionMap by default, this being the sigma (not the FWHM) of the Gaussian. This map also has an error dataset (the error returned by the fitter task used to fit the Gaussian).

3. A `SimpleImage` of the maximum flux (`maxFluxMap`) and of the integrated flux (`lineIntensityMap`), where the integrated flux is calculated from the equation for a Gaussian. These maps also have an error dataset (the error returned by the fitter task used to fit the Gaussian).

4. A vel[ocity]Cube, which is your input cube but with axes of km/s rather than the input spectral unit.

5. And a fittedLineCube, which contains the model fit to each spaxel from which the velocities and fluxes were computed. The X-axis here is also km/s, so you can compare the model to the data by opening the Spectrum Explorer on the velCube and the fittedLineCube together (see Section 6.6.5).

- *The moments method*. This method calculates the three moments (see the *URM in HCSS User's Reference Manual* entry for the equations): M0 is the integrated flux, M1 is the velocity, M2 is the velocity dispersion (similar to the sigma of the Gaussian fit). This method also returns the maximum flux in the array. The moments are calculated on a spectral region that is hunted for by the task, i.e. for each spaxel/pixel, the task itself identifies where your spectral line lies. This is explained in the *URM in HCSS User's Reference Manual* entry of the task. Before computing the velocities the spectral grid is converted to velocity using the input reference (wavelength or frequency).

  The task will produce the following:

  1. A velocity map: a `SimpleImage` called velocityMap by default, containing the datasets image (the velocities), error (the propagated error computed with the standard deviation values that are calculated by the moments method: see the *URM in HCSS User's Reference Manual* entry to know what these stddev values are), and windows (the velocity windows that the task computes [see its *URM in HCSS User's Reference Manual* entry to learn about this] and inside of which the spectral line should be located).

  2. A dispersion map: a `SimpleImage` called dispersionMap by default, this being the sigma (not the FWHM) of the Gaussian. This map also has an error dataset (the propagated error computed with the standard deviation values that are calculated by the moments method: see the *URM in HCSS User's Reference Manual* entry to know what these stddev values are).

  3. A `SimpleImage` of the maximum flux (`maxFluxMap`) and of the integrated flux (`lineIntensityMap`), where the integrated flux is the sum of the flux-data-points in the spectral line. These maps also have an error dataset (the propagated error computed with the standard deviation values that are calculated by the moments method: see the *URM in HCSS User's Reference Manual* entry to know what these stddev values are).

  4. A vel[ocity]Cube, which is your input cube but with axes of km/s rather than the input spectral unit.

  5. No fittedLineCube is produced, instead an emptyCube is produced (and this should be ignored).

Before computing the velocities the spectral grid is converted to velocity using the input reference (wavelength or frequency). Note that the radio astronomy convention is used: moving from frequency to velocity: $v = c \cdot (f_0 - f)/f_0$, moving from wavelength to velocity: $v = c \cdot (w - w_0)/w$.

The unit of the integrated flux map is the flux unit [e.g. Jy] *times* km/s, the velocity maps are in km/s, and the peak flux map is in the flux unit of your data.

The images created by this task will appear in the *Spectrum panel* as inactive displays (no selections can be done from them), each in its own tab, and the image products will appear in the *Variables* pane of HIPE, from were they can be double-clicked and displayed with an image viewer. The cubes also

appear in the *Spectrum panel* as inactive displays, and in the *Variables* pane from where you can open them with the Spectrum Explorer. Remember that you can change the spectral layer the cube image is made from in the *Data Selection panel* using its slide bar at the bottom. A second running of the same task will overwrite previously-created displays, but the products created will always be found in the *Variables* pane: each subsequent product created with the same name will have an iterator added (1, 2, 3....) to the name.

6.7.12. Position-velocity maps

The PV map task will create a 2d image: along the horizontal axis is position (offset along a line drawn on a cube) and the vertical axis is velocity, and the flux units are those of your data. You can input the width of the line (in spaxel/pixel size, e.g. 1 is +/-0.5 spaxel width). To run this task select *computePVMap in HCSS User's Reference Manual* from the drop-down menu of the Cube Toolbox. Then draw your slit (your line) on the cube using the line selection icon (see: Section 6.6.2): where you click in the cube image becomes the bottom-left end of the line, and you can resize and move the line from there (to any direction and angle). The starting and ending rows and columns appear in the corresponding parameter boxes of the panel as you move this line about. Then enter the reference wavelength/frequency (where 0 km/s is to be found) and click *Accept*.

Before computing the velocities the spectral grid is converted to velocity using the input reference (wavelength or frequency). Note that the radio astronomy convention is used: moving from frequency to velocity: $v = c \cdot (f_0 - f)/f_0$, moving from wavelength to velocity: $v = c \cdot (w - w_0)/w$. For more information on the computation, see the *URM in HCSS User's Reference Manual* entry of the task.

There are two parameters to help deal with the maps created from data with extreme aspect ratios (e.g. when you spectral range is very much longer than your spatial range). If you set to True the "correct aspect ratio" (check the radio button), then the task checks the aspect ratio of the PV map against the minimum value you give as the minAspectRatio parameter (value cannot be greater than 1). This minAspectRatio parameter is the minimum dimension ratio of the map, and is row/column (height/width: position is along the width and velocity along the height). If the aspect ratio exceeds your minimum, it calls the *scale in HCSS User's Reference Manual* task to rescale the map. It sets the values for the parameters "x" and "y" of the scale task (which are column, row) in the following way:

- x = (row/column)*minAspectRatio when row/column >1.0; and y = 1
- y = minAspectRatio / (row/column) when row/column <= 1.0; and x = 1

The task produces an image in the *Variables* pane and in the *Spectrum panel*. The value and units of the map appear on the bottom-right of the image in the *Spectrum panel* as you scroll over it.

6.7.13. Removing the continuum from cubes

In the Cube Toolbox there is a task to remove the "baseline", or continuum. This is a necessary prerequisite for creating accurate velocity and flux maps using the Cube Toolbox (as detailed in the previous section).

The output of this task is the subtracted or divided cube, and remember that if you want to continue working on it, you will need to open a new instance of the Spectrum Explorer on the new cube.

This task does an automatic fit to the continuum. To indicate the spectral regions to use in the fitting use the range selection (Section 6.7.2: the icon) on a spectrum from any spaxel/pixel of your cube. These ranges are added to the "end|startOfRanges" boxes as space-separated lists. You can also create the ranges yourself:

```
start=Double1d([88.158,88.428])
end=Double1d([88.269,88.578])
```

Example 6.21. Defining some Double1d arrays for range selection.

and drag and drop these variables from the *Variables pane* to in the Cube Toolbox task panel, next to the "end|startOfRanges" boxes (look for the the small circles, which are grey when not filled, and green when filled). You then input the degree of the polynomial (polyDegree). The parameter *divide* is to allow you to choose whether to subtract the fitted continuum or to divide by it. Then *Accept* the fitting and the whole cube is fit with the ranges and order you defined. The fit is then subtracted from or divided into the cube spectra.

"fitInfo" is a product that contains a `TableDataset` with the information of the fitting to each spaxel. If you click on "fitInfo" in *Variables* to open it with a Product viewer, and from the viewer click on the "FitInfo" in the "Data" section, you will see this information: for each spaxel row and column coordinate, the parameters of the polynomial are given, as well as the standard deviation and χ^2.

Upon pressing *Accept* two cubes are created: `baseCube` which contains the polynomial fits, and `subCube` which contains the continuum-subtracted|divided spectra. To compare one to the other, open either with the Spectrum Explorer and drag and drop the second into the *Data Selection panel*; you can then select the same spaxels from both to see the overplotted spectra in the *Spectrum panel*: Section 6.6.5.

Do remember that if using this task before running the velocity mapping tasks, you will need to open the cube in the current or a new Spectrum Explorer instance. The cube you ran `removeBaseline-FromCube` from is not overwritten.

6.7.14. Dealing with baseline issues

To learn how to use the baseline smoothing and fitting tasks, written for HIFI but usable for other instruments, read the section in the previous chapter: Section 5.5. These tasks remove regular signals, sine-waves, from the baselines. However, they do not work on a full cube, rather they work on one spectrum at a time. Hence, the instructions in the previous chapter will work for cubes also.

6.7.15. Exporting to ASCII or FITS

The task `exportSpectrumToAscii` in *HCSS User's Reference Manual* allows you to write out your cube/spectra to a text file. It is a task that can be found in the Spectrum Toolbox, and it is described in the ASCII chapter of the Data Analysis Guide: Section 2.12.

The task `simpleFitsWriter` is also offered via the Spectrum Toolbox, and this will save spectral products to FITS files. See Section 1.16 for more detail.

6.7.16. Converting units for Cube Toolbox flux maps

With the Cube Toolbox you can make flux maps via the velocity task and via the integrate flux task. The flux units you get from these are different: the velocity task returns flux unit [eg. Jy] *times* km/s and the integrated task returns flux unit [eg. Jy] *times* spectral_unit [e.g. µm]. This is because the tasks sum up under (or over) an emission (absorption) line over the velocity/spectral dimension.

To help you compare the results from the two tasks that return flux maps, and also to convert the units to the more physical W/m^2, you can use these equations:

```
# To convert from Jy·km/s to Jy·µm
# (or from K·km/s to K·GHz, or any other similar combination)
#
# Let f_jykm be the flux in e.g. Jy·km/s
# Let f_jymum be the flux in e.g. Jy·µm
# Let w_mum be the line wavelength in e.g. µm
# Let c be the speed of light in km/s
c = herschel.share.unit.Constant.SPEED_OF_LIGHT.value
c=c/1000.0
# then:
f_jymum = f_jykm * (w_mum/c)

# To convert from Jy·km/s to W/m²
```

```
#
# Let f_wm be the flux in W/m²
f_wm = f_jykm * (w_mum/c) * (2.99e-12/w_mum**2)

# To convert from Jy·µm to W/m²
#
# Let f_wm be the flux in W/m²
f_wm = f_jymum * (2.99e-12/w_mum**2)
```

Example 6.22. Converting the units of the flux maps generated by the Cube Toolbox

Note

Obviously, "Jy" is a unit of flux density rather than flux, but the word "flux" here is short-hand for whatever the signal of your spectrum is recorded in. Similarly, when we say "integrated flux" we mean that the "flux" in the area under the emission line (or over the absorption line) has been summed up.

To apply this to a map (a `SimpleImage`) edit the following to your specifications:

```
# To convert the units of a map, a SimpleImage,
# e.g. from f_jykm to f_wm
c = herschel.share.unit.Constant.SPEED_OF_LIGHT.value
c=c/1000.0
mapdata=map.getImage()
w_mum = 88.3 # the central wavelength of the line
mapdata = mapdata * (w_mum/c) * (2.99e-12/w_mum**2)
map.setImage(mapdata)
map.setUnit(herschel.share.unit.Unit.parse("W/m2"))
```

Example 6.23. Converting the units of a map represented as a SimpleImage.

6.8. Combining the PACS and SPIRE full SED for point sources

It is possible to observe the entire SED from 50 to 680 µm with two PACS observations—covering the bands B2A and B2B and their accompanying R1 ranges—and one SPIRE observation—covering the SSW and SLW bands—and in HIPE there is a script to combine the spectra of point sources that cover part of all of this range. The script can be found in the HIPE (PACS and SPIRE builds) menu: Scripts → PACS or SPIRE Useful scripts → Spectroscopy: Combine PACS and SPIRE spectra. It uses public observations as a demonstration.

Note that this task does not mathematically combine the spectra, it simply pushes them together into the same *Spectrum1d*. The advantages of this is that you can keep them together, including writing the *Spectrum1d* out to disk as FITS. A *Spectrum1d* is a class of spectrum that contains columns of flux, weight, flag, wavelength, and segment: with this segment number you can distinguish spectra that came from different sources (e.g. PACS can get segment numbers 1 to 4 and SPIRE can get segment numbers 5 and 6).

The process is quite straightforward, despite the length of the script. Most of the script is taken up with the functions that do the combining. These need to be read into HIPE before they can be used: you do by aligning the cursor arrow that is on the left-hand side of the *Editor* pane with each of the 4 occurrences of "def" (e.g. "def prepPacsSpec") and pressing the single green arrow at the top of HIPE. Further instructions for running the script are included in the script. The steps are:

1. Extract the point source spectra from your cubes; this has to be done on a PACS HIPE build for PACS data and a SPIRE HIPE build for SPIRE data, or on an all-instrument build, since these tasks are instrument-specific.

   For PACS there are two tasks to extract the point source spectrum, these are explained in the PACS Data Reduction Guide (chps 8.3 an 8.4), and the combining script includes an example of how to run the most commonly-used task of the two.

For SPIRE this is done in a task that is provided (as a jython function) in the combining script.

Tip

If you don't have an all-instrument build, you can run e.g. the PACS part in a PACS build of HIPE, write the file out as FITS, and then open a SPIRE build and read the PACS file back in to HIPE.

2. The PACS and SPIRE spectra will each be pushed into two `Spectrum1d` products, one for PACS and one for SPIRE, in which the different input spectra will have different segment numbers. If you are working with full SED coverage data then you will have 4 PACS spectra which you push into a `Spectrum1d` and 2 SPIRE spectra which you push into a `Spectrum1d` but you can work with more or fewer spectra. This can be done on any build of HIPE, as the tasks involved are instrument-independent. For PACS you are offered the chance to set flags for bad data/regions and for SPIRE you are offered the chance to change the wavelength units to micrometers.

3. Finally you will combine the PACS `Spectrum1d` with the SPIRE `Spectrum1d` product into a single, new `Spectrum1d`, with all spectra being converted to the same X-axis units ("GHz", "micrometer", "cm-1"). The flux units must be Jy (aka Jy/pixel or Jy/spaxel).

4. Note that we have added an error column to the PACS+SPIRE *Spectrum1d*. This is not a standard part of the *Spectrum1d*, but since SPIRE has errors we did not want lose them and so added that as a free-style column. For PACS this is filled with 0, but the weights column *is* filled: taking the weights from the input spectra.

You can view all the `Spectrum1d` you create with the Spectrum Explorer. The new spectra will appear in the *Variables* pane and double click on the output will show the result in the Spectrum Explorer, as shown in Figure 6.10, after adding an auxiliary upper x-axis with the wavelength in microns. You can also inspect their Meta data: the script copies over the relevant Meta data from the input products. Use the Product viewer or Dataset viewer to see Meta data. Finally, any `Spectrum1d` can be saved as a FITS file.

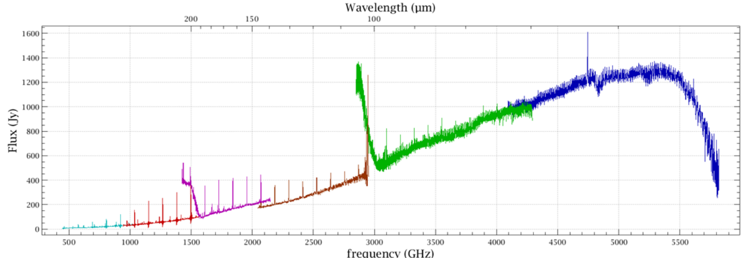


Figure 6.10. A screenshot of the combined PACS and SPIRE spectra for AFGL 618 (=CRL 618).

Chapter 7. Spectral Fitting

7.1. Spectrum fitting

In this chapter we show how to fit spectra. Fitting can be done using the *Spectrum Fitter GUI* (SFG), which provides a user-friendly interface and allows you to immediately see the results of the fitting. For command-line fitting you can use the *Spectrum Fitter* (which is the core of the SFG). Any `SpectrumContainer` can be fit, such as `Spectrum1D`, `Spectrum2D`, the HIFI and SPIRE extensions of `Spectrum1/2D`, `SpectralSimpleCubes` and the PACS extension of `SpectralSimpleCube`.

In addition to fitting features in a single spectrum, which is how you will typically start fitting, the fitter tools allow you to automatically fit models to a set of data (*multi-fitting*), which is useful when working with large sets of data and with spectral cubes. It is also possible to make fits using several models with linked parameters (*combo-fitting*), which is useful, for example, for fitting hyper-fine lines; this options can only be done with command-line fitting. You can also constrain fitting parameters to within defined limits. You can save the models you define to HIPE to be re-used in another session and there are various export options available for the fitting results. It is also possible to create your own models to use with the fitting tools.

This chapter concerns fitting *spectra*. It is possible to fit to array datasets such as `Double1d` using the various fitter functions available in HIPE and this is described in the *Scripting Guide in Scripting Guide*. The Spectrum Fitter and Spectrum Fitter GUI are built upon these fitter functions but are designed to work specifically on spectra.

We begin with a general introduction on how the GUI and the command-line approaches work, and follow that with worked examples that are tuned to what Herschel users are likely to do. The rest of this chapter contains detailed information about the usage of the SFG and the Spectrum Fitter that will allow you to fine-tune your spectrum fitting and take best advantages of the capabilities offered by the spectrum fitting package in HIPE.

Tip

Spaxels and pixels: mean the same thing, but HIFI uses "pixel" while PACS and SPIRE use "spaxel". These are the spatial-spectral unit of the cube, so one spaxel/pixel is one spatial element of your cube (one "square") with a full spectrum contained within it. If you change your spatial grid, e.g by regridding the cube, the spaxels/pixel is still the same thing, it is just that their sizes have changed.

Tip

Worked examples: scripts that take the user through the processing of fitting cubes using command-line fitting have been provided in the PACS build of HIPE ("Scripts" menu). While they work from PACS example observations and are designed for those type of data, the fitting parts of these scripts will also work, with only slight changes, on HIFI and SPIRE cubes.

7.1.1. Using the Spectrum Fitter GUI: an overview

7.1.1.1. Starting the GUI

Both the SFG and the Spectrum Fitter can be used with anything that you can display in the Spectrum Explorer. If you are unfamiliar with fitting then you should begin with the Spectrum Fitter GUI, particularly because—as shown in the worked examples in the following sections—you can export a script from the Spectrum Fitter GUI that can be used as a seed for future scripts you can write to do spectrum fitting.

The SFG is available in the *Tasks* panel (because it must be registered to work on Spectrum Containers, which makes HIPE show it as an available task) but it should only be opened from the Spectrum Explorer. To open the SFG from the Spectrum Explorer, follow these steps (and see the figure below):

1. From your spectrum/cube in the *Variables* view, right click and select the *Spectrum Explorer* (often a double click will also do this). (The use of the Spectrum Explorer is explained in Chapter 6 for spectral cubes and Chapter 5 for single spectra.) The Spectrum Explorer opens in the *Editor* view of HIPE.

2. From there, display *one* spectrum—the one you want to fit or, if working with multi-spectra datasets such as cubes, the one on which you wish to test the models to fit to the entire dataset. The way a spectrum is selected/deselected for display depends on whether you are fitting to a spectrum in a spectrum dataset or in a cube. For a spectrum dataset, click on a square in the row indicating the spectrum you want to fit in the *Data Selection panel* (see Section 5.3.1 to learn more). For a cube, click on a single spaxel/pixel from the cube image of the *Data Selection panel* of the Spectrum Explorer (see Section 6.6.2 to learn more). In both cases, the selected spectrum will appear in the *Spectrum panel*, this is illustrated in Figure 7.1 for a SPIRE cube and a HIFI WBS spectrum.

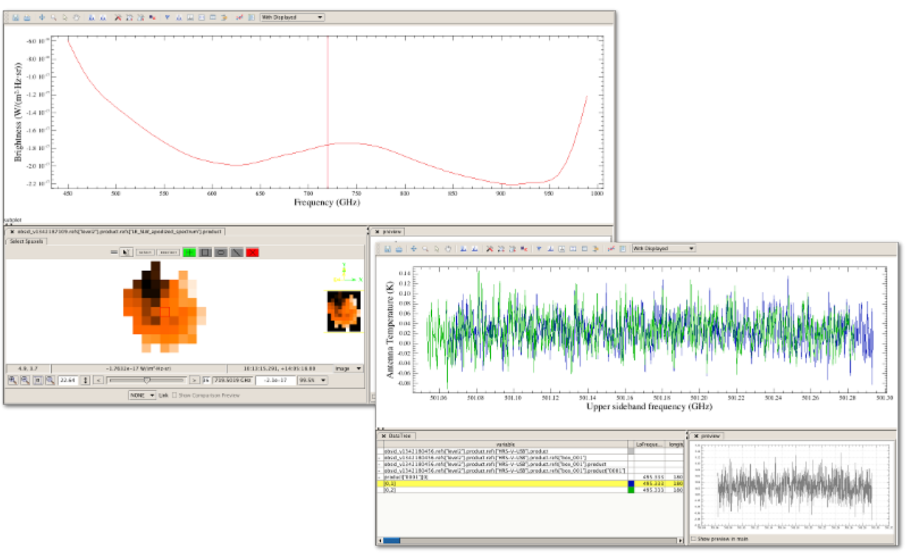


Figure 7.1. Selecting one spectrum to fit with the Spectrum Fitter GUI. Here a pixel near the centre of the cube (highlighted with a green box) is displayed in the top left Spectrum Explorer and the second subband of a HIFI WBS spectrum is displayed in the bottom right Spectrum Explorer.

3. Click on the SFG icon in the Spectrum Explorer button bar: . Alternatively, you can open the SFG from the *Dialogue* menu you get when you right-click in the *Spectrum panel*.

4. A new panel will appear to the right of the spectrum plot: this is the *SpectrumFitterGUI* panel.

If you try to open the SFG on multiple spectra—or on no spectra—a warning message will appear informing you that have not selected any spectra to fit and that you should do so. Clicking OK will generate a message in the toolbox *"Initializing failed, press Reset to retry"*. Ensure that you have selected a spectrum before reselecting the SFG.

If you do have multiple spectra plotted, you can *select* (or highlight) the spectrum of interest by clicking on the arrow icon in the Spectrum Explorer button bar ( ) and then clicking on the spectrum, which should then be plotted with a solid line and dot marker symbols. Selecting the SFG will then open it on the selected spectrum. To learn how to select cube spectra from the Spectrum Explorer, read Section 6.7.2, and for other spectra read Section 5.4.2. However, this is not a recommended way of using the Spectrum Explorer, as the plot will later become very busy with more spectra created by the SFG.

7.1.1.2. What you see when the SFG is opened

The SFG opens up in the same space as the tasks parameter form in the Spectrum Toolbox or the Cube Toolbox. There are four panels in this Spectrum Explorer, each with a different purpose.

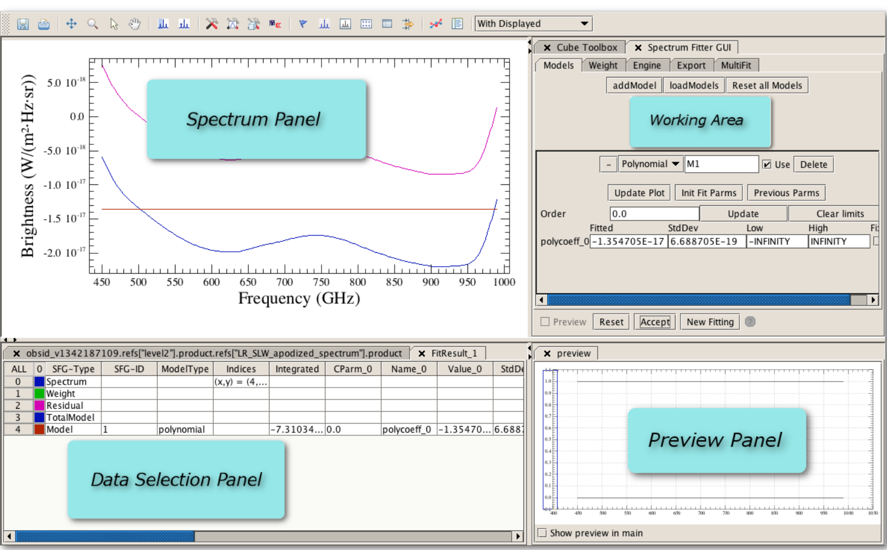


Figure 7.2. The SFG accessed via the Spectrum Explorer, and showing a Polynomial fitted to one pixel from a SPIRE cube. The labelled Spectrum Explorer and Fitter GUI panes are described in the text below.

- *Spectrum panel*: The selected spectrum and a weight line are plotted here. Models will be added as you start fitting, and the final fit and its residual will also be displayed here.
- *Data Selection panel*: In here a new tab, called `FitResult`, will appear. It contains selectable rows including the input spectrum and a weight line (click on the coloured button to show or hide them in the plot panel above), and once fitting is started it will also contain the models, the total (or global) model, and the residual after fitting. In each row representing a model the following are also shown: the names of the model parameters, their values and the standard deviation of the fit from the data as well as the integrated flux under the model.

  The data plotted in the *Spectrum panel* is colour-matched with the coloured button in the *Data Selection* under the column headed "0", you can plot or remove from the plot by clicking on these buttons. In Figure 7.2 the colours are: the spectrum you chose (dark blue), the total model (dark blue, again, because it is not plotted), the polynomial model (brown—it is the only model and thus the same as the total model), the residual (magenta) and the weights (green).
- *Working area*: The working area of the SFG is in the Toolbox panel, on the right. In different tabs you are able to: work with models; assign weights to data; select the fitter engine to be used; export the fitting results and models; and use the multifitter to apply models to multiple sets of data. These are described in the following sections.
- *Preview panel:* This panel in the lower right will show a preview of a spectrum in the Data Selection panel when clicking on its row or hovering over a spaxel of a cube. The preview panel is not significant in the use of the SFG.

7.1.1.3. Procedure to fit to a single spectrum

A typical procedure to follow when fitting to a single and relatively uncomplicated spectrum may go like this:

1. Plot *one* spectrum in the Spectrum Explorer.

2. Open the Spectrum Fitter GUI.

3. Add a model. This is done in the working area. See Figure 7.3 (you may need to resize the panel to see it all):

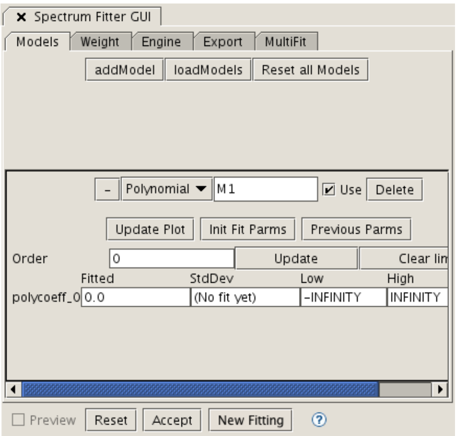


Figure 7.3. The working area of the SFG.

To add new models, you will click on *addModel*. Then select the type of model you want from the drop-down menu. An initial estimate will be applied to the data and at the same time the model drawn on the spectrum plot. You can add as many models as you wish. For simple spectra containing one spectral line and a simple continuum the result you see upon initialisation (this happens as soon as a model is selected) is usually pretty good, and when adding new models the SFG usually makes a decent guess at the parameters.

4. Adjust the model(s). You can change the model parameters in the area assigned to each model in the *Models* tab. For the polynomial model you will need to select and *Update* the order. For many of the parameters for the other models you can use the mouse to make selections in the *Spectrum Panel*, see Section 7.6 and Section 7.11 for instructions. If, upon initialisation, your chosen model could not make a first guess at the parameters, you can set the manually in this way. There are other steps you can take to optimise a model fit:

   - In the *Weights* tab behind the Models tab you can set weighted regions that are taken into account when fitting is done so that you can give more significance to a strong feature or less significance to very noisy regions. You can also mask out regions so they are not included in the fitting (e.g. mask out spectral lines so that a polynomial can fit a continuum well) by setting weights. See Section 7.8.

   - In that same *Weights* tab you can also ask that the weights in the spectrum (i.e. that are already present in the data) are included in the fitting. *However, note that you cannot use data weights and also set weighted regions*: the data weights are never taken into account if weighted regions are set.

   - You can add more models and you can also choose to not use them in a fit, or to delete them, see Section 7.14.

   - In the *Engine* tab behind the Weights tab you can change the fitter engine to use. The default usually produces a good result but see Section 7.26.

5. Press *Accept* at the bottom on the Working Area to make a global fit. This fit applies all the models you have defined to the original spectrum (the one you displayed when opening the SFG). The fitter always works on the original data (not to e.g. the residual of a previous fit), see Section 7.12.

6. Inspect the fitting results in the Spectrum Explorer. The individual models, the total model and the residual will all be found in the `FitResult` tab of the *Data Selection Panel*, you can plot/remove

them by clicking on their coloured buttons. You can also find this FitResult variable in *Variables* (this product contains the details of the fit), see Section 7.13.

7. Save (export) the fit results. Using the *Export* tab in the working area, you can save a script of your actions and save the original data, the residual and the models, see Section 7.16 and Section 7.17. (The residual can be run through the SFG again for additional fitting.)

   Upon fitting, a set of results is immediately produced and sent to the *Variables* pane: a single `context` called `SFGResultsContext` containing the: model spectrum(a), total spectrum, residual spectrum as individual products. The class of these spectra is the same as the class of the input, meaning that if you fit a cube then these outputs are also cubes, but with only the selected spaxel of each cube having a spectrum in it. The export tab is useful if you want to save to disk in the same or a different format. A product called "FitResult" is also produced: this contains the fitting results as shown in the FitResult tab you can see in the *Data Selection* panel of the Spectrum Explorer.

8. You may wish to work on the residual with other data processing tools. When you export the fit results you have an option to save the residual or the models in the same format as the original data. Taking this option will add a product called (by default) `SFGResidualDataset` to the `SFGResultsContext`. This contains your residual in some spectral format. You can click on this variable name as listed in the `SFGResultsContext` product (e.g. in the Product Viewer or Outline pane) to display it in the Spectrum Explorer and you can drag it into the *Variables* view to make a new variable to work on. From the *Variables* you can also export as FITS.

7.1.1.4. Procedure to fit a cube or multi-spectrum dataset

Warning

You should make a variable of the dataset you wish to fit rather than opening a spectrum/cube directly from the Observation Context in the Observation Viewer. If you do not work from a variable (i.e. if you do not actually extract out the spectrum or cube from the Observation Context), the SpectrumFitter will not know where to look for the rest of the spectra you fit to and will hang. No warning message is given in this circumstance.

The procedure to fit multi-spectrum datasets, such as cubes, follows on from the fitting to a single spectrum.

1. First fit a single spectrum and get the model-set that you like, as described above. Alternatively, load a model XML file from a previous fitting, see Section 7.17 and Section 7.20.

2. Then go to the *MultiFit* tab in the Working area. You can choose to save the parameters of the multi-fit (upon execution) to an ASCII file on disk from here. Fit the entire dataset with the model(s) by clicking the *Accept* button at the bottom of the panel.

   If any spaxels/pixels could not be fit, a message will appear informing you of the number of failed fits. Any spaxel/pixels that are entirely NaNs (e.g. the edge spaxels of mosaic cubes) will result in a failed fit.

3. Inspect the fitting results. In addition to any ASCII file you produced you will see, in the *Variables* View, the following new products: MultiFit_Residual, MultiFit_TotModel, MultiFit_M1[M2, M3...], which are all `SpectralSimpleCubes` with the indicated spectral results in them; M1, M2, M3, etc are the individual models you defined).

   Also created are: MultiFit_Parms, which is a product containing `TableDatasets` that hold your results, and MultiFit_ParameterCube, which contains the fitting results as a `ParameterCube` (see Section 7.28 to learn more about this).

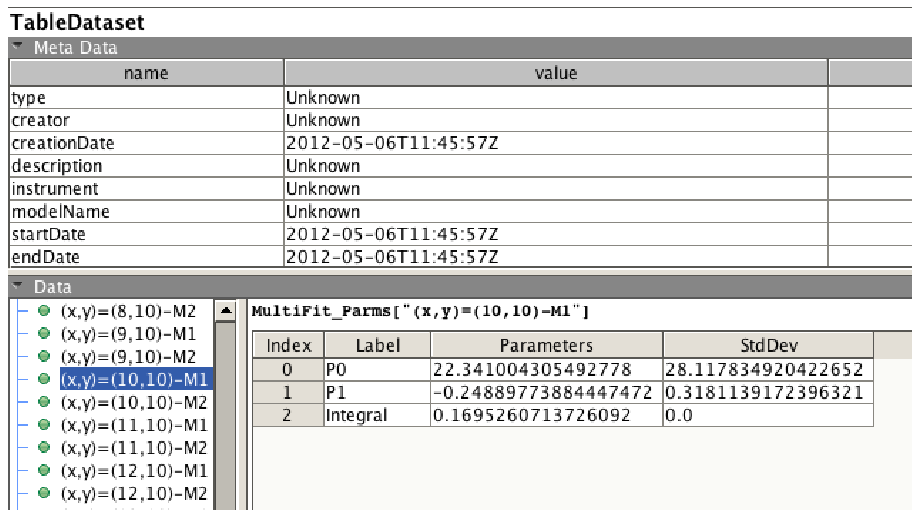


Figure 7.4. The MultiFit_Parms output.

If you want to tie models together (*combo model*) then you need to do spectrum fitting on the command line, using the Spectrum Fitter. See Section 7.23.

Bear in mind that when working with a cube, it is possible that the spectra differ substantially from spaxel/pixel to spaxel/pixel, and this could affect the accuracy of the results. The Spectrum Fitter GUI in multi-fit mode is an automatic fitter, and so cannot account for sharp and large changes to the spectra across the cube.

7.1.2. Using the Spectrum Fitter (command-line fitting): an overview

The best way to learn how to script fitting is to follow the Worked Examples in the following sections, which each contain a full script of the fitting done after a GUI-based description. This includes fitting single spectra and multi-spectral datasets, and includes creating images from the results of fitting on cubes. The remainder of this chapter also contains code snippets in each section to show how to perform the actions described in the command line, again after the GUI-based description.

Whether you use the graphical interface or a script to fit spectra is primarily down to your own preference. Scripting is, of course, very efficient and can be the best way to deal with large amounts of data. The fitting tools *require* initial estimates before making a fit, which can be difficult to do by eye, and—as for all fitting tools—if the initial estimate supplied is not a good one the fitting can be very bad or even fail. An advantage of using the GUI is that it makes the initial estimates for you based on the original data. If you prefer to script it can be helpful to make initial fits to a good-quality spectrum in your dataset using the GUI, before making a script of that fit and modifying it for your needs.

In the command line, the format in which data is passed to the SpectrumFitter depends on the type of data (spectrum or spectral cube) being used.

