

A Basic User's Manual

Scripting in the Herschel Data Processing System

**version 0.24.1730, Document Number: HERSCHEL-HSC-DOC-0517
14 October 2008**



A Basic User's Manual: Scripting in the Herschel Data Processing System

Table of Contents

The Herschel Common Science System and Data Processing (DP)	ix
1. Brief Overview	ix
2. Availability of DP and Operating Systems	ix
3. Related Documentation	x
4. Versioning	x
5. What's New and Previous Versions of DP User's Manual	x
6. List of Contributors	xviii
1. HCSS Downloading and Installation	1
1.1. Introduction	1
1.2. Platform	1
1.3. Minimum System Requirements	1
1.4. Pre-Installation Requirements	1
1.5. User Installation Procedure	2
1.6. DP Property Initialisation	3
2. Using JIDE or the JIDE View in HIPE	4
2.1. Introduction	4
2.2. DP Scripting Using the Editor View of HIPE	4
2.2.1. File Menu	6
2.2.2. Edit Menu	6
2.2.3. Run Menu	7
2.2.4. Exiting HIPE	7
2.2.5. Window and Help Menus	7
2.3. DP Scripting Using JIDE	7
2.3.1. File Menu	9
2.3.2. Console Menu	9
2.3.3. Edit Menu	9
2.3.4. Run Menu	10
2.3.5. Help Menu	10
2.4. Quitting JIDE	13
2.5. Standard Settings for JIDE and HIPE	14
2.6. DP working directory and file access	14
2.7. Getting Command-line Help in JIDE or HIPE	15
2.8. Programming Loops in JIDE and HIPE	15
2.8.1. Loop Performance on Arrays	16
2.8.2. Using the Editor view with loops	17
2.9. Multiline Statements in the Console View of HIPE or JIDE	17
2.10. Pausing during script execution and debugging in JIDE (ONLY)	18
2.11. Background script execution in JIDE and HIPE	18
2.12. Running Scripts from a Shell Command Line	19
2.13. Errors and Exceptions in DP	19
2.13.1. Overview of the Libraries Used in a DP Session	19
2.13.2. The Error Traceback Mechanism	20
2.13.3. The HCSS exception and logging mechanism	22
3. Some DP Basics & Beginning Jython	24
3.1. Basics	24
3.2. Comments	24
3.3. Variables	24
3.4. Numbers and basic arithmetic	24
3.5. Boolean values	25
3.6. Strings	25
3.7. Type conversions	26
3.8. Lists and Dictionaries	26
3.8.1. Setting up and Accessing Lists	26
3.8.2. Slicing Lists	27
3.8.3. Setting Up and Using Dictionaries	27

3.8.4. Nested Dictionaries	28
3.9. Augmenting Values and Lists	28
3.10. Lists and Jython Tuples	29
3.11. Basic programming statements	29
3.11.1. if/elif/else	29
3.11.2. for	30
3.11.3. while	31
3.11.4. Loop control: break and continue	31
3.12. Printing to the screen and files	31
3.13. Defining and Using Functions	32
3.14. Object Oriented Programming	34
3.14.1. Classes and Objects	34
3.14.2. Interface, Implementation and Encapsulation	35
3.14.3. Inheritance	36
3.14.4. Packages and Namespaces	36
3.14.5. Advantages of OOP	37
3.14.6. Concluding Remarks	37
3.15. Defining a Class in DP	37
3.16. Writing Scripts - Programming in DP	38
3.17. Some Useful Extra Items on Scripts	39
3.18. Interactivity in Jython Scripts	40
3.18.1. Basic Interactivity	40
3.18.2. A Little Bit of Swing	41
3.19. Useful Java bits	44
3.20. Jython and DP Quirks	45
3.20.1. Two functions for one goal	45
3.20.2. Long Names versus Short Names	45
3.20.3. Naming conventions	46
3.20.4. Miscellaneous quirks	46
4. Handling Array Data Objects, Datasets and Products	48
4.1. Introduction	48
4.2. Getting started	48
4.3. Types of Array Data Objects	48
4.3.1. DP Numeric Array Access and Slicing	49
4.4. Creating a Simple 1D DP Numeric Array	49
4.5. Creating and Handling Complex Array Data Objects	50
4.6. Creating and Accessing Multi-Dimensional Array Data Objects	50
4.7. Adding Attributes to Create an Array Dataset	51
4.7.1. Dataset Attributes and Metadata	51
4.8. Creating and Viewing a TableDataset	52
4.8.1. Row-wise appending of TableDatasets	53
4.8.2. Assigning Units	53
4.9. Creating and Accessing a Composite Dataset	56
4.10. Spectrum Datasets	57
4.10.1. Spectrum1d and SpectralSegments	57
4.10.2. Spectrum2d	58
4.10.3. Expanding Spectrum1d and Spectrum2d Datasets	59
4.11. Image and Cube Datasets	61
4.12. Assigning a World Coordinate System (WCS) to SimpleImage and SimpleCube.....	62
4.13. Wrapping it all up: Products	65
4.13.1. Mandatory Parameters in Products	65
4.13.2. Setting Date Information	66
4.13.3. Additional Metadata	66
4.13.4. Inserting and Getting Datasets from a Product	66
4.14. The Dataset Inspector	67
4.14.1. The TablePlotter	67
4.14.2. The Power Spectrum Viewer	73
5. DP Numeric: Basic Functions for Herschel DP	76

5.1. Introduction	76
5.2. Getting Started	76
5.3. Basic Numeric Array Arithmetic	76
5.4. Numeric Functions and Lambda Expressions	77
5.5. Selection, Data Filtering and Masking Methods	77
5.6. Array Access and Slicing	80
5.7. Making sense of logical operators	80
5.8. Advanced Tips for Improved Performance	81
5.9. Type Conversions	82
5.9.1. Explicit conversion	82
5.9.2. Implicit conversion	82
5.10. Function Library	82
5.10.1. Basic Functions	83
5.10.2. Integral Transforms	83
5.10.3. Convolution	84
5.10.4. Boxcar and Gaussian Filters	85
5.10.5. Interpolation Functions	85
5.10.6. Basic Fitter Routines	87
5.10.7. Spectral Fitting	94
5.10.8. Matrix Manipulations	101
5.10.9. Random numbers generation	102
5.10.10. Numeric Integration	103
5.10.11. Interpolating Discrete Data	104
5.11. Example Programs	105
5.12. Mathematical Operations on Spectra	105
5.12.1. Introduction	105
5.12.2. Toolbox Primer: Selection	106
5.12.3. Toolbox Primer: Average Spectra	107
5.12.4. Toolbox Primer: Subtract Spectra	108
5.12.5. Toolbox Primer: Divide Spectra	108
5.12.6. Toolbox Primer: Add and Multiply Spectra	108
5.12.7. Toolbox Primer: Resample and Smooth Spectra	108
5.12.8. Toolbox Primer: Statistics on Spectra	109
5.12.9. Summary of Toolbox Operations	109
6. DP Plot: Basic Plotting of Data	111
6.1. Introduction	111
6.2. What do I need to make a simple XY plot?	112
6.2.1. Introducing PlotXY	112
6.3. How to setup your PlotXY properties	114
6.3.1. How to modify properties	115
6.3.2. Plot properties	115
6.3.3. Layer properties	116
6.3.4. Axis properties	118
6.3.5. How to use properties	119
6.4. Manipulating Layers, Axes, and Annotations in DP Scripts	121
6.4.1. What about these Layers?	121
6.4.2. What can I do with Axis?	126
6.5. Adding Error Bars to a Plot	132
6.6. How can I annotate, decorate and save my plot?	134
6.7. How can I make my plots more colourful?	136
6.8. Creating file output and printing a plot without displaying	136
6.8.1. Using batch mode	137
6.9. Windows containing more than one plot	137
6.10. Mouse Interactions with Plots	139
6.11. What about a complete PlotXY example?	140
7. Display: Handling Images and Cubes with Herschel DP	141
7.1. Introduction	141
7.2. Using SimpleImages and SimpleCubes	141

7.3. Working with Flags	142
7.4. How can I display my Image?	143
7.5. Display in more Detail.	146
7.6. Working with the World Coordinates System	147
7.7. How can I use Operations on my Images?	148
7.7.1. Clamping (or clipping) an Image	149
7.7.2. Cropping an Image	149
7.7.3. Histogram of an Image	149
7.7.4. Rotating an Image	153
7.7.5. Scaling an Image	155
7.7.6. Translating an Image	155
7.7.7. Transposing an Image	156
7.7.8. AperturePhotometry	156
7.7.9. Contour plotting	157
7.7.10. 2D Profile Plotting	157
7.7.11. Smoothing Images	158
7.7.12. Making Noise Images	158
7.7.13. Flagging saturated Pixels	158
7.8. How can I display my own numeric2d datatypes?	158
7.9. How to Use Different Layers?	159
7.10. How to place annotations on an Image?	160
7.10.1. Annotations from the Command Line in your DP session	160
7.10.2. Annotations using the annotation toolbox	162
7.11. ImageAnalysisToolbox : Image Analysis with Herschel DP	164
7.11.1. General Description of the Toolbox	164
8. Introduction to Tasks	175
8.1. The Task framework	175
8.2. My first Task	175
8.2.1. Before the Task	175
8.2.2. What makes a Task?	176
8.2.3. An Example of a Task: Average	177
8.3. Guideline on How to Work With GUIs Within Tasks	183
8.3.1. The use of task parameters handled via a dialog	183
8.3.2. The use of more enhanced GUIs	183
8.3.3. Example Task Handled by a Dialog	183
8.3.4. Example Task Controlled by a GUI	184
9. Other DP Packages: What is Available?	185
9.1. Introduction	185
9.2. Overview of JavaDocs Documentation for DP Packages	185
9.3. Package view	186
9.4. Class view	188
9.5. Tree view	190
9.6. Deprecated view	190
9.7. Index view	190
9.8. DP Packages And Documentation	190
9.8.1. herschel.ia.dataflow	190
9.8.2. herschel.ia.dataset	190
9.8.3. herschel.ia.demo	191
9.8.4. herschel.ia.doc	191
9.8.5. herschel.ia.document	191
9.8.6. herschel.ia.help	191
9.8.7. herschel.ia.image	191
9.8.8. herschel.ia.inspector	192
9.8.9. herschel.ia.io	192
9.8.10. herschel.ia.jconsole	192
9.8.11. herschel.ia.numeric	192
9.8.12. herschel.ia.plot	193
9.8.13. herschel.ia.task	193

9.8.14. herschel.ia.ui	193
10. IO of DP Variables, Tabular ASCII and FITS Files	194
10.1. Introduction	194
10.2. Saving and Restoring DP Variables	194
10.3. Getting Started with ASCII Import/Export	195
10.3.1. Basic ASCII Table Import/Export Tool Usage	195
10.3.2. Examples of How to Import/Export ASCII Tables in DP	198
10.4. Overview of FITS IO	200
10.4.1. Getting Started With FITS IO	200
10.4.2. Parameter Name Conversion and FITS Header	201
10.4.3. Caveats	203
11. Using Time in the DP Environment	204
11.1. Introduction	204
11.2. Time Definitions	204
11.2.1. System time in DP	204
11.2.2. International Atomic Time (TAI) and FineTime	205
11.2.3. Coordinated Universal Time (UTC)	205
11.2.4. DecMec Time [PACS only]	205
11.3. Time in Instrument House-Keeping (HK) Data	206
11.4. Time conversion	206
11.4.1. Time conversion in HCSS	206
11.4.2. CucConverter	207
12. Accessing and Retrieving Data	208
12.1. The Product Access Layer and Product Pools	208
12.1.1. Available Product Pools	208
12.1.2. Local Pools	208
12.1.3. DbPool	214
12.1.4. HsaReadPool	214
12.1.5. CachedPool	214
12.1.6. Setting up and Accessing Remote Pools	214
12.1.7. Special Imports into Pools	215
12.1.8. Common Problems	216
12.1.9. Storage Product Versioning	217
12.1.10. The Product Browser	219
12.2. Databases	223
12.2.1. Introduction	223
12.2.2. Starting Up A Database:	223
12.2.3. Schema Evolution	224
12.2.4. Providing Database Access for a DP Session	224
12.2.5. Changing the Database to be Accessed	225
12.2.6. Browsing a Database	225
12.2.7. Getting Data Frames From a Database	226
12.2.8. Accessing Housekeeping (HK) Data	230
12.2.9. Removing a Database	233
A. Data Reduction Tutorial -- contributed by Russ Shipman	234
A.1. Introduction	234
A.2. Getting Data into Your Session	234
A.3. Products and Data Wrappers	235
A.4. Numerical Calculations	236
A.5. Plotting	237
A.6. Writing a Task	241
A.7. Fitting a Model	243
A.8. Saving Data and Session	248
B. Example User's Property File	250
C. Jython Operators	252
D. Demo script	254
D.1. Introduction	254
D.2. Demonstrations illustrating specific functionality	254

E. Naming Conventions 256

The Herschel Common Science System and Data Processing (DP)

1. Brief Overview

The Herschel Common Science System (HCSS) is being developed by the Herschel Science Center (HSC) and Herschel Instrument Control Centers (ICCs) to provide the complete software system for the Herschel Observatory mission. The intention is to provide a common system that is able to handle test data, observation planning, mission planning and instrument data from observations within one common development. An important element of this common development is Data Processing (DP).

DP handles computed, stored or simulated data and has access to much of the software developed for other purposes within the HCSS (e.g., Quick Look Analysis, which runs on real-time data or replayed data streams from a database).

Branches of the HCSS have also been developed for handling Herschel instrument-specific tasks. So software packages for HIFI, PACS and SPIRE also reside within the HCSS framework and are available within DP.

Since the Herschel DP uses Java and Jython programming, it is very flexible and Java classes can be imported into a session. However, the basic DP system is already a fully-fledged standalone system being developed to deal with data from the Herschel spacecraft, so that users should not need to import additional Java modules, unless stated otherwise.

This manual is intended for the more **advanced user** who is interested in developing scripts and tools within the DP system. It places an emphasis on command-line interactions which can be put together to make flexible scripts for specific user tasks. It should be noted that such command-lines often mimic the capabilities of HIPE tools -- which are displayed in the console view of HIPE when being used interactively. This allows for copying and editing of interactive operations into user scripts such as is described in this manual.

2. Availability of DP and Operating Systems

DP is available free of charge as part of the HCSS and can be downloaded for use on networked or individual desktop/laptop machines. Current operating systems supported by DP include

- Solaris 2.8+
- Linux (Red Hat 8.0+, SuSE 9.1+ and LSB 1.3 compliant distributions)
- Mac OS X
- Windows (2000, XP)
- Note that Windows Vista is NOT currently supported and an installer is not provided for this system.

In order to allow full use of the system, including download, the following browsers should be used.

- IE 6+ ,
- Netscape 7+,
- Mozilla (Firefox) 1.5+,
- Safari (Mac)

For download and installation instructions see Chapter 1.



Note

Being DP a multiplatform software, screenshots in this manual come from different operating systems. Do not worry if the look and feel you get on your system is different from what you see in this manual; while things like window decorations may vary, all the relevant features are system independent.

3. Related Documentation

The current document is intended to complement the "cookbook" approach to using the HIPE user interface by users -- incorporated in the "HowTo's Manual." The current document is intended for the more advanced user who intends to do more involved scripting as compared to the cookbook (often GUI-based) interactions described in the "HowTo's manual."

Currently available is a *User's Reference Manual* that contains a command dictionary for all available DP tasks.

4. Versioning

DP is still very much a system under development and it is intended that this manual will be updated with the regular user release updates of the system. The first version of this manual is associated with User Release v0.3 of the HCSS. Each new version will report the HCSS User Release number it is associated with. Every care has been put into ensuring that the text and example code are consistent with the corresponding HCSS User Release; however, no guarantee can be given on compatibility with future releases or developers' builds.

This version of the User's Manual is associated with User Release v0.4.3 of the HCSS.

5. What's New and Previous Versions of DP User's Manual

Example scripts for v0.24 were tested using build 1776 of the HCSS, equivalent to HCSS user release version 0.6.5.

- **The following was changed for v0.24**
- *Subtitle*: Updated to indicate hierarchical view of UM more describing scripting possibilities.
- *Preface*: Updated preface to accentuate use of Basic User's Manual
- *Chapter 1*: Installation instructions updated to use installer, plus system recommendations included.
- *Chapter 2*: Complete upgrade to include full description of Classic(JIDE) perspective for HIPE and updates within JIDE. Background processing corrected/updated for both HIPE and JIDE. Added more prominently tips on best performance methods (especially with respect to loops).
- *Chapter 5*: Tips on speed improvements updated.
- *Chapter 7*: Updates with respect to Display and WCS API changes.
- *Chapter 12*: HsaReadPool added to pools available.

Example scripts for v0.23 were tested using build 1708 of the HCSS, equivalent to HCSS user release version 0.6.4.

- **The following was changed for v0.23**
- *Preface*: Updated preface to contain information on supported platforms and browsers plus updated section on related documents.

- *Chapter 1*: Installation instructions updated to use installer, plus system recommendations included.
- *Chapter 2*: Added information on the use of "jylauncher."
- *Chapter 5*: Updates to spectral arithmetic and spectral fitting added.
- *Chapter 5, 6, 7 and 8*: Updates to examples due to changes in API.
- *Chapter 7*: Updates to Display incorporated
- *Chapter 12*: Complete revamp of PAL section of the chapter. Added information on importing FITS and PNG files into pools.

Example scripts for v0.22 were tested using build 1602 of the HCSS, equivalent to HCSS user release version 0.6.3.

- **The following was changed for v0.22**

- *Chapter 4*: Added examples of setting dates in a Product; Added information on Spectrum1d, Spectrum2d, SimpleImage and SimpleCube datasets and the application of WCSs; Extended information on the use of Units.
- *Chapter 6*: PlotXY updates w.r.t. updated API.
- *All*: Updated all chapters to new documentation framework.

Example scripts for v0.21 were tested using build 1547 of the HCSS, equivalent to HCSS user release version 0.6.2.

- **The following was changed for v0.21**

- *Chapter 2*: JIDE "Run" menu updates.
- *Chapter 5*: Added information on input of own non-linear fitter function with examples; SIGMA removed.
- *Chapter 6*: Updated information and examples to match new PlotXY API.
- *Chapter 7*: Complete document update of all Image functions with updated Image API.

Example scripts for v0.20 were tested using build 1480 of the HCSS, equivalent to HCSS user release version 0.6.1.

- **The following was changed for v0.20**

- *Chapter 2*: JIDE interactive debugger (pause()); JIDE "File" menu updates.
- *Chapter 3*: Warning on maximum Jython script size inserted.
- *Chapter 4*: Updated TablePlotter documentation included. Mandatory Product metadata updated and date creation for Products illustrated.
- *Chapter 5*: Spectrum arithmetic section added. Spectrum fitter section added with example.
- *Chapter 7*: Removed deprecated Histogram() task.
- *Chapter 12*: Simplified query included and example time query. Removed PoolManager section. Added short section indicating how to turn off product versioning in pools.

Example scripts for v0.19 were tested using build 1403 of the HCSS, equivalent to HCSS user release version 0.6.0.

- **The following was changed for v0.19**

- *All chapters*: Checks on code examples.

- *Chapter 4*: DatasetInspector updates included. TableDataset section updated to include TableModel method of obtaining/changing table values. A section was added about units and how to handle them based on the herschel.ia.share.unit package.
- *Chapter 6*: Plot introduction changed, no reference to old plot package and removal of composite plot discussions. All plot properties (plot, layer and axis) descriptions updated and extended.
- *Chapter 7*: All nanoTITAN library discussions removed and examples now use share.unit instead.
- *Chapter 10*: RegexpParser described in ASCII table input section plus example given.

Example scripts for v0.18 were tested using build 1349 of the HCSS, equivalent to HCSS user release version 0.5.2.

- **The following was changed for v0.18**

- *All chapters*: Checks on code examples.
- *Chapter 3*: More information on type conversion.
- *Chapter 4*: Convenience setter methods added to section 4.10.
- *Chapter 5*: Interpolating discrete data -- FitterFunction -- section added to the end of the chapter.
- *Chapter 6*: Removed deprecations (e.g. setText) and indicated TEX-like method for getting subscripts and superscripts in labels.
- *Chapter 12*: Local store section moved from 12.2.4 to 12.2.1.9. Added information on PoolDaemon in sections 12.2.1.9 and in association with PoolManagers in section 12.2.2. Added section on Storage Product Versioning.

Example scripts for v0.17 were tested using build 1278 of the HCSS, equivalent to HCSS user release version 0.5.1.

- **The following major changes were made in v0.17**

- *Chapter 5*: Further feedback on example code for fitters. Added sub-section on numerical integration capabilities within DP.
- *Chapter 12*: Updates and fixes to parts of the PAL section, including local store usage. Updated database setup info for the user (requires Versant client software).
- *Appendix A*: Updated the data analysis tutorial.

Example scripts for v0.16 were tested using build 1225 of the HCSS, equivalent to HCSS user release version 0.5.0.

- **The following was changed for v0.16**

- *Chapter 2*: Extensive updates on the new JIDE console capabilities including find/replace and Goto line edit capabilities.
- *Chapter 5*: Extended documentation of examples in the chapter. Updated one example script.
- *Chapter 6*: Documented Axis class quirks and information on default property storage.
- *Chapter 7*: Updated information on Histogram plots -- new plot types available in the Image package. Also gave more examples on getting image information, e.g. sky coordinates.
- *Chapter 7*: Included image analysis task information -- aperture photometry, area histograms, 2D profiles and contouring.
- *Chapter 12*: Updates and fixes to parts of the PAL section.

Example scripts for v0.15 were tested using build 1176 of the HCSS, equivalent to HCSS user release version 0.4.3.

- **The following was changed for v0.15**
- *All chapters*: Comprehensive checks on code examples.
- *Chapter 6*: Documented several new methods of the `LayerXY` and `Axis` classes.
- *Chapter 12*: Updates and fixes to the PAL section.

Example scripts for v0.14 were tested using build 1106 of the HCSS, equivalent to HCSS user release version 0.4.1.

- **The following was changed for v0.14**
- *Chapter 2*: Documented the new `CompileAndRun` function in JIDE.
- *Chapter 3*: Modified Section 3.20 on Jython and DP quirks. Now the shorter descriptions are grouped in a single section, *Miscellaneous quirks*.
- *Chapter 4*: Added documentation on how to access the contents of `Products` in Section 4.13. Added a section on the `Dataset Inspector` and the `TablePlotter` manual (Section 4.14).
- *Chapter 5*: Added documentation on logical operators in Section 5.7. Reference to this section added to the list of quirks.
- *Chapter 7*: Fixed error in Section 7.10.2 on how to start the `Annotation Toolbox`.
- *Chapter 8*: Added a warning on the old and new interaction styles. Added features of the new interaction style in Section 8.2.2. Added Section 8.2.3.5 on how to get help on `Tasks`.
- *Chapter 12*: Added ??? on the `Pool` and `Storage Managers`.
- *Appendix D*: Naming conventions document added as new appendix.

Example scripts for v0.13 were tested using build 1023 of the HCSS, equivalent to HCSS user release version 0.3.6.

- **The following was changed for v0.13**
- *Chapter 2*: Added short section on blank line treatment in the debugger window (section 2.8.1, SPR886). Added short section on working directories and accessing files in DP (section 2.6, SPR1721). Short section added on running scripts in the background (section 2.10).
- *Chapter 3*: Added "clear" method on how to clear some or all variables from a session (section 3.3). Wrote new section: "basic programming statements" which includes subsections on `for`, `while`, `if/elif/else`, `continue`, `break` (section 3.10). Added information on formatting of printed output (section 3.11). Added material on defining user functions (section 3.12). Updated and enlarged the DP "quirks" section at the end of the chapter.
- *Chapter 5*: Implemented SPR 2304 -- removal of `INDEX` from numerics.
- *Chapter 6*: Added sections on i) creation of composite plots with new `PlotXY` API (section 6.9), ii) placing error bars on plots (section 6.5), iii) mouse interactions with plotted information (section 6.10).
- *Chapter 8*: Added section on keyword and positional parameter settings when calling tasks (section 8.2.3.6). Corrected one of the task examples.
- *Chapter 12*: Included documentation on the use of local store in the `Product Access Layer` (section 12.2.3). Updated `ProductBrowser` section to fit changed API (section 12.2.2).

Example scripts for v0.12 were tested using build 994 of the HCSS.

- **The following was changed for v0.12**

- *Chapter 2*: Removed warning about `jide_new`, updates on Help access, other minor updates about JIDE
- *Chapter 3*: Fixed typos and US spelling, added warning on loading of Jython libraries, new sections on interactivity and Jython/DP quirks
- *Chapter 5*: Implemented SPR 2183, added sections on random numbers generation and 2D fitting, updated old plot examples, updated section on matrix manipulation
- *Chapter 6*: Updated notice on new and old plot
- *Chapter 7*: Implemented SPR 2103
- *Chapter 8*: Updates on Task interaction
- *Chapter 10*: Reorganised chapter structure
- *Chapter 12*: Added section on Product Browser, updated section on database access, reorganised chapter structure
- *Appendix A*: Removed notice on new plot

Example scripts for v0.11 were tested using build 898 of the HCSS.

- **The following was changed for v0.11**

- *Chapter 12*: Minor textual corrections to the Product Access Layer Description

Example scripts for v0.10 were tested using build 898 of the HCSS.

- **The following was changed for v0.10**

- *Preface*: added note on different look & feels in screenshots
- *Chapter 2*: added notes on usage of `jide_new` instead of `jide` and on usage of debugging window. More detailed explanation of save and exit options.
- *Chapter 5*: added note on the proper usage of selection and on the difference between `max`, `MAX` and other similar Jython/Numeric function pairs
- *Chapter 6*: Updated with new plot interface.
- *Chapter 8*: Fixed broken example layout.
- Added *Appendix C* with list of Jython operators.
- *Chapter 12*: Added a description of the Product Access Layer
- Updates to plotting code in all the other chapters.

Most example scripts in v0.9.1 were "randomly" tested (i.e. most but not all) using build 848 of the HCSS. In particular, there was no systematic attempt to find out which "from" imports are necessary or redundant with startup initialization.

- **The following was changed for v0.9.1**

- *Chapter 5*: Update to UM on "and" and "&" closing AI DP-CCB-14/1.
- *Chapter 6*: Warning on use of the "old" plotting package.
- *Chapter 10*: Added xref "FITS-start" as section ID (reference from tutorial).
- *Appendix A and B*: Included Russ Shipman's tutorial on data processing as Appendix A; moved existing Appendix A to "B" position

Example scripts for v0.9 were tested using build 800 of the HCSS.

- **The following was changed for v0.9**

- *Chapter 1*: Updated installation instructions and reference platform information.
- *Chapter 3*: Expanded textual explanations and a number of examples. Added if/elif/else example.
- *Chapters 4 and 5*: Removed redundancy between two chapters.
- *Chapters 5*: Improved fitter usage explanations.
- *Chapter 6 and 7*: Fixed typos, clarified DP command usage for PlotXY and Image.
- *Chapter 8*: Tidied up Task examples to run under build.
- *Chapter 10*: Expanded FITS conversion explanation.
- *Chapter 11*: Expanded time conversion discussion. Included use of `java.text.SimpleDateFormat`.
- *Chapter 12*: Small code corrections in the examples.

Scripts for v0.8 were tested using build 766 of the HCSS.

- **The following was changed for v0.8**

- *Chapter 1*: Updated installation instructions.
- *Chapter 5*: Reduced use of lambda functions (replaced with "where"). Amplified information on how to do non-linear model fitting. Extended examples on non-linear model fitting.
- *Chapter 6*: Simplified introduction. Improved presentation of different plot initiation possibilities. Introduced `nT.quantity` library and provided URL link to nanoTITAN.com who supply this third-party library in section discussing Units in PlotXY.
- *Chapter 7*: Provided `nT.quantity` link.
- *Chapter 8*: Tidied up examples to run under build.
- *Chapter 10*: Minor changes to examples.
- *Chapter 12*: Small code corrections in the examples.
- *All Chapters*: minor typos and inconsistencies fixed, code examples adapted to newer build.

Scripts were tested using build 728 of the HCSS.

- **The following was changed for v0.7**

- *Chapter 1*: Updated JRE version.
- *Chapter 2*: Double, not single click needed in JIDE to get info about an error. Fixed. Also documented the `System.exit(1)` command.
- *Chapter 3*: Added introduction on Object Oriented Programming with brief explanation of its advantages. Fixed links to Jython and Python homepages. Fixed a couple of bugs in the `Basket` class. Added the *Useful Java bits* section with a brief explanation on frequently used Java components.
- *Chapter 4*: Fixed confusion between array data objects and array datasets.
- *Chapter 7*: Link to example JPG file moved to the beginning of the chapter.
- *Chapter 8*: Brand new chapter on the `Task` framework. Therefore Chapters 8 to 11 in version 0.6 are now Chapters 9 to 12.

- *Chapter 10*: Changed section on ASCII table import/export. Now a table is first exported and then imported, so that the ASCII file already has the correct format.
- *All Chapters*: minor typos and inconsistencies fixed, code examples adapted to newer build.
- **The following was changed for v0.6**
- *Preface*: Updated "What's New..." section. Added Editorial Board membership list.
- *Chapter 1*: Updated known installation bugs section. CLASSPATH length no longer an issue.
- *Chapter 2*: Revised section 2.8 on errors and exceptions.
- *Chapter 3*: Added a section on naming conventions (short/long names and upper/lower case) in DP.
- *Chapter 5*: Updated numerics section to include more information on linear and non-linear model fitting. Updated example to provide polynomial model fit example. Moved all "Numeric User Guide" information into User Manual (fitters, models, matrix manipulations, integral transforms).

In all sections -- updated/removed "import" commands used in examples (as appropriate). Also updated all example scripts for running with build 645 of the HCSS.

- **The following was changed for v0.5**
- *Preface*: Updated contributors and "What's New..." section
- *Chapter 1*: Added full link names to text in section 1.4
- Updated property initialization section 1.5
- *Chapter 2*: Added more information on on-line help within JIDE.
- Updated text and figures dealing with dataset and session inspectors.
- *Chapter 3*: Updated to include more on basic JIDE usage as well as components on numbers/conversions/booleans and string handling
- Added explanation of lists/tuples and the differences with numeric arrays (in chapter 5)
- *Chapter 4*: Added sub-section on row-wise appending of TableDatasets
- Short explanation Jython/Java shortcuts when discussing meta data
- *Chapter 5*: Extended section 5.4 on filtering
- Added section on array slicing
- Added section on multi-dimensional arrays
- Added section on complex arrays
- Updated scripts to new numeric scheme -- notably the use of fitting routines
- *Chapter 6*: Removed all deprecated plot mode components discussed and in examples (and it chapters which used plots for illustration).
- Updated figures of PlotXY properties windows (plot, layers and axes)
- Added section on CompositePlot -- two new figures added
- *Chapter 7*: Updated text on image import
- *Chapter 8*: Updated IO (including FITS) description
- Updated discussion of numerics

- Added descriptions of numeric toolbox sub-packages
- Small update to dataflow description (event as well as thread based dataflows)
- *Chapter 9*: Renamed to include DP variable IO
- Added section on `save` and `restore`
- Updated to include new `FitsArchive` capability
- *Chapter 11*: Updated examples to remove plot deprecations
- **The following was changed for v0.4.1**

The major difference between this and the previous version is that the source for this document is no longer in Windows Word format, but in *DocBook XML* format. This greatly simplifies the maintenance of the document e.g. now several writers can work on selected chapters concurrently. In addition it is very easy to generate different formats from this DocBook XML format i.e. XHTML for the Herschel DP Web page, `JavaHelp` for the JIDE on-line help, PDF for printing a paper version of this manual.

The main content of the User Manual is currently left untouched. There are a few minor changes thought:

- The first chapter is turned into a *Preface* which means the number of all other chapters is decreased by one.
- The Appendix B is left out since it is merely a duplication of the Javadoc for `herchel.ia`.
- At several places, though not all, the acronym 'IA' is replaced by 'DP' and the word 'jconsole' has been replaced by 'JIDE'.
- **The following was added in v0.4**
 - *Introduction*: Added full list of contributors.
 - *Chapter 1*: Changed chapter 1 to allow for description of updates. Added list of contributors.
 - *Chapter 2*: Updated installation information. Provided pointer to HCSS installation.
 - *Chapter 3*: Section 3.2.5 was added providing short descriptions on new components added to the Jconsole environment (i.e., session and dataset inspectors).
 - Figure 3-1 was updated to the new view of Jconsole and Figures 3-2, 3-3 and 3-4 were added in section 3.2.5.
 - Added section 3.7 on error and exception handling in IA.
 - *Chapter 4*: Augmented discussion on classes and methods in section 4.7.
 - Clarified last paragraph in section 4.9
 - Added section on script writing in IA (section 4.8).
 - *Chapter 5*: Changed required imports section.
 - Added components on complex and multi-dimensional datasets.
 - Basic numeric arithmetic moved into chapter 6.
 - *Chapter 6*: Changed required imports section.
 - Added basic numeric arithmetic from chapter 5.

- Added two figures illustrating fitting capabilities.
- *Chapter 7*: Updated introduction to reflect new setup of the HCSS.
- Extended PlotXY introduction in section 7.2. First example split into two.
- Added sections 7.2.1.1 and 7.2.1.2 on handling arrays and datasets in PlotXY.
- Updated all examples to present system.
- Added section 7.4.2.1 and 7.2.2.2 to better illustrate command axes adjustment.
- *Chapter 8*: Updated required imports section.
- Included new subsections on the use of each of the Image operations.
- Updated use of numeric2d arrays.
- Examples rewritten and extended to include new information on Image and Image operations.
- *Chapter 9*: Updated import information with regard to IA startup.
- Added subsections on 'inspector' and 'help' packages.
- *Chapter 10*: Updated information regarding required package imports.
- Updated introduction to highlight current FITS usage.
- Examples updated.
- *Chapter 11 (NEW)*: Chapter added on time usage within the HCSS and time conversions.
- This is based on the original user HowTo document, heavily revised.
- *Chapter 12 (previously chapter 11)*: Revised package imports needed for using databases and examples. Reworded and typo corrected sections 1 to 6.
- Significantly revised (made clearer?) sections on getting Dataframes and Housekeeping (HK) data into an IA session.
- *Appendix B*: Updated listing of classes (including links) available in IA packages.
- **The following new sections were added in v0.3.1**
 - Section 2.4.1 on updating Versant databases and schema evolution
 - Section 2.6.3 on known installation problems.
 - Updates were included in the following places
 - Section 2.5.2 Windows installation instructions updated.
 - Chapter 4 typo edits
- V0.3.1, 22 December 2004 (A.Marston)
- V0.3, 22 July 2004 (A. Marston & H. Siddiqui)

6. List of Contributors

The following people have contributed to the creation of this manual:

Philip Appleton, Jorgo Bakker, Helen Bright, Jon Brumfitt, Nicola de Candussio, Diego Cesarsky, Alessandra Contursi, Steve Guest, Rik Huygen, Juliet Kemp, Sarah Leeks, Tanya Lim, Andrea Lorenzani, Anthony Marston, Wim de Meester, Craig Porrett, Sarah Regibo, Davide Rizzo, Peter Roelfsema, Bernhard Schulz, Russ Shipman, Hassan Siddiqui, Ivan Valtchanov, Roland Vavrek, Michael Wetzstein, Ekkehard Wieprecht, Peer Zaal, Rob Zondag.

The following people comprised the Editorial Board for this edition of the User's Manual.

Anthony Marston, Philip Appleton, Bernhard Schulz, Vanessa Doublier, Markos Trichas, Russ Shipman, Ivan Valtchanov, Gillian Thornhill.

Chapter 1. HCSS Downloading and Installation



Important

In case of any problems during installation please contact the Herschel Helpdesk via the Herchel Science Centre website.

1.1. Introduction

In this chapter we explain how to download and install the Herschel Common Science System (HCSS) software. For local area networks this is likely to be done by a system manager. The system can then be run by anyone on the network. However, personal versions (e.g., for laptops) can also be set up by a user.

If you are not worried about using Versant databases (only typically for Herschel calibration scientists) for now then the Section 1.5 section is probably all you need to follow at present.

This chapter describes how to set up a basic user (or user-as-developer) HCSS environment. A key component of the HCSS is its interaction with local and remote databases storing test data and (later) observations. Upgrading your installation to allow for database interactions is discussed in Chapter 12 of this manual. Chapters 2 and 3 introduce the user to DP/Jython and do not require database interactions.

1.2. Platform

The reference platform used for Unit and System testing the HCSS software, prior to release, is now SuSE 9.1 (previously used RedHat 8.0) running on an Intel processor.

Note that this OS version is LSB (Linux Standard Base) v1.3 so theoretically one should be safe using another Linux distribution providing it has been certified LSB v1.3, see: [LSB certified products for more information](#).

1.3. Minimum System Requirements

Software can be run on a server or individual workstation running Windows XP, Linux or Solaris. The minimum recommended system is Windows/Linux 32-bit w/1GB RAM or 64-bit W/Lin/Mac w/1GB RAM; Browsers for use with the system (including download) IE 6+ , Netscape 7+, Mozilla (Firefox) 1.5+, Safari (Mac).

1.4. Pre-Installation Requirements

The following third-party software is required to be installed prior to run (or develop software for) the HCSS. In order to run all the facilities of the HCSS the necessary components are completely available with the HCSS installer.

- In many cases users will not require any additional software in order to install and run the HCSS.
- *ALL USERS*: You will need access to a Java JRE (Java Runtime Environment), which can be downloaded from the SUN web pages. A Java runtime environment is usually available as standard on most modern computer systems. The reference platform build is currently version 1.5.0_06. You can check the Java version recommended in the Reference Platform at <ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/releases/doc/refPlatformVersion> . To see which Java version is installed on your machine type the following in a terminal window:

```
>> java -version
```

- *For database usage:* Versant Database System (see notes on Versant in the full installation instructions at <ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/releases/doc/Install.html>) will need to be installed. This will allow setting up databases and accessing databases. Not needed if you are not using HCSS Versant databases. The setup and use of databases within DP is described in Chapter 12.
- *For users of TestControl:* If you are using HCSS in a Herschel instrument testlab environment for ILT/AIV tests then you will also need TclBlend. This can be downloaded from: <http://sourceforge.net/projects/tcljava>
- For users who want to become involved in the development of HCSS, the following should be installed. Note that development of DP/Jython scripts can be done with the HCSS Users software needs noted above.
 - Java JDK (Java Development Kit), which can be downloaded from: <http://java.sun.com/j2se>
 - Versant Database System (see notes on Versant below)
 - JavaCC, which can be downloaded from: <https://javacc.dev.java.net/servlets/ProjectDocumentList>.
 - CVS (client/server version), which can be downloaded from: <http://ccvs.cvshome.org/servlets/ProjectDownloadList>
 - TclBlend (only needed if you are developing the TestControl package), which can be downloaded from: <http://sourceforge.net/projects/tcljava>.
 - Together (optional), can be downloaded from: www.borland.com/together.



Note

the exact version numbers of the applications listed above, can be obtained from: <ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/releases/doc/refPlatformVersion>



Warning

Please note that you may/will need system administrator support and/or privileges in order to install one or more of the component(s) listed above.

All other third-party libraries required (see the HCSS reference platform specification for a complete list), can be redistributed and are included with the HCSS installer.

For those who are considering HCSS development, the full third-party packages may be required (including its Javadoc, sample code, etc.). A compressed TAR-file containing these libraries (matching the latest reference platform set) can be downloaded from the HCSS ftp area: <ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/refPlatformDownloads>. Alternatively you can download all libraries from the supplier site (most of the URLs can be found in the HCSS reference platform specification, <ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/releases/doc/refPlatformVersion>).

You must now configure your environment to include the above listed packages in your PATH and CLASSPATH environment variables, following the installation instructions provided by the suppliers. In addition, developers should include the JavaCC library 'javacc.jar' in their CLASSPATH, because of the way that the HCSS 'jake' tool invokes JavaCC.

1.5. User Installation Procedure

Installation of the HCSS/DP system is relatively straightforward and has recently been simplified for both UNIX and Windows users. It can installed with a software installer (see Herschel Science Centre

website, or HCSS installer script). Software can be run on a server or individual workstation running Windows XP, Linux or Solaris. The minimum recommended system is Windows/Linux 32-bit w/1GB RAM or 64-bit W/Lin/Mac w/1GB RAM; Browsers for use with the system (including download) IE 6+ , Netscape 7+, Mozilla (Firefox) 1.5+, Safari (Mac). The system is Java based, and indeed general Java scripts can be run on the system. Installation instructions are provided at the bottom of the FTP page.

Once the software is installed, HIPE or JIDE can be started by several means. Under Windows, Herschel software can be started under the "Start" menu after a standard installation. Alternatively, HIPE or JIDE can be started from a command line, e.g.,

```
$ jide  
or  
$ hipec
```

If the installation is done by the installer then the user at the end of the installation receives a message informing her/him where the applications reside on their computer and the links to the applications.

The maximum java size for an application can be set via an option in the expert panel of the installer. the downside of this implementation is that it can only be set once. A more flexible solution is envisaged in the future.

Once downloaded, some environmental properties need to be set up. This is now handled automatically by the DP installer.

For an up-to-date list of installation problems please see

<ftp://ftp.rssd.esa.int/pub/HERSCHEL/csd/releases/doc/Install.html#KnownInstallationProblems>

1.6. DP Property Initialisation

Standard user properties are set up when the DP system is installed. However, the HCSS environment that has been set up can be configured to user specifications. This can, for example, change the database being used for interactions or change the memory allocation to JIDE and HIPE (the prime interfaces for running the HCSS and DP). For those new to the HCSS it is not necessary to adjust these properties unless database interactions are to be immediately attempted. Later, with more sophisticated interactions, users will want to make changes to their properties. Storage of user properties is in the `.hcss/myconfig` file. Changes can be made to properties while working within the HCSS - no restart is required for the updated properties to be made available. This can be useful when, for example, you are changing the database with which you wish to work.

Properties can be set in the `$HOME/.hcss/myconfig` (under the Windows systems properties are held in the file `hcss.props` within the user's home directory) file with the use of the HCSS tool "Property Generator". Use the following command to initiate the tool (also see the property generator user manual).

```
propgen
```

For the most part, system default values should be adequate for most users. A list of these properties is available at ftp://ftp.rssd.esa.int/pub/HERSCHEL/csd/releases/doc/Release.html#A_List_of_user_properties_in_HCSS. Property setting allowing the use of databases is discussed in Chapter 12

After the initial download, the Property Generator tool is useful to run every time you download and install a new build, as it will inform you of added properties that are not defined in your property files.

Chapter 2. Using JIDE or the JIDE View in HIPE

2.1. Introduction

A DP session involving scripting is typically initiated within a console window of HIPE or JIDE. This window includes help and history for the session. Individual commands can be input to the console using DP/Jython commanding, which is discussed later in this chapter. Alternately, the console and associated editor window allow for the construction and running of complete algorithms based on the Jython language or even sections/individual lines of algorithms. Since no separate compilation is required, individual lines or sections of algorithms can be checked for validity very quickly. DP scripts that use GUIs can also be started from within the HIPE/JIDE view. *Example HIPE/JIDE input code is provided throughout the text in shaded boxes. Comments on the code and, frequently, example output are provided within the boxes on lines preceded by the "#" mark.*

In this chapter we discuss how to start working in the DP console view of HIPE or JIDE. We provide some simple DP interactions to illustrate its use. We discuss some more detailed DP capabilities in Chapter 3.

2.2. DP Scripting Using the Editor View of HIPE

HIPE has a full set of abilities that is described at the beginning of the "DP HowTo's" manual. A similar perspective can be obtained by selecting the JIDE perspective from HIPE (see "DP How To's Manual" introduction to HIPE). In describing the use of the DP system for more advanced, scripting purposes with HIPE we will concentrate on the Classic(JIDE) perspective available within HIPE.

The user can start the HIPE console can be initiated from the computer "Start" menu after installation using the HCSS installer.

Alternately, it can be started at a command window prompt.

```
$ hipe
```



Note

For Windows users, open a command window and type in the same thing, or execute `hipe.bat` from the `bin` directory of your HCSS build.

Starting HIPE does the following:

- Loads a customised DP environment (imports a set of libraries and defines a set of variables).
- Keeps a history of successful DP statements.
- Implements a set of basic editing functions (copy, paste, find and replace).

On startup, HIPE displays a Welcome window. From the initial HIPE Welcome window the user should select the "Classic(JIDE)" icon at top right of the screen ().

Alternatively, we can obtain the "Classic(JIDE)" perspective for doing more advanced scripting work by going to the "Show Perspectives" area of the "Window" pull-down menu at the top of the screen (see Figure 2.1).

Figure 2.1. Selecting the Classic(JIDE) perspective in HIPE.

This provides a perspective of three windows with an Editor view to the top of the screen, a Console view towards the bottom left and a History view towards the bottom right (see Figure 2.2).

Figure 2.2. The Classic(JIDE) perspective in HIPE.

The bar at the bottom of the perspective shows the amount of allocated memory used by the session. As memory usage increases the bar will turn from green to yellow and then to red. Finally, note the indicator in the right corner which will show a rotating set of emphasized dots during periods when a DP command is being performed.

An interactive **Console view** is given to bottom left of the view with a customizable "IA>>" prompt. Individual DP commands can be run here. Click in the bottom left window with your mouse, then type in

```
print 5 + 3
```

Followed by **Enter**. The answer should be provided on the next line, prior to receiving the "IA>>" prompt back again:

```
IA>> print 5 + 3
8
IA>>
```



Note

In a plain Python or Jython console it would be enough to type "5 + 3" followed by the **Enter** key to get the result. In DP we have to use the `print` keyword, otherwise we would get no output.

The bottom right of the perspective contains a **History window** that lists the commands (including those inside algorithms) used in the current session. Any command highlighted by a red cross next to it caused an error. Some information on the error that occurred can be obtained using the mouse to click on the command highlighted. A response with the error is shown in the traceback column of the History window. Try the following

```
sign 5
```

After hitting **Enter** the user will see the history window has a command highlighted by a red cross next to it. Click on this using the left button of the mouse. This then expands the information on the error incurred.

The top pane of the perspective is available for the user to develop his/her own scripting algorithm using the available DP commands.

In order to start scripting in this pane, go to the "File" menu and pull down to "New" -> "Jython script". This will produce a white screen that allows input of DP commands that can be formulated into scripts (see Figure 2.3).

Figure 2.3. The Classic(JIDE) with script screen made available.

Click in this window, type in a similar print command to the above example. Hitting return will not run this simple script. To run the one line, click in the grey margin to the left of the line you have typed. An arrow should appear beside the line. Now go to the line of icons at top left of the HIPE screen and click on the single arrow (). This will run your one line algorithm and the result will appear in the lower left command line window (again). If you want to "print" a string it needs to be in quotes (e.g., print "Hello World").



Note

The top pane is not meant to be a fully-fledged text editor, nor a sophisticated IDE (Integrated Development Environment). It offers basic editing and debugging capabilities

for developing simple scripts, but larger projects should be developed in external tools and then loaded into the window for execution.

Now that we have a brief introduction to the three windows of HIPE Classic(JIDE) perspective we will consider each of the menu and icon items in turn.

2.2.1. File Menu

Only one of the **File menu** items has an associated icon (the "Save" capability).

New creates a new window for algorithm development ("Jython script") or text ("Text file") in the top "Editor" view of HIPE (note that a new "Tool" window feature is yet to be developed).

Open File allows a file to be opened in the Editor that is chosen from anywhere within the system (ASCII - DP script files are typically stored with the suffix .py, in ASCII format). If the suffix is .py the window is always a Jython script window -- otherwise a text window.

Close closes the current window shown in the Editor view. **Close All** closes all the windows showing in the Editor view.

Save and **Save As** for saving the current algorithm shown in the top window. The "Save" capability is also available using the icon shown in the line of icons to top left of the HIPE window.

Revert Reverts back to the original version of the file currently being edited.

Refresh this IS NOT FOR USE WITH THE EDITOR. This capability is for the Navigator view available in HIPE. The Navigator view automatically updates every 5 seconds so that new/changed files in the computer system (e.g., copied files) are made available in the Navigator view of HIPE. Hitting F5 or "Refresh" does this instantaneously.

Rename this IS NOT FOR USE WITH THE EDITOR. This capability is for the Navigator view available in HIPE. It allows the renaming of a highlighted file showing in the Navigator view of HIPE.

Print prints text of HIPE Editor session to a printer (various page types and setups) or postscript file.

Exit exits from the HIPE session. For any unsaved changes to any of the files showing in Editor windows the user is given the opportunity to accept or reject changes before HIPE is closed down.

2.2.2. Edit Menu

Most of the Edit Menu functions (except Cut, Copy, Paste and Open) have an associated icon at the top of the HIPE panel. The associated shortcut icons are shown next to the function name in the menu. Each function also has an (standard) associated CTRL combination (except for Open and Open With). See Figure 2.4.

Figure 2.4. The Classic(JIDE) with script screen made available.

Undo (CTRL-Z) and Redo (CTRL-Y) and allows edits (cut/paste or deletion from the keyboard) to be undone or redone.

Cut (CTRL-X), Copy (CTRL-C) and Paste (CTRL-V) These provide the usual cut, copy and paste facilities, using the mouse to select and position text in the Editor window.

Open (enter key), Open With, and Delete (delete key) these are NOT FOR USE WITH THE EDITOR. This capability is for the Navigator view available in HIPE. It allows the highlighted file in the navigator view to be opened in the HIPE Editor view -- as Jython Script, text editor (default for Open) or File Overview (gives size/type of file info), or delete the highlighted selection from the system.

Find/replace (CTRL-F) does the usual find and replace of text within the current window of the HIPE Editor view.