The `SpectrumFitter` class accepts all the main Herschel spectral types: `SpectralSegments`, `Spectrum1d`, `Spectrum2d` and spectral cubes. Whenever you start the Spectrum Fitter, a plot window is opened by default, unless you set an additional parameter to `False`:

```
sf = SpectrumFitter(mySpectrum)  # Plot window created
sf = SpectrumFitter(mySpectrum, False)  # Plot window not created
```

Example 7.1. Creating a new instance of the Spectrum Fitter with and without a plot window.

If your `SpectrumDataset` contains multiple spectra, you can specify the spectrum to fit to by giving the `pointSpectrum` and `segment` number:

```
sf = SpectrumFitter(mySpectrumContainer, spectrum, segment)
```

Example 7.2. Creating a Spectrum Fitter specifying the particular spectra by segment number(s).

Remember that `pointSpectra`, corresponding to the dataset number, are numbered starting from 0, while `segments` are numbered starting from 1.

If your input is a cube, you can instead indicate the spaxel `row` and `column` coordinate of the spectrum you want to select for fitting:

```
sf = SpectrumFitter(myCube, column, row)
```

Example 7.3. Creating a Spectrum Fitter specifying the spectra by cube coordinates.

Coordinates in cubes

In the cube image: At the bottom left of the cube display you will see indicated the (row, column) of the spaxel/pixel under the mouse. If this says, e.g (5,1), then your mouse will be 6 spaxels/pixels high along the vertical axis (the counting starts at 0) and 2 spaxels/pixels along the horizontal axis. This is the reverse of the usual convention for Astronomical images.

The Spectrum Fitter follows normal Astronomer convention and of (column, row) and it calls these (X,Y). Therefore, if you see (5, 1) as you hover your mouse over your cube image, you should use the coordinates (1, 5) in the Spectrum Fitter.

You can also select one spectrum from a cube using spectrum and spectrum segment indices with the following syntax, including setting the last parameter to `True`:

```
sf = SpectrumFitter(myCube, spectrum, segment, True, True)
```

Example 7.4. Creating a new instance of the Spectrum Fitter specifying both segment number and the display of the plot window.

For cubes, the segment value is *always* 1. The first `True` parameter refers to the creation of a plot window.

Instead of writing a script from scratch, you can use the *Save as script* option in the *Export* tab of the SFG to save a script that will recreate the last global fit performed (Section 7.1: and remember to press Accept to actually save the script). You can use this script as a template to create your own for other data. Hence, beginning with the SFG before using the Spectrum Fitter on the command line is a good idea.

Note that the script saved by the SFG starts at the point where the data to be fit is loaded into the Fitter (depending on your previous actions, the data passed to the Fitter may be a `FitResult`), you must script any earlier actions yourself; use the History tab of the Console to see the commands you ran before. Note also that the script is not completely tailored to what you just did before you saved it—some parts of the script never change.

Two useful extra things that can be done with the command-line fitting are:

- **The combo model**, with which you can tie models parameters to each other. For example, two Gaussians could be constrained to have the same FWHM as each other. See Section 7.23. **This is unique to command-line fitting**.
- **Limiting fit parameters**, where you can limit a fit to a parameter to fall within a given range. See Section 7.9.

7.1.3. Fitting tips

- When fitting a Polynomial to a baseline and also fitting a spectral line you may have more success if you use an initialisation range for the Polynomial, see Section 7.6, rather than trying to use weights. The reason for this is that weights are applied to *all* models, including the one you use for the line.

- When fitting a baseline/continuum and line profiles, you could try first fitting and subtracting or dividing the baseline/continuum, and then fitting the lines only. When fitting to a cube with the Spectrum Fitter, a residual cube is created automatically; if you wish to divide by the continuum instead of subtract you can use the Spectrum Toolbox "divide" task with the original-cube and the fit-cube as inputs (see Chapter 5 and Chapter 6 for more information).

- If a fit fails, try allowing any parameters you have fixed to float. This should help you to identify where the fit is failing. If the fitting still fails then try temporarily not using one model at at time until the data can be fit, you can then try tweaking the parameters of the problematic model until a fit is attained.

  When fitting to spectral cubes, the MultiFit_Parms output from the multi-fitter will indicate the spaxel/pixel coordinates of the fits that failed, so you can check on those (and not forgetting the swap in coordinates between the SFG and the display, mentioned in the Warning above).

- When automatically fitting to complicated or varying spectral profile in a dataset try running the multifitter several times and constraining the fitter to a specific feature on each attempt. For example, in the case of an emission line that changes to an absorption across a field, fit to the emission first by setting the limits of the amplitude to be positive. Then fit to the absorption by limiting the amplitude to be negative, see Section 7.9. (Then you may have to merge the results.)

- For information on the fitting routines upon which Spectrum Fitter is built, see the *Scripting Guide*: Section 5.8 in *Scripting Guide*. See also the fitting reference documentation.

- When chosing the reference spectrum to fit first before multifitting on a cube, try selecting one of the brightest ones rather than an average one; it can produce better overall results.

7.2. Worked Example: Fitting a polynomial to the baseline/continuum

In this simple example we make a Polynomial fit to the baseline of a PACS spectrum. The data used is the public pointed line spectroscopy observation with obsid 1342191353.

1. Get the observation:

```
obs = getObservation(1342191353, useHsa=True)
```

Example 7.5. Retrieving an observation whose data will be used for a Polynomial interpolation.

Take the level 2 `HPS3DPB: N1 R(0,0) ListContext` and create a variable from the `0: L2N1 SpectralSimpleCube` within it by dragging the cube to the *Variables* view. Or, on the command line type:

```
cube = obs.refs["level2"].product.refs["HPS3DPB"].product.refs[0].product
```

Example 7.6. Extracting a product from the observation that will be used for Polynomial interpolation.

Open the cube variable you just created in SpectrumExplorer and click on the (40, 63) spaxel to plot the spectrum.

Why make a variable? Not all aspects of the SFG, e.g. multifitting, will work properly if you work from the `ObservationContext`; it is good practice to create a variable to work on.

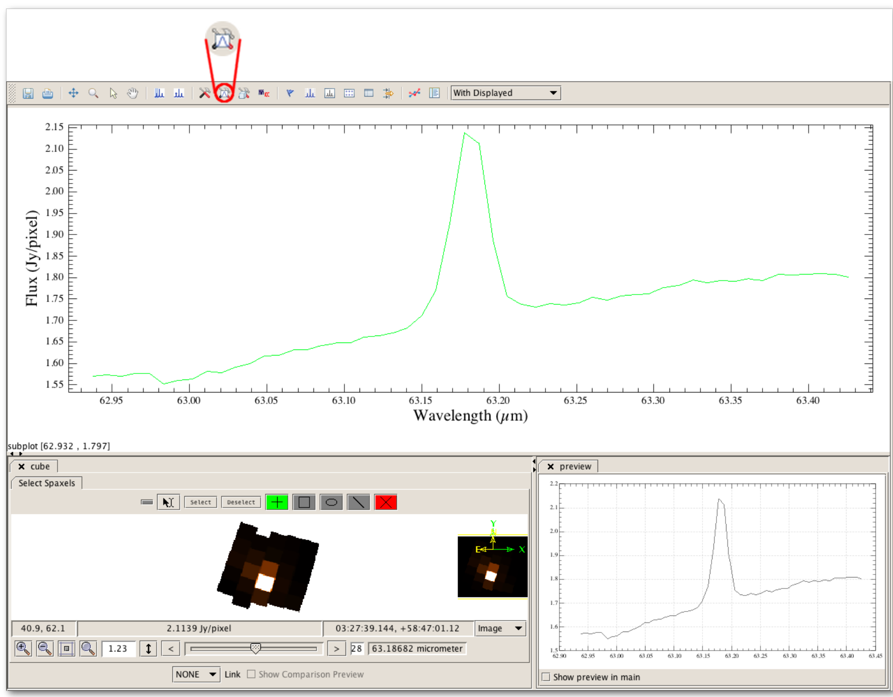


Figure 7.5. Plot one spectrum (a spaxel/pixel) and open the Spectrum Fitter GUI.

2. Open the SFG by pressing the fitting icon in the Spectrum Explorer button bar, see Figure 7.5. The SFG will open as described in Section 7.1.1.

   Note that the cross-hair that indicates the layer the cube is displayed at is still displayed and cannot be removed. If it interferes with your fitting you can move it away from the line by adjusting the layer the cube is displayed at to be the first or last layer by going back to the tab for the cube in Spectrum Explorer (called *cube* in Figure 7.15) and moving the slider below the cube, see Figure 7.15.

3. Add a Polynomial Model by pressing *addModel* and selecting a *Polynomial* model from the drop-down menu (the default choice is a Gaussian). Chose and *Update* the order of the polynomial. The model is automatically initialised and plotted against the data.

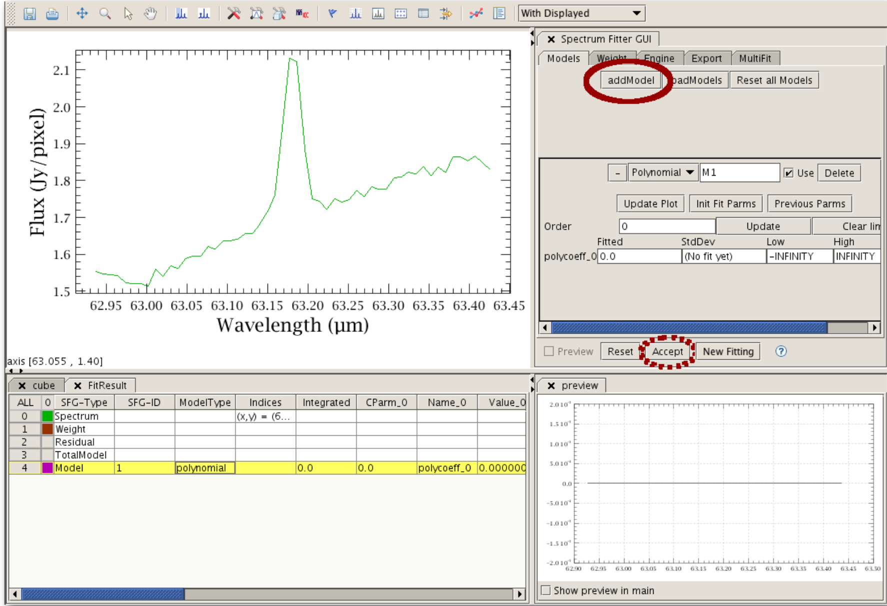


Figure 7.6. Add a Polynomial model using *addModel* and press *Accept* to fit.

4. Let's see how well the fitter will do automatically. Press *Accept*, see Figure 7.6, to fit using this initial Polynomial as input to the fitter.

   The fitted model and residual are added to the plot and we can see that the fit to the baseline is not very good near to the line - the end points of the spectrum have influenced the fit more than desired. We need to lower the weighting of the fit there.

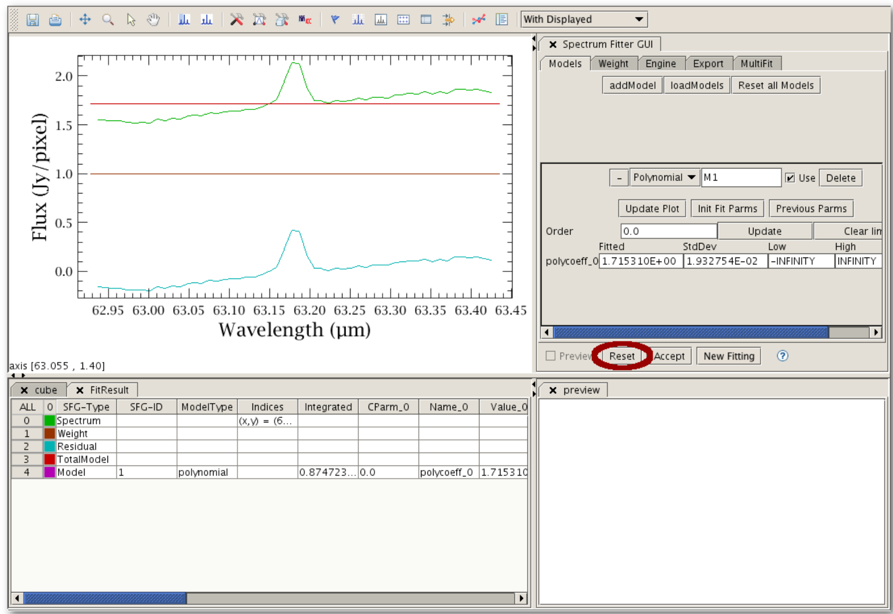


Figure 7.7. Reset the Spectrum Fitter GU to start work on the original spectrum again.*addModel* and press *Accept* to fit.

5. So, let's start over. Press *Reset*, see Figure 7.7, and add the Polynomial model again.

   A `FitResult` variable was created when you started the SFG. when you reset it a `FitResult_1` is created, this is the variable that will contain the fitting results from this restarted round of fitting.

6. Now set weights by going to the *Weight* tab. Weights are set by drawing a range on the spectrum using the *Select Ranges* mode of the Spectrum Explorer, see Figure 7.8.

   Weights are automatically set to one in the weighted region and zero elsewhere so the simplest way to lower the weighting at the edges of the spectrum is to draw a range over the central part of the spectrum that you want to fit to.

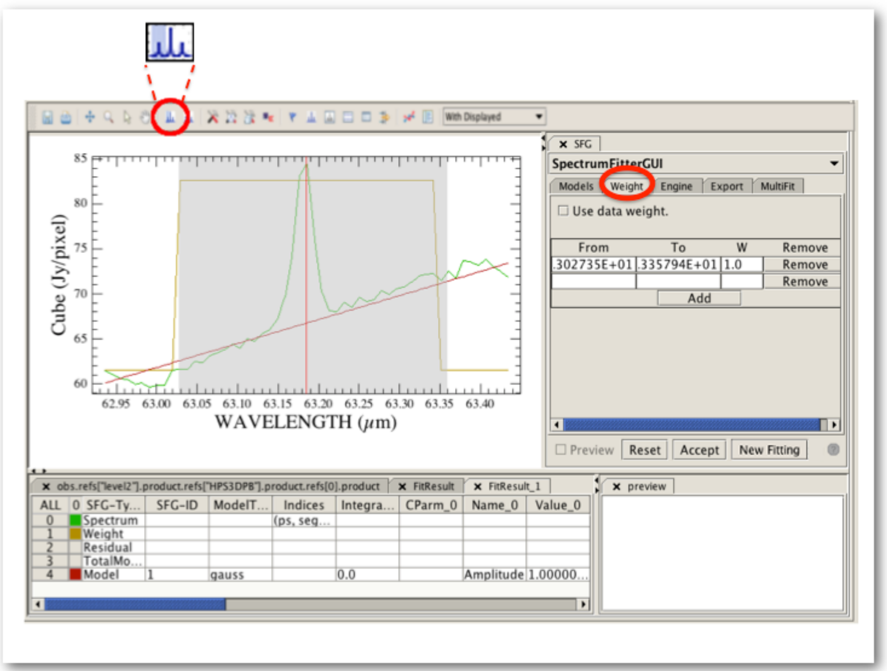


Figure 7.8. Set weights by opening the *Weights* tab and drawing a range on the spectrum.

   Note that it is not necessary to 'mask out' the line when doing this: the Polynomial will be initialised according to the first and last 10% of the data in the region where the weight is non-zero, so for this spectrum the line has no impact on the Polynomial initialisation.

7. Now go back to the *Models* tab, the weights will still be drawn on the spectrum but the ranges drawn will disappear - but you will see them again, and be able to modify them when you go back to the *Weights* tab - and re-initialise the Polynomial by pressing *Init Fit Parms*, see Figure 7.9.

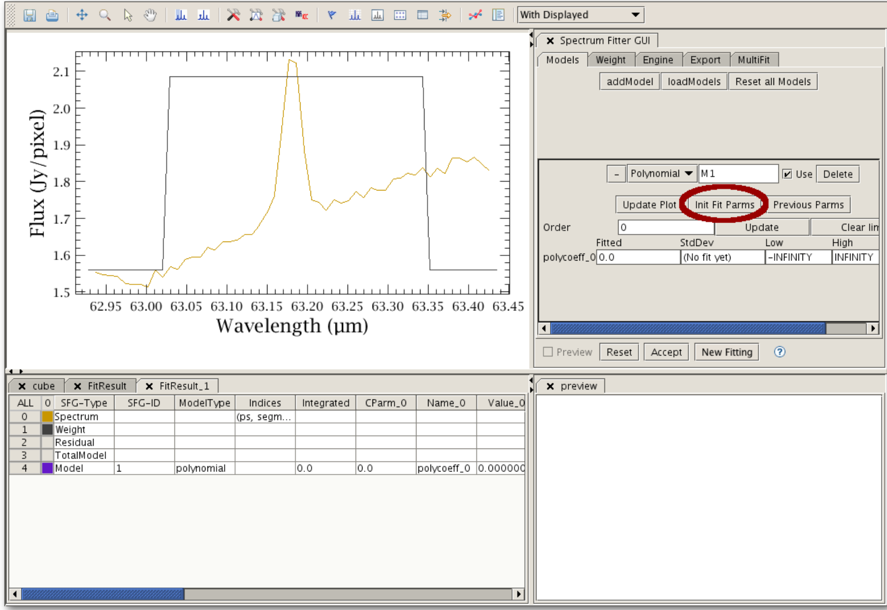


Figure 7.9. After setting weights go back to the *Models* tab and re-initialise the fit.

8. Press *Accept* again to make the fit and view the results in the plot; the model parameters can be found in the table below the plot.

   The fit is still not satisfactory. In this case, the line wings are influencing the fit and we actually do need to mask out the line. Take note that if you do mask out the line and then tried to also fit a Gaussian in the same fitting session, the Gaussian fit would fail because the weighting at the line would be zero.

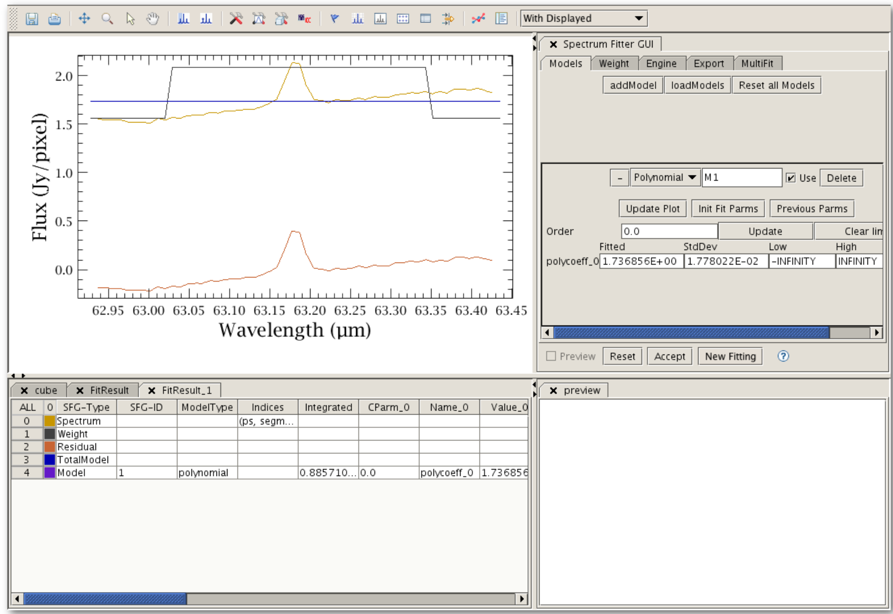


Figure 7.10. Setting the weights to zero at the line edges still did not produce a satisfactory fit.

9. Reset the fitter again and this time set weights either side of the line.

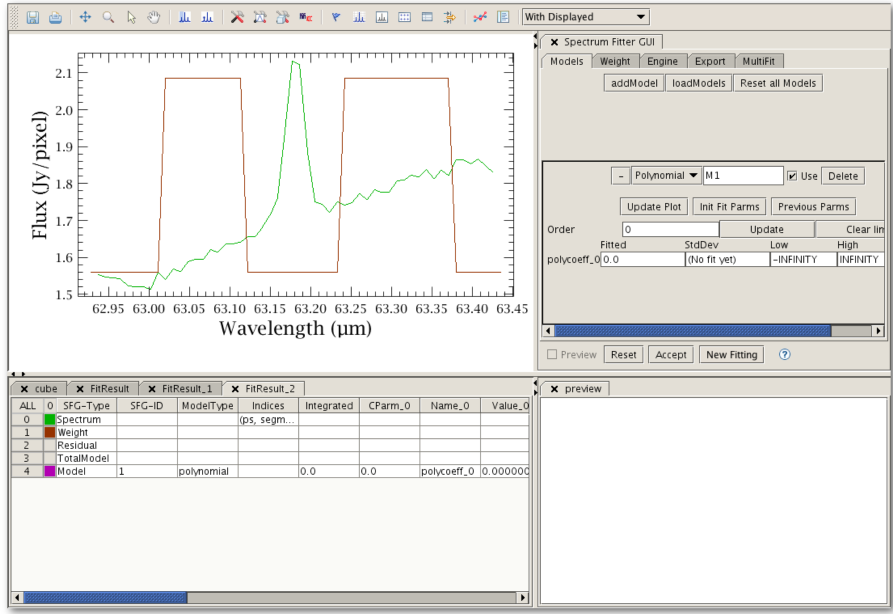


Figure 7.11. Set weights to one either side of the line in the *Weights* tab before reinitialising the Polynomial fit in the *Models* tab.

10.Press *Accept*. Finally! A decent fit to the baseline around the line. In the case of this data you could go on to attempt fitting a second order Polynomial to fit the slope at the edges of the spectrum.

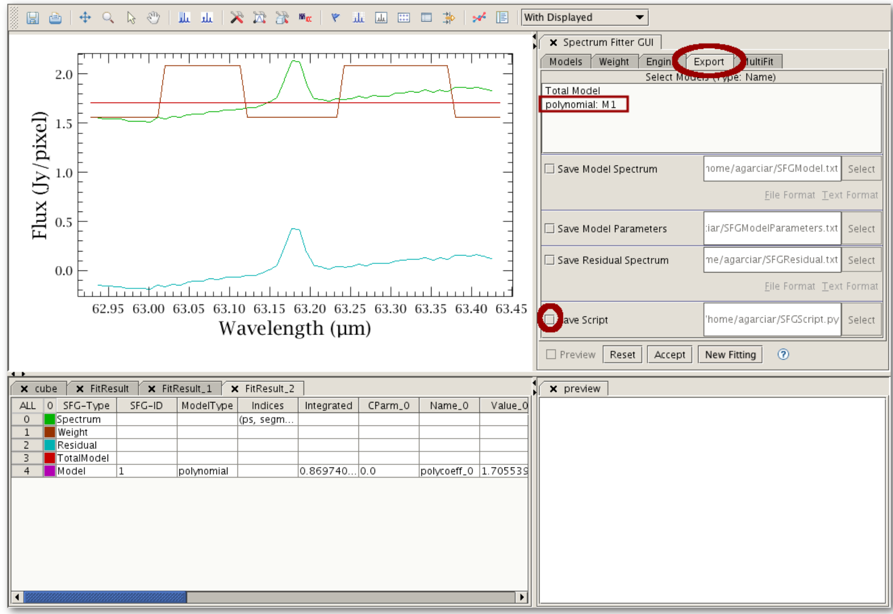


Figure 7.12. A script, the models and the residual can be saved in the *Export* tab.

11.You can quickly inspect the fit parameters in the GUI. In the *Model* tab the fit parameters are updated with the final fit values, while the box to the right contains the standard deviation. In the Data Selection Panel you find the fit parameter for each value in the model and the standard deviation in the row for each model, the total fit parameters are not given. Note that the widths reported in the Data Selection Panel are sigma, not FWHM, irrespective of how you set up the width parameter in the GUI.

12.To save a script, go to the *Export* tab and check the *Save script* box, enter the file name and location of choice and press *Accept* to save a script of the work done since the last Reset, see Figure 7.12.

From that save tab you can save other results (see Section 7.16 and Section 7.17), however the point of this exercise is just to see how the fitting works, and there is not much point saving the results of a continuum fit to one spaxel of a cube. Later examples will demonstrate the saving aspects of the SFG.

7.2.1. Worked Example: Fitting a polynomial to the baseline/continuum in the command line

The script produced by the SFG for the above example is below. Note that because the fitting was done following a resetting of the SFG, the input is taken from `FitResult_1` spectrum rather than the original cube; a line has been added to the script to allow you to extract the spaxel from the cube used in the example, and a second line has been added to show you how to print out the fitting results.

```
#
# Script written by SpectrumFitter, version: SpectrumFitter 9.51
#
# Start the fitter as SpectrumFitter(data, i, j, False).
# Your data is not a CUBE, so i, j are taken as PointSpectrum, SpectralSegment
 indices.
# For CUBE data, they are taken as x,y in the CUBE.
# The 'False' takes care the there is no visualisation. Remove it,
# or set it to True to have visualization.
#

# NOTE (not written by the Spectrum Fitter)
# After following the walkthrough above, the fitting is from the FitResult_1
 variable
# and the real instantiation of the Spectrum Fitter is the following (commented)
 line
# which would appear in the script written by the Spectrum Fitter.
# sf = SpectrumFitter(FitResult_1, 0, 0, False)

# NOTE (not written by the SpectrumFitter)
# For a self-contained example that fits the data from the cube in one go,
# use the following three lines (that were not written automatically by the Spectrum
 Fitter)
obs = getObservation(1342191353, useHsa = True)
cube = obs.refs["level2"].product.refs["HPS3DPB"].product.refs[0].product
sf = SpectrumFitter(cube, 63, 40, False)

#
# Specify fit engine (1 = LevenbergMarquardt, 2 = Amoeba, 3 = Linear,
# 4 = MP, 5 = Conjugated Gradient).
#

sf.useFitter(1)

#
# Add the models and set the 'fix'ed.
#

M1 = sf.addModel('Polynomial', [0.0], [1.7055387364757806])
M1.setLimits([Double.NEGATIVE_INFINITY],[Double.POSITIVE_INFINITY])

#
# Set the weight masks.
#

sf.setMask(63.01283, 63.12284, 1.0)
sf.setMask(63.23385, 63.38187, 1.0)

#
# Do the fit, calculate the residual.
#

sf.doGlobalFit()
sf.residual()
```

```
# see the parameter, p1, for the polynomial fitted here
print M1.getFittedParameters()

#
##
## Following are statements to export the data. Un-comment (remove
## the '# ') to execute them.
##
## To export a model, the total model, or the residual in the same
## format as the input:
##
#res = sf.getResidualAsInput( )
#tm = sf.getTotalModelAsInput()
#m1 = sf.getModelAsInput(M1) # replace M1 by any other model if needed.
##
## The models, total model and residual can be saved into an ASCII
## file. For every item that is saved, two file are written. One with
## the wave and flux data in two columns, the other with some meta
## information. The name of the files are formated as:
## [base]_[item].[ext] and [base]_[item]_info.[ext], where [item] is
## either 'res' for residual, 'tm' for total model' or the model name
## as given in the 'addModel' command.
## [base] and [ext] are:
##
#base = 'BaseName' # Change into anything you want, it may include an
## absolute of relative path.
#ext = 'txt' # Change into anything you want.
##
## You also must set the column separator character:
##
#sep = ' '
##
## Save the residual, total model and the first model into ASCII files:
# NOTE: Not written by the Spectrum Fitter
# Creating a temporary directory for writing the fitting results
from java.nio.file import Files
tempdir = Files.createTempDirectory("sf")
base = str(tempdir)+"/mymodel"
ext = "asc"
sep = ";"
sf.saveResidualAsAscii(base, ext, sep)
sf.saveTotalModelAsAscii(base, ext, sep)
sf.saveModelAsAscii(M1, base, ext, sep)
##
## The models can also be save to an XML file: [base].xml. This file
## can be loaded into a next session with the SpectrumFitterGUI, use
## the button: 'loadModels'.
## Add more Mi if you have more models.
sf.saveModelParmsToXML(str(tempdir)+'/models.xml', [M1])
# Load back the model (not possible with the ASCII version of the model)
# This currently only works with the MultiFitter
# sf.addModelXML(str(tempdir)+'/models.xml')
```

Example 7.7. Script automatically (some manual changes added for the sake of clarity) generated by Spectrum Fitter (example 1).

Warning

The order of the coordinates for the spaxels for a cube that the "SF" expects is column, row, **not** (row, column) which is the usual order that coordinates are handled in HIPE.

7.3. Worked Example: Fitting a polynomial to a spectral cube (or any multi-spectrum dataset)

This example follows on from that of fitting a Polynomial to a single spectrum: Section 7.2, using the same observation (PACS: obsid=1342191353). You will follow that same procedure as described

above, and once you are satisfied with your fit to a single spaxel, you can continue on to fit all the spaxel/pixels as below.

1. Turn to the MultiFit tab, see the figure below. Pressing *Accept* from here will then fit to the entire cube the models you defined for the single spectrum. The progress bar at the bottom of HIPE will show you the progress of the fitting (it should be fast for this simple case).

2. Before "Accepting" to fit the entire cube, you can choose to have an ASCII file sent to disk. This file will contain the results of the fits to all spaxels/pixels for all the models you defined. See the figure below for an explanation of the file's contents, which are described in the comments at the top lines of the file.

```
#
# Fit parameters for models.
# Written by: SpectrumFitter 9.51, 24JUN2013
# Date: 26SEP2013-02:15:55
#
# The output contains the following models:
# M1 = Polynomial; parameters [p0, p1, ..., integrated value]
#
# Output columns are:
# ps segm (indices of the spectrum in the SpectrumContainer.)
#      [model name] [model parameters and standard deviations in order]
#           [background value (for the non-poly models only)]
#
# Column      Name           Unit
#    C1      polycoeff_0      Jy/pixel
#    C2      stdev of C1      Jy/pixel
#    C3      Integral         Jy pixel-1 micrometer

#
#         C1         C2         C3
0 0 M1  +1.71E+00 +1.37E-02 +8.72E-01
1 0 M1  +2.00E+00 +2.57E-02 +1.02E+00
2 0 M1  -1.79E-02 +1.37E-02 -9.11E-03
3 0 M1  +1.73E+00 +1.06E-03 +8.82E-01
4 0 M1  +1.73E+00 +1.06E-03 +8.82E-01
```

   You can also ask to save a script of the fitting from here: using the Export tab to do this will not include the multifitting aspects.

Warning

The order of the coordinates for the spaxels for a cube that the "SF" expects is column, row, **not** (row, column) which is the usual order that coordinates are handled in HIPE. This also applies to the order of the spaxel coordinates in the output files that list the parameter results: the output sent to HIPE and the ASCII files sent to disk.

3. The fitter will tell you how many spectra it failed to fit - note that this will happen a lot for PACS cubes since their border spaxels are NaN arrays (because there is no spatial coverage there) for which clearly no fit can be done.

4. Several products are created and sent to *Variables* View. These are all prefaced with "MultiFit_" and all except one are cubes. These cubes are the: residual ("MultiFit_Residual"), total model ("MultiFit_TotModel"), and the model you defined ("MultiFit_M1"). You can open these cubes in the Spectrum Explorer and compare them to each other (see Section 6.6).

   The fourth output product contains the fit parameters for each spaxel/pixel and each model (but not for the total model) and is called MultiFit_Parms - double-clicking on this in *Variables* will open it in a Product viewer, where you can see it contains a set of individual `TableDatasets`, one per spaxel and model.

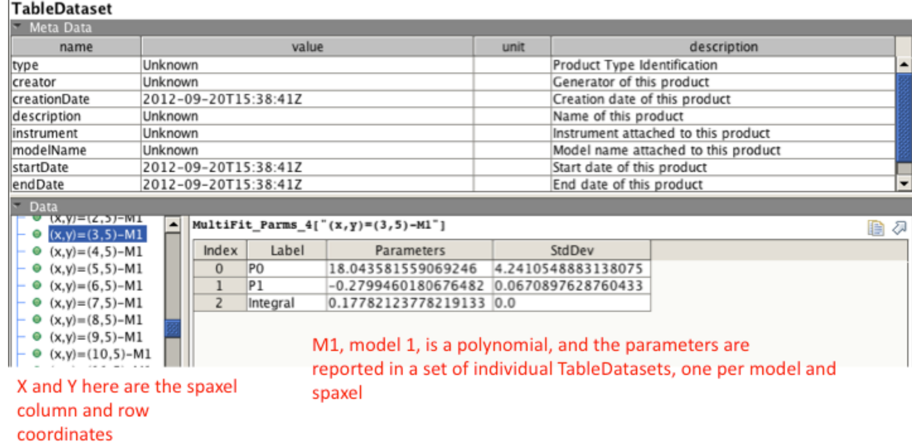


Figure 7.13. The contents of MultiFit_Parms

A final product created is MultiFit_ParameterCube, which is a `ParameterCube` holding the details of the model fit and the fitting results. In a later example (Section 7.5) we will use this to make images from fitting results.

Note, for the cube used in this example you may notice that the parameters of the fits for neighbouring spaxels are exactly the same: don't worry about this, it is a feature of this PACS cube, it is not a mistake of the fitting.

Warning

The order of the coordinates for the spaxels for a cube that the "SF" expects is column, row, **not** (row, column) which is the usual order that coordinates are handled in HIPE. This also applies to the order of the spaxel coordinates in the output files that list the parameter results: the output sent to HIPE and the ASCII files sent to disk.

7.3.1. Worked Example: Fitting a polynomial to a spectral cube (or any multi-spectrum dataset) in the command line

To do the multi-fitting described above on the command line you can follow the script given below.

First, fit a single spaxel/pixel to define the models (and check the result), we will use the same example from Section 7.2, and hence the same parameters for the model. This script below is a slightly annotated version of that produced by "Save script" from the SFG.

```
#
# Working on a SpectralSimpleCube called "cube", on a spaxel
#  of row 40, column 63, and asking to not show the fit results
#  in a PlotXY display
#  Remember that the coordinates input into the SpectrumFitter are swapped
#  around with respect to the coordinates you see in the display image of the
#  cube in the SFG (or in any image Displayer)
#
sf = SpectrumFitter(cube, 63, 40, False)
```

```
#
# Specify fit engine (1 = LevenbergMarquardt, 2 = Amoeba, 3 = Linear,
# 4 = MP, 5 = Conjugated Gradient).
#
sf.useFitter(1)

#
# Add the models and set the 'fix'ed. (Note: setting "fix" has not yet
#  been demonstrated)
M1 = sf.addModel('polynomial', [1.0], [-1990.952694249804,32.56995326749514])

#
# Set the weight masks.
#
sf.setMask(63.02211, 63.13405, 1.0)
sf.setMask(63.22926, 63.39769, 1.0)

#
# Do the fit, calculate the residual.
#
sf.doGlobalFit()
sf.residual()

# Export the model to disk, so they can then
# be read into the Multifitter
model = [M1]
sf.saveModelParmsToXML("/Users/me/firstspecfit.xml", model,True)
# (the True saves the masks also)
```

Example 7.8. Script automatically generated by Spectrum Fitter (example 1), and modified for this example.

Now extend this to the entire cube, and then extract the resulting cubes and parameters of the fits:

```
# Fit
mf = MultiFit(cube)
mf.setModels("/Users/me/firstspecfit.xml") # the models saved above
mf.doFit()

# Extract the result cubes
residual = mf.getResidual()
totalModel = mf.getTotalModel( )
Poly = mf.getModel(0)
# -->model nr 0 is the polynomial model (the only one defined)

# Extract the parameters
Params = mf.getProduct()
ParamCube = mf.getParameterCube
```

Example 7.9. Multifitting a cube using exported models.

"Params" is the table of parameter results, the same as is produced by the Multifitter of the SFG, to understand its contents see Figure 7.13. "ParamCube" is a `ParameterCube`, which holds the details of the models fit and the fitting results, done on a cube, and from which we will later make fitting maps (Section 7.5).

You can open the three cubes in the Spectrum Explorer to inspect them and compare e.g. the total model to the original data (see Section 6.6).

7.4. Worked Example: Fitting Gaussians and a polynomial to a spectrum

In this simple example we fit several Gaussians and a polynomial to a HIFI spectrum. The data used is the public DBS raster map observation with obsid 1342205481.

1. Get the observation:

```
obs = getObservation(1342205481, useHsa=True)
```

Example 7.10. Retrieving an observation from the HSA to use its data for multimodel fitting.

Navigate through the Observation Context tree to the *Level 2.5* → *cubesContext* → *cubesContext_WBS-H-USB* → *cube_WBS_H_USB_1* and "create a variable" (right-click menu) from the `cube_WBS_H_USB_1` cube, or drag-and-drop that cube to the *Variables* panel. It will get the name "obs_level_2_5_cubesContext_cubesContext_WBS_H_USB_cube_WBS_H_USB_1 (but if you drag-and-drop the cube, immediately after dropping you can type to change its name).

Open the cube variable you just created in SpectrumExplorer and plot the (0, 2) pixel.

Why make a variable? Not all aspects of the SFG, e.g. multifitting, will work properly if you work from the observation context, it is good practice to create a variable to work on.

2. Open the SFG by pressing the fitting icon in the Spectrum Explorer button bar, see Figure 7.14. The SFG will open as described in Section 7.1.1.

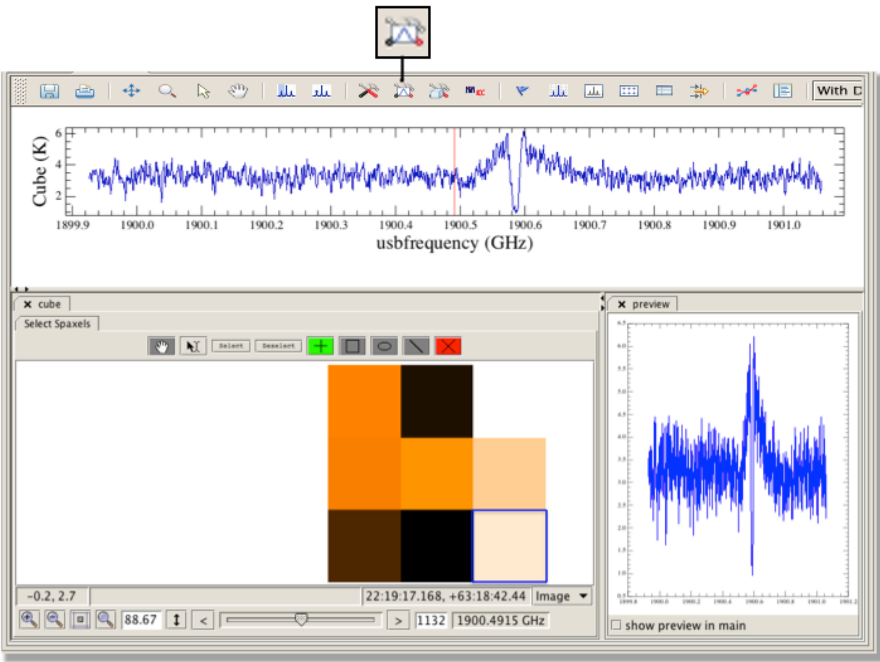


Figure 7.14. Plot one spectrum (a spaxel/pixel) and open the Spectrum Fitter GUI.

Note that the cross-hair that indicates the layer the cube is displayed at is still displayed and cannot be removed. If it interferes with your fitting you can move it away from the line by adjusting the layer the cube is displayed at to be the first or last layer by going back to the tab for the cube in Spectrum Explorer (called *cube* in Figure 7.15) and moving the slider below the cube, see Figure 7.15.

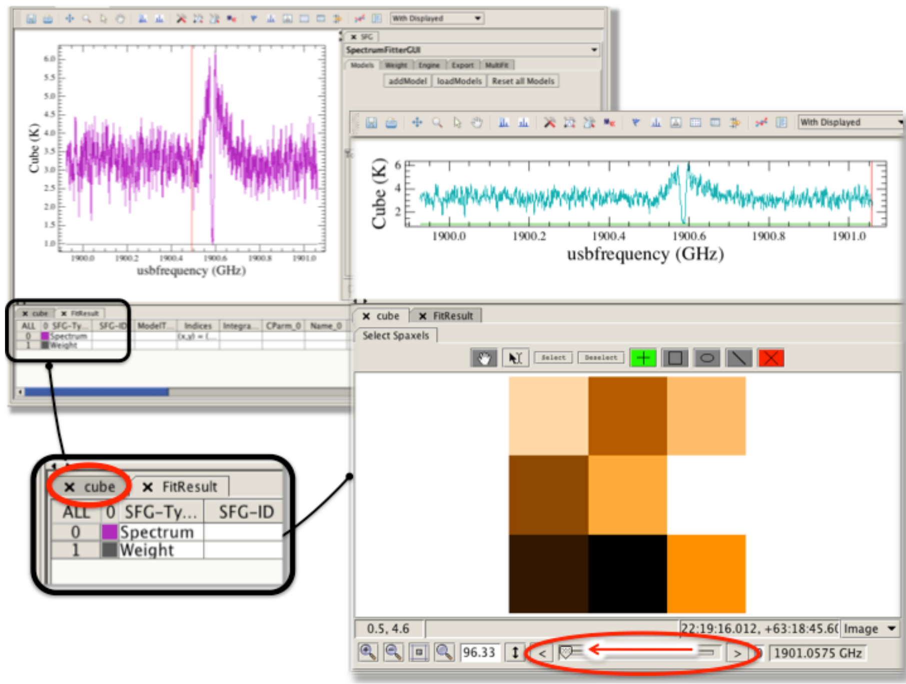


Figure 7.15. The cross-hair indicating the layer the cube is displayed at may be obtrusive, you can move it to the edge of the spectrum in the Spectrum Explorer *Data Selection Panel*.

3. Add a Polynomial Model by pressing *addModel* and selecting a *Polynomial* model from the drop-down menu (the default choice is a Gaussian), see Figure 7.16. The model is automatically initialised and plotted against the data. To learn more detail about fitting Polynomial models see the worked example in Section 7.2 above.

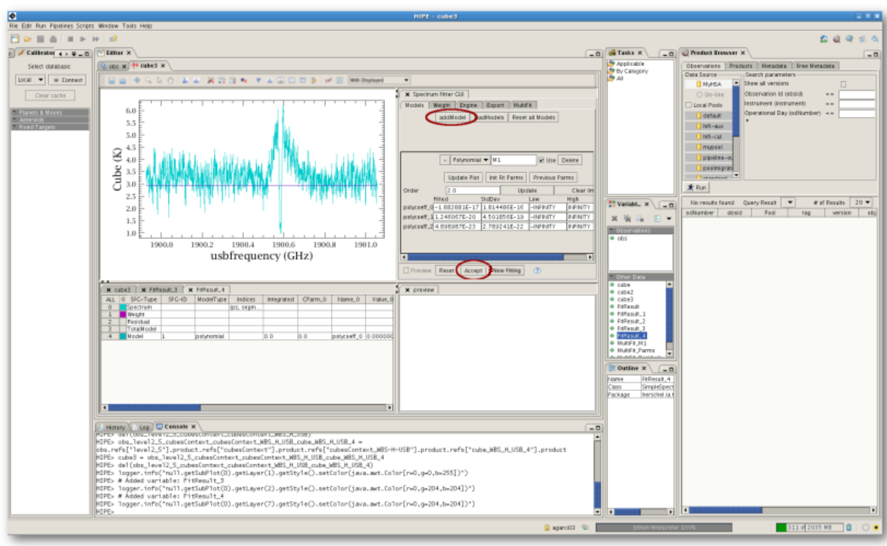


Figure 7.16. Add a Polynomial model using *addModel* in the *Models* tab.

4. Add a Gaussian by pressing *addModel* again. The Gaussian model is the default and it will initialise one model on the brightest line feature in the spectrum. For this data, more than one Gaussian model will be required so set the initial guess for the amplitude and position for this Gaussian by clicking in one of the *Amplitude* and *x-position of Peak* fields in the GUI - the two fields will turn yellow - and then choosing the position with the mouse on the plot, see Figure 7.17). The width of the Gaussian is set in the same fashion, by clicking the FWHM/sigma point to one side of the line.

You can toggle between the Gaussian sigma and FWHM using the button beside that field in the GUI; the description to the very left is the parameter in force).

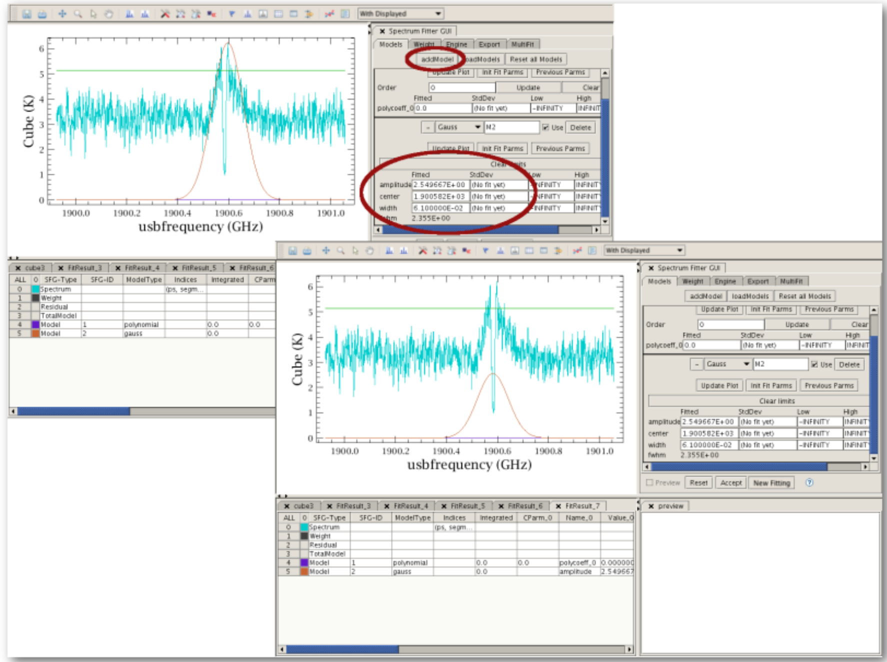


Figure 7.17. Add a Gaussian model using *addModel* and set the position and amplitude of the peak by clicking on the spectrum.

5. Now add a second Gaussian, offset from the first to account for the small 'bump' seen to the right of the main line (see Figure 7.18). The Gaussian will, as always, initialise on the brightest feature in the spectrum, so set the position, amplitude and width as required. All of the added models are described in the table in the *FitResult* tab in the Data Selection Panel, the models can be displayed and removed from the plot in the same way as any spectrum can.

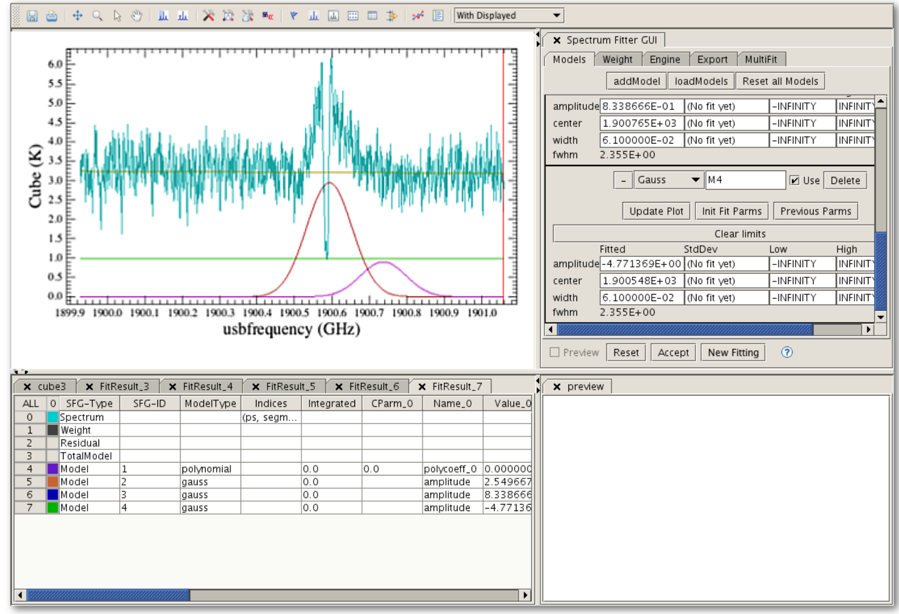


Figure 7.18. Use *addModel* to add another Gaussian.

6. Finally, add a Gaussian for the absorption. You will want to set the amplitude to have the appropriate negative value by clicking on a point below the zero level (at about -5 in this case) at the desired

line centre. To do so, zoom out by drawing around and below the line using the zoom mode of the Spectrum Explorer, see Figure 7.19.

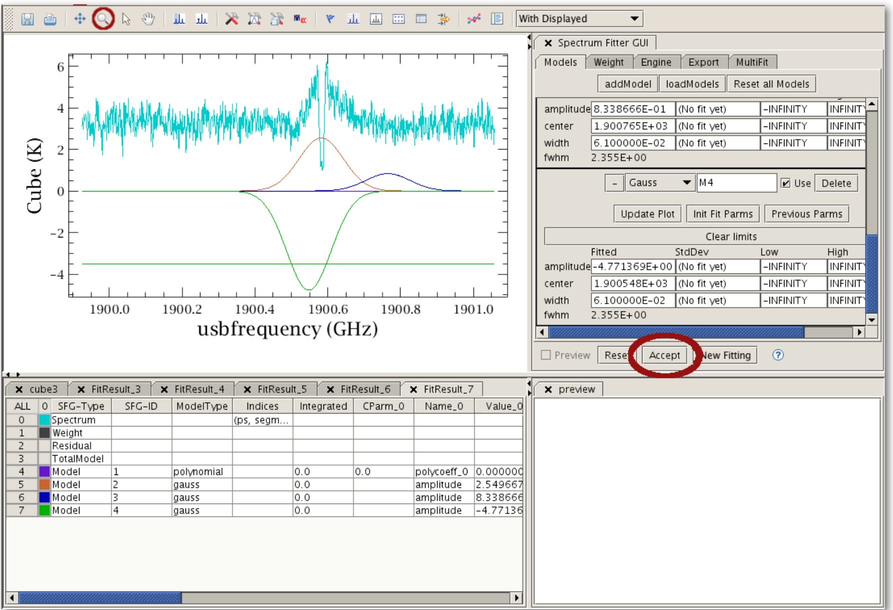


Figure 7.19. Add another Gaussian for the absorption, you will need to zoom out in order to set the amplitude of the line below zero. Press *Accept* to fit.

7. Now you should press *Accept* to fit to the line (circled in Figure 7.19). In Figure 7.20, the plot has been rescaled again using the zoom mode to show the fit and models more clearly.

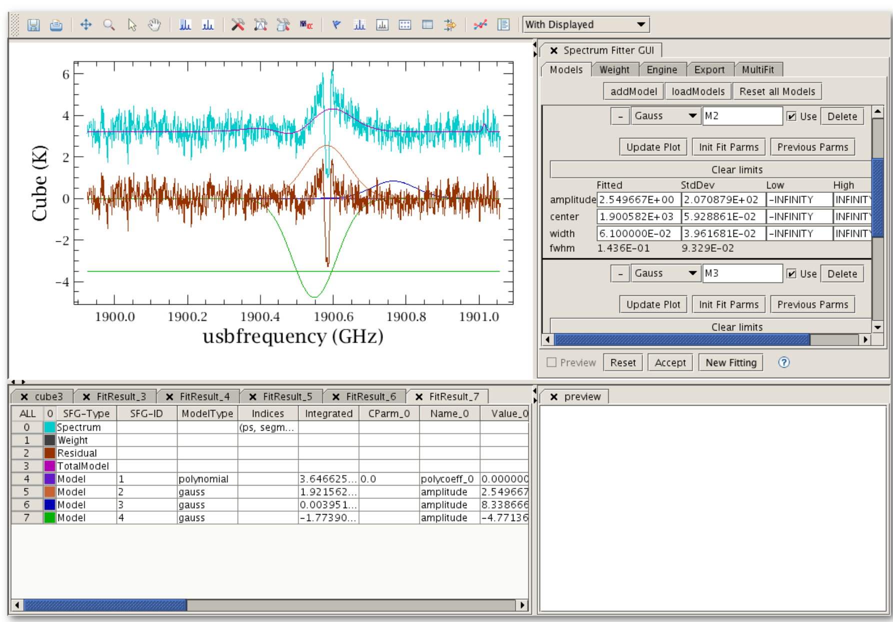


Figure 7.20. The total model and Polynomial fit are plotted over the original spectrum, while the Gaussian models appear below with the residual.

8. Upon fitting, a set of results is automatically sent to the *Variables panel*, with the name "SFGResultsContext". Inside this will be several spectral products: each model, the total model, the residual, with names such as "SFGModelM4Product". Since the product fitted in this example is a `SpectralSimpleCube`, these outputs are also, but with real spectra only in the spaxel that was fit.

A "FitResult" is also sent to the Variables panel, inside of which are the parameter values and model types that you just fit: this is a copy of the FitResult tab that is at the bottom half of the Spectrum Explorer.

9. To save a script of the fitting to disk, go to the Export tab and select *Save a Script*, enter the file name and location to save the file to, see Figure 7.21, and press Accept. You can also save model results or fitted spectra to disk from this tab, first selecting the models to save from the top of the panel; on some operating systems (particularly Windows) you may need to Shift + left click on each model individually to successfully save the parameters for all the models, on other operating systems you may be able to only Shift + left click the first and last models in the list in order to save the fit parameters. Saving to disk also saves to the HIPE, and an "SFGResultsContext" will again be sent to the *Variables panel*, containing whatever spectrum products you asked to save.

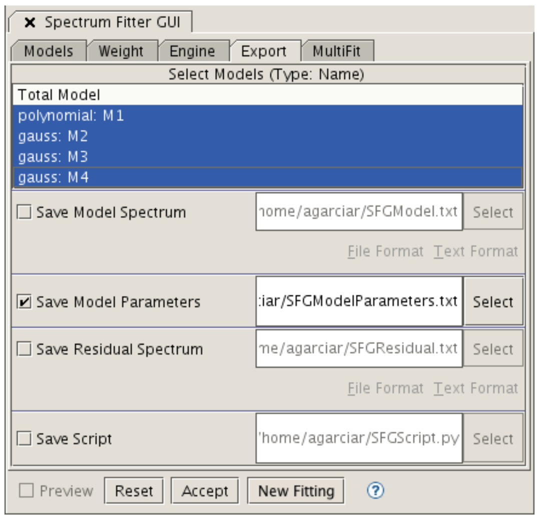


Figure 7.21. A script and the model parameters (and also the residual) can be saved from the *Export* tab.

7.4.1. Worked Example: Fitting Gaussians and a polynomial to a spectrum in the command line

The script saved by the Spectrum Fitter for all the actions described above is given below. (Note that your script may not be exactly the same, e.g. if the Gaussians were defined in a different order or the cube has a different name.)

```
#
# Script written by SpectrumFitter, version: SpectrumFitter 9.51
#
# Start the fitter as SpectrumFitter(data, i, j, False).
# Your data is not a CUBE, so i, j are taken as PointSpectrum, SpectralSegment
 indices.
# For CUBE data, they are taken as x,y in the CUBE.
# The 'False' takes care the there is no visualisation. Remove it,
# or set it to True to have visualization.
#

# NOTE (not written by the Spectrum Fitter)
# The two lines below are the access to the data used in the walkthrough documented
 in the manual
# They are not present in the script written automatically by the Spectrum Fitter
# The instantiation of the constructor for the Spectrum Fitter has been modified
 accordingly.
hifiObs = getObservation(obsid = 1342205481, useHsa = True)
hifiCube =
 hifiObs.refs["level2_5"].product.refs["cubesContext"].product.refs["cubesContext_WBS-
H-USB"].product.refs["cube_WBS_H_USB_1"].product
```

```
sf = SpectrumFitter(hifiCube, \
     2, 0, False)
#

#
# Specify fit engine (1 = LevenbergMarquardt, 2 = Amoeba, 3 = Linear,
# 4 = MP, 5 = Conjugated Gradient).
#

sf.useFitter(1)

#
# Add the models and set the 'fix'ed.
#

M1 = sf.addModel('Polynomial', [0.0], [3.221400533512354])
M2 = sf.addModel('Gauss',
 [10.867101459399871,1900.5222727424598,0.07054248387645083])
M3 = sf.addModel('Gauss',
 [0.3682626632453154,1901.0146803934288,0.004280717797218616 5])
M4 = sf.addModel('Gauss',
 [-10.682926709786162,1900.5165568434227,0.06624463178887545])
M1.setLimits([Double.NEGATIVE_INFINITY],[Double.POSITIVE_INFINITY])
M2.setLimits([Double.NEGATIVE_INFINITY,Double.NEGATIVE_INFINITY,Double.NEGATIVE_INFINITY],
[Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY])
M3.setLimits([Double.NEGATIVE_INFINITY,Double.NEGATIVE_INFINITY,Double.NEGATIVE_INFINITY],
[Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY])
M4.setLimits([Double.NEGATIVE_INFINITY,Double.NEGATIVE_INFINITY,Double.NEGATIVE_INFINITY],
[Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY,Double.POSITIVE_INFINITY])

#
# Do the fit, calculate the residual.
#

sf.doGlobalFit()
sf.residual()

# print out the parameters of e.g. the first Gaussian with
print M2.getFittedParameters()
# The parameters are listed in the same order that they are given in "sf.addModel"
 above


#
##
## Following are statements to export the data. Un-comment (remove
## the '# ') to execute them.
##
## To export a model, the total model, or the residual in the same
## format as the input:
##
#res = sf.getResidualAsInput( )
#tm = sf.getTotalModelAsInput()
#m1 = sf.getModelAsInput(M1) # replace M1 by any other model if needed.
##
## The models, total model and residual can be saved into an ASCII
## file. For every item that is saved, two file are written. One with
## the wave and flux data in two columns, the other with some meta
## information. The name of the files are formated as:
## [base]_[item].[ext] and [base]_[item]_info.[ext], where [item] is
## either 'res' for residual, 'tm' for total model' or the model name
## as given in the 'addModel' command.
## [base] and [ext] are:
##
#base = 'BaseName' # Change into anything you want, it may include an
## absolute of relative path.
#ext = 'txt' # Change into anything you want.
##
## You also must set the column separator character:
##
#sep = ' '
##
```

```
## Save the residual, total model and the first model into ASCI file:
##
#sf.saveResidualAsAscii(base, ext, sep)
#sf.saveTotalModelAsAscii(base, ext, sep)
#sf.saveModelAsAscii(M1, base, ext, sep)
##
## The models can also be save to an XML file: [base].xml. This file
## can be loaded into a next session with the SpectrumFitterGUI, use
## the button: 'loadModels'.
## Add more Mi if you have more models.
#
#sf.saveModelParmsToXML('models.xml', [M1])
#
```

Example 7.11. Script automatically generated by the Spectrum Fitter (example 3), slightly modified for this example.

7.5. Worked Example: Fitting multiple lines (Gaussians) and a Polynomial baseline to a cube and making maps of the results

7.5.1. With the GUI

For the purposes of this example we will use a SPIRE spectrum and limit the fitting to a few emission lines. If you want to instead tie models together (combo model) or limit fit parameters within certain values (set limits) then you can follow the information given in Section 7.23 and Section 7.9, inserting the relevant lines of code into the script example given here.

Fitting multiple Gaussians and a polynomial using the Spectrum Fitter GUI is not so very different to the procedure for fitting just a polynomial to a cube (Section 7.3). We will begin the tutorial with a fit to a single spaxel, and then proceed to the MultiFitting.

1. Get the observation:

   ```
   obs = getObservation(1342204919, useHsa=True)
   ```

   Example 7.12. Retrieving an observation from the HSA to use it in a multimodel fitting through the GUI.

   Take the level 2 cube, the HR_SLW_apodized_spectrum, by opening the observation with the Observation Viewer and dragging and dropping the cube to the *Variables* View, or typing on the command line:

   ```
   cube = obs.refs["level2"].product.refs["HR_SLW_apodized_spectrum"].product
   ```

   Example 7.13. Extracting a product to use the data for multimodel fitting through the GUI.

   Open the cube in SpectrumExplorer and click on the (6, 2) spaxel to plot the spectrum.

   Why make a variable? Not all aspects of the Spectrum Fitter GUI, e.g. multifitting, will work properly if you work from the observation context; it is good practice to create a variable to work on.

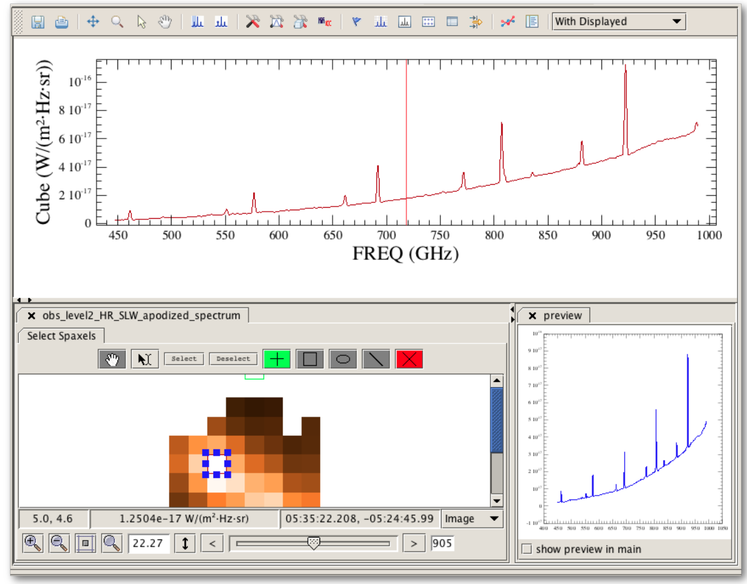


Figure 7.22. Plot one spectrum (a spaxel/pixel) and open the Spectrum Fitter GUI.

2. Open the SFG by pressing the fitting icon in the Spectrum Explorer button bar (). The SFG will open as described in Section 7.1.1.

   Note that the cross-hair in the plot indicates the layer the cube is displayed at, and cannot be removed. If it interferes with your fitting you can move it away from the line by adjusting the layer the cube is displayed at to be the first or last layer by going back to the tab for the cube in Spectrum Explorer.

3. In a change from the procedure before, this time we are going to first set the weights to mask out regions that are not wanted in the fitting. This is because we want to fit to only three emission lines over a short wavelength range in this example.

   Set weights by going to the *Weight* tab. Weights are set by drawing a range on the spectrum using the *Select Ranges* mode of the Spectrum Explorer. Weights are automatically set to one in the weighted region and zero elsewhere, so by drawing a range over the three emission lines plus some continuum will make the fitter only work within that range. In this example we will set the range (value 1) from about 754 to 860 GHz. It will also help if you now zoom in on this range (click the zoom icon at the top of the Spectrum explorer and then draw your zoom on the plot: if you want to see the residual the fitter will sent to this plot, ensure your range goes below 0).

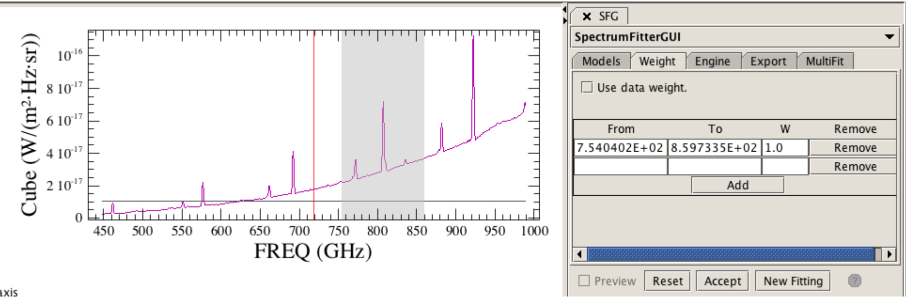


Figure 7.23. Set weights by opening the *Weights* tab and drawing a range on the spectrum.

Note that the Polynomial will be initialised according to the first and last 10% of the data in the region where the weight is non-zero, so make sure these areas are continuum.

4. Add a Polynomial Model by pressing *addModel* and selecting a *Polynomial* model from the drop-down menu (the default choice is a Gaussian). The model is automatically initialised, but you first need to set the order and press *Update*. The initial fit and plotted against the data, and the fit results are sent to a FitResults tab in the Data Selection panel in the lower part of the Spectrum Explorer. Adjusting the polynomial order can be done by typing it in a box and "updating", and after changing you can press *Accept* to update the fit and plot the fitting results (model and residual). Naturally, the polynomial will not fit well if your data contain several emission lines, but at present this does not matter.

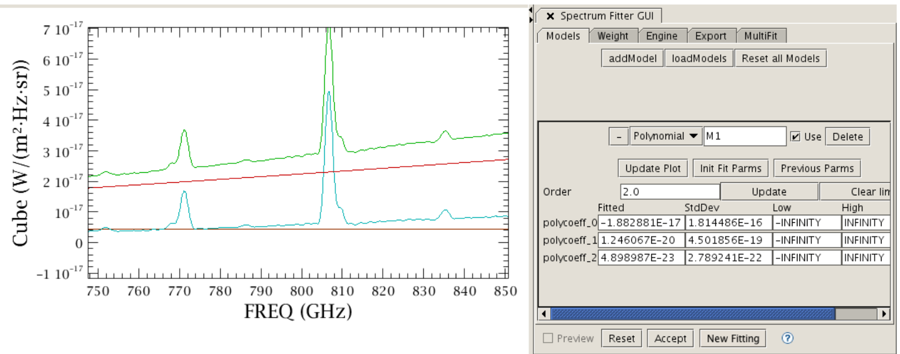


Figure 7.24. Add a Polynomial model using *addModel* and press *Accept* to fit.

5. In this example we will further define 3 Gaussians, at about 771, 807 and 835 GHz. You will see from the spectrum that in fact some of these lines have a second feature on one side, but for this example we will ignore these. (You could fit them yourself by defining additional Gaussians.)

   Add a Gaussian by pressing *addModel* again. The Gaussian model is the default and it will initialise one model on the brightest line feature in the spectrum. Alternatively you can set the initial guess for the amplitude and position for this Gaussian by clicking in one of the *Amplitude*, *X-Position of peak* fields in the GUI - the two fields will turn yellow. The x+y-position is chosen with a subsequent mouse-click on the plot at the correct x+y-axis value, and this will work whether you are asking to fit an emission or an absorption feature.

   If you want to define the width also (not necessary for this example), click on the *Sigma* field and then click to one side of the line near its sigma point (sigma $\approx$ FWHM/2.3). You can toggle between the Gaussian sigma and FWHM using the button beside that field in the GUI; the left-most description beside the box tells you which parameter is in force.

   The Spectrum Fitter will make a Gaussian fit, and again don't worry if the fit is bad (although the x-axis position should be correct!).

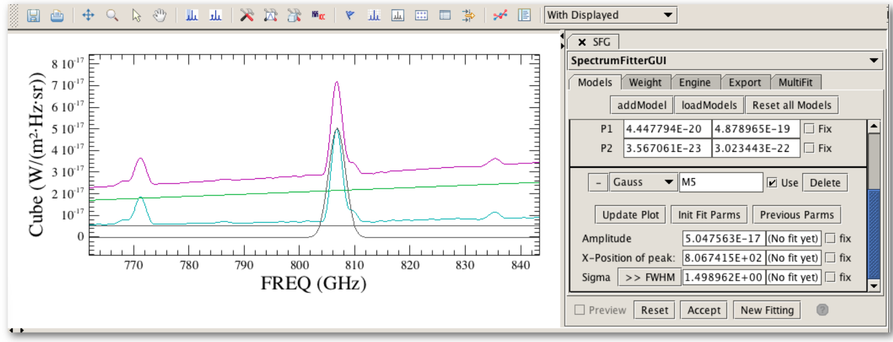


Figure 7.25. Add a Gaussian model using *addModel* and set the position and amplitude of the peak by clicking on the parameter boxes ("Amplitude" or "X-Position") to the right and then on the spectrum.

6. Now add a second (and after that a third) Gaussian, which will still initialise on the brightest feature in the spectrum (it always does), so set position and amplitude as before. Press *Accept*, to fit these

3 Gaussians and the polynomial together to the spectrum. The fitted model and residual are added to the plot, and all of the added models are described in the table in the FitResult tab in the *Data Selection Panel*. The models can be displayed and removed from the plot in the same way as any spectrum can (see Section 5.3.1).

7. If you are satisifed with this, you can now go to the MultiFit tab and apply those models to the entire cube. Pressing *Accept* from here will then fit to the entire cube the models you defined for the single spectrum.