Go to Line (CTRL-L) allows the user to go to a specified line number.

2.2.3. Run Menu

The Run Menu items all have associated icons at the top of the HIPE window.

Stop (ALT-T) - stops a script being executed. Click on this button or choose **Stop** from the pulldown menu to stop the execution of a script before it reaches the end. Note that this icon is greyed out when there is no script in execution.

Run (ALT-U) - runs a single line or logical block of a script. A selected set of lines can be highlighted using the mouse and these can be executed by then clicking the Run button or selecting Run from the menu. The lines are iterated to the console window and their status shown in the History window to bottom right. While running, the red stop button is lit.

Run all using pulldown or icon, this allows all DP commands in the current Editor window of HIPE to be run in sequence. The lines are iterated to the console window. The stop button turns red while running.

2.2.4. Exiting HIPE

To exit go to **Exit** under the **File** menu of HIPE. For any unsaved changes to any of the files showing in Editor windows the user is given the opportunity to accept or reject changes before HIPE is closed down.

2.2.5. Window and Help Menus

The "Window" menu allows access to HIPE perspectives (such as the Classic(JIDE) discussed here) and views. There are a number of views available which are discussed more extensively in the "DP HowTo's" document. By selecting one of the offered views an extra panel is added to your HIPE perspective. For example, in Figure 2.5 the Navigator view showing the available directories and files on your system is added in a panel to the right on the HIPE screen.

Figure 2.5. Adding the Navigator view to the Classic(JIDE) perspective in HIPE.

The "Help" menu, in addition to providing access to Help inside of HIPE (together with Help search facilities) also provides "About" information on HIPE and access back to the Welcome page that you get on starting up HIPE.

2.3. DP Scripting Using JIDE

DP users who wish to do scripting may choose to work within DP JIDE separately from HIPE. After installing the HCSS (see Chapter 1), the user can start the JIDE console can be initiated from the "Start" menu after installation using the HCSS installer.

Alternately, it can be started at a command window prompt.

```
$ jide
```



Note

For Windows users, open a command window and type in the same thing, or execute `jide.bat` from the `bin` directory of your HCSS build.

Note that some feedback from the DP session is provided to the terminal window from which it was started. This includes information on the settings used on JIDE startup and information on database access (basically feedback on where interactions occur with systems outside the immediate DP session). The JIDE shell performs the following tasks:

- Loads a customised DP environment (imports a set of libraries and defines a set of variables).
- Keeps a history of successful DP statements.
- Implements a set of basic editing functions (copy, paste, find and replace).

It is an extension of the standard Jython shell. Here, we provide some basic startup information.

If entering the JIDE command from a terminal window, information on preloaded elements in the DP session appear in the terminal window. Startup from the "Start" menu goes directly to the following. After any feedback, a separate three-paned console window should appear (see Figure 2.6). The bar at the bottom of the window displays the amount of memory used by the session: in the case of Figure 2.6 we are using just three per cent of the available memory. As memory usage increases the bar will turn from green to yellow and then to red. Finally, note the clock at the lower right corner.

Figure 2.6. The JIDE window set-up.

The JIDE window has three components. An interactive **command line/console window** is given to bottom left of the view with a customizable "IA>>" prompt. Individual DP commands can be run here. Click in the bottom left window with your mouse, then type in

```
print 5 + 3
```

Followed by **Enter**. The answer should be provided on the next line, prior to receiving the "IA>>" prompt back again:

```
IA>> print 5 + 3
8
IA>>
```



Note

In a plain Python or Jython console it would be enough to type "5 + 3" followed by the **Enter** key to get the result. In JIDE we have to use the `print` keyword, otherwise we would get no output.

The bottom right of the console contains a **command history window** that lists the commands (including those inside algorithms) used in the current session. Any command highlighted by a red cross next to it caused an error. Some information on the error that occurred can be obtained using the mouse to click on the command highlighted. A response with the error is shown in the command line window to bottom left. Try the following

```
sign 5
```

After hitting **Enter** the user will see the history window has a command highlighted by a red cross next to it. Click on this using the left button of the mouse. This then expands the information on the error incurred.

The top pane of the console is available for the user to develop his/her own algorithm using the available DP commands. Click in this window, type in a similar print command to the above example. Hitting return will not run this simple script. To run the one line, click in the grey margin to the left of the line you have typed. An arrow should appear beside the line. Now go to the line of icons and



click on the single arrow (). This will run your one line algorithm and the result will appear in the lower left command line window (again). If you want to "print" a string it needs to be in quotes (e.g., `print "Hello World"`).



Note


The top pane is not meant to be a fully-fledged text editor, nor a sophisticated IDE (Integrated Development Environment). It offers basic editing and debugging capabilities

for developing simple scripts, but larger projects should be developed in external tools and then loaded into JIDE for execution.


Now that we have a brief introduction to the three windows of JIDE we will consider each of the menu and icon items in turn.

2.3.1. File Menu

Each of the **File menu** items has an associated icon except for exit. These are the first 5 icons on the bar under the menu headings.

New  creates a new window for algorithm development. New history and/or command line windows are not created.

Open allows a file to be opened in the top window (ASCII - DP files are stored in ASCII format).

Save and **Save As**  for saving the current algorithm shown in the top window.

Close closes the file in the top window pane. Only closes the window showing the current algorithm.

Print prints text of JIDE session to printer or postscript file.

Screenshot as JPG creates JPG file of screen view.

Screenshot as PNG creates PNG file of screen view.

Exit exits from the JIDE session.

2.3.2. Console Menu

Execute in the console requests the input of a DP script file, loads it and runs it inside of JIDE.

Execute does a similar thing, except it runs the whole script on the system rather than using the JIDE console

Execute in the background does the same as Execute, but runs the script in the background.



Save history and **Save history as ...** saves a history of successful JIDE commands from this session.


2.3.3. Edit Menu

Each of the Edit Menu functions (except Goto) has an associated icon at the top of the JIDE panel (middle section of icons).

Import history allows the import of the history of a saved JIDE session.

Undo and redo and allows edits (cut/paste or deletion from the keyboard) to be undone or redone.

Cut and paste  **and**  the usual cut and paste using the mouse to select and position text.


Find/replace  does the usual find and replace of text within the upper window of the JIDE console.


Goto allows the user to go to a specified line number.


2.3.4. Run Menu


The next four icons at the top of the JIDE window relate to the **Run menu**.

Script mode This only appears in the Run Menu. The default is that the script mode is disabled, the Run, Run selection and Run all buttons then work as if on the command line for lines of code written in the debug window and the commands are reiterated to the console. In script mode, only requested output (e.g., from a "print" command) will have output sent to the console.

Stop  - stops a script being executed. Click on this button or choose **Stop** from the pulldown menu to stop the execution of a script before it reaches the end. Note that this icon is greyed out when there is no script in execution.


Run  - runs a single line or logical block of a script. The line is iterated to the console window, unless in script mode (see under "Run Menu") when only explicit outputs from script commands appear at the console. In script mode the button turns red.

Run selection  select a set of commands by dragging the mouse over them. Pull down to **Run selection** (or click the icon) to run these DP commands only. The lines are iterated to the console window, unless in script mode (see under "Run Menu") when only explicit outputs from script commands appear at the console. In script mode the button turns red.

Run all  using pulldown or icon, this allows all DP commands in the top pane of JIDE to be run in sequence. The lines are iterated to the console window, unless in script mode (see under "Run Menu") when only explicit outputs from script commands appear at the console. In script mode the button turns red.

2.3.5. Help Menu

The last four icons at the top of the JIDE window relate to various forms of help that are also available under the **Help** pulldown menu.

Dataset Inspection  allows the user to view datasets (notably tables) currently available in the DP session in a separate `Dataset Inspector` code window (see Figure 2.7). Since the Dataset Inspector involves advanced concepts like Products and Table Datasets, a detailed treatment is deferred until Section 4.14.

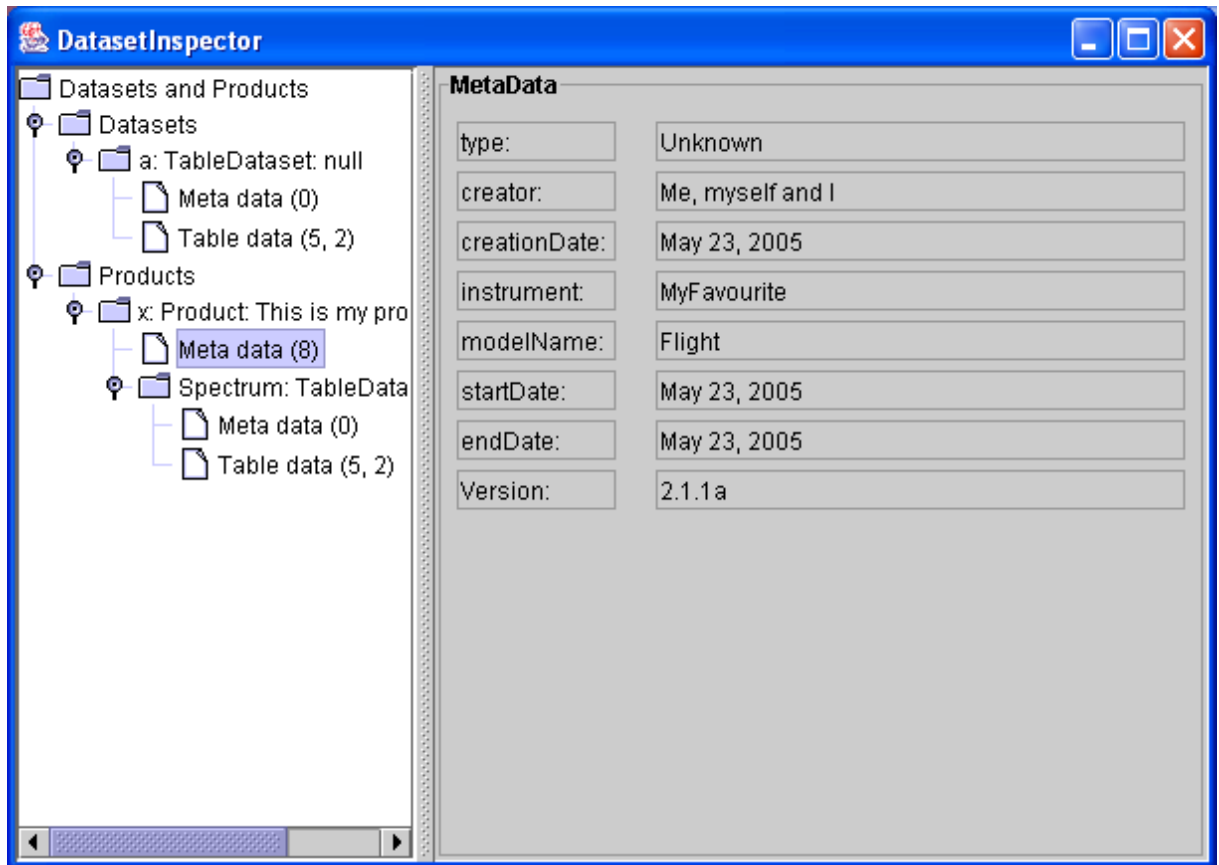



Figure 2.7. The Dataset Inspector window

Session Inspection  allows the user to view the classes (programs) and functions available in the current DP session. Also allows the user to inspect all variables used in a session. See Figure 2.8. Further classes and functions can be made available by importing "packages" (see Chapter 7).

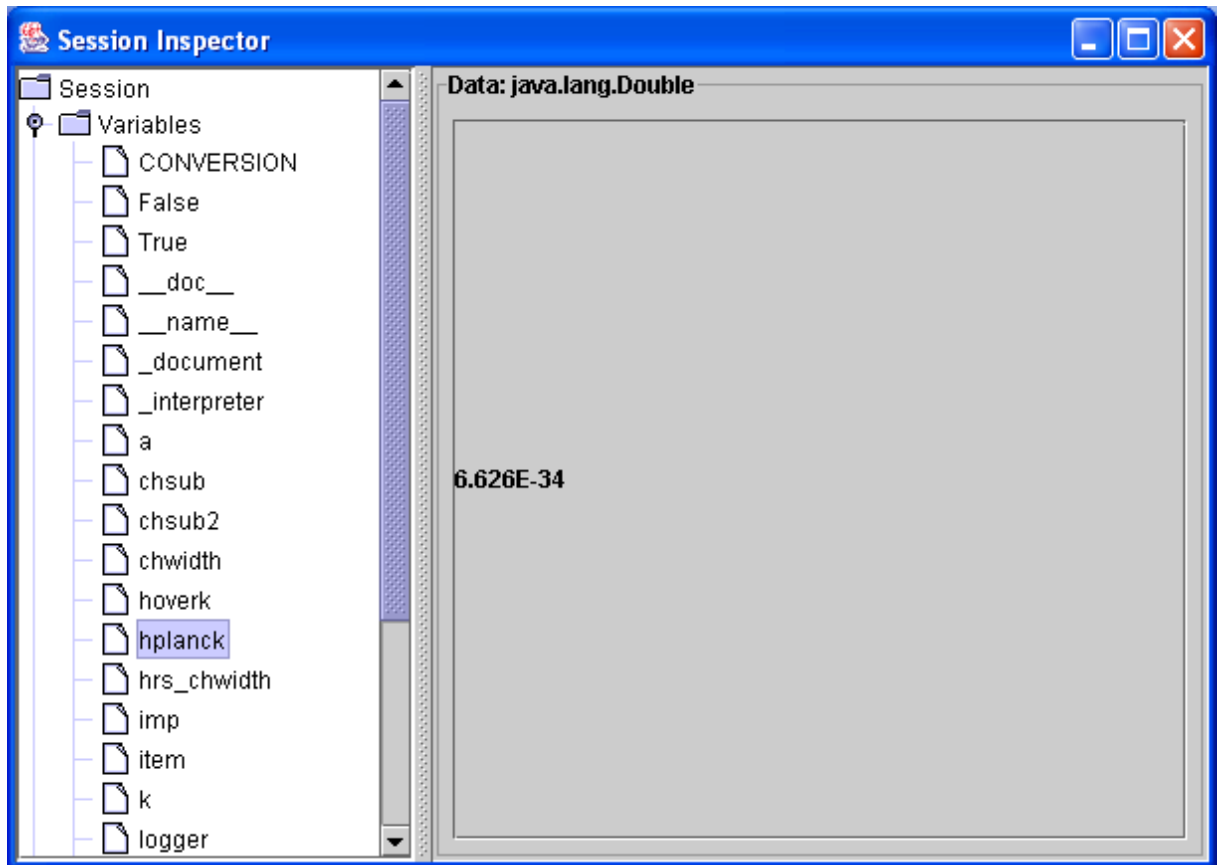



Figure 2.8. The Session Inspector window

 **Log Window** provides a listing of the feedback from running commands in the system, including error messages. These appear in a separate Log window. The log can be saved when exiting from JIDE.

 **Access to On-line Help Documentation** clicking this icon allows access to full set of current (website) documentation in a separate window. See Figure 2.9.

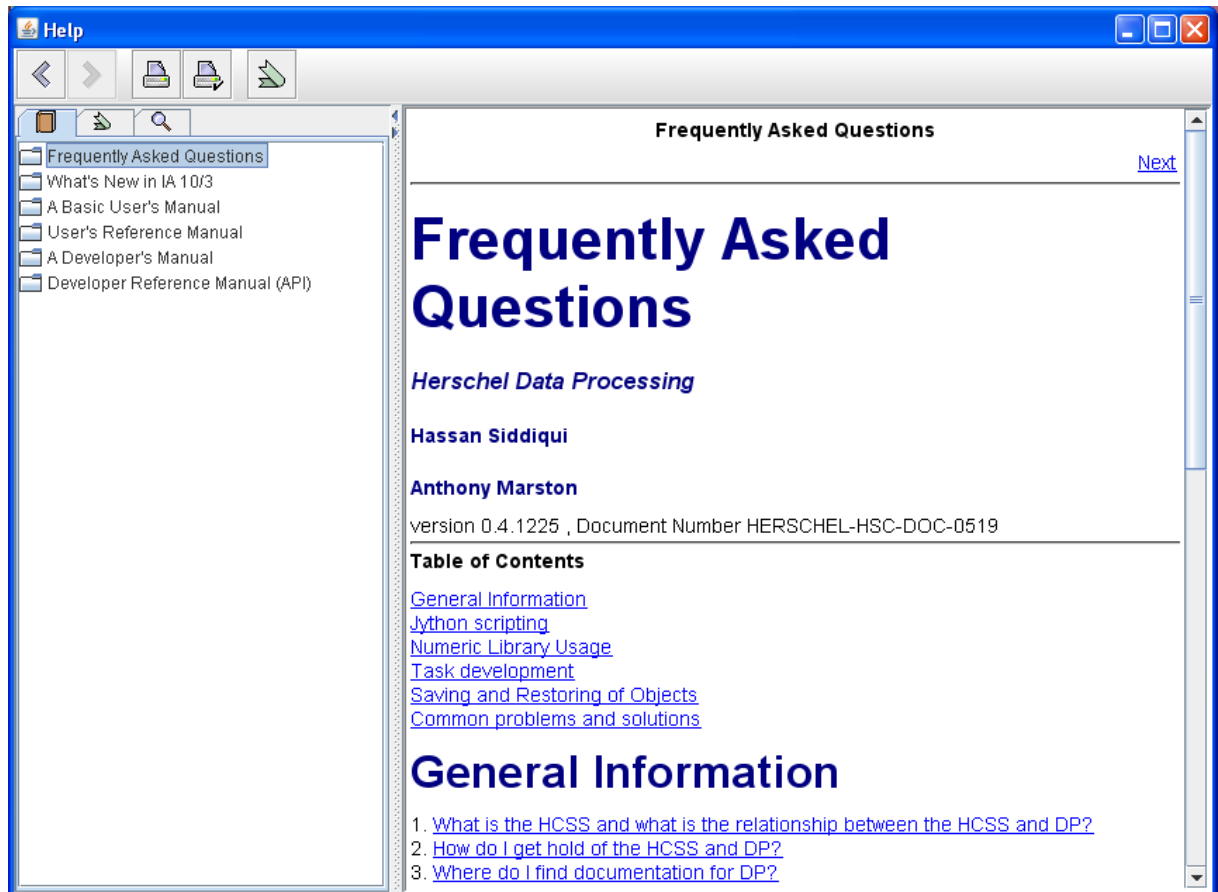


Figure 2.9. The JIDE Help window

For the Help window there are tabs at the top of the left hand column. These provide, from left to right, a table of contents for help, an index of help documents in alphabetical order, a listing of favourites and a help search capability.

The main help documentation folders are included on the left hand side of the panel. These include access to this User's Manual, the User's Reference Manual, and the equivalent for developers. Clicking on the folders expands these so that individual chapters or sections can be selected.



Note

The main difference between the User's Manual and User's Reference Manual is the fact that the User's Manual attempts to guide the user through linked DP commands typically found in a user's data processing session, while the User's Reference Manual mainly contains the listing of available commands and their usage. For a listing of available DP commands (including any JTasks the user has imported -- see Chapter 8) go to the *DP Commands* section of the User's Reference Manual. Commands are placed in alphabetical and task type order.

2.4. Quitting JIDE

We already know that the Exit entry in the File menu can be used to quit JIDE. In this case a new window appears, prompting the user to save the current work (scripts and command history). You will get a list of all unsaved files, together with entries like

- [New-1]: -no file associated-. This is a script that has not been saved yet (beware that it could be an *empty* script).
- [History of Console1]: -no file associated-. This is the history of the commands you have issued, listed in the lower right panel. Useful if you want to save to a script what you have typed.

To select an item click on it. You can select multiple items by holding **Ctrl** while clicking on them; if they are contiguous you can select them in one go by clicking on the first one and then clicking on the last one while holding **Shift**.

Below the list of unsaved items there are four buttons: Select all to select all the items, Save Selected to save the selected items, Cancel to go back to JIDE without quitting, and Close to quit JIDE.

After pressing Close, a second confirmation window is displayed. Click Yes to quit or No to go back to JIDE.

An alternative way to quit is to type `System.exit(1)` at the `IA>>` prompt and press the Enter key. This command can also be added to a script (for more information about writing scripts, see Section 3.16).



Warning

The `System.exit(1)` command causes the current JIDE session to terminate immediately. All unsaved work will be lost.

2.5. Standard Settings for JIDE and HIPE

JIDE and HIPE come with a memory specification that is dependant on the installer information supplied by the user when setting up the system initially. The settings are specified in the startup script for JIDE. This script is located in the `$HCSS_DIR/bin` directory (named `jide.lax`). These settings can be modified by editing this JIDE startup script. The following two lines adjust the initial and max memory allocations.

```
lax.nl.java.option.java.heap.size.initial=134217728
lax.nl.java.option.java.heap.size.max=536870912
```

A similar `hipe.lax` file has the same editable lines. Make sure that the environment variable `HCSS_PROPS` is properly defined (see Chapter 1).

Make sure `HCSS_PROPS` contains the specification of the standard `var.hcss.dir` property (this should be the property defined in your `$HOME/.hcss/myconfig` file IF you have set up your own environment and are not using a local network installation or an installer). And be sure that `var.hcss.dir` points to the HCSS build directory. You can check any property with a command such as the following in the Console area.

```
print Configuration.getProperty("var.hcss.dir")
```

There are several properties for JIDE and HIPE that are set up during initialisation (for example, see under Set Up in the JIDE HowTo document). These can be used to determine such things as window size. However, window size can be adjusted in the usual fashion by clicking and dragging corners and/or sides of the JIDE or HIPE window.

2.6. DP working directory and file access

The current working directory of DP is the directory from which JIDE/HIPE was started. Jython has some limitations, inherited from Java, with regard to navigation of the underlying operational system. *So there is no method to change the current working directory.* The user is advised to start JIDE/HIPE from a directory where he/she is going to read/write files by default and to use absolute paths for the file names.

When using "Save" under the File menu of JIDE/HIPE the user can specify any directory.

A view of the current directory contents is available through the HIPE navigator view. Such a view is not possible with JIDE. Opening a file in either HIPE or JIDE under the "File" menu does allow a view of the available files in any directory on the system.

It is possible to print the file contents of the current working directory using the following in a console window.

```
import os

# print the working directory
print os.getcwd()
# print the names of the files in the working directory
print os.listdir(os.getcwd())
# any directory name can be placed in the brackets
```

This provides an unsorted listing of all files and directories in the working directory. If the user wants to filter the file list, e.g. to select only the fits files, then a *glob* module can be used with search pattern following the UNIX shell rules, i.e. "*", "?", "[" etc which are interpreted in the same way as in the UNIX shell.

```
import glob

ffiles = glob.glob("*.fits")
# or even more elaborate example to provide the list of all fits file
# in a given directory and perform some action on them
ffiles2 = glob.glob("/home/user/scratch/fitsfiles/*.fits")
fits = FitsArchive(reader = FitsArchive.STANDARD_READER)

for fi in ffiles2:
    product = fits.load(fi)
    # do something on the products, like print the dimensions
    print fi, product.default.data.dimensions
#
```

2.7. Getting Command-line Help in JIDE or HIPE

Further help in JIDE or HIPE is available through command-line interaction. There are two methods.

- `help()` -- which provides an overview of the help system via a separate popup window (see Figure 2.10). The window also includes all documentation provided by each of the instruments, for specific aspects associated with handling instrument information, providing more specialised documentation.
- In HIPE, selection of help through any button marked provides access to Help that is shown in a browser. Search and full Help document selection is available through this system.

Figure 2.10. The online `help()` popup window

2.8. Programming Loops in JIDE and HIPE

Earlier in the chapter we tried some basic commands to illustrate the components of the HIPE and JIDE windows. One particular capability of JIDE and HIPE is allowing block support for DP coding. Suppose we want to take a basic print command typed in the command line window.

```
a = 5 [Enter]
```

```
print a [Enter]
# 5
```

Now simply input (the [Enter] means you have to press the enter key on your keyboard)

```
for i in (1,2,3): [Enter]
```

This will return a response in the command line. Note that the colon at the end of the line is important for starting the block. The command is incomplete. Input a `print i` command *indented by at least one space*. A further is returned. Hit **Enter** once more, the command is now complete.

The whole session should look like (again, note the indent prior to the print statement on line 2):

```
for i in (1,2,3):
... print i
....
#1
#2
#3
```

We could have added a number of commands to this `for` loop. The block statement continues until a blank line is produced. The history of the window is now available. The up arrow will provide the previous command, which can then be edited as desired and re-entered

```
for i in (1,2,3):
    print i
```

You can edit this block statement in the bottom left panel of JIDE by using the LEFT and RIGHT keys (not UP and DOWN, these are used to move through the history) and deleting/adding characters.

Blocks within blocks (nested `for` loops or `if` statements) are also possible. Basic rules about the use of blocks follow Jython language syntax.

- Each statement in a block must begin in the same column;
- Each of the DP key statements and clauses (**class**, **def**, **for/else**, **if/elif/else**, **try/except/else**, **try/finally** and **while/else**) denotes the beginning of a new block;
- A new block must be indented at least one space from the enclosing block;
- The end of a block is marked by having the next statement begin in the same column as the enclosing blocks.

For example

```
for x in (1,2,3):
    print x # outer block
    for y in (4,5,6):
        if y == 5: # inner block
            print y # inner-inner block
        print x*y # inner block
        # insert inner block statement here
    # insert outer block statement here
```

As usual, end with a blank line! Note the end of each `for` loop is determined by where the indentation ends.

2.8.1. Loop Performance on Arrays

Numeric Arrays are discussed in Chapter 4 of this manual. But we mention here how loops can be computationally expensive when used with numeric arrays in the system.

In performance checks using the HCSS timing differences for standard operations (e.g., division and multiplication many times on arrays) are found to be very similar to using similar programming languages such as Python. However, Jython/HCSS loops can be slow and for large computations this can become very inefficient for the user.

One means of reducing quite significantly the computation time for simple arithmetic computations on arrays is to use the ability of the HCSS language to do in-line calculations. For example:

```
z=Double1d(x.size) # create a 1d numeric array of the same size as an original
                  # array called "x"
for i in range(1000):
    z.set(x)      # assign, not allocate
    z-=y         # inline subtraction
    z/=c         # inline division

# instead of the following -- which is much slower
for i in range(1000):
    z = (x-y)/c
```

For large operations this can reduce computation time by nearly an order of magnitude.

Some further advanced tips to improve performance are provided in Section 5.8.

2.8.2. Using the Editor view with loops

The top edit window of JIDE and Editor view of HIPE can be used to keep lines of code in your session. To run things in this window we have three "arrows" at the top of the JIDE screen (two in HIPE). The single arrow on the left of these will run things as if you were putting them on the command line. So if we have a "for" loop a blank line will stop the loop. However the middle arrow (runs a highlighted section of code -- incorporated into single arrow also for HIPE) and the double arrow (which runs everything within the currently opened edit window) run commands within the whole group in the editor window and ignores blanks. For example, we may consider the following lines of code.

```
for i in range(4):
    if i > 0:
        print i

    j=i
    print j-i
print "Finished"
```

If run line-by-line (mouseclick to produce arrow next to the "for i in range(4)" line -- then hit the single arrow at the top of JIDE or HIPE) then only the first loop is run before a blank line is encountered. If the double arrow is used then the blank is ignored and the whole thing is run.



Warning

This means that the way blank lines are treated depends on how the DP code is run. Your code will run differently if you run it line-by-line as compared to running it as a complete script.

2.9. Multiline Statements in the Console View of HIPE or JIDE

Another improvement of JIDE/HIPE compared to other Jython interpreters is that it allows multiline statements. The backslash (\) character at the end prevents execution of the line when hitting Enter and the statement can be continued.

The following example breaks up a longer definition of a tuple into three lines:

```
IA>> a = ("meaning", "of", "life", \
... "shrubbery", "killer rabbit", \
... "holy hand grenade")
IA>> print a
('meaning', 'of', 'life', 'shrubbery', 'killer rabbit', 'holy hand grenade')
IA>>
```

Note that the backslash initiates a continuation mode. The mode is left upon hitting Enter after the first line without backslash, and the entire line is executed.

2.10. Pausing during script execution and debugging in JIDE (ONLY)

A script may be paused at any point using the `pause()` command. This allows values to be changed while a script is paused in the "Debug window". See the following example script.

```
from herschel.ia.jconsole.util import * # import pause
def test(arg=1):
    a=12
    for i in range(arg):
        pause() # pause here, change of a within the debugger is allowed !
        a=a+i
        print a
        pause() # and here
        print a

test(10) # run the example
```



Warning

This feature DOES NOT WORK IN HIPE RIGHT NOW and causes an error.

Once the change has been made in the "Debug window" use the "console" menu in the "Debug window" to scroll down to "Resume" to continue the script.



Note

Note that this should only be done in JIDE. This capability is not available in HIPE.

2.11. Background script execution in JIDE and HIPE

There are two ways to run time consuming scripts in background. One is from the drop-down menu under "Console" -> "Execute in background" which executes, in the background, the script which is loaded in the JIDE editor window. This is not available in HIPE.

The other method is by using the `execfile` capability, e.g., `bg('execfile("script_name.py"))` from the JIDE or HIPE command line. Print statements are redirected to the console and can be used to monitor the state of the execution.

Statements passed as parameters to the function are evaluated in the global namespace therefore the following example is legal:

```
IA>> a = 5
IA>> bg('execfile("print a"))
```

```
IA>> bg('execfile("a = anExtensiveComputation(12)"))')
IA>> bg('execfile("b = aComputation(a)"))')
```

There is no guarantee however that the last statement will be executed after the preceding returns the value and that uncertainty can easily lead to cases where "aComputation" is run passing the value 5 (the first assigned to a) or the value returned by "anExtensiveComputation(12)". This is unpredictable and should be carefully avoided.

2.12. Running Scripts from a Shell Command Line

it is possible to run user-created DP scripts from the command line of a shell window using the `jylaunch` command.

The following line at the command prompt can be run from a shell.

```
> jylaunch myscript.py
```

where, of course, `myscript.py` should be replaced with the filename of the script you want to run.

The `jylaunch` command can also be run from the Start menu for the 'hcss' provided by the HCSS installer script.

With the use of the HCSS installer, the `jylaunch` capability is also available under the Program Files start menu as a stand-alone task.

2.13. Errors and Exceptions in DP

Here we explain how errors are generated within DP and how these are reported back to the user. Following from this the user should be able to:

- understand error messages that might show up (i) while running an application, or (ii) during a DP session.
- report the error to the custodian of a HCSS module in case a badly described exception occurred, i.e., one which cannot be handled by the user.

2.13.1. Overview of the Libraries Used in a DP Session

The base routines for DP are written in JAVA, but DP user development uses the more friendly Jython. Typical user development is expected to take place in the console panel with plots and images appearing in separate windows. Within a DP session one can run commands from the JIDE tool that enables the execution of DP/Jython commands, saves and loads scripts, and provides command history support. This tool often provides the core of a user's DP session.

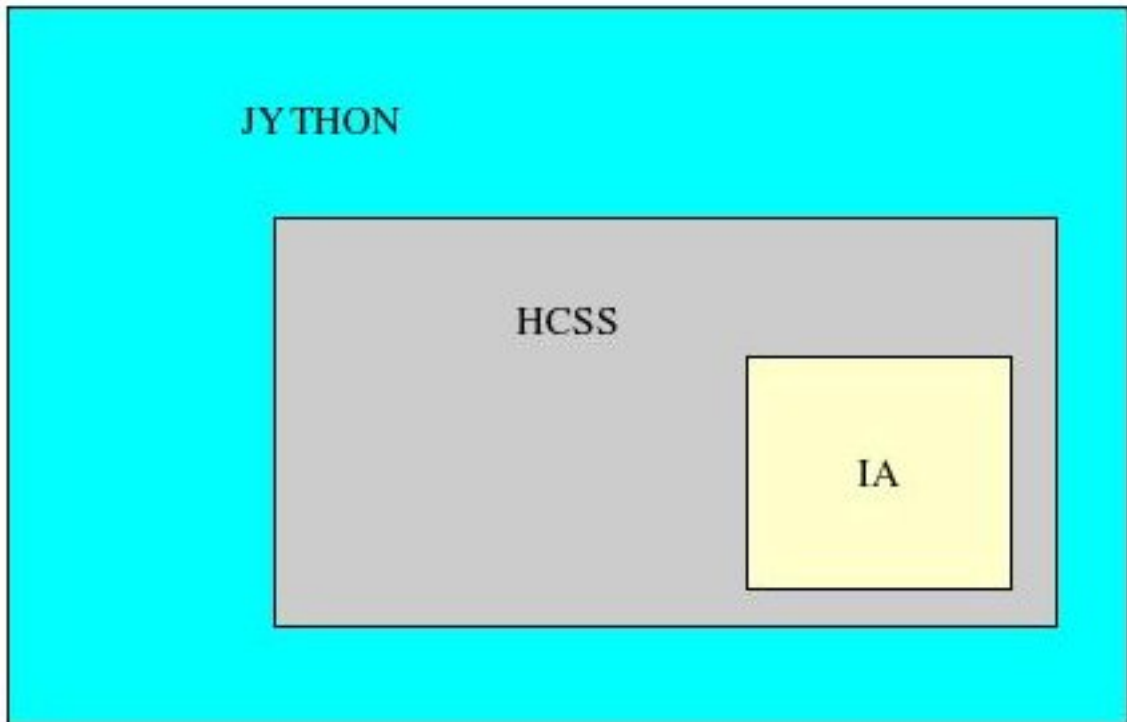


Figure 2.11. The overall library structure for a DP session

Library usage for a DP session is illustrated in Figure 2.11. Errors, as thrown by Jython and/or JAVA classes, have the same means by which they follow the error back down the program layers to find the root of the error -- "traceback mechanism" (although they differ in the way they present error messages to the user, as shown in the next section).

Interpretation of these error messages allows the user to identify the place where the exception/error originated from.

2.13.2. The Error Traceback Mechanism

In this section we describe the differences in the way Jython and JAVA libraries present error messages.

2.13.2.1. The way Jython presents error messages

Errors in the use of Jython are typically returned directly to the user after their attempted implementation. An example of how Jython presents error messages is given in the following short code example:

```
array = [1,2,3,4,5]
print array[5]
# IndexError: index out of range: 5
```

Another typical Jython error form is a syntax error. Consider the following lines of code

```
x = 2
y = 3
a = x + 2y
```

An error message using this piece of code has the form

```
# Traceback (innermost last):
# (no code object) at line 0
```

```
# SyntaxError: ('invalid syntax', ('<string>', 1, 10, 'a = x + 2y'))
```

which indicates the fault happening in line 1 of the block of code (we only have one line in this case) at the position of character number 10. Note that this information appears in the bottom right panel, by double clicking on the red line corresponding to the error and selecting the Traceback entry.

2.13.2.2. The way JAVA presents error messages

Most DP packages use JAVA classes. If JAVA classes are run within a DP session and an error occurs, an exception is thrown which is propagated upwards to the DP level. An example:

```
dbl = Double("wrong arg")
# java.lang.NumberFormatException: For input string: "wrong arg"
```

In the history window the command line will be indicated by a red cross, showing that there is an error for this command. Information on the command can be obtained by clicking on the indicator to the left of the red cross. This provides access to the error message and traceback of the error (again, via a mouse click on the indicator).

A Log window can be obtained by using a right-click of the mouse on the history line, in JIDE ONLY, and using the pull-down menu. This provides a separate window with all the information on the problem command.

```
INFO:
<COMMAND>
  <STATEMENT>
    dbl = Double("wrong arg")
  </STATEMENT>

  <EXCEPTION>
    <MESSAGE>
      java.lang.NumberFormatException: For input string: "wrong arg"
    </MESSAGE>
    <STACK_TRACE>
      Traceback (innermost last):
        File "<string>", line 1, in ?
          java.lang.NumberFormatException: For input string: "wrong arg"
          java.lang.NumberFormatException: For input string: "wrong arg"
            at java.lang.NumberFormatException.forInputString\
(NumberFormatException.java:48)
            at java.lang.FloatingDecimal.readJavaFormatString\
(FloatingDecimal.java:1207)
            at java.lang.Double.valueOf(Double.java:202)
            at java.lang.Double.<init>(Double.java:277)
            at sun.reflect.NativeConstructorAccessorImpl.newInstance0\
(Native Method)
            at sun.reflect.NativeConstructorAccessorImpl.newInstance\
(NativeConstructorAccessorImpl.java:39)
            at sun.reflect.DelegatingConstructorAccessorImpl.newInstance\
(DelegatingConstructorAccessorImpl.java:27)\
            at java.lang.reflect.Constructor.newInstance\
(Constructor.java:274)\
            at org.python.core.PyReflectedConstructor.__call__\
(PyReflectedConstructor.java)
            at org.python.core.PyJavaInstance.__init__(PyJavaInstance.java)
            at org.python.core.PyJavaClass.__call__(PyJavaClass.java)
            at org.python.core.PyObject.__call__(PyObject.java)
            at org.python.pycode._pyx113.f$0(<string>:1)
            at org.python.pycode._pyx113.call_function(<string>)
            at org.python.core.PyTableCode.call(PyTableCode.java)
            at org.python.core.PyCode.call(PyCode.java)
            at org.python.core.Py.runCode(Py.java:1136)
            at org.python.core.Py.exec(Py.java:1158)
            at org.python.util.PythonInterpreter.exec(PythonInterpreter.java)
    </STACK_TRACE>
  </EXCEPTION>
</COMMAND>
```


The places in JAVA classes where the code breaks down are indicated. Typically, the traceback starts with the line number of the original program where the problem occurs and follows this with the line numbers in the classes accessed where the problem propagates from. In the above example we have simply tried to attach a string, "wrong arg", to a numeric double. So it is of the wrong format -- as indicated in the first line of the traceback. On other occasions, a more fundamental JAVA error may be occurring deeper in the system. The traceback allows the user to find where this may be happening.

2.13.3. The HCSS exception and logging mechanism

Next to the standard JAVA exception handling mechanism the HCSS is using, it also has a logging mechanism which forwards information, error and warning messages to the user.

2.13.3.1. Exceptions Thrown From HCSS Classes

In case an error occurs inside the HCSS, for example due to a missing or incorrectly defined configuration variable, the information as part of the exception thrown should explain to the user the cause of the exception. In this way the user should be capable to adjust his/her input arguments and/or property settings. Property settings can be set using the Property Generation tool ("propgen") -- see Chapter 1. For example:

Let us assume the user has set the configuration variable "var.database.devel" to a database name that does not exist:

```
var.database.devel = "idonotexist@iccdb.sron.rug.nl"
```

when trying to access this database in a DP-session by:

```
from herschel.access import *
tm = PacketAccess(1030)
packets = HcssConnection.get(tm)
```

Here, a query is done on the database as set by the above property and the exception, appearing in the command line window, reads:


```
# herschel.access.LocationException: Exception in constructor of
  herschel.access.db.LocalConnection:
# herschel.access.LocationException: Failed to get store
# herschel.store.api.StoreException: Failed to create store for
  idonotexist@iccdb.sron.rug.nl:
# herschel.store.api.StoreException: Failed to create
  ObjectStore "idonotexist@iccdb.sron.rug.nl
# Cannot open database: idonotexist@iccdb.sron.rug.nl
# Error while accessing database: idonotexist@iccdb.sron.rug.nl
# { VException(7001:UT_DB_NOT_FOUND: DB directory not found) }
```

In cases where the information as passed by the `Exception` thrown is not sufficient (for example a `NullPointerException` without any textual explanation), then there is a problem with the current system and the user is encouraged to provide feedback to the HSC regarding the lack of exception handling information (currently, this is best achieved through the SPR/SCR system).

In the above example the "access" package might improve its exception notification by adding information to the `LocationException`, including a hint for the user that the database is not existing and that the user should check whether `var.database.devel` is properly defined.

2.13.3.2. The HCSS logging mechanism

The logging mechanism allows (HCSS) classes to pass errors, warnings and/or info to the end-user. To

view the error logging mechanism, go to the Help menu or click on the  icon (see also Section 2.2.5).

For the HCSS end-user this mechanism, rather than the analysis of exception handling, is likely to be used more often, especially when HCSS software is fully matured. The difference between the two is that exception handling is more often used by the developer for debugging purposes, whereas the logging mechanism is intended to be used by the end-user to get insight into the behaviour of an (HCSS) application or class. The logging mechanism enables the developer to include messages when an exception is thrown on how the class internally handles possibly thrown exceptions.

To give an example why, next to the exception mechanism, the logging mechanism was introduced: suppose we have a layered HCSS component (i.e. within an instance of a class there are calls to instances of other classes and these will call others on their turn), deep within this component an exception occurs and at a higher level this exception is caught again. In such a scenario the end-user of the component will not be aware of the fact that this exception occurred. However, by use of the logging mechanism the developer of the component can pass a message (an error, warning or info; depending on how severe this exception was) next to the exception thrown, as well as being able to pass relevant information to the user when the exception is caught.

More detailed information on the logging mechanism and how it may be used with user-developed scripts is discussed in [herchel.share.log.api.Log](#) (which is a link to HCSS javadoc)

Chapter 3. Some DP Basics & Beginning Jython

3.1. Basics

The Herschel DP is a development system based on programs written in Java or Jython. Jython is a Java implementation of the Python language. The syntax is therefore well defined and there is plenty of documentation freely available.

Remember however that, while the C implementation of Python (what we usually refer to as just "Python") is already at version 2.4, the last stable version of Jython is still 2.1. This means that not all available Python documentation will be applicable to Jython.



Warning

Standard Jython libraries are *not* automatically imported into JIDE. If you want to try Python/Jython examples from external sources such as books and tutorials, you will have to import them manually.

3.2. Comments

Comments on a line can be added after a hash (#) mark.

3.3. Variables

Variables do not have to be declared (`integer x, xmax` etc. is not required). They appear when you assign to them and disappear when you do not use them anymore. Assignment is done by the = operator and equality testing is via the == operator. You can also assign several variables at once.

```
x, y, z = 1, 2, 3
a = b = 123
```

If you need to clear some or all variables then the command `clear` can be used as in the example:

```
IA>> clear("x,y,z")
# to clear all variables, but not the loaded classes and methods
IA>> clear(all=True)
```

3.4. Numbers and basic arithmetic

The interpreter acts as a simple calculator. Expression syntax is similar to other languages, e.g. the operators +, -, * and /, and parentheses can be used for grouping. For example, we can type the following into the command line window of JIDE at the `IA>>`:

```
IA>> print 2+2
4
IA>> # This is a comment
IA>> print 2+2
4
IA>> print 2+2 # and a comment on the same line as code
4
IA>> print (50-5*6)/4
5
IA>> print 7/3 # Integer division returns the floor
2
IA>> print 7/-3
-3
```

A list of Jython operators is provided in Appendix C.

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point.

```
print 3 * 3.75 / 1.5
# 7.5
print 7.0 / 2
# 3.5
```

Complex numbers are also supported; imaginary numbers are written with a suffix of "j" or "J". Complex numbers with a nonzero real component are written as "(real + imag j)", or can be created with the "complex(real, imag)" function.

```
print 1j * 1J
# (-1+0j)
print 1j * complex(0,1)
# (-1+0j)
print 3+1j*3
# (3+3j)
print (3+1j)*3
# (9+3j)
print (1+2j)/(1+1j)
# (1.5+0.5j)
```

To extract these parts from a complex number z, use z.real and z.imag.

```
a=1.5+0.5j
print a.real
# 1.5
print a.imag
# 0.5
```

3.5. Boolean values

Boolean values are available in the Jython environment.

```
val = Boolean(0)
print val
# false
```

3.6. Strings

Jython can also manipulate strings. These can be in either single or double quotes.

```
print 'spam eggs'
# spam eggs
print "doesn't"
# doesn't
```

String literals can span multiple lines in several ways. Continuation lines can be used, with a backslash as the last character on the line indicating that the next line is a logical continuation of the line:

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is \
significant."

print hello
```

Note that newlines still need to be embedded in the string using `\n`; the newline following the trailing backslash is discarded. This example would print the following:

```
This is a rather long string containing
```

```
several lines of text just as you would do in C.  
    Note that whitespace at the beginning of the line is significant.
```

We can access individual characters using

```
print hello[2]  
# i  
print hello[10:16]  
# rather
```

Note that numbering of the characters starts at 0.

Our variable `hello` essentially contains an array of characters (including blank spaces). We can find the length of such an array using the `len()` function.

```
print len(hello)  
# 157
```

NOTE: This also illustrates the means by which functions are applied in Jython.

3.7. Type conversions

Conversion functions exist to change numbers into floating point and integer (`float()`, `int()` and `long()` arrays).

```
a = 1  
print float(a)  
# 1.0
```

These conversions DO NOT work with complex numbers.

There are also a number of methods to convert string representation of a number to a number. Here are a couple of examples using `java.lang` methods:

```
from java.lang import *  
s = "01234.56"  
print Double.valueOf(s)  
# 1234.56  
print s + 2.22  
# TypeError: __add__ nor __radd__ defined for these operands  
print Double.valueOf(s) + 2.22  
# 1236.78
```

Note that with this method when you try to convert a string representation of a floating point to integer you will get an error:

```
s = "01234.56"  
print Integer.valueOf(s)  
# java.lang.NumberFormatException: For input string: "01234.44"
```

3.8. Lists and Dictionaries

Lists and dictionaries are important data structures available in Jython.

Lists are simple arrays written in a specific order.

Dictionaries are like lists that can be accessed via a key (or label). To access an element you use a key or "name". This is what is used to look up the value of an element.

3.8.1. Setting up and Accessing Lists

Lists are formulated within square brackets, which can be nested. E.g.,

```
name = ["Rolf", "Harris"]
```

(note - strings of characters need to be placed inside quotation marks)

```
y = z = 5
x = [[1,2,3],[y,z],[1,[2,[3,4]]]]
print x
print x[0]
print x[2]
print x[2][1]
print x[2][1][1]
```

In the first line we have set both the variables `y` and `z` to the value 5. In the second line, the variable `x` is associated with a Jython array which itself contains three arrays, the third of which contains further nested arrays. The print commands that follow show how the nested arrays can be accessed (counting of array elements starts from 0). The last line therefore indicates we take the third element of `x`, take the second element of that and then the second element of the array we are left with (i.e., `[3, 4]`).

You can access lists by individual names or groups

```
print name[0], name[1] # prints "Rolf Harris"
print name[0:2] # gives list in brackets ['Rolf', 'Harris']
print name[:2] # ditto
```

In the first instance the parts of the name list are picked up individually, in the second part a range of list components is picked out (0 to 2) and in the last case all components up to `name[2]` are picked out. Notice how in the last two cases the command is interpreted as going up to but not including the number range being given. We can try the same with the list `'x'`.

```
print x[0] # gives the first element in the list "[1,2,3]"
```

Try printing the other elements of the list (`x[1]` and `x[2]`) to see if you get what you expect!

3.8.2. Slicing Lists

The last two examples using the list name (above) are also examples of slicing. Slicing of this type can also be performed with numerical and string arrays. For instance,

```
y = ["The", "quick", "brown", "fox", "jumped", "over", "the", "lazy", "dog"]
print y[1:4] # prints the list ['quick', 'brown', 'fox']
```

Again - the end integer value given for the slice is not included, so the above example only gives the values for `y[1]`, `y[2]` and `y[3]`.

- Choosing `y[:4]` means "take every element from the beginning of the list up to element 4, *not including element 4*."
- We can also have `y[4:]` which means "take every element from number 4 up to the end" - note that this will *include* element number 4.
- Lastly, negative numbers mean count from the end of the list `y[-3]` means take the third element from the end of the list.

3.8.3. Setting Up and Using Dictionaries

A dictionary has a set of {key: value} pairs. E.g.,

```
person = {"Alice": 111, "Boris": 112, "Clare": 113, "Doris": 114}
print person.get("Alice")
# 111
print person["Alice"]
# 111
```

We "get" the associated value for "Alice" within the dictionary "person". Alternatively, the key can be given between square brackets as with the array notation. To see all the "keys" and "values" separately use the `keys()` and `values()` methods of the dictionary "person".

```
print person.keys()
# ['Clare', 'Alice', 'Boris', 'Doris']
print person.values()
# [113, 111, 112, 114]
```

The use of the empty brackets at the end indicate that we are not passing a parameter on to "keys" or "values" in order to get a printout of their current settings. In fact, no parameters are allowed for these commands, but we still need the brackets.

Also note how the commands `keys()` and `values()` are applied/work on the dictionary "person". We will see this frequently when running DP code in the future.

If we want to change the dictionary then we need to write something like

```
person['Alice'] = 222
```

Here, the value associated with Alice in the dictionary called person has been changed to the number 222.

3.8.4. Nested Dictionaries

Dictionaries can hold other dictionaries too. So advanced data structures can be made.

Let us set up a dictionary called abc

```
abc = {"John": 12345, "Jerry" : 23456, "Joe" : 34567}
```

We will now put this inside another dictionary called dict

```
dict = {"Alice" : 111, "Boris" : abc, "Charlie" : "angel"}
```

Note here that we have NOT got inverted commas around the value abc since we want it to point to our dictionary abc and not be a string.

So now we can look at the value of "Boris"

```
print dict.get("Boris")
```

Which should simply give us the dictionary abc printed on our screen. Whereas,

```
print dict.get("Charlie")
```

Simply prints the string we gave as the value (we know it is a string since it has inverted commas around it).

If we now want to get the value of "John" we would need to do

```
print dict.get("Boris").get("John")
```

First we get the dictionary abc which is pointed to by the key "Boris", then we look for the key "John" inside. This returns the value 12345.

3.9. Augmenting Values and Lists

Jython allows a full range of augmentation assignment operators (including `+=`, `-=`, `*=`, and `/=`). These all behave in a similar fashion.

```
a = 5
a += 2 # Adds 2 to the value of a
a *= 3 # Multiplies a by 3
```

We can add to lists too.