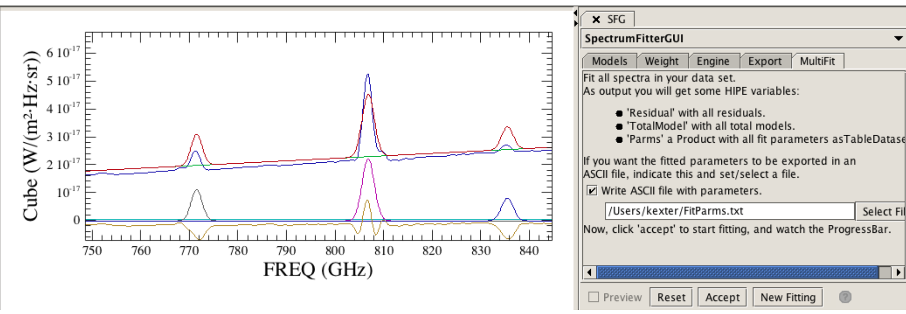


Figure 7.26. To fit an entire cube, use the MultiFit tab

8. Before "Accepting" to fit then entire cube, you can ask to have an ASCII file sent to disk. This file will contain the results of the fits to all spaxels/pixels for all the models you defined. See the output below with an explanation of the file's contents. The explanations are inside curly brackets ({ }).

```
#
# Fit parameters for models
# Written by: SpectrumFitter 9.34, 27MAR2012
# Date: 20SEP2013-04:02:25
#
# The output contains the following models:
# M1 = polynomial; parameters [x-axis crossover, slope, integrated volue]
# M2 = gauss; parameters [amplitude, centre, fwhm, integrated value]
# M3 = gauss; parameters [amplitude, centre, fwhm, integrated value]
# M4 = gauss; parameters [amplitude, centre, fwhm, integrated value]
#
# Output columns are:
# x y (indices of the spectrum in the Cube.)
#      [model nameJ [model parameters and standard deviations in orderJ
#          [background value (for the non-poly models only)]
# {two first columns are spaxel col, row}
0 0 Fit Failed.
1 0 Fit Failed.  {M1: model is a polynomial: parameters are p0,stddev(p0),p1,
 stddev(p1) and the integration under the line}
2 0 M1  -3.50E-17 +4.39E-18 +5.52E-20 +5.48E-21 +2.53E-15
2 0 M2  +1.65E-17 +8.86E-19 +8.07E+02 +6.44E-02 +2.60E+00 +1.52E-01 +4.57E-17
 9.56E-18
2 0 M3  +3.95E-18 +8.80E-19 +7.71E+02 +1.83E-01 +2.89E+00 +4.30E-01 +1.21E-17
 7.60E-18
2 0 M4  +2.11E-18 +9.63E-19 +8.35E+02 +2.85E-01 +2.08E+00 +6.70E-01 +4.67E-18
 1.11E-17
3 0 M1  -3.52E-17 +4.14E-18 +5.63E-20 +5.17E-21 +2.83E-15
3 0 M2  +2.06E-17 +8.46E-19 +8.07E+02 +5.25E-02 +2.53E+00 +1.24E-01 +5.53E-17
 1.02E-17
3 0 M3  +4.42E-18 +8.33E-19 +7.71E+02 +1.64E-01 +2.82E+00 +3.86E-01 +1.33E-17
 8.22E-18
3 0 M4  +2.40E-18 +9.04E-19 +8.35E+02 +2.56E-01 +2.15E+00 +6.04E-01 +5.50E-18
 1.18E-17
4 0 Fit Failed.
5 0 Fit Failed.   {M2|3|4: models 2,3,4 are Gaussian, and the parameters are amp,
 stddev(amp), central wave,
6 0 Fit Failed.   stddev(central wave), fwhm, stddev(fwhm) and the integrated
 (line) flux [with no error]}
7 0 Fit Failed.
8 0 Fit Failed.
0 1 Fit Failed. {Failed fits: no parameters are reported}
```

You can also ask to have a script that contains the fitting commands—from when you selected the "reference" spaxel to fit until the multifitting is done—to disk.

Warning

The value of the **line width** for the Gaussian profiles reported in this ASCII file is the FWHM (full-width-half-maximum, which is sigma*2.3548). In contrast, the value of the line width that is in the MultiFit_Parms product that the MultiFitter also creates and which you will find in the HIPE *Variables* panel is the sigma of the line.

The same is true for the output of the SpectrumFitter, i.e. when you fit a single line: any ASCII output to disk reports the FWHM, any output to HIPE reports the sigma, if your line model is a Gaussian.

Warning

The order of the coordinates for the spaxels for a cube that the "SF" expects is column, row, **not** (row, column) which is the usual order that coordinates are handled in HIPE. This also applies to the order of the spaxel coordinates in the output files that list the parameter results: the output sent to HIPE and the ASCII files sent to disk.

9. The fitter will tell you how many spectra it failed to fit (in this example it should be about 37). Fits fail in the case of spaxel/pixels containing NaNs, for example, in the location of missing bolometer detectors, or wherever the defined model is a very poor fit to the data.

10. Several products are created and sent to *Variables*. These are all prefaced with "MultiFit_" and all except one are cubes. These cubes are the: residual ("MultiFit_Residual"), total model ("MultiFit_TotModel"), and each of the individual models you defined ("MultiFit_M1" etc). You can open these cubes in the Spectrum Explorer and compare them to each other (see Section 6.6).

    A fourth output product contains the parameters of the fits and is called MultiFit_Parms - double-clicking on this in *Variables* will open it in a Product viewer, where you can see it contains a set of individual `TableDatasets`, holding the individual model parameters for each spaxel/pixel (recall the warning about the reported width values):

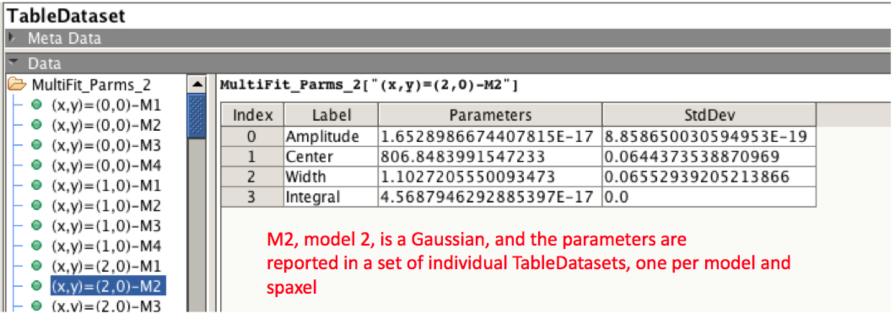


Figure 7.27. The contents of MultiFit_Parms

Warning

The order of the coordinates for the spaxels for a cube that the "SF" expects is column, row, **not** (row, column) which is the usual order that coordinates are handled in HIPE. This also applies to the order of the spaxel coordinates in the output files that list the parameter results: the output sent to HIPE and the ASCII files sent to disk.

A final output is called MultiFit_ParameterCube, and this is what will be used to make images of the fitting results. This `ParameterCube` is not a spectral cube, but rather it contains the results of fitting done on a cube, and the details of the models, in such a way that one can recreate the model-fit cubes from it. This will be used to make images of the fitting results later.

To do the multi-fitting on the command line, follow the script given next.

7.5.2. On the command line

First, fit a single spaxel/pixel to define the models (and check the result). This script has been modifed slightly from that produced by "Save a script", which produced the script that will repeat the fit you did on the single spaxel in the explanation above.

```
# Get the data
obs = getObservation(1342204919, useHsa=True)
cube = obs.refs["level2"].product.refs["HR_SLW_cube_apod"].product

# (1) fit a single spaxel/pixel

sf = SpectrumFitter(cube, 2, 6, False)
# Remember that the coordinates input into the SpectrumFitter are swapped
# around with respect to the coordinates you see in the display image of the
# cube in the SFG (or in any image Displayer)

#
# Specify fit engine (1 = LevenbergMarquardt, 2 = Amoeba, 3 = Linear,
# 4 = MP, 5 = Conjugated Gradient).
#

sf.useFitter(1)

#
# Add the models and set the 'fix'ed. (Setting "fix" has not yet
#  been demonstrated)

M1 = sf.addModel('polynomial', [2.0], \
[-4.348775725735947E-17,4.6906344783025225E-20,5.092175923218101E-23])
M2 = sf.addModel('gauss',
 [4.262995605854649E-17,806.7468159200247,1.1695420595470503])
M3 = sf.addModel('gauss',
 [1.1423712924644597E-17,771.0924013923408,1.1168446099436533])
M4 = sf.addModel('gauss',
 [5.0414019995950075E-18,835.6878789721305,1.3976400964618703])


#
# Set the weight masks.
#

sf.setMask(754.0402, 859.7335, 1.0)

#
# Do the fit, calculate the residual.
#

sf.doGlobalFit()
sf.residual()

# Export the model to disk, so they can then
# be read into the Multifitter
model = [M1,M2,M3,M4]
sf.saveModelParmsToXML("/tmp/firstspecfit.xml", model,True)
# (the True saves the masks also)

# Get the SpectralLineLists
# Lists for all the models
sll = sf.getLineList()
# List for specific models
models = [M1, M2]
lls = sf.getLineList(models)
```

Example 7.14. Example script fitting a spectrum with multiple models as exported by HIPE.

Now extend this to the entire cube, and then extract the resulting cubes and parameters of the fits:

```
# Fit
mf = MultiFit(cube)
mf.setModels("/tmp/firstspecfit.xml")
mf.doFit()

# Extract the result cubes
residual = mf.getResidual()
totalModel = mf.getTotalModel( )
Poly = mf.getModel(0)
Gauss1 = mf.getModel(1)
Gauss2 = mf.getModel(2)
Gauss3 = mf.getModel(3)

# Extract some parameters
Params = mf.getProduct()
ParamCube = mf.getParameterCube
```

Example 7.15. Multifitting a cube with a set of exported models.

"Params" (a table of fitting results) and "ParamCube" (a `ParameterCube` of fitting results+model details) are the same as produced by the Multifitter of the SFG: see Figure 7.27.

7.5.3. Making 2d maps from the fit results

To make 2d maps of the fit parameters for the peaked profiles (e.g. Gaussians) you fit to a cube is a matter of extracting them from the `ParameterCube` created by the fitting: either "MultiFit_ParameterCube" or "paramCube", depending on whether you followed the GUI or the command-line example above. It is only possible to extract images created from the parameters of the models, e.g. for a Gaussian model, the wavelength, peak, and sigma width, but it is possible to use these parameter datasets as a basis to create other maps, such as velocity or integrated line flux (intensity), or to change the units.

1. First you need to locate the variable that contains the parameters. If you did command-line fitting you now type:

   ```
   ParamCube = mf.getParameterCube
   ```

   Example 7.16. Extracting the parameters from the results of the multifitting.

   If you used the GUI, then the variable you want is called "MutiFit_ParameterCube".

2. The image layers are held in the `ParameterCube` in the same order that the models were specified and the same order that each model holds its parameters. Hence, if the first model is a 2nd order polynomial, the first three image layers is p0, p1, and p2 for the polynomial. If the second model is a Gaussian, then the next three image layers are amplitude, centre, and width, and so on.

   To extract the images of the wavelength, peak flux, sigma, and fwhm of the first Gaussian fitted in the example here, e.g.:

   ```
   peakMap  = MultiFit_ParameterCube.getSimpleImage(3)
   waveMap  = MultiFit_ParameterCube.getSimpleImage(4)
   sigmaMap = MultiFit_ParameterCube.getSimpleImage(5)
   fwhmMap  = sigmaMap.copy()
   fwhmMap.image = fwhmMap.image*2.3548 # to convert from sigma to FWHM
   fwhmMap.error = fwhmMap.error*2.3548
   ```

   These are `SimpleImages`, each with the error array taken from the standard deviation fitting results and the WCS taken from the cube that was originally fitted.

3. However, we want a map of the radial velocity and of the integrated flux. The maps extracted above can be converted to an "intensityMap" and a "velocityMap" in the following way:

   ```
   # Get all the SimpleImages
   ```

```
peakMap       = MultiFit_ParameterCube.getSimpleImage(3)
waveMap       = MultiFit_ParameterCube.getSimpleImage(4)
sigmaMap      = MultiFit_ParameterCube.getSimpleImage(5)

# Get the data arrays
peakArray     = peakMap.getImage()
sigmaArray    = sigmaMap.getImage()
waveArray     = waveMap.getImage()
peakError     = peakMap.getError()
sigmaError    = sigmaMap.getError()
waveError     = waveMap.getError()

# Conversions - using the equations of a Gaussian and a
# basic error propagation
speedC = herschel.share.unit.Constant.SPEED_OF_LIGHT.value
speedC = speedC/1000. # into km/s
restLine = 806.7 # rest wavelength for your fitted line
intensityArray = peakArray * sigmaArray * SQRT(2*java.lang.Math.PI)
intensityError = SQRT((peakError/peakArray)**2 * \
                 (sigmaError/sigmaArray)**2)*intensityArray
intensityMap   = peakMap.copy()
intensityMap.setImage(intensityArray)
intensityMap.setError(intensityError)
velocityArray  = speedC*(waveArray-restLine)/waveArray
velocityError  = velocityArray*waveError/waveArray
velocityMap    = waveMap.copy()
velocityMap.setImage(velocityArray)
velocityMap.setError(velocityError)

# The radial velocity image here follows the radio convention:
#  c*(fitted_wavelength-rest_wavelength)/fitted_wavelength
```

Herschel is an FIR telescope so we use the radio convention for computing velocites from frequency or wavelength. The equation given above applies to spectra in frequency units. For spectra in wavelength units (i.e. PACS) you will use the equation v = c*(lambda-lambda0)/lambda if you wish to replicate this convention. The difference with respect to optical astronomy is in the denominator for the equation: lambda rather than lambda0.

4. **Map Units**. The output from the Spectrum Fitter does not include units. To can add the name of units, follow these examples

```
velocityMap.setUnit(herschel.share.unit.Speed.KILOMETERS_PER_SECOND)
intensityMap.setUnit(cube.getFluxUnit().multiply(cube.getWaveUnit()))
```

Example 7.17. Manually setting the unit of the velocity and intensity maps

The intensity map is the multiplication of the peak and the Gaussian sigma, and hence the units are the mutliplication of those units. Hence, for the SPIRE data fitted here, the units are W/(m^2·Hz·sr)·GHz The unit of the peak flux ("amp" as written above) is the flux unit of your data, and the unit of the integrated flux is that flux unit *times* the wavelength/frequency unit.

Note that some of the units of the fitted cube are carried though into the ParameterCube as Meta data: the flux and wave unit and their description.

Converting units. If you want more physical units, such as W/m^2, for the integrated flux maps, you can convert the units in the following way. These examples are of the most common PACS, SPIRE, and HIFI units that you will encounter.

- **To convert from Jy·micron to W/m^2:** Jy and micron are the units of **PACS** data. While you are building your flux image, the changes to the script above are:

```
...
intensityArray = peakArray * sigmaArray * SQRT(2*java.lang.Math.PI)
intensityArray = intensityArray * 2.99e-12/waveArray[row,col]**2
...
intensityMap.setUnit(herschel.share.unit.Power.WATTS.divide\
```

```
  (herschel.share.unit.Area.SQUARE_METERS))
```

Example 7.18. Worked example of a manual conversion of all data in a cube from Jy·u to W/m while creating flux maps.

- **To convert from W/(m^2Hz Sr)·GHz to W/(m^2Sr):** this is the unit of **SPIRE** cubes (which are mostly of extended sources). You only need to get rid of the "GHz", and so change the script that is given above to:

```
...
intensityArray = peakArray * sigmaArray * SQRT(2*java.lang.Math.PI)
intensityArray = intensityArray * 1e9
...
intensityMap.setUnit(cube.getFluxUnit().multiply(herschel.share.unit.Frequency.HERTZ))
```

Example 7.19. Worked example of a manual conversion of all data in a cube from W/(m2 Hz Sr) to W/m while creating flux maps.

If you want to get rid of the "Sr" in the unit, it is recommended you convert the cube units before you do any fitting. A task to do this should be provided as part of the SPIRE extended pipeline.

- For **HIFI**, to get units of T *times* frequency [K km/s] it is recommended that you first convert the cube to have velocity on the X-axis using the task convertWavescale. Other conversions, such as to Jy, should also be done on the cube prior to fitting, using the HIFI-provided tasks (check the HIFI documentation).

See the *Scripting Manual* for more information about working with `SimpleImages`, `Wcs`, `TableDatasets`, and `SpectralSimpleCubes` and also for the list of units accepted in HIPE.

7.6. Adding and Initialising Models

When using the SFG (the GUI):

- If you have set a preference (via the HIPE Preferences panel) to have models automatically fitted when the SFG is opened you will find these already displayed in the plot, with the model parameters filled in the Models panel to the right of the plot.
- The initial centroid and intensity parameters of "peaky" models, such as Gaussians and Sincs, are estimated from the position and value of the channel that deviates most from the mean of the spectrum, while the initial width is estimated from the position of the channel closest to the estimated centroid that has a value half of the estimated intensity. A Polynomial model is always initialised with order 0. The start and end points of the Polynomial are taken to be the average of the first 10% and last 10% of the data (containing non-zero weights) and a line is drawn between the two.
- To add a model, press the *addModel* button in the Models tab. By default a Gaussian model is applied, However, you can choose what model will be the default model to apply from the drop-down *Default Model* menu in the HIPE preferences. The initial fit estimates are written to the GUI form and also in the Data Selection Panel beneath the plot.
- By default the models have the names M1, M2, etc, but you can change this.
- You can change the model from the drop-down menu of model types (click on "Gauss") and the GUI form will update for the new model selected.
- Rather than initialise the model over the entire spectrum, you can draw ranges to initialise the model in. You can do this using the *select range mode* of Spectrum Explorer: press the *select one or more ranges* icon ( ) in the button bar at the top of the Spectrum Explorer; click and drag to select ranges in the plot window; and then press *Init Fit Parms* (initialise fit parameters) in the model panel. Removing the markers afterwards does not affect the initialisation. When drawing ranges you may find it helpful to zoom in on the plot, you can do this using the *zoom* mode of the Spectrum Explorer.

Note

This is *not* the same as masking regions in the spectrum for inclusion or exclusion in the fit, that is done by setting weights (see Section 7.8), and which must be done from the Weight tab. You can tell the difference between the two as setting an initialisation range will not add rows to the table in the Weights tab.

When initialising "peaked" models such as Gaussians, you will only draw one initialisation range per model: if you draw more than one range then the last one drawn will be the one taken when you "Init Fit Parms" for any particular model. You can draw as many initialisation ranges as you need for more linear models, like the Polynomial model. The Polynomial is *always* initialised as a first order model, even if you select a higher order. Linear models require less precise initial estimates (0 would be sufficient) so this is enough for the fitter to work from. The Polynomial initialisation is calculated based on the first and last ten percent of the selected region.

The advantage of this approach is that you do not need to mask out a line in the middle of your spectrum in order to apply a Polynomial fit.

- Further models can be added to the data by pressing the addModel button again.

When using the command line: we assume you have begun the SpectrumFitter ("sf") following the instructions in Section 7.1.2.

- Models are added and initialised in one step. For example, a Gaussian model is set up by:

```
a0 = 12.3  # amplitude
x0 = 5.6   # location of line peak
s0 = 2.5   # sigma (NOT FWHM)
#
model = sf.addModel('gauss', [a0, x0, s0])
# and you can add any number of other models (M1, M2, etc)
```

Example 7.20. Adding a new model to an instance of the Spectrum Fitter

7.7. Configuring the Spectrum Fitter GUI to automatically apply a fit upon opening

It is possible to configure the SFG to automatically fit models to spectra on start up. You can set this up in the SFG section of HIPE preferences (found in the *Edit* menu). You may choose to have a Polynomial and a Gaussian model automatically fitted (typical choices for PACS and HIFI spectra), or a Polynomial and a Sinc model (a typical choice for SPIRE spectra), or you can choose in another of the models available to the SFG from the drop-down the *User models* menu, see Table 7.1 and Table 7.2 for a list of available models.

You can also set a typical width for Gaussian, Sinc, Lorentz and Voigt models in the preferences.

7.8. Setting weights

Using the SFG: selecting weighted ranges in the Weights tab.

- Weighted regions can be set to direct the fitting to or away from certain parts of a spectrum. They should not be used to try to set ranges over which to *initialise* models, instead the initialisation should be done as described in section Section 7.6 above. However, effectively-speaking you can set weighted regions at any time in the fitting process (even before you set the models).
- You can define any number of weighted regions. Each region can have its own weight. After clicking on the Weights tab of the SFG, you can define the region using the *select ranges* icon in the

button bar and drawing the range on the plot with the mouse. The values will be entered in the GUI automatically and a new box will appear for the next weight. Values in the GUI can be edited by dragging the edge of the marker in the plot, and a right click will give access to a menu that includes the option to remove the range. Alternatively you can enter and edit the ranges by hand, in this case you need to click *Add* to add a new range.

On the command line instructions: we assume you have begun the SpectrumFitter ("sf") following the instructions in Section 7.1.2.

- Weighted regions are set by:

```
x0 = 4.5 # start of weighted region
x1 = 8.5 # end of weighted region
w = 0.5 # weight value
sf.setMask(x0, x1, w)
```

Example 7.21. Setting weighted regions for the fitting using the setMask method.

The above will set weights to be 0.5 in the region `x0` to `x1` (if `w` were not specified the weight would be set to 1) and all other regions receive a weighting value of 0.

Here a potential point of confusion sets in. Outside of any specified region, weights will automatically set to zero. Even if you set weights to zero in the range x0 to x1! This means that if you want to effectively mask a region out (exclude it from the fit) then you need to set the weight of the data points outside of that region to something higher than zero, e.g.:

```
# Set weights in range x=1 to x=3 to zero (exclude x=1 to x=3 from fit),
# in a spectrum spanning x=-10 to x=10
#
x0 = -10
x1 = 1
x2 = 3
x3 = 10
#
sf.setMask(x0, x1, 1)
sf.setMask(x2, x3, 1)
```

Example 7.22. Setting "binary" weights to effectively mask out ranges of the spectrum from the fit.

You can also set a weight for the entire x-axis with:

```
# weight is a Double1d with the same length as the x-axis
sf.setWeight(weight)
#
# Give the x-axis the weights in the data, for a spectrum called
# "segm"
sf.setWeight(segm.getWeight())
```

Example 7.23. Setting a mask array that weights every point in the x-axis.

- You can define any number of weighted regions. Each region can have its own weight.

General instructions for both methods:

- A weight has a value greater than 0. By default, all weights are set to 1. However, if you select a region and set weights in it, then the weights everywhere else in the data (in which a weight is not set) will be set to zero.

- Weights are used to assign a weighting to the data when a fit is applied and apply to *all* models that are initialised when the *Accept* button is pressed of the fitting is commanded. This means, for example, that if you have defined weights everywhere except for the line in a spectrum in order to fit a Polynomial and then decide to fit a Gaussian to the line you should also add a weight to the line otherwise the weight there will be zero (see point above).

However, recall that the Polynomial model is initialised on the averages first and last 10% of the data (of each region containing non-zero weights) in the spectrum, see Section 7.6, so it may not be necessary to assign weights for fitting Polynomials for fitting to uncomplicated spectra. If you find it is necessary to use weights to get a good Polynomial fit then you may get better results by fitting to the residual after the Polynomial has been subtracted.

- Weights can be used effectively to mask out a region, or regions, from the global fit.
- If some regions of your data are particularly noisy and you do not wish them to affect the fit too much then either mask them out completely (by setting the weights everywhere else to a non-zero value) or give them a low weighting.
- If your data contains a weights column (in a dataset called "weight"), then you can also apply any weighting already present in the data, such as pipeline defined weights or weights you have applied yourself by checking the "use data weight" box. *However, if you also define weighted regions, then the data-weights are not into account*.

To learn about using weights when fitting multi-spectra datasets (e.g. cubes) see Section 7.21.

7.9. Setting limits to model parameters

It is possible to set limits on parameters to be used in models. In the GUI this is done by clicking on the *Low* and *High* boxes for each model parameter in the *Models* panel and filling in the desired limits by hand. By default, these boxes are filled with `NEGATIVE_INFINITY` and `POSITIVE_INFINITY` and do not set any limit on the parameter. It is not possible to click on the spectrum to fill the limits.

Limits applied to a model or models will be passed to the multifitter, see Section 7.21 for more information about fitting models to multiple datasets using the *multifitter*.

In the command line, limits are set using `setLimits`, following the format,

```
m.setlimits(parameter index, lower limit, upper limit)
```

Example 7.24. Setting limits for MultiFitting.

For example, assume you add a Gaussian model and want the centroid (the parameter with index 1) to be within 12.5 and 13.5:

```
sf = SpectrumFitter([your data])
m = sf.addModel('gauss', [...])
m.setLimits(1, 12.5, 13.5)
```

Example 7.25. Setting limits for a particular model parameter in the MultiFitter.

You can also do the same thing by limiting *all* the parameters in the model. This is using the format:

```
m.setLimits([lower limits for all the parameters of the model, comma seperated],\
 [upper limits for all the parameters of the model, comma separated])
```

Example 7.26. Setting limits for all the parameters of a model in the MultiFitter (generic).

The Gaussian model has three parameters: amplitude, centroid and (sigma) width. To limit the centroid parameter as above you would use:

```
m.setLimits([0, 12.5, 0], [0, 13.5, 0])
```

Example 7.27. Setting limits for all the parameters of a model in the MultiFitter (with values).

In this case we have not applied any limits to the first (amplitude) and third (width) fit parameters because the lower and upper limits are the same (0), therefore the second case is exactly the same as the first.

If you do not set any (or all of the) limits in the GUI and then export a script, you will find that the unset limits are set to `NEGATIVE_INFINITY` and `POSITIVE_INFINITY`, which are the default values filled in the GUI. This is equivalent to setting the max and min limits both equal to zero (or any other identical pair of numbers).

Note that if a parameter value is outside the interval [low, high], it will be silently changed to the nearest boundary (lower or higher limit). This is a useful indication that the fitter could not find a uniquely good fit within the ranges you set.

7.10. Fixing model parameters

You can fix a model parameter so that it can not be varied in the fitting routine. Do to so in the GUI, check the *Fixed* box beside each parameter field in the *Models* panel.

In the command line, use the `setFixed` method. This uses the format **`m.setFixed([0,1,2,...])`**, where `m` is the name of the model and 0, 1, 2,... are the indices or the model parameters (for as many model parameters as you need). For example:

```
# set a model Gaussian
m = sf.addModel('Gauss' [10.0, 4.5, 0.35])
# fix amplitude (first parameter)
m.setFixed([0])
# or fix amplitude and width (third parameter)
m.setFixed([0,2])
```

Example 7.28. Fixing the value of a model parameter during fitting.

When you fix a parameter the errors and chi squared are zero.

If you fix a parameter for a model fit and then apply that model to multiple datasets using the multifitter (see Section 7.21), the parameter will be fixed for all datasets.

7.11. Modifying Models

Models can be edited by modifying the parameters in the panel that appears on pressing addModel, or by using the command-line equivalents. "Modifying" in most cases means: the order, the initial parameters, setting limits, fixing values. These various actions are all described in separate sections of this chapter: this is not repeated here.

The most commonly used models have specialised forms in the *Models* tab of the GUI, that allow you to easily modify the models.

- **Polynomial.** The order of the Polynomial is set by entering a number in the box and clicking on "Update". The GUI will be updated for the order of the Polynomial selected. The remaining parameters are filled when the fit is done.

- **Peaked profiles.** Specialised GUI forms exist for Gaussian, Lorentzian, Voigt and Sinc models. In the case of these peaked profiles, the model parameters can be modified by entering numbers into the boxes or by mouse interaction. Click in the box beside "Intensity" or "X-position of peak"-they both will turn yellow-and click on the plot where the peak (or absorption dip) should be.

  The width of the profile is passed in the same way, e.g., for a Gaussian click in the box beside “width"-the contents will be highlighted yellow again-and then click near the X-axis position of the

line width in the plot (in other words, move the mouse to one side of your spectral line and click on the line at the X-and-Y point that is half-way between the line peak and the continuum level). When clicking on the plot be careful to avoid plot axes or plotted lines, which will select these objects (selection is denoted by a thicker line).

For the Gaussian profile the FWHM value for the width value is given below the width box. Note that the value reported in the FitResult tab will always be sigma, not FWHM.

- See Table 7.1 and Table 7.2 for more information about models. You can also find command-line help about the models and their parameters, for example for the Polynomial model, one you have set up an "sf" on the command line :

```
print sf.info('polynomial')
```

Example 7.29. Printing the model information, including expected parameters.

- After modifying your models you can see the change to the initialisation by clicking on the Update Plot button.
- You can fix parameters by checking the *fix* check box next to each parameter. To fix parameters in the command line:

```
m = sf.addModel(...)
fixed = [i0, i1, ...]
m.setFixed(fixed)
```

- You can go back to the previous initial fit values by pressing Previous Parms (previous parameters). You can keep on going back until you reach the very first set of initial parameters.
- All models can be reset to their initial values (or values before the last fit) by pressing the *Reset all Models* button in the Models tab.

7.12. Applying a fit

- Once you have your model initialised in a way that you are happy with, you can click on Accept at the bottom of the panel. A global fit will be performed, summing all of the models you have applied to the entire spectrum (that contains weights).
- You can continue to apply new models at this stage, the models will be applied to the original data.
- In the command line, the global fit is done with:

```
sf.doGlobalFit()
```

Example 7.30. Running the fit after setting parameters and (optional) limits and masks.

7.13. Inspecting fit parameter results

For fits to single spectra,

- Parameter results appear in a table inside a new tab in the *Data Selection Panel* (below the *Spectrum Panel*). Values and standard deviations for fit parameters are shown next to the parameter names, which are self-explanatory; the individual parameters are listed in columns called Name_0(1,2..), Value_0(1,2,..) and StdDev_0(1,2,..).

  Note that the width reported for Gaussian models is not the FWHM but the Gaussian σ: $FWHM = 2\,(2\ln 2)^{0.5}\sigma$ or approximately 2.3548 σ.

In addition, the *Integrated* column shows the integrated flux of the model over the whole spectrum region. See Section 7.19.

- You can print the model fit parameters for a model called "m" using:

```
print m.getFittedParameters()
```

Example 7.31. Printing the fitted parameters of a model.

- You can get a table of the fitted results from "sf" with:

```
table= sf.getResult()
```

Example 7.32. Creating a table of fitted parameters.

This includes the fitted values and their fitting errors.

For fitting to cubes and other multi-spectra products,

- When using the MultiFit part of the GUI, before pressing to *Accept* the fitting, you can chose to send the fitting results to an ASCII file on disk, and that file will look something like this:

```
#
#
# Fit parameters for models.
# Written by: SpectrumFitter 9.51, 24JUN2013
# Date: 18NOV2016-07:24:13
#
# The output contains the following models:
# M1 = Polynomial; parameters [p0, p1, ..., integrated value]
# M2 = Gauss; parameters [p0, p1, ..., integrated value]
#
# Output columns are:
# x y (indices of the spectrum in the Cube.)
#      [model name] [model parameters and standard deviations in order]
#           [background value (for the non-poly models only)]
#
# Column     Name           Unit
#    C1     polycoeff_0      Jy/pixel
#    C2     stdev of C1      Jy/pixel
#    C3     polycoeff_1      Jy pixel-1 micrometer-1
#    C4     stdev of C3      Jy pixel-1 micrometer-1
#    C5     Integral         Jy pixel-1 micrometer
#    C6     amplitude        Jy/pixel
#    C7     stdev of C6      Jy/pixel
#    C8     center           micrometer
#    C9     stdev of C8      micrometer
#    C10    width            micrometer
#    C11    stdev of C10     micrometer
#    C12    Integral         Jy pixel-1 micrometer
#    C13    Background       Jy/pixel

#
#         C1        C2        C3        C4        C5        C6        C7
 C8        C9        C10       C11       C12       C13
0 0 Fit Failed.
1 0 Fit Failed.
2 0 Fit Failed.
3 0 Fit Failed.
…
4 4 M1  -3.30361721E+00 +1.20588841E+00 +5.33732251E-02 +1.90837532E-02
 +3.58062675E-02
4 4 M2  +3.61274193E-02 +1.60796976E-02 +6.31961934E+01 +6.52472109E-03
 +1.27967941E-02 +6.68855547E-03 +1.16536816E-03 6.88E-02
5 4 M1  -1.03159484E+00 +8.75524873E-01 +1.78953934E-02 +1.38554343E-02
 +5.13172872E-02
5 4 M2  +5.07607838E-02 +1.13305452E-02 +6.31914419E+01 +3.47677301E-03
 +1.36031077E-02
```

```
...
```

In this file are the spaxel column and row coordinates, the fitting results, the errors, integrated fluxes, and for peaked models also the value of the continuum/baseline of the spectrum at the wavelength/frequency of the peak.

Warning

The value of the **line width** for the Gaussian profiles reported in this ASCII file **is** the FWHM (fwhich is sigma*2.3548). In contrast, the value of the line width that is in the MultiFit_Parms product that the MultiFitter also creates and which you will find in the HIPE *Variables* panel is the sigma of the line.

The same is true for the output of the SpectrumFitter, i.e. when you fit a single line: any ASCII output to disk reports the FWHM, any output to HIPE reports the sigma, if your line model is a Gaussian.

The names of the models and the order the parameters (and errors) are given in is indicated at the top of the file. The first two columns are spaxel column and row. Note that this order is swapped from the order that spaxel coordinates are usually handled in HIPE. Failed fits occur always for spectra that contain only NaN values, as will be the case for spectra at the edge of mosaic cubes.

- When using the GUI, after the fitting is done a set of results is sent to the *Variable panel*: MultiFit_Params contains the fitted parameter values. Click on the product in the *Variables* and the viewer will open in the *Console*. The models are listed in the Data tab, with M1, M2, M3.... and the spaxel (column, row) coordinates indicated. Click on any of these lines, and a table of those parameters will open.
- When using the MultiFitter on the command line, the same parameter file as MultiFit_Params can get obtained with

```
Params = mf.getProduct()
```

before you do the fitting. Saving the output as ASCII to disk is done by setting

```
mf.setFileName("/Users/me/parameters.txt") # or whatever filename you want
```

before executing the multifiting, and the ouput is the same as that produced by the GUI.

For more information about the output from the MultiFitter, see Section 7.21.

7.14. Deleting models and excluding models from a fit

- To delete a model, click on the delete button in the model panel. It will be deleted from the model panel and also the selector panel, where it will be replaced by *DELETED*. The global fit will not be modified until the next time you calculate one.

  To delete model `M1` in the command line:

```
sf.removeModel(M1)
#
# Delete all models
sf.empty()
```

Example 7.33. Removing one or all models from an instance of the Spectrum Fitter.

- To exclude a model from the global fit without deleting it, uncheck the *use* box in the model panel.

7.15. Resetting and restarting fitting

Or "resetting" and "starting afresh". What's the difference?

If you Reset the SFG you will still be able to plot the data in an old `FitResult` but you will lose access to the models used. If you "start afresh" using the *New Fitting* button you open a completely new version of the SFG and retain access to the models used in the old version. This can be helpful for comparison if you are fine-tuning two different modelling approaches.

- **Resetting on the original spectrum.** If you have multiple items from the `FitResult` in the top tab in the Data Selection Panel plotted in the display, then resetting the SFG by pressing the *Reset* button at the bottom of the task GUI will:

  - Remove all the model panels from the task GUI.

  - Remove all of the spectra in the `FitResult` you were working on from the plot.

  - Create a new `FitResult` variable in which to work (called `FitResult_1,2,3`... by default). Loaded in this `FitResult` tab will be the spectrum that you chose to do the fitting on previously.

  The original spectrum is now displayed in the plot and you can start fitting the data again.

  Pressing *Reset* when only the original spectrum in the `FitResult` is plotted will also cause the SFG to reset on the original spectrum as described above.

- **Resetting on the residual.** Pressing *Reset* when only the residual in the `FitResult` is plotted will also cause the SFG to reset on the residual as described above.

  To continue fitting on the residual in the command line:

```
sf = SpectrumFitter(MySpectrum) # Start the fitter, here on MySpectrum
m1 = sf.addModel(...) # Add models
sf.doFit( ) # fit the model that has been added most recently to the data.
sf.residual( ) # calculate the residual
sf.fitOK() # take the residual as new data
m2 = sf.addModel(...) # Add a model, now to the residual
sf.doFit( ) # fit the model, here m2
```

  Example 7.34. Mostly complete example on iterative fitting (using the residual for the next step).

- **Resetting on a different spectrum.** If the top tab in the Data Selection Panel is *not* a `FitResult` and only one spectrum is displayed in the *Spectrum Panel* then pressing *Reset* will reset the Spectrum Fitter GUI on the displayed spectrum.

- **Starting a new Spectrum Fitter GUI.** The *New Fitting* button follows the rules described for resetting above but creates a completely new SFG (SpectrumFitterGUI[2]) that will run in parallel with the original SpectrumFitterGUI. You can switch between the different fittings using the drop-down box just below the *Spectrum Fitting* tab in the top of the SFG window.

  When you create a new instance by pressing the *New Fitting* button the spectra and model plots of the previous instance of the SFG are also removed from the display, except the “Spectrum" that was being fitted there. The new instance of the SFG creates a new `FitResult` variable.

  This can make it somewhat confusing to keep track of which spectra and models belong to which fitting result. At present, the best recommendation is to rename your `fitResult` variables to something meaningful and to take advantage of the "All" button in the selector panel to add and remove all plots from the display as you inspect each `FitResult`.

7.16. Saving a script

You can save a script of the actions in the SFG. In the Export tab check the *Save Script* box. Enter a name, remembering to include the .py extension, and press Accept at the bottom of the task GUI to

save the script to the directory from where HIPE was started. Alternatively, press the *Select* button to choose the location the script will be written to.

It is a common mistake to forget to click on Accept after selecting the file location-if you forget, no script will be saved!

In the command line a script is saved with:

```
sf.writeScript(filename, name of input spectrum product)
# The input spectrum product is a string, with quotes, e.g. "spec"
```

Example 7.35. Exporting a fitting script with the same actions performed in the GUI

For MultiFitting via the SFG, saving a script is done in the *MultiFit* tab, this saving all commands from when you chose the reference spaxel until the multifitting.

On the command line you can save a script also, but this includes only the multifitting part. This is done by setting the filename of the script just after having set "mf=MultiFit(cube)"

```
mf.setScriptFilename(filename)
```

7.17. Saving the residual and models

We start with an explanation of the single spectrum case, i.e. using the spectrum fitter to one spectrum (either a single-spectrum product or a single spaxel/pixel in a cube).

Always remember to press *Accept* at the bottom of the task GUI to save after having selected what and how to save.

In the command-line examples below it is assumed that you started the SpectrumFitter (`sf`) following the instructions in Section 7.1.2.

- **Residuals.** In the Export tab, check the *Save Residual Spectrum* box and enter the path and filename to which the residual should be saved. You can enter this by hand or use the *Select* box to the right.

  Then select the file format from the *File Format* drop-down menu. Choose from:

  - *Plain Text File:* this will save the residual spectrum as an ASCII file. You also need to select how the columns will be separated from the *Text* drop-down menu, where you can choose from *Single Space*, *Tab* or *comma* separation.

    The command-line version is:

    ```
    sf.saveResidualAsAscii(filename, extension, separator)
    # Where the separator is one character, e.g., ',')
    ```

    Example 7.36. Saving a residual data as an ASCII file.

    The file name you selected is appended with *_res* when it is written to disk. An *info* file is also generated, which gives the date of creation, the column headers and the type of data saved.

  - *Local Pool:* this will create a `SFGResultsContext`, which will appear in the HIPE *Variables* pane. This will contain a `SFGResidualProduct`, which is the residual in the same format as the input data. Note that if you fitted to one pixel in a cube, the result will be a cube of the same size containing only the residual in the pixel you fitted to and zeros in other pixels. The `SFGResultsContext` can be saved to local pool by right-clicking on the variable name and selecting Send To → Local Pool. This option is useful if you intend to continue working on the residual in HIPE.

    To obtain the residual in the same format as the input data, use the following in the command line.

```
sf.getResidualAsInput()
```

Example 7.37. Retrieving the residual in the same format as the input was.

You would need to wrap this into a `Product` before you can save it in a local pool on disk.

You can choose both of these options together to save the residual in both text and lcoal pool format at the same time. You can also save the residual along with the model parameters and model spectra, as descrbed below.

- **Model Spectra.** In the Export tab, click on the name(s) of the model(s) you want to export. You can select more than one by holding down the Shift key (you may need to click on each model name with the mouse rather than use Shift + drag or Shift + ↑/Shift + ←, this seems to be platform dependent). These model names are those defined in the model tab next to the drop-down model type menu, by default they are M1, M2, M3...

  Then check the box *Save Model Spectrum* and enter the path and filename to which the model spectra should be saved. You can enter this by hand or use the *Select* box to the right.

  Then select the file format from the *File Format* drop-down menu. Choose from:

  - *Plain Text File:* this will save the residual spectrum as an ASCII file. You also need to select how the columns will be separated from the *Text* drop-down menu, where you can choose from *Single Space*, *Tab* or *comma* separation.

    On the command line this is:

```
# To save model M1
sf.saveModelAsAscii(M1,filename, extension, separator)
# or to save the total model
sf.saveTotalModelAsAscii(filename, extension, separator)
```

Example 7.38. Exporting the (total) model details as an ASCII file.

The filename you specified will be appended with "_M1", "_M2", etc for each model you selected to save, where the model number is associated with the model name. However, if you already have some models in your session you will find that the model numbering continues on from those models. The file for the total model will be appended with "_tm". In addition, an info file that contains the column headers for the ascii file and details of the date of formation is created for each model file and is appended with "_info".

To give a concrete example, if you saved model1 as an ascii file with:

```
sf.saveModelAsAscii(M1,"/Users/me/model1", "txt", ",")
```

Example 7.39. Exporting model details as an ASCII file.

then *model1_M1*`.txt` and *model1_M1_info*`.txt` will be created in `/Users/me/`.

  - *Local Pool:* this will create a `SFGResultsContext`, which will appear in the HIPE *Variables* pane. This will contain a `SFGModelXProduct`, which is the model spectrum in the same format as the input data and where "X" is the model number. Note that if you fitted to one pixel in a cube, the result will be a cube of the same size containing only the residual in the pixel you fitted to and zeros in other pixels. The `SFGResultsContext` can be saved to local pool by right-clicking on the variable name and selecting Send To → Local Pool.

    On the command line you can obtain the model spectra in the same format as the original data in the following way:

```
# To save model M1
```

```
M1model = sf.getModelAsInput(M1)
# To save the total model
TMmodel = sf.getTotalModelAsInput()
```

Example 7.40. Getting the model or total model in the same format as the input data.

You would need to wrap this into a `Product` before you can save it in a local pool.

Note that if you are working on a multi-spectrum dataset, such as a cube, if you fit only one spaxel/pixel of that cube and then save the result in this way, the result will be a cube with only the chosen spaxel/pixel fitted.

- **Model Parameters.** To save the parameters you need to select which model(s) to save from the top panel of the of the Export tab. You can select more than one by holding down the Shift key (you may need to click on each model name with the mouse rather than use Shift + drag or Shift + ↑/Shift + ←, this seems to be platform dependent). These model names are those defined in the model tab next to the drop-down model type menu, by default they are M1, M2, M3... The total model is a combination of all the models used in the fitting and, as such, it is not possible to save the total model parameters.

  Then choose from saving as a text file or generating a product to save to local pool as described above for the residual and model spectra. The parameters are stored in the `SFGResultsContext` as `SFGModelMXParameters`, where "X" is the model number.

  The model parameters can be saved to disk in various formats in the command line.

  - *Text:*

    ```
    models = [M1, M2, ...]
    sf.saveFitParametersAsASCII(filename, models)
    ```

    Example 7.41. Saving the fit parameters to an ASCII file.

  - *XML file:* this is done as part of *Save model spectrum as text* described in the list above: the file created will be called "SFGModelParameters.xml". This file can be used to load models back into HIPE for future use.

    On the command line this is:

    ```
    models = [M1, M2, ...]
    sf.saveModelParmsToXML(filename, models)
    ```

    Example 7.42. Saving the model fit parameters as an XML file.

  - *As a TableDataset:*

    ```
    # To save the parameters of a model M1 as a TableDataset
    # called "M1_tds":
    M1_tds = sf.getModelAsTds(M1)
    # or, to save the data as a TableDataset wrapped in a product:
    Model1 = sf.getModelAsProduct(M1)
    # To save the parameters of all models as individual
    # TableDatasets wrapped up in a Product called "Mall_tds"
    models = ['Model1', 'Model2' ...]
    Mall_tds = sf.getModelsAsProduct(models)
    ```

    Example 7.43. Getting the model parameters as a TableDataset.

    Note, there is an issue when using `getModelsAsProduct`. You *must* specify the models as 'Model1', 'Model2', ... exactly as written, irrespective of what the models are labelled in the SFG. The SFG by default labels models as 'M1', 'M2', etc - and you are free to label models as you please - however, the code underneath `getModelsAsProduct` only recognises the models under the name of 'Model1', 'Model2'. This is a 'feature' that will remain in HIPE.

The script generated with the *Save script* option also shows how to save models and parameters in all these formats in the command line. In addition, any cubes or spectra that are created by the fitting can be *Saved As* a FITS file by right-clicking on the name of the product in the *Variables* pane.

For MultiSpectra fitting, a description is given in Section 7.21. To summarise:

- Fitting multiple-spectra products via the MultiFitting tab of the GUI: upon executing a fit, a set of results is produced:

  1. MultiFit_M1,2..., MultiFit_TotModel, MultiFit_Residual: these are all products of the same format as that fit (i.e. they are cubes if you fit a cube), and contain the model spectra for each model, for the total model, and the residual of the total model subtracted from the data

  2. MultiFit_Params: see Section 7.21 for an explanation of this file, which contains the fitting parameter results.

  3. MultiFit_ParameterCube: a "cube" containing the model details and the fitting results in such a way that images and model cubes can be created from it. See Section 7.28 for more detail.

  The Export tab does not apply to multifitting.

- Using the multifitter on the command line: after having done the fitting the results can be extracted with various methods:

  ```
  residual = mf.getResidual()
  totalModel = mf.getTotalModel()
  model_1 = mf.getModel(0)
  ```

  The output spectral products are of the same class as the product fit, i.e. fit a cube, get cubes back.

7.18. Saving a SpectralLineList

You can also save the model fit parameters into a `SpectralLineList`, but only when fitting on the command-line. This is a Product containing a Dataset which holds the type of line-like model(s) such as, Gaussian, Lorentzian, Sinc, etc. applied to the spectrum and their position, widths and amplitudes along with their standard deviations.

You can use the LineList to record your model information, as a type of line identification annotation in a plot, or you can use it to help identify lines in other spectra.

- To retrieve a LineList in the SpectrumFitter use:

  ```
  sf.getLineList()
  # or for a subset of models models = ['name1', 'name2', ...]
  sf.getLineList(models)
  ```

  Example 7.44. Saving the model fit parameters as a SpectralLineList for use with other spectra.

- You can overlay the LineList on your spectrum by dragging the LineList product into an already open plot in Spectrum Explorer.

- To save the LineList to disk, right click on the Product name in the Variables pane and select the Send To option. You can save a LineList as a FITS file, a text file or to local pool.

For more information see *SpectralLineLists*.

7.19. Obtaining a line integral

- The SFG integrates under every fit. The value is shown in the selector panel and, more usefully, in the text file when the models are saved as ASCII. The values in the text file have an associated standard deviation and, for the non-Polynomial models, a background value.

- In the command line, the integral is obtained by:

```
# where "m" is a model that has been defined, see
# Section 7.6:
intflux = m.getIntegral()
```

Example 7.45. Getting the integral of the model line.

- The integrated flux (or, more correctly, in the case of PACS and SPIRE, integrated flux density) is calculated using the Trapezium rule to connect the datapoints and "add up" the flux under the line. Note that, the numerical result from this method does depend on the spectral dispersion: if you have a low spectral resolution (big gaps between datapoints) then the flux may be very slightly underestimated. Hence, that the value this reports may not be exactly the same as the value you will get with the analytical formula for your model, e.g., for a Gaussian, the area under the curve can be calculated as $amplitude * fwhm * (2 *\pi)^{0.5}$.

  An example of how to calculated the integrated flux for a 2d image taken from a MultiFitting done via the SFG on a cube can be found in Section 7.5, and we repeat the relevant lines here:

```
# Get all the SimpleImages
peakMap       = MultiFit_ParameterCube.getSimpleImage(3)
sigmaMap      = MultiFit_ParameterCube.getSimpleImage(5)

# Get the data arrays
peakArray     = peakMap.getImage()
sigmaArray    = sigmaMap.getImage()
peakError     = peakMap.getError()
sigmaError    = sigmaMap.getError()

# Conversions - using the equations of a Gaussian and a
# basic error propagation
intensityArray = peakArray * sigmaArray * SQRT(2*java.lang.Math.PI)
intensityError = SQRT((peakError/peakArray)**2 * \
                 (sigmaError/sigmaArray)**2)*intensityArray
intensityMap   = peakMap.copy()
intensityMap.setImage(intensityArray)
intensityMap.setError(intensityError)
```

Example 7.46. Computing the integrated flux after MultiFitting a spectral cube.

  See the example in Section 7.5, and also Section 7.28 to learn about converting the units and adding them to the images.

- **Errors**. The stddev of the parameters are also reported, as you can see in the example above.

7.20. Using Saved models

To use previously saved models, click on *loadModels* in the Models tab and locate the XML file on disk.

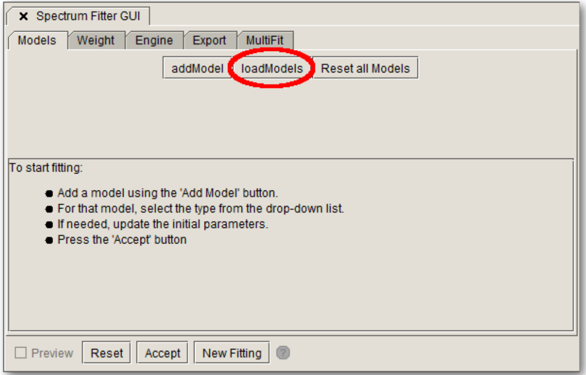


Figure 7.28. Loading previously saved fitting models.

7.21. Automatic fitting of multiple datasets

Note

You should make a variable of the dataset you wish to fit rather than opening a spectrum from the Observation Context in the Observation Viewer. If you do not work from a variable, the SpectrumFitter will not know where to look for the rest of the spectra you fit to and will hang. No warning message is given in this circumstance.

Having defined a fit (including any parameter limits, see Section 7.9, or fixed parameters, see Section 7.10,) to one spectrum you can now apply these initial model parameters to many spectra in one dataset automatically, for example, a spectral cube. You can also use a model that has been saved to XML file, see Section 7.17 and Section 7.20.

- Go to the *MultiFit* tab. Pressing Accept will fit the entire dataset with the same set of initial model parameters just defined. The data are fit in the order they appear in the dataset (look in the *Dataset Viewer* if necessary to see this).
- Note that the same initial model parameters are applied to all the data, *not* the same model parameters found as a best fit for the initial spectrum. If your initial parameters were not good but the fit was acceptable, you could fit the single spectrum again with the better parameters given as the initial guesses.
- If any of the spaxels/pixels/spectra in dataset cannot be fitted a message will appear informing you of the number of failed fits.
- The fitter ignores NaN data so any fully-NaN spectra in you data set should have a residual and model that is NaN. However, these fully-NaN spaxels at the edge of the field of view of PACS mosaic cubes seem to fail in the fitting, rather than being ignored. For such cubes some of the "failed" fits will simply be NaN spectra rather than 0 spectra.
- The fitted parameters can be written to text file by checking the "Write ASCII file with parameters" box in the multi fit tab.

An example of the ASCII file output of the multi-fitter is below:

```
#
# Fit parameters for models.
# Written by: SpectrumFitter 9.34, 27MAR2012
# Date: 06MAY2012-01:45:56
#
# The output contains the following models:
# M1 = polynomial; parameters [x-axis crossover, slope, integrated value]
# M2 = gauss; parameters [amplitude, centre, fwhm, integrated value]
```

```
#
# Output columns are:
# x y (indices of the spectrum in the Cube.)
#      [model name] [model parameters and standard deviations in order]
#           [background value (for the non-poly models only)]
#
0 0 Fit Failed.
1 0 Fit Failed.
2 0 Fit Failed.
...
5 1 M1  +2.21E+00 +6.97E+00 -2.48E-02 +7.88E-02 +1.19E-02
5 1 M2  +1.99E-01 +5.19E-02 +8.83E+01 +5.51E-03 +5.37E-02 +1.30E-02 +1.14E-02
 2.51E-02
6 1 M1  +2.21E+00 +6.97E+00 -2.48E-02 +7.88E-02 +1.19E-02
6 1 M2  +1.99E-01 +5.19E-02 +8.83E+01 +5.51E-03 +5.37E-02 +1.30E-02 +1.14E-02
 2.51E-02
7 1 M1  +2.21E+00 +6.97E+00 -2.48E-02 +7.88E-02 +1.19E-02
...
```

- The "Fit failed" are obvious messages. The "0 0", "1 0" are the spaxels/pixels column, row indices:

Warning

About coordinates. The coordinates that the SFG reports in the ASCII file are the flip of the coordinates you see when you move your mouse over a cube image in the Spectrum Explorer.

In the cube image: At the bottom left of the cube display you will see indicated the (row,column) of the spaxel/pixel under the mouse. If this says, e.g (5,1), then your mouse will be 6 spaxels/pixels high along the Y-axis (the counting starts at 0) and 2 spaxels/pixels long the X axis.

In the Spectrum Fitter multifit output, as well as the coordinates reported in the FitResult tab for a single spaxel/pixel fit : the coordinates—5,1 above for the first line with results—are flipped with respect to this, i.e. `5, 1` is `column, row`.

- The parameters are reported in order,
  - For the case reported above a first order **polynomial** was fit as model M1, so the first and second, third and fourth values are the value+standard deviation for the first parameter (x-axis crossover) and second parameter (slope), and the fifth value is the integrated value (flux under the line over the spectral range that your spectrum has in it).
  - For the second model in the case above, M2, the **Gaussian** parameters amplitude(+stddev), centre(+stddev), and FWHM(+stddev) [**note: FWHM not sigma**] are the first 6 values, followed by the line integrated flux(+stddev) and finally the value of the continuum under the peak of the Gaussian.
  - The integrated value is calculated as described in Section 7.19.

The output of the multi-fitter also produces the following new products in the *Variables* pane: MultiFit_Residual, MultiFit_TotModel, MultiFit_M1, MultiFit_M2 [and MultiFit_M3,4.. if you defined more models), which are all `SpectralSimpleCubes` with the indicated spectra in them.

The product MultiFit_Parms is also created, which is a product containing `TableDatasets` that hold your results:

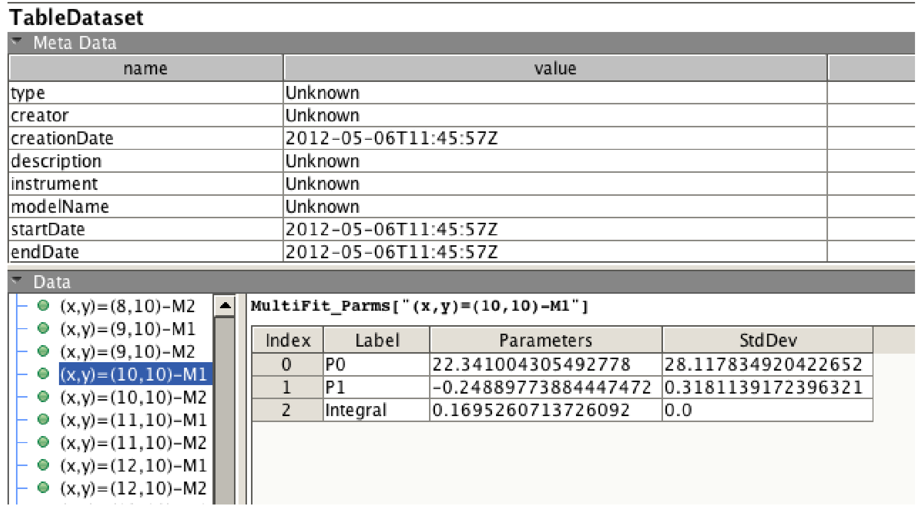


Figure 7.29. The MultiFit_Parms output

The coordinates reported are the same as those given in the ascii output.

You can access the MultiFit_Parms data thus:

```
print MultiFit_Parms["(x,y)=(4,1)-M2"]["Parameters"].data
print MultiFit_Parms["(x,y)=(4,1)-M2"]["StdDev"].data
```

Example 7.47. Printing several data in MultiFit_Parms.

where the wording in quotes above is exactly what you see when you look at the MultiFit_Parms with the Product viewer, as the figure above shows. The order of the parameters is indicated in the TableDataset when you view it as shown above and is the same order as printed to the ascii output file (e.g. amplitude, centre, width, and integral for a Gaussian model). **The "width" reported here is the sigma value, not the FWHM**. Unlike with the ascii output, the continuum value under the peak for peaked models (e.g. the Gaussian) is not reported.

Bear in mind that when working with a cube, it is possible that the spectra differ substantially from spaxel/pixel to spaxel/pixel, and this could affect the accuracy of the results. The SFG in multi-fit mode is an automatic fitter, and so cannot allow for all possibilities of variation of spectrum from spaxel/pixel to spaxel/pixel.

Using the multifitter in the command line requires you to use previously saved XML files. A command-line 'recipe' for multi-fitting goes as follows:

```
mf = MultiFit(data) ## data must be a SpectrumContainer

mf.exclude(p, s) ## do _not_ fit PointSpectrum p, SpectralSegment s
# (for cubes this does not work on spaxel coordinates, so the "s" is
# always 0 and the p is the spaxel in PointSpectrum order
# see Chap 6 for more information on "cube coordinates")
... # exclude any number of spectra you want
mf.setModels("/path_to_previously_saved_XML_file/file.xml")

mf.setMask(x0, x1, w) # see note about weights/masks below
... # set any number of Masks you want

mf.doFit()

residual = mf.getResidual()
totalModel = mf.getTotalModel( )
model_n = mf.getModel(n) # (n is number of model in the XML file)
```

Example 7.48. MultiFitting a SpectrumContainer with a set of previously exported models.

and you can also see the worked examples near the top of this chapter for a longer recipe.

On the command line, **weights** are handled differently by the multi-fitter than by the SFG and the Spectrum Fitter. Setting a weight in the Spectrum Fitter and Spectrum Fitter GUI automatically sets the weight in that region to 1 (or whatever you specify) and zero elsewhere. Therefore you can 'mask out' a region for fitting in the GUI by setting weights either side of the region you wish to exclude. When using the multi-fitter, setting a weighted region does not change the weight values outside of the masked region and so if you wish to exclude a part of the spectrum from the fit you must explicitly set the weight to zero there.

It is **not possible** to ask for MultiFitter to use the data-weights when doing fitting from the command line, and it will ignore any such request if MultiFitting from the GUI.

7.22. Continuing work on the residual outside of the Spectrum Fitter GUI

If you export the residual (or model) in the same format as the original data a `FitProduct` will be created. As it is a product, you can save it to pool or to a FITS file, or to export to VO tools. Within the product the residual is stored as the same spectral type of data as your input.

To view this spectrum you can double click on its name in the *Variables* pane and use the Product Viewer, from where the spectrum can then be plotted in the Spectrum Explorer.

If you wish to pass this spectrum to other data processing tools in HIPE, such as the tasks of the Spectrum Toolbox, then you need to extract the spectrum from the product. You can do this by clicking on the product, and from the *Outline* pane or from the display in the Product viewer open in the *Editor* pane, dragging and drop the spectrum name into the *Variables* view. The command-line syntax to do this will be echoed in the console.

7.23. Using the Combo Model

The Combo model only works for fitting single spectra, it cannot be carried though to multi-fitting.

It is possible to fit several models that have a fixed relation between them using the *ComboModel*. This facility is only available in the command line.

A ComboModel is a combination of multiple 'normal' models where a relation can be set between parameters of the 'normal' models (called internal model). For example, a ComboModel can have 2 Gaussians where the centre positions have a fixed distance, the amplitudes have a relation like A1:A2 = 1:0.75, and the sigmas are equal.

It is only possible to combine the same type of model (e.g., Gaussian, or Lorentzian), but you can combine as many of them as you like.

When setting up ComboModels you must first specify the relation between the parameters and only then can you set the parameters.

The example below is a script for a Combo of two Gaussians. The distance is fixed to 5 [units of the X axes], the relation between amplitudes is 1:0.75, the sigmas are allowed to run free.

```
x_1 = ... # initial guess for X-position of the first gaussian
a_1 = ... # initial guess for amplitude of first gaussian
s_1 = ... # initial guess for sigma of first gaussian
s_2 = ... # initial guess for sigma of second gaussian

sf = SpectrumFitter([your data])
# (see Section 7.1.2) to know how to
# properly do the above command
```

```
cm = sf.addCombo('gauss', 2) # add two Gaussian models
# linking parameter index 1 (the wavelength here), and the linking values
cm.setAddParms(1, [0, ])
# fix the relative peak values (parameter index 1)
cm.setMultParms(0, [1, 0.75])
cm.setParameters([a_1, x_1, s_1, s_2])
sf.doFit()
```

Example 7.49. Using ComboModels with a specific relationship between fit parameters.

(To see all the methods that the combo model has, look up the *DRM* entry for *SpectrumComboModel*.)

Fixing the distance between parameters is done with:

```
setAddParms(index, [p1, p2, ..., pn])
```

Example 7.50. Setting the parameters of the added model.

where the index is the index of the parameter in the internal model, so 0 for the Gaussian amplitude, 1 for the wavelength, 2 for the sigma. The [p1, p2, ..., pn] are the values of the parameters for the n internal models relative to 'some value'. In the example ([0, 5]), this 'some value' is the actual position of the first Gaussian, hence '0'. The second Gaussian is distance 5 from the 'some value' (in whatever units your data have, so for a spectrum in microns, this distance would be 5 microns). If you happen to know that the first Gaussian has a distance of 'd' from some position, then you could specify: [d, (d+5)].

Fixing the 1:0.75 relation is done with a multiplicator relation: setMultParms(index, [p1, p2, ..., pn]). Since this is a multiplication, when fixing a relation with respect to the actual value in the first Gaussian, p1 must be 1 (where it is 0 in the additive relation). Of course p1 need not be 1. If you would have specified [1.25, 0.75] then the relation between the amplitudes would have been 1:0.6.

To set parameters to be equal, you would specify:

```
setAddParms(index, [0, 0, ...])
# or
setMultParms(index, [1, 1, ...])
```

Example 7.51. Setting the parameters of a multi model fitting or a MultiFitting.

Specifying the initial parameters can be with the full number of parameters, so 3xn for n Gaussian, or with the reduced number of parameters. For every relation in an n-model Combo, you 'lose' (n-1) parameters. Hence the 4 parameters in the example: n = 2, and (3xn)-2x(n-1) = 4. Note, that even if you give the full nx3 parameters, the plots take the relations into account.

7.24. Models available to the fitter

`SpectrumFitter` is used in conjunction with `SpectrumModel`, which allows you to select and change models and fitting parameters. The three models you are most likely to use are Gaussian, Lorentzian and Polynomial, described in the following table. Note that the s_0 parameter of the Gaussian model is the sigma of the Gaussian, not the FWHM.

Table 7.1. Most common model fits and their parameters

Model	Mathematical fit	Parameters	Usage
Gaussian	$f(x) = a_0\, exp\left\{\frac{-(x-x_0)^2}{2s_0^2}\right\}$	a_0 = amplitude of line x_0 = location of line peak s_0 = width of line (sigma, not FWHM)	**`sf.addModel ('gauss', [a0,x0,s0])`**

Model	Mathematical fit	Parameters	Usage
Lorentzian	$f(x) = p_0\left[\frac{p_2^2}{(x-p_1)^2 + p_2^2}\right]$	p_0 = amplitude of line p_1 = location of line peak p_2 = half width at half maximum of line	`sf.addModel ('lorentz', [p0,p1,p2])`
Polynomial	$f(x) = c_0 + c_1x + ... + c_nx^n$	n = order of polynomial c_0 .. c_n = polynomial coefficients	`sf.addModel ('polynomial', [n], [c0,c1, ..., cn])`

Other available models are listed in the following table:

Table 7.2. Spectrum fit model types and their use.

Name	Example use – names in brackets should be replaced by numerical values representing the initial guess for the parameter(s)
`atan`	mod=sf.addModel('atan',[amplitude, slope, offset])
`exp`	mod=sf.addModel('exp',[amplitude, exponent])
`harmonic`	mod=sf.addModel('harmonic',[order, period],[params]). Number of parameters provided = 2*order + 1
`pade`	mod=sf.addModel('pade',[num, denom],[params]). Number of parameters provided = Num + Denom + 1
`power`	mod=sf.addModel('power',[degree], [param]). Number of parameters provided = 1
`powerlaw`	mod=sf.addModel('powerlaw',[amplitude, x-shift, power])
`sinc`	mod=sf.addModel('sinc',[amplitude, position, width])
`sine`	mod=sf.addModel('sine',[frequency, cosine amp, sine amp])
`sineamp`	mod=sf.addModel('sineamp',[frequency], [two params])
`voigt`	mod=sf.addModel('voigt',[amplitude, centre, gwidth, lwidth])

For more information about the SpectrumFitter and SpectrumModel classes, see the corresponding Javadoc entries in the *HCSS Developer's Reference Manual*.

7.25. How to add your own model

It is possible to create your own models to use with the command-line version of the spectrum fitting tool.

To do so, you need to create a Jython script containing a class that implements a NonLinearPyModel. The following example defines a class that implements this non-linear function with two parameters (*a* and *b*): *y = a cos(bx) + b sin(ax)*.

```
import java
from herschel.ia.numeric.toolbox.fit import NonLinearPyModel
from herschel.ia.numeric import Double1d
from java.lang import Math

ModelName = 'MyModel'

class MyModel(NonLinearPyModel):
  npar = 2  #  Define number of fit parameters

  def __init__(self):
    NonLinearPyModel.__init__(self, self.npar)

  def pyResult(self, x, p):
    # implement the model function

    y = p[0]*Math.cos(p[1]*x)+p[1]*Math.sin(p[0]*x)

    return y
```

Example 7.52. Creating a non-linear model for use with the Spectrum Fitter.

In the `SpectrumFitter`, this class can be loaded and its contents added to the `ModelLibrary` using one of two following methods:

- If you have defined the custom model class by running the script above and then created an object executing `customModel = MyModel()`, use this method: `SpectrumFitter.addJythonModel(org.python.core.PyObject pModel, double[] fParms) : SpectrumModel`

  This will add the object in `customModel` already in memory to the `SpectrumFitter` instance, with the initial *fitting* parameters as an array of double values. Example: `sf.addJythonModel(customModel, [50,50])`

- If you have placed the above Jython script in a file, you should use this other method: `SpectrumFitter.addJythonModel(String fName, String modelName, double[] fParms) : SpectrumModel`

  This allows to add the model with name `modelName` from the file located at `fName`. Note that this allows the file to have multiple models and reference them by name. You should also pass the initial *fitting* parameters as an array of double values.

In both cases the initial *fitting* parameters can be changed afterwards with `SpectrumModel.setParameters(...)`:

Warning

There are two sets of parameters: the initial *fitting* parameters (`fParms`) and the *constructor* parameters (`cParms`). If the defined custom model requires parameters for its constructor, you should use these alternative methods:

- `SpectrumFitter.addJythonModel(org.python.core.PyObject pModel, double[] cParms, double[] fParms) : SpectrumModel` providing the appropriate values for `cParms`.
- `SpectrumFitter.addJythonModel(String fName, String modelName, double[] cParms, double[] fParms) : SpectrumModel` providing the appropriate values for `cParms`.

```
sf = SpectrumFitter(...)
```

```
# (see Section 7.1.2) to know how to
# properly do the above command

jm = sf.addJythonModel(filename, modelname, parameters)
jm.setParameters(...) # If required
sf.doFit()
```

Example 7.53. Adding a custom model, previously exported as a Jython file.

7.26. Selecting the best fitter engine

In the Engine tab, you can select the fitting engine to be used. This can be done at any stage before a model is finalised. There are five fitter engines available:

- *'levenbergmarquardt'* or *'lbm':* The Levenberg-Marquardt algorithm is the default model. A robust method commonly used in fitting software and in many cases will find a solution for a non-linear fit even if the initial guess is far away from the solution
- *'amoeba':* The Amoeba fitter can be faster than the Levenberg-Marquardt algorithm and is the best at finding the absolute minimum in the Chi-squared.
- *'linear':* The Linear algorithm can be faster than the Levenberg-Marquardt algorithm.
- *'mp':* The MP fitter.
- *'conjgrad'*: The ConjugateGradientFitter (CGF) does not use matrix inversions to iterate to the solution (as LevenbergMarquardt). It calculates the gradient to the (local) ChiSq function and proceeds along that direction until it encounters a minimum. From that new position it iterates further. This behaviour makes it fit for solving problems with many (>10 or so) parameters.