```
b = [1]
b += [2] # Now b = [1, 2]. Note that the result is NOT b = [3]!
```

Note that here we have appended an element to the end of the list. This we could also do with the `append()` method.

```
b.append(3) # Now b = [1, 2, 3]
```

3.10. Lists and Jython Tuples

A possibly confusing aspect of Jython is the use of brackets in producing what appear to be identical lists. True Jython **lists are mutable** - they can be changed/sorted (represented by square brackets, "`[]`"). Whereas **tuples are immutable** and represented by curved brackets, "`()`" and are therefore unalterable, including ordering. So while we can append new elements to a list, we can not do so to a tuple.

```
a = [1,2,3,4]
c = ["x","y","z"]
a.append(c)
print a
# [1, 2, 3, 4, ['x', 'y', 'z']]
```

The list `["x", "y", "z"]` has been added as a single fifth element of the list `a`. Whereas...

```
a = (1,2,3,4)
c = ("x","y","z")
a.append(c)
```

...gives an error:

```
# AttributeError: 'tuple' object has no attribute 'append'
```

"Adding" lists or tuples can be done to form a resultant third list or tuple. For example

```
a = (1,2,3,4)
c = ("x","y","z")
b = a + c
print b
(1, 2, 3, 4, 'x', 'y', 'z')
```

If we wish to do arithmetic with one or more arrays of numbers, rather than individual list or tuple elements, then we need to deal with **numeric arrays**. These have been developed for use in DP and are discussed in Chapter 4.

3.11. Basic programming statements

The basic programming statements are the conditional statement *if/elif/else*, the loop statements *for* and *while* and the loop control statements *break* and *continue*. The conditional and loop statements serve to execute blocks of commands depending on a given condition. Blocks are indicated by indentations and only through indentations (and can be handled within JIDE - see Chapter 2). No begin/end braces are required.

3.11.1. if/elif/else

The *if/elif/else* statement executes blocks of commands depending on given conditions. The syntax is:


```
if condition1:
    block1
elif condition2:
    block2
else:
    block3
```

A few examples to illustrate

```
x = 13

if x < 5 or (x > 10 and x < 20):
    print "The value is OK"

if x < 5 or 10<x<20:
    print "This value is OK"

if 0<= x <= 10:
    print "The value is in the range [0,10]"
elif 10<x<20:
    print "The value is in the range [10,20]"
else:
    print "The value is not in the range [0,20]"
```

The first two examples are identical.

3.11.2. for

The for loop was briefly discussed in Section 2.8, where its use within the JIDE environment was illustrated. The syntax of the for loop is the following:

```
for variable in list:
    block
```

where list can be an array of values, sequence of dictionary keywords, tuples, strings.

Some examples:

```
for i in [1,2,3]:
    ...print i
```

The above for loop goes through values in an array indicated in the square brackets. A simpler way - particularly for large numbers of iterations - is to use the inbuilt range function to create an array.

The following example prints the values from 0 to 99 using the range function -- it actually creates a list of rising integer values that can then be looped through.

```
for value in range(100):
    ... print value
```

Note how values start from 0 and end one below the value assigned to the range function. Currently, the print output is going to the command line window of JIDE.

A combined example of using for loop and if/elif/else is given below. Note the indentation of the different blocks.

```
person = {"Alice" : 111, "Boris": 112, "Clare": 113, "Doris": 114}
# first we get the list of people's names
list = person.keys()
# for each name in the list we get the associated value -- this
# could be a test score, for example.
for i in list:
    pval=person.get(i)
    # we check if the person is on the cutoff, and print the name
    if pval == 112:
        print i, "is at the cutoff"
```

```
# below the cutoff
elif pval < 112:
    print i, "is below the cutoff"
# or else, above the cutoff
else:
    print i, "is above the cutoff"
```

3.11.3. while

The `while` loop executes a block of commands, *while* a given condition is true. The syntax is:

```
while condition:
    block
```

The condition can be any expression which results to a value: the numeric zero is `False`, as well as empty string, tuple, list, otherwise the condition is `True`.

Some examples:

```
x = 0
while x <= Math.PI:
    ...y = SIN(x)
    ...x += 0.1
```

3.11.4. Loop control: break and continue

The command `break` can be used to immediately exit from a loop and `continue` is used to jump to the next iteration of the loop without executing the rest of the block.

An example for their usage is given below.

```
x = 0
while 1:
    y = TAN(x)
    if y < 0:
        break
    print x,y
    x += 0.1
```

The above example shows an infinite while loop (the condition is always true) and inside the loop block we check for a given condition and jump out of the loop once it is true, so at the first negative tangent we exit the loop.

```
for i in range(100):
    if i % 2: continue
    print i
```

The above example shows how we can skip the printing of the odd numbers (`i % 2` is `i` modulus 2 and it is zero for all even numbers).

3.12. Printing to the screen and files

We have already seen how a `print` command can produce a result

```
print 1, 2, 1+2
# 1 2 3
print a
# (1, 2, 3, 4)
```

(... following on from the above augmentation example).

The printout can be formatted in the same way as with the C `sprintf` format codes. Some examples:

```
print "When %s is %i years old then PI will be %8.10f" %("John",23,Math.PI)
```

```
# When John is 23 years old then PI will be 3.1415926536
print "When %8s is %04i years old then PI will be %016.12f" %("John",23,Math.PI)
# When      John is 0023 years old then PI will be 003.141592653590
```

To print lists or arrays it is necessary to make a loop:

```
a = [1,1,2,3,5,8,13,21,34]
for i in range(len(a)):
    print "Line: %3i" %a[i]
```

Another useful usage of formatted printout is with dictionaries as shown in the following example:

```
record = {"name": "John", "Room": 112, "class": "manager", "age": 27}
print "Extracted record\n Name: %(name)10s Room: %(Room)4i" % record
# Extracted record
# Name:      John Room:  112
```

We can also print to a file.

```
file = open("output.txt", 'w') # 'w' allows write access overwriting
                                # previous contents.
                                # 'a' would append at the end of the file.
print >> file, 2 # Puts the number 2 into output.txt
```

Or

```
print >> file, a # Puts the array "a" into output.txt
```

For printing an array/list to a file.

Note that it is not necessary to close access to a file within your DP session. To overwrite the original text file, reopen the file. Reopening the file will remove the contents.

3.13. Defining and Using Functions

Here we name a piece of code, call it with some parameters and have it return a result. Functions are set up with the keyword `def`. e.g.,

```
def square (x):
    ... return x*x
    ...
print square(2)
# 4
```

The arguments of the functions are passed by value, i.e. the input argument is not changed outside the function:

```
def myfunc(a):
    a = a + 1
    return a
#
x = 4.0
print myfunc(x)
# 5.0
print x
# 4.0
```

Note that variables from the main JIDE session have global scope, i.e. they are accessible inside functions but *cannot* be changed. The example below will produce an error:

```
def myfunc(a):
    a = a + 1
    x = x + 5
    return a
#
```

```
x = 4.0
print myfunc(x)
# UnboundLocalError: local: 'x'
```

However, the following example shows a dangerous effect:

```
def myfunc(a):
    b = a*z + 1
    return b
#
x = 4.0
z = 10.0
print myfunc(x) # this one works as z is global and accessible inside the function
# 41.0
```

This may have side effects especially when one has plenty of variables in the JIDE session and seemingly the defined user functions work. There is no guarantee though that next time the same global variables will be available or they may have different values, in which cause the functions will throw errors or worse give wrong results. That is why our advice is when it is necessary to use global variables inside user functions to pass them as arguments.

Some arguments of the functions may have default values. This is illustrated by the following example:

```
def myfunc(x,y=1.0,verbose=True):
    z = x*x + y
    if (verbose):
        print "The input is %f %f and the output is %f" %(x,y,z)
    return z
#
myfunc(5.0) # using default values for y and verbose
# The input is 5.000000 1.000000 and the output is 26.000000
print myfunc(5.0,y=5.0,verbose=False)
# 30.0
print myfunc(5.0,5.0,False) # the same as the previous
# 30.0.
print myfunc(5.0,5.0)
# The input is 5.000000 5.000000 and the output is 30.000000
# 30.0
```

The arguments of a function can be functions themselves, like in the following example:

```
def func1(x):
    return x*x
def func2(x):
    return x/2.0
def myfunc(f1,f2,x):
    return f1(x) + f2(x)
#
x = 3.0
print myfunc(func1,func2,x)
# 10.5
# Even the user can input any available function of one argument
print myfunc(SIN,func1,x)
# 1.6411200080598671
```

In actual fact, DP has a sophisticated numeric functions package that can allow squaring of values and numeric arrays of various types (double, integer etc.). Numeric functions available in DP are discussed in Chapter 5.

If you want to call a function without arguments then the () brackets are required.

A useful thing to know is that functions are values in Jython. So taking an example from the previous section

```
print person.values()
```

Could be changed to

```
pvalue = person.values
print pvalue
# which indicates "pvalue" is a Jython values type
print pvalue()
# which actually prints out the values
```

3.14. Object Oriented Programming

JIDE is based on Jython and Java. Java is an object oriented language, and Jython *can* be used as an object oriented language, although it is mostly used in its procedural form. Object-oriented programming, or OOP for short, has been (and still is) the subject of much hype, several misconceptions and a few urban legends. It is not the remedy to all evils, but in many cases it can help to write cleaner, more reusable and more maintainable code. Although you will not have to write a single line of OOP code to use JIDE, being familiar with some of its concepts may help to gain a better understanding of the DP system. We will now briefly explain the basic words of the trade and describe the advantages of the OOP approach.

3.14.1. Classes and Objects

The traditional, or procedural, way of programming is relatively straightforward. We take program inputs and store them in variables, which can be of many types (integer, string, float etc.). We process this input using the set of commands provided by the language we are using. Other variables are employed to store the outputs and any intermediate values we might need. Finally, the outputs are given back to the user in some way and the program terminates.

To tidy up our code, we might want to group sets of commands that perform particular tasks into blocks called *functions* or *subroutines*. Such blocks can be called multiple times using loops, thus avoiding the need to duplicate code. At any point our program can decide to execute one function instead of another, based on whatever criteria we set: this would be achieved via a *control flow statement* such as an `if...then` block. By organising code into functions/subroutines we just made the leap from *unstructured* to proper *procedural* programming.

Object oriented programming takes it one step further. The old ingredients are still there: variables, functions (here called *methods*) and a set of commands such as control flow statements. So, where is the big difference?

The difference lies in the way all these tools are organised. An *object* is a bundle of related variables and methods (functions) acting on these variables. A *class*, on the other hand, is like a mould from which objects are created.

The best way to grasp these concepts is to think of a concrete example. Imagine that, for some reason, we have to code a model of an airplane. We all have a general idea of what an airplane *is* (it has a fuselage, wings, one or more engines, landing gears...) and of what it *does* (it can take off, land, climb and descend...). Also, we are probably not thinking of a particular aircraft, but of our *idea* of a plane. This idea is what in OOP terms is called a *class*. A class is a general description of an object, of what it *is* and what it *does*. What our `Airplane` object is, or its *status*, is described by *instance variables* (just so you know, there is a distinction between *instance* and *class* or *static* variables; more on this later). An instance or class variable could be of a primitive type (e.g. a float called `wingspan`) or a full-fledged object (we could think of creating an `Engine` object). What an object does is described by functions called *methods*.

As we said, a class is not the real thing, it is just a mould. When we create an object from a class it is said that we *instantiate*, or create an *instance* of the class. In other words, besides the `Airplane` class, which represents no specific plane, we now have the `myAirplane` object, which is a real plane we can climb on and fly.

Finally, there can be properties that are specific of each instance of a class, i.e. of each particular object; these are aptly called *instance variables*, as we already know. But there could be variables having the

same value for all the objects of a given class, which would then be better defined inside the class itself and then shared by all its instances. These are called *class* or *static* variables. The same distinction also applies to methods, but let us stop here for now. What we say below referring to instance variables can also be applied to static ones, unless stated otherwise.

3.14.1.1. A Note about Terminology

You might be confused about the exact meaning of the words *method*, *function* and *subroutine*. All the three words denote a *subprogram*, i.e. a separate block of code that may be invoked from elsewhere in the program. This block of code may take input values and return an output. The term *method* is typically used in OOP to indicate a subprogram inside a class (or an object, which is an instance of a class), while *function* or (less frequently) *subroutine* denote a subprogram in procedural code. Thus we will usually speak of a method in a Java class, but a function in a Jython script.

Just when you think you got it, you may encounter the notion of *function object*. Why would a *function* be mentioned in connection with an *object*? According to what we just said, we should call it a method, right?

Not really. Function objects, also known as *functors* or *functionoids*, are objects that can be invoked or called as if they were functions. For example, if you write `y = SORT(x)` in JIDE to sort a vector, you are using an object, namely an instance of the `herschel.ia.numeric.toolbox.basic.Sort` class. If you do not believe what you are reading, try issuing this command in JIDE:

```
print SORT
```

You will get something like

```
herschel.ia.numeric.toolbox.basic.Sort@b65e0
```

The hex number after the '@' will likely be different. What you got is the output of the `toString` method, whose aim is to give a meaningful string representation of an object. Whether the above output is meaningful can be a matter of debate, but the point is that by default it contains the class name of the object.

3.14.2. Interface, Implementation and Encapsulation

You already know that actions performed by objects are coded in functions called *methods*. Our `Airplane` class will have methods like `takeOff`, `land` and so on. Some or all of these methods will be *public*, i.e. visible (and callable) from other pieces of code. This is what is called the *interface* of a class: a set of methods to operate on the object, make it do stuff and enquire about its internal state.

Going on with our airplane example, the interface is made of all the dials, displays, buttons and levers in the cockpit. We can operate the plane and read the value of all the relevant variables (speed, fuel, altitude...). The nice thing is that we do not have to know in detail how the controls work in order to use them. It may be the latest fly-by-wire technology, or the old mechanical one, but in both cases we know that pulling on the yoke the plane will climb. In OOP terms, the user just needs to know the *interface* of an object, not its *implementation*, i.e. the gears and cogwheels behind its shiny surface. The implementation is said to be *hidden*, with the advantage that it can be modified, tweaked and patched as much as the developer wishes. As long as the interface remains the same, the user will not notice anything.

It is good practice to prevent users from directly accessing instance variables. These are part of the implementation, and could have to be changed (e.g. from `int` to `float`) possibly breaking external code accessing our object. A much better way is to provide methods to get and set the value of a variable (these methods are usually known as *getters* and *setters*). It may seem overkill, but it helps keeping the code more maintainable. It is said that our instance variables are neatly *encapsulated* inside our class. To say it with a metaphor, we want the pilot of our plane to read the fuel level from a dial

(the `getFuelLevel` method) rather than tampering with the fuel tank to get a look inside (trying to directly access the `fuelLevel` instance variable).

3.14.2.1. Interfaces, the Java Way

Interface is a generic programming concept, but it is also a specific Java construct. Without getting into too much detail, a Java interface is a collection of methods and constants. If a class *implements* an interface, you can be sure that all the methods and constants listed by the interface are right there in the class and in all of its instances, ready to be used.

3.14.3. Inheritance

This is a slightly more advanced concept, which can be safely skipped without trouble. However it is not very complicated. When you think of all the different kind of airplanes existing today, from tiny ultralights to huge jets, you may wonder how a single `Airplane` class could represent them all. Actually, it cannot: that is why we can define *subclasses* of `Airplane`. These subclasses receive, or *inherit*, the variables and methods of their parent class, and we can override them, or add new ones, to suit our needs. We can create the `Boeing787` and `Airbus380` subclasses of `Airplane`, with specialised methods and different values of instance variables (like `numberOfEngines`). Note that there are ways to *prevent* subclasses from inheriting certain variables or methods, but this goes beyond the scope of this manual.

One more example: suppose we have a class `Seat` to describe airplane seats. We can subclass it into `FirstClassSeat` and `EconomySeat`. Each of them will have (very) different values of the `seatPitch` instance variable. Also, we could add a `turnIntoBed` method to `FirstClassSeat`, which will definitely be absent from `EconomySeat`.

By creating such hierarchy of classes we can reuse general pieces of code many times, to tackle several specialised tasks.

3.14.4. Packages and Namespaces

Common problems in programming are name clashes and, as a consequence, running out of meaningful (or suitably short) names for variables, methods and the like. This is even more serious when we use several different pieces of code, each developed by several people. Think about the DP system, for instance: we are putting together Java, Jython and a lot of Herschel-specific code. How can we be sure that nobody thought of the same name for completely unrelated entities? How can we avoid such confusion?

To answer this question, let us take a look at HCSS Javadoc here:

<ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/releases/doc/api/index.html>

Look at the upper left corner of the page. There is a list of names such as `herschel.access`, `herschel.access.db` and so on. Click on any of these item. The box below will change to show a list of the classes and interfaces contained in that *package*. Now go back to the list of packages and scroll it from top to bottom. As you can notice, everything starts with "herschel". Then there are subpackages such as `herschel.ia` and `herschel.ccm`, and finer subdivisions like `herschel.ia.dataset` and `herschel.ia.help`. You get the picture: packages are used to organise classes, interfaces and other programming constructs into a meaningful hierarchical structure. To use the functionality of a package in a Jython script, we can *import* it with a command such as `import herchel.ia.numeric`.

That makes a lot of sense, but how can it prevent name clashes? In a way, it does not: it just makes them harmless. The point is that every package is a separate *namespace*, i.e. a separate domain where we can choose names as we please (well, almost), without worrying about names in other packages. And what happens if we import two packages containing a class with the same name? For example, `herschel.ia.numeric.toolbox.basic` and `herschel.ia.dataset` both have classes named `Product` (doing completely different things). In that case we can use the *fully qualified* class name, i.e. write `herschel.ia.dataset.Product` instead of just `Product` to get rid of any ambiguity.

3.14.5. Advantages of OOP

The most commonly cited advantages of OOP can be summarised as follows:

- *Modularity.* Organising code into a hierarchy of classes is a natural invitation to build modular programs. Natural, but not automatic: nobody prevents you from designing few enormous classes doing several unrelated tasks at once. To reap the most benefits from modularity, classes should have one well-defined purpose (in object oriented jargon they are said to have high *cohesion*) and interact with other classes only through their interfaces, without having to know about their internal state (low, or loose, *coupling*). To get a picture of the concept, think of a plumber working with several specialised tools rather than fumbling with a Swiss Army knife.
- *Reuse of previous work.* This is probably the most cited benefit. A set of modular classes, following the guidelines mentioned above, are relatively easy to plug into one another, which allows creation of new programs. As before, benefits are the result of good planning and design.
- *Increased quality.* We do not mean here that programmers developing object oriented code are intrinsically better than their procedural colleagues. Increased quality is largely a result of the previous point, code reuse. The more existing, tested code can be employed to develop a new application, the less will have to be built and debugged from scratch.
- *Faster development.* Again, this is not because of some mysterious power of OOP that leads developers to type much faster. Like the previous point, it is mainly an advantage of code reuse: if a large part of a new application consists of existing code, this will automatically translate into faster development.
- *Better mapping to the problem domain.* What we mean by this statement is that with OOP it is easier to model the software on the real-world problem that has to be solved, rather than bending the problem to the constraints of the programming language. New objects can be created representing all sorts of things, like customers, machinery, banks or, well, airplanes. When dealing with the `Task` framework in Chapter 8 we will discover that OOP works well even for representing more abstract concepts, like the different stages of a data reduction pipeline.

3.14.6. Concluding Remarks

For people with a long tradition of writing procedural code, switching to the object oriented paradigm can be painful at first, leading to decreased productivity and a strong desire to give up and keep writing code the old way. A little perseverance will pay in the end, keeping in mind that the time lost at first will be more than regained at the end.

As we said at the beginning, it is also important to remember that OOP, despite its advantages, is not the solution to all problems. It is indeed possible to write excellent and easily maintainable procedural code and absolutely messy object-oriented code. No coding approach, however ingenious, will avoid ill-designed algorithms, cryptic variable names and inextricable spaghetti-like loops. Most important of all, no piece of code, whether object-oriented or not, will spontaneously document itself at night.

Now it is time to put theory into practice. The following section deals with the `Basket` class, an example class written in Jython.

3.15. Defining a Class in DP

The following is an example that can be placed in the top pane of JIDE. Remember to keep proper/accurate indentation. **Note that program command lines can be extended to the following line by the use of a backslash, "\", at the end of a line. Although not needed for the example class given here it appears in several example scripts later on this manual**

```
class Basket:
    # always remember the self argument
    def __init__(self, contents=None):
        self.contents = contents or [] # ❶
```



```
def add(self, element):
    self.contents.append(element) # ❷
def print_me(self):
    result = ""
    for element in self.contents:
        result = result + " " + `element` # ❸
    print "Basket contains: "+result
```

- ❶ this bit does a logical or - if a parameter is passed to it, it becomes the contents, otherwise we get an empty basket!
- ❷ this adds the element to the contents (`self.contents`)
- ❸ this prints the contents of the Basket. Note the use of upper left keyboard single inverted commas around element.

We have created a class called `Basket` and it has two associated methods `add()` and `print_me()` (following `def` in the above example).

Try placing the above within the top pane of JIDE. Here we create an object to work on, called `self` - which is customary. This is initiated by the `def __init__` command (by the way, that is two underscores on either side of `init`).

Leave a blank line at the end of the script when placing it within the edit pane of JIDE. Now hit the double arrow icon to load this into your DP session.

Once created, we can run the class by typing `Basket()` in JIDE via the command window (bottom left).

Now try the following in the command line window.

```
a = Basket() # ❶
a.add("saw") # ❷
a.add("hammer") # ❸
a.print_me() # ❹
```

- ❶ this line sets up an empty basket which we have called `a`
- ❷ this line adds the item `saw` to the basket. It runs the `add()` method on the object `a`.
- ❸ this line adds the item `hammer` to the basket.
- ❹ this line prints the contents of the basket we called `a`, which should be 'saw' and 'hammer'. This runs the `print_me()` method on the object `a`.

We could equally have started our basket with one item

```
a = Basket(["saw"])
```



Note

If we had written `a = Basket("saw")` (without the square brackets) the `print_me()` method would have returned this: `Basket contains: 's' 'a' 'w'`.

Basically we have `object.method(arg1, arg2)`

In the above case `a` is the object and we have the methods `add()` and `print_me()`.

`__init__` is a special method that is said to be a constructor setting things up in the first place. The **constructor** (initial call to the routine) creates an **instance** of the object (in the above case it creates a basket we can put things in).

3.16. Writing Scripts - Programming in DP

Scripts take individual DP statements and combine them to make more complex routines. The user can edit a script directly in the edit/debug window of JIDE. A series of DP commands/instructions can then be input and then run in the DP environment.

Following on from our Basket example. If the class Basket has already been created we can create a script that uses it. For example, we can place the following in our JIDE edit pane.

```
a = Basket()
a.add("saw")
a.add("hammer")
a.add("chisel")
b = Basket()
b.add("bread")
b.add("cheese")
b.add("milk")
a.print_me()
b.print_me()
```

Now if we hit the "Run all" button then we create two baskets the contents of which will be printed to the command window (bottom left).

This script can be saved using the "File" pulldown menu or save icon (default is ".py" extension).

3.17. Some Useful Extra Items on Scripts

- Some arguments can be optional and can be given a default value. E.g.,

```
def spam(age=32):
    tammy_age = age - 5
    print "Tammy is ", tammy_age
    print "Tammy's brother is ", age
```

Here, spam can be called with zero or one parameter. If no parameters are given it will be called with the default parameter of age=32. If a parameter is given with the call then that will be assigned to age instead.

Our little script can now be run using, for example,

```
spam()
spam(age=34)
```

- Backquotes (`) convert an object to its string representation (so the number 1 can be converted to string "1").

```
age = 32
message = "Tammy is "+`age`
print message
```

Here we add (via the plus sign) the string value of age to our message.

- The + sign can be used to append string lists.
- One change to make printing easier. We can change to the special method `__str__` so that our last function starts with the line

```
def __str__(self):
```

Instead of

```
def print_me(self):
```

We should also change

```
print "Basket contains: " + result
```

to

```
result = "Basket contains: " + result
return result
```

Now we can use

```
print a
```

to show our basket contents rather than

```
a.print_me()
```

Most useful classes and functions are put into modules or packages. These are then imported into a given environment or program with the line(s), e.g.,

```
import math
from math import sin, cos
```

This is the means by which classes and functions are brought into the DP environment from within already existing packages. The above two lines show how to import the `math` package and how to just import `sin` and `cos` classes from the `math` package into your current DP session.

A basic set of packages most relevant to users is loaded when a DP session is started. Other packages can simply be imported into a user's session (or included in an `import.py` file that automatically imports packages when DP is started - see ???).

3.18. Interactivity in Jython Scripts

Sometimes all we need is a script that is launched, performs all its calculations without asking anybody, and then outputs the result and exits. Other times we would like the user to interact, give input while the script is running, take decisions that influence what the script will do. This section takes a look at the tools Jython offers to do just that.

3.18.1. Basic Interactivity

The most common case is for the script to ask the user to input a value. We can use the `raw_input` function, as the tiny example that follows demonstrates.

```
myAnswer = ""
myAnswer = raw_input("Please write something, anything\n")
print "You wrote " + myAnswer + "\nWell done."
```

Here is an interesting fact. When we run this script in JIDE, a small window pops up (see Figure 3.1) with the text we passed to `raw_input`, a box where we can input text and two buttons, OK and Cancel. Save this script and call it `tinyScript.py`, then execute it from the command line, outside JIDE, issuing `python tinyScript.py` or `jython tinyScript.py`, or try double-clicking on the file icon. You will see no fancy windows this time, everything will happen inside a text console. In other words, the window we got is a feature courtesy of JIDE, not a Jython feature.

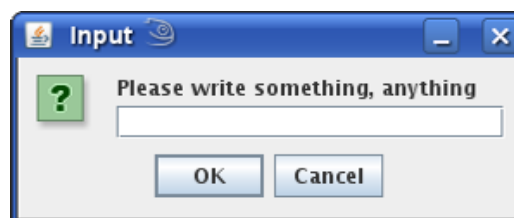


Figure 3.1. The window that appears calling the `raw_input` function from within JIDE.

**Warning**

Remember that `raw_input` takes everything the user inputs and turns it into a string, including numbers. So be careful when comparing this input to other numbers: you might need to cast your variable to a numerical type.

A fundamental flaw of our little example is that it does not check the input in any way. We could even get away with writing absolutely nothing in the text box, and JIDE would give the seemingly sarcastic reply

```
You wrote
Well done.
```

Of course if we had initialised `myAnswer` to anything else than an empty string, we would get that value in the output. Worse still, if we press the Cancel button, regardless of whether we wrote something or not, the `myAnswer` variable will be set to `None` and the following line will give an error.

One way to have the user input something sensible is to embed the request into a `while` loop, as the following example demonstrates.

```
myAnswer = ""
while myAnswer == "":
    myAnswer = raw_input("Write something, anything\n")
if myAnswer == None:
    myAnswer = ""
print "You wrote " + myAnswer + "\nWell done."
```

This way the window will not go away until we write something and press OK, and if we try to bypass the check by pressing Cancel the following `if` clause will at least prevent an error on the last line.

More complicated checks can be put in place, for example to make sure that a numerical value stays within the allowed range, and more sophisticated loops may be needed, but the principle is the same.

The above example can also be useful when we want to stop the execution of a script, for whatever reason, and wait before resuming it until the user lets us know that he is in front of the computer and is paying attention. In this case the input does not matter at all, since we just want the user to acknowledge a request by pressing a button.

Well, it works but it is far from optimal. Why having a box for entering text if the text itself does not matter? Wouldn't it be much better to have a window with *Press OK to continue* written on it, the OK button, and nothing else? This is the subject of the next section.

3.18.2. A Little Bit of Swing

To put it simply, *Swing* is the name given to that part of Java that deals with creating graphical user interfaces (or GUIs). Yes, you read correctly: Java, not Jython. Please do not let this scare you. We have used Java bits before, almost without realising it (after all, it is what makes Jython so powerful) and this case will not be different. As a matter of fact, using Swing within Jython is easier than doing so within Java.

This section will teach you enough about Swing to get you started, but if you want to become a GUI guru you may want to look elsewhere. The first chapter of the *Jython Essentials* book has something more to say about Swing. You can find it here:

<http://www.oreilly.com/catalog/jythoness/chapter/ch01.html>

3.18.2.1. `showMessageDialog`

The first thing we will do is to invoke a Swing method to display a message in a window, together with an OK button:

```
from javax.swing import *
```

```
print "Let's stop for a while"
JOptionPane.showMessageDialog(None, "Press OK to continue")
print "Well done."
```

The first line imports the swing package (yes, it is `javax` rather than `java`). Then we have the line creating the window, embedded between two lines printing text messages to demonstrate that the script will not advance until we press the OK button.

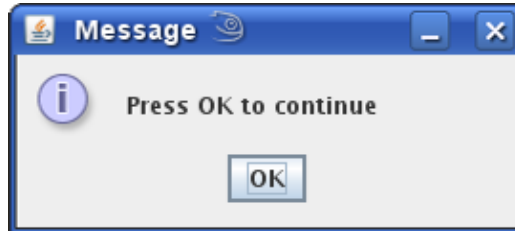


Figure 3.2. The window that appears calling the Swing `showMessageDialog` method.

You have probably noticed that the `showMessageDialog` method takes two parameters, and we have set the first one to `None`. It is used to indicate the "parent" element of the dialogue box we are creating. In this case (and in everything that follows) we are just creating a single window and nothing else, so we will not worry about this parameter anymore.

Actually the `showMessageDialog` can take *more* than two parameters. Notice that the text in the title bar of our window was just "Message". In order to customise it we have to add another parameter, like this:

```
JOptionPane.showMessageDialog(None, "Press OK to continue", "Title bar text")
```

Try this and you will get... an error. This is because this third argument *must* go with a fourth one, telling what kind of window we are creating. Let us try again:

```
JOptionPane.showMessageDialog(None, "Press OK to continue", "Title bar text", \
JOptionPane.ERROR_MESSAGE)
```

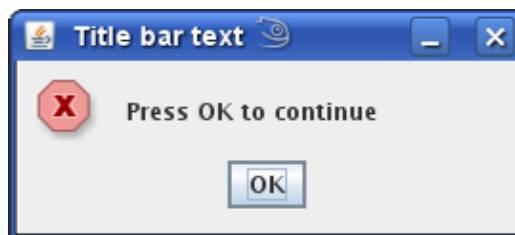


Figure 3.3. Customising the icon and the window title.

Now it works, and it even allows us to change the icon to a nice "error" one. There are a number of possibilities for this fourth parameter, all of which are self-explanatory: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE` and `PLAIN_MESSAGE`. Feel free to try them at your leisure.

If you are sharp-eyed you might have noticed that the previous error message said "expected 2 or 4-5 args; got 3". This mysterious fifth argument is used to add a custom icon to the window, in case you are not satisfied with the predefined ones. Since this is pure eye candy and adds nothing to the functionality of the window, we will not cover it here.

3.18.2.2. `showInputDialog`

Now we would like to take it a step further and create a window for entering text, just like we did with the `raw_input` function. We just have to use a different method, like this:

```
myAnswer = JOptionPane.showInputDialog(None, "Please write something, anything")
```

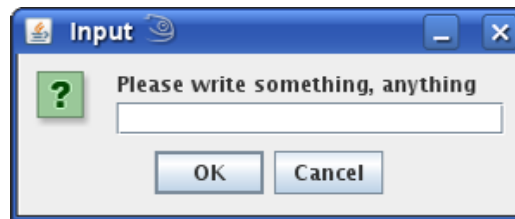


Figure 3.4. The window that appears calling the Swing `showInputDialog` method.

You can put this line in the scripts we used to describe the `raw_input` function and you will obtain the same behaviour, quirks included (even the two windows look exactly the same). The big difference is that, even if you are launching the script from a command line interface *outside* JIDE, a window will still pop up.

Granted, a wealth of additional options is available for this method as well. The ones we saw before are still valid:

```
myAnswer = JOptionPane.showInputDialog(None, "Please write something, anything", \
"Big question", JOptionPane.QUESTION_MESSAGE)
```

But there is more. We can put a default string of text in the box like this:

```
myAnswer = JOptionPane.showInputDialog(None, "Please write something, anything", \
"Default text")
```

If we want the user to choose from a predefined set of options, we can use the `showInputDialog` with a whopping seven parameters, as the following script demonstrates:

```
from javax.swing import *
myAnswer = ""
possibleAnswers = ["HIFI", "PACS", "SPIRE", "No clue", "All three"]
while myAnswer == "":
    myAnswer = JOptionPane.showInputDialog(None, "Favourite Herschel instrument?", \
"Test", JOptionPane.QUESTION_MESSAGE, None, possibleAnswers, possibleAnswers[2])
if myAnswer == None:
    myAnswer = ""
print "Your answer is: " + myAnswer
```

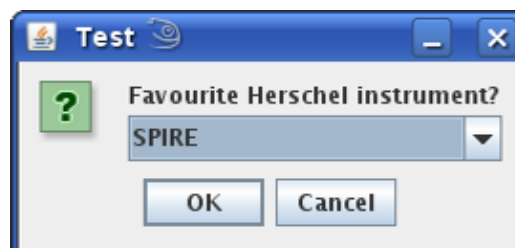


Figure 3.5. A more complex window with a combo box.

Let us go through the parameters one by one:

1. None: the "parent" element.
2. "Favourite Herschel instrument?": the window text.
3. "Test ": the window title text.
4. `JOptionPane.QUESTION_MESSAGE`: the type of window.
5. None: the custom icon. We choose to provide no one and stick with the default one.

6. `possibleAnswers`: the array of possible answers.
7. `possibleAnswers[2]`: the default answer (showing the personal bias of the author).

3.18.2.3. `showConfirmDialog`

Next we take a look at the `showConfirmDialog` method, which can be used to display a window asking the user to confirm or block a certain action. One example will clarify what we mean:

```
from javax.swing import *
myAnswer = JOptionPane.showConfirmDialog(None, "Yes or no?")
if myAnswer == 0:
    print "You agree"
elif myAnswer == 1:
    print "You disagree"
else:
    print "You have no opinion on this"
```

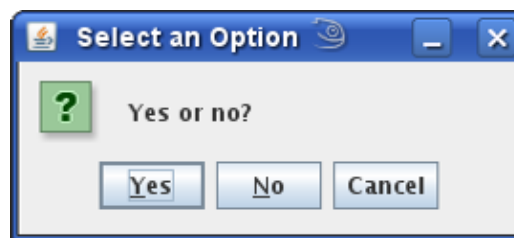


Figure 3.6. Using the Swing `showConfirmDialog` method.

Note that we can use predefined constants to make the code easier to understand, if a little more verbose, as the following, slightly expanded example shows:

```
from javax.swing import *
myAnswer = JOptionPane.showConfirmDialog(None, "Yes or no?")
if myAnswer == JOptionPane.YES_OPTION:
    print "You agree"
elif myAnswer == JOptionPane.NO_OPTION:
    print "You disagree"
elif myAnswer == JOptionPane.CANCEL_OPTION:
    print "You have no opinion on this"
elif myAnswer == JOptionPane.CLOSED_OPTION:
    print "You closed the window. How rude!"
```

As always we are free to make things more complicated than that. We can add another two parameters to provide a title for the window and the type of buttons we want:

```
myAnswer = JOptionPane.showConfirmDialog(None, "Yes or no?", "Question", \
    JOptionPane.YES_NO_OPTION)
```

Here we decided to drop the Cancel button. Other possible options are `YES_NO_CANCEL_OPTION`, `OK_CANCEL_OPTION`, both self-explanatory, and `DEFAULT_OPTION`, which will just display an OK button.

3.19. Useful Java bits

The Jython language is an implementation of Python written in Java, which means that it is as good-natured yet powerful as Python, but with the added benefit of thousands of packages and classes developed for Java. We will be using some of these classes in the next chapters, and here is a brief description of what they do.

- **The `java.awt` package.** As you already know a *package* is a collection of related classes, like a binder on your desk keeping related documents together. The `java.awt` package contains all of

the classes for painting graphics and images. It will be particularly useful in Chapter 6 for plotting and Chapter 7 for viewing images.

- **The `java.awt.Color` class.** With this class you can specify a colour for an object. There are thirteen predefined colours available: `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE` and `YELLOW`. If you feel you need a fancier shade you can provide the red, green and blue values individually, as three `ints` between 0 and 255 or `floats` between 0.0 and 1.0, like this: `java.awt.Color(0.3, 0.2, 0.5)`. You can also add the alpha (transparency) value as a fourth parameter: 0.0 means completely transparent and 1.0 completely opaque.
- **The `java.awt.Font` class.** This class allows you to select fonts for annotations on your graphical objects, together with their style and size. The syntax of the constructor (i.e. the special method called to instantiate an object from a class) is like this: `Font("SansSerif", 0, 64)`, where we have the font name, its style code (0 for plain, 1 for bold, 2 for italic) and its size in points.
- **The `java.awt.Window` class.** This class deals with the drawable area of a window on your desktop (not with borders or menu bars). One useful method, especially for plotting, is `setLocation`, inherited from `java.awt.Component`. It accepts two `int` parameters, the `x` and `y` position of the top left corner of the object you want to move.

For more information on these and other classes of the standard Java API you should browse the official Javadoc. If you are looking for a less traumatic introduction to the Java language, the Java Tutorial is an excellent resource.

3.20. Jython and DP Quirks

Every programming language or software system has its *quirks*. Jython and DP are no exception, and this section deals with some of the features you might find confusing.

3.20.1. Two functions for one goal

There are some mathematical function in DP existing in two forms, one in the usual *FirstLetterCapitalised* form (the so-called *CamelCase* convention), the other in *UPPERCASE*. The first form is the recommended way to go, since it is consistent with the rest of the system; the alternative syntax (technically known as *Jython wrapper*) is being kept for backward compatibility, but is not recommended for use in new code and is no longer described in this manual. Examples of Jython wrappers are `MATMUL` and `SOLVE` instead of the classes `MatrixMultiply` and `MatrixSolve`, or `RESHAPE` instead of `Reshape` to change the shape of arrays. You might still bump into them when browsing legacy code.

Unfortunately Jython wrappers are not the only names in uppercase letters, so this is not a good way to identify them, since also e.g. *static instances* (see Section 3.20.3) such as `SIN` and `COS` use the same convention.

3.20.2. Long Names versus Short Names

The general rule used in developing the classes used in the DP system is to use long descriptive names, e.g., `TableDataset` rather than `TDset`. An exception to the rule is, e.g., `IOException` rather than `InputOutputException`

The general rule is that a class name must be self descriptive (easier to remember) which sometimes conflicts with the requirement "I should do every thing by typing three-six letters". The latter was a restriction in F77, and language developers fortunately diverted from that (as it introduced names like `CCDF12`, `CCEFLT`, `EMPXFF`), which are indeed less typing but make the code less (if not completely un-) readable. Exceptions are usually dealing with "well-known"

abbreviations. Acronyms such as "IBM Type Writer" is taken to become "IbmTypeWriter" rather than "IndustrialBusinessMachinesTypeWriter."

Any Jython user can create aliases by do things like:

```
TDS=TableDataset
t1=TDS(description="Hello world, this is still a tabledataset!")
print TDS
# herschel.ia.dataset.TableDataset
print t1
# {description="Hello world, this is still a tabledataset!", meta=[], columns=[]}
print t1.__class__
# herschel.ia.dataset.TableDataset
```

Here, in effect, we have created a shortened version of the command we can use to set up a TableDataset called "TDS". We then create a TableDataset, called "t1", which initially contains only a description in the second line. This is equivalent to writing

```
t1=TableDataset(description="Hello world, this is still a tabledataset!")
```

The last two lines indicate the contents of "t1" and the class that created it.

3.20.3. Naming conventions

A potentially confusing aspect to the naming of DP classes is the mix of upper- and lower-case letters. A comprehensive description of the naming convention used in the HCSS is given in Appendix E and here we just shortly describe the most important aspects. The upper-case/lower-case scheme used in predefined DP classes has the following conventions.

- *Classes*

Class definitions have names that consist of words of which each first letter is capitalised:

```
MyOwnClass
TableDataset
HifiProduct
```

- *Class instances -- objects*

Objects (variables) of a particular class have names that should start with the first letter in lower case. In general, this translates to

```
myOwnClass=MyOwnClass(...)
table=TableDataset
a=2
```

- *Class instances as constants*

Certain class instances (or simple variables) are used as constants. The convention is to use names with all their letters capitalised and words separated by an underscore '_'. These are sometimes referred to as static instances. An example is SIN: it is the only (allowed) instance of class Sin, as it does not make sense to have multiple instances of these. Examples are:

```
VARIANCE
IS_FINITE
ALL_PRESENT
```

3.20.4. Miscellaneous quirks

- **Working directories.** Restrictions are placed on dealing with working directories due to the use of Java. This is discussed in Section 2.6.

- **Loops, indentation and blank line usage.** Indentation in loops is very strict within JIDE. Blank lines can have particular significance, particularly with respect to setting up loops. These quirks are described in Section 2.8.
- **Logical operators.** The presence of Jython original features together with DP specific ones can result in counter-intuitive behaviour and unexpected results Section 5.7 in Chapter 5 deals with these quirks.



Warning

Each jython script is compiled by the Java virtual machine into one single non-native, non-abstract method and such Java methods cannot exceed certain limit, usually 65536 bytes. If your jython script is very long (more than few thousands of lines) then it is advisable to split it into separate scripts.

Chapter 4. Handling Array Data Objects, Datasets and Products

4.1. Introduction

This chapter aims to familiarize the user with the DP Array data objects, Datasets and Algorithms concepts. This is not an exhaustive reference to all the functionality provided, the full set of available array object and dataset capabilities are discussed in the *herschel.ia.numeric* and *herschel.ia.dataset* packages Javadoc.

There are three types of basic datasets:

- array datasets (datasets containing single `ArrayData` objects, holding numbers, strings, etc. in 1D, 2D, 3D, 4D or 5D)
- table datasets (x rows by y columns of numeric or string arrays). Table datasets can have columns of various data types mixed in the same dataset and can also contain unit and descriptive information for individual columns.
- composite datasets (combines multiple connected arrays/tables in a single dataset).

One of the major advantages of DP numeric array objects (as opposed to Jython lists) is the ability to do array arithmetic in single line commands rather than having to loop through arrays.

In this chapter, we discuss how to formulate and use each array object and dataset type.

4.2. Getting started


All classes and methods associated with handling datasets and numeric functions are automatically loaded when the DP session is started in this manner.

The DP *numeric* package currently contains many functions and is discussed in more detail in Chapter 5. Here we include the use of portions of it to help illustrate how datasets may be handled.

4.3. Types of Array Data Objects

DP numeric array data objects can have up to 5 dimensions and have the types shown in the following table.

Table 4.1. Numeric types available in DP (N = 1...5)

Name	Type	Dimensions		
		1	2	3+
BoolNd	boolean	yes	yes	yes
ByteNd	byte	yes	yes	yes
ShortNd	short	yes	yes	yes
IntNd	integer	yes	yes	yes
LongNd	long	yes	yes	yes
FloatNd	float	yes	yes	yes
DoubleNd	double	yes	yes	yes
ComplexNd	complex	yes	yes	yes
String1d 	string	yes	NO	NO

- ❶ The `String1d` array type is not strictly numeric.

4.3.1. DP Numeric Array Access and Slicing

The numeric package introduces the following square brackets notation:

```
[i_0, ..., i_n-1]
```

where each element is separated by a comma, and the number of elements must be equal to the rank of the array. Arrays are zero-based which means the first element of an array has index 0 (zero) and the index of the last element of an array is `array.length()-1`.

In addition the package supports the colon (`:`) notation to designate a slice. A slice is a range of indices defined as `i:j`, where `i` is the starting index and inclusive, and it is zero if not specified. The ending index `j` is exclusive and it is equal to `array.length()` if not specified and `array.length()-j` if negative.

The following example illustrates the access to elements in a multi-dimensional array and the use of slices. More examples can be found in the section on Multi-Dimensional Arrays.

```
# define something that is like a rectangular 2x3 array:
#  1 2 3
#  4 5 6
x=Int2d([[1,2,3],[4,5,6]])# Int1d can swallow the jython sequence.
print x                # [[1,2,3],[4,5,6]]
print x[1]             # 2 (second element of the first row)
print x[1,:]           # access a row i.e. [4,5,6]
print x[1,1]           # access an individual element i.e. 5
print x[:,:]           # [[1,2,3],[4,5,6]]
print x[:,1]           # access a column i.e. [2,5]
```

4.4. Creating a Simple 1D DP Numeric Array

In order to create an array data object we only need to do something like the following:

```
a = Int1d()
```

This provides us with an empty integer array. We can now add elements to this by

```
a.append(2)
```

Or

```
a.append(Int1d([1,2,3,4,5]))
```

to append a whole 1D integer array.

Alternately, we could have created the array in one go, like this:

```
a = Int1d([1,2,3,4,5])
```

The following show various ways in which numeric 1D arrays can be created in the DP environment.

```
y = Double1d([1.0,2.0,3.0,4.0]) # Create from a Jython array
y = Double1d(4) # [0.0,0.0,0.0,0.0]
y = Double1d(4, 42.0) # [42.0,42.0,42.0,42.0]
y = Double1d.range(4) # [0.0,1.0,2.0,3.0]
```

4.5. Creating and Handling Complex Array Data Objects

The numeric library has a `Complex` class and a `ComplexNd` class for N-dimensional arrays of complex numbers (N = 1, 2, 3, 4 or 5).

```
z = Complex1d([1,2,3,4],[4,3,2,1]) # Set up complex array
print z # [(1.0+4.0j),(2.0+3.0j),(3.0+2.0j),(4.0+1.0j)]
print z.getReal() # Print real part
print z.getImag() # Print imaginary part
print z.conjugate() # [(1.0-4.0j),(2.0-3.0j),(3.0-2.0j),(4.0-1.0j)]
```

Complex numbers in the numeric package are constructed using the `Complex` constructor (with an upper-case 'C'):

```
z1 = 2 + 3j # Jython complex (2+3j)
z2 = Complex(2,3) # Numeric Complex (2.0+3.0j)
```

In other respects, `Complex` arrays are used in much the same way as `Double` arrays. Their main use, at present, is for discrete Fourier transforms.

4.6. Creating and Accessing Multi-Dimensional Array Data Objects

Creating and manipulating multi-dimensional arrays occurs in a similar way to the 1D case. The DP numeric library supports arrays of up to 5 dimensions. For example, to create a `Double2d` array:

```
x = Double2d([[2,4,6],[1,3,5]])
```

Multi-dimensional arrays are conceptually arrays of lower-dimensional arrays. For a two-dimensional array, the first subscript selects a row and the second subscript selects an element within that row (the column).



Note

This is the opposite order to some other computer languages, but it is the same behaviour as in the Java programming language.

For example:

```
print x[1,:] # Get row 1 i.e. [1.0,3.0,5.0]
print x[1,2] # 5.0, the element in row 1, column 2
```

Note: indexing multi-dimensional arrays is done differently in DP numeric arrays as compared to Jython arrays. The following code examples show the syntax for Jython and DP numeric arrays. The reason for this is to allow slicing on multi-dimensional arrays in DP which is technically not possible using the Jython syntax.

```
# Jython array:
x = [[1,2,3,4],[5,6,7,8]]
print x[1][2] # 7
print x[1][1:3] # 6, 7
```

```
# DP numeric array:
y = Int2d([[1,2,3,4],[5,6,7,8]])
print y[1,2] # 7
print y[1,1:3] # 6, 7
```

Individual elements or slices can be set as follows:

```
x[1,2] = 22 # Set an element in place
x[0,1:3] = 42
print x # [
# [2.0,42.0,42.0],
# [1.0,3.0,22.0]
# ]
```

It is possible to set a row to a copy of a 1d array of the same length:

```
x[0,:] = [5,6,7,8] # Set a row to (a copy of) a Jython array
y[1,:] = Int1d([9,7,6,5]) # Set a row to a Double1d array
```

4.7. Adding Attributes to Create an Array Dataset

Let's start by creating a simple dataset. Let's assume that we want to create a dataset containing one component: a 1D array of double precision numbers (doubles in an array we will call 'x').

Type in the following steps (without the comments preceded by '#'):

```
x = Double1d.range(10) # ❶
s = ArrayDataset(data=x,description="range of double values") # ❷
```

- ❶ The `range()` function creates a 1D array of integers with the values 0, 1, 2...9. Putting `Double1d` in the front converts the array values to doubles.
- ❷ This actually creates the array dataset with data being the array `x` of values 0.0, 1.0, 2.0...9.0 and some associated information, a description.

This creates an object `x`, corresponding to a 1D array of 10 doubles from 0 to 9, and writes that to a dataset object, `s`, which also contains a description of the dataset. The `range` command produces ten integer numbers from 0 to 9. This is placed in a 1D array of doubles by the first line.

Now let's look at the contents of the dataset `s`:

```
print s
```

If you want to be specific and print individual components of the dataset, you may do so using the special description and data attributes:

```
print s.description # Just print the description that is attached to the dataset
print s.data # Print only the data contained in the dataset
```

And even individual elements of the data component:

```
print s.data[2] # View the value of the third element of the array
# contained in the dataset
```

4.7.1. Dataset Attributes and Metadata

In the previous section, we have seen that the `ArrayDataset` `s` possesses at least 2 attributes: `description` and `data`. They have in addition a third attribute not so far illustrated, `meta`. The `description` and `meta` attributes are common across all dataset types.

The **description attribute** is used to store a human-readable text that helps the user to understand the role of the dataset.

The **meta attribute** stores a map of keyword-value pairs of data that can be used to identify that data in a database (for example) - the so-called *meta-data*. Examples of metadata for an observation include *the date of the current observation; the name of the source; the coordinates of the source*, etc. **These are basically the DP equivalent of FITS keywords**. The allowed data types for meta-data elements are String, Double, Boolean, Long, and Date (e.g., `StringParameter`, `DoubleParameter` etc.). See the JavaDoc on the class `MetaData` for more information on the allowed types.

The following code snippet shows how to add parameter information (in the form of strings or doubles) to the `meta` attribute:

```
s.meta["observation"] = StringParameter("NGC 4151")
s.meta["principal investigator"] = StringParameter("Anthony Marston")
s.meta["ra"] = DoubleParameter(182.836)
s.meta["dec"] = DoubleParameter(39.405)
```

These are actually shortcuts to Java usage. For example, the first line could also have been written as

```
s.getMeta().set("observation", StringParameter("NGC4151"))
```

4.8. Creating and Viewing a TableDataset

What is often required is to store data in a tabular format with N columns. The `TableDataset` provides such a means. A `TableDataset` is made up of a number of columns. Each column contains an `ArrayDataset` (data), a description and a quantity (unit -- require the `Unit` package import, see below) value associated with the `ArrayDataset`. Each `ArrayDataset` can have up to 5 dimensions and can be of varying types. In the following example, a `TableDataset` is created with 3 columns each containing a 1D dataset, one being a sequence of numbers from 1 to 100, the second being the sine value of each of the numbers in the first column, and the final column containing the values in the first column multiplied by 100. The column names are `x`, `sin` and `y` respectively.



Note

For reasons of flexibility, memory consumption and performance, this class is not checking whether all columns are of the same length: this is the responsibility of the user.

```
from herschel.share.unit import * # to allow the use of the Unit package

x = Double1d.range(100)
t = TableDataset(description="This is a table") # ❶
t["x"] = Column(data=x, unit=Duration.SECONDS) # ❷
t["sin"] = Column(data=SIN(x),description="sin(x)") # ❸
t["y"] = Column(data=x*100,description="x*100")
```

- ❶ This sets up the table dataset with an associated description
- ❷ This creates our first column which has the data, `x` and its associated units, which in this case is a time duration of `SECONDS`.
- ❸ Here we have applied the `SIN` function from the numeric package, and we have also added a description for the second column.

`TableDatasets` can be viewed using the `DatasetInspector` GUI button. Values can also be obtained using the following steps which show how the data can be listed (plotting the data graphically is discussed in Chapter 6):

```
print t # Print a TableDataset called t (see above)
print t.meta # Print the metadata (empty in this case)
print t["x"] # Print a column by name
print t[2] # Print a column by index
print t[2].data # Print the data inside the column
a = t[2].data # Assign data in column to a list variable, "a".
print t[2].data[4] # Print element with index=4 in the last (third!) column
b = t[2].data[4] # Assign the data value to variable "b".
```

```
print t[2].description # Prints column description only
print t["x"].unit # print the associated unit values for the column
```

Alternately, we can access columns via the `getColumn` method

```
print t.getColumn("y") # Print a column by name
print t.getColumn(2) # Print a column by index
print t.getColumn(2).data # Print the data inside the column
print t.getColumn(2).data[4] # Print element with index=4 in the third column
print t.getColumn(2).description # Prints column description only
```

We can also get row values

```
print t.getRow(1) # Gets a list of the values in the second row.
```

And here is how data can be modified:

```
print t["y"].data[0]
t["y"].data[0]=999.
print t["y"].data[0]
```

We may also get and set values at a position in a `TableDataset`.

```
t.getValueAt(0,1) # gets the value contained in row=0, column=1
t.setValueAt(30.5, 0, 1) # sets the value 30.5 at row=0, column=1
```

4.8.1. Row-wise appending of TableDatasets

It is possible to append the data from one table dataset to data in another, provided that they have the same number of columns and each column in either dataset is of the same type. The following example adds `t2` as a row to table `t1`.

```
t1 = TableDataset()
t1["x"] = Column(data=Int1d.range(5))
t1["y"] = Column(data=Double1d.range(5))
t2 = TableDataset()
t2["a"] = Column(data=Int1d.range(10))
t2["b"] = Column(data=Double1d.range(10))

# The following will append the data in t2 to the data in t1
# t1.rowCount will then report 15 rows:
t1.addRow(t2)
```

If we now use `print t1["x"].data` we can see that the "x" column has the values `[0,1,2,3,4,0,1,2,3,4,5,6,7,8,9]`.

4.8.2. Assigning Units

This section explains what units can be assigned and how they may be manipulated. As we have noted above, we can assign units to the columns in our dataset. In order to use the `Unit` package we have to import it:

```
from herschel.share.unit import *
```

Note that the `Unit` package are used in the whole `HCSS` and not only in the interactive analysis, that is why it is part of the `herschel.share` library.

The units fall into several category types, as they are shown in alphabetical order in Table 4.2. To assign a unit the type and value `s` required to be given. For example -- the variable "a" can be assigned to be a unit of angle in degrees with


```
a = Angle.DEGREES # Type.VALUE
```

This can be associated with a column's unit in a table using

```
t["x"].unit = Angle.DEGREES
```

Table 4.2. All available basic units types

Type	VALUES
Acceleration	METERS_PER_SECOND_SQUARED
Angle	RADIANS, DEGREES, MINUTES_ARC, SECONDS_ARC
AngularMomentum	JOULE_SECOND
AngularSpeed	RADIANS_PER_SECOND, DEGREES_PER_SECOND
Area	SQUARE_METERS, SQUARE_KILOMETERS
Constant	H_PLANCK, K_BOLTZMANN, ELECTRON_CHARGE, SPEED_OF_LIGHT
Duration	SECONDS, MINUTES, HOURS, DAYS
ElectricCapacitance	FARADS, MILLIFARADS, MICROFARADS, NANOFARADS, PICOFARADS
ElectricCharge	COULOMBS
ElectricConductance	SIEMENS
ElectricCurrent	AMPERES, MILLIAMPERES
ElectricInductance	HENRIES
ElectricPotential	VOLTS, MILLIVOLTS
ElectricResistance	OHMS
Energy	JOULES, ERGS, ELECTRON_VOLTS
Entropy	JOULES_PER_KELVIN
Flux density	JOULES_PER_SQUARE_METER, JANSKYS, MILLIJANSKYS, MICROJANSKYS
Force	NEWTONS, DYNES
Frequency	HERTZ, KILOHERTZ, MEGAHERTZ, GIGAHERTZ, TERAHERTZ
Length	METERS, ANGSTROMS, KILOMETERS, CENTIMETERS, MILLIMETERS, MICROMETERS
Mass	GRAMS, KILOGRAMS
NEP (Noise Equivalent Power)	WATTS_PER_SQRT_HERTZ
Power	WATTS, KILOWATTS, MEGAWATTS
Pressure	PASCALS, BARS, MILLIBARS
Scalar	This class represents scalar units and provides some constants:ONE, PERCENT,DECIBELS
SolidAngle	STERADIANS, SQUARE_MINUTES_ARC, SQUARE_SECONDS_ARC
Speed	KILOMETERS_PER_SECOND, METERS_PER_SECOND
Temperature	CELSIUS, KELVIN
ThermalConductivity	WATTS_PER_METER_KELVIN
TimeInstant	TAI, UTC
WaveNumber	RECIPROCAL_METERS, RECIPROCAL_CENTIMETERS

4.8.2.1. Manipulating Units

We may manipulate units to obtain derived units. Examples are the following

```
N = Force.NEWTONS
m = Length.METERS
m2 = m**2           # Square meters
Pa = N / m2         # Pascals
J = N * m           # Joules
```

4.8.2.2. Converting Units to Strings and Back Again

We can convert a unit variable to a string in several ways:

```
A = Length.ANGSTROMS
print A           # angstrom [1.0E-10 m], no conversion
print A.name      # angstrom. This is a string quantity.
print A.dialogName # Angstrom symbol. This is a string quantity.
um = Length.MICROMETERS
print um          # micrometer [1.0E-6 m], no conversion, includes factor
                  # with respect to SI unit
print um.name     # micrometer, only ASCII characters. This is a string.
print um.dialogName # μm. This is a string quantity.
```

We can also convert a string to a unit

```
print Unit.parse("km s-1")
# or print (Unit.parse("km") / Unit.parse("s"))
print Unit.parse("km s-1") # Speed.KILOMETERS_PER_SECOND
print Unit.parse("arcsec") # Angle.SECONDS_ARC
print Unit.parse("eV") # Energy.ELECTRON_VOLTS
print Unit.parse("cm") # Length.CENTIMETERS
print Unit.parse("mm") # Length.MILLIMETERS
print Unit.parse("microm") # Length.MICROMETERS)
```

4.8.2.3. Derived Units

We can also provide derived units by application of `.milli`, `.micro` and `.nano` methods.

```
s = Duration.SECONDS
us = s.micro # micro seconds
ns = s.nano # nano seconds
```

4.8.2.4. Conversion to SI and Other Units

If the SI unit is needed rather than the unit used then SI unit and the factor between the two can be provided.

```
print Angle.DEGREES.asSI # gives unit as Angle.RADIANS
print Energy.ERGS.asSI # gives unit as Energy.JOULES
print Speed.KILOMETERS_PER_HOUR.asSI # gives unit as Speed.METERS_PER_SECOND
print Unit.parse("g cm s-2").asSI # gives unit as Unit.parse("kg m s-2")
#
print Length.ANGSTROMS.toSI # 1.0E-10
print Duration.HOURS.toSI # 3600.0
print FluxDensity.MILLIJANSKYS.toSI # 1.0E-29
print Unit.parse("g cm s-2").toSI # 1.0E-5
# or factor compared to other units
min = Duration.MINUTES
ms = Duration.MILLISECONDS
print min.to(ms) # 60000.0
mV = Unit.parse("mV") # millivolts
print mV.to(mV.asSI) # 0.001; same as mV.toSI
```

4.8.2.5. Physical Constants

Physical constants can also be provided to the system with their correct units, e.g.

```
h = Constant.H_PLANCK
print h.value # 6.62606896E-34
print h.unit # J s
print h # 6.62606896E-34 J s
k = Constant.K_BOLTZMANN
print k.value # 1.3806505E-23
print k.unit # J K-1
print k # 1.3806505E-23 J K-1
```

4.8.2.6. Unit Compatibility

We can compare units to see if they are of compatible types.

```
kg = Mass.KILOGRAMS
g = Mass.GRAMS
m = Length.METERS
print kg.isCompatible(g) # true
print kg.isCompatible(m) # false
print kg.isCompatible(Mass) # true
print kg.isCompatible(Area) # false
print Unit.parse("g cm s-2").isCompatible(Force) # true
print Unit.parse("g cm s-2").isCompatible(Power) # false
```

4.8.2.7. Unit Equivalence

We can use the `.isEquivalent` method to determine if two unit types are the same.

```
kg = Mass.KILOGRAMS
s = Duration.SECONDS
m = Length.METERS
N = Force.NEWTONS
dyn = Force.DYNES
print N.isEquivalent(dyn) # false
print N.isEquivalent(kg * m / s**2) # true
```

4.9. Creating and Accessing a Composite Dataset

The `ArrayDataset` and `TableDataset` types enable the user to encapsulate arrays and tables of primitive data types easily. However, they do not allow arbitrary structures of data, or data within data, to be constructed. Examples of complex datasets are grouped observations (making a map with an offset reference position, for instance), which could have 1D and 2D array data together with a table which might contain (for example) calibration data. Such complex structures can be built using the `CompositeDataset`. Example 4.1 creates a `CompositeDataset` containing in turn an `ArrayDataset`, a `TableDataset`, a few `StringParameters`, and another nested `CompositeDataset`. It also illustrates how we can access the components of the composite dataset.

```

# First we set up a one-dimensional array of doubles (0.0, 1.0 ... 9.0)
x = Double1d.range(10)
# Then we create an array dataset with an added description
s = ArrayDataset(data=x,description="Range of doubles")
# This sets up an empty table with a description
t = TableDataset(description="This is a table")
# The array 'x' is then added to the table and given a
# column heading "x"
t["x"]=Column(x)
# Each of the array elements of 'x' is multiplied by 4
# and becomes the data in the table column labeled "y".
# The table column also has a description added to it.
t["y"]=Column(data=x*4,description="x*4")
# c is an empty composite dataset.
c=CompositeDataset()
# We add a description to c
c.description="This is a composite dataset. It contains three datasets!"
# We add the author's name as a string parameter
c.meta["author"]=StringParameter("Jorgo Bakker")
# We input a version number as a string parameter
c.meta["version"]=StringParameter("2.0")
# We put the array dataset s into the composite dataset c
# and give it the name mySimple so that we can refer to it
c["mySimple"] = s
# We do the same for the table
c["myTable"] = t
# This just shows you can add a composite dataset into another
# composite dataset (nesting)
c["myNest"] = CompositeDataset("Empty nested composite dataset")

print c          # View contents of the complex dataset.
tab = c["myTable"] # Gets our TableDataset back. Now called "tab".
print tab       # We see that it has two columns called "x" and "y"
print tab["x"]  # Prints out what is in the "x" column.
print tab["x"].data # To just print out the data values.

```

Example 4.1. Example of how to create a composite dataset

4.10. Spectrum Datasets

Spectra are contained within datasets that also contain raw data counts together with metadata that allows for the correct handling of combinations of spectra (e.g., spectral arithmetic) and display of spectra. Basic spectral types are `SpectralSegment`, `Spectrum1d` and `Spectrum2d`.

4.10.1. Spectrum1d and SpectralSegments

A one-dimensional representation of a spectrum. Container has a `TableDataset()` that has columns for flux, flag, weights and numbered segments (components of the 1d spectrum). It contains

- A flux column (`Double1d`). This can be obtained from a `SpectralSegment` using the `getFlux()` method. For example; `a = %spectrum1d_name%.getFlux()`.
- A wavelength/frequency column (`Double1d`). The wavelength column can be obtained using the `getWave()` method.
- A weight column (`Double1d`). The weight column can be obtained using the `getWeight()` method.
- A segments column (`Double1d`). The segments column can be obtained using the `getSegment()` method.
- A flag column (`Int1d`). The flags can be obtained using the `getFlag()` method.

A `Spectrum1d` can also have metadata (header information) added. The following illustrates how a `Spectrum1d` dataset can be built from scratch.

```

flux = Double1d([12.2,12.5,13.0,11.8,11.9,12.6,14.2,15.8,12.2,15.2])
segs = Int1d([0,0,0,0,0,1,1,1,1]) # segment id for each point
wave = Double1d([1000.0,1000.2,1000.4,1000.6,1000.78,
 \ 1100.0,1100.2,1100.4,1100.6,1100.78])
flag = Int1d(10) + 1
weight = Int1d(10) + 1.0
a = Spectrum1d(flux,weight,flag,segs) #indicate the fluxes and segments.
a.set("wave", wave) # add the wavelengths column
a.setMeta("name","Arp220") # sets keyword name in metadata of Spectrum
# other metadata can be added, as needed.
print a.getWave() # shows the "wave" column
# Using the Dataset viewer, the full information can be viewed (see
Section 4.14)
    
```

The spectrum can be made of several segments. A `SpectralSegment` is the smallest spectrum component dealt with by the DP system. This can be a piece of a spectrum extracted from a larger one-dimensional spectrum to be used for fitting purposes (for example). It can be extracted from a `Spectrum1d` using the following.

```

b=a.getSpectralSegment(1) # get second spectral segment (numbering starts at 0)
print b.getWave() # provides the wavelengths associated with this segment
    
```

Many of the spectral tools (arithmetic, fitters) work with the basic unit of a spectral segment.

4.10.2. Spectrum2d

For multiple spectra taken in an observation, a 2D structure is required. The components of a `Spectrum2d` dataset is similar to that of a `Spectrum1d` dataset, except for having a second dimension. An additional component is the ability to contain subbands. A clear example of the usefulness of this comes in the output from the HIFI spectrometers where several CCD or autocorrelator readouts lead to several "chunks" (subbands) of spectra in one data frame. Having subbands is an option for the `Spectrum2d`. It contains

- A flux column (`Double2d`). This can be obtained from a `SpectralSegment` using the `getFlux()` method. For example; `a = %spectrum1d_name%.getFlux()`.
- A wavelength/frequency column (`Double2d`). The wavelength column can be obtained using the `getWave()` method.
- A weight column (`Double2d`). The weight column can be obtained using the `getWeight()` method.
- A flag column (`Int2d`). The flags can be obtained using the `getFlag()` method.
- (optional) a subbandstart column (`Int1d`). Indicates where in the arrays that a subband starts.
- (optional) a subbandlength column (`Int1d`). Indicates the length of array section that a subband takes up.

The number of channels is automatically generated in the metadata when setting up a `Spectrum2d`. An example of setting up a `Spectrum2d` from scratch is given below.

```

flux2 = Double2d([[12.2,12.5,13.6,12.8],[12.8,12.2,13.3,12.9],
 \ [10.2,14.5,12.5,11.4],[12.2,12.5,13.6,12.8]])
flag2 = Int2d([[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]])
weight2 = Double2d([[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]])
a2 = Spectrum2d(flux2,weight2,flag2) # sets up 4 channels each with 4 pixels
wave2 = Double2d([[1000.0,1000.2,1000.4,1000.6],[1000.0,1000.2,1000.4,1000.6],
 \ [1000.0,1000.2,1000.4,1000.6],[1000.0,1000.2,1000.4,1000.6]])
a2.set("wave", wave2) # add the wavelengths
print a2.getWave() # to print out the wavelengths
print a2.getFlux() # to print out the fluxes.
    
```

We can also set up a `Spectrum2d` with associated subbands. This basically allows us to set up, in one dataset, a container which holds many individual spectra which as many subbands each covering a different wavelength range, if necessary (e.g., with the individual subbands of the HRS spectrometer of HIFI). This forms the basis of how spectral observations, which typically are made up of many frames, are stored in the Herschel DP environment.

```
# Now deal with subbands.
# Create the container for the spectra
a3 = Spectrum2d()
# indicate the number of subbands it will have
a3.setSubbands(2)
a3.setSubbandStart(Int1d([0,2]))
a3.setSubbandLength(Int1d([2,2]))
flux3 = Double2d([[12.2,12.5,13.6,12.8],[12.8,12.2,13.3,12.9]])
flux4 = Double2d([[10.2,14.5,12.5,11.4],[12.2,12.5,13.6,12.8]])
a3.set("flux_1",flux3)
a3.set("flux_2",flux4)
print a3.getFlux(1)
wave3 = Double2d([[1000.0,1000.2,1000.4,1000.6],[1000.0,1000.2,1000.4,1000.6]])
a3.set("wave_1",wave3)
a3.set("wave_2",wave3)
# get wavelengths for second subband
# note that there are two sets of measurements
print a3.getWave(2)
# get fluxes for first set of measurements
# of subband number 1.
print a3.getFlux(1).get(0)
# or second set
print a3.getFlux(1).get(1)
# this way you can go through multiple
# measurements using the same subband that are
# stored in the same dataset.
# We can do the same for wavelengths, e.g.,
print a3.getWave(1).get(0)
# instrument pipelines producing spectra store the data in Spectrum2d
# or a variant (see next section).
```

4.10.3. Expanding Spectrum1d and Spectrum2d Datasets

Extensions to the basic `Spectrum1d` and `Spectrum2d` datasets have been created that allow for more convenient access to specific instrument data types. Typically, the full spectral information, including metadata, is created from the original instrument dataframes and housekeeping information coming from the spacecraft. However, it can be instructive to formulate things from their basic components.

4.10.3.1. HIFI Extensions

Examples of HIFI extensions to the `Spectrum1d` and `Spectrum2d` datasets are the `WbsSpectrumDataset` and `HrsSpectrumDataset` available for the two types of spectrometer data from HIFI. These can be created by obtaining HIFI dataframes and housekeeping telemetry source packets (these are not generally available to most users).

```
# creating a WBS spectrum dataset
from herschel.hifi.pipeline.product import *
w = WbsSpectrumDataset(array of WBS dataframes, array of HK telemetry)
```

Such a spectrum dataset automatically includes more metadata such as observation identification and data creation date. It can also contain the information for the wavelength as a model -- typically polynomial fit information.

Displaying the table of dataset, for each spectrum not only is flux and wavelength listed but other, HIFI-specific, information such as chopper position and on-board buffer storing the data (see Fig.***).

Typical observations actually contain groupings of such datasets. For example, internal flux calibrator dataframes, science dataframes and frequency calibrator data frames. These are typically grouped together in a HIFI timeline product. So a typical HIFI observation with all four spectrometers used would have four HIFI timeline products.

```
# Creating a HIFI timeline product
from herschel.hifi.pipeline.product import *
htp = HifiTimelineProduct(array of WBS dataframes, array of HK telemetry)
```

For the most part users will not need to create the datasets/products but will need to access the data in them. We can use the `getFlux()` and `getWave()` methods as before. For HIFI spectra, the `getWave()` method provides the IF frequency values. The lower or upper sideband frequencies can also be obtained using the `getLsbFrequency()` or `getUsbFrequency()` methods. So we can crudely plot -- with labels to be attached later -- the spectrum (upper or lower sideband) using the following.

```
# Continuing from above.
# Get the first dataset in the product
wbs = htp.get(1)
# Plot of flux against IF frequency
p = PlotXY(wbs.getWave().get(1),wbs.getFlux().get(1))
# This provides a plot of the second frame, called frame number 1.
# Similar but now will plot the LSB frequency which takes
# the local oscillator frequency information into account
p = PlotXY(wbs.getLsbFrequency().get(1),wbs.getFlux().get(1))
```

4.10.3.2. SPIRE extensions to Spectrum1d

The SPIRE instrument also uses an extension of `Spectrum1d`. The basic component dataset for the spectrum obtained by a single SPIRE pixel is the `SpireSpectrum1d`. As opposed to `Spectrum1d`, complex data are possible (stores `Numeric1d` inputs as `Complex1d`). The data is composed of complex values of flux and flux error with associated units. A mask can also be added (type `Int1d`).

Individual spectra from separate pixels can be grouped together to formulate a single SPIRE scan dataset. This in turn can be grouped into a set of scans that would be more typical of a single SPIRE observation.

```
from herschel.share.unit import *
from herschel.spire.ia.dataset import *
c = Complex1d([2+3j, 3+2.1j,3.6 +2.4j,0.9+2.1j])
err = Complex1d([0.2+0.2j, 0.8+0.3j,0.4+0.3j,0.15+0.1j])
flu = FluxDensity.JANSKYS
wu = WaveNumber.RECIPROCAL_METER
wn = Double1d([0.3,0.4,0.5,0.6])
mask = Int1d([1,1,1,1])
sps = SpireSpectrum1d("Pixel name")
sps.setComplexFlux(c,flu)
sps.setComplexFluxError(err,flu)
sps.setWavenumber(wn,wu)
sps.setMask(mask)
# Now we can get the data by replacing set by get,
# and removing the arguments, e.g.,
sps.getComplexFlux() # returns the flux data
# and we can get the units separately, e.g.,
sps.getComplexFluxUnits()
## Now we can place a number of pixels in a single unit
## a SpireSpectrumCompositeDataset.
## Create sps, sps1, sps2, sps3 etc.
spire_cds = SpireSpectrumCompositeDataset("Scan number")
## Scan number can be a string name (as above) or a long numeric value.
## add pixels of data.....
spire_cds.setPixel(sps)
```

```

spire_cds.setPixel(sps1)
spire_cds.setPixel(sps2)
spire_cds.setPixel(sps3)
## pixel names are as set up in the original SpireSpectrum1d
## we can get a pixel using
wanted_sps = spire_cds.getPixel("Pixel name")
### Most SPIRE spectrometer observations are composed of many scans
### which we can then place several composite datasets in a single dataset.
spire_sds =SpectrometerDetectorSpectrum() # create empty dataset
spire_sds.setScan(spire_cds) # add in scan, given next scan number available = 0.
spire_sds.setScan(spire_cds1) # add in scan, given next scan number available = 1.
### Now access a scan.
wanted_cds = spire_sds.getScan(0) # for the first scan

```

4.10.3.3. PACS Spectrum1d and Spectrum2d extensions

PACS spectral is based on handling the Frames and Ramps based on the readout of the PACS spectrometer. The handling of these data is currently discussed in the PCSS User's Manual.

4.11. Image and Cube Datasets

Image and cube datasets are composed of Double2d and Double3d components that represent intensity, masks and errors. They also contain metadata information that provide for coordinate information.

SimpleImage contains a standard two-dimensional image which contains the following.

- Image made of a Numeric2d (e.g., Double2d or Int2d) component.
- Error made of a Numeric2d (e.g., Double2d or Int2d) component can be added.
- Exposure made of a Numeric2d (e.g., Double2d or Int2d) component can be added.
- Flag made of a Short2d (e.g., Double2d or Int2d) component can be added. is created.

Units can be added/set to the image contained and World Coordinate System information.

An example of creating a SimpleImage from an imported JPG image is given below.

```

from herschel.ia.gui.image import *
from herschel.ia.dataset.image.wcs import Wcs
from herschel.share.unit import *
# choose units
myQuant = FluxDensity.MILLIJANSKYS
# create WCS to assign
myWcs = Wcs(crpix1 = 29, crpix2 = 29, crval1 = 30.0, crval2 = -22.5)
# create the simple image with an assigned WCS and a description
myImage2 = SimpleImage(description="Veil nebula",unit = myQuant, wcs = myWcs)
# import an image -- converted into Double2d/Int2d for inclusion
# Note: to import the image with the following command, the JPG file
# needs to be in the same directory as the the HCSS interface (JIDE or HIPE)
# was started from.
myImage2.importFile("ngc6992.jpg")
# Assign a reference wavelength to the image
myImage2.setWavelength(12.0,Length.MICROMETERS)
# print reference wavelength in millimetres.
print myImage2.getWavelength(Length.MILLIMETERS)
# print the units being used
print myImage2.getUnit()
#print intensity at pixel position 30, 35
print myImage2.getIntensity(30, 35)
# We can add exposure and error maps.....
# Use myImage2.setExposure(<a Double/Int2d image>) or
# myImage2.setError(<a Double/Int2d image>) to include
# exposure maps or error maps with the image.
# Using the .getError and .getExposure methods extracts these images from
# the SimpleImage dataset.
# To display we can use

```



```
Display(myImage2)
# Some display edit functions are available using right button mouse click on
# the image.
```

In a similar vein to the above, we can also create a `SimpleCube` which allows us to store three-dimensional images (or multiple stacked 2D images). The `SimpleCube` currently can also include error, flag and/or exposure maps, which must also be 3D arrays. A single WCS only can be applied to the `SimpleCube`. For example, it is not possible to provide different WCS's for each image in an image stack.

To create a `SimpleCube` we need to import a `Double/Int3d` object. For simplicity, we can create this from `myImage2`.

```
l1 = myImage2.getImage()
l2 = myImage2.getImage()
d3 = Double3d()
d3.append(l1,0) # which appends the image along the 0 axis (stacking)
d3.append(l2,0) # append the same image.
# Now we create the SimpleCube.
myImage3 = SimpleCube(description="Veil nebula in 3D",
    \unit=MyQuant, image=d3, wcs = myWcs)
# We can obtain the units.
# print the units being used
print myImage3.getUnit()
#print intensity at pixel position 30, 35 in layer (depth) 0 -- the first layer
print myImage3.getIntensity(0,30, 35)
# We can create an array of SimpleImages from the cube.
sa = myImage3.decomposeToSimpleImages()
```

4.12. Assigning a World Coordinate System (WCS) to SimpleImage and SimpleCube

We are able to assign WCS information to images and cubes. The World Coordinates System (`wcs`) describes the coordinates of a `SimpleImage` or `SimpleCube`. It makes it possible to convert `imageCoordinates` to `worldCoordinates` and the other way around. The WCS can have a lot of parameters, as defined in the FITS standard :

- `naxis` : the number of axes
- `crval1` : First coordinate of the centre
- `crval2` : Second coordinate of the centre
- `crpix1` : Reference pixel X coordinate
- `crpix2` : Reference pixel Y coordinate
- `cdelt1` : Pixel scale axis 1. Step per pixel or number of degrees per pixel along x-axis when converting to Sky Coordinates. These parameters are no longer used in modern `Wcs` definition, but are included in the `CDi_j` matrix.
- `cdelt2` : Pixel scale axis 2. Step per pixel or number of degrees per pixel along y-axis when converting to Sky Coordinates. These parameters are no longer used in modern `Wcs` definition, but are included in the `CDi_j` matrix
- `ctype1`, `ctype2` : Projection type name. This can be "LINEAR", "PIXEL" or the FITSconvention. The default value for `ctype1` and `ctype2` is "LINEAR". When using the FITSconvention, first 4 characters are:
 - o RA-- and DEC- for equatorial coordinates
 - o GLON and GLAT for galactic coordinates

o ELON and ELAT for ecliptic coordinates

The next 4 characters describe the projection. Possibilities are:

o -AZP: Zenithal (Azimuthal) Perspective

o -SZP: Slant Zenithal Perspective

o -TAN: Gnomonic = Tangent Plane

o -SIN: Orthographic/synthesis

o -STG: Stereographic

o -ARC: Zenithal/azimuthal equidistant

o -ZPN: Zenithal/azimuthal PolyNomial

o -ZEA: Zenithal/azimuthal Equal Area

o -AIR: Airy

o -CYP: CYlindrical Perspective

o -CAR: Cartesian

o -MER: Mercator

o -CEA: Cylindrical Equal Area

o -COP: COnic Perspective

o -COD: COnic equiDistant

o -COE: COnic Equal area

o -COO: COnic Orthomorphic

o -BON: Bonne

o -PCO: Polyconic

o -SFL: Sanson-Flamsteed

o -PAR: Parabolic

o -AIT: Hammer-Aitoff equal area all-sky

o -MOL: Mollweide

o -CSC: COBE quadrilateralized Spherical Cube

o -QSC: Quadrilateralized Spherical Cube

o -TSC: Tangential Spherical Cube

o -NCP: North celestial pole (special case of SIN)

o -GLS: GLobal Sinusoidal (Similar to SFL)

-
- Other types are also possible (for example 63TEMP for temperature.)

- o cunit1 : The Unit of Axis 1.
- o cunit2 : The Unit of Axis 2.
- o epoch : Epoch of coordinates
- o Radesys : The reference frame, default value is "ICRS"
- o pc1_1 : Element (1,1) of the linear transformation matrix. The pc1 and pc2 parameters are no longer used in modern Wcs definition, but are together with CDELTA1 and CDELTA2 included in the CDi_j matrix
- o pc1_2 : Element (1,2) of the linear transformation matrix.
- o pc2_1 : Element (2,1) of the linear transformation matrix.
- o pc2_2 : Element (2,2) of the linear transformation matrix.
- o cd1_1 : Element (1,1) of the corrected linear transformation matrix
- o cd1_2 : Element (1,2) of the corrected linear transformation matrix
- o cd2_1 : Element (2,1) of the corrected linear transformation matrix
- o cd2_2 : Element (2,2) of the corrected linear transformation matrix

For the situation where there is a third dimension the following also apply.

- ctype3 : Description of what the 3rd axis represents, e.g. Wavelength, Time, M1 Temperature, ...
- cunit3 : The Unit of Axis 3.
- crval3 : [Optional - in case of equidistant 3rd dimension]. Wavelength, time, ... of reference layer; unit : length, time, ...
- crpix3 : [Optional - in case of equidistant 3rd dimension] Reference layer index
- cdelt3 : [Optional - in case of equidistant 3rd dimension] Scale in 3rd dimension - unit : length, time, ...
- PC elements
 - o pc1_3 : Element (1,3) of the linear transformation matrix
 - o pc2_3 : Element (2,3) of the linear transformation matrix
 - o pc3_1 : Element (3,1) of the linear transformation matrix
 - o pc3_2 : Element (3,2) of the linear transformation matrix
 - o pc3_3 : Element (3,3) of the linear transformation matrix

To create a WCS object that can be assigned to an image we can use something like the following.

```
from herschel.ia.dataset.image.wcs import Wcs
# create WCS object, units in degrees by default
myWcs = Wcs(crpix1 = 29, crpix2 = 29, crval1 = 30.0, crval2 = -22.5,
            \ cdelt1=0.0004, cdelt2 = 0.0004, cunit1="DEGREES",
            \ cunit2="DEGREES", ctype1 = "RA---TAN", ctype2= "DEC--TAN")
# we can assign the world coordinates to the an image
myImage2 = SimpleImage(description="Veil nebula", wcs = myWcs)
# and can obtain the world coordinates at any pixel on the image.
```

```
print myImage2.getWcs().getWorldCoordinates(31,31)
# This provides an array of sky coordinates in degrees.
# We can get the intensity at a given WCS position.
# First put an image in...
myImage2.importFile("ngc6992.jpg")
# Get the intensity at a given WCS position.
print myImage2.getIntensityWorldCoordinates(30.0012,-22.498)
```

For the `SimpleCube` we can do this almost identically.

```
from herschel.ia.dataset.image.wcs import Wcs
# create WCS object, units in degrees by default
myWcs = Wcs(crpix1 = 29, crpix2 = 29, crval1 = 30.0, crval2 = -22.5,
  \ cdelt1=0.0004, cdelt2 = 0.0004, cunit1="DEGREES",
  \ cunit2="DEGREES", ctype1 = "RA---TAN", ctype2= "DEC--TAN")
# We need to have a Double/Int3d image to put in our cube (call it "d3").
# we can assign the world coordinates to the an image
myImage3 = SimpleCube(description="Veil nebula", image=d3, wcs = myWcs)
# and can obtain the world coordinates at any pixel on the image.
print myImage3.getWcs().getWorldCoordinates(31,31)
# Get the intensity at a given WCS position. We need three
# arguments now, with the first argument being the layer number (depth)
# from which we want the intensity measure. Count starts from 0.
print myImage3.getIntensityWorldCoordinates(0,30.0012,-22.498)
```

4.13. Wrapping it all up: Products

Let us briefly run through what we have covered so far. We started with simple arrays in Section 4.3, went on with multidimensional arrays in Section 4.6 and introduced array datasets in Section 4.7. Then it was time for table datasets in Section 4.8 and composite datasets in Section 4.9. As you can see, every object we have examined acted as a container for the previous ones. Now we complete the journey by introducing the highest level of them all, the *Product*.

A *Product* is an object containing a set of metadata entries (some of which are mandatory) and one or more datasets. The mandatory metadata values are `description`, `creator`, `creationDate`, `instrument`, `startDate`, `endDate`, `modelName` and `type`. They will be automatically added whenever you create a new product. Let us check:

```
myProduct = Product()      # Creating a new, empty Product
print myProduct.meta      # Printing its metadata
print myProduct.getMeta() # Same thing, "Java style"
```

4.13.1. Mandatory Parameters in Products

As you can see some entries are already set to meaningful values, others are set to `Unknown`. You can now modify the mandatory metadata and add as many new entries as you wish. There are so-called "setter" methods for setting values of the mandatory metadata, which currently includes a description, the creator, an instrument, model name of the instrument in use and type, as shown below:

```
myProduct.setDescription("My SPIRE product")
myProduct.setCreator("Myself")
myProduct.setInstrument("SPIRE")
myProduct.setModelName("PFM")
myProduct.setType("UM")
```

Alternately, these can be set using

```
myProduct.creator = "Myself"
myProduct.instrument = "SPIRE"
etc...
```

Finally, we can include many of these settings on a single line

```
myProduct=Product(creator="Myself", instrument="SPIRE", \
description="My SPIRE product", modelName="PFM", type="UM")
```

4.13.2. Setting Date Information

The creation, start and end dates for a Product need to be expressed in terms of a `FineTime`. If all of these are the current date then we can convert a Java date to a `FineTime` and include it as metadata in our product. For example:

```
from herschel.share.util.fltdyn.time import FineTime

myProduct.setCreationDate(FineTime(java.util.Date()))
myProduct.setStartDate(FineTime(java.util.Date()))
myProduct.setEndDate(FineTime(java.util.Date()))
```

Because the `startDate`, the `endDate` and the `creationDate` are mandatory metadata parameters, they are set to the current date and time at the moment when the product is created. If those dates are not the current date then it is possible to set it up using UTC or TAI representation of a calendar day (see e.g. Section 11.2), like it is shown in the following example:

```
from herschel.share.fltdyn.time import *

formatter = SimpleTimeFormat(TimeScale.UTC)
timeUtc = formatter.parse("2008-01-31T12:35:00.0Z") # Z at the end is mandatory
for UTC

formatter = SimpleTimeFormat(TimeScale.TAI) # or just SimpleTimeFormat()
timeTai = formatter.parse("2008-01-31T12:35:00.0TAI") # TAI at the end is mandatory
for TAI

myProduct.setCreationDate(timeUtc) # or
myProduct.setCreationDate(timeTai)
```

Note that the two previous dates, represented as `FineTime`, are different:

```
print timeUtc # 2008-01-31T12:35:33.000000 TAI (1580474133000000)
print timeTai # 2008-01-31T12:35:00.000000 TAI (1580474100000000)
```

4.13.3. Additional Metadata

Now, to add, modify and read additional metadata:

```
myProduct.getMeta().set("Here goes a name", StringParameter("Here goes a value"))
print myProduct.meta["Here goes a name"]
# {description="", string="Here goes a value"}
```

In the example above we set a name and a value for the metadata. In this case the value was represented by a `String` object, but as you already now you can also assign other types of values with `LongParameter`, `DoubleParameter`, `BooleanParameter` and `DateParameter`.

4.13.4. Inserting and Getting Datasets from a Product

But how do you insert and get the contents of the datasets in a product? You can use the `getDefault()` method to get the first dataset stored in the product, or the `get()` method to get any stored dataset, whose name you have to provide as argument. The name is a string assigned when the dataset is first inserted into the product. Here is an example:

```
myTable = TableDataset()
myTable.setDescription("This is a Table Dataset")
myComposite = CompositeDataset()
myComposite.setDescription("This is a Composite Dataset")
myProduct.set("oneDataset", myTable) # We have to give a name to every
```

```

                                # dataset we insert
myProduct["anotherDataset"] = myComposite # Jython style to add a dataset
myProduct.set("anotherDataset", myComposite) # Java style
print myProduct.getDefault() # As you will see from the description,
                                # this is the Table Dataset
print myProduct["anotherDataset"] # Getting the Composite Dataset,
                                # Jython style...
print myProduct.get("anotherDataset") # ...and Java style

```

Instead of just printing out the datasets you get, you can assign them to variables and execute other operations on them. To see how to explore the contents of datasets please refer to the previous sections of this chapter.

If you are not a fan of the command line you can use the handy Dataset Inspector tool to view and manipulate datasets and products. This tool is described below, in Section 4.14.

Products are also treated in Appendix A, Section A.3.

4.14. The Dataset Inspector

As we have seen above, inspecting Datasets and Products using the command line can quickly become cumbersome, especially when dealing with several large instances. Luckily there is a quick and efficient way to carry out these tasks via a graphical tool, the *Dataset Inspector*, already briefly introduced in Chapter 2 (see Section 2.3.5 and Figure 2.7).

Using it is very simple. Once invoked via its icon on the toolbar or the `DatasetInspector` command, it will display its main window, divided in two panes. The left pane shows a tree-like folder structure whose root is called `Datasets` and `Products`, with two main branches called `Datasets` and `Products`. The former will contain any datasets not included in products, while the latter will list the products themselves. Whenever the icon of a folder appears, clicking on it will display its contents. A similar tree-like structure will appear in the right panel, which is also used to display the objects' contents, like metadata and table data.

Figure 4.1 shows Dataset Inspector displaying the metadata of a product. The table is divided in three columns showing the name, value and unit (if any) of each keyword. When the value of a keyword is undefined this is signalled with a red `undefined` label.

Figure 4.1. The Dataset Inspector showing product metadata.

Additional features are available for parameters such as `obsid` and `bbid` (the identification numbers of observations and of their building blocks). By right-clicking on the value of these parameters we can switch between decimal and hexadecimal representations.

Dates and times are shown by default in UTC (Coordinated Universal Time), with their `Finetime` representation in brackets (for more information on time in DP see Chapter 11). By right-clicking on the parameter values we can switch between UTC and TAI (International Atomic Time).

The Dataset Inspector can do much more than displaying products and datasets. It also contains a number of plugin viewers that allow more advanced data manipulation. Two of them are described below.

4.14.1. The TablePlotter

4.14.1.1. Introduction

The *TablePlotter* utility is a GUI tool to graphically view and analyze Table Datasets which are organized in columns with an equal number of rows, for instance, time ordered detector signals.

4.14.1.2. Invoke TablePlotter

- **Invoke TablePlotter as a Viewer in DatasetInspector**

TablePlotter works as one of the viewers available within the Dataset Inspector in JIDE. It can be invoked as follows:

1. Place the mouse over a dataset
2. Press the right mouse button
3. Select TablePlotter from the "View With" floating menu.

- *Invoke TablePlotter from a Command Line or Script*

TablePlotter can also be invoked from the command line. First we need to import TablePlotter with:

```
from herschel.ia.gui.explorer.table import *
```

Assuming `tbs` is a Table Dataset, then the TablePlotter 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(tableDataset).component, 1)
```

Caveat: The dataset extraction function of TablePlotter will not work if invoked from the command line. An appropriate modification to the API will be implemented in a future version.

- *Limitation on Datasets*

Among the three generic datasets, TablePlotter supports only the TableDataset.

4.14.1.3. Layout of the TablePlotter

When TablePlotter is invoked, a GUI appears, displaying an x/y-plot of the first two columns of the selected Table Dataset (See Figure 4.2). The TablePlotter GUI contains three major components, a 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.

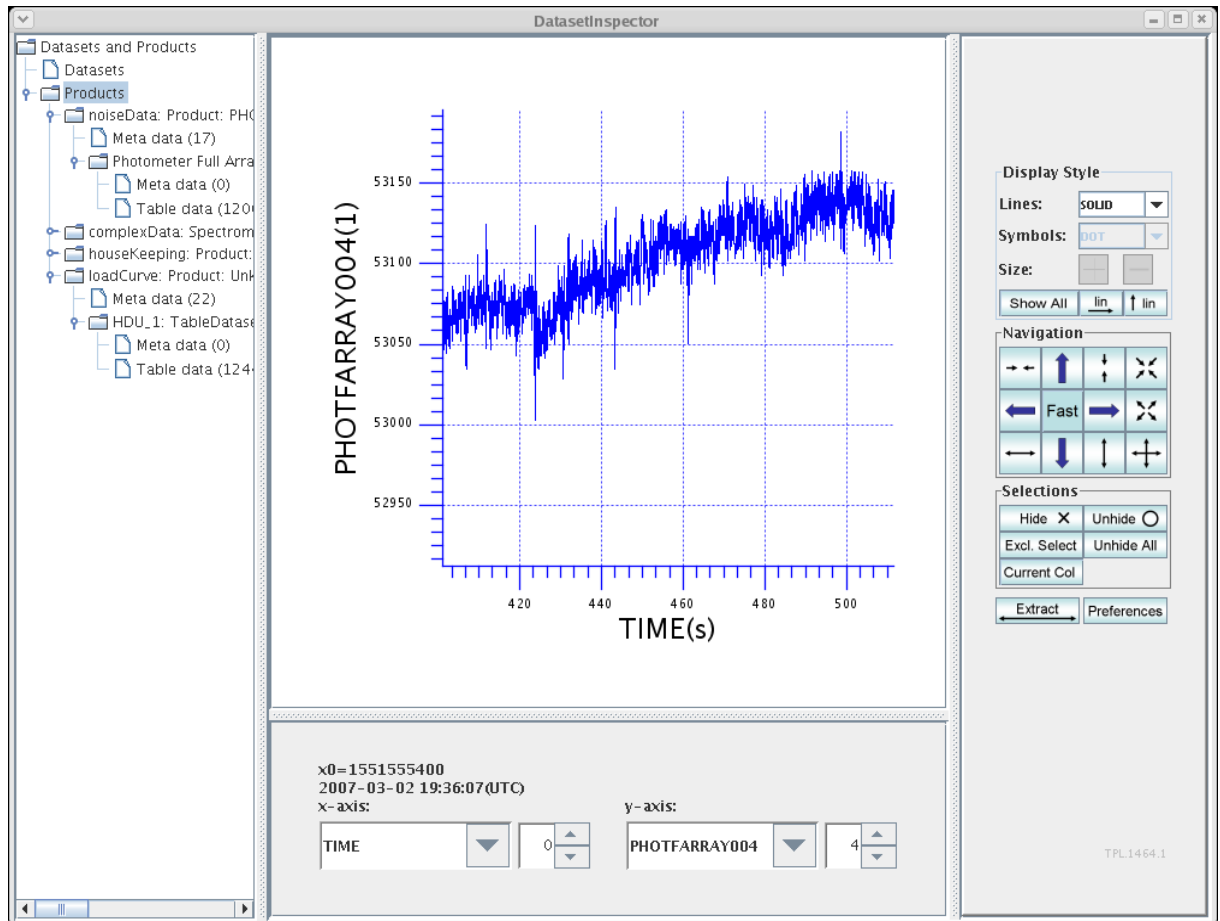


Figure 4.2. Layout of the TablePlotter GUI.

4.14.1.4. Controls and their functions

The *TablePlotter* provides the following control buttons to view and analyze data.

- **X and Y- Axis Selection:**





Under the graphics display area, two sets of Combo Box buttons and spinner buttons allow users to select X and Y-axis data. The first column of the TableDataset is associated with X-axis by default. The second column is initially associated with the Y-axis.



Users can choose a column by name in the Combo Box and by number in 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 to select.

- **Display Style:**

The control buttons in this section allow to change the axis style (linear or log), line style (solid or dashed and more), and symbol style.

The default axis scaling is linear. The toggle buttons  /  and  /  allow to switch between linear and logarithmic scales in the X / Y axes respectively.

The pull-down menus of Lines and Symbols allow to select line style and symbol style. The selection of symbol styles is only available when the line styles are either *MARKED*, *MARK_DASHED* or *NONE*. To increase or decrease the symbol size, click either  or .




Another toggle button  /  determines whether all data points or only the selected ones are shown (see detail in Selections below).

• **Navigation:**


Two buttons are provided to simultaneously zoom in  and zoom out  in both X and



Y directions. The buttons  and  zoom out individually in either X or Y direction .

Four buttons with arrows     pan the view onto the graph in X or Y direction.

The size of each zooming or panning step is controlled by a toggle button  /  and the exact factors of both fast and slow modes can be adjusted in the  menu (for details see the *Preferences* section below).



To Zoom-in on specific areas of the graph, press the left mouse button, and hold-and-drag a rectangle around the area of interest with the mouse pointer.




There are three *Free Scaling* buttons. The  button will adjust the scales of both axes such

that all visible datapoints are optimally distributed within the display, while  and  will do the same but for either X or Y axis alone.

• **Selections:**

The selection feature of TablePlotter allows to hide or select a particular portion of the data points.

In combination with  /  in the Display Style section, and Multi Column Mode, this feature can be used to display only selected data to get fast automatic scaling when scanning through many columns of data. The main purpose, however, is the extraction of specific data points into new datasets. A typical purpose could be for instance to remove electronic glitches from detector data, or to extract a specific piece of signal from a sequence of instrument configurations.

The following buttons   , hide, un-hide or exclusively select all data points within a rectangular area in the plot. This area is selected after pushing one of those three buttons by holding and dragging the mouse pointer in the same way as for zooming in.

Clicking the button  will re-select all hidden data points.

If the **Current Col** toggle button is visible, the TablePlotter is in single column mode. In this mode hiding or selecting operations will only apply to the current column. Clicking on this button

will toggle into all columns mode and the button will change to **All Cols**. Now all the columns are affected and selections are done based on the selected intervals on the X-axis only. The Y-coordinate will be ignored in this mode.

In **Show All** mode the hidden data points will be marked with red symbols. See Figure 4.3 below. Clicking on **Show All** toggles to **Sel Only** mode, where all hidden data points disappear from the graph.

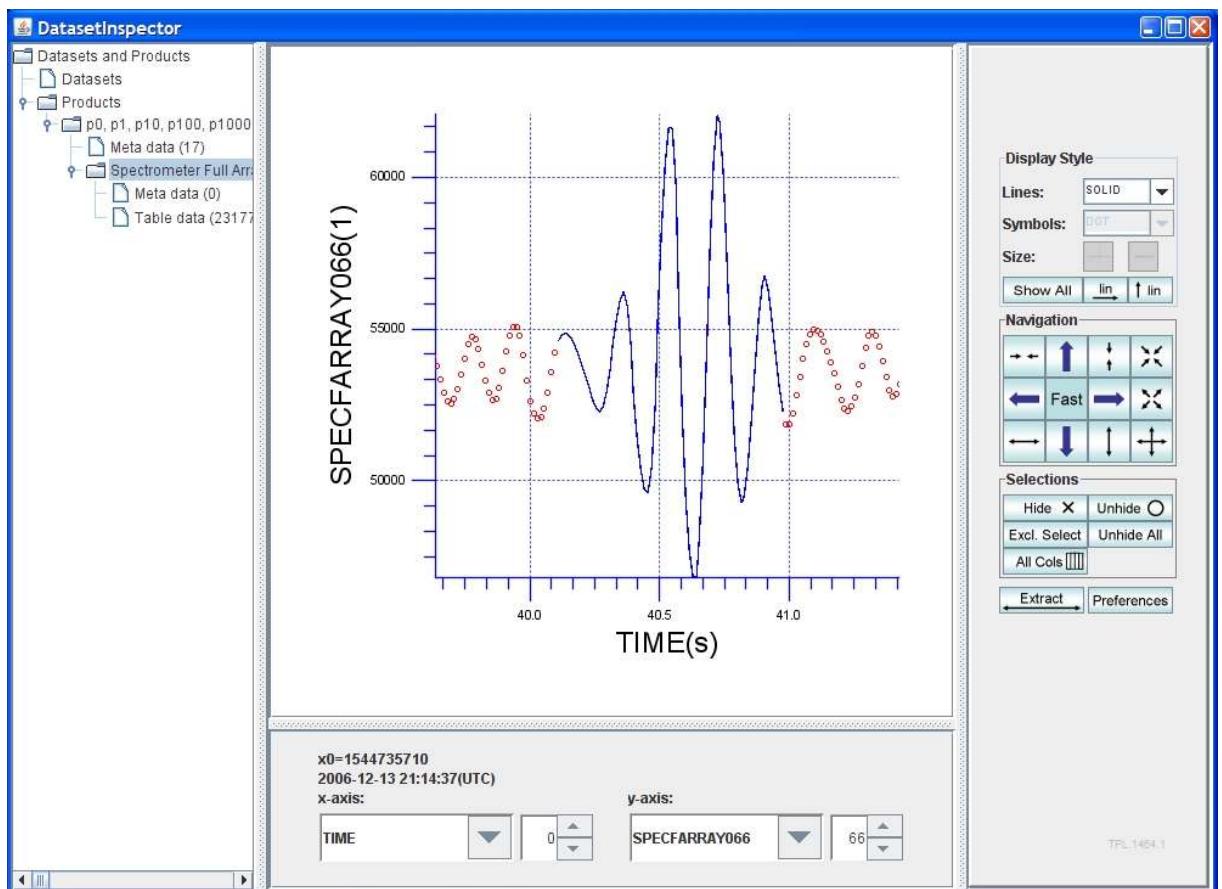




Figure 4.3. The plot with selected and hidden data points.