In the command line the fitter engine is chosen using the name given in italics in the list above:

```
# Levenberg-Marquardt algorithim
sf.useFitter('lbm')
```

Example 7.54. Selecting the fitting algorithm to use.

7.27. NaNs and the Spectrum Fitter

The Spectrum Fitter handles NaNs (Not a Number)s in data by ignoring them. You do not need to replace NaNs in your data.

7.28. Making images from fitting results to cubes: the ParameterCube

When fitting the spectra in a cube using the MultiFitter via the command line or via the SFG, one can make images from the fit parameters—most commonly this will be integrated flux or velocity maps made from spectral line fits. A worked example (Section 7.5) shows you how to use the `ParameterCube` to do this, and we explain more about this product here. This "cube" is not a cube in the sense of having axes of Ra, Dec, and wavelength/frequency, but rather is a products that stores fitting results from cubes.

7.28.1. After fitting with the MultiFitter tab of the Spectrum Fitter GUI

The MultiFitter of the SFG produces a new product called "MultiFit_ParameterCube"—you should see it in the *Variables* pane once you have run the fitting on the cube.

It is from this parameter cube that you can extract fitting results to make images. The order that these images are held in is set by the order that the models were defined within the Spectrum Fitter GUI. For example, if your first model was a 1st order polynomial and your second a Gaussian, then the parameters (and images) are in the following order: poly_param_0, poly_param_1, Gaussian peak, Gaussian centre, and Gaussian sigma (width). To take any one of these images out of MultiFit_ParameterCube the following syntax can be used:

```
map1 = MultiFit_ParameterCube.getSimpleImage(1) # for the second parameter
error1= MultiFit_ParameterCube.getSimpleImage(1).getError() # the associated error
 image
```

Example 7.55. Extracting images from the ParameterCube after MultiFitting a cube

Where the number is the position in the order mentioned above (beginning with 0).

The images will not have units, which you will have to add yourself: see the next section.

It is possible to view these images directly: from the *Variables* pane, double click on MultiFit_ParameterCube (or right-click and select to Open With the Standard Cube Viewer) and you will get a display containing a series of images, which you can click through using the scroll bar at the bottom of the viewer. To extract the currently viewed image out of the parameter cube, right-click on the Standard Cube Viewer and choose the menu item "Extract current layer".

It is also possible to extract the total model cube from the `ParameterCube`. If "cube" is the name of the cube you fit, then:

```
totalModelCube = MultiFit_ParameterCube.getFittedCube(cube.wave)
```

There are several other useful functionalities of the `ParameterCube` (e.g. getting the ChiSq map, getting the parameters): more information can be taken from the entry for "ParameterCube" in the *DRM* (Developer's Reference Manual).

Note that this cube will not have descriptions for the units, and that can make it difficult to compare the "totalModelCube" to the original cube in the Spectrum Explorer. You can add the necessary information yourself

```
totalModelCube.setFluxDescription(cube.getFluxDescription())
totalModelCube.setWaveDescription(cube.getWaveDescription())
```

7.28.2. After fitting with the MultiFitter on the command line

When multi-fitting on the command line, a `ParameterCube` can be extracted from the fitting (called "mf" in this chapter) contains a `ParameterCube`.

To get the parameter cube from "mf", type:

```
parCube = mf.getParameterCube
```

Example 7.56. Retrieving the ParameterCube from the MultiFitter results.

It is now straightforward to use this product to make images from any of the models that you fit to your data. The parameters are stored in parCube in the same order that you defined the models and the order that the models require the parameters to be in: so, for example, if you fit the original cube with a 2nd order polynomial and a Gaussian, you will have 6 parameters in the following order: p0, p1, p2 for the polynomial, then peak, wavelength and sigma from the Gaussian. For each of these you can get the associated image using:

```
# Get Gaussian peak flux image
```

```
peakMap = parCube.getSimpleImage(3)
```

Example 7.57. Getting the peak flux image from the ParameterCube.

The images will not have units, which you will have to add yourself: see the next section

You can also open the "parCube" with the Standard Cube Viewer: from the *Variables* pane, double click on it (or right-click and select to Open With the Standard Cube Viewer) and you will get a display containing a series of images, which you can click through using the scroll bar at the bottom of the viewer. These images are held in the order discussed above. To extract the currently viewed image out of the parameter cube, right-click on the Standard Cube Viewer and choose the menu item "Extract current layer".

It is also possible to extract the total model cube from the `ParameterCube`. If "cube" is the name of the cube you fit, then:

```
totalModelCube = parCube.getFittedCube(cube.wave)
```

There are several other useful functionalities of the `ParameterCube` (e.g. getting the ChiSq map, getting the parameters): more information can be taken from the entry for "ParameterCube" in the *DRM* (Developer's Reference Manual).

Note that this cube will not have descriptions for the units, and that can make it difficult to compare the "totalModelCube" to the original cube in the Spectrum Explorer. You can add the necessary information yourself

```
totalModelCube.setFluxDescription(cube.getFluxDescription())
totalModelCube.setWaveDescription(cube.getWaveDescription())
```

7.28.3. Manipulating the images taken from the ParameterCube.

The parameter cube contains images of the fitting results, but they may not be directly what you want an image of. In addition, they do not contain units. For example, the integrated flux under a Gaussian is usually more interesting than the peak flux, or a wavelength/frequency map is less useful than a velocity map. Fortunately, once the images have been extracted from the parameter cube it is possible to manipulate them mathematically. We show this here using the example from fitting on the command line (but exactly the same process will work on products created by the MultiFitter of the SFG).

To turn the peak flux into an image of the integrated flux, you will need to apply the equation of the area under the Gaussian. Using the example of a fitting of a 2nd order polynomial (first model, three parameters) and a Gaussian (second model, three parameters), on data in Jy and microns:

```
peak     = parCube.getSimpleImage(3).getImage()
width    = parCube.getSimpleImage(5).getImage()
peakErr  = parCube.getSimpleImage(3).getError()
widthErr = parCube.getSimpleImage(5).getError()
intensity    = pk * width * SQRT(2*java.lang.Math.PI)
intensityErr = SQRT( (peakErr/peak)**2 * (widthErr/width)**2 )*intensity
# Quick and (a bit) dirty: grab any image from ParameterCube and
# replace the values with "intensity"
intensityMap = parCube.getSimpleImage(5)
intensityMap = setImage(intensity)
unit = FluxDensity.JANSKYS.multiply(Length.MICROMETERS)
intensityMap.setUnit(unit)
intensityMap.setError(intensityErr)
```

Example 7.58. Converting the peak flux image of the ParameterCube into an image of the integrated flux, for a PACS cube.

And you can do similar to create a velocity map from the wavelength/frequency map (see also the example in Section 7.5).

Converting units. If you want more physical units, such as W/m^2, for the integrated flux maps, you can convert the units in the following way. These examples are of the most common PACS, SPIRE, and HIFI units that you will encounter.

- **To convert from Jy·micron to W/m^2:** Jy and micron are the units of **PACS** data. While you are building your flux image, the changes to the script above are:

```
...
intensity = pk * width * SQRT(2*java.lang.Math.PI)
intensity = intensity * 2.99e-12/wave[row,col]**2
...
int_ima.setUnit(herschel.share.unit.Power.WATTS.divide\
  (herschel.share.unit.Area.SQUARE_METERS))
```

Example 7.59. Manual conversion of map units from Jy·u to W/m

- **To convert from $W/(m^2Hz\ Sr)$·GHz to $W/(m^2Sr)$:** this is the unit of **SPIRE** cubes (which are mostly of extended sources). You only need to get rid of the "GHz", and so change the script that is given above to:

```
...
intensity = pk * width * SQRT(2*java.lang.Math.PI)
intensity = intensity * 1e9
...
int_ima.setUnit(cube.getFluxUnit().multiply(herschel.share.unit.Frequency.HERTZ))
```

Example 7.60. Manual conversion of map units from W/(m2 Hz Sr) to W/m

If you want to get rid of the "Sr" in the unit, it is recommended you convert the cube units before you do any fitting. A task to do this should be provided as part of the SPIRE extended pipeline.

- For **HIFI**, to get units of T *times* frequency [K km/s] it is recommended that you first convert the cube to have velocity on the X-axis using the task convertWavescale. Other conversions, such as to Jy, should also be done on the cube prior to fitting, using the HIFI-provided tasks (check the HIFI documentation).

To learn more about the various methods of the `ParameterCube` (setting and getting models, parameters, noise, units, writing results out, etc) see the *DRM* (Developer's Reference Manual) and look for "ParameterCube". For example:

```
# Get the cube of the total fitted model
wave=cube.getWave() # get the wavelength (Double1d) array from the cube you fit
fittedCube=parCube.getFittedCube(wave)
# Get the fitted parameters and errors (for all spaxels fit) as 3d datasets:
parameters = parCube.getFitParameters()
errors = parCube.getStandardDeviations()
```

Example 7.61. Getting the cube of the total fitted model, and parameters and errors datasets.

7.29. Calculating uncertainty and error after fitting

7.29.1. Introduction to errors or fitting and confidence

To better explain the concept of fitting errors, let us assume that we have some sample data, `D`, and a model, `F`. In the most simple case the model has only one parameter, `F(p)`. (We will later expand it to more than one.) Let us also assume that we know the optimal setting for the parameter i.e. we have fitted the model prefectly to the data. Let us call the optimal parameter values `P`. For this optimal model we can calculate `chi^2` as the `SUM(D - F(P))^2`. The goodness-of-fit, `chi^2`, is related to the likelihood, `L`, as `L~exp(-0.5chi^2)`. The optimal model is where `chi^2` is minimum (least squares solution) or equivalently maximum likelihood (solution).

Concerning the errors of a fit there are two things you need to keep in mind.

- The "noise scale". This is the standard deviation of the data minus the model, or `sqrt(chi^2/dof)`, where `dof` is the degrees of freedom: number of data points minu number of parameters. The noise scale is also known as "standard deviation of the fit" and several other names. Noise scale is exactly that: the scale of the noise remaining after subtraction of the model. If the model is the right one, then the noise scale is equivalent to the the noise in the data (the true RMS of the spectrum). If you happen to have twice as many data points, the noise scale stays the same: twice as many residuals divided by `2*N`.

  The noise scale is not the same as error in the fitted parameter values.

- A more interesting error/standard deviation—for the person fitting in HIPE—is the one related to the fitted parameters. To find the error/standard deviation of a parameter you need to multiply the noise scale by the square root of the diagonal elements of the covariance matrix = inverse of the hessian (see note) matrix. In our simple case of one parameter, the hessian is one number, which generally increases with the number of data points. So our standard deviation on this single parameter goes down with the square root of the number of data points. More observations make your estimate better. If you have more than one parameter the off-diagonal elements of the covariance matrix tell you how much the parameters are correlated. See `MonteCarloError`, which uses the standard deviation plus covariance matrix to calculate a confidence region in where the fit can wiggle.

Note

The Hessian is the matrix formed by the second derivatives of `F` to the parameters `p`.

  What does it all mean? In Bayesian terms it means that the thing that `p` was supposed to measure, has most probably the value `P`. Actually there is a Gaussian distribution centred at `P` with a sigma equal to the standard deviation of the parameter, telling you how probable each value for `p` is. With more parameters there is a multidimensional Gaussian distribution, rotated according to the covariance matrix, telling you how probable each combination of parameters is. Note that by assuming a Gaussian distribution for the errors in the data (by doing least squares, chi^2 etc.), you get a Gaussian in the results.

  This standard deviation *is* a value returned by the fitters in HIPE, called either "standard deviation" or "error".

How does this change when you introduce weights to the fitting, i.e. data-point weights? Essentially, you need to make more assumptions. It is assumed in the HIPE fitter classes that one observation of data `D_i` with weight `M` is equivalent to `M` observations (all with the same value for the data: `D_i`) of weight 1. A weight equal to 0 entails that the point does not participate in the fitting. Where the weights originate from is entirely up to the user. Now you can play the game again: `chi^2 = SUM(w) * (D - F)^2 = SUM((D - F)/sigma)^2`. You can find a weighted solution for the parameters and the standard deviation in them. When all your models are OK, i.e. they are the "true" models that explain the data completely, then all residuals `(D - F)/sigma` are on average 1.0. In this case `chi^2` equals `N`, and the noise scale equals 1.0. Unfortunately, this last point is not true in HIPE: some unknown systematic effect is playing its part. If you are very concerned about working out the best fitting errors (the best standard deviation values for the parameters) you could calculate the noise scale, and with that the standard deviation values, from your spectral data directly. *Having said that*, we have carried out tests of the reliability of the fitting errors using the fitter classes in HIPE. We have fit a Gaussian+polynomial to a set of spectral lines of varying flux each 100 times over, and find that the RMS of the fitting results for the parameters (FWHM, central wavelength, and peak flux) for each line fit, is similar to the fitting errors returned for those parameters for that line, for a wide range of SNRs. This suggests that the errors returned by the fitter are a good estimate of the uncertainty of the model fit to the data, at least for this simple case.

Error and noise in compound models are treated just as for basic models. Think e.g. to fitting a line. It consists of a constant part plus a linear part. One can use a `PolynomialModel(1)`. Exactly the same can be achieved with `PolynomialModel(0) + PowerModel(1)`. All parameters are

fitted simultaneously by finding the minimum in `chi^2`. If you construct a model of a Gauss plus a constant: `F = GaussModel() + PolynomialModel(0)`, you can construct `chi^2` as before and do all the things described above.

7.29.2. Practical information for getting the fitting errors in HIPE

The errors for the fitted parameters are extracted as the "standard deviation" using methods appropriate for the different output results/products:

- When fitting multi-spectra products such as a cubes, the output products—the model, total model, and residual cubes—will each have an error dataset, but these will contain only zeros. The errors of the returned parameters (e.g. peak, width, wavelength for a Gaussian) are output along with the parameter values and can be found in the MultiFit_parms product created by the MultiFitter of the SFG (and the equivalent product if multi-fitting on the command-line, see Section 7.13), as "stddev".
- When fitting single spectra (with the SFG or on the command line), the output model and total model spectrum and residual may include a weight array but it will also contain only zero values. The error in the parameters are given as "stddev" in the output of the fitting, either in the FitResults tab, from the exported fitting results, or in the table of fitted results (see Section 7.13).
- When fitting a cube, one of the outputs is a `ParameterCube`. The "stddev" values therein are the parameter errors. In the worked example of Section 7.5.3 and in Section 7.28 we show how to extract these errors, so you can e.g. add the error layer to flux maps created from fitting results. The `ParameterCube` is an automatic product created when you work in the MultiFit tab of the SFG. If multifitting on the command line the `ParameterCube` can be extracted from "mf": see e.g. Section 7.28.
- When fitting via the SFG or on the command line, either for single or multiple spectra, one of the returned values is the Chi-squared. This is a goodness-of-fit parameter, with one value no matter how many models were fit to the spectrum/a. If fitting using the SFG on single spectra, the Chisq value is unfortunately inaccessible. If fitting single spectra on the command line you can use the following:

```
print sf.getChiSquared()
```

Example 7.62. Getting the Chi-squared on the command line

If fitting cubes via the command-line or the GUI, the Chisq value is added as a dataset to the output `ParameterCube`, and from there can be dealt with as you would the image or error datasets (see links given above).

7.29.3. Advanced practical information

As discussed previously, a fitting is only good as long as we can calculate with a certain confidence an estimation of the error of the fitting. To do that, there are several methods available in HIPE.

Using the `MonteCarloError` class for **linear models** (a fit that involves only instances of subclasses of the `LinearModel` interface). There is an example using the class in a common fitting scenario in the User's Reference Manual in *HCSS User's Reference Manual*. These are the models whose error can be estimated with this method:

- `BinomialModel` in *HCSS User's Reference Manual*
- `ChebyshevPolynomialModel` in *HCSS User's Reference Manual*
- `FreeShapeModel` in *HCSS User's Reference Manual*
- `FunnyModel`
- `HarmonicModel` in *HCSS User's Reference Manual*

- `LinearPyModel`
- `PolynomialModel` in *HCSS User's Reference Manual*

```
#
# Purpose: Simple fitting example
#
# Demonstration: 1. Linear Model
#   2. Parameter fit
#   3. Autoscaling, Chisq, Standard deviations
#   4. Confidence region for the fit
#
# Author: Do Kester
#
# This demo exercises some classes from main/herschel/ia/numeric/toolbox/fit
# the jython scripts are in fit/demo

from herschel.ia.numeric.all import *
from herschel.ia.gui.plot import *
import java
from java.awt import Color

# define plot colors
color = [Color.yellow,Color.green,Color.blue,Color.red,\
        Color.cyan,Color.magenta,Color.orange,Color.gray]

# Make a 3-degree polynomial
x = Double1d.range(21) / 2 - 5
ym = 1.2 + 0.5 * x + 0.13 * x * x + 0.01 * x * x * x
y = Double1d( 21 )

# Add gaussian noise to form data point.
noise = 0.4
seed  = 12345
rng = java.util.Random(seed)  # random number generator with a seed
for i in range(21): y[i] = ym[i] + noise * rng.nextGaussian()

F = DataFormatter()
# plot the data and the thruth
style=Style( line=Style.NONE, symbolShape=SymbolShape.FSQUARE, symbolSize=3 )
p = PlotXY( titleText="Simple Fit" )
p.addLayer( LayerXY( x, y, style=style, name="data" ) )
style.setColor( java.awt.Color.green )
p.addLayer( LayerXY( x, ym, style=style, name="model" ) )
style.setColor( java.awt.Color.black )

# define a 3rd order polynomial
deg = 3
poly = PolynomialModel( deg )
np = poly.getNumberOfParameters()
print "Polynomial degree ", np-1

# define a fitter. in this case a linear one as the model is linear
fitter = Fitter( x, poly )
param = fitter.fit( y )    # actually fit the data
print "Fit params      ", F.p( param )   # compare with ym.

# plot the fitted function on a 10 times finer grid.
xi = Double1d.range(201) / 20 - 5
yi = poly( xi )     # == poly.result( xi )
p.addLayer( LayerXY( xi, yi, style=style, name="fit" ) )
style.setLine( Style.SOLID )
style.setColor( java.awt.Color.red )

# show chisq and the scale of the noise: compare with noise = 0.14
chisq = fitter.getChiSquared()
print "    chisq       ", F.p( chisq )
print "    scale       ", F.p( fitter.autoScale() )

# print the standard deviations of the parameters.
stdev = fitter.getStandardDeviation()
print "    stdev       ", F.p( stdev )
```

```
# Calculate a confidence region. it should encompass most of the true model.
ye = fitter.monteCarloError( xi )
p.addLayer( LayerXY( xi, yi + ye, style=style, name="deviation" ) )
style.setColor( java.awt.Color.blue )
p.addLayer( LayerXY( xi, yi - ye, style=style, name="deviation" ) )

# Show the legend
p.legend.visible = 1
```

Example 7.63. Simple polynomial fitting with error calculation using MonteCarloError.

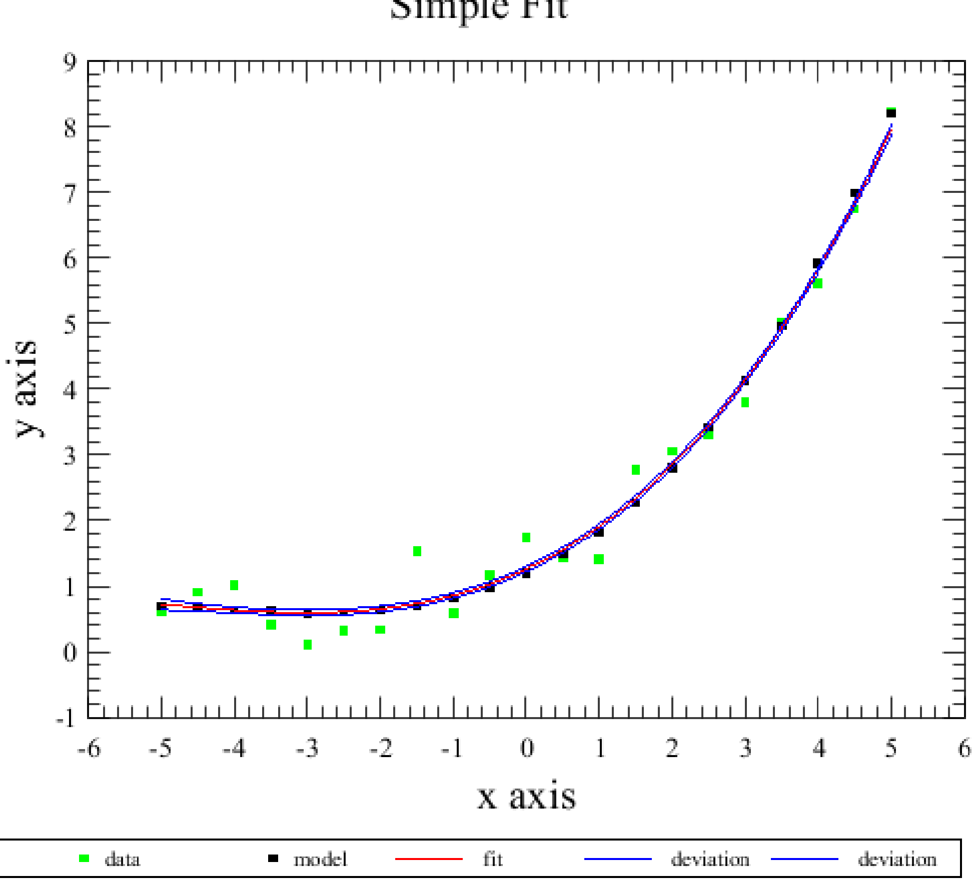


Figure 7.30. Fitting using a linear model with error calculation.

- PolySurfaceModel in *HCSS User's Reference Manual*
- PowerModel in *HCSS User's Reference Manual*
- SineAmpModel in *HCSS User's Reference Manual*
- SplinesModel in *HCSS User's Reference Manual*
- SurfaceSplinesModel in *HCSS User's Reference Manual*

A generic way of describing uncertainty for every model is using the next partial differentiate of the model with the current parameters. Many of the models which are subclasses of `NonLinearModel` can be described **with an equation** and some of them are **analytically differentiable**. These classes contain a method called `partial(double input, Double1d params)` to calculate the partial derivative for the data (`input`) for the variable.

Another method to estimate the confidence on the fit involves the use of a **Bayesian prior and evidence** which is retrieved along with the chisq distribution and the standard deviation to form a set of confidence indicators.

Also note that non-linear models can be converted to **mixed models** (linear plus non-linear parameters) which then allow the use of the `MonteCarloError` technique described for linear models (see below for an example using a sine mixed model).

Below there is a list of the models that can use one of these two methods:

- `ArctanModel` in *HCSS User's Reference Manual*

```
# Purpose: Non-linear fit
#
# Demonstration: 1. Arctan Model
#   2. LevenbergMarquardtFitter
#   3. Keeping parameters fixed
#
# Author: Do Kester

from herschel.ia.numeric import *
from herschel.ia.numeric.toolbox.fit import ArctanModel
from java.awt import Color
import java.util
from math import *

color = [Color.red,Color.green,Color.blue,Color.yellow,Color.cyan,Color.magenta]

# define some constants
N = 101
x0 = 95    # x position of arctanian

seed = 3456
rng = java.util.Random( seed )  # random number generator with a seed

# make data as a set of +1 or -1 values.
x = ( Double1d.range(N) - 45 ) / 15
y = Double1d( N ) - 1

for i in range( N ):
    x[i] = SINH( x[i] ) - 1.3
    if ( x[i] + 5 * rng.nextGaussian() > 0 ): y[i] = 1
    x[i] += 80

F = DataFormatter()

# plot the data and the model
style=Style( line=Style.NONE, symbolShape=SymbolShape.FSQUARE, symbolSize=3 )
p = PlotXY( titleText="" )
p.setXtitle( "Money" )
p.setYtitle( "Probability" )
p.addLayer( LayerXY( x, ( y + 1 ) / 2, style=style, name="data") )
style.setColor( java.awt.Color.black )

xax = p.getXaxis()
xax.setTitleText( "Money" )
yax = p.getYaxis()
yax.setTitleText( "Probability" )

# define the Gaussian Model
arctan = ArctanModel()
arctan += PolynomialModel(0)
print arctan
initial = Double1d( [0.5,1,80,-0.5] )
arctan.setParameters( initial )

#arctan.keepFixed( Int1d(1) )
# define the fitter: LevenbergMarquart
fitter = LevenbergMarquardtFitter( x, arctan )
```

```
# find the parameters
param = fitter.fit( y )
print "Parameters %s" % F.p( param )

# plot the fitted function
xi = x
yi = arctan( xi )
p.addLayer( LayerXY( xi, (yi+1)/2, name="arctan", style=style ) )
style.setColor( color[0] )
style.setLine( Style.SOLID )

# show the legend
p.legend.visible = 1

# standard deviation of the parameters
print "St Dev      %s" % F.p( fitter.getStandardDeviation() )
# amount of remaining (unexplained) noise scale = SQRT( chisq / N )
print "ChiSq       %s" % F.p( fitter.getChiSquared() )
print "Scale       %s" % F.p( fitter.autoScale() )
prior = Double1d( arctan.getNumberOfParameters() + 1 ) + 100
evid = fitter.getEvidence( prior )
print "Evidence    %s" % F.p( evid )

# done.
```

Example 7.64. Arctan fit with evidence estimation using a prior.

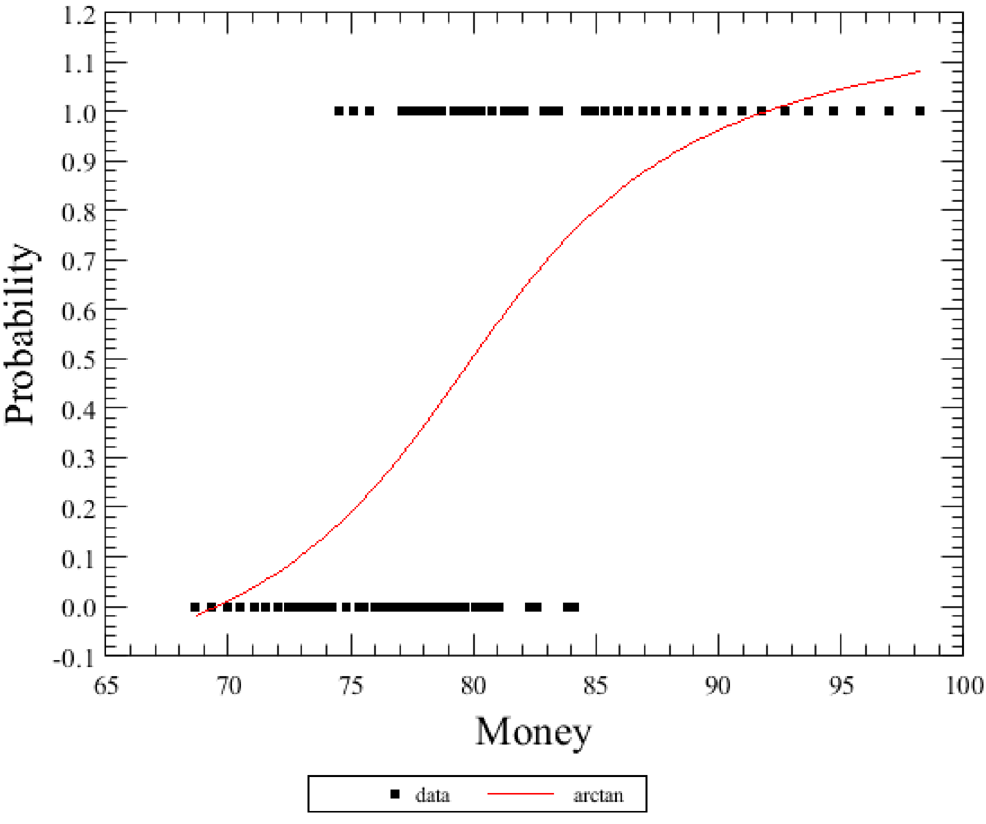


Figure 7.31. Fitting using a non-linear arctan model.

- `ExpModel` in *HCSS User's Reference Manual*
- `LorentzModel` in *HCSS User's Reference Manual*
- `PadeModel` in *HCSS User's Reference Manual*

```
# Purpose: Fitting a non-linear model
```

```
#
# Demonstration: 1. Pade model
#   2. Limits on some parameter ranges
#   3. Mixed models
#   4. MonteCarlo confidence regions
#
# Author: Do Kester

from herschel.ia.numeric.all import *
from herschel.ia.numeric.toolbox.fit import PadeModel
from java.awt import Color
import java
import time

color = [Color.red,Color.green,Color.blue,Color.yellow,Color.cyan,Color.magenta]

F = DataFormatter()

npt = 100
x = Double1d.range( npt )
y = SIN( 0.05 * x )
rng = java.util.Random( 12345 )
for i in range(npt): y[i] += 0.2 * rng.nextGaussian()

pade = PadeModel( 2, 1 )
npar = pade.getNumberOfParameters()
print npar
# define the fitter
fitter = LevenbergMarquardtFitter( x, pade )

print F.p( fitter.fit( y ), npar )

print F.p( fitter.getStandardDeviation(), npar )

print fitter.autoScale()
pade.setPriorRange( Double1d( npar + 1 ) + 10000.0 )
print fitter.getChiSquared()
print fitter.getEvidence();

style=Style( line=Style.NONE, symbolShape=SymbolShape.FSQUARE, symbolSize=3 )
p = PlotXY( titleText="Pade model" )
p.addLayer( LayerXY( x, y, style=style, name="data") )
style.setColor( java.awt.Color.green )
p.addLayer( LayerXY( x, pade( x ), style=style, name="fit(2,1)" ) )
style.setColor( java.awt.Color.black )
style.setLine( Style.SOLID )

pade = PadeModel( 3, 1 )
npar = pade.getNumberOfParameters()

# define the fitter
fitter = LevenbergMarquardtFitter( x, pade )

print fitter.fit( y )

print F.p( fitter.getStandardDeviation() )

print fitter.autoScale()
pade.setPriorRange( Double1d( npar + 1 ) + 10000.0 )
print fitter.getChiSquared()
print fitter.getEvidence();

p.addLayer( LayerXY( x, pade( x ), style=style, name="fit(3,1)" ) )
style.setColor( java.awt.Color.red )
style.setLine( Style.SOLID )

p.legend.visible = 1
```

Example 7.65. Fitting with PadeModel and evidence estimation using a prior range.

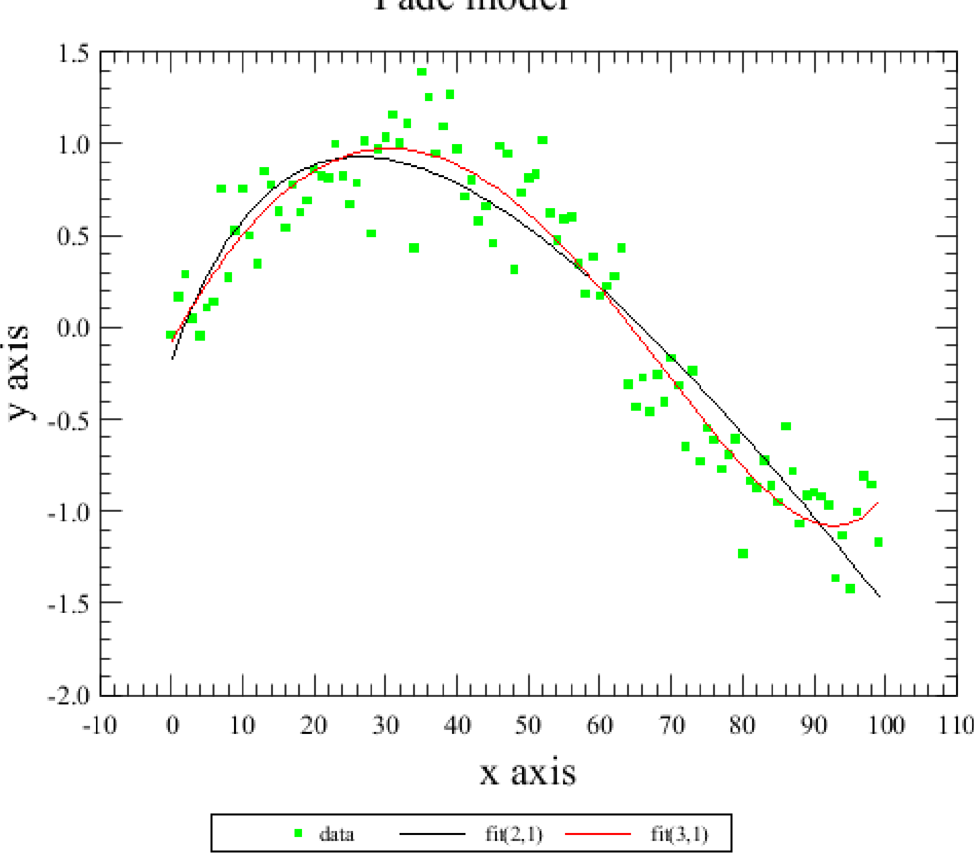


Figure 7.32. Fitting using a non-linear pade model.