- **Dataset Extraction:**

To extract a subset of the data after performing the necessary selection operations, press the **Extract** button. The selected data will be extracted into a new dataset that will be fed back to DataInspector, where it will appear in the leftmost panel under "Datasets".

If **Current Col** is selected, only the selected data points in the currently displayed column will be extracted.

If  is selected, the selected data points in all the columns become available for extraction. After clicking , a column selection window will pop up to allow users to **Add** individual columns or **Add All** columns to a list. Users can also **Remove** individual columns or **Remove All**. **Up** and **Down** buttons allow to change the order of columns in the new dataset (see Figure 4.4).

Hitting the **Close** button will complete the extraction and an option is provided to change the default name of the new dataset.

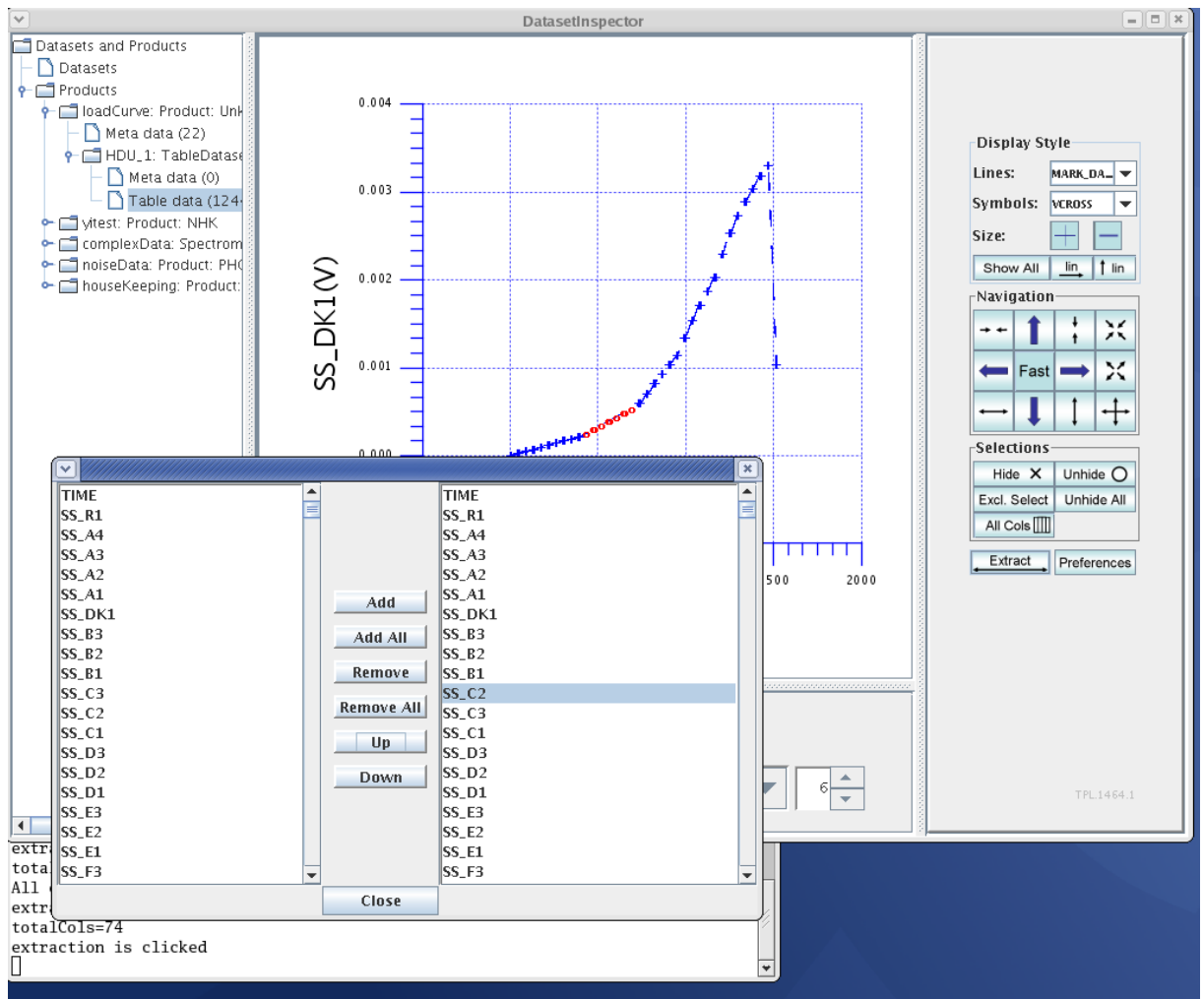


Figure 4.4. Extract Selected Data from Multi Columns to a New DataSet.

- *Preferences:*

Finally the Table Plotter provides a Preferences menu with two options. The first one is Set properties... where preferred zooming and panning factors for Fast and Slow modes can be set.

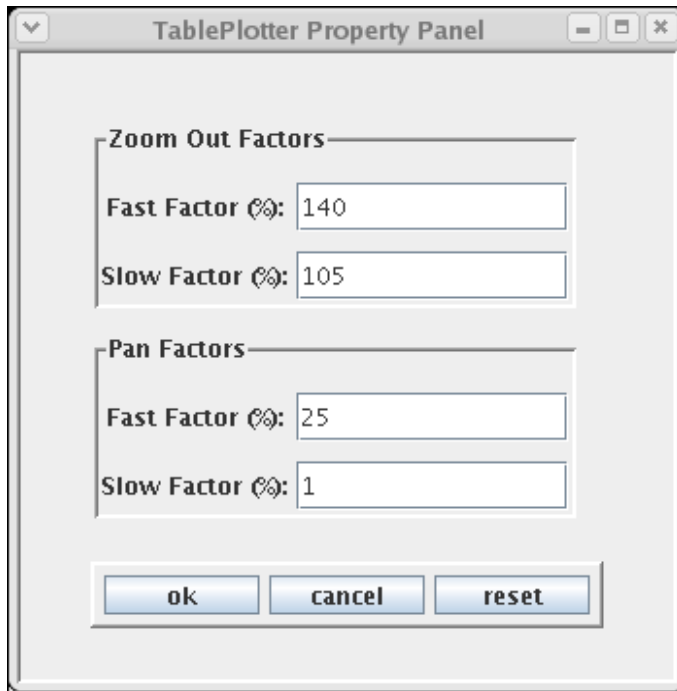


Figure 4.5. Preferences: Set Properties

The second one controls the display of Complex Data. TablePlotter allows only one graph to be displayed at a time. Here the user has three choices: plot modulus only, plot real part only, or plot imaginary part only.

The selected preferences are stored in a properties file and will be "remembered" in the next call to Table Plotter.

4.14.2. The Power Spectrum Viewer

4.14.2.1. Introduction

The Power Spectrum Viewer, which can be accessed under the right-click menu item Power Spectrum, will generate a power spectrum for each column of the dataset, provided there is a time column, and then use the TablePlotter to display the result graphically.

4.14.2.2. Power Spectrum Generator

In the same floating menu, there is another menu item named Power Spectrum. When Power Spectrum is invoked, two input boxes and one push button named Start FFT will be displayed (see Figure 4.6, below).

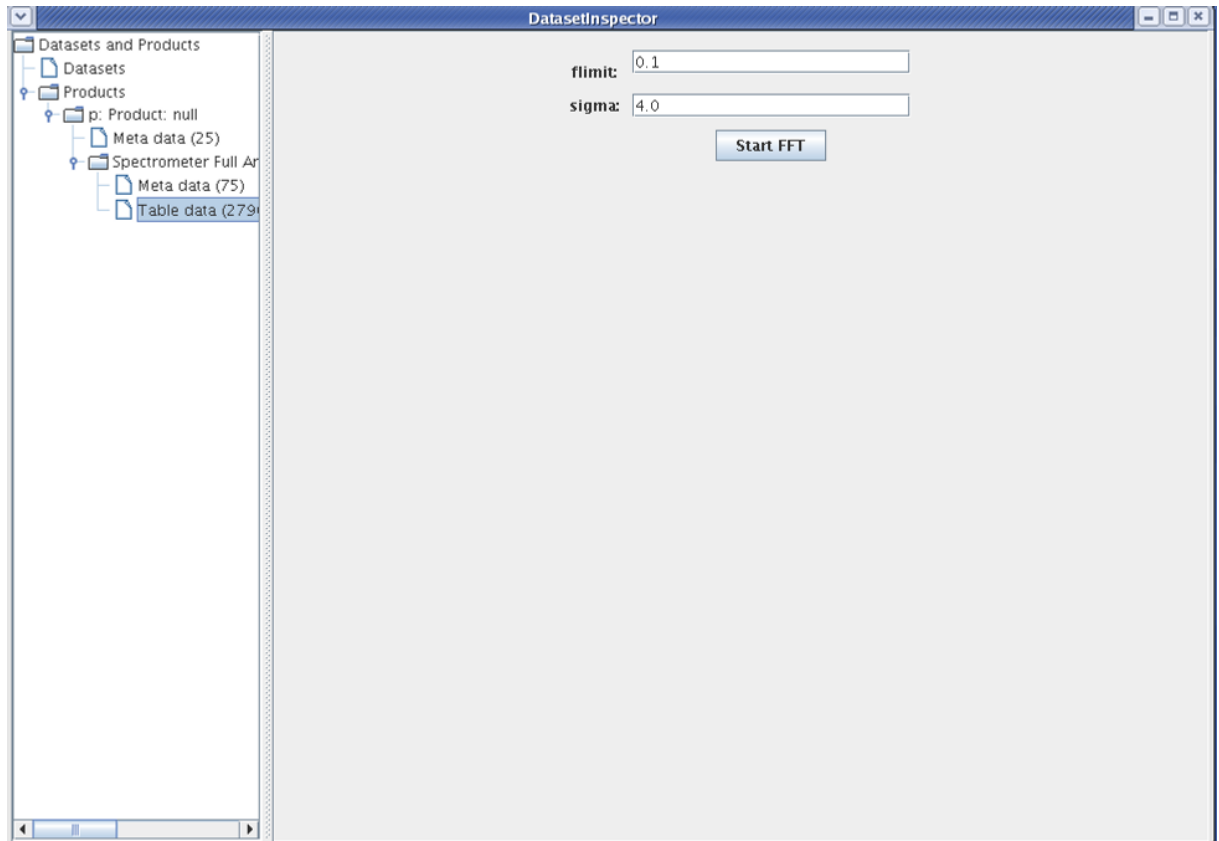


Figure 4.6. Power Spectrum Generator Viewer.

Two text boxes are pre-filled with the cut off frequency (flimit) and the deglitcher threshold (sigma). The inverse cut off frequency determines the length of the intervals, that the data timeline will be subdivided into before performing the FFT. Each of these datasets is Fourier transformed individually and the resulting power spectra are quadratically co-added to yield a power spectrum with a better S/N, i.e. a higher cut off frequency will yield a better S/N for the resulting power spectrum.

The sigma value controls a simple sigma/kappa deglitcher, that eliminates all datapoints that are more than sigma times the standard deviation away from the mean. After eliminating those data points the procedure is repeated until no more data can be discarded. Both flimit and sigma can be changed in the menu.

After clicking the Start FFT button, and a short processing time, the power spectrum will be displayed in the TablePlotter as shown in Figure 4.7. The newly created dataset will be fed back into Dataset Inspector and appears under "Datasets" in the left most section. If the dataset does not contain *Time* as a column, an error message will be displayed.

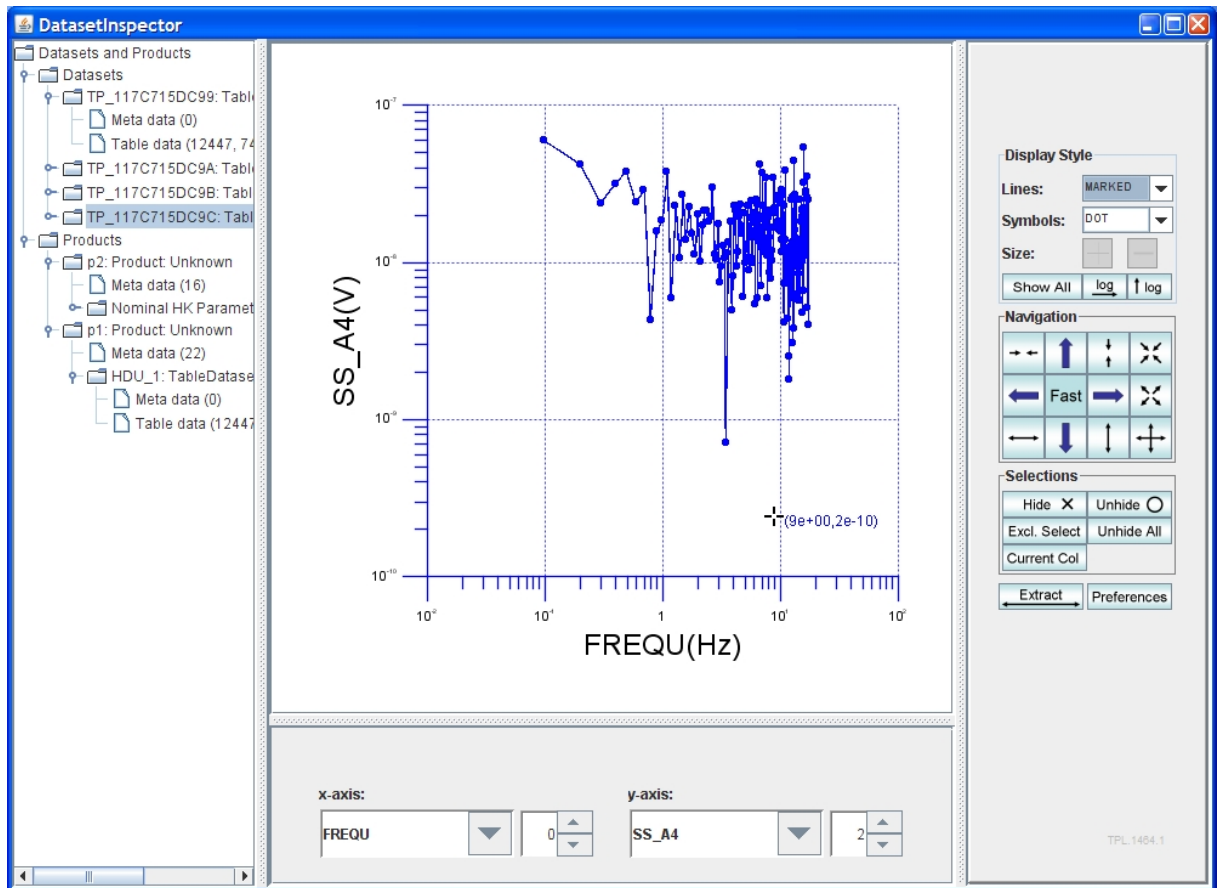


Figure 4.7. Power Spectrum Plot.

Chapter 5. DP Numeric: Basic Functions for Herschel DP

5.1. Introduction

This chapter describes how to use the DP *numeric* library from the interactive Jython environment (JIDE). For further details of the functions provided, or use of the library from Java programs, please see the API documentation for `herschel.ia.numeric`.

The purpose of the numeric library is to provide an easy-to-use set of numerical array classes (programs) and common numerical functions. The library also supports arrays of booleans and strings.

5.2. Getting Started

The DP numeric packages are loaded and available to the user on starting an DP/JIDE session. Basic setup and arithmetic manipulation of array datasets of various types are discussed in Chapter 4.

5.3. Basic Numeric Array Arithmetic

DP numeric arrays support arithmetic operations that are applied element-by-element. For example:

```
y = DoubleId.range(5)           # [0.0,1.0,2.0,3.0,4.0]
print y * y * 2 + 1             # [1.0,3.0,9.0,19.0,33.0]
```

This is much simpler (and runs much faster) than writing an explicit loop in Jython. **It is important to appreciate that the '+' operator does not concatenate arrays, as it does with Jython arrays.** For example:

```
# Adding Jython arrays
print [0,1,2,3] + [4,5,6,7]      # [0, 1, 2, 3, 4, 5, 6, 7]

# Adding DP numeric arrays
print DoubleId([0,1,2,3]) + DoubleId([4,5,6,7])    # [4.0,6.0,8.0,10.0]

# Concatenate two DP numeric arrays
print DoubleId([0,1,2,3]).append(DoubleId([4,5,6,7]))
# [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0]

# Adding Jython arrays to DP numeric arrays
print [0,1,2,3] + DoubleId([4,5,6,7])             # [4.0,6.0,8.0,10.0]
print DoubleId([0,1,2,3]) + [4,5,6,7]             # [4.0,6.0,8.0,10.0]
```

All arrays currently support the following arithmetic operators:

```
+, -, *, /, %, **
```

Note that the 'modulo' operator '%' provides the normal Jython semantics for this operation, which is not the same as that of the Java '%' operator. The Jython definition is more consistent with the mathematical notion of congruence for negative values.

The following relational operators are also provided, which return a BoolId array:

```
<, >, <=, >=, ==, !=
```

For example:

```
y = DoubleId([0,1,2,3,4])
print y > 2                    # [false,false,false,true,true]
```

5.4. Numeric Functions and Lambda Expressions

In DP, functions can be applied very simply as follows:

```
print Sqrt(16) # 4.0 (applied to a scalar)
y = Double1d([1,4,9,16])
print Sqrt(y) # [1.0,2.0,3.0,4.0] (applied to a DP numeric array)
```

As shown by this example, functions on scalars (such as `Sqrt`) are implicitly mapped over each element of an array. Functions may be combined with arithmetic operators to perform complex operations on each element of an array:

```
t = Double1d([1,2,3,4])
print SIN(1000 * t * (1 + .0003 * COS(3 * t)))
# [0.6260976237441638,0.5797470124743422,0.8629107307631398,
#-0.9811675382238753]
```

The **names of functions in the numeric library have ALL LETTERS capitalised**. This is to avoid ambiguity, as Jython already defines certain functions, such as `'abs'`, which are not applicable to our DP numeric arrays.

There are various types of functions in the numeric library:

```
y = Double1d([1,2,3,4])

print Sqrt(4) # double->double
print Sqrt(y) # double->double (mapped)
print REVERSE(y) # Double1d->Double1d
print MEAN(y) # Double1d->double
```

It is possible to define **new functions** as *lambda expressions* in Jython and apply them to DP numeric arrays. For example:

```
y = Double1d([1,2,3,4])

f = lambda x: x*x + 1 #take the given array, call it 'x' and
#return the value x^2 +1 to an array called f.

print f(y) #[2.0,5.0,10.0,17.0]. Each element of y was
#taken -> x then each element was squared
#plus 1 added.
```

However, in this case, it's much easier and faster to do this with array operations.

```
print y * y + 1
```

Lambda expressions are not as fast as the standard Java functions provided by the numeric library, but this is often not a problem. Where performance is an issue, new functions can be defined in Java (see the JavaDoc of the `herschel.ia.numeric` library).

More complex functions (equivalent to subroutines) can be created using the `def` command, which is discussed in Section 3.13.

5.5. Selection, Data Filtering and Masking Methods

The numeric library provides operations, such as `'filter'`, which allows the selection of array elements based on a given criterion (e.g., element with values between 3 and 6). There is no `'map'` operation because mapping is implicit with the array style of processing.

The 'filter' method returns a DoubleId array. The selection criterion for the filter method MUST be declared using a lambda function:

```
u = DoubleId.range(10)
print u.filter(lambda x: x>3 and x<6)
```

Note: The Jython filter operation can be used but returns a Jython array:

```
print filter(lambda x: x>3 and x<6, u)
# __class__ returns org.python.core.PyList
print filter(lambda x: x%2==1, u)
```

Jython list comprehensions can be used but also return Jython arrays:

```
print [x for x in u if x>3]
print [x*x for x in u if x>3 and x<6]
print DoubleId([x*x for x in u if x>3 and x<6])
#this last now provides us with a numerical array as we have also
#translated into a DoubleId array.
```

The SQUARE function could equally have been applied:

```
print u.filter(lambda x: x>3)
print SQUARE(u.filter(lambda x: x>3 and x<6))
```



Warning

If a lambda expression is applied to an array, remember that it is applied to the entire array and not mapped over the elements. This can lead to unexpected behaviour as in the following example:

```
u = DoubleId.range(10)
print (lambda x: x>2 and x<4)(u)
# [true,true,true,true,false,false,false,false,false]
```

This is equivalent to the following:

```
u > 2 and u < 4
```

The expression 'u>2' results in a BoolId array. The Jython 'and' treats this as 'true', as it is a non-empty list, and returns the result of the second expression 'u<4', which is not the intended result.

One way of overcoming this problem is to use the '&' operator instead of 'and' to give the intended result:

```
print (lambda x: (x>2) & (x<4))(u)
# [false,false,false,true,false,false,false,false,false]
```



Warning

This shows how the '&' operator and the 'and' operator are not identical operators.

If you wish to select elements of an array based on a given criterion then we can find out 'where' in a sequence of data a certain type resides (e.g., at what position the maximum value of an array occurs) and how to get the data that fits your selection.

For example, the 'where' method returns the array indices of elements that satisfy a predicate often given as a lambda function. The input to the 'where' method is a Boolean array. This differs from the 'filter' where the actual elements themselves are obtained. Using the modulo function (%) we can find where within an array odd values occur.

```
y = DoubleId([2,6,3,8,1,9])
```

```
print y.where(y%2==1) # [2,4,5] indices of odd elements
```

Now return the actual elements, which can be done in three ways

```
print y[y.where(y%2==1)] # [3.0,1.0,9.0]
print y.filter(lambda y: y%2==1) # [3.0,1.0,9.0]
print y.get(y%2==1) # [3.0,1.0,9.0]
```

Predicates support standard jython operators such as not, and and or:

```
y = Double1d([1,2,3,4])
print y.where(lambda x: x<3 and x>1) # [1]
```

Java/C-style logical operators '!', '&&', and '||' are not allowed.

It can be useful to have the indices, rather than the values, when there are two or more arrays with a predicate applied to one of them. For example:

```
x = Double1d([5,6,7,8])
s = y.where(y%2==1)
print x[s] + y[s] # [6.0,10.0]
```

The **'where' function** can also be used to set values:

```
s = y.where(y%2==1)
y[s] = 0 # Set all matching elements to 0
print y # [0.0,2.0,0.0,4.0]
y[s] = [9,8] # Set matching elements using an array of values
print y # [9.0,2.0,8.0,4.0]
```



Note

You can't use the where function like this:

```
a=Double1d.range(10)
b=a.where(a < 3)
print b[0]
print b[0:2]
print a[b[0]]
```

The last three lines will give an error. Technically, this is because `b` is a `Selection` object rather than a `Jython` or `Numeric` array. For the above to work you need to convert it to `Int1d`:

```
c = b.toInt1d()
print c[0] # Now these three lines will work
print c[0:2]
print a[c[0]]
```

The **'get' method** enables you to grab individual elements or a subset of element values from an array. It requires the input of a Boolean array (e.g., a mask). Along with getting individual elements, there are three other forms. One enables you to select element values based on a `Bool1d` mask:

```
y = Double1d([5,7,8,9])
mask = Bool1d([0,0,1,0])
x = y.get(mask) # x == [8.0]
```

The second form enables you to select on a set of indices, contained in a `Selection` object:

```
indices = Selection(Int1d([2,3]))
x = y.get(indices) # x == [8.0,9.0]
```

The third form enables you to select elements from a range, specified by a `Range` object:

```
range = Range(2,4)
```

```
x = y.get(range) # x == [8.0,9.0]
```

It is possible to combine 'get' calls to perform the same operation as a compound IDL WHERE execution. Let's set up a few arrays first:

```
a = Double1d([1, 2, 3, 4, 5, 6])
b = Double1d([2, 3, 4, 5, 6, 7])
c = Double1d([3, 4, 5, 6, 7, 8])
```

The following operations on the three arrays are the equivalent of the IDL WHERE statement 'where (a ge 2 and b lt 6 and c gt 5)':

```
q = (a >= 2) & (b < 6) & (c > 5)
x = a.get(q),b.get(q),c.get(q) # x == ([4.0], [5.0], [6.0])
```

5.6. Array Access and Slicing

The numeric package introduces the following square brackets notation:

```
[i_0,...,i_n-1]
```

where each element is separated by a comma, and the number of elements must be equal to the rank of the array. Arrays are zero-based which means the first element of an array has index 0 (zero) and the index of the last element of an array is `array.length()-1`.

In addition the package supports the colon (`:`) notation to designate a slice. A slice is a range of indices defined as `i:j`, where `i` is the starting index and inclusive, and it is zero if not specified. The ending index `j` is exclusive and it is equal to `array.length()` if not specified and `array.length()-j` if negative.

The following example illustrates the access to elements in a multi-dimensional array and the use of slices. More examples can be found in the section on Multi-Dimensional Arrays.

```
# define something that is like a rectangular 2x3 array:
# 1 2 3
# 4 5 6
x=Int2d([[1,2,3],[4,5,6]])# Int1d can swallow the jython sequence.
print x # [[1,2,3],[4,5,6]]
print x[1] # 2 (second element of the first row)
print x[1,:] # access a row i.e. [4,5,6]
print x[1,1] # access an individual element i.e. 5
print x[:,1] # [[1,2,3],[4,5,6]]
print x[:,1] # access a column i.e. [2,5]
```

5.7. Making sense of logical operators

Here we try to guide you through the jungle of logical operators you are likely to encounter when using DP.

First of all, since Jython is embedded in DP, it won't surprise anyone that the **Jython logical operators** `and`, `or` and `not` are available. These work like normal Boolean operators (see Appendix C for more details), but using them with arrays (both the native Jython ones and those from the DP Numeric package) can give unexpected and seemingly inexplicable results. See below and also Section 5.5 for an example. The important thing to keep in mind is that these operators do *not* work on an element-by-element basis when applied to arrays, but they evaluate the entire array at once.

Another tool coming straight from the Jython language are the **bitwise operators**, represented by the symbols `&`, `|` and `^`. See again Appendix C for more details. The possible source of confusion here is that these symbols can be used with Numeric arrays (e.g. `Int1d`, `Bool3d` etc.), but what you get

is *not* a bitwise comparison. Instead, these operators perform the usual boolean comparisons, but this time working element by element. Precisely what `and`, `or` and `not` do *not* do.

Finally, Numeric array classes have the `and`, `or` and `xor` *methods* acting like boolean *operators* working element by element. An example will hopefully clarify the differences among all the operators described here:

```
jythonOne = [1, 0, 0, 1]
jythonTwo = [0, 0, 1, 1]
numericOne = Bool1d(jythonOne)
numericTwo = Bool1d(jythonTwo)
print jythonOne and jythonTwo
## [0, 0, 1, 1] # jythonOne is not empty so it is treated as true, which means that
                # jythonTwo is evaluated and returned
print numericOne and numericTwo
## [false,false,true,true] # Same thing as with the Jython native arrays
print jythonOne & jythonTwo
## Here an error is returned
print numericOne & numericTwo
## [false,false,false,true] # Here the operator works element by element
print numericOne.and(numericTwo)
## [false,false,false,true] # Same thing as the & operator
```

5.8. Advanced Tips for Improved Performance

The underlying array operations and functions are very fast, as they are implemented in Java. The overhead of invoking them from Jython is relatively small for large arrays. However, the advanced user may find the following tips useful to improve performance in cases where it becomes a problem.

The arithmetic operations, such as '+', have versions that allow in-place modification of an array without copying. For example:

```
y = Double1d.range(10000)
y = y + 1 # The array is copied
y += 1 # The array is modified in place
```

Copying an array is slow as it involves allocating memory (and subsequently garbage collecting it). For simple operations, such as addition, the copying can take longer than the actual addition.

Function application also involves copying the array. This can be avoided by using the Java API instead of the simple prefix function notation. For example:

```
x = Double1d.range(10000)
x = SIN(x) * COS(x) # This operation involves three copies
x = x.apply(SIN).multiply(x.apply(COS)) # Only one copy
```

When writing array expressions, it is better to group scalar operations together to avoid unnecessary array operations. For example:

```
y = Double1d([1,2,3,4])
print y * 2 * 3 # 2 array multiplications
print y * (2 * 3) # 1 array multiplication
print 2 * 3 * y # 1 array multiplication
```

It is better to avoid explicit loops in the HCSS DP system over the elements of an array. It is often possible to achieve the same effect using existing array operations and functions. For example:

```
sum = 0.0
for i in y:
    sum = sum + i * i # Explicit iteration
```

```
sum = SUM(y * y) # Array operations
```

5.9. Type Conversions

Since the numeric library supports different types it would be very convenient to be able to convert an array from one type to another. The numeric library supports both implicit conversion from within jython for all supported dimensions and explicit conversion from one data type to another.

5.9.1. Explicit conversion

Explicit conversion is supported for all data types by constructing a numeric array from another DP numeric array of the same or a different type. Note however that some explicit conversions may result in rounding and/or truncation of the values e.g. an explicit conversion from Long1d to Double1d will reduce the number of significant digits.

```
i = Int1d([1,2,3])           # [1,2,3]
r = Double1d(i)             # [1.0,2.0,3.0]
c = Complex1d(r)           # [(1.0+0.0j),(2.0+0.0j),(3.0+0.0j)]
b = Byte1d(r)              # [1,2,3]
```

5.9.2. Implicit conversion

Implicit conversions are conversions that can be done by the DP package automatically, provided that such a conversion is a widening operation e.g. from Int1d to Double1d. Implicit narrowing conversions are not allowed and result in an error message as shown below:

```
TypeError: Conversion of class org.python.core.PyFloat to class java.lang.Long implies narrowing.
```

The library supports implicit conversions in the following cases:

- access: [...]
- operators: +, -, *, /, ^ and %
- in-line operators: +, -, *, /, ^ and %

The few examples below show allowed implicit conversions.

```
d = Double1d(5)             # [0.0,0.0,0.0,0.0,0.0]
d[1] = 3                    # [0.0,3.0,0.0,0.0,0.0]
d[1:4] = [-5, 0, 5]        # [0.0,-5.0,0.0,5.0,0.0]
```

Please note that the DP package considers the conversion from int to float and from long to float/double as an automatic widening operation, but some of the least significant digits of the value may be lost during the conversion. You will not be notified of this loss of significant digits.

Another thing to notice is that floating point operations will never throw an exception or error. As shown in the following example, a division by zero results in NaN or Infinity.

```
d = Double1d.range(5)
l = Long1d.range(5)
print d/l                    # [NaN,1.0,1.0,1.0,1.0]
print d/SHIFT(1, 1)        # [0.0,Infinity,2.0,1.5,1.3333333333333333]
```

5.10. Function Library

The numeric package includes a library of basic numeric processing functions, which will continue to grow as development of the library progresses.

The functions that are currently available are outlined below. For further details, reference should be made to the **Javadoc documentation** and **demo programs**.

5.10.1. Basic Functions

Basic double->double functions applicable to double, Double1d, Double2d and Double3d arrays:

```
ABS, ARCCOS, ARCSIN, ARCTAN, CEIL, COS, EXP, FLOOR, LOG,
LOG10, ROUND, SIN, SQRT, SQUARE, TAN
```

These are applied in the form

```
b = SIN(a)
```

b will be an array of the same dimension as a or a single value if a is single valued.

Array functions on Double<n>d returning a double:

```
MIN, MAX, MEAN, MEDIAN, RMS, SUM
```

```
b = MIN(a) #'b' has the minimum value of the array 'a'.
```

Double1d->Double1d functions:

```
REVERSE
```



Warning

Many of these functions have lower case equivalents built-in in Jython. Be aware of which one you are using, because their behaviour could differ in some cases, as shown by the example below which creates a table with Not-a-Number's (NaNs) in it.

```
tt=Double1d.range(10)
tt[0]=Double.NaN
print max(tt)
# NaN
print min(tt)
# NaN
tt[1]=Double.NaN
tt[0]=1.0
print max(tt) # By using the built-in Jython functions
# 9.0
print min(tt)
# 1.0
print MAX(tt) # By using the DP Numeric functions
# NaN
print MIN(tt)
# NaN
```

5.10.2. Integral Transforms

A Discrete Fourier Transform is provided for Complex1d arrays. This uses a radix-2 FFT algorithm for array lengths that are powers of 2 and a Chirp-Z transform for other lengths. Future releases might support multi-dimensional arrays, if required, and optimised transforms of real data.

Window functions are provided for reducing 'leakage' effects using the **Hamming** or *Hanning* window.

Example 5.1 shows the generation of a frequency modulated signal, followed by a FFT both with and without windowing:

```

ts = 1E-6          # Sampling period (sec)
fc = 200000       # Carrier frequency (Hz)
fm = 2000         # Modulation frequency (Hz)
beta = 0.0003     # Modulation index (Hz)
n = 5000          # Number of samples

pi = java.lang.Math.PI # define pi

t = Double1d.range(n) * ts
# t is a 5000 element array holding time values

signal = SIN(2 * pi * fc * t * (1 + beta * COS(2 * pi * fm * t)))
#create the modulated signal with modulation frequency fm and carrier
#frequency fc, t is the array we created above for the time elements.

spectrum = ABS(FFT(Complex1d(signal)))
#spectrum holds the absolute value (ABS) of the FFT of the signal.
#We need to handle these arrays as Complex1d rather than Double1d.

freq = Double1d.range(n) / (n * ts)
#The frequency values for the spectrum.

# Repeat with apodizing
spectrum2 = ABS(FFT(Complex1d(HAMMING(signal))))

```

Example 5.1. FFT of a modulated signal , with and without HAMMING smoothing

The *Inverse Fourier Transform* of a `Complex1d` array (only "x" can be obtained using, e.g., `inverse = IFFT(x)`).

5.10.3. Convolution

Convolution is currently supported for `Double1d` arrays. A direct convolution algorithm is used, although a future release might implement Fourier convolution to improve the speed for large arrays and large kernels. An example of its use is given in Example 5.2.

```

from herschel.ia.numeric.toolbox.filter.Convolution import *
x = Double1d.range(100)
# Create array [0.0, 1.0, 2.0 ... 99.0]
kernel = Double1d([1,1,1])
#provide kernel for the convolution
f = Convolution(kernel)
#create the convolution
y = f(x)
#apply it to the array x. The result is in array y

```

Example 5.2. Example of the use of the convolution algorithm

This illustrates a general approach with the numeric library i.e. general function objects may be instantiated using parameters to create a customised function which can then be applied to one or more sets of data.

The constructor of the `Convolution` class allows optional *keyword* arguments to be specified, to further customise the function:

- The 'center' parameter allow selection of a causal asymmetric filter for time domain filtering or a symmetric filter for spatial domain filtering.
- The 'edge' parameter controls the handling of edge effects, as well as allowing a choice between periodic (circular) and aperiodic convolution.

The following examples show construction of filters using these options:



Note

Make sure you have input the following import line before trying these out.

```
from herschel.ia.numeric.toolbox.filter.Convolution import *
```

Use zeroes for data beyond edges, causal

```
f = Convolution(kernel, center=0, edge=ZEROES)
```

Circular convolution, causal

```
f = Convolution(kernel, center=0, edge=CIRCULAR)
```

Repeat edge values, causal

```
f = Convolution(kernel, center=0, edge=REPEAT)
```

Use zeroes for data beyond edges with centred kernel

```
f = Convolution(kernel, center=1, edge=ZEROES)
```

Circular convolution with centred kernel

```
f = Convolution(kernel, center=1, edge=CIRCULAR)
```

Repeat edge values with centred kernel

```
f = Convolution(kernel, center=1, edge=REPEAT)
```

5.10.4. Boxcar and Gaussian Filters

Finite Impulse Response (FIR) filters and symmetric spatial domain filters can be defined by instantiating the `Convolution` class with appropriate parameters. *In addition, special filter functions are provided for Gaussian filters and box-car filters :*

```
from herschel.ia.numeric.toolbox.filter.Convolution import *

f = GaussianFilter(5, center=1, edge=ZEROES)
f = BoxCarFilter(5, center=0, edge=ZEROES)
```

These filters are subclasses of `Convolution` and hence inherit the use of similar keyword arguments.

5.10.5. Interpolation Functions

Interpolation functions are provided for a variety of common interpolation algorithms.

Example 5.3 illustrates the use of the currently available interpolation functions. The plotting package available for displaying the different interpolation forms (`PlotXY`) is discussed more fully in Chapter 6.


```
from herschel.ia.numeric.toolbox.interp import *
# Create the array x [0.0, 1.0, 2.0, ..., 9.0]
x = Doubleld.range(10)
print x # [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0]
# Create an array y which contains the sine of each element in x
y = SIN(x)
# u contains the values at which to interpolate
u = Doubleld.range(80) / 10 + 1
print u #[1.0,1.1,1.2,1.3...8.6,8.7,8.8,8.9]
# Linear interpolation
# This sets up the interpolation, linear x-y fit
# Interpolate at specified values
interp = LinearInterpolator(x,y)
# Prints out the values interpolated at each position noted in array u
print interp(u) #[0.8414709848,0.848253629...0.5275664375,0.4698424613]

# NearestNeighbour and CubicSpline interpolation may be performed
# in the same way:

# Cubic-spline interpolation
interp = CubicSplineInterpolator(x,y)

# Nearest-neighbour interpolation
interp = NearestNeighborInterpolator(x,y)
```

Example 5.3. Interpolation functions in DP

The result of the interpolations used in the above example is illustrated in Figure 5.1.

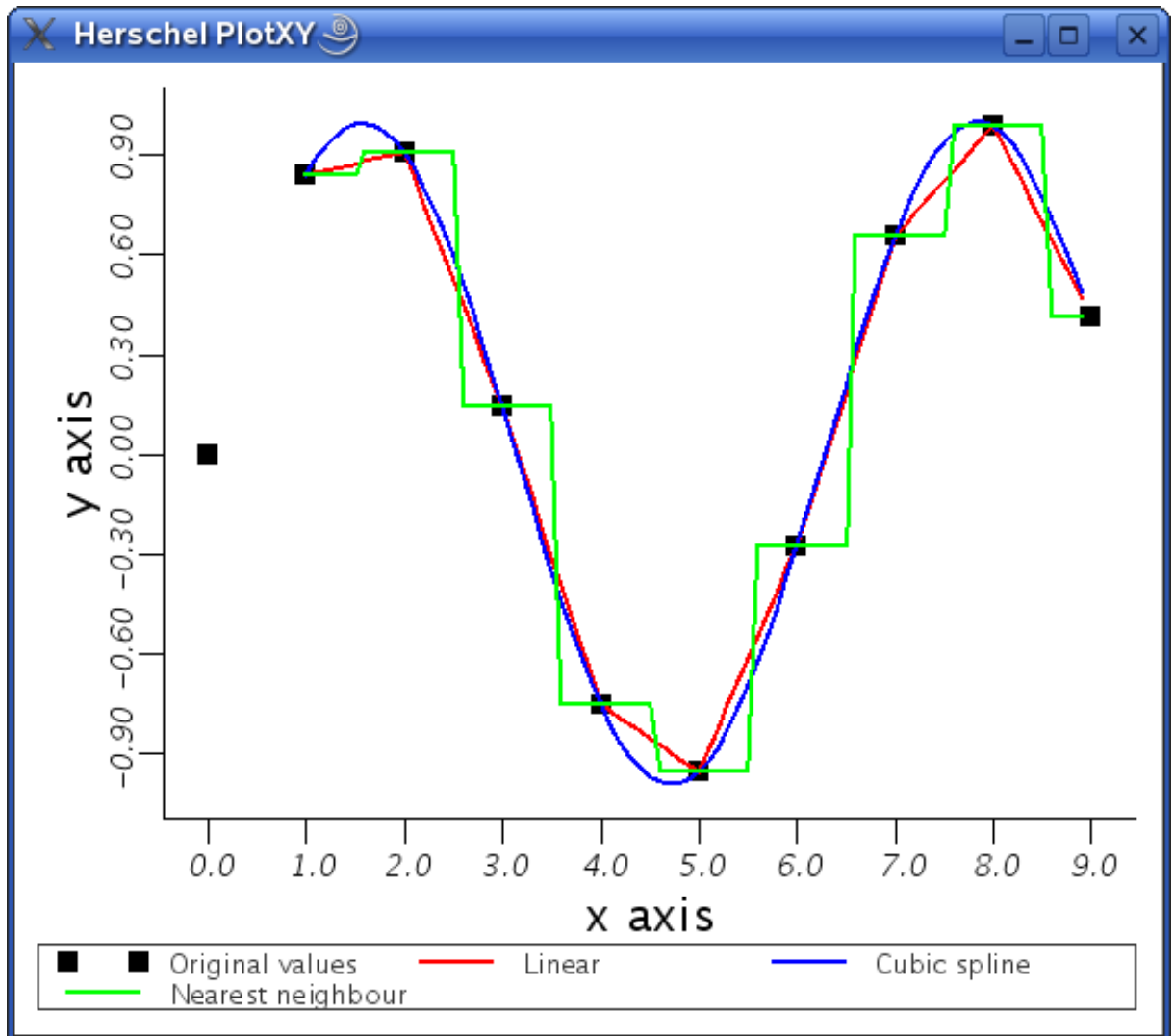


Figure 5.1. Illustration of various forms of interpolation functions.

5.10.6. Basic Fitter Routines

A complete package of advanced data-fitting routines is available and will be more fully discussed in future versions of the User Manual. Here, we provide information on the basic linear and non-linear fitting routines available within DP.

5.10.6.1. General Approach

Input Data: The fitter package expects your data to be in two datasets that are related to each other. Typically, these are `DoubleId` arrays, e.g.,

```
# Data points: each element in x and y define a data point
x = DoubleId.range(12) # Make x vector (the data positions/channels)
y = DoubleId([1.0,1.2,0.9,2.2,3.3,\
4.5,3.6,2.7,1.8,1.2,1.0,1.1]) # Make y vector (the data values)
```

Model Selection: Fitting means adjusting the parameters of a known function, called *model*, so that it best matches the input data. This toolbox provides some pre-defined linear models as well as non-linear models. Viewing your data will hopefully give you some hints about what function model would reflect your input data. For example, if it seems to be polynomial of a certain degree, you would choose a `PolynomialModel`.

**Note**

For the case of non-linear fitters (e.g., used with Gaussians) it is also necessary to provide initial guesses in the form of a parameter set to the model before invoking a fitter. The closer the initial guess for the parameter set to the true values the higher the likelihood that the minimisation will not find a local minimum with wrong/unrealistic parameter estimation.

An example of the use of a linear fitter:

```
# Choose a model: 4th degree polynomial
myModel = PolynomialModel(4)
# Create a fitter and feed it your positions/channels along the array
# (x, a Double1d array) and your model
myFitter = Fitter(x, myModel)
```

Or for a non-linear fitter applied to our array 'x':

```
myModel = GaussModel()
peak = 4.5
channel = 5.5
width=1.0
initialvalues = Double1d([peak, channel, width])
# Apply the initial estimates: peak height, channel position and
# width of gaussian
myModel.setParameters(initialvalues)
# Choose non-linear fitter to use
myFitter = AmoebaFitter(x, myModel) # see later section on available fitters
```

Fit Execution (with and without weights)

```
# Now actually fit the data values at each x position (the y array) to the model
fitresults = myFitter.fit(y)
# Or with associated weights array
fitresults = myFitter.fit(y, yWeights)
```

Results Now the fitter has done its job. We can print the results (`fitresults`) to see the parameters fitted.

```
print fitresults # from using the polynomial fitter
# [1.0993589743591299,-1.1096331908843398,0.8923489704745665,
# -0.14688390313399513,0.006825466200470528]
# provides coefficients of the polynomial fit
print fitresults # from using the Gaussian fitter
# [3.751009700481534,5.353351564022887,2.5098951536394383]
#peak of fit, channel of Gaussian peak, width of Gaussian
```

The fit parameters model are computed and we can start using that model to e.g. re-sample your model fit data:

```
# Re-sample with equally spaced x data points and a finer grid:
xs = Double1d.range(1200) / 100 # Re-sampled x positions
ys = myModel(xs) # Computed y data points
#a plot of xs versus ys plots out 1200 points with the fit.
```

Statistical Information The above procedure demonstrates how to use the fit package to fit your data against a certain model. However, it does not tell you how good the fit actually is. The fitters provide ways to extract such information from the fit.

```
# After fitting
print myFitter.getChiSquared() # Goodness of the fit
# e.g., 2.5765684980727577 for Gaussian fit
print myFitter.autoScale() # How well does the data fit the model.
```

```
# e.g., 0.5350564350372312 for Gaussian fit
print myFitter.getStandardDeviation() # Standard deviations for the parameters.
# e.g., [0.30907540430060004,0.24531121048289006,0.2525757390634412]
# for Gaussian fit parameters
print myFitter.getHessian()           # Retrieve the Hessian matrix
es = myFitter.monteCarloError(xs)     # Errors on the resampled datapoints
# es is now an error array with a length the same as "xs" -- 1200 samples
```

5.10.6.2. Available Linear Models

There are several models that can be used for linear fitting.

In the descriptions below, the models provide parameter fit values $p_0, p_1 \dots p_k$.



Note

In the following examples the parameter subscripts match the position of the parameter in the output array (`fitsresult` in the previous section). So p_0 will be the first element of the `fitsresult` array, p_1 the second one, and so on.

BinomialModel, which allows for the fitting of a binomial model with two variables -- $f(x,y;p) = \sum p_k x^k y^{(d-k)}$, where d is the degree. *Usage: BinomialModel(4)* -- provides a binomial model of degree 4.

PolynomialModel, which allows for the least squares fitting of a polynomial to the data -- $f(x;p) = \sum p_k x^k$. *Usage: PolynomialModel(3)* -- provides a third order polynomial fitting of the data.

SineAmpModel, which allows for the fitting of cosine and sine waves of a given frequency to get amplitudes -- $f(x;p) = p_0 \cos(2 \pi f x) + p_1 \sin(2 \pi f x)$, where x is the data. *Usage: SineAmpModel(f)* -- which provides cosine/sine fits with a frequency, f .

PowerModel, which allows for the fitting of a power law of order k -- $f(x;p) = p_0 x^k$. *Usage: PowerModel(3)* -- provides a third-order power-law fit

CubicSplinesModel, which allows for the fitting of a cubic splines with arbitrary knots settings. *Usage: CubicSplinesModel(5)* -- provides a cubic splines fit with 5 knots.

5.10.6.3. Available Non-Linear Models

There are a number of models that can be used for non-linear fitting. For fitting of these models we need initial values (guesses) for parameters labelled p_0, p_1 and p_2 (see example given in the "General Approach" section).