- PowerLawModel in *HCSS User's Reference Manual*
- SincGaussModel in *HCSS User's Reference Manual*
- SincModel in *HCSS User's Reference Manual*
- SineModel in *HCSS User's Reference Manual*

All non-linear models allow one to convert the model to a mixed model using `setMixedModel`. For an example of making the `SineModel` mixed, which allows the use of `MonteCarloError` with it, see below:

```
# Purpose: Fitting a non-linear model
#
# Demonstration: 1. AmoebaFitter with annealing.
#    2. Limits on some parameter ranges
#    3. Mixed models
#    4. MonteCarlo confidence regions
#
# Author: Do Kester

from herschel.ia.numeric.all import *
from java.awt import Color
import java
import time

color = [Color.red,Color.green,Color.blue,Color.yellow,Color.cyan,Color.magenta]

F = DataFormatter()
```

```
# make data with an underlying sinusoidal model.
x = Double1d.range(21) / 2.0 - 5
y = Double1d( 21 )

ym = 0.6 * COS( 6.28 * 0.44 * x ) + 0.5 * SIN( 6.28 * 0.44 * x )
rng = java.util.Random( 12345 )
for i in range(21): y[i] = ym[i] + 0.2 * rng.nextGaussian()

# plot the data and the model
style=Style( line=Style.NONE, symbolShape=SymbolShape.FSQUARE, symbolSize=3 )
p = PlotXY( titleText="Sinusoidal model" )
p.addLayer( LayerXY( x, y, style=style, name="data") )
style.setColor( java.awt.Color.green )
p.addLayer( LayerXY( x, ym, style=style, name="model" ) )
style.setColor( java.awt.Color.black )

# define the model and set limits (there is a good chance that the fitter
#  wont find the parameters when you leave them unrestricted)
sine = SineModel()
sine.setLimits( Double1d( [0,0,0]), Double1d([1,10,10]) )

# define the fitter: amoeba
fitter = AmoebaFitter( x, sine )

# Initialize the simplex; avoid awkward points like [0,0,0]
fitter.setSimplex( Double1d([0.1,1,1]), Double1d([0.2,1,1]) )

# Set the temperature to make it an annealing simplex. this is needed to
# get out of local minima, of which there are a lot in these kind of models.
fitter.setTemperatureSteps( 100 )
fitter.setTemperature( 20 )

# find the parameters
starttime = time.time()
param = fitter.fit( y )
stoptime = time.time()
print "Elapsed time %8.2f sec" % ( stoptime - starttime )
print "Parameters %8.2f %8.2f %8.2f" % (sine[0], sine[1], sine[2])

# find the standard deviations of the parameters
stdev = fitter.getStandardDeviation();
print "Stand Devs %8.2f %8.2f %8.2f" % (stdev[0], stdev[1], stdev[2])

print F.p( fitter.getHessian() )

# apply the model on a much finer grid, just for plotting purposes
xx = Double1d.range(201) / 20 - 5
yy = sine( xx )

# plot the model in red over the data etc.
p.addLayer( LayerXY( xx, yy, name="fit", style=style ) )
style.setLine( Style.SOLID )
style.setColor( color[0] )

# Calculate a confidence region. it should encompass most of the true model.
ye = fitter.monteCarloError( xx )
p.addLayer( LayerXY( xx, yy + ye, name="deviation", style=style ) )
style.setColor( color[2] )
p.addLayer( LayerXY( xx, yy - ye, name="deviation", style=style ) )


# If this was not very satisfying, or it took way too long, the rest assured
# that it is not the `best' way to fit a sine to a data set. It is better
# to use a so-called mixed model, taking advantage that 2 of the 3 parameters
# are in fact linear.

# plot the data and the model
style=Style( line=Style.NONE, symbolShape=SymbolShape.FSQUARE, symbolSize=3 )
p = PlotXY( titleText="Sinusoidal mixed model" )
p.addLayer( LayerXY( x, y, style=style, name="data") )
style.setColor( java.awt.Color.green )
p.addLayer( LayerXY( x, ym, style=style, name="model" ) )
```

```
style.setColor( java.awt.Color.black )

#sinem = SineMixedModel()   # alternate way
sinem = SineModel()
sinem.setMixedModel( Int1d( [1,2] ) )
sinem.setLimits( Double1d( [0,0,0]), Double1d([1,0,0]) )

# define the fitter: amoeba
mfitter = AmoebaFitter( x, sinem )

mfitter.setSimplex( Double1d([0.1,1,1]), Double1d([0.2,1,1]) )
mfitter.setTemperature( 20 )
#mfitter.setMaxIterations( 1 )

# find the parameters
starttime = time.time()
param = mfitter.fit( y )
stoptime = time.time()
print "Elapsed time %8.2f sec" % ( stoptime - starttime )
print "Parameters %8.2f %8.2f %8.2f" % (sinem[0], sinem[1], sinem[2])


# find the standard deviations of the parameters
stdev = mfitter.getStandardDeviation();
print "Stand Devs %8.2f %8.2f %8.2f" % (stdev[0], stdev[1], stdev[2])

print F.p( mfitter.getHessian() )

# apply the model on a much finer grid, just for plotting purposes
xx = Double1d.range(201) / 20 - 5
yy = sinem( xx )

# plot the model in red over the data etc.
p.addLayer( LayerXY( xx, yy, name="mixed-fit", style=style ) )
style.setColor( color[5] )
style.setLine( Style.SOLID )

# Calculate a confidence region. it should encompass most of the true model.
ye = mfitter.monteCarloError( xx )
p.addLayer( LayerXY( xx, yy + ye, name="deviation", style=style ) )
style.setColor( color[4] )
p.addLayer( LayerXY( xx, yy - ye, name="deviation", style=style ) )

# Show the legend
p.legend.visible = 1
```

Example 7.66. Fitting data with a non-linear SineModel, using a Hessian matrix as confidence; then fitting with a mixed SineModel and estimating error with MonteCarloError.

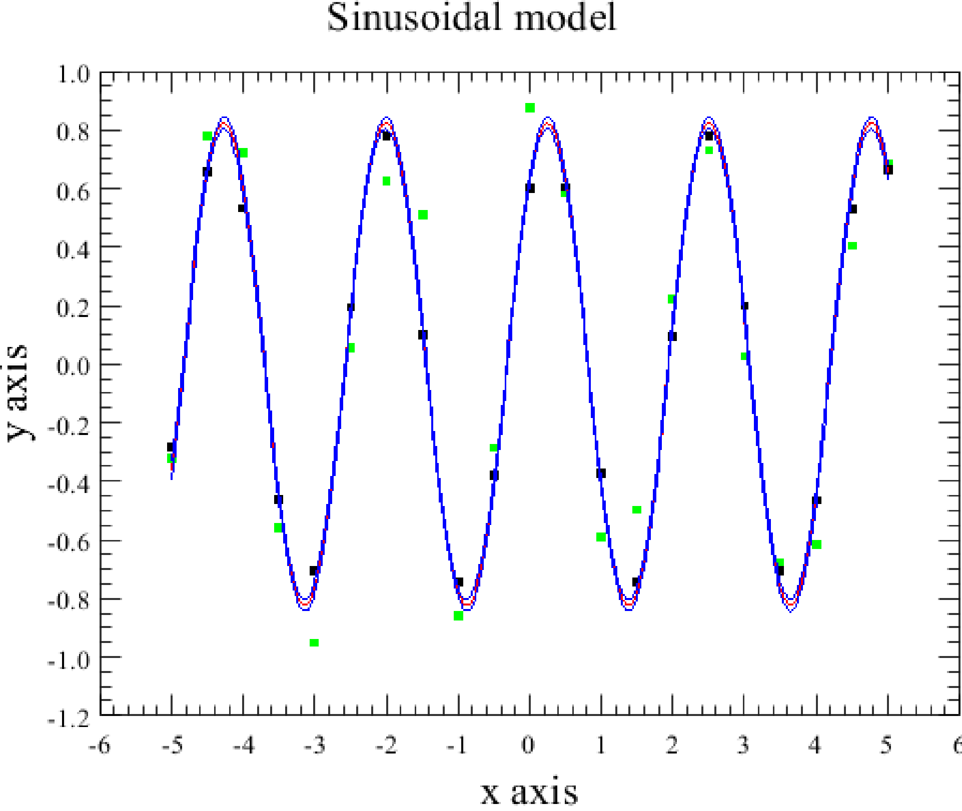


Figure 7.33. Fitting using a non-linear sine model.

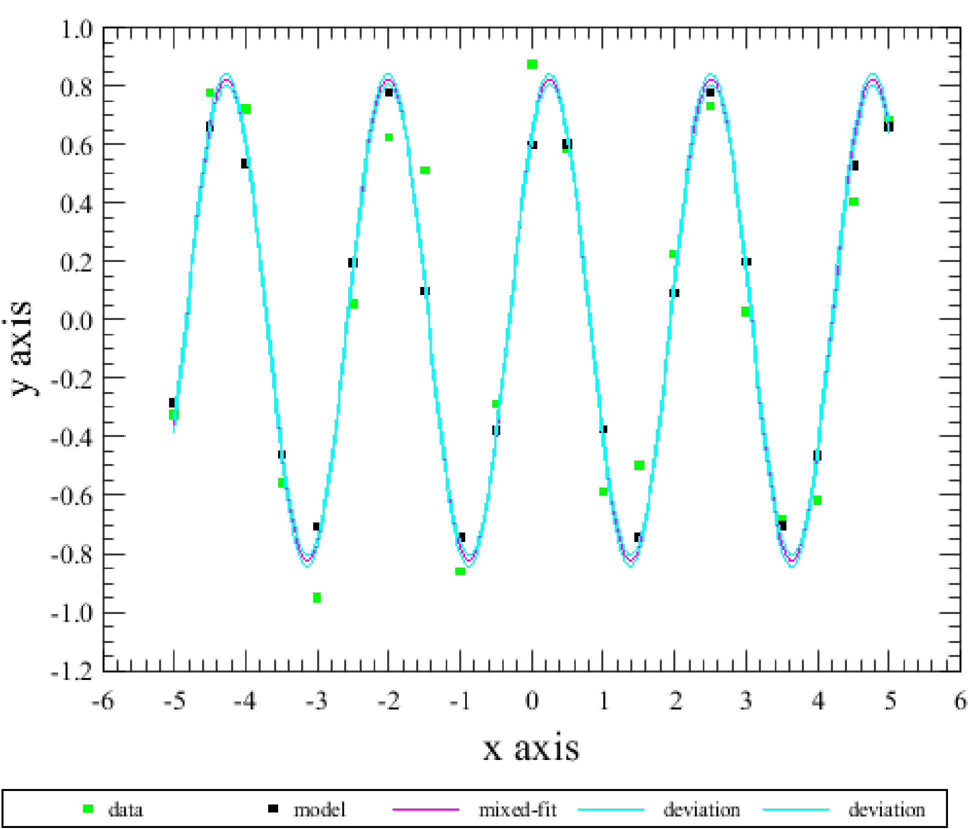


Figure 7.34. Fitting using a mixed sine model (linear and non-linear).

Note

SkewGaussModel_ in *HCSS User's Reference Manual* has an equation but the class doesn't include the `partial` method. Since HIPE does not do symbolic differentiation, the partial derivative must be calculated outside of HIPE and then translated to a Jython expression in order to calculate the error as part of a script.

Using the Monte Carlo method to estimate (brute-force) the statistical indicators of the error. This procedure is indicated for models which are **non-linear and don't have an equation**. This is an abstraction of the procedure:

1. Generate a sample of about 10.000 values for each parameter, assuming Gaussian distribution. Parameters can be correlated which complicates the generation of the sample.

2. Fit the data with every set of parameters.

3. An estimation of the uncertainty could be the statistical indicators for the distribution of the results.

Note

As a brute-force method it has its shortcomings and the best approach would be to develop a Bayesian approach. This would not require Gaussian assumptions for the parameters and also the cross-correlations could be tackled in some way. However this is out of the scope of this documentation.

If you are willing to use a Bayesian approach, note that the error of these **non-linear models without equation** can be estimated using a prior, as well. See examples for some models below.

- ComboModel in *HCSS User's Reference Manual*: Whenever any or the models are non-linear or their combination is non-linear.
- Gauss2DModel in *HCSS User's Reference Manual*
- Gauss2DRotModel in *HCSS User's Reference Manual*
- GaussModel in *HCSS User's Reference Manual*

The example below creates a combined model of Gaussian, Polynomial and Sine models for fitting the data. For estimating the error and confidence, it calculates evidence based on a prior, standard deviations, Hessian matrix and chisq distribution.

```
# Purpose: Non-linear fit
#
# Demonstration: 1. Gauss Model
#   2. LevenbergMarquardtFitter
#   3. Compound Model (Gaussian + polynomial)
#   4. Compound Model (Gaussian + polynomial + sine)
#   5. AmoebaFitter with annealing
#
# Author: Do Kester

from herschel.ia.numeric import *
from java.awt import Color
import java.util
from math import *

color = [Color.red,Color.green,Color.blue,Color.yellow,Color.cyan,Color.magenta]

# define some constants
N = 201
x0 = 0.7   # x position of gaussian
a0 = 10.0   # amplitude of gaussian
s0 = 0.4   # width
b0 = 1.0   # offset of background
b1 = 0.2   # slope of background
c0 = 0.4
c1 = 8.0
c2 = 0.0

# make data od a Gaussian + straight line plus a small sinusoidal ripple.
x = Double1d.range(N) / 25.0 - 2
ym = a0 * EXP( -0.5 * SQUARE( (x - x0) / s0 ) ) + b0 + b1 * x + \
c0 * SIN( c1 * x + c2 )

y = Double1d( N )
seed = 3456
rng = java.util.Random( seed )  # random number generator with a seed
for i in range( N ): y[i] = ym[i] + 0.2 * rng.nextGaussian()

F = DataFormatter()

# plot the data and the model
style=Style( line=Style.NONE, symbolShape=SymbolShape.FSQUARE, symbolSize=3 )
p = PlotXY( titleText="" )
p.addLayer( LayerXY( x, y, style=style, name="data") )
style.setColor( java.awt.Color.black )
#p.addLayer( LayerXY( x, ym, style=style, name="model" ) )
#style.setColor( java.awt.Color.black )
#style.setLine( Style.SOLID )
annotation = Annotation( 5.0, 11.0, "Evidence" )
p.addAnnotation( annotation )

# define the Gaussian Model
gauss = GaussModel()
print gauss

# define the fitter: LevenbergMarquart
fitter = LevenbergMarquardtFitter( x, gauss )
```

```
# find the parameters
param = fitter.fit( y )
print "Parameters %s" % F.p( param )

# plot the fitted function on a 10 times finer grid.
#xi = Double1d.range(201) / 50 - 2
xi = x
yi = gauss( xi )
p.addLayer( LayerXY( xi, yi, name="Gauss", style=style ) )
style.setColor( color[0] )
style.setLine( Style.SOLID )

print "Scale      %s" % F.p( fitter.autoScale() )
prior = Double1d( gauss.getNumberOfParameters() + 1 ) + 100
evid = fitter.getEvidence( prior )
print "Evidence   %s" % F.p( evid )
annotation = Annotation( 5.0, 10.0, F.p( evid ) )
annotation.setColor( color[0] )
p.addAnnotation( annotation )


# add a straight line to the GaussModel (polynomial of order 1)
#gauss += PolynomialModel( 1 )  # alternate method equivalent to next
gauss.addModel( PolynomialModel( 1 ) )
print gauss

# define the fitter: LevenbergMarquart
fitter = LevenbergMarquardtFitter( x, gauss )

# find the parameters
param = fitter.fit( y )
print "Parameters %s" % F.p( param )

# plot the fitted function on a 10 times finer grid.
#xi = Double1d.range(201) / 50 - 2
xi = x
yi = gauss( xi )
p.addLayer( LayerXY( xi, yi, name="Gauss + polynome", style=style ) )
style.setColor( color[1] )

print "Scale      %s" % F.p( fitter.autoScale() )
prior = Double1d( gauss.getNumberOfParameters() + 1 ) + 100
evid = fitter.getEvidence( prior )
print "Evidence   %s" % F.p( evid )
annotation = Annotation( 5.0, 9.0, F.p( evid ) )
annotation.setColor( color[1] )
p.addAnnotation( annotation )


gauss += SineModel( )
print gauss

#gauss.setMixedModel( Int1d( [0,3,4,6,7] ) )
gauss.setLimits( Double1d( [0,0,0,0,0,0,0,0] ), Double1d( [0,0,0,0,0,20,0,0] ))

mixfit = AmoebaFitter( x, gauss )

initpar = param
initpar = initpar.append( 0.1 )
initpar = initpar.append( 1.0 )
initpar = initpar.append( 1.0 )
# Initialize the simplex; avoid awkward points like [0,0,0]
mixfit.setSimplex( initpar, Double1d([0.01,0.01,0.01,0.01,0.01,0.2,1,1]) )

# Set the temperature to make it an annealing simplex. this is needed to
# get out of local minima, of which there are a lot in these kind of models.
mixfit.setTemperature( 2 )

# find the parameters
par1 = mixfit.fit( y )      # it may take a while ...
print "Parameters %s" % F.p( par1, gauss.getNumberOfParameters() )
```

```
print "Parameters %s" % F.p( mixfit.getParameters(),
 gauss.getNumberOfParameters() )

print mixfit.getNumberOfTransforms()

# plot the fitted function on a 10 times finer grid.
#xi = Double1d.range(201) / 50 - 2
xi = x
yi = gauss( xi )
labl = "Gauss + poly + sine"
p.addLayer( LayerXY( xi, yi, name=labl, style=style ) )
style.setColor( color[2] )

print "Scale      %s" % F.p( mixfit.autoScale() )
prior = Double1d( gauss.getNumberOfParameters() + 1 ) + 100
evid = mixfit.getEvidence( prior )
print "Evidence   %s" % F.p( evid )
annotation = Annotation( 5.0, 8.0, F.p( evid ) )
annotation.setColor( color[2] )
p.addAnnotation( annotation )

# show the legend
p.legend.visible = 1

# find the standard deviations of the parameters
std1 = mixfit.getStandardDeviation();
print "Stand Devs %s" % F.p( std1 )

print "ChiSq      %s" % F.p( mixfit.getChiSquared() )
print "Hessian"
print F.p( mixfit.getHessian() )

# done
```

Example 7.67. Fitting data with a combined model of Gaussian, Polynomial and Sine models.

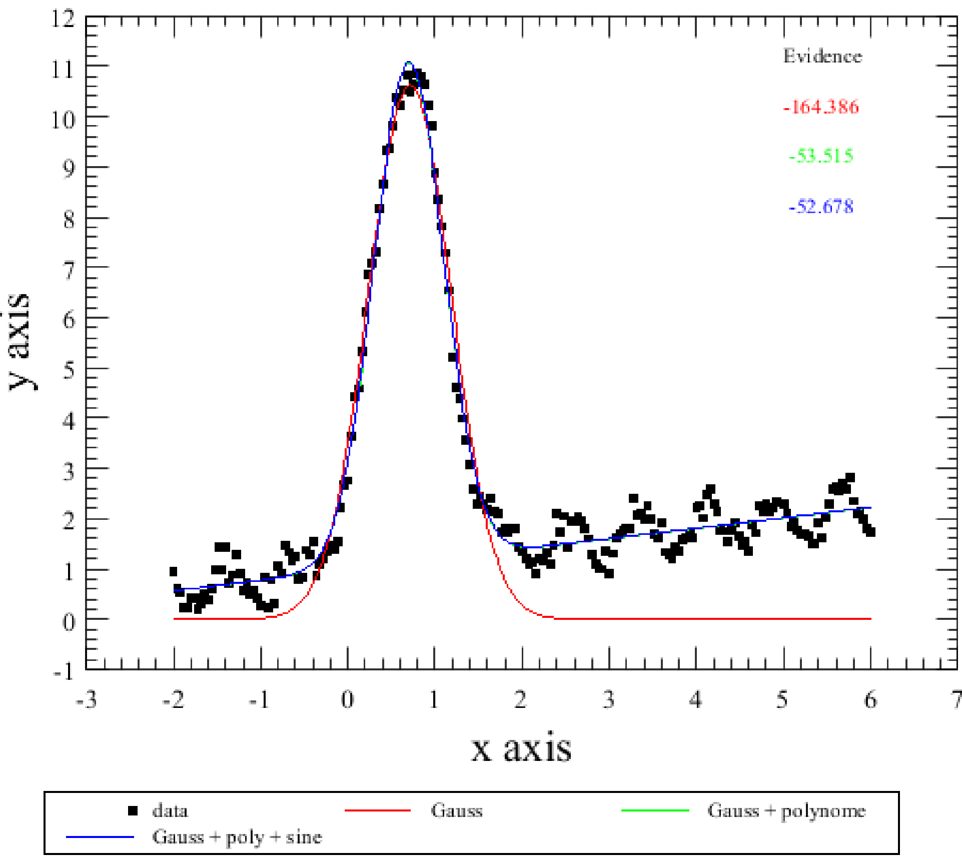


Figure 7.35. Fitting using combined gauss, polynomial and sine models.

- `GaussNoPartial`
- `Kernel2dModel` in *HCSS User's Reference Manual*
- `KernelModel` in *HCSS User's Reference Manual*
- `NonLinearPyModel`: This depends on the implementation.
- `VoigtModel`

Using the Monte Carlo method to estimate (brute-force) the statistical indicators of the error. This procedure is indicated for models which are **non-linear and don't have an equation**. This is an abstraction of the procedure:

1. Generate a sample of about 10.000 values for each parameter, assuming Gaussian distribution. Parameters can be correlated which complicates the generation of the sample.
2. Fit the data with every set of parameters.
3. An estimation of the uncertainty could be the statistical indicators for the distribution of the results.

Note

As a brute-force method it has its shortcomings and the best approach would be to develop a Bayesian approach. This would not require Gaussian assumptions for the parameters and also the cross-correlations could be tackled in some way. However this is out of the scope of this documentation.

If you are willing to use a Bayesian approach, note that the error of these **non-linear models without equation** can be estimated using a prior, as well. See examples for some models below.

- ComboModel in *HCSS User's Reference Manual*: Whenever any or the models are non-linear or their combination is non-linear.
- Gauss2DModel in *HCSS User's Reference Manual*
- Gauss2DRotModel in *HCSS User's Reference Manual*
- GaussModel in *HCSS User's Reference Manual*

The example below creates a combined model of Gaussian, Polynomial and Sine models for fitting the data. For estimating the error and confidence, it calculates evidence based on a prior, standard deviations, Hessian matrix and chisq distribution.

```
# Purpose: Non-linear fit
#
# Demonstration: 1. Gauss Model
#    2. LevenbergMarquardtFitter
#    3. Compound Model (Gaussian + polynomial)
#    4. Compound Model (Gaussian + polynomial + sine)
#    5. AmoebaFitter with annealing
#
# Author: Do Kester

from herschel.ia.numeric import *
from java.awt import Color
import java.util
from math import *

color = [Color.red,Color.green,Color.blue,Color.yellow,Color.cyan,Color.magenta]

# define some constants
N = 201
x0 = 0.7    # x position of gaussian
a0 = 10.0    # amplitude of gaussian
s0 = 0.4    # width
b0 = 1.0    # offset of background
b1 = 0.2    # slope of background
c0 = 0.4
c1 = 8.0
c2 = 0.0

# make data od a Gaussian + straight line plus a small sinusoidal ripple.
x = Double1d.range(N) / 25.0 - 2
ym = a0 * EXP( -0.5 * SQUARE( (x - x0) / s0 ) ) + b0 + b1 * x + \
c0 * SIN( c1 * x + c2 )

y = Double1d( N )
seed = 3456
rng = java.util.Random( seed )  # random number generator with a seed
for i in range( N ): y[i] = ym[i] + 0.2 * rng.nextGaussian()

F = DataFormatter()

# plot the data and the model
style=Style( line=Style.NONE, symbolShape=SymbolShape.FSQUARE, symbolSize=3 )
p = PlotXY( titleText="" )
p.addLayer( LayerXY( x, y, style=style, name="data") )
style.setColor( java.awt.Color.black )
#p.addLayer( LayerXY( x, ym, style=style, name="model" ) )
#style.setColor( java.awt.Color.black )
#style.setLine( Style.SOLID )
annotation = Annotation( 5.0, 11.0, "Evidence" )
p.addAnnotation( annotation )

# define the Gaussian Model
gauss = GaussModel()
print gauss

# define the fitter: LevenbergMarquart
fitter = LevenbergMarquardtFitter( x, gauss )
```

```
# find the parameters
param = fitter.fit( y )
print "Parameters %s" % F.p( param )

# plot the fitted function on a 10 times finer grid.
#xi = Double1d.range(201) / 50 - 2
xi = x
yi = gauss( xi )
p.addLayer( LayerXY( xi, yi, name="Gauss", style=style ) )
style.setColor( color[0] )
style.setLine( Style.SOLID )

print "Scale      %s" % F.p( fitter.autoScale() )
prior = Double1d( gauss.getNumberOfParameters() + 1 ) + 100
evid = fitter.getEvidence( prior )
print "Evidence   %s" % F.p( evid )
annotation = Annotation( 5.0, 10.0, F.p( evid ) )
annotation.setColor( color[0] )
p.addAnnotation( annotation )


# add a straight line to the GaussModel (polynomial of order 1)
#gauss += PolynomialModel( 1 )  # alternate method equivalent to next
gauss.addModel( PolynomialModel( 1 ) )
print gauss

# define the fitter: LevenbergMarquart
fitter = LevenbergMarquardtFitter( x, gauss )

# find the parameters
param = fitter.fit( y )
print "Parameters %s" % F.p( param )

# plot the fitted function on a 10 times finer grid.
#xi = Double1d.range(201) / 50 - 2
xi = x
yi = gauss( xi )
p.addLayer( LayerXY( xi, yi, name="Gauss + polynome", style=style ) )
style.setColor( color[1] )

print "Scale      %s" % F.p( fitter.autoScale() )
prior = Double1d( gauss.getNumberOfParameters() + 1 ) + 100
evid = fitter.getEvidence( prior )
print "Evidence   %s" % F.p( evid )
annotation = Annotation( 5.0, 9.0, F.p( evid ) )
annotation.setColor( color[1] )
p.addAnnotation( annotation )


gauss += SineModel( )
print gauss

#gauss.setMixedModel( Int1d( [0,3,4,6,7] ) )
gauss.setLimits( Double1d( [0,0,0,0,0,0,0,0] ), Double1d( [0,0,0,0,0,20,0,0] ))

mixfit = AmoebaFitter( x, gauss )

initpar = param
initpar = initpar.append( 0.1 )
initpar = initpar.append( 1.0 )
initpar = initpar.append( 1.0 )
# Initialize the simplex; avoid awkward points like [0,0,0]
mixfit.setSimplex( initpar, Double1d([0.01,0.01,0.01,0.01,0.01,0.2,1,1]) )

# Set the temperature to make it an annealing simplex. this is needed to
# get out of local minima, of which there are a lot in these kind of models.
mixfit.setTemperature( 2 )

# find the parameters
par1 = mixfit.fit( y )      # it may take a while ...
print "Parameters %s" % F.p( par1, gauss.getNumberOfParameters() )
```

```
print "Parameters %s" % F.p( mixfit.getParameters(),
 gauss.getNumberOfParameters() )

print mixfit.getNumberOfTransforms()

# plot the fitted function on a 10 times finer grid.
#xi = Double1d.range(201) / 50 - 2
xi = x
yi = gauss( xi )
labl = "Gauss + poly + sine"
p.addLayer( LayerXY( xi, yi, name=labl, style=style ) )
style.setColor( color[2] )

print "Scale      %s" % F.p( mixfit.autoScale() )
prior = Double1d( gauss.getNumberOfParameters() + 1 ) + 100
evid = mixfit.getEvidence( prior )
print "Evidence   %s" % F.p( evid )
annotation = Annotation( 5.0, 8.0, F.p( evid ) )
annotation.setColor( color[2] )
p.addAnnotation( annotation )

# show the legend
p.legend.visible = 1

# find the standard deviations of the parameters
std1 = mixfit.getStandardDeviation();
print "Stand Devs %s" % F.p( std1 )

print "ChiSq      %s" % F.p( mixfit.getChiSquared() )
print "Hessian"
print F.p( mixfit.getHessian() )

# done
```

Example 7.68. Fitting data with a combined model of Gaussian, Polynomial and Sine models.

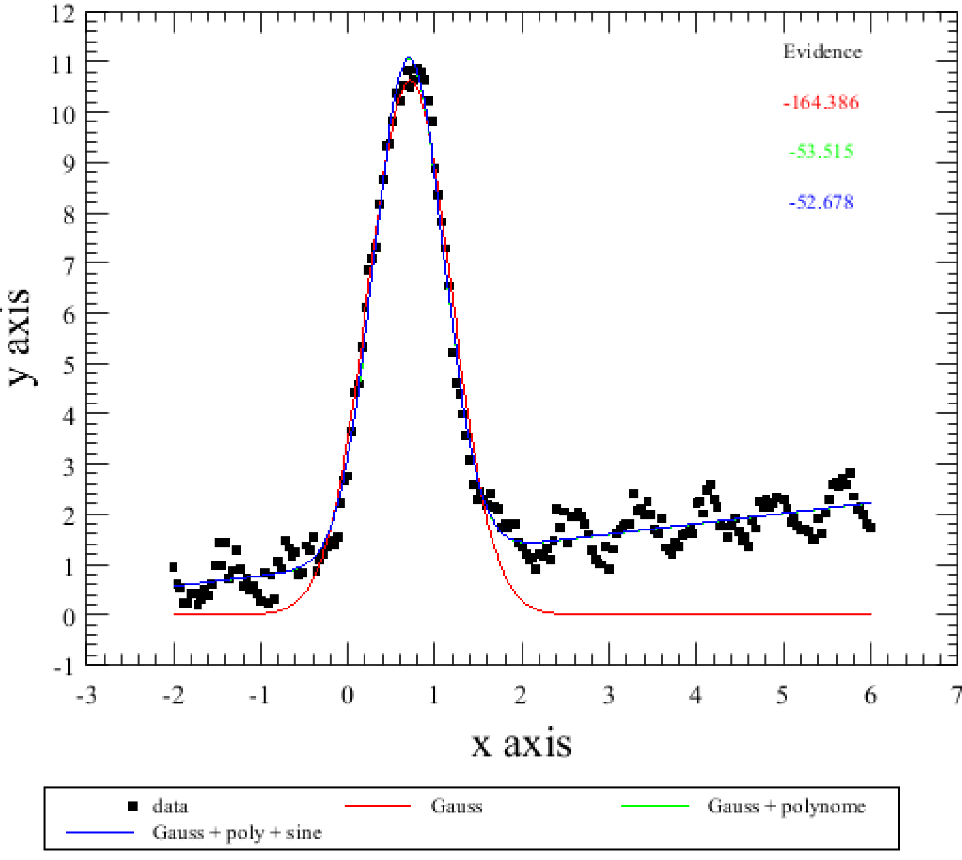


Figure 7.36. Fitting using combined gauss, polynomial and sine models.

- `GaussNoPartial`
- `Kernel2dModel` in *HCSS User's Reference Manual*
- `KernelModel` in *HCSS User's Reference Manual*
- `NonLinearPyModel`: This depends on the implementation.
- `VoigtModel`

7.30. Troubleshooting and limitations of the fitter

Although the theory of model fitting is quite straightforward, the implementation can be cumbersome. This is mostly due to the fact that we have limited precision computers and limited amounts of time. Even though all the computation is done in 8-byte doubles, limited resources and sometimes the obnoxious shape of the χ^2-landscape can make life for the fitter difficult. Another common bear on the road is the (near-)degeneracy in the model with respect to the data. It sometimes looks more like an art than a craft.

For diagnostic (and debugging) purposes in the iterative fitters (`LMFitter` and `AmoebaFitter`) there is the `setVerbose(int n)` option, which prints the parameters every n-th iteration. And also the `setPlotter` method where an `IterationPlotter` can be attached to the Fitter, which makes a plot of the data and the results at each k-th iteration. An example of a IterationPlotter can be found in `ia.toolbox.fit` . The reason why it is not incorporated in the Fitter or even present

in the package is because it would create a circular dependency between the fit package and the plot package. Users can make their own `IterationPlotter` by implementing the interface `IterationPlotable`.

Here are some guidelines that might help to get useful results.

- The independent variable(s) `x` should be roughly of order 1. Mostly the fitter obtains solutions by manipulating a matrix consisting of a direct product of the partial derivatives of the model to each of its parameters. If the elements of this matrix wildly vary in size, loss of precision is quickly attained. E.g. a polynomial model of order 3 with an independent variable which has values from 1 to 100, will have a matrix with values ranging between 1 and 1012. Mathematically this is all OK, computationally it is a nightmare. To inspect this (Hessian) matrix use `getHessian()`. The Fitter software can not and does not scale its inputs in any way. It takes it all at face value. It is up to the user to present the Fitters with usable data.
- The dependent variable `y` has fewer constraints. Still there is a silent assumption in the algorithms that the amount of noise in the data is of the order 1. This is only of importance in the stopping criterion of iterative fitters. There is no way to do it right in all imaginable cases. To redress this condition you can either use weights (inverse of the noise squared) or use the `setTolerance()` method to adapt the stopping criterion to your problem or scale the dependent variable such that the noise level attains a more useful value. Check whether χ^2, `getChiSquared()`, is of the order of the number of datapoints.
- Sometimes the model is degenerated, meaning that 2 (or more) of its parameters are essentially measuring the same thing. Trying to fit data using Fitter to such a model results in a singular matrix. The `SingularValueDecompositionFitter` has less problems as it evenly distributes the value over the degenerated parameters. Try `hasDegeneracy()` to check for this condition. In general it is better to use models which are not degenerated.
- When a non-linear fitter searches for a minimum, it might happen that for almost all values of the parameters χ^2 does not have a gradient. That is when the parameters move away from the sought minimum, the χ^2 does not or hardly change. Mostly this can be the case with models of periodic functions or when you are far from the global minimum. Only at a small area in the parameters space around the minimum there is a gradient to move along. If you can in some other way localise the area where your parameters should be found, feed them to the system as initial parameters. Otherwise you have to use the `AmoebaFitter` in the annealing mode or even do exhaustive search. Both strategies take a lot more time.
- A similar and sometimes simultaneous condition happens when the landscape is multimodal, i.e. it has more than one minimum. Of course only one minimum is the lowest, the global minimum. And that one is the one we want to find. In general fitters don't have a global view on the χ^2 landscape. They fall for the first minimum they encounter. The same advice as on the previous item.
- The model is (almost) degenerate under the data it is presented to. In general this signals that the model is not the best one for the data at hand. Use a simpler model.
- The size of the Simplex in the `AmoebaFitter` is by default 1. When your `x`-data is very much larger (or smaller) than 1 and you set your `startValues` accordingly, you might want to adapt the size of the Simplex too. In extreme cases the 1 vanishes in the precision of the `startValues` and the simplex is frozen in some dimension(s). Use `setSimplex(startValues, sizeSimplex)`.
- χ^2 is zero. This can only happen when the data fit the model perfectly, either because the parameters of the models have been fixes or occasionally it could be due to digitisation of the data: the data is only perfect after they have been digitised. This digitisation gives each data point a noise of `1/sqrt(12)` , the standard deviation of a uniform distribution. The actual position of each datapoint can be anywhere between two digitisation levels (presumably at distance 1). This noise can be added to the internal χ^2 with `setChiSquared(N/12)`.

- When you think that the standard deviations on the parameters are ridiculously large, please calculate the confidence region on the model fit (use `fitter.monteCarloError()`). This 1-σ confidence interval is obtained directly from the standard deviation and its covariance matrix. Due to strong covariance between the parameters, the standard deviations can be large while the confidence region is still acceptable. The confidence region is actually is better indicator for how well the model performs than the standard deviations on the parameters.
- The standard deviations of the parameters are calculated as the square root of chisq multiplied by the diagonal elements of the inverse Hessian matrix at the chisq minimum, divided by the degrees of freedom. The chisq landscape at that point is a stationary minimum, curving up in all directions. The shape of the minimum is approximated by the inverse Hessian (also known as Covariance Matrix), upto the 2nd power in the Taylor expansion of the chisq function. When you have set limits on the parameters and the unconstraint minimum of chisq would be outside the limits, then the constraint minimum of chisq is at one of the limits. This constraint minimum is not a stationary point any more so the assumptions above about a neat valley in the chisq landscape, curving up to all sides are not met. What the resulting calculations will give is anyone's guess.

Chapter 8. Unit Conversion

8.1. Units in HIPE

In HIPE you can assign units to every product. Doing so, you can perform conversions and calculations that retain a meaningful unit in their results. The core of this functionality resides in the `unit` namespace and the `ConvertUnitsTask`. Even if you are doing the conversion manually, you need the classes and constants contained within `herschel.share.unit`. A simple example of manual conversion can be found in the fourth step of this worked example Section 7.5.3. Below you can find the most relevant lines of that example, commented, reordered and extracted for the sake of explanation. The variable `Params` is obtained from the `MultiFitter` results. It contains a fitted product:

```
Params = mf.getProduct()
```

Example 8.1. Getting the product stored in the results of a MultiFitting.

An empty matrix with the dimensions of the cube is created to contain the frequency values of the fitted product:

```
freq=Double2d(cube.dimensions[1],cube.dimensions[2])
```

Example 8.2. Creating an empty matrix that can hold the frequency values of the fitted products.

Now it is possible to fill the matrix with data from the fit:

```
for row in range (cube.dimensions[1]):
   for col in range (cube.dimensions[2]):
      freq[row,col]=Params[name]["Parameters"].data[1]
```

Example 8.3. Filling the matrix with the fitting results data.

Set up the manual conversion:

```
c = herschel.share.unit.Constant.SPEED_OF_LIGHT.value
c = c/1000. # to get into km/s
# f0 value is made up as a frame frequency
f0 = 806.00
```

Example 8.4. Defining constants required for the conversion.

In the `freq` matrix we have a default unit of frequency. This will be transformed into velocity below:

```
vel = c*(f0-freq)/f0
```

Example 8.5. Converting the whole matrix values taking advantage of the capabilities of HIPE arrays.

The conversion has been applied to the entire matrix (`Double2d`) but the units are still frequency. As the purpose of the script is creating a map, the `vel` matrix is used to create a `SimpleImage`:

```

VelIm = SimpleImage(image=vel)
```

Example 8.6. Creating a new image object to hold the velocity values.

To modify the units of the image, it is possible to use a `SimpleImage` method, called `setUnits`:

```
VelIm.setUnit(herschel.share.unit.Speed.KILOMETERS_PER_SECOND)
```

Example 8.7. Manually setting the units of the output velocity map.

In this case, the unit is applied to all pixels of an image (created from a 2D array of velocity values assigned to the variable vel). All pixels of the `SimpleImage` are assigned *km/s* as their unit. As you can see, the process is cumbersome, prone to error and requires prior knowledge of the conversion conventions in use with HIPE. If the conversion is not the usual, the objective of keeping meaningful units is not achieved. The `convertUnits` task is able to convert from and to a set of the most common units in HIPE, for the product types that are more used.

8.2. Built-in units and how to define new ones

The unit framework in HIPE has many common units already available. To simplify the handling of complex units in code, they are usually passed as a `String` to the `Unit` constructor. This avoids the requirement of recalling the defined constants for each unit (which, of course, are totally equivalent) and enhances the legibility of the code. For example, `"m-1"` is equivalent to `WaveNumber.RECIPROCAL_METERS`. Built-in units are grouped conceptually and all the conversions between them are bidirectional.

Note

Common names of the units are given in American English spelling (e.g.: meter versus metre) because that is the way they are stored within the software. Both the *Constant* and *Friendly name* columns contain string literals of HIPE classes while the *Common name* column is a description that keeps the American spelling for consistency.

Table 8.1. Units supported by the `convertUnits` task.

Physical quantity	Unit common name	Constant	Friendly name
Acceleration	Meters per second squared	`Acceleration.METERS_PER_SECOND_SQUARED`	`"m s-2"`
Angle	Degrees	`Angle.DEGREES`	`"deg"`
	Arcminutes	`Angle.MINUTES_ARC`	`"arcmin"`
	Radians	`Angle.RADIANS`	`"rad"`
	Arcseconds	`Angle.SECONDS_ARC`	`"arcsec"`
Angular momentum	Joule second	`AngularMomentum.JOULE_SECOND`	`"J s"`
Angular speed	Degrees per second	`AngularSpeed.DEGREES_PER_SECOND`	`"deg s-1"`
	Radians per second	`AngularSpeed.RADIANS_PER_SECOND`	`"rad s-1"`
Area	Square centimeters	`Area.SQUARE_CENTIMETERS`	`"cm2"`
	Square kilometers	`Area.SQUARE_KILOMETERS`	`"km2"`
	Square meters	`Area.SQUARE_METERS`	`"m2"`
Duration	Days	`Duration.DAYS`	`"d"`
	Hours	`Duration.HOURS`	`"h"`
	Julian years	`Duration.JULIAN_YEARS`	`"a"`
	Microseconds	`Duration.MICROSECONDS`	`"microseconds"`
	Milliseconds	`Duration.MILLISECONDS`	`"ms"`
	Minutes	`Duration.MINUTES`	`"min"`
	Seconds	`Duration.SECONDS`	`"s"`
Electric capacitance	Farads	`ElectricCapacitance.FARADS`	`"F"`

Physical quantity	Unit common name	Constant	Friendly name
	Microfarads	`ElectricCapacitance.MICROFARADS`	`"microF"`
	Millifarads	`ElectricCapacitance.MILLIFARADS`	`"mF"`
	Nanofarads	`ElectricCapacitance.NANOFARADS`	`"nF"`
	Picofarads	`ElectricCapacitance.PICOFARADS`	`"pF"`
Electric charge	Coulomb	`ElectricCharge.COULOMB`	`"C"`
Electric conductance	Siemens	`ElectricConductance.SIEMENS`	`"S"`
Electric current	Amperes	`ElectricCurrent.AMPERES`	`"A"`
	Milliamperes	`ElectricCurrent.MILLIAMPERES`	`"mA"`
Electric inductance	Henries	`ElectricInductance.HENRIES`	`"H"`
Electric potential	Millivolts	`ElectricPotential.MILLIVOLTS`	`"mV"`
	Volts	`ElectricPotential.VOLTS`	`"V"`
Electric resistance	Ohms	`ElectricResistance.OHMS`	`"Ohm"`
Energy	Electronvolts	`Energy.ELECTRON_VOLTS`	`"eV"`
	Ergs	`Energy.ERGS`	`"erg"`
	Joules	`Energy.JOULES`	`"J"`
Entropy	Joules per Kelvin	`Entropy.JOULES_PER_KELVIN`	`"J K-1"`
Flux	Ergs per second per square centimeter	`Flux.ERGS_PER_SECOND_PER_SQUARE_CENTIMETER`	`"erg s-1 cm-2"`
	Watts per square centimeter	`Flux.WATTS_PER_SQUARE_CENTIMETER`	`"W cm-2"`
	Watts per square meter	`Flux.WATTS_PER_SQUARE_METER`	`"W m-2"`
Flux density	Ergs per second per square centimeter per hertz	`FluxDensity.ERGS_PER_SECOND_PER_SQUARE_CENTIMETER_PER_HERTZ`	`"erg s-1 cm-2 Hz-1"` (F_v)
	Ergs per second per square centimeter per micron	`FluxDensity.ERGS_PER_SECOND_PER_SQUARE_CEN-`	`"erg s-1 cm-2 micrometer-1"` (F_λ)

Physical quantity	Unit common name	Constant	Friendly name
		TIMETER_PER_MICRON	
	Ergs per second per square centimeter per wavenumber	FluxDensity.ERGS_PER_SECOND_PER_SQUARE_CENTIMETER_PER_WAVENUMBER	"erg s-1 cm-2 cm"(F_k)
	Janskys	FluxDensity.JANSKYS	"Jy"(F_v)
	Microjanskys	FluxDensity.MICROJANSKYS	"microjanskys"(F_v)
	Millijanskys	FluxDensity.MILLIJANSKYS	"mJy"(F_v)
	Watts per square centimeter per Hertz	FluxDensity.WATTS_PER_SQUARE_CENTIMETER_PER_HERTZ	"W cm-2 Hz-1"(F_v)
	Watts per square centimeter per micron	FluxDensity.WATTS_PER_SQUARE_CENTIMETER_PER_MICRON	"W cm-2 micrometer-1"(F_λ)
	Watts per square centimeter per wavenumber	FluxDensity.WATTS_PER_SQUARE_CENTIMETER_PER_WAVENUMBER	"W cm-2 cm"(F_k)
	Watts per square meter per Hertz	FluxDensity.WATTS_PER_SQUARE_METER_PER_HERTZ	"W m-2 Hz-1"(F_v)
	Watts per square meter per micron	FluxDensity.WATTS_PER_SQUARE_METER_PER_MICRON	"W m-2 micrometer-1"(F_λ)
	Watts per square meter per wavenumber	FluxDensity.WATTS_PER_SQUARE_METER_PER_WAVENUMBER	"W m-2 cm"(F_k)
Force	Dynes	Force.DYNES	"dyn"
	Newtons	Force.NEWTONS	"N"
Frequency	Gigahertz	Frequency.GIGAHERTZ	"GHz"
	Hertz	Frequency.HERTZ	"Hz"
	Kilohertz	Frequency.KILOHERTZ	"KHz"
	Megahertz	Frequency.MEGAHERTZ	"MHz"
	Terahertz	Frequency.TERAHERTZ	"THz"
Length	Angstroms	Length.ANGSTROMS	"angstrom"

Physical quantity	Unit common name	Constant	Friendly name
	Astronomical Units	Length.ASTRONOM-ICAL_UNITS	"ua"
	Centimeters	Length.CENTIME-TER	"cm"
	Kilometers	Length.KILOMETER	"km"
	Meters	Length.METERS	"m"
	Micrometers	Length.MICROME-TERS	"micrometer"
	Millimeters	Length.MILLIME-TERS	"mm"
Mass	Grams	Mass.GRAMS	"g"
	Kilograms	Mass.KILOGRAMS	"kg"
Noise Equivalent Power (NEP)	Watts per square root Hertz	NEP.WATTS_PER_SQRT_HERTZ	"W Hz-1:2"
Power	Kilowatts	Power.KILOWATTS	"kW"
	Megawatts	Power.MEGAWATTS	"MW"
	Watts	Power.WATTS	"W"
Pressure	Bars	Pressure.BARS	"bar"
	Millibars	Pressure.MIL-LIBARS	"mbar"
	Pascals	Pressure.PASCALS	"Pa"
Scalar	Beam	Scalar.BEAM	"beam"
	Decibels	Scalar.DECIBELS	"dB"
	One	Scalar.ONE	"1"
	Percent	Scalar.PERCENT	"%"
	Pixel	Scalar.PIXEL	"pixel"
Solid Angle	Square degress	SolidAn-gle.SQUARE_DE-GREES	"deg2"
	Square arcminutes	SolidAn-gle.SQUARE_MIN-UTES_ARC	"arcmin2"
	Square arcseconds	SolidAn-gle.SQUARE_SEC-ONDS_ARC	"arcsec2"
	Steradians	SolidAn-gle.STERADIANS	"sr"
Speed	Kilometers per hour	Speed.KILOMETER-S_PER_HOUR	"km h-1"
	Kilometers per second	Speed.KILOMETER-S_PER_SECOND	"km s-1"
	Meters per second	Speed.METER-S_PER_SECOND	"m s-1"
Surface brightness	Janskys per beam	SurfaceBright-ness.JAN-SKYS_PER_BEAM	"Jy/beam"

Physical quantity	Unit common name	Constant	Friendly name
	Janskys per pixel	SurfaceBrightness.JANSKYS_PER_PIXEL	"Jy/pixel"
	Janskys per square arcsec	SurfaceBrightness.JANSKYS_PER_SQUARE_ARCSEC	"Jy/arcsec2"
	Megajanskys per steradian	SurfaceBrightness.MEGAJANSKYS_PER_STERADIAN	"MJy/sr"
Temperature	Celsius	Temperature.CELSIUS	"degC"
	Kelvin	Temperature.KELVIN	"K"
Thermal conductivity	Watts per meter per Kelvin	ThermalConductivity.WATTS_PER_METER_PER_KELVIN	"W m-1 K-1"
Wave number	Reciprocal centimeters	WaveNumber.RECIPROCAL_CENTIMETERS	"cm-1"
	Reciprocal meters	WaveNumber.RECIPROCAL_METERS	"m-1"

8.2.1. Defining new units

A section on how to define derived units and a general explanation on the usage of the package is available in the Scripting Guide in *Scripting Guide*. To define a completely new unit, you should subclass the abstract class `Unit`, providing the dialog name, the unit name, equivalencies and providing a `compareTo` method suitable for implementing the `Comparable` interface. For more information on how to create new classes in Jython, see this walkthrough in the Scripting Guide in *Scripting Guide*. Since that section does not cover the intrinsics of subclassing Java abstract classes from Jython, you can also consult the last example in the Using Java Within Jython Applications section of the JythonBook, the official reference book for Jython (take care, it only covers the first version of the language and syntax could differ slightly).

8.3. How to convert data products units

At the beginning of this chapter, you could read an example on how to manually convert data in a two-step approach. The first step was to apply the conversion factor between the units to the data and the second step was to explicitly set the unit for the new values. This is correct if you have strange units in your data and don't want to go through all the hassle of creating a Java class and adding it to HIPE as a plug-in or subclassing the `Unit` class directly in Jython. There are additional ways to convert between different units in HIPE through the use of specific converters:

- Convert dates using either `DateConverter` or `CucConverter`.
- Convert flux density values using the `FluxDensityConverter` which is aware of the three different units commonly used as reference for this quantity (frequency, wavelength and wavenumber).
- Convert frequency values using `FrequencyConverter` that handles conversion between `Hz`, `m` and `cm-1`.

- Convert between different formats of the same units with the `UnitFormatConverter`, for example the sexagesimal and decimal formats for angular degrees (`deg`).
- Convert from a `double` (which is the default numeric type in HIPE) to another unit using `DefaultUnitConverter`.

In case your data does not require a specific converter, the safest option is to use the `convertUnits` task. It will convert any unit to another, if they belong to the set of predefined units listed above. The documentation for the task already covers trivial cases like converting arrays of double values (`Double1d`) from `GHz` to `MHz`. Since the task does not directly convert products, the purpose of the rest of this chapter is providing the information and worked examples to convert the units of more complex products like the subclasses of `SimpleCube`, `Spectrum1d`, `Spectrum2d` and `SimpleImage`.

8.3.1. Worked Example: Converting the units of an instance of a subclass of Spectrum1d

In order to convert the units of a SPIRE spectrum of type `SpireSpectrum1d`, you should first extract the `Double1d` array that stores the spectrum data. For this example, `GHz` will be converted to `MHz`. This is a very simple product but still requires the conversion to be made in two steps, since there are two places where the units must be converted.

- The first place is the `Double1d` array, where the values must be converted from `GHz` to `MHz` in the example. All data points will be multiplied by 1000.
- The second place is the metadata property that stores the units for `SpireSpectrum1d`. This property is `waveunit` and must be manually set after converting the values.

```
# Imports
from herschel.share.unit import *

# SPIRE obs
obsid_v1342187084 = getObservation(obsid = 1342187084, useHsa = True)

# SpireSpectrum1d (it is a subclass of Spectrum1d, so valid for the purpose)
spireSpec =
 obsid_v1342187084.refs["level2"].product.refs["HR_spectrum_ext_apod"].product["0000"]
["SLWB2"]

# Check the wave units
print spireSpec.meta['waveunit']

# Convert the wave column
waveMHz = convertUnits(data = spireSpec.wave, sourceUnit="GHz", targetUnit="MHz")

# Make a deep copy of the original spectrum and assign the new data to it
spireSpecMHz = spireSpec.copy()
spireSpecMHz.wave = waveMHz

# Set the new units in the metadata of the copy
spireSpecMHz.setWaveUnit(Unit.parse("MHz"))
```

Example 8.8. How to convert an instance of SpireSpectrum1d.

8.3.2. Worked Example: Converting the units of an instance of a subclass of Spectrum2d

This is very similar to converting an instance of `Spectrum1d`. The main difference is that you must take extra precautions with the subband layout when converting each subband. To emphasise the fact that subbands are independent columns with independent units, an in-place conversion of each wave dataset will be done in the example using a for loop.

Note

This is something very important to remember. If you change the values (and the internal units) of, for example, the wave datasets of subbands 2 and 3 through the execution of `convertUnits` you will end up with inconsistent units in the product. The metadata property `waveunit` of `HrsSpectrumDataset` will still hold the original unit while the converted subband datasets will hold the converted units. Depending on the purpose of the conversion, this could be desired.

```
# Imports
from herschel.share.unit import *

# HIFI observation from the HIFI UM
obsid_v1342190841 = getObservation(obsid = 1342190841, useHsa = True)

# This product is an instance of HrsSpectrumDataset
hrsSpec2d = obsid_v1342190841.refs["level2"].product.refs["HRS-H-
USB"].product.refs["box_001"].product["0001"]

# In place conversion
hrsSpec2d.set("usbfrequency_1", convertUnits(data=hrsSpec2d.getWave(1),
 sourceUnit="GHz", targetUnit="MHz"))

# Note that the waveunit property will be inconsistent if you only convert one
 subband
for index in range(2,5):
   hrsSpec2d.set("usbfrequency_"+`index`,
 convertUnits(data=hrsSpec2d.getWave(index), sourceUnit="GHz", targetUnit="MHz"))

# Now you can update the waveunit property
hrsSpec2d.setWaveUnit(Unit.parse("MHz"))
```

Example 8.9. How to convert an instance of HrsSpectrumDataset.

8.3.3. Worked Example: Converting the units of an instance of a subclass of SimpleCube

To convert an instance of `SpectralSimpleCube`, the procedure is very similar to other products. First, you must extract the data values to be converted from the product, then execute the `convertUnits` task to actually convert the values. To finish, create a copy of the original product with the converted values and the appropiate units stored in its metadata.

- The `Double3d` dataset containing the values of the cube is stored in the `image` property of the product. In this case, the unit for this dataset is `K` and the conversion will modify the values to degrees Celsius, subtracting 273.15 to each data point.
- The second step (as is the case with the other examples in this chapter) is to manually update the metadata property holding the old unit. Be it in the original object or in a copy, the unit property must be updated if consistency is required.

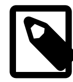
```
# Imports
from herschel.share.unit import *

# HIFI obs
obsid_v1342180474 = getObservation(obsid = 1342180474, useHsa = True)
# SpectralSimpleCube
hifiCube =
 obsid_v1342180474.refs["level2_5"].product.refs["cubesContext"].product.refs["cubesContext_HRS-
H-LSB"].product.refs["cube_HRS_H_LSB_1"].product

# Extract the antenna temperature information and convert the units of the dataset
cubeTemp = hifiCube['image']
convertedTemp = convertUnits(data = cubeTemp, sourceUnit="K", targetUnit="degC")

# Assemble a converted copy of the cube
convertedCube = hifiCube.copy()
```

```
convertedCube['image'] = convertedTemp

# Set the units of the product
convertedCube.setFluxUnit(Unit.parse("degC"))
```

Example 8.10. How to convert an instance of SpectralSimpleCube.

8.3.4. Worked Example: Converting the units of an instance of SimpleImage

The conversion of instances of `SimpleImage` is very similar to the conversion of a cube perfomed above. The image data can be converted in one go and the unit is updated afterwards. Usually, the units of a `SimpleImage` are composite units of the quantity Surface Brightness (see Table 8.1).



Note

Please note that converting between *different* Surface Brightness units is not possible because the relationships between pixels, steradians and beams are instrument-specific. Converting between multiples and sub-multiples of the same unit like `Jy/pixel` to `MJy/pixel` is perfectly possible.

```
# Imports
from herschel.share.unit import *

# PACS Photo observation from the PACS photo manual
obsid_v1342182962 = getObservation(obsid = 1342182962, useHsa = True)

# SimpleImage present in the observation
simpImg = obsid_v1342182962.refs["level2"].product.refs["HPPPMAPB"].product

# Image data
imgData = simpImg.image

# In-place conversion from Jy/pixel to MJy/pixel
simpImg.setImage(convertUnits(data = imgData, sourceUnit = "Jy/pixel", targetUnit =
 "MJy/pixel"))

simpImg.setUnit(Unit.parse("MJy/pixel"))
```

Example 8.11. How to convert an instance of SimpleImage.

Index

A

Adding images, 184
Aladin, 52, 52, 54
Annotation toolbox, 178
  generating Jython code , 181
Annotations
  on images , 178
  on plots, 110, 112
annularSkyAperturePhotometry , 212
Aperture correction, 221
Aperture photometry, 211
  annular, 212
  centroiding, 211
  fixed, 220
  on point sources , 212
  rectangular, 218
  units, 212
ASCII files, 56
  adding a header, 85
  adding metadata, 85
  and FITS files, 56, 56
  and source list products, 60
  and spectra, 56, 58
  and Spitzer spectra, 62
  choosing how to separate data values, 86
  configuration files for reading, 84
  configuration files for writing, 86
  defining lines to ignore, 80
  formatters, 87
    creating and configuring, 90
  ignoring white space, 82
  parsers, 87
    creating and configuring, 89
  prefix for commented lines, 86
  reading a generic file, 71
  reading column names, 80
  reading from comma-separated-value files, 67
  reading from space-separated-value files, 68
  specifying data types, 82
  specifying how data are separated, 83
  templates, 87
    creating and configuring, 88
  writing to comma-separated-value file, 72
  writing to generic file, 79
  writing to space-separated-value file, 73
autoAdjustWindowSize, 95
automaticContour, 194
Auxiliary context, 1
Axis (of a plot), 104, 109
AxisTick, 107
AxisTickLabel, 108
AxisTitle, 106

B

boxCarSmoothing, 187

C

Calibration context, 1
circleHistogram, 196, 200
Clamping images, 182
CLASS
  opening Herschel spectra in, 49
Clipping images, 182
Colour map, 177
Combo-fitting, 309
ComboModel, 358
Comma-separated-value file
  reading into HIPE, 67
  writing a table dataset to, 71
Compass (on images), 179
Configuration file (ASCII files), 58
Contour plots, 194
  deleting, 195
Convolving images, 188
Cropping images, 182, 182
CSAT (see Cube Toolbox)
CSV file (see Comma-separated-value file)
CsvFormatter, 90
CsvParser, 89
Cube Spectrum Analysis Toolbox (see Cube Toolbox)
Cube Toolbox, 270
  accessing tasks, 286
Cubes, 269
  baseline issues, 306
  combining PACS and SPIRE full SED, 307
  continuum, removing the, 305
  coordinates, 271
  cropping, 292
  displaying, 274
    changing axes, 280
    changing properties, 280
    flags, 283
    grid layout, 278
    metadata, 284
    multiple, 279
    overplotting, 277
    plot-mouse interactions, 282
    real-time spectral display, 277
    showing and hiding, 276
    zooming and panning, 277
  errors, 272
  fitting
    polynomial to, 323
    polynomial+gaussians to, 333
    with multi fit, 323
  flags, 272
  flux maps, 302
  flux maps without spectrum fitting, creating, 301
  position-velocity maps, 305
  printing, 283

saving as image, 283
selecting spectra, 286
spectra input, 286
spectra output, 286
spectrum
arithmetics, 293
averaging, 294, 296
flag propagation, 301
flagging, 298
folding, 296
gridding, 296
replacing, 296
resampling, 296
smoothing, 296
statistics, 294
stitching, 296
summing, 296
unit conversion, 300
weight/error propagation, 301
spectrum extraction, 290
over spectral domain, 290
random spaxels, 290
velocity maps, 303
weights, 272
Curve of growth, 213
Cut levels, 177
Cut levels in images, 188

D

Delimiter (ASCII files), 58
dictionary (Jython data structure), 57
Dividing images, 185
DS9, 46, 52, 53
opening Herschel FITS files in, 52

E

ellipseHistogram, 196, 201
Error bars (in plots), 118
Exponential function of an image, computing
base 10, 186
base N, 186
natural, 186
exportSpectrumToAscii, 56, 58, 74

F

FitFringe, 260
running, 260
FITS
header character limit, 46
History extension, 45
importing non-Herschel files into HIPE, 46
keywords, 39
loading Herschel data in external apps, 48
loading products from, 38
multi-extension files in DS9, 46
saving products to, 36
structure of Herschel products, 41
HifiTimelineProduct, 44
PacsRebinnedCube, 43
SimpleImage, 41
SpectralSimpleCube, 42
SpectrometerPointSourceSpectrum, 45
troubleshooting, 46
fitsReader, 38, 46
fitting
making maps
making images, 362
mapping, 362
Fitting spectra, 309
fixedSkyAperturePhotometry, 220
FixedWidthFormatter, 91
FixedWidthParser, 89
flagPixels, 254, 298
flagSaturatedPixels, 188
Formatter (ASCII files), 57, 87
creating and configuring, 90

G

gaussianSmoothing, 187
General Standing Wave Removal Tool, 260
getObservation
browsing an observation online, 15
download a single observation, 13
download multiple observations, 18
loading an observation into HIPE, 20
multiple versions, 16
retrieving observation from disk, 26

H

Herschel Science Archive
browsing an observation online, 14
downloading a single observation, 13
downloading data from, 7
downloading multiple observations, 17
inspecting query results, 11
logging in, 8
querying, 10
hiClass (HIFI task), 49
HifiSpectrumDataset, 243
Histograms
in images , 196
in plots, 119
History, 1
HistoryParameters, 45
HistoryScript, 45
HistoryTasks, 45
HrsSpectrumDataset, 243
HSA (see Herschel Science Archive)

I

IDL
loading Herschel data in, 48
Image analysis, 170
(see also Images)

running tasks , 170
imageHistogram, 196
ImageIndex, 271, 272
Images
absolute value , 185
adding, 184
aperture photometry , 211
ceiling, 185
clamping, 182
clipping, 182
colour map , 177
contour plots , 194
converting units , 187
convolving, 188
cropping, 182, 182
cut levels , 177, 188
distances, measuring , 174
dividing, 185
embedding in plots, 121, 122
exponential function
base 10, 186
base N, 186
natural, 186
exporting from HIPE , 172
flagging saturated pixels , 188
flipping Y axis , 174
floor, 185
histograms, 196
importing into HIPE , 171
intensity profiles , 192
logarithm
base 10 , 186
base N , 186
natural, 186
modulo, 185
multiplying, 184
raising to a power , 185
RGB, 189
rotating, 182
rounding, 185
saving, 176
scaling, 183
smoothing, 187
source extraction , 203
source fitting , 210
square root , 186
squaring, 186
stitching, 189
subtracting, 184
translating, 183
transposing, 183
viewing, 173
viewing metadata , 175
zooming, 173
Intensity profiles, 192
IPAC, 70
IRSA, 70

J
java.awt.Color, 104, 108, 113, 126, 126
java.awt.Font, 108, 126

L
Layers (of plots), 94, 97, 98, 99
Level 0, 1
Level 0.5, 1
Level 1, 1
Level 2, 1
Level 2.5, 1
Line styles (for plots), 102
list (Jython data structure), 57
Local pools, 5
Local stores (see Local pools)
Logarithm of an image, computing
base 10, 186
base N, 186
natural, 186

M
manualContour, 194
Masks, 175
medianSmoothing, 187
Metadata
in images , 175
to FITS keywords, 39
multi fit, 323
Multi-fitting, 309
Multiplying images, 184
MyHSA, 4, 21
advanced, 23

N
NaN, 362
Navigator view, 20
Not a Number (see NaN)
Numeric arrays
exporting to ASCII file, 57

O
Observation, 1
browsing online, 14
downloading from the Herschel Science Archive, 7, 13
exporting, 34
loading into HIPE, 19
removing from disk, 35
retrieving from disk, 24
saving to disk, 30
Searching by target name, 28
Observation context, 1
Observation ID
finding, 12
Observation log context, 1
Observing log, 12
obsid (see Observation ID)

On-demand reprocessing, 35
Over Plotter, 164
  controls and functions, 166
  invoking, 164
  layout, 164

P

PACS
  combining full SED with SPIRE, 307
PACS projected cube, 42
PacsRebinnedCube, 270
Parser (ASCII files), 57, 87
  creating and configuring, 89
photApertureCorrectionPointSource , 221
Plots, 94
  annotations, 110, 112
  auxiliary axes, 116
  axes, 104
  axis labels, 104
  axis methods, 109
  axis range, 104
  axis thickness, 118
  batch mode, 125
  classes, 129
  colours, 126, 126
  creating, 94
  drawing lines, 115
  embedding images
    monochromatic, 121
    RGB, 122
  error bars, 118
  filled areas, 113
  fonts, 126
  grid lines, 110
  histogram mode, 119
  invisible, 127
  layers, 94, 97, 98, 99
  legend, 100
  line styles, 102
  logarithmic axes, 105
  margins, 102
  math and special symbols, 124
  mouse coordinates, 128
  multiple plots, 125
  printing, 102
  properties, 100
    command line, 101
  saving, 102
  size, 95
  styles, 104
  subplots, 120
  symbol styles, 103
  tick marks, 105
  ticks, 107
    labels, 108
  title and subtitle, 95
  visibility, 126
  worked examples, 129, 138, 155
PlotTitle, 96
polygonHistogram, 196, 202
Pools, 4, 5
  Unknown format version, 32
Power Spectrum Generator, 168
Power, raising an image to a , 185
Product Browser, 24, 35
  customising results, 27
Products
  loading from FITS, 38
  removing from disk, 35
  retrieving from disk, 35
  saving to disk, 31
  saving to FITS, 36

Q

Quality context, 1
Quality control report, 11

R

rectangleHistogram, 196, 201
rectangularSkyAperturePhotometry , 218
RegexParser, 89
Regular expression, 58, 90, 92
RGB images, 189
  embedding in plots, 122
RgbSimpleImage, 171, 190
Rotating images, 182
Rounding images, 185

S

SAMP, 52
  Hub Monitor, 53
SAOImage DS9 (see DS9)
saturated pixels in images, flagging , 188
Save Products tool, 31
saveObservation
  and saveProduct, 32
saveProduct, 31
Scale (on images), 179
Scaling images, 183
SED
  PACS and SPIRE, combining, 307
SExtractor, 70
Shopping basket, 17
SimpleCube, 270
simpleFitsReader, 38, 46
simpleFitsWriter, 37
Sky intensity plot, 213
SkyMask, 175, 207
  SkyMaskCircle, 207
  SkyMaskIntersection, 208
  SkyMaskRectangle, 207
  SkyMaskUnion, 208
SmoothBaseline, 265
  running, 265
Smoothing images, 187

Source extraction, 203
common problems , 210
customising source circles , 206
filtered map , 205
known sources , 206
point response function , 205
custom, 207
regions of interest , 207
removing sources , 206
source lists in FITS files , 210
source lists in text files , 209
Source fitting, 210
sourceExtractorDaophot, 203
sourceExtractorSussextractor, 203
SourceListProduct, 60, 204
saving as ASCII files, 60
Space-separated-value file
reading into HIPE, 68
writing a table dataset to, 73
Spectra, 243
arithmetics, 255
averaging, 255
baseline smoothing, 265
displaying, 243
changing axes, 247
changing properties, 248
filtering and sorting, 251
flag/mask information, 251
large datasets, 249
multiple, 246
showing and hiding, 244
zooming and panning, 247
exporting to ASCII file, 56, 58, 74
extracting, 253
fitting, 309
cube or multi-spectrum dataset, 313
cube; maps, 339
Gaussians to spectrum, 326
polynomial to baseline/continuum, 316
polynomial to cube, 323
polynomial+gaussians to cube, 333
single spectrum, 311
flag/mask propagation, 259
flagging, 253
folding, 257
gridding, 257
importing from ASCII file, 59
line masking, 265
overplotting, 245
printing, 252
replacing, 257
resampling, 257
saving as image, 252
smoothing, 257
standing wave removal, 260
statistics, 255
stitching, 257
unit conversion, 257
weight/error propagation, 258
Spectral cubes (see Cubes)
SpectralLineList, 243
SpectralSimpleCube, 270, 270
SpectrometerDetectorSpectrum, 243
SpectrometerPointSourceSpectrum, 243
Spectrum Explorer, 243, 270, 274
changing preferences, 283
Spectrum Fitter, 309
adding and initialising models, 341
adding your own model, 360
applying a fit, 346
automatic fit of multiple datasets, 355
automatic fit upon opening, 342
combo model, 358
cubes, 362
deleting and excluding models, 348
fixing parameters, 345
from the command line, 314
inspecting fit results, 346
models, 359
modifying models, 345
NaN handling, 362
obtaining a line integral, 353
resetting and restarting, 348
residuals, 358
saving a script, 349
saving residuals and models, 350
selecting best fitter engine, 362
setting limits to model parameters, 344
setting weights, 342
starting the GUI, 309
using saved models, 354
Spectrum Toolbox, 253, 270
Spectrum1d, 243
SpectrumContainer, 243, 270
SpectrumDataset, 243, 270
SPIRE
combining full SED with PACS, 307
Spitzer
reading spectra into HIPE, 62
Square of an image, computing , 186
Square root of an image, computing , 186
SSV file (see Space-separated-value file)
Stitching images, 189
Storages, 4, 5
Subtracting images, 184
Symbol styles (for plots), 103

T

Table Plotter, 156
controls and functions, 158
invoking, 157
layout, 158
Table template (ASCII files), 58
Tags, 26
Template (ASCII files), 87
creating and configuring, 88

Ticks (of a plot axis), 107
    labels, 108
Topcat, 52, 54
Translating images, 183
Transposing images, 183
Trend analysis context, 1
tuple (Jython data structure), 57

U

Using IRSA, 12
Using Vizier, 13

V

Virtual Observatory, 52
    troubleshooting, 53
VizieR
    reading catalogue into HIPE, 64
VO (see Virtual Observatory)
VOSpec, 52, 52

W

WbsSpectrumDataset, 243
WCS (see World Coordinate System)
Workflow, 7
World Coordinate System, 190