ArctanModel, which allows for the fitting of a general arctan function -- $f(x;p) = p_0 \arctan(p_1 (x - p_2))$. *Usage: ArctanModel()*

ExpModel, which allows for the fitting of a general exponential function -- $f(x;p) = p_0 \exp(p_1 x)$. *Usage: ExpModel()*

LorentzModel, which allows for the fitting of a Lorentz function -- $f(x;p) = p_0 (p_2 / ((x - p_1)^2 + p_2^2))$. *Usage: LorentzModel()*

PowerLawModel, which allows for the fitting of a general power-law function -- $f(x;p) = p_0 (x - p_1)^{p_2}$. *Usage: PowerLawModel()*

SincModel, which allows for the fitting of a sinc function -- $f(x;p) = p_0 \sin((x - p_1)/p_2) / (x - p_1)/p_2$. *Usage: SincModel()*

SineModel, which allows for the fitting of a general cosine/sine wave -- $f(x;p) = p_1 \cos(2 \pi p_0 x) + p_2 \sin(2 \pi p_0 x)$. *Usage: SineModel()*

GaussModel, which allows for the fitting of a 1-D gaussian -- $f(x;p) = p_0 \exp(-0.5((x-p_1)/p_2)^2)$, where p_0 is the amplitude, p_1 the x-shift (from zero) and p_2 the sigma of the fit, with initial values of 1.0, 0.0 and 1.0 respectively. Note that *Gauss2DModel* produces a fit to 2D data. Usage: *GaussModel()*

User supplied non-linear function, which allows for fitting a function (linear or non-linear) constructed by the user. This function must be put in a jython class and optionally the user could provide an analytical calculation of the partial derivatives with respect to the parameters (otherwise they are calculated numerically). This is shown in the following example for the following function of four parameters: $f(x;p) = p_0/(1+(x/p_1)^2)^{p_2} + p_3$ (the so called beta-profile):

```
from herschel.ia.numeric.toolbox.fit import NonLinearPyModel

class BetaModel(NonLinearPyModel):
# the full 4-parameter beta-model with partial derivatives
# f(x:p) = p0/(1+(x/p1)**2)**p2 + p3
#
#
# npar = 4
# def __init__(self):
#     # Constructor
#     NonLinearPyModel.__init__(self, self.npar)
#     self.setParameters(DoubleIcd([1,1,-1,1]))
#
# def pyResult(self,x,p):
#     model = p[0]/(1.0 + (x/p[1])**2)**p[2] + p[3]
#     return model
#
# def pyPartial(self,x,p):
#     # the partial derivatives
#     arg1 = 1.0 + (x/p[1])**2
#     dp = DoubleIcd(self.npar)
#     #
#     dp[0] = 1.0/arg1**p[2] # df/dp0
#     dp[1] = 2.0*p[0]*p[2]*x*x/((p[1]**3)*arg1**(p[2]+1.0)) # df/dp1
#     dp[2] = -p[0]*Math.log(arg1)/arg1**p[2] # df/dp2
#     dp[3] = 1.0 # df/dp3
#     return dp
# def myName( self ):
#     # Return an explicatory name (String). Optional.
#     return "beta-profile: f(x:p) = p[0]*{1 + (x/p[1])2}^p[2] + p[3]"
```

Once we define the function as shown in the example then we can proceed as before and create a model and then perform the fitting using either the Lavenberg-Marquardt or Amoeba fitters:

```
bm = BetaModel()
bm.setParameters(DoubleIcd([10.0,1.0,-2.0,5.0]))
myfit = LevenbergMarquardtFitter(x, bm) # see section on available fitters below
# or myfit = AmoebaFitter(x, bm)
result = myfit.fit(y)
print result
```

5.10.6.4. Compound and Mixed Models

It is possible to add two models, e.g. if one wants to fit a spectral line (a Gaussian) on a background (a Polynomial). The resulting model is non-linear.

```
myModel = GaussModel() # Define a Gaussian
myModel += PolynomialModel(1) # Add a Polynomial to it of order 1. Only with +=
print myModel.toString() # Information about the model
```

More models can be added if wished.

5.10.6.5. Available Fitters

Fitter. Fitter for linear models. You create a fitter by providing the model assumption and the x points of the data. With that information you compute the parameters within the model by fitting the y data

points. Once the computation of those parameters is done, you can extract statistical information from the fitter. **Syntax:** `myFitter=Fitter(xDataPoints, model)`

LevenbergMarquardtFitter. Fitter for non-linear models. The LMFitter is a gradient fitter, which means that it goes downhill from the starting location until it cannot go down anymore. There is no guarantee that the minimum found is an absolute or global minimum. If the chisq-landscape is multimodal it ends in the first minimum it finds. See also Numerical Recipes, Ch 15.5. **Syntax:** `myFitter = LevenbergMarquardtFitter(xDataPoints, model)`

AmoebaFitter. Fitter for non-linear models. The AmoebaFitter implements the Nelder-Mead simplex method. It comes in 2 varieties, one where the simplex simply goes downhill (temperature = 0) and one which implements an annealing scheme. Depending on the temperature, the simplex sometimes takes an uphill step, while a downhill steps always is taken. This way it is able to escape from local minima and it has a better chance of finding the global minimum. No guarantee, however. *AmoebaFitter* is also able to handle limits on the parameter range. Parameters stay within the limits when they are set. See also Numerical Recipes, Ch. 10.4 and 10.9. **Syntax:** `myFitter = LevenbergMarquardtFitter(xDataPoints, model)`

5.10.6.6. Obtaining a Model Fit to 1D and 2D Data

1D Fit Example

Example 5.4 shows how a polynomial can be fitted to a set of 1D data.

```

# Create some data
x = Double1d([3,4,6,7,8,10,11,13]) # These are the positions of the 1D data
y = Double1d([2,4,5,6,5,6,7,9]) # These are the data values at each position
# The created arrays are:
print x # [3.0,4.0,6.0,7.0,8.0,10.0,11.0,13.0]
print y # [2.0,4.0,5.0,6.0,5.0,6.0,7.0,9.0]

# Decide that we will fit it with a polynomial

model = PolynomialModel(3)

# The Fitter class expects the 'x' data point positions and the model.
# In the binomial case, a Double2d array of x,y values is required.
# The Fitter class deals with non-iterative models only.
# [Note: For non-linear models the fitter toolbox provides
# the AmoebaFitter and the LevenbergMarquardtFitter]

fitter = Fitter(x, model)

# Now we fit the data values(y); the returned array contains the parameters
# that make up a 3rd degree polynomial.
# Note: the model that we fed into the fitter is modified along the
# way, such that it contains the computed parameters of the polynomial.

poly = fitter.fit(y)

# Printing the fit results (truncate to 3 decimal places to fit in line)

print poly # [-6.921,4.463,-0.543,0.022]

# ..and also getting the Chi-squared. The fitter has already been applied
# and we can use the getChiSquared() method to determine the fit

print "Chi-Squared = ", fitter.getChiSquared()
# Chi-Squared = 0.9933079890409999

# The fitted polynomial can then be applied as a function to interpolate
# between fitted points. Interpolate at 'n' uniformly spaced x values

n = 100
u = MIN(x) + Double1d.range(n + 1) * ((MAX(x) - MIN(x)) / n)

# Apply the model
umodel = model(u)

# Now we can plot the data (x vs y) and the polynomial fit (u vs umodel)
# Set up the plot space
plot = PlotXY()
# Plot x against y in blue.
plot[0] = LayerXY(x, y, name = "Data")
# Overlay a second plot showing the polynomial fit in green.
plot[1] = LayerXY(u, umodel, name = "Fit", color = java.awt.Color.green)

```

Example 5.4. A 1D polynomial fit.

The final plotted display should look like Figure 5.2

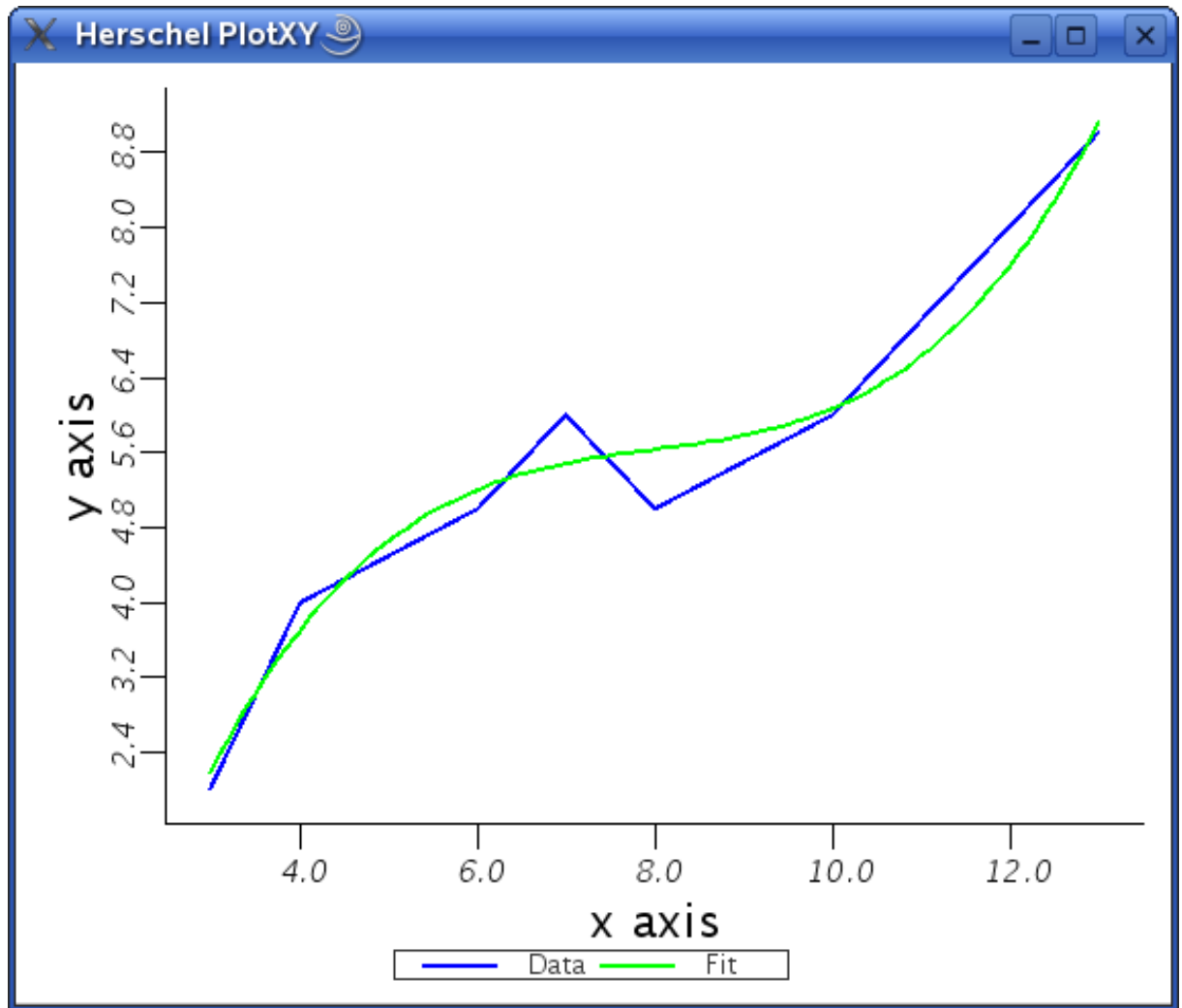


Figure 5.2. Illustration of polynomial fit.

2D Fit Example

For 2D data we express the positions at which we have data by a `Double2d` array -- this is basically a list of x , y positions at which we have known data values that we will fit a 2D Gaussian to. So the x array in our previous example is now replaced by a 2D array of data positions. The y array has the data values at those positions.

In Example 5.5, an array with values that provide a Gaussian with random noise added is fitted by the `Gauss2D` model.


```

# We start by making a little routine that creates the data for us.
# The output contains the 'xy' positions as a Double2d array and the data
# values are held in in the Double1d array 'y2'.
def makeData():
# Define some constants
    N = 9          # We will create an array that is NxN
    a0 = 10.0     # Amplitude of gaussian
    x0 = 0.7      # x position of gaussian
    y0 = -0.3     # y position of gaussian
    s0 = 0.4      # Width
# Make data with an underlying gaussian model.
    x = Double1d.range(N) / 2.0 - 2 # create x values
    NN = N * N # the number of x and y positions (NxN)
    xy = Double2d(NN, 2) # Create empty array of xy positions
    ym = Double1d(NN) # Create empty array for amplitude of pure Gaussian
    y2 = Double1d(NN) # Create empty array for Gaussian with noise (our
    data).
# These have amplitude values only.
    rng = java.util.Random( 12345 ) #provide a random amplitude (noise)
# To add to our model Gaussian with a seed value.
    si = 1.0 / s0 #just inverse of Gaussian width to be used
    for i in Int1d.range(NN):
        xy[i,0] = x[i / N] # Fills x positions for our data array
        xy[i,1] = x[i % N] # Fills y positions for our data array
        xx = ( xy[i,0] - x0 ) * si
        yy = ( xy[i,1] - y0 ) * si
        ym[i] = a0 * EXP(-0.5 * xx * xx) * EXP(-0.5 * yy * yy)
        # Fills 1d array with amplitude values...
        y2[i] = ym[i] + 0.2 * rng.nextGaussian() # ...and adds noise to it
    return xy,y2

# Create the array with a 2D gaussian in it using the above routine.
a = makeData()
# The first item in "a" has the xy positions in it
xy=a[0]
# The second item has the data values
y2=a[1]

# Define the model to be used in the fit
gaus2d = Gauss2DModel()

# Define the fitter: LevenbergMarquardt, a non-linear fitter is needed for
# a gaussian fit. We could use an AmoebaFitter here also -- user preference.
fitter = LevenbergMarquardtFitter(xy, gaus2d)

# A useful way to make data formats prettier for the printout of our results
F = DataFormatter()
# Find the parameters
param = fitter.fit(y2)
print "Parameters %s" % F.p(param)
# Parameters [ 9.645  0.694  -0.300  0.413]
print "Parameters are: gaussian height, x position, y position, width"
#Parameters are: gaussian height, x position, y position, width
# Find the standard deviations of the all four parameters...
stdev = fitter.getStandardDeviation();
print "Stand Devs %s" % F.p(stdev)
#Stand Devs [ 0.218  0.009  0.009  0.007]

# ...and the chi-squared for the fit
print "ChiSq      %s" % F.p(fitter.getChiSquared())
#ChiSq          3.552

```

Example 5.5. A 2D Gaussian fit

5.10.7. Spectral Fitting.

This section describes how to use the spectrum fitting toolbox in HCSS to fit a spectrum. To access the toolbox it will need to be loaded from the into the session. This can be done by typing in the following in the JIDE command line interface.

```
from herchel.ia.toolbox.spectrum.fit import *
```

The toolbox is continuing to be developed and it is expected that new features will be added to what is described here. Features that are certain to be added are listed in the 'To Be Added' section below.

5.10.7.1. Data format

The data that is used by the classes can be any Java or Jython object, as long as it implements the `SpectralSegment` interface (e.g., extracted from a `SpectrumId` object).

You can create a `SpectralSegment` using a little helper class, `FitData`. This class takes two `DoubleId`'s (representing wavelength/frequency and flux/values) and wraps them into a `SpectralSegment`.

If you have two `DoubleId` arrays, `x` and `y`, then the statement:

```
data = FitData(x,y)
```

creates a `SpectralSegment`.

5.10.7.2. General Usage

In general, data to be fitted contains three kinds of features:

- a background/continuum level
- >one or more spectral lines
- noise

These can be fitted using the `SpectrumFitter` tool.

The purpose of the `SpectrumFitter` is to fit models to the background and the spectral lines in such a way that when the models are subtracted from the data, the residual only contains the noise.

Although fitting spectral lines and the background does not differ mathematically, the two cases must be handled separately. That is, you better first fit the background, subtract that from the data, and only then fit the lines.

5.10.7.3. Fitting your data

As the user you interact with the `SpectrumFitter` tool. To have more control over the models (see below) you can also interact with the class `SpectrumModel`.

Note that you normally must know where (approximately) you expect a spectral feature in your data to be, plus its expected shape, and rough shape parameters. So, an initial guess is required - if this guess is completely wrong you may end-up fitting noise rather than your spectral lines.

The `SpectrumFitter` tool provides graphical information on the fitted data to assess the fits that are made.

5.10.7.4. A Simple Fit Case

The simplest spectral fitting case involves data with one spectral line and with no background/continuum.

The basics are, a) create a `SpectrumFitter`; b) add models to it.

We assume you know that you have a `SpectralSegment` which contains the spectral line has a Gaussian shape that is located near x_0 , has an amplitude of about a_0 , and a width of about s_0 (the exact values of a_0 and s_0 are not so important). The following is an example:

```
x = Double1d.range(15)
y = Double1d([0.0,0.1,-0.1,0.05,0.1,0.2,1.0,3.6, \
  2.5,1.5,0.7,0.0,0.1,-0.13,-0.01])
data=FitData(x,y)
# This has a peak near value number 7 with an
# amplitude of 3.6 and a width close to 1.
x0 = 7.0
a0 = 3.6
s0 = 1.0
# These are our initial guesses.
```

We can fit this using the `SpectrumFitter`:

```
sf = SpectrumFitter(data) # note that a plot of the data is
                          # automatically drawn in a separate window
# see Figure 5.3
sf.addModel('gauss', [a0, x0, s0]) # note the square brackets
                                   # and the order of the parameters
print sf # this prints out the fitted Gaussian parameters
        # and their standard deviations.
# Fit results:
        # p0 = 3.3890821693817763, stddev= 0.2568383201833762
        # p1 = 7.444866152807009, stddev= 0.09308190130219554
        # p2 = 1.0796490360796016, stddev= 0.09333220808910589
# for the amplitude, position and width respectively.
```

Figure 5.3. Spectrum fit data setup.

The result of adding the model is the production of two further plots. One plot contains:

- the data (blue line)
- the input model as given by you (green line)
- the resulting fit (red line)

The second plot displays the residuals. See Figure 5.4 and Figure 5.5.

Figure 5.4. Data fit - data in blue, input model in green, fit in red

Figure 5.5. Residuals on the fitted data

5.10.7.5. Available Models For Fitting

There are a number of models available for fitting. In order to see the available models in the system at any time you can use the following.

```
print SpectrumFitter().info()
```

This command provides a listing of available models that can be fit. If we pick one of these models we can get more information on it. For example we can look to fit a polynomial -- the 'poly' model.

```
print SpectrumFitter().info('poly')
```

This indicates there is one constructor (only one way of calling it). The order needs to be given in one array and initial parameter guesses in a second array.

```
from herschel.ia.toolbox.spectrum.fit import *
from herschel.ia.toolbox.spectrum.fit.testdata import *
#There are 7 inbuilt datasets for spectrum fit checking and illustration
m = MakeData(3) # integer value represents different models
m.addNoise(10) # add some noise to the data
# now do fit -- the guess and final model fit are displayed overlaid on the data
sf = SpectrumFitter(m) # setup spectrumfitter
mod=sf.addModel('poly',[3],[1.0,0.0,0.0,0.0])
# 3rd order poly model and guess for fit parameters
sf.doFit() # fit displayed.
print sf # provides fitted parameters with their standard deviations
```

The models currently available and an illustration of their use is given in Table 5.1.

Table 5.1. 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])
'gauss'	mod=sf.addModel('gauss',[amplitude, position, width])
'gaussmix'	mod=sf.addModel('gaussmix',[amplitude, position, width])
'harmonic'	mod=sf.addModel('harmonic',[Order,Period],[params]). Number of parameters provided = 2*order + 1
'lorentz'	mod=sf.addModel('lorentz',[amplitude, shift, gamma])
'pade'	mod=sf.addModel('pade',[Num,Denom],[params]). Number of parameters provided = Num + Denom + 1
'poly'	mod=sf.addModel('poly',[Order],[params]). Number of parameters provided = Order + 1
'power'	mod=sf.addModel('power',[Degree],[param]). Number of parameters provided = 1
'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])
'sinemixed'	mod=sf.addModel('sinemixed', [frequency, cosine amp, sine amp])

5.10.7.6. Multiple Line Fitting

If, in the simple line case above, the residual is only noise, you have completed your fit. If not, then there may be another spectral line in your data. From the original data or from the residual you can often determine the initial parameters of a second line: a_1 , x_1 , s_1 . In order to include a fit to this second line also we can simply add another model to the fitter by using the 'addModel' method:

```
sf.addModel('gauss', [a1, x1, s1])
```

This will update the fit and plots automatically. In the first plot you will now also see the two models separately using the fitted parameters as black lines.

5.10.7.7. Background/continuum Fitting

Background/continuum fitting is not treated differently from the above. The only difference is the model used to fit the background.

When being combined with spectral line fits, it is best to fit the background first then add the spectral line model fit. If you don't, the fit of your spectral lines will initially be quite poor.

One model to use for a background is a polynomial. For a first order Polynomial ($y = c_0 + c_1*x$):

```
sf.addModel('poly', 1, [c0, c1]) # the second value is the polynomial order
```

For a higher order (n):

```
sf.addModel('poly', n, [c0, c1, ..., cn])
```

5.10.7.8. Fit of Line and Continuum

We can fit a line and continuum simultaneously by adding more than one model before doing the fit (e.g., a polynomial and gaussian model). We can then do a global fit. An example is given below.

```
#import the appropriate packages
from herschel.ia.toolbox.spectrum.fit import *
from herschel.ia.toolbox.spectrum.fit.testdata import *
m = MakeData(5) # values represent different models
m.addNoise(10) # add noise to the model
sf = SpectrumFitter(m)
mod=sf.addModel('gauss',[4.0,30.0,1.0]) #also plots initial guess
mod=sf.addModel('gauss',[1.2,10.0,2.0]) #second line
mod=sf.addModel('poly',[3.0],[0.0,0.0,0.0,0.0]) # polynomial for continuum
sf.doGlobalFit() #fits all models at the same time -- residual plot also shown
```

The results of this are a plot of the data, initial guess and fit (in red) plus a separate plot of the fit residuals (see Figure 5.6 and Figure 5.7).

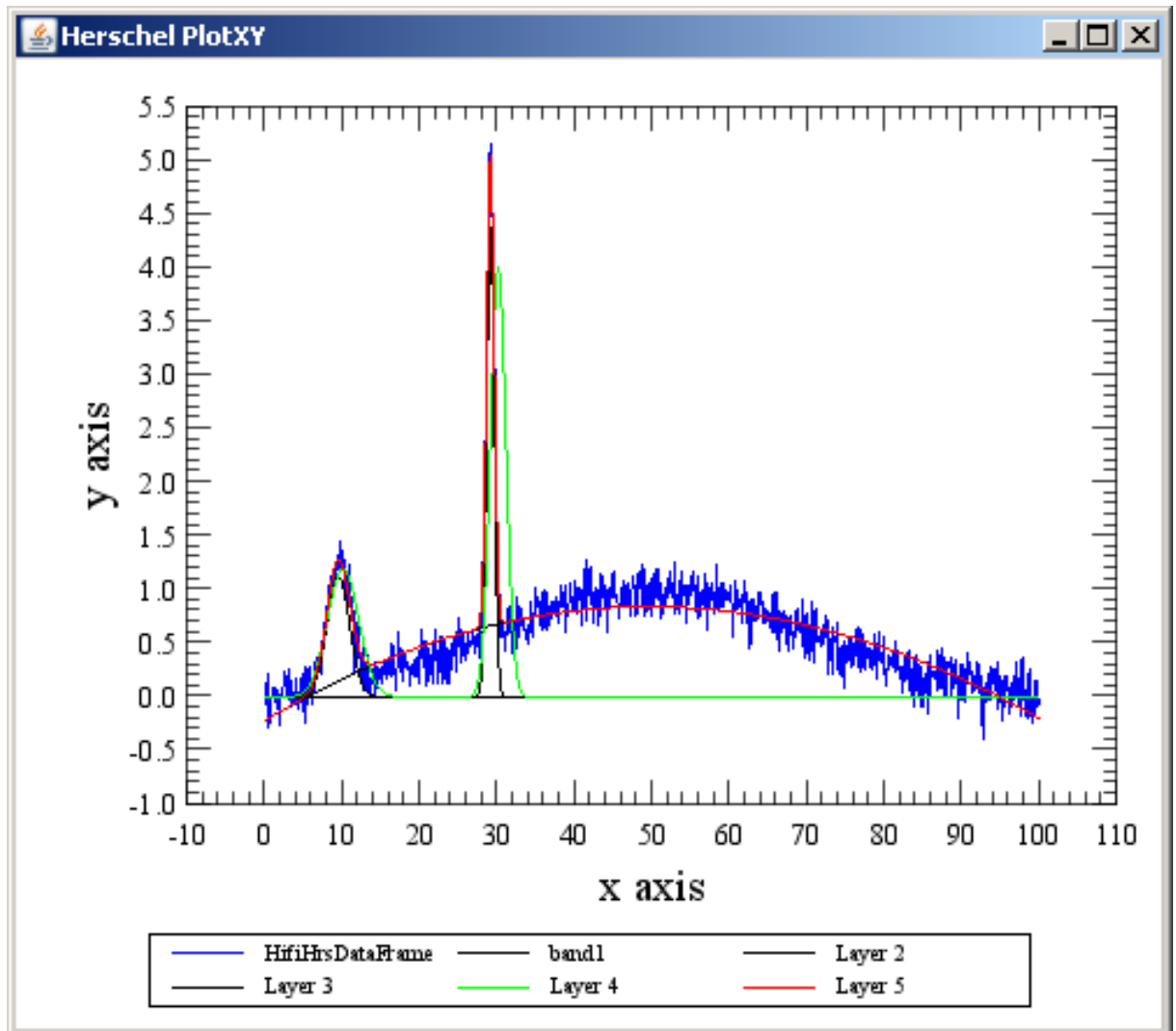


Figure 5.6. Fit using multiple models. In black are the individual guesses, in green the total initial guess and in red the final fit.

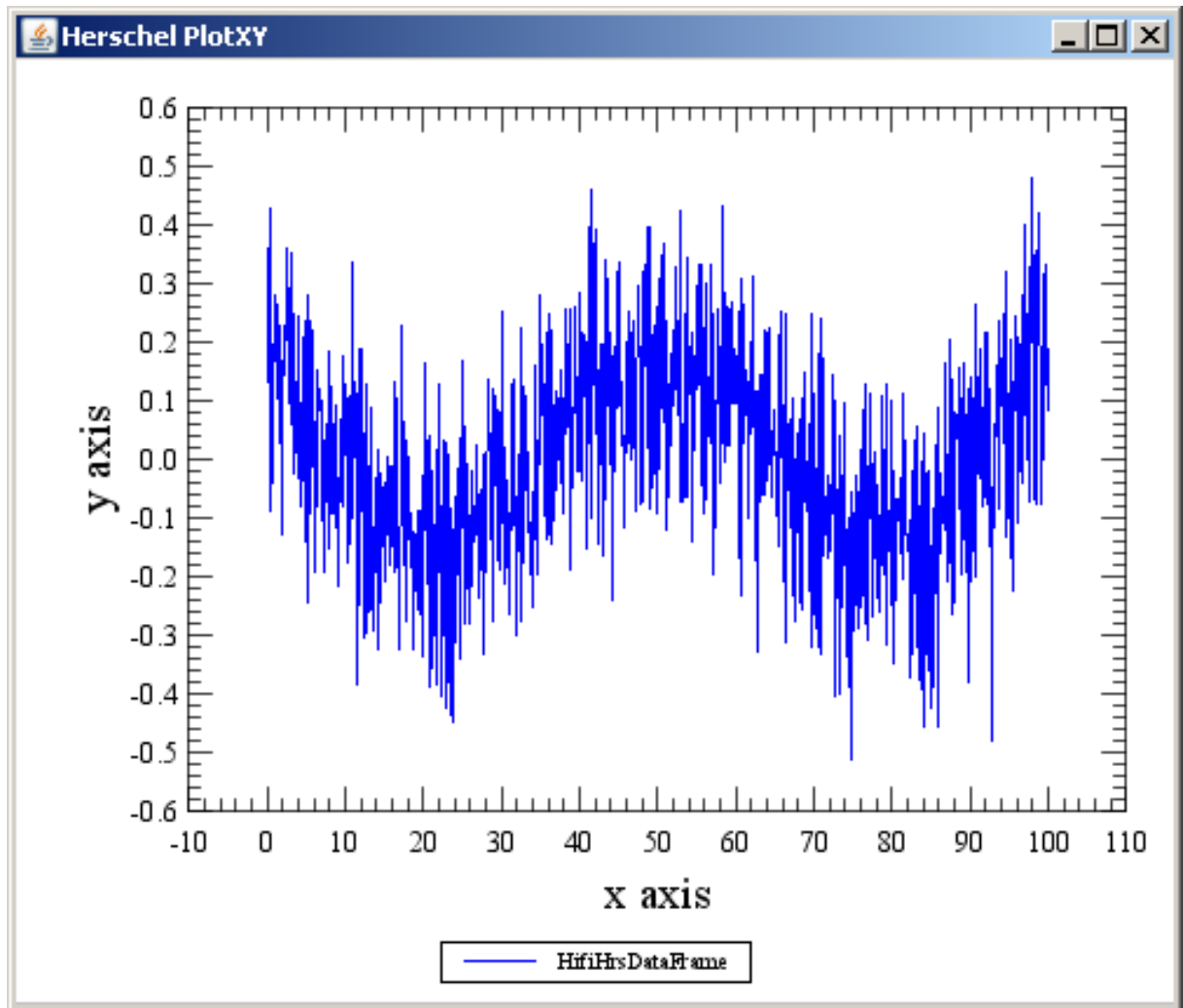


Figure 5.7. Residuals on the multiple model fit data shown in Figure 5.6.

5.10.7.9. Changing Parameters

If you wish to change the initial parameters of any of the models, you can use the `setParameters` method of the models. To use them you must have a reference (label) to the model. This is in fact the return value of the `addModel` operation. In the example below, the label is simply 'm':

```
m = sf.addModel(...) # m is now a reference we can do things with
```

To change the initial parameters of the model

```
m.setParameters([...])
```

A new fit will be made on the fly and your plot display updated.

5.10.7.10. Removing Fitted Models

Removing models can only be done when you have a reference to the Model (as above). There are two ways to remove models:

```
sf.removeModel(m)
```

Or:

```
m.remove()
```

5.10.7.11. Using Fit Parameters

Once you are satisfied with a fit, you can set the fitted parameters as the default for the models:

```
m.useResults()
```

This may be useful when using the same models for a following dataset.

5.10.7.12. Subtracting a Fit

You can subtract the model from the dataset:

```
sf.subtractModel(m)
```

This also removes the model from the fitter tool.

5.10.7.13. New Data

Once you are satisfied with your models, you may want to apply them to a different dataset as well. This can be done with the operation:

```
sf.setData(otherData) # this replaces the data held in the
                       # SpectrumFitter with the SpectralSegment
                       # held in the variable 'otherData'
```

Once again, the fit will be redone on the fly.

5.10.7.14. Functions To Be Added in the Future

>The following features are likely to be added to the system:

- add more model types
- subtract a model from the data, continue with the residuals;
- fix any of the given parameters in a model;
- select parts of the X-axis to include/exclude in the fit;
- make an initial guess for the model parameters.

5.10.8. Matrix Manipulations

Most of the utilities for dealing with matrices are provided by the `numeric.toolbox.matrix` package. However, we must not forget that simple vectors are just matrices with just one row (or one column), so even vector classes like `Double1d` provides tools like a `dotProduct` method for scalar multiplication of vectors:

```
x = Double1d([1,2,3,4])
y = Double1d([1,3,5,7])
print x.dotProduct(y) # 50.0
```

Now let us take a closer look to the `numeric.toolbox.matrix` package and its special classes and function objects for matrix multiplication and transposition. We will start right away with a short example:

```
x = Double2d([[2,4,6],[1,3,5]])
y = TRANSPOSE(x)
z = MatrixMultiply(y)(x)
print z
```

Hence, it is important **not** to use the Jython `*` operator for matrix multiplication. However, the `+` operator performs element-wise addition as required.

It is also possible to multiply a matrix by a vector as follows (since, as we already pointed out, a vector is nothing more than a matrix with just one row or column):

```
a = Double2d([[1,2,3],[7,5,4],[7,4,9]])
b = Double1d([4,1,7])
print MatrixMultiply(b)(a) # [27.0,61.0,95.0]
```



Warning

The correct syntax to multiply matrix *a* by matrix *b* is `MatrixMultiply(b)(a)`.

Another matrix class can be used to solve matrix equations. For example, if we wanted to solve the matrix equation: $A \cdot X = B$

```
x = MatrixSolve(b)(a)
print x # [-0.9838709677419352,0.5322580645161287,1.3064516129032258]
```

Other useful tools for matrix manipulation are listed below.

DETERMINANT Yields the determinant of a square matrix given by a `Double2d` array.

```
A=Double2d([[1,2],[3,4]])
print DETERMINANT(A) # -2.0
```

Note: This currently does not work for complex matrices.

INVERSE Returns the inverse of a square matrix.

```
A=Float2d([[1,2],[3,4]])
print INVERSE(A) # [[-2.0,1.0],[1.5,-0.5]]
```

Note: This currently does not work for complex matrices.

TRANSDPOSE Gives the transposed matrix.

```
A=Int2d([[1,2],[3,4],[5,6]])
print TRANSDPOSE(A) # [[1,3,5],[2,4,6]]
```

You might find a bit confusing that some names, like `dotProduct`, start with a lowercase letter and have all the other initials capitalised, while other names, like `MatrixMultiply`, have *all* initials capitalised, and there is a fair share of names like `TRANSDPOSE` with all uppercase letters. You can find more about these quirks in the appropriately named Section 3.20.

5.10.9. Random numbers generation



Note

For simplicity we will speak of *random* numbers throughout this section, even if we know very well that a computer can only create *pseudorandom* numbers. Discussing the subtleties of generating (pseudo-)random numbers on a computer is beyond the scope of this section.

To create random numbers with DP we first have to instantiate a *generator*. There are three generators currently available:

- `RandomUniform`: generates random numbers in the range $0 \leq x < 1$ if invoked without parameters, like this:

```
myGenerator = RandomUniform()
```

It is also possible to give a maximum value different from 1 to have random numbers created in the range $0 \leq x < \text{max}$:

```
myGenerator = RandomUniform(max)
```

- `RandomGauss`: generates random numbers following a Gaussian distribution.
- `RandomPoisson`: generates random numbers following a Poisson distribution of specified mean value greater than zero. It is instantiated like this:

```
myGenerator = RandomPoisson(mean)
```

It can only produce integer-type random numbers (`int`, `short` and `long`).

In all cases what is being used under the hood is the Donald Knuth generator (see *The Art of Computer Programming*, Volume 2, Section 3.2.1) as implemented in the `java.util.Random` class.

Once we have a generator in place, how do we create random numbers? The handy feature is that we can create a single scalar random number or an array of any size and dimension we like (as long as it fits in memory). Just put the type of numeric value you want as input, and the output will be the same thing, but populated with random numbers. A few examples:

```
myGenerator = RandomUniform() # Generating random numbers between 0 and 1
print myGenerator(0.0) # We want a floating point random number...
# 0.8754230073094597 # ...and there it is (don't expect to get the
# same number)
x = Double1d(10) # Now for an array of ten doubles...
print myGenerator(x) # We leave it to you to see the result
print myGenerator(Double1d(10)) # Of course you can create the input on the fly
print myGenerator(Int1d(100)) # What's the result of this one? Does it make sense?
```

You might have been puzzled to see a hundred zeroes scroll on your screen after executing the last command of the example. It's not so surprising if we think that we asked the computer to produce *integer* random numbers between zero and one, *excluding one*. The choice of possible values was pretty limited.

If we want to change the *seed* of the random number generator we can do so by the `setSeed` method, which takes a long parameter as an input:

```
myGenerator.setSeed(54653856L)
```

5.10.10. Numeric Integration

Numeric integration in DP is implemented via an *Integrator* interface. The function to be integrated has to be declared as a class of a *RealFunction* containing a method called *calc* which takes one argument, the independent variable.

The following Integrators for a standard integration interval [a,b] are available:

- `RectangularIntegrator`
- `RombergIntegrator`
- `SimpsonIntegrator`
- `TrapezoidalIntegrator`
- `GaussianQuad4Integrator`
- `GaussianQuad5Integrator`
- `GaussLegendreIntegrator`

All these integrators have two arguments for initialisation: the lower limit of integration (a) and the upper limit (b). Once the integrator is initialised and the user function is defined then to perform the

integration a method called *integrate()* is executed with an argument the user function. This is shown in the following example:

```
from herschel.ia.numeric.toolbox import RealFunction

class MyFunction(RealFunction):
    def calc(self,x):
        return x*x

f = MyFunction()
a = -3.0
b = 3.0
i = RombergIntegrator(a, b)
print i.integrate(f) # 18.0
print "Analytical answer: ",(b**3 - a**3)/3.0
```

The following special cases of numeric integration are also implemented:

- GaussHermiteIntegrator: for integration with limits $(-\infty, +\infty)$ of a special class of functions

$$\int_{-\infty}^{+\infty} e^{-x^2} f(x) dx$$

- GaussLaguerreIntegrator: for integration with limits $[0, +\infty)$ of a special class of functions

$$\int_0^{+\infty} x^\alpha e^{-x} f(x) dx$$

The input for the integrator initialisation is α .

- GaussJacobiIntegrator: for integration with limits $[-1, 1]$ for a special class of functions

$$\int_{-1}^1 (1-x)^\alpha (1+x)^\beta f(x) dx$$

The input for the integrator initialisation are α and β .

If a tabular data of x,y is to be integrated then it is necessary to interpolate first and then apply a suitable integrator. This is shown in the following example:

```
from herschel.ia.numeric.toolbox import RealFunction

x = 0.1 + 1.9*DoubleItd.range(11)/10.0 # 11 points between 0.1 and 2.0
y = 1.0/x

f = CubicSplineInterpolator(x,y) # interpolate first.
a = 0.1
b = 2.0
integrator = SimpsonIntegrator(a, b) # use Simpson's rule

res = integrator.integrate(f) #
print "Result: ",res
print "Analytical result: ",LOG(b) - LOG(a)
```

5.10.11. Interpolating Discrete Data

If the objective is to integrate discrete data, this can be done by means of a *FitterFunction*, which is a function that interpolates the given data, with a specific model. For example:

```

from herchel.ia.toolbox.fit import FitterFunction

# x, y are DoubleIcd that represent the abscissas and values of our data
f = FitterFunction(x, y, PolynomialModel(3)) # Uses a Fitter
g = FitterFunction(x, y, PolynomialModel(2), FitterFunction.AMOEBA)
# Uses an AmoebaFitter

```

If more precise fitting is needed, you can do it by yourself, and then pass the already built fitter (or the model) to the FitterFunction:

```

# x, y are DoubleIcd that represent the abscissas and values of the data
model = CubicSplinesModel(x)
fitter = AmoebaFitter(x, model)
fitter.setSimplex(params, range) # customize the fitter as you want
fitter.fit(y)
f = FitterFunction(fitter) # or f = FitterFunction(model)

```

If one of the defined interpolators suites your needs, it can be used directly, instead of a FitterFunction. For example:

```

# x, y are DoubleIcd that represent the abscissas and values of the data
f = CubicSplineInterpolator(x, y)

```

5.11. Example Programs

The HCSS distribution includes a number of Jython example programs that demonstrate not only basic arrays functions but also use of filters, fitters, Fourier transforms, etc. They are currently kept at ftp://ftp.rssd.esa.int/pub/HERSCHEL/csd/releases/doc_ia/ia/demo/scripts. These are:

numeric_whatisNew.py	Example of the newest components of the numeric package.
numeric_demo.py	Example of how to use the 1D functionality.
numeric_2D_demo.py	Example of how to use the 2D functionality
convolution_demo.py	Example of how to use the convolution functionality
polyfitter_demo.py	Example of how to perform polynomial fitting

5.12. Mathematical Operations on Spectra

5.12.1. Introduction

The spectrum arithmetic toolbox allows to combine Herschel spectrum data. Operations are performed either on subclasses of spectrum datasets (`Spectrum1d`, `Spectrum2d`), on cubes (`SimpleCube`, `SlicedCube`), or on products containing such data structures (e.g., `HifiTimelineProduct`).

Operations on Spectra include Selection and Arithmetic Operations.

- *Selection*: Provide means of selecting those spectra that can be combined. For instance HIFI cold-load spectra, ON spectra, etc. Selection can be applied to datasets, such as rows of a `Spectrum2d`, or to tables within a product, such as datasets included in a `HifiTimelineProduct`.
- *Arithmetic Operations*: Provide means of combining the selected spectra. This includes:
 - Basic arithmetic operations such as addition, subtraction, multiplication, or applications of scalar functions.

- Statistical operations such as mean, median, variance, standard deviation or percentiles for samples / selections of spectra.
- Data transformations such as smoothing or frequency re-sampling.

It is planned that the arithmetic toolbox will provide generic functionality for all instruments (HIFI, PACS and SPIRE). Instrument-specific behaviour will be pre-configured by defaults in the system but can also be overwritten by the user.

5.12.2. Toolbox Primer: Selection

We present the power of the toolbox with a few code examples. Assume we have started a jide session and loaded a `Spectrum2d` dataset with name 'data' from a local pool or a database.

We might want to work only with a sub-set of the spectra included in our data. For a `Spectrum2d` this means we have to (1) select specific rows from the data and (2) combine them into a new dataset by applying some arithmetic operations on the selection. Task (1) is performed with the `SelectSpectrum` task,

```
from herschel.ia.toolbox.spectrum import SelectSpectrum
```

The `SelectSpectrum`-task can be configured and used in many different ways. A frequent usage is to identify all the rows of the dataset that have a specific value in a particular column:

```
ds1 = SelectSpectrum()(ds=data, selection_lookup={"bctype": [3260]})
```

The example above selects all the rows with a value=3260 in the column named 'bctype'. Hence, the selection is performed by using the keyword `selection_lookup` in the call of the task, using what is called a *python dictionary*. This py-dictionary contains the name of the attribute to look up as key (column name) and the attribute value as value. All the rows in the resulting dataset `ds1` have values 3514 in the `bctype` column.

Using py-dictionaries suggests that we may combine several selections by adding further lookup properties to the dictionary. Indeed, all the rows in the dataset resulting from

```
ds1 = select(ds=data, selection_lookup={"bctype": [3260], "buffer": [1]})
```

```
ds2 = select(ds=data, selection_lookup={"bctype": [3260], "buffer": [2]})
```

have values 3260 in the `bctype` column and values 1 in the `buffer` column (hence `ds2` is a subset of `ds1`). Note that the lookup values are specified as py-lists. By specifying a list of admissible values those spectra are selected that match one of values found in the list. As will be explained below, there are other selection models better suited for floating point values.

5.12.2.1. More on selection methods

- Lookup specific attribute value(s):
For one (or several) discrete criteria use the keyword `selection_lookup`:

```
ds1 = select(ds=data, selection_lookup={"bctype": [3413]})
```

Spectra with `bctype=3413` are selected and included in the result container.

```
ds2 = select(ds=data, selection_lookup={"bctype": [3412, 3413]})
```

Spectra with `bctype=3412` or `bctype=3413` are selected and included in the result container.

```
ds3 = select(ds=data, selection_lookup={"bbtype":[3413],"buffer":[1]})
```

Spectra with bbtype=3413 and buffer=1 are selected and included in the result container.

- **Index selection:**
If you want to select specific spectra included in the container by its index, use the keyword `selection_index`:

```
ds1 = select(ds=data, selection_index=[1,5,12])
```

The spectra with indices 1, 5, 12 are selected and included in the result container.

- **More general selection model:**
Use the keyword `selection` and use one of the selection models found in the package

```
herschel.ia.toolbox.spectrum.selections.models
chopperSelection = RangesSelectionModel("Chopper", [-4.4, 5.9], 0.1)
```

The first parameter specifies the name of the attribute, the second parameter gives an array of centers of the ranges and with the third parameter you specify the radius of the ranges to be considered. In summary, this ranges selection model will identify all spectra for which the attribute "Chopper" has values located within a distance $r = 0.1$ around one of the centers $[z1=-4.4, z2=5.9]$.

```
ds4 = select(ds=data, selection=chopperSelection)
```

For further selection models see further down in the documentation.

5.12.3. Toolbox Primer: Average Spectra

After selecting the data, we can move to task (2), the application of some arithmetic operations to the selected spectra. For example, if we now want to average the selection, we can invoke the `AverageSpectrum` task:

```
from herschel.ia.toolbox.spectrum import AverageSpectrum
avg21 = AverageSpectrum()(ds=ds2)
```

The selection explained in task (1) can also be included in the average spectrum task, thus allowing to perform selection and averaging in one step:

```
avg22 = AverageSpectrum()(ds=data, selection_lookup={"bbtype":[3260],"buffer":[2]})
```

This result is identical to the separate operations. It includes a single row with the average flux. The resulting dataset contains exactly the same columns as the input dataset. Thus, what values should we fill in the columns not affected by the operation? This is determined by a default action that depends on the input data type (sub-class of `Spectrum2d` in our example). For the `Spectrum2d`, the default action consists of copying the values found in the input spectrum.

This way of processing the data is general: we always try to keep as much information as possible. All columns and also the meta data are set in a type specific, instrument specific, or user specific way. The output data type is the same as the input data type.

The toolbox operations are not restricted to operations on `Spectrum2d` as our example may suggest. In all the operations in the `herschel.ia.toolbox.spectrum` no reference is made to `Spectrum2d`. The operations only refer to a specific contract (a java-interface), the `SpectrumContainer`-interface. `Spectrum2d` also fulfills this contract. All the datastructures that obey this contract can be processed by the arithmetic tools. The efforts to have this contract implemented for other data types is relatively small.

5.12.4. Toolbox Primer: Subtract Spectra

Other arithmetic operations are available such as pair operations (subtract, divide, pair-wise add/multiply) and scalar operations (add/subtract or multiply/divide by a scalar quantity). Here is an example that shows how to use the subtraction:

```
from herchel.ia.toolbox.spectrum import SubtractSpectrum
diff12 = SubtractSpectrum()(ds1=ds1, ds2=ds2)
```

Here, the datasets ds1 and ds2 either must have the same number of rows, or one of them must have only a single row. If they have the same number of rows, the subtraction is carried through for the flux data on a row-by-row basis. If the second contains only one row, this row is subtracted from all the rows in the first dataset (or the other way around).

The same task can also be used for subtracting a scalar:

```
ds_m2= SubtractSpectrum()(ds=data, param=2.0)
```

Here the number two is subtracted from all the flux columns in our data.

5.12.5. Toolbox Primer: Divide Spectra

The use of the DivideSpectrum -task is identical:

```
from herchel.ia.toolbox.spectrum import DivideSpectrum
ratio12 = DivideSpectrum()(ds1=ds1, ds2=ds2)
ds_d2 = DivideSpectrum()(ds=data,param=2)
```

5.12.6. Toolbox Primer: Add and Mulipty Spectra

Similarly, for multiplication and addition we can import tasks that can be used in a similar fashion.

```
from herchel.ia.toolbox.spectrum import MultiplySpectrum
from herchel.ia.toolbox.spectrum import AddSpectrum
```

These tasks work in exactly the same way.

5.12.7. Toolbox Primer: Resample and Smooth Spectra

Additional tasks included in the toolbox include smoothing, frequency resampling or extracting/cutting the spectra. The system again provides the instance

```
from herchel.ia.toolbox.spectrum import ReamplingFrequency
resample = ReamplingFrequency()
```

which allows for resampling non-equidistant grids to linear grids and the other way around. Resampling to a linear grid with given resolution (width) would look like

```
data_resampled = resample(ds=data, density=true, resolution=1.0)
```

where the resolution is given in the same units as the frequencies in the data. The density parameter indicates whether the flux is specified as a per channel (true) or as a per frequency unit quantity (false).

For the smoothing, the instance

```
from herschel.ia.toolbox.spectrum import SmoothSpectrum
smooth = SmoothSpectrum()
```

is again loaded automatically by the system and it can be used by

```
data_smoothed = smooth(ds=data, filter="box", width=10)
```

5.12.8. Toolbox Primer: Statistics on Spectra

Finally, the toolbox also allows to compute the statistics for the spectra included in a spectrum container.

```
from herschel.ia.toolbox.spectrum import SpectrumStatistics
statistics = SpectrumStatistics()
```

There are two alternative ways to compute the statistics for the spectra included in a spectrum container, the statistics computed on a per channel basis over all the spectra included in the container, or the statistics computed for each spectrum included in the container across the channels, possibly restricted to a range.

```
stats = statistics(ds=data)
```

The result of this operation stats is a product which contains the per channel statistics in `Spectrum1d` and the across channel statistics in a suitable `TableDataset`.

5.12.9. Summary of Toolbox Operations

Operations are available both at the task level and at the java level. The tasks are most suited for being used from the command line. The java classes which are wrapped by the tasks might be more helpful when developers want to integrate the functionality into other modules. The java classes will be discussed in the developer's sections.

- *SelectSpectrum (use select)*: Select spectra from a container and create a new spectrum container of the same runtime type.
- *AverageSpectrum (use avg)*: Average the spectra included in the container on a channel by channel basis. Restrict the average to specific selections or define groups and apply the average on a per group basis.
- *AddSpectrum (use add)*: Pairwise or scalar add.
- *SubtractSpectrum (use subtract)*: Pairwise or scalar subtract.
- *DivideSpectrum (use divide)*: Pairwise or scalar divide.
- *MultiplySpectrum (use multiply)*: Pairwise or scalar multiply.
- *ResampleFrequency (use resample)*: Resample each spectrum included in the container to a new, not necessarily linear grid.
- *SmoothSpectrum (use smooth)*: Smooth each spectrum included in the container.
- *ExtractFreqRanges (use extract)*: Cut the spectra included in the container to given frequency intervals.

- *ReplaceFreqRanges (use replace)*: Replace spectrum information in one container by information from another.
- *SpectrumStatistics (use statistics)*: Compute statistics of the spectra in the container - either on a per channel basis or across the channels.

5.12.9.1. Remarks

1. Fitting: There is a separate documentation on fitting: see the module ...
2. Datastructures: As indicate in the primer, all the data structures that fulfill the contract a spectrum container must have can be processed by the toolbox modules. Currently:
 - Spectrum1d: implements contract.
 - Spectrum2d: implements contract.
 - Cubes: under consideration.
 - Other instrument-specific data structures (such as HifiTimelineProduct or SpectrometerDetectorSpectrum): under consideration.

Chapter 6. DP Plot: Basic Plotting of Data



Important

This chapter is about the "new" plotting package (`herschel.ia.gui.plot`) that is automatically loaded by default in a DP; an "old" plotting package (`herschel.ia.plot`) is no longer available.

6.1. Introduction

This chapter describes how to do basic 2D plots in DP. It is primarily conceived as a step-by-step guide to support you while getting familiar with the visualisation of two-dimensional data. In addition, it is being enlarged with the final aim of documenting the complete set of functionalities of the `PlotXY` package. Not all the available commands have been introduced yet; for a complete list please refer to the related API documentation for the `herschel.ia.gui.plot` package.

Four main classes are described in this chapter: the `PlotXY` class, which is the representation of a two-dimensional plot, and its related classes `Axis`, `LayerXY` and `Annotation` which represent the different building blocks from which the plot is constructed. We will also cover some features of `Style`, handling the style of a plot (e.g. type, size and colour of plot symbols).

Pages containing more than a single plot component are created by placement of plot "layers" (created by the `LayerXY` class).

The following image shows the place of four of these classes within the general plot architecture, using as an example a page of four plots (the yellow rectangles).

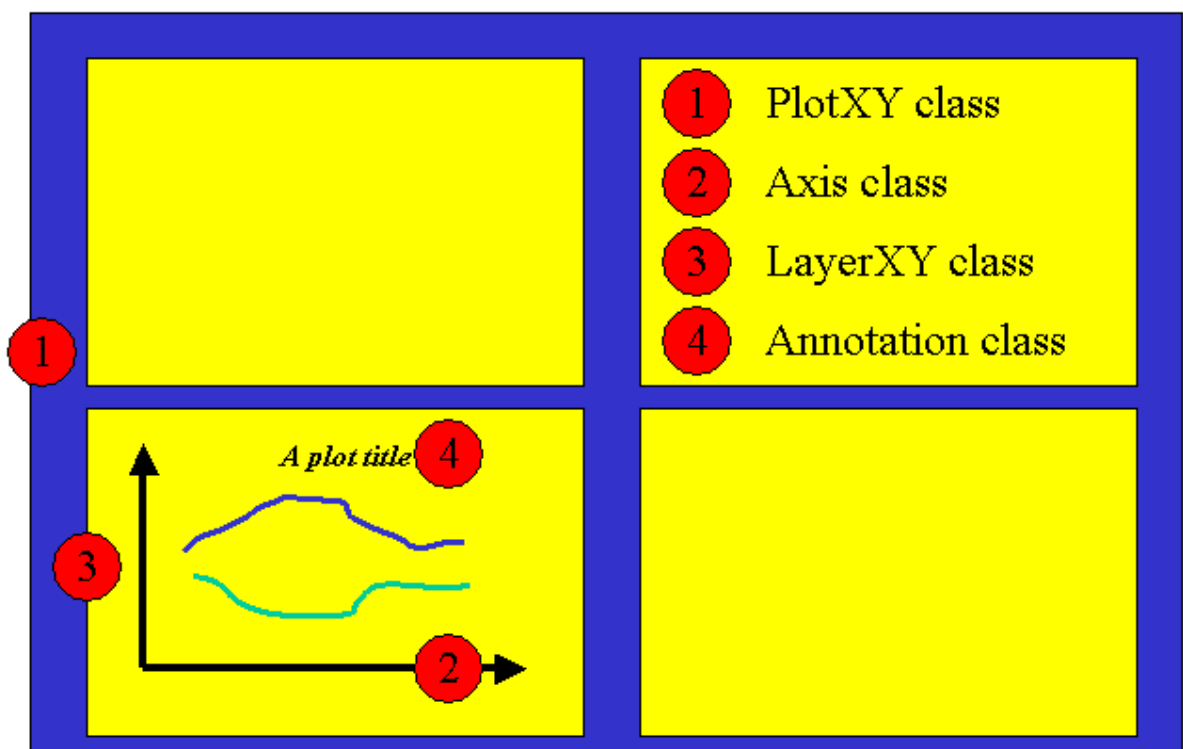


Figure 6.1. Classes involved in plot operations.

Depending on how you work with plots, either writing scripts or designing your plots interactively, we recommend different approaches. For writing scripts you need to use the command line interface.

This way the plot is completely defined by written commands. If you design your plots interactively it will be easier to use the graphical interface to manipulate plot properties which allows for button and pulldown menu selection of plot properties such as fonts, labels, line types and colours.

6.2. What do I need to make a simple XY plot?

The 2D plotting package currently works on Numeric1d data which is a one-dimensional array of numbers of any type (Int1d, Float1d or Double1d). Two numeric arrays are input, one as x-data and the other as y-data.

6.2.1. Introducing PlotXY

The class used for 2D plotting is called PlotXY. This produces a plot whose properties can be changed via command line input or through a properties GUI. Multiple plots can be added in "layers" to an initial base plot and the default scales for a given plot will automatically adjust to allow all points in all layers of a plot to be visible, although the x and y ranges for a plot can also be set by the user.

6.2.1.1. Using PlotXY to plot one Numeric1d array against another

Plotting numeric 1D arrays against each other can be done in a simple call such as

```
a = Double1d([1,2,3,4,5])
b = Double1d([0.3,0.8,1.5,2.3,2.0])
PlotXY(a, b, titleText="A plot")
```

Where a and b are two numeric 1D arrays and we give it a title ("A plot"). If we want a way of labelling the plot so we can do something to it later, we can do the following

```
p=PlotXY(a, b, titleText="A plot")
p.title.text="Better title for plot"
```

Here we have given the plot a label, p, and put a new title on it with the second line.

PlotXY has a number of other variables that can be set when initiating a plot. In the above examples we get no labels on the axes, a default line style and colour is used and the window size is a default setting.

The following example, Example 6.1, illustrates some key points in the use of PlotXY for plotting 1D arrays against each other.

```

n = DoubleId.range(20) / 10. # ❶
e = EXP(n) # ❷
plot = PlotXY(n, e) # ❸
p = PlotXY(layers=[LayerXY(n, e)], titleText = "Plot example", width=600, \
height=400, style=Style(line=Style.MARKED, symbol=Style.TRIANGLE, \
color=java.awt.Color.green), visible=Boolean.TRUE) # ❹
layer0 = p.getLayer(0) # ❺
layer0.style=Style(line=Style.MARK_DASHED, symbol=Style.CIRCLE, \
color=java.awt.Color.red) # ❻
layer0.style=Style(line=Style.NONE, symbol=Style.FSQUARE, \
color=java.awt.Color.red, symbolSize=7) # ❼
plot.close() # ❽

```

- ❶ n is set up to be an array with the range of numbers = 0...19 divided by 10. Placing the `DoubleId` element in front turns the integers created by the range command into a numeric array of doubles. So we have an array of 20 numbers going from 0 to 1.9
- ❷ e is an array which contains the exponent of all the n array elements
- ❸ this line will make a "default" plot of the exponent. It also identifies the plot window with the variable `plot`.
- ❹ here we define more plot variables in a single line call. After creating a layer explicitly (it was done automatically in the previous `PlotXY` call) we set the plot height and width, define some properties of the line style and set the plot as visible (you might wish to set a plot as invisible e.g. when you only want to print it or save it to file, without displaying it on screen).
- ❺ here we get the first layer (the only one in this case) identified by its index 0 inside the plot `p`. A second layer would have index 1, and so on.
- ❻ here we change the layer style: dashed line and red circles.
- ❼ another change to the layer style, plus a change to the size of the symbols to 7 point.
- ❽ closes the plot window

Example 6.1. A simple `PlotXY` example

The result of running this example is shown in the figure below.

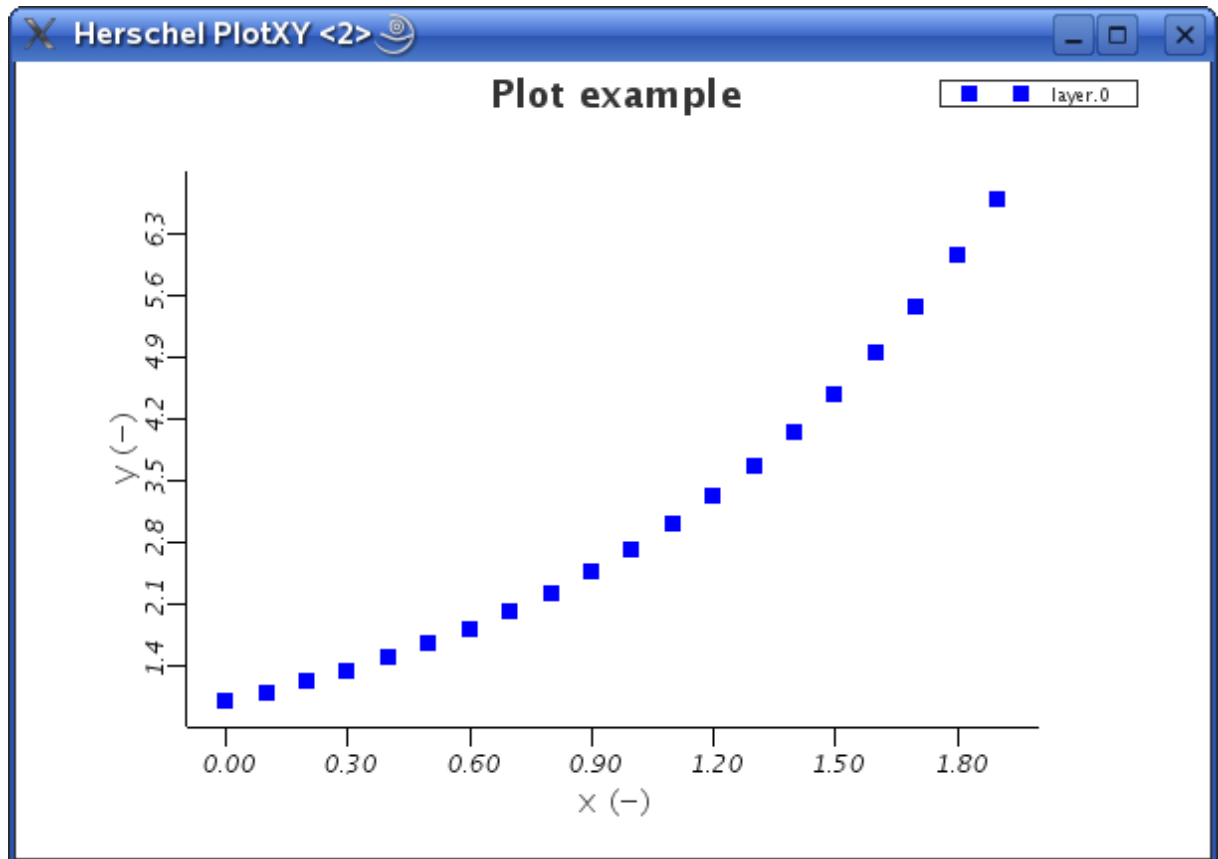


Figure 6.2. A simple plot of an exponential array.

6.3. How to setup your PlotXY properties

Plot properties allow the definition of items such as *colours*, *linetypes* etc. with your personal preferences. To setup your personal properties try the following:

- Construct a plot object `p` in JIDE,

```
p = PlotXY(n, e, width=600, height=400)
```

`n` and `e` being the `Double1d` arrays defined in Example Example 6.1.

- Open the plot properties window with the following command

```
p.props()
```

- Define your properties in this window and save them as default. The description how to save properties as default is given in Section 6.3.1.

or

- Open the Property Generator (command **propgen**) and select the Plot tab. Here you can change all the properties related to the plot interface and set them to default by pressing the Set to default button.



Note

Properties for PlotXY are saved under the user's home directory in `.hcss/user.props`. The HCSS properties path needs to have this file in it so that plot properties are restored

correctly in the next session. To use the saved properties immediately, right-click on the plot and go to "Reload Default Properties" on the menu.

6.3.1. How to modify properties

Properties can be manipulated with a graphical interface.

Do the following:

- construct a plot object with any constructor, for example

```
p=PlotXY(n, e)
```

n and e being the `Double[]` arrays defined in Example Example 6.1

- type the command

```
p.props()
```

Now the graphical interface for manipulation of the plot properties appears (see Figure 6.3). It consists of a tree-like structure on the left with all the objects composing the plot (like layers and axes). The properties of the highlighted object appear in the right panel.

The buttons at the bottom have the following functions:

Apply	applies any changes to the plot, without closing the properties window.
Refresh	reads in the properties of the visible register card (plot, layer or axis). This button is useful if you have the plot property GUI visible and change properties from the command line. Refresh updates the GUI afterwards.
Save as default	saves the properties as default and thus updates the <code>PlotXY.props</code> file in the <code>~/ .hcss</code> directory in the file <code>user.props</code> .



Warning

The global variable `HCSS_PROPS` should include this file for the default properties to be written and reused.

Note that if you set a property for a layer or an axis as default, the property set will be used for all layers and axis and not only for the one you have chosen in the moment of pressing the button.

6.3.2. Plot properties

The plot properties available for a "PlotXY" object are shown in Figure 6.3. There are four sections.

Plot	This allows the size of the plot window to be determined (in terms of physical size or pixels).
Title	The plot title can be typed in here and the result will appear at one of seven positions available in the pulldown menu (left, right or centre at either top or bottom or customised positioning). The title appears after the Apply window button is clicked. Note that a mouse click on the title will allow click-and-drag of the title to any position on the plot. The font type and size can be customised using the Change... button below the title box in the properties window.
Subtitle	Subtitles work in a similar way to titles except that the default positioning is below the title and with a smaller font. Again, the subtitle can be dragged to anywhere on the plot surface and font changed.

- Boxed Plot** If this is ticked, then the plot is a box (otherwise only the left and bottom axes are plotted). This is applied when the initial plot -- base layer -- is created.
- Legend** The checkbox indicates whether a legend is shown or not, while the pulldown menu provides eight different positions at which the legend can be placed. Again, the legend position can be changed by a simple click-and-drag.

All changes are applied by clicking the Apply button.

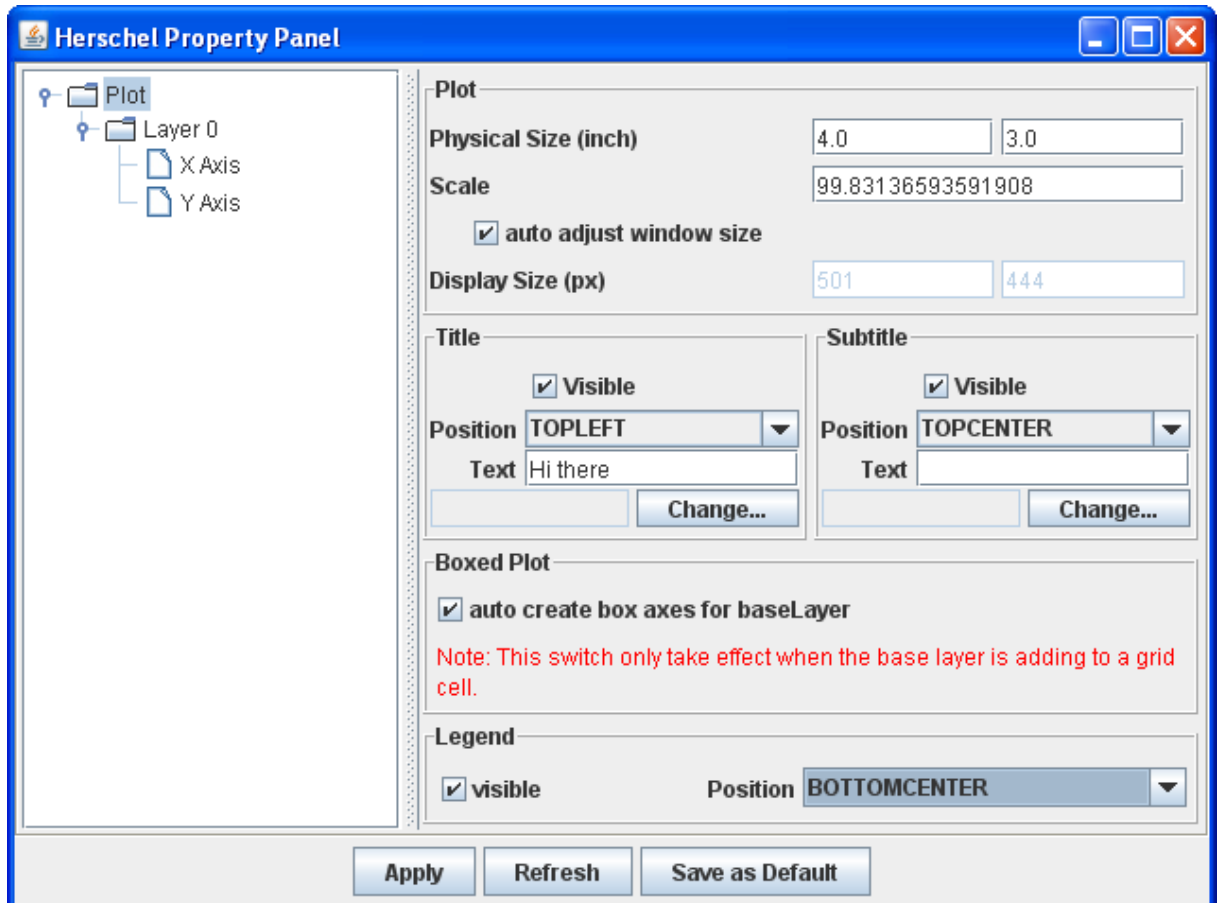


Figure 6.3. The Plot section of the PlotXY properties dialog.

6.3.3. Layer properties

The layer properties are used to define default layer properties or to manipulate the properties of already constructed layers. This includes the layer name and style properties. In order to work on a given layer, the user needs to click on the appropriate layer on the left hand side of the properties panel. This brings up the layer properties dialog. See Figure 6.4).

The layer id number is automatically assigned, in numerical order starting from zero. Layers added to the same plot are numbered from 1 upwards. Applying a new name will update the name given in the legend of the plot for the layer.

The *Style* properties are applied to a particular layer of a plot. Here is where we can change the colour and form of a plot.

- Chart Type** The pulldown allows for either a LINECHART or a HISTOGRAM plot.
- Symbol** The symbol type to be used for points on a plot can be chosen from 25 possibilities in the pulldown menu. The symbol type number is also given (SQUARE = "8").

Color	The colour can be changed by clicking on the coloured square and choosing from the colour menu in the popup window.
Size	Provides a scaling for the symbol size (in font points) used for plotting points on a scatter plot.
Stroke	Provides a scaling for the width of lines used for line plots.
Line Style	Provides the options of no line (NONE), a solid line (SOLID), a line with each point marked (MARKED), a dashed line (DASHED) or a dashed line plot with points marked (MARK_DASHED).
Dash Array	The two values that are typed in here indicate the size of the dashes and the distance between dashes. If a dashed plot is requested.

The layer itself can be removed using the Remove button.

Finally, an annotation to the plot can be made using the Add Annotation button. This brings up the an annotations properties window (see Figure 6.5).

Annotation	The actual annotation and font type can be selected here.
Position	Placement in the plot area (x and y) and the angle (in an anti-clockwise direction) at which the annotation is displayed.
Alignment	Indicates where relative to the position that the annotation is to be made. Essentially, above it, below it or centred on it (vertical) and to left, to right or centred on it (horizontal).

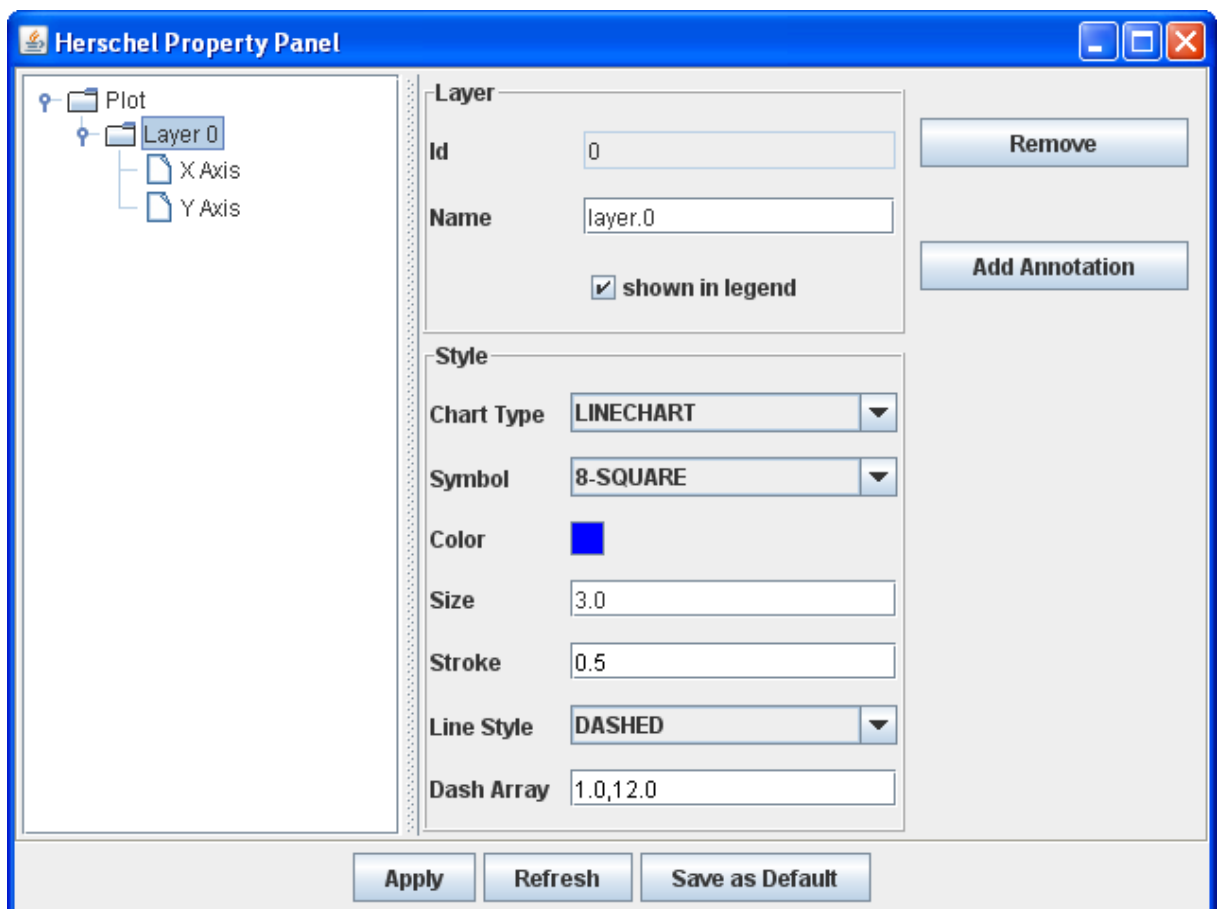


Figure 6.4. The Layer section of the PlotXY properties dialog.

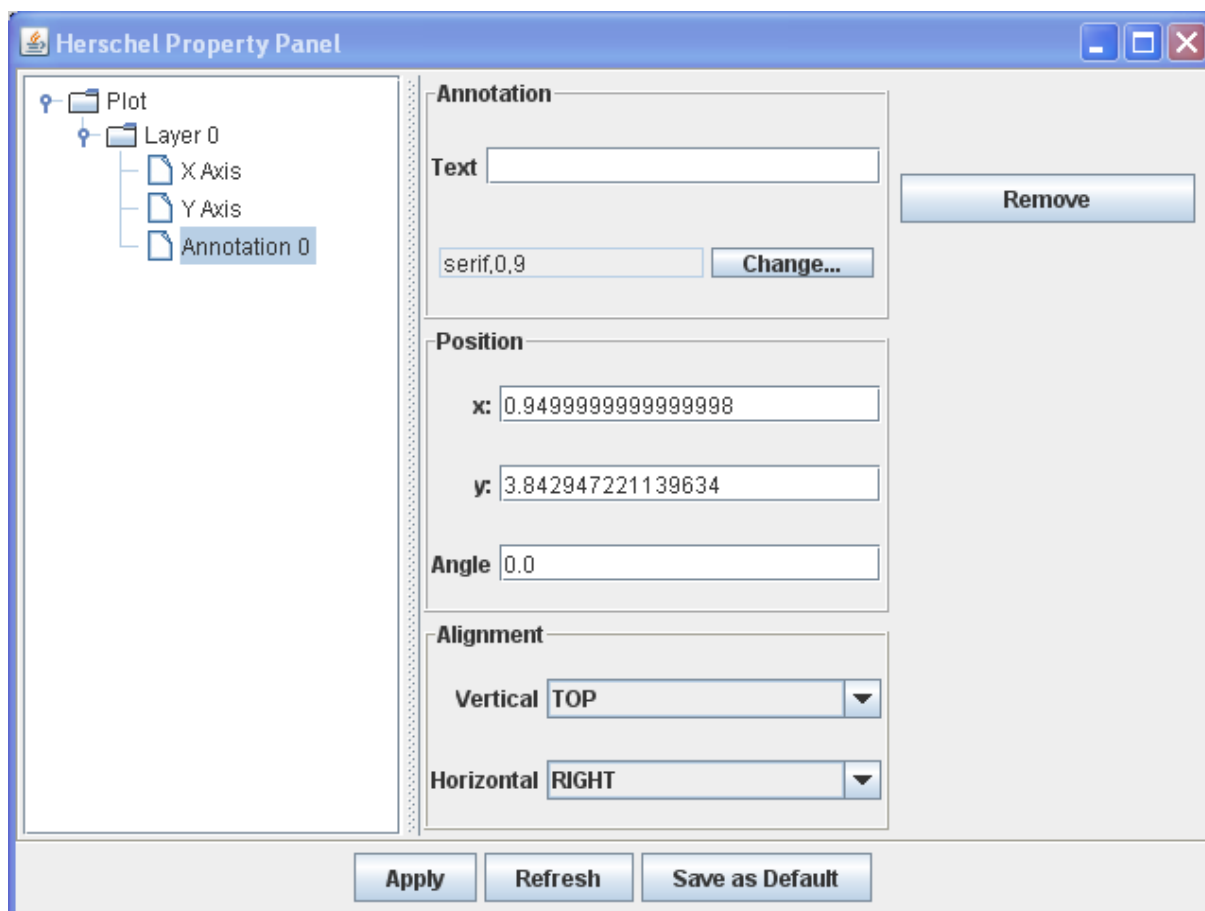


Figure 6.5. Dialog for adding an annotation to a Layer.

6.3.4. Axis properties.

The `Axis` properties dialog (see Figure 6.6) is used in the same way as for the layer properties. In order to work on a given axis the appropriate "X-axis" or "Y-axis" label in the left column display of the properties window (as in Figure 6.6). This then brings up the `Axis` properties dialog.

There are two elements that can be changed in this dialog:

Axis The user has options for where the axis is, on top/bottom (the `POSITION` pulldown menu), left/right; whether it is linear or log; whether it is inverted or even invisible. Colour of the axis can be selected by clicking on the coloured box (black is the default) and choosing from the colour selection popup.

The range can be set or left to be generated automatically.

The title/label for the axis can chosen to be displayed either side of the axis and the font type and size is selectable by clicking the "Change..." button.

Ticks The tick position is with reference to the axis. Choices are for either side of the axis, crossing the axis (`MIDDLE`) or having no tick marks.

Grid lines for each axis can be chosen individually.

The tick mark intervals can be chosen or done automatically. The size of major and minor tick marks can be typed in and the number of minor tick marks per major tick mark interval also typed in (0 means there are no minor interval tick marks). Tick labels can be vertical or horizontal on either axis. The number of decimal places for label values can also be explicitly given (e.g., "%.2f" gives values to 2 decimal places) or left be calculated automatically.

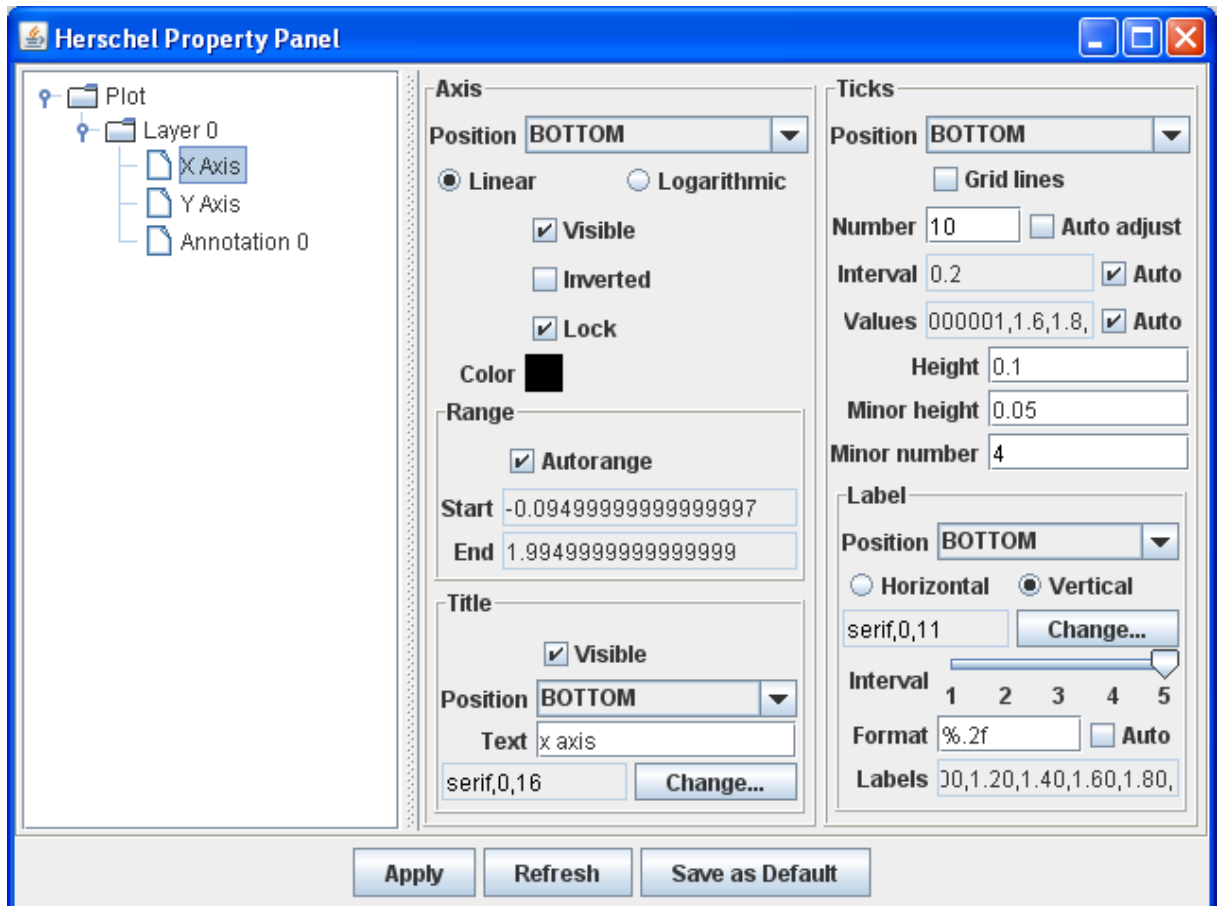


Figure 6.6. The Axis section of the PlotXY properties dialog.

6.3.5. How to use properties.

The result of a property setup procedure (with a defined set of properties) is given in Example 6.2 which follows on from Example 6.1. This can be used to set up properties from the command line window of JIDE or for generating plots from within scripts.

```
p.props() # ❶
p[0] = None # ❷
p[0] = LayerXY(n, n*n, name="anotherLayer") # ❸
p[0].style.stroke = 5 # ❹
p[1] = LayerXY(n, 2*n*n, name="yetAnotherLayer") # ❺
p[1].style.stroke = 7 # ❻
```

- ❶ this command allows graphical interface property setup, it fires the Plot Property GUI.
- ❷ removes the first (and only) layer of the plot. Press the Refresh button in the Properties window to see the change
- ❸ overlays on the graph a plot of n versus n -squared and calls it "anotherLayer". `p[0]` can be used to refer to this layer, like you would do with an element of an array.
- ❹ sets the line stroke for overlay plot anotherLayer
- ❺ adds yet another layer to the plot "p"...
- ❻ ...and changes the line stroke on this plot too!

Example 6.2. Command line control of properties

The result of running above example is shown below.

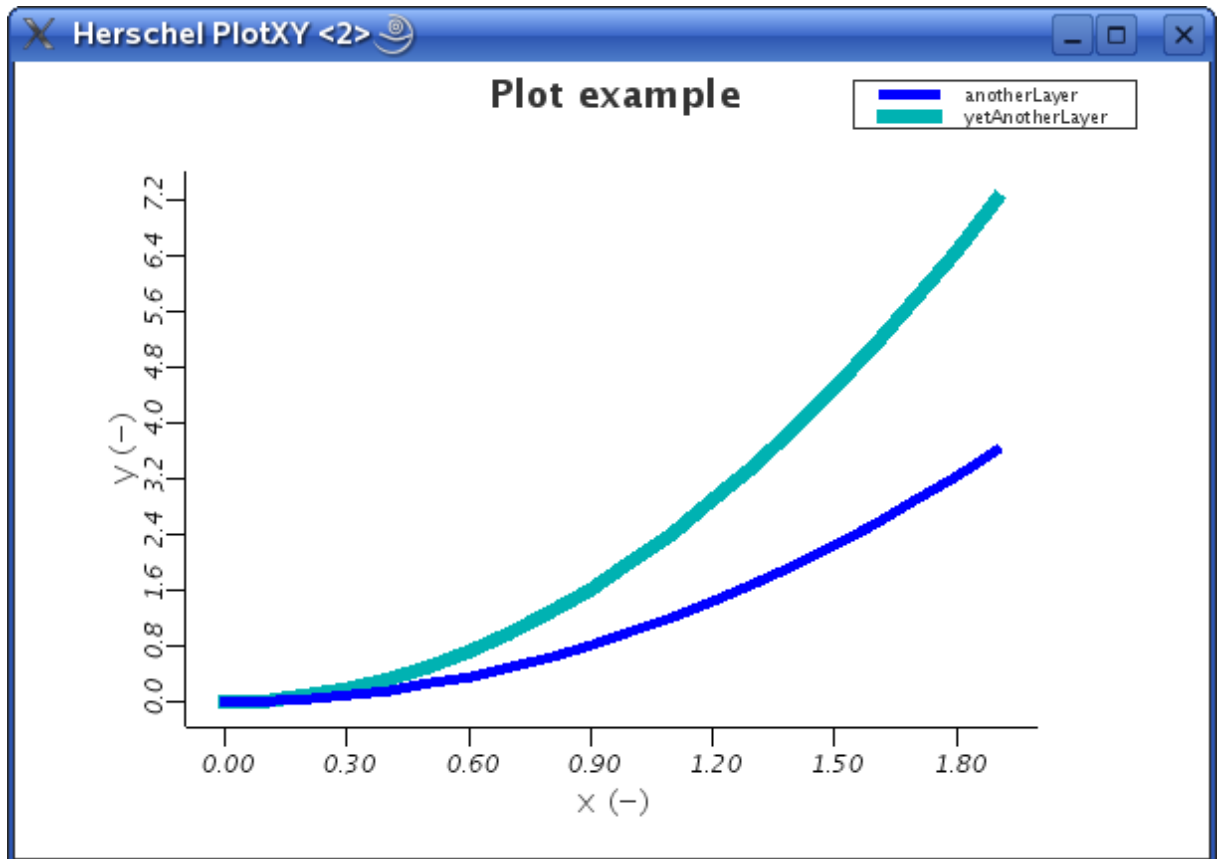


Figure 6.7. This plot is the result of Example 6.2.

Note that if a new layer is added without defining either colour or line type, the current set of default properties are used.

If colour and line type are specified in the constructor, they are used as specified.

```
p[2] = LayerXY(n, 8*n*n, name="moreLayers", symbol = Style.TRIANGLE, \
  color = java.awt.Color(250,100,0))
```



Note

the backslash (\) symbol provides continuation of the command onto the next line and should be immediately followed by a CARRIAGE RETURN.

The result of the above command line is shown below. In this case we have also illustrated how you can **create your own colour through a mixture of red, green and blue hues** (values up to 256). In this case, the result is an orange colour for our third plot layer.

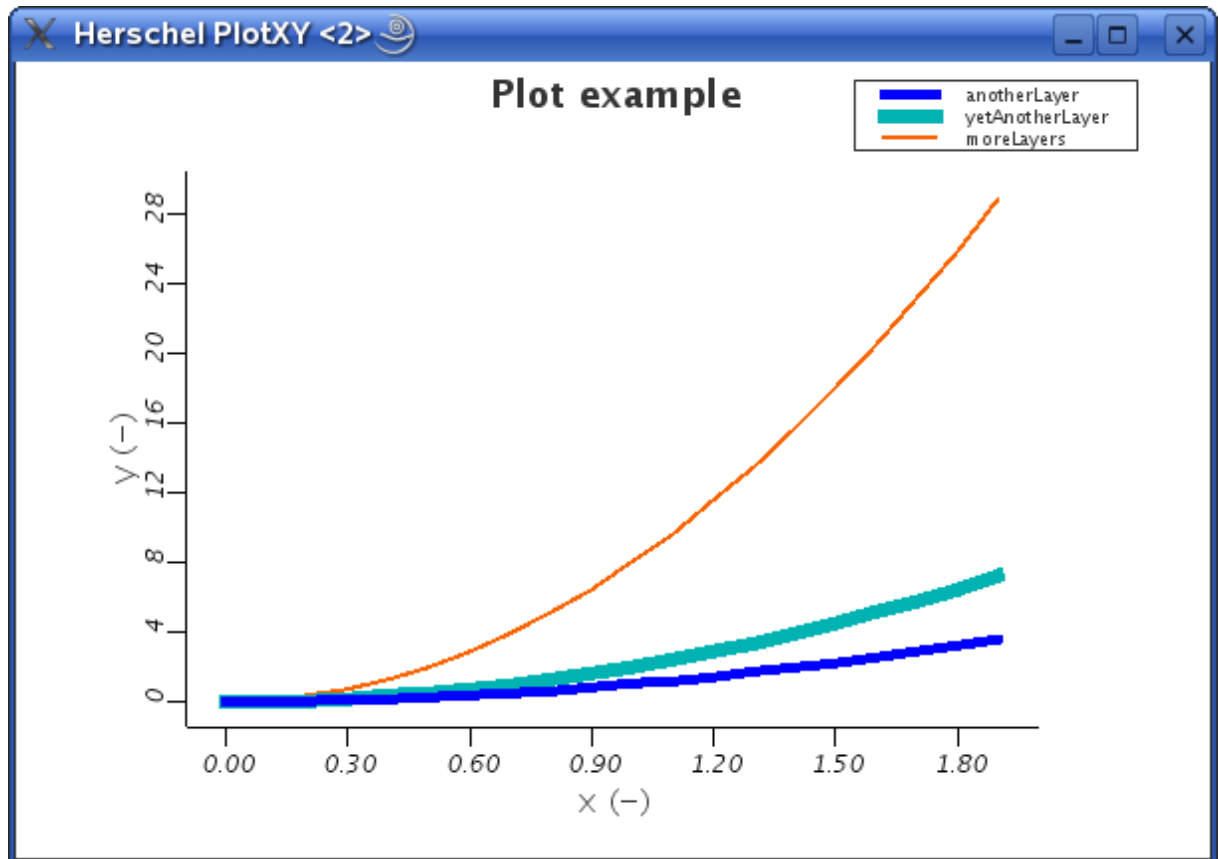


Figure 6.8. Adding in another layer gives the orange curve (see text).

6.4. Manipulating Layers, Axes, and Annotations in DP Scripts

In this section we show how to manipulate plots from the command line. Such manipulations can be placed in scripts to make plots appear the way the user requires.

In DP scripts it is necessary to access all the properties from the command line (either bottom left of JIDE for interactive work or in the upper pane of JIDE when doing script development).

There is one general rule to do so.

1. get the object:

```
layer = p.getLayer(layer index) or axis = layer.getXaxis()
```

2. use the methods provided by the object:

```
layer.setColor(color)
```

color is a java.awt.Color, e.g., java.awt.Color.red

6.4.1. What about these Layers?

Any plot is built up from layers. Even a simple 2D plot as we've created above has one layer that contains the data from the two one-dimensional arrays we have used to build it. If you need to plot multiple sets of data you add one layer for each additional set.

As stated before the manipulation that you need to do on layers should be done through the layer object. One such command is the `setColor(color)` that we have used above.

Let's create a simple plot again with two layers and do some basic manipulations on the individual layers. Example 6.3 plots two curves, one is the analytical function \exp , the other curve has added noise.

In the first three lines we generate some noise on top of the exponential function.

```
r = RandomUniform() # ❶
rn = DoubleId(20).apply(r) - 0.5 # ❷

n = DoubleId.range(20)/10
e = EXP(n) # ❸
en = e+rn # ❹

p = PlotXY(layers=[LayerXY(n, e, name="e", color=java.awt.Color.red)], \
  titleText="Exponential plot", width=600, height=400) # ❺
p[0].setStyle(Style(line = Style.NONE, symbol = Style.FSQUARE, symbolSize = 3.5, \
  color = java.awt.Color.blue))

p[1] = LayerXY(n, en, name="en") # ❻

layer_en = p.getLayer(1) # ❼
layer_en.setLine(Style.NONE)
layer_en.setSymbol(Style.FCIRCLE)
layer_en.setColor(java.awt.Color.red)

layer_en.setLine(1) # ❽
```

- ❶ DP utility to produce random numbers between 0 and 1.
- ❷ generates a set of 20 random double (real) numbers between -0.5 and 0.5.
- ❸ The array `e` was defined in a previous example, but lets recreate it...`e` is an array of 20 numbers which are $e^{0.5}$, $e^{1.0}$, $e^{1.5}$ etc.
- ❹ adds the random numbers to the array `e` i.e. add noise to the data.
- ❺ Plot the array `e`, give the layer a name and in the following line change some of the layer's properties to make it a scatter plot.
- ❻ Add the noise data to the plot as a layer with name `en`
- ❼ In these four lines it is demonstrated how to make this layer a scattered layer with red circles as symbols. Code 0 means "no line", while 14 is "filled circle".
- ❽ reset the layer back to a line plot. Note how setting the line to "solid" (code 1) the symbols automatically disappear.

Example 6.3. Working with layers from the command line.



Note

Please do not take the above as an example of the proper way to add noise to a function, the 'noise' here is just to illustrate the layer concept.

Some of the more useful methods that work on layers are listed in the tables below. Please read carefully the following note in order to interpret the tables correctly.



Note

In order to save space we do not explicitly list all the available methods, as the Javadoc does, but adopt the shortcuts described below.

- 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, you guessed it, are used to set a value. For every setter there is usually a *getter*, a method whose name

begins with " get " and whose work is to retrieve a value. The tables only list setters, adding *Get method available* when the corresponding getter exists. 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`.

- This is not a shortcut but is worth mentioning anyway. The name of a method can offer useful clues about its behaviour. For example, the method `setSomething` will *replace* the preexisting `Something`, while `appendSomething` will *add* `SomethingElse` to the existing `Something`.

Table 6.1. Methods for handling Annotations in layers.

<code>addAnnotation(Annotation annotation)</code>	Adds an <code>Annotation</code> object to the layer.
<code>addAnnotations(Annotation[] annotations)</code>	Adds several <code>Annotation</code> objects to the layer. The input <code>Annotations</code> are passed as an array.
<code>setAnnotation(int id, Annotation annotation)</code>	Sets an annotation to a given <code>id</code> , replacing what was there before.
<code>setAnnotations(Annotation[] annotations)</code>	Replaces all the annotations with the ones provided in the array.
<code>getAnnotation(int i)</code>	Retrieves one annotation from the layer.
<code>getAnnotations()</code>	Retrieves all the annotations from the layer. The annotations are returned as an array.
<code>removeAnnotation(int id)</code>	Removes the annotation with the specified <code>id</code> .
<code>clearAnnotations()</code>	Removes all the annotations.

Table 6.2. Methods for handling error bars in layers.

<code>appendErrorX(double low, double high)</code>	Appends a low and high error value of <code>x</code> .
<code>appendErrorX(Ordered1dData low, Ordered1dData high)</code>	Appends a set of low and high error values of <code>x</code> .
<code>setErrorX(Ordered1dData[] error)</code>	Sets low and high error values of <code>x</code> .
<code>setErrorX(Ordered1dData low, Ordered1dData high)</code>	Sets the low and high error values of <code>x</code> .
<code>getErrorX()</code>	Returns an array of <code>Ordered1dData</code> with length equal to 2.

Table 6.3. Axis-related methods of the Layer class. All can equally be applied to the y-axis by replacing "X" with "Y".

<code>setXaxis(Axis axis)</code>	Sets the x axis to the specified <code>Axis</code> instance. Note: the x axis will be reinstated with its default settings plus whatever is indicated in the <code>Axis</code> instance. So any prior manipulations of the axis are lost.
<code>setXrange(double[] range)</code>	Sets the range of the x axis. Get method available.
<code>setXtitle(String title)</code>	Sets the title of the x axis. Get method available.
<code>setXtype(Axis.Type type)</code>	Sets the type of the x axis based on the axis types available. <code>LINEAR</code> is type 0, <code>LOG</code> is type 1. Get method available.
<code>setXy(Ordered1dData[] xy)</code>	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 <code>setYx</code> method!
<code>setXy(Ordered1dData x, Ordered1dData y)</code>	Sets the x and y values, passed as two separate arrays. Note there is no <code>setYx</code> method!
<code>setY(Ordered1dData y)</code>	Sets the ordinate values. Get method available. Note there is a <code>getX</code> method but not a <code>setX</code> method.
<code>shareXaxis(Axis axis)</code>	Removes the x axis and uses the given axis as a shared one.

Table 6.4. Miscellaneous setters of the Layer class.

<code>setName(text)</code>	Changes the name (and thus the legend) of the layer. Get method available.
<code>setLine(line code)</code>	Changes the plot to a line plot for the specified layer. Get method available.
<code>setSymbol(symbol code)</code>	Changes the plot to a scatter plot for the specified layer. Get method available.
<code>setSymbolSize(int size)</code>	Sets the size of a the symbol. Get method available (note that it returns a <code>double</code> rather than an <code>int</code>).
<code>setSymbolShape(SymbolShape shape)</code>	Sets the shape of the symbol. The input parameter is an instance of the class <code>SymbolShape</code> . Get method available.
<code>setColor(colour)</code>	Sets the colour of the symbols and lines for the specified layer. Get method available.
<code>setStroke(stroke)</code>	Sets the stroke of the line for the specified layer (only for line plots). Get method available.
<code>setStyle(Style style)</code>	Sets the style of the layer. The input parameter is an instance of the <code>Style</code> class. Get method available.

Table 6.5. Other methods of the Layer class.

<code>addPoint(double x, double y)</code>	Adds a point to the layer.
<code>addPoint(Ordered1dData x, Ordered1dData y)</code>	Adds a set of points to the layer.
<code>getCoords()</code>	Waits for mouse click and returns the coordinates of the pointer. Returns a <code>double[]</code> .
<code>getCoords(int n)</code>	Like the previous method, but this one does the job for <code>n</code> successive clicks. Returns a <code>double[][]</code> .
<code>getDataCoords()</code>	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 <code>double[]</code> .
<code>getDataCoords(int n)</code>	Like the previous method, but this one does the job for <code>n</code> successive clicks. Returns a <code>double[][]</code> .
<code>getId()</code>	Returns an <code>int</code> representing the index of the current layer inside the <code>PlotXY</code> .
<code>setInLegend(boolean)</code>	True if the layer is shown in the legend.
<code>isInLegend()</code>	Returns True if the layer is shown in the legend.
<code>setNotifyWarningAsExceptional(boolean)</code>	True if exceptional values like NaN and infinity are notified as errors, False if they are only logged.
<code>isNotifyWarningAsExceptional()</code>	Returns True if exceptional values like NaN and infinity are notified as errors, False if they are only logged.

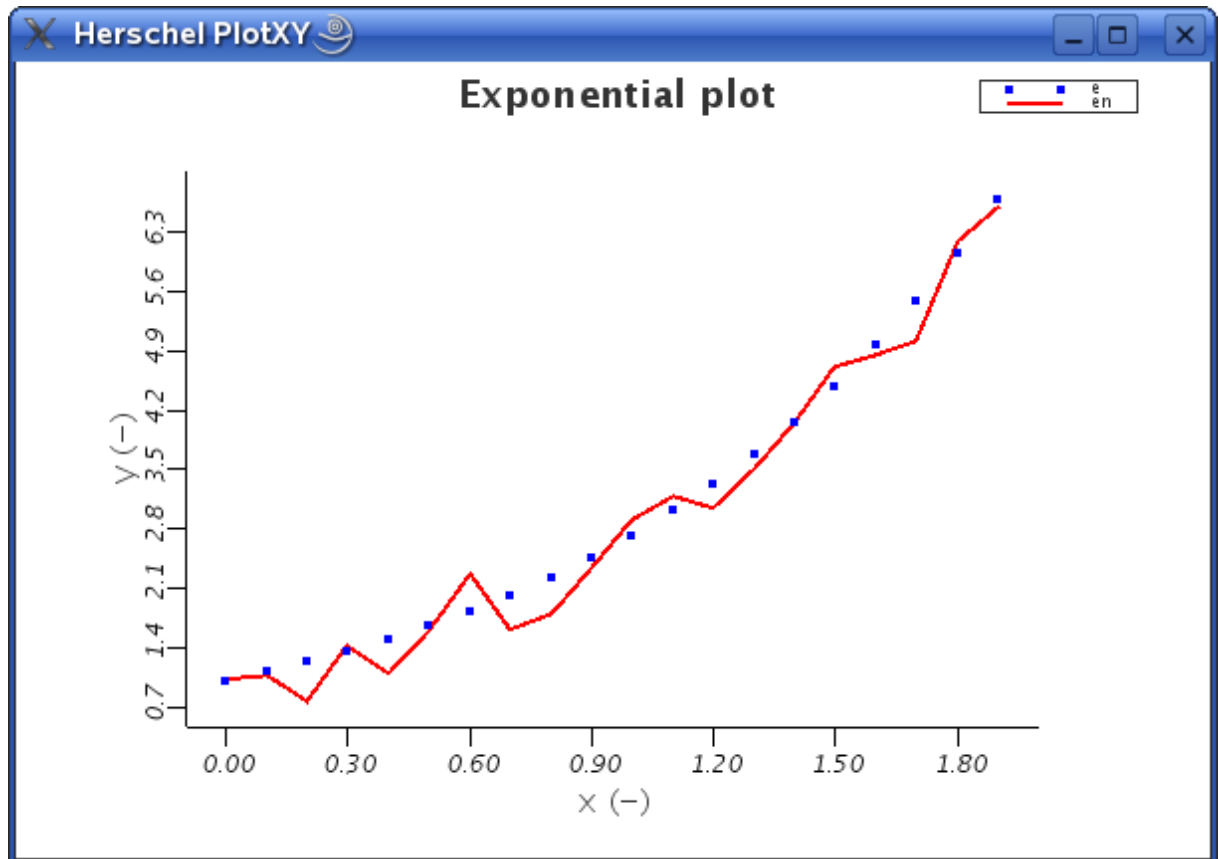


Figure 6.9. Plot showing the result of manipulation of layers from the command line.

The `LayerXY` class provides a much larger number of methods to specify the appearance of data points in layers. Next to simple line and scatter plots, lines and symbols can be combined and symbols can be circles, rectangles, triangles, squares etc. which can be filled or not with a specified colour. Lines can be solid or dashed with their own colour. Find the possible predefined symbols in the `Style` class and access them for example by `line = Style.SOLID`.

We are not going into detail for all these methods but you should try them out with the API documentation for `LayerXY` lying next to you.

6.4.2. What can I do with Axis?

As with `Layers` most manipulations of both X and Y axes can be done through the `Axis` class.

6.4.2.1. Log Axes, Labels and Gridlines

Let's continue with our previous example and make some changes to the axes illustrating how we can adjust labels, grid lines and change axes to a logarithmic scale.

```

# Set up our overlay plot again
r = RandomUniform() #
rn = Double1d(20).apply(r) - 0.5
n = Double1d.range(20)/10
e = EXP(n) #
en = e+rn
p = PlotXY(layers=[LayerXY(n, e, name="e", color=java.awt.Color.red)], \
  titleText="Exponential plot", width=600, height=400)
p[0].setStyle(Style(line = Style.NONE, symbol = Style.FSQUARE, symbolSize = 3.5, \
  color = java.awt.Color.blue))
p[1] = LayerXY(n, en, name="en")
# The y axis is a bit cluttered, but a couple of commands will tidy up the mess
# First of all we change the format of the tick labels...
p.yaxis.tick.label.format="%3.1f"
# ...then we display a label every two ticks
p.yaxis.tick.label.interval=2
# Now we change the axis label
p.yaxis.title.text="log(exp(x/10))"
# This shows the y axis gridlines, TRUE = 1
p.yaxis.tick.gridLines=1
# Change x axis label
p.xaxis.title.text="index"
# ...and finally we adjust the range of y values that we
# want the plot to have.
p.yaxis.setRange([0.5, 10])

```

Example 6.4. Axes, labels and grid lines

It is also possible to use TEX-like labelling for subscripts and superscripts. For example:

```
p.xaxis.title.text="$x_1^{2a}$"
```

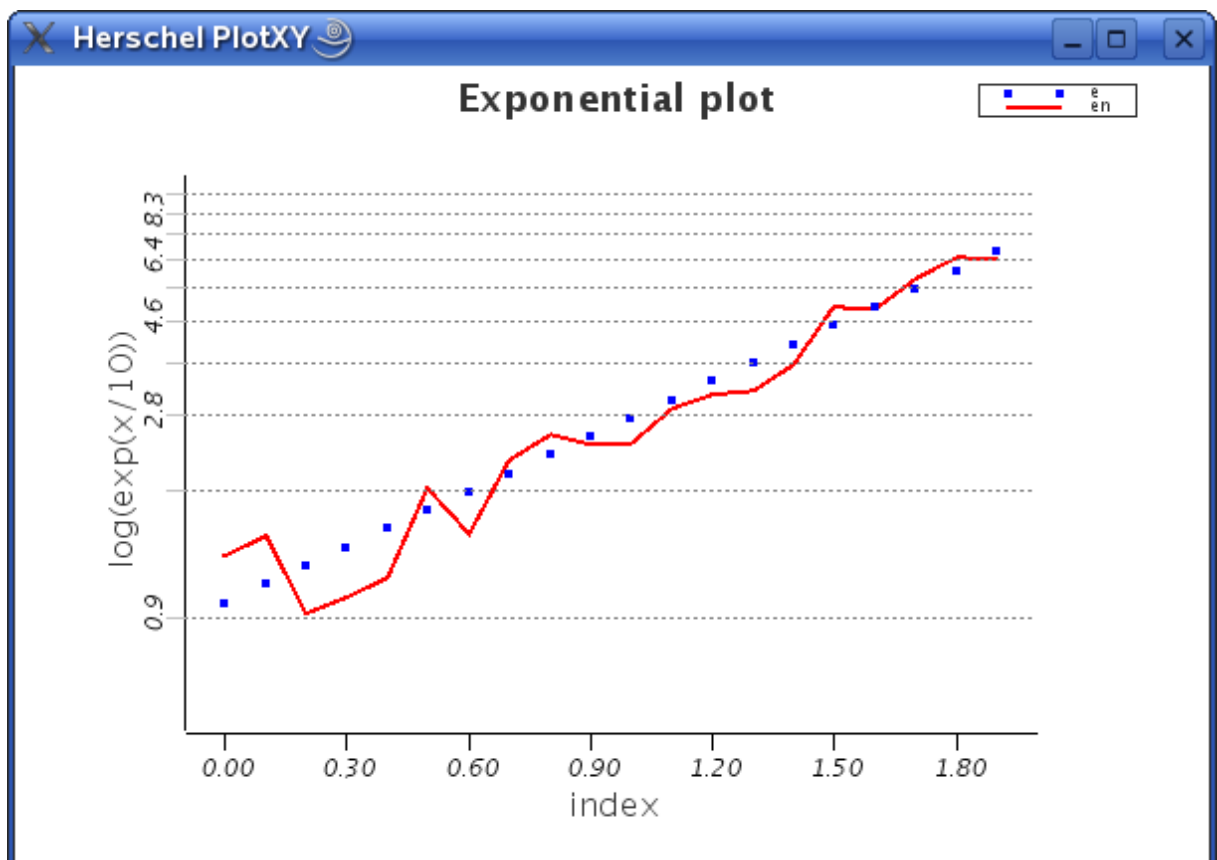


Figure 6.10. Changing Axes, labels and added grid lines.

6.4.2.2. Multiple Axis Labels

Each layer can have at most two axes (the first layer of a plot has two axes by default). If we have more than one layer in the plot, we can add and visualise new axes. This is illustrated in the following example.

```
# Set up our overlay plot again
r = RandomUniform() #
rn = DoubleIcd(20).apply(r) - 0.5
n = DoubleIcd.range(20)/10
e = EXP(n) #
en = e+rn
p = PlotXY(layers=[LayerXY(n, e, name="e", color=java.awt.Color.red)], \
  titleText="Exponential plot", width=600, height=400)
p[0].setStyle(Style(line = Style.NONE, symbol = Style.FSQUARE, symbolSize = 3.5, \
  color = java.awt.Color.blue))
p[1] = LayerXY(n, en, name="en")
# Get the layer we want to change
layer=p.getLayer(1)
# Add a new x axis
layer.setXaxis(Axis())
### NOTE: when using Axis() to create a new axis or recreate an axis the default
### axis scaling/range values are taken and overwrite any axis manipulations
### that may have been done before.
# Release the lock on the new x axis
layer.xaxis.setLock(0)
# Restrict the range of the plot to x values between 0.5 and 1.5
layer.xaxis.setRange([0.5, 1.5])
# Add a label to this new axis
layer.xaxis.title.text="New X axis"
# Update the en layer so that it is half the value it was
# before and replot
layer.setXy(n, en/2)
# Now put the plot in a situation where the new y axis value range
# is automatically calculated.
layer.xaxis.setAutoRange(1)
```

Example 6.5. Putting multiple axes on the same plot.



Note

If after the second instruction (`layer.setXaxis(Axis())`) you get the error `TypeError: no public constructors for herschel.ia.image.Axis` it means that JIDE thinks you are referring to the `Axis` class in the image rather than the plot package. Issuing the command from `herschel.ia.gui.plot import *` should fix the problem.

The result of running this example is shown in Figure 6.11.

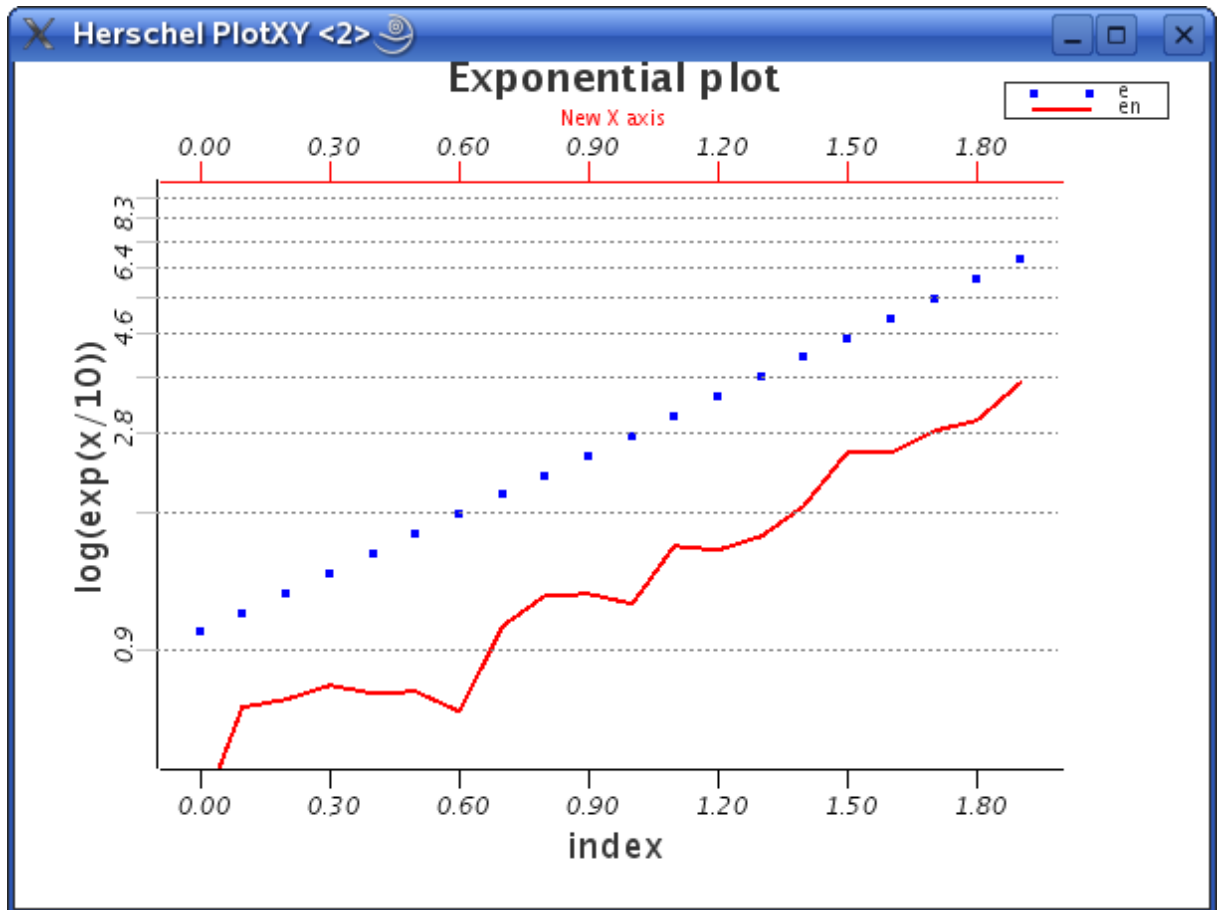


Figure 6.11. Example of a second X-axis label relevant to the red line plot.

Some of the more useful methods that work on axes are listed in the tables below. For a complete reference of the methods that can be used to manipulate and tune the appearance of the axes please consult the API documentation of `Axis`.

Table 6.6. Useful ways of manipulating axes from the command line

<code>axis = layer.getXaxis() or getYaxis()</code>	Gets the X or Y Axis object to do direct manipulations on the corresponding axis
<code>setAutoRange(flag)</code>	If <code>flag</code> is true, adjusts the range of the specified axis so that all datapoints will be shown
<code>setRange([lower, upper])</code>	Set the range of the specified axis to values between <code>lower</code> and <code>upper</code> . Note that we no longer have two arguments for the lower and upper limits, but one array argument containing both values.
<code>setGridlines(flag)</code>	Show grid lines for the specified axis if <code>flag</code> is true, hide the grid lines if <code>flag</code> is false.

Table 6.7. Methods for handling labels on axes.

<code>getTick().getLabel().setColor(java.awt. colour)</code>	<code>Color</code> Sets the colour of labels. Get method available.
<code>getTick().getLabel().setFont(java.awt. font)</code>	<code>Font</code> Sets the font of labels. Get method available.
<code>getTick().getLabel().setFontSize(double size)</code>	Sets the physical size of labels. Get method available.
<code>getTick().getLabel().setInterval(int n)</code>	Sets the interval (in ticks) between successive labels. Get method available.
<code>getTick().getLabel().setOrientation(int n)</code>	Sets the orientation of the labels (0 for horizontal, 1 for vertical). Get method available.
<code>getTick().getLabel().setStrings(String[] labels)</code>	Replaces the current labels with the values in an array of <code>String</code> objects. Get method available.
<code>getTick().getLabel().setPosition(AxisConstants. position)</code>	Sets the position of the labels with respect to the axis. Possible values are TOP or BOTTOM for abscissa axis and LEFT or RIGHT for ordinate axis. Get method available.

Table 6.8. Methods for handling ticks on axes.

<code>getTick().setColor(java.awt.Color colour)</code>	Sets the colour of ticks. Get method available.
<code>getTick().setHeight(double size)</code>	Sets the physical height of the major ticks. Get method available.
<code>getTick().setInterval(double interval)</code>	Sets the interval (in axis units) between ticks. Get method available.
<code>getTick().setPosition(AxisConstants.Position position)</code>	Sets the position of the ticks with respect to the axis. Possible values are TOP or BOTTOM for the abscissa axis and LEFT or RIGHT for ordinate axis. Get method available.
<code>getTick().setNumber(int ticks)</code>	Sets the number of major ticks displayed on the axis. Get method available.
<code>getTick().setMinorNumber(int minors)</code>	Sets the number of minor ticks displayed between two major ticks. Get method available.
<code>getTick().setValues(Double[] values)</code>	Sets the values where ticks are to be placed. Get method available.
<code>getTick().setAutoAdjustNumber(boolean)</code>	True if the number of ticks on the axis is set automatically.
<code>getTick().isAutoAdjustNumber()</code>	Returns true if the number of ticks on the axis is set automatically.
<code>getTick().setAutoValues(boolean)</code>	True if the positions of the ticks on the axis are chosen automatically.
<code>getTick().isAutoValues()</code>	Returns true if the positions of the ticks on the axis are chosen automatically.

Table 6.9. Miscellaneous setters/getters of the Axis class.

<code>setType(Axis.Type type)</code>	Sets whether the axis is linear (0) or logarithmic (1). You can also use <code>Axis.LINEAR</code> and <code>Axis.LOG</code> as input parameters. Get method available.
<code>setLinear()</code>	Sets the axis to a linear scale. Equivalent to <code>setType(Axis.LINEAR)</code> .
<code>setLog()</code>	Sets the axis to a logarithmic scale. Equivalent to <code>setType(Axis.LOG)</code> .
<code>setColor(java.awt.Color colour)</code>	Sets the colour of the axis. Get method available.
<code>setAutoRange(boolean isAutoRange)</code>	Sets whether the range is automatically determined. Get method <code>isAutoRange</code> available.
<code>getTick().setGridLines(boolean)</code>	Sets whether grid lines are displayed. Get method <code>isGridLines</code> available.
<code>setInverted(boolean)</code>	Sets whether values on the axis are displayed in inverted order (e.g. right to left for abscissa). Get method <code>isInverted</code> available.
<code>setPosition(AxisConstants.Position position)</code>	Sets the position of the axis with respect to the plot. Possible values are <code>TOP</code> or <code>BOTTOM</code> for abscissa axis and <code>LEFT</code> or <code>RIGHT</code> for ordinate axis. Get method available.
<code>setRange(double[] range)</code>	Sets the range of the axis. The lower and upper limit are passed inside an array. Get method available.
<code>setRange(double low, double high)</code>	Sets the range of the axis. The lower and upper limit are passed as separate <code>double</code> parameters.
<code>getTitle().setPosition(AxisConstants.Position position)</code>	Sets the position of the axis title with respect to the axis. Possible values are <code>TOP</code> or <code>BOTTOM</code> for abscissa axis and <code>LEFT</code> or <code>RIGHT</code> for ordinate axis. Get method available.
<code>setVisible(boolean isVisible)</code>	Sets whether the axis is visible. Get method <code>isVisible</code> available.

It is also possible to set the Axis in one go using GUI plot' Axis class. An example of this is:

```
x = DoubleId.range(10)
y = x*x
plt = PlotXY()
plt[1] = LayerXY(x,y)
plt[1].xaxis = Axis(text="My x-axis")
```



Warning

Users should beware that use of the Axis class in this way will take a set of axis defaults, such as axis ranges. If instead of the last line above the following two lines are used in the given order

```
plt[1].xrange=[-1.0,15.0]
plt[1].xaxis = Axis(text="My x-axis")
```

The Axis command defaults will override the previously set plot axis range.

If only the axis label requires changing it is better to use the following

```
plt[1].xaxis.text = "New text"
```

6.5. Adding Error Bars to a Plot

Error bars can be added to any layer of a plot. In order to add errors to points in a layer we use the "setErrorX and "setErrorY" methods on a layer. For example:

```
layer.setErrorX(xerror_up, xerror_down)
```

and

```
layer.setErrorY(yerror_up, yerror_down)
```

Where "up" and "down" indicate the extent of the errors with increasing and decreasing values of x or y.

The following example indicates how we can apply error bars to the default, first layer of a plot.

```
x = 1.0 + Double1d.range(10) # create x and y data arrays
y = x+5.0
yerr = SQRT(x) # associate errors with them
xerr = SQRT(x)/x

p = PlotXY(x,y) # create the plot
p.style = Style(line=Style.MARKED,symbol=6,color=java.awt.Color.red) # set style
p.xaxis = Axis(titleText="x-axis (cm)",type=Axis.LOG) # make it a log-log plot
p.yaxis = Axis(titleText="y-axis (cm)",type=Axis.LOG)
p.xrange=[1.0,11.0] #set how large the plot will be in the x/y directions
p.yrange=[5.0,16.0]
p.setErrorY(yerr,yerr) #apply error bars
p.setErrorX(xerr,xerr)
p.getLegend().setVisible(0) # remove the legend
p.setTitleText("Error bar example plot") # give the plot a title
```

Example 6.6. Adding error bars to plots

The above example produces the plot shown in Figure 6.12.

It is also possible to access non default layers. For example, *carrying on from the previous example above* we could add a second layer and apply error bars to that too.

```
x2 = 3.0 + Double1d.range(10) # create new x and y values to plot
y2 = x+ 4.0
y2err = SQRT(x)/4 # create new error bars for plotting
x2err = SQRT(x)/(2*x)
p[1] = LayerXY(x2,y2)
p[1].style = Style(line=Style.MARKED,symbol=6,color=java.awt.Color.blue)
p[1].setErrorX(x2err,x2err) # apply different error bars
p[1].setErrorY(y2err,y2err)
```

The final plot is shown in Figure 6.13.

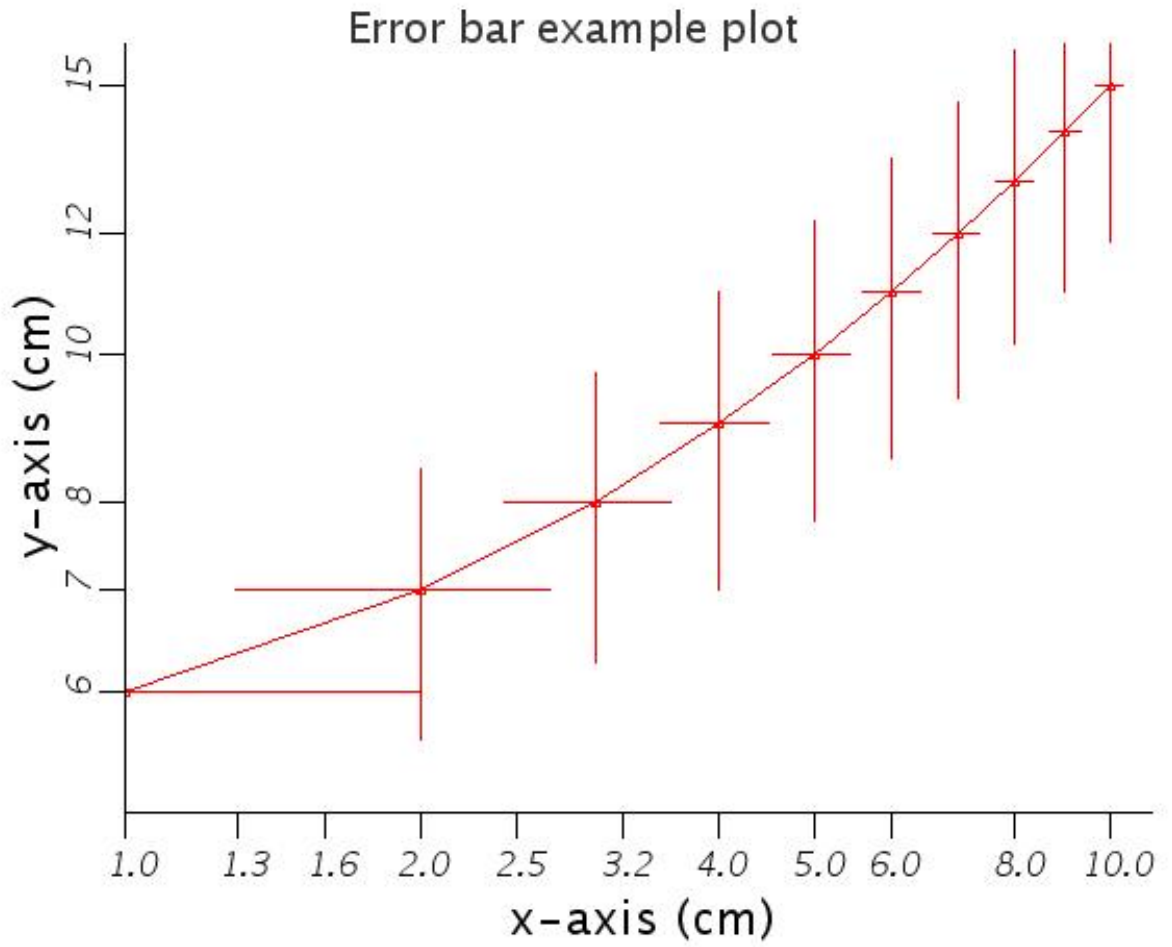


Figure 6.12. Setting errors in a plot

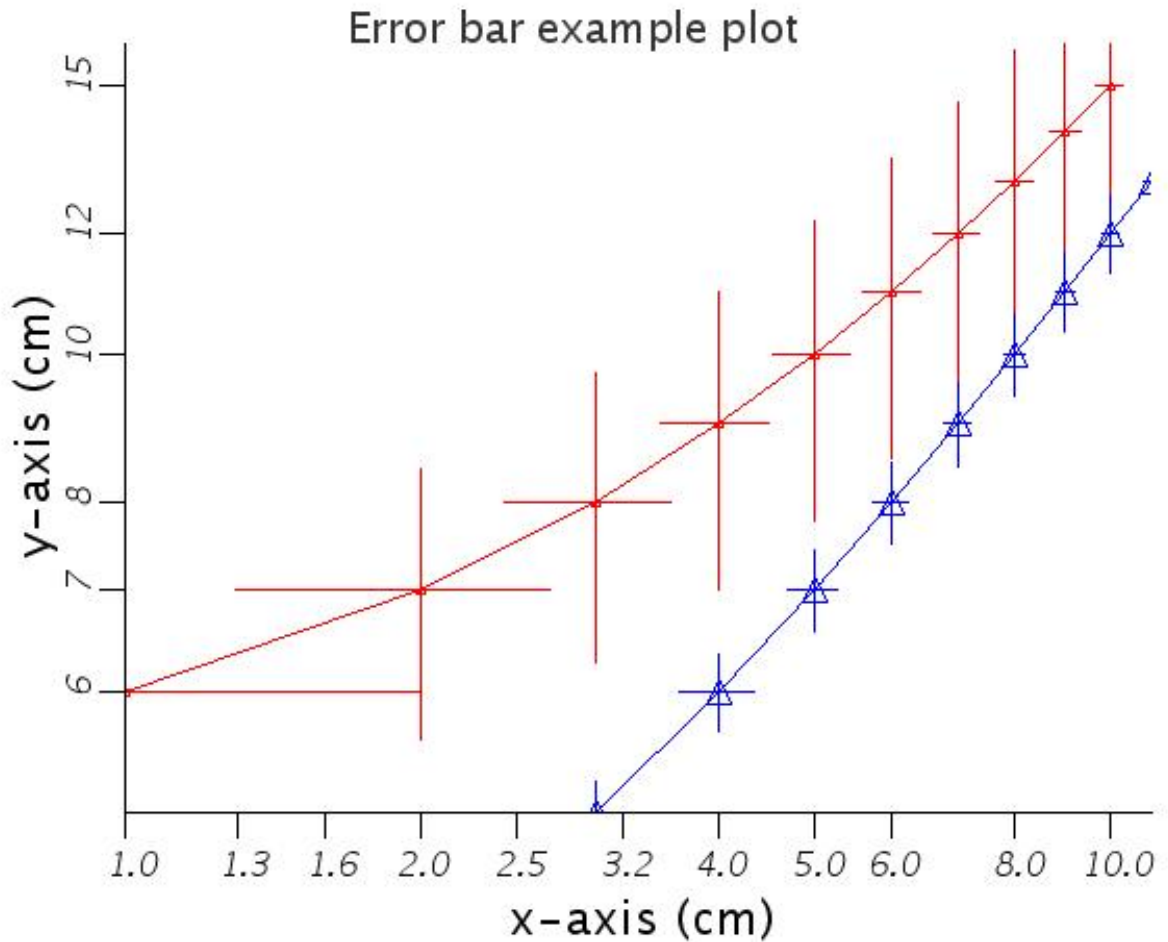


Figure 6.13. Applying errors to a specific layer of a plot

6.6. How can I annotate, decorate and save my plot?

There are quite a number of methods that we can use to make our plot more appealing and informative. A number of these methods were already mentioned in the sections on layers and axes, but we are going to put them into practice here. We continue with our example and add proper names for layers, annotate some datapoints and put a title on top of the figure (see Figure 6.14). The example below also shows how to extract the Layer objects from the plot in order to manipulate them directly.

```

# Set up our overlay plot again
r = RandomUniform() #
rn = DoubleIid(20).apply(r) - 0.5
n = DoubleIid.range(20)/10
e = EXP(n) #
en = e+rn
p = PlotXY(layers=[LayerXY(n, e, name="e", color=java.awt.Color.red)], \
  titleText="Exponential plot", width=600, height=400)
p[0].setStyle(Style(line = Style.NONE, symbol = Style.FSQUARE, symbolSize=3.5, \
  color = java.awt.Color.blue))
p[1] = LayerXY(n, en, name="en")
# Get the layer we want to change
layer = p.getLayer(1)
# Change the name (and the legend) for this layer to say what we want
layer.setName("exp+noise")
# Place some annotation at position 1, 2
layer[0] = Annotation(3, 6, "Noise on top of exp()", color=java.awt.Color.blue)
# Get the first layer of the plot...
layer = p.getLayer(0)
# ...and change its name
layer.setName("exp")
# Set a new style
layer.setStyle(Style(line = Style.MARKED, symbol = Style.FTRIANGLE, \
  color = java.awt.Color.green, symbolSize=7))
# Give the plot a title
p.title.text = "Example of a layered plot"
# Save it as a PNG file for importing as a picture into documents etc.
p.saveAsPNG("myPlot.png")
# Alternatively, save it as a JPEG file...
p.saveAsJPG("myPlot.jpg")
# ...or an EPS file
p.saveAsEPS("myPlot.eps")

```

Example 6.7. Decorating a plot.

Note that we changed the name of both layers in the second and fifth line of the script. Changing the name also changes the legend displayed on the plot.

For the `exp+noise` layer we put an annotation at a specific point (layer coordinates) in the plot. Please check the detailed package documentation of the `Annotation` class for methods to change the font, the size and other properties of an annotation.

For the `exp` layer we have changed the appearance of the datapoints to a line with triangles on top of it. Please refer to Section 6.4.1 for information on basic manipulation methods for layers.

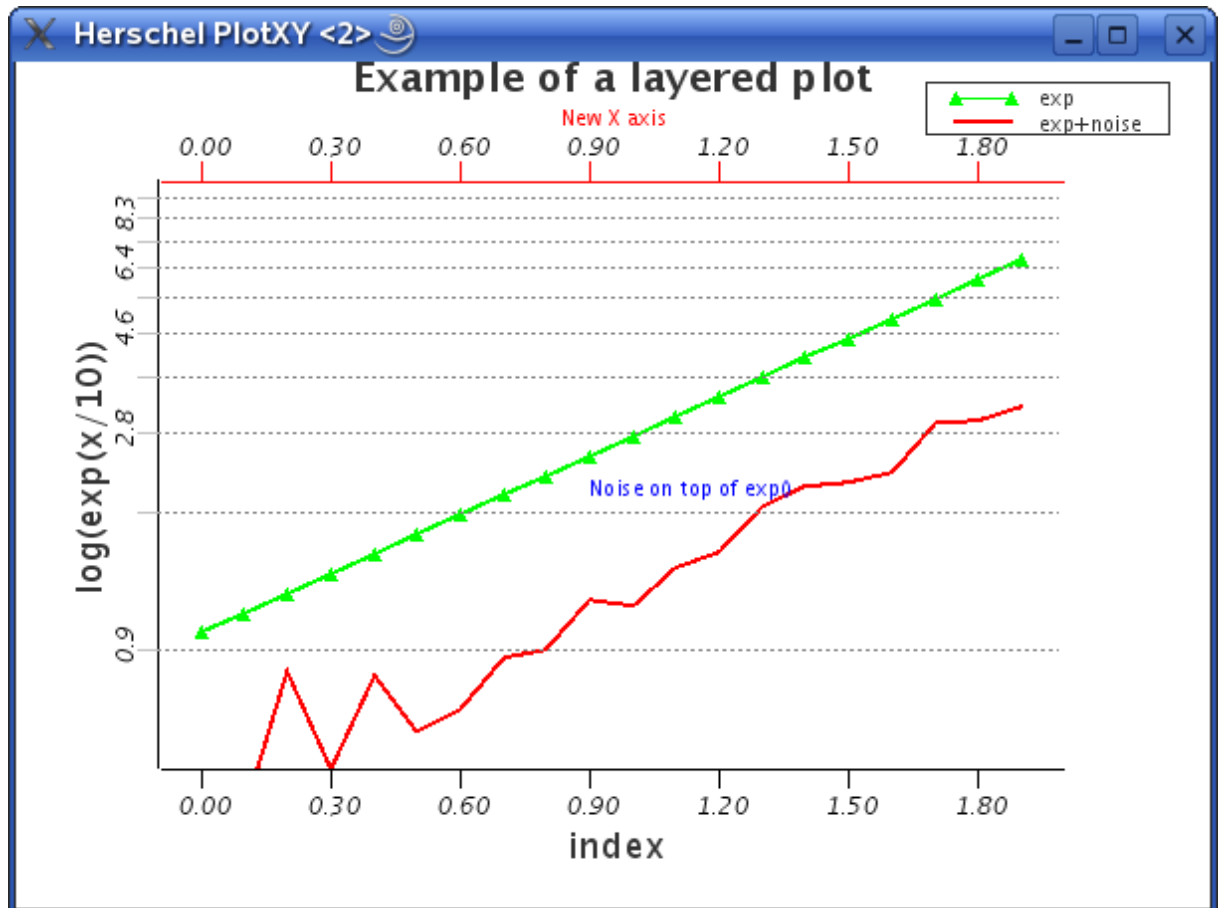


Figure 6.14. The plot has been annotated and decorated.

6.7. How can I make my plots more colourful?

Colours can be set for a number of parts within a plot. Methods can normally take a colour at creation time e.g. when adding a layer to the plot you can specify the colour to be used for its datapoints or for individual layers, labels etc. the colour can be specified with dedicated commands.

To specify a colour as an argument you have to pass a `java.awt.Color` object. The easiest way to do this is to use their default names as e.g. `java.awt.Color.blue`. If you don't want to write the `java.awt.` bit every time you will need to import the `awt` package as illustrated in the ????. Once imported colours can be changed as follows:

```
layer.setColor(Color.green)
```

The default names for colours are: black, blue, cyan, darkGray, gray, lightGray, green, magenta, orange, pink, red, white and yellow (all preceded by `Color.`). Another easy way to use a custom colour is to specify the red, green, blue value in ranges from 0 to 255: `Color(red, green, blue)`. So we could also do the following to get a similar green colour.

```
layer.setColor(Color(0,250,20))
```

6.8. Creating file output and printing a plot without displaying

Sometimes you do not want to plot to the screen, but would rather write your plots directly to files.

- We can generate a plot using the basic constructor (`p=PlotXY()`), setting it to invisible (`p.setVisible(0)`) which can later be filled by plot information such as x and y data. This works, but will cause window flashes on the computer screen. Better is to completely render the plot. The last value of "0" in the second form of the plot construction, below, indicates that the plot will not be made visible when it is created but can be made visible at a point of the user's choosing.

```
# Create an array with 100 doubles in it
data = Double1d(range(100))/10.0
# Hide an unfilled plot... but still showing the window!
p = PlotXY(visible=0)
# Hide a completed plot of data versus data squared. Causes window flashes
p2 = PlotXY(data, data.copy().power(2), titleText = "Title", width = 700, \
            height = 500, visible = 0)
```

Our plot can now be made visible using

```
# Now make the plot visible
p.setVisible(1)
```

- To save the plot directly to file you can then use the following two methods:

```
p.saveAsJPG("filename") # for a JPG file
p.saveAsPNG("/home/mypath/filename") # for a PNG file
p.saveAsEPS("filename") # for an EPS file
```

6.8.1. Using batch mode

Imagine you have written a script for drawing a plot made of several layers. Normally, when you execute the script, the plot will first be created and then redrawn each time a new layer is added. You may want the plot to be drawn just once with all the layers already in place, rather than being updated at each intermediate step. You can do that by invoking the `setBatch` method on your plot object. For example, here is a script snippet where the batch mode is turned on right after creating a plot:

```
# ...previous script commands...
myPlot = PlotXY()
myPlot.setBatch(True) # We could also write myPlot.setBatch(1)
# ...the script goes on...
```

After the last plot commands you may set the batch mode back to false with `myPlot.setBatch(False)` or `myPlot.setBatch(0)`, and all the layers will be drawn at once.

6.9. Windows containing more than one plot

More than one `PlotXY` plot can be placed within a single window using the `setLayer` method. Each layer that a user creates can be placed in a grid which is x units long by y units in height. The layer is given an integer identifier that indicates where in the grid it should be put.

```
plot.setLayer(int id, LayerXY layer, int gridx, int gridy)
```

Following this we can place previously created `PlotXY` components into each of the window positions. We indicate their position along the width (starting from 0) then the height (starting from 0). So we might place the 4 plots (`plot1`, `plot2`, `plot3`, `plot4`) into our composite window using code such as in Example 6.8.

```

# Create the data
data = DoubleId.range(100)/10.0
data2 = data.copy().power(2)
data3 = data.copy().power(3)
data4 = data.copy().power(4)
# Create individual plots to
# add to our composite plot
plot1 = LayerXY(data, data)
plot1.setName("linear")
plot1.setColor(java.awt.Color.red)
plot2 = LayerXY(data, data2)
plot2.setName("Square")
plot2.setColor(java.awt.Color.green)
plot3 = LayerXY(data, data3)
plot3.setName("Cubic")
plot3.setColor(java.awt.Color.blue)
plot4 = LayerXY(data, data4)
plot4.setName("4th power")
plot4.setColor(java.awt.Color.orange)
# start adding in the layers in grid
# positions 0,0 to 1,1
p = PlotXY()
p.setLayer(0,plot1,0,0)
p.setLayer(1,plot2,0,1)
p.setLayer(2,plot3,1,1)
p.setLayer(3,plot4,1,0)
# Let's change the colour of plot1
# we use it's id number '0'
p[0].setColor(java.awt.Color.black)
# We can also change other things such
# as the axis labels for just one plot
# within the grid.
p[0].xaxis.title.text = "Unit"
p[0].yaxis.title.text = "Linear"

```

Example 6.8. Multiple plotting

The above code produces the multiple plot window shown in Figure 6.15. Alternately, layers can simply be added to plots -- no id number is then required.

```

pp = PlotXY()
pp.addLayer(plot1,0,0)
pp.addLayer(plot2,0,1)
pp.addLayer(plot3,1,1)
pp.addLayer(plot4,1,0)

```

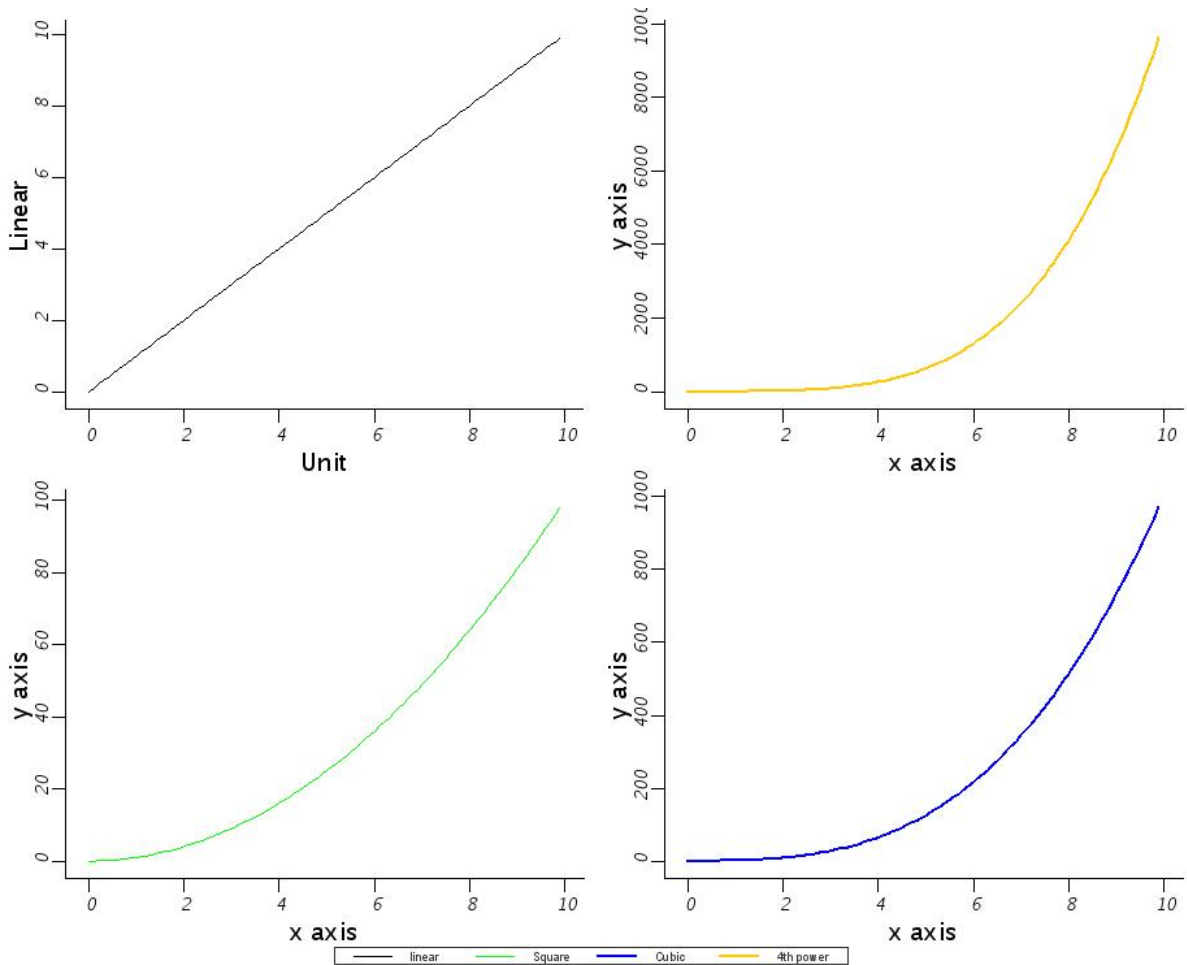


Figure 6.15. Example of multiple plots in a window from Example 6.8.

The properties of any one of the layers in the PlotXY window can be adjusted, e.g.,

```
p.props()
```

6.10. Mouse Interactions with Plots

We can get information from plots using a mouse command. Two basic mouse commands allow point values to be obtained from plots and nearest data points values to be found.

In order to find mouse coordinates within a given layer of a plot we can use the "getCoords" method. This allows multiple points to be obtained and stored in an array.

```
#Mouse Coordinates:
#get mouse coordinates from the first of our
#multiple plots (click on plot layer 3 times)
points=plot1.getCoords(3)      #
print points
```

This produces x and y coordinates in two arrays of doubles.

```
# x positions in a Double1d array
xarray = Double1d(points[0])
# y positions in a Double1d array
yarray = Double1d(points[1])
```

Similarly we can get nearest data points

```
#Data coordinates:
#get 5 Data points (click on plot layer 5 times)
dataxy=plot1.getDataCoords(5)      #
print dataxy
```

Once again, the output is in two arrays of x and y coordinates.

6.11. What about a complete PlotXY example?

You can find some demo scripts packed in a ZIP file at this address:

ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/ia/ia-8.3/plot_demo.zip.

Chapter 7. Display: Handling Images and Cubes with Herschel DP

7.1. Introduction

This chapter describes how to use `SimpleImage` and `SimpleCube` to store your image and cube data, display and do some basic operations on your images. It is not intended to elaborate on the complete set of functionalities available for image manipulation. For this the reader is referred to the API documentation for the image dataset package, API documentation for the image gui package and API documentation for the image toolbox package



Note

A number of classes that are used with the DP image display need to be imported before use or they will produce a strange `NameError` message. You can simply do this with the following statements in JIDE:

```
from java.awt import Font
from herschel.share.unit import *
```



Note

The `image` packages `herschel.ia.dataset.image`, `herschel.ia.dataset.image.wcs`, `herschel.ia.toolbox.image` and `herschel.ia.gui.image` are automatically loaded when starting the default version of DP. However, these may need to be imported by hand.

Throughout this chapter an example JPEG file of NGC 6992 will be used. This file is available in the `doc/ia/document/um/images` folder of your HCSS installation (click here for a local link: [ngc6992.jpg](#)).

7.2. Using SimpleImages and SimpleCubes

A `SimpleImage` is a special type product composed of :

- the *image*: described as a `Numeric2d` (2D number array, this can be a `Double2d`, `Float2d`, `Long2d`, `Int2d`, `Short2d` or `Byte2d`)
- the *errors* of the image: described as a `Numeric2d` 2D number array, this can be a `Double2d`, `Float2d`, `Long2d`, `Int2d`, `Short2d` or `Byte2d`). The errors are not mandatory.
- the *exposure* of the image: described as a `Numeric2d` 2D number array, this can be a `Double2d`, `Float2d`, `Long2d`, `Int2d`, `Short2d` or `Byte2d`). The exposure is not mandatory.
- a *flag*: described as a `Flag` (see later). The flag is not mandatory.

The `SimpleImage` also holds information to do coordinate conversions (using the World Coordinate System, WCS) and information of the wavelength at which the image was taken.

When constructing a `SimpleImage`, you should usually first construct a `WCS` object for the coordinate information, and the `Numeric2ds` needed for the image.

The following example shows how to construct an `SimpleImage` with `WCS` coordinates associated with it, an image of 60x40 pixels, no errors, no exposure and with one pixel flagged out (the pixel 55, 35). The reference pixel is at position `crpix1`, `crpix2` with the first row and column starting at position 0, 0. The `crval` keywords are given in decimal degrees in RA and Dec.


```

from herschel.share.unit import *

myIm = Float2d(60, 40) # ❶
for i in Int1d.range(60):
    for j in Int1d.range(40):
        myIm.set(i, j, i + j)

myFlag = Flag(60, 40) # ❷
flaggedOut = Bool2d(60, 40)
flaggedOut.set(55, 35, True)
myFlag.setFlag("UNVALID", flaggedOut);
myUnit = FluxDensity.MILLIJANSKYS # ❸
myWcs = Wcs(crpix1 = 29, crpix2 = 29, crval1 = 30.0, crval2 = -22.5, \
    cdelt1 = 0.00028, cdelt2 = 0.00028, ctype1="RA---TAN", ctype2 = "DEC--TAN")
# ❹
myImage = SimpleImage(description="test image", image = myIm, flag = \
    myFlag, unit = myUnit, wcs = myWcs) # ❺
myImage2 = SimpleImage(wcs = myWcs) # ❻
importImage(image = myImage2, filename="ngc6992.jpg")
# where we now import our JPG image into the SimpleImage

```

- ❶ A `Float2d` is constructed which describes the image and the pixel values are set to $i + j$ (where i is the x coordinate and j is the y coordinate) in the following three lines.
- ❷ These four lines create and set the flag. We only set the `UNVALID` flag. It is possible to set other flag types, but you'll learn more about this in the `Flag` topic. A 2D boolean array is created (`Bool2d(60,40)`) in the second line. In the third line, pixel (55, 35) is flagged, indicating one bad pixel. The fourth line sets the `UNVALID` pixel.
- ❸ This sets the unit for the pixels. The flux associated with one count in the image (equivalent to `BUNIT` in a FITS image).
- ❹ A `WCS` (World Coordinates Class) object `myWcs` is created. The centre pixel is set to (29, 29) which is a projection of the sky coordinate with right ascension 2h00m00s and declination -22d30'00". For more information consult the `WCS` class documentation
- ❺ Constructs the `SimpleImage`. Note that there is no error or exposure set for this `SimpleImage`.
- ❻ Construct another `SimpleImage` and apply the same `WCS` to it. Then import a JPEG of NGC6992, there is no mask, error or exposure. Using the `importFile` method, it is also possible to import FITS files. The `Wcs` information from the FITS file will also be included.

Example 7.1. Constructing a `SimpleImage`

Instead of constructing a new `SimpleImage`, you can also import an image (`*.jpeg`, `*.jpg`, `*.tiff`, `*.tif`, `*.png`, `*.fits`, `*.fts` or `*.fit`) into the `SimpleImage`.

A `SimpleCube` works very similarly as a `SimpleImage`. 3-dimensional datatype should be given as parameters.

7.3. Working with Flags

A flag can be used to flag out bad pixels. Using a flag, it is possible to tell the reason why a certain is flagged out.

```

myFlag = Flag(60,40) # ❶
myFlag.addFlagType("SATURATED", "Saturated pixels") # ❷
flaggedOut = Bool2d(60, 40)
flaggedOut.set(55, 35, True)
myFlag.setFlag("UNVALID", flaggedOut) # ❸
flaggedOut2 = Bool2d(60, 40)
flaggedOut2.set(50, 35, True)
myFlag.setFlag("SATURATED", flaggedOut2) # ❹
print myFlag.getFlagTypes() # ❺
print myFlag.getFlag() # ❻
print myFlag.getFlag("UNVALID") # ❼

```

- ❶ A `Flag` is created with the same dimensions as the image or the cube. In this case, we create a flag with 60x40 pixels. A flag for a cube can be made using `Flag(row, column, layer)`.
- ❷ The `Flag` class makes it possible to create up to 15 different types of flags. In this example, we create a new flag type with the name `SATURATED`. There is always one standard flag type available : `UNVALID`
- ❸ The `UNVALID` flag is set using the `Bool2d` which is constructed in the two previous lines. The pixel with coordinates (55, 35) is flagged out.
- ❹ The `SATURATED` flag is set using the `Bool2d` which is constructed in the two previous lines. The pixel with coordinates (50, 35) is flagged out.
- ❺ The `getFlagTypes()` method returns the existing flag types for this flag. In this example, the existing flags are `UNVALID` and `SATURATED`.
- ❻ The `getFlag` method returns a `Bool2d` with the same dimensions as the `Flag` (60, 40). All flagged pixels are marked as true. In this case, pixels (55, 35) and (50, 35) are true, the rest is false.
- ❼ This method returns a `Bool2d` with the same dimensions as the `Flag`(60, 40). All pixels which are flagged as `UNVALID` are marked as true. In this case, pixel (55, 35) is true, the rest is false.

Example 7.2. Constructing a `Flag`

7.4. How can I display my Image?

Let's display the images that were produced in the previous section.

```

myDisplay = Display(myImage)
myDisplay2 = Display(myImage2)

```

The variables `myDisplay` and `myDisplay2` allow us to refer to the displays and their contents separately.

The result of these two commands is shown below. In each display, the image itself is shown in the window. The *masked out pixels are shown as black pixels*.



Note

When you create a `Double2d(a, b)` and then visualize it as an image, it will appear as a rectangle of width `b` pixels and height `a` pixels, in other words an array of pixels with `a` rows and `b` columns.

At the top right, there is a second, smaller, frame where an *overview of the image* is shown. On this overview, axes are also drawn. For this image, North is down and East is to the right.

You can add an extra parameter when initiating `Display`, which decides whether the display should be shown or not. This can be very useful in scripts, where you don't want all the images to be displayed on the screen, but where you want to look at some images after the execution of the script.

Example :

```
d = Display(SimpleImage, False)
```

will not show the image. You can show the image later using

```
d.setVisible(True)
```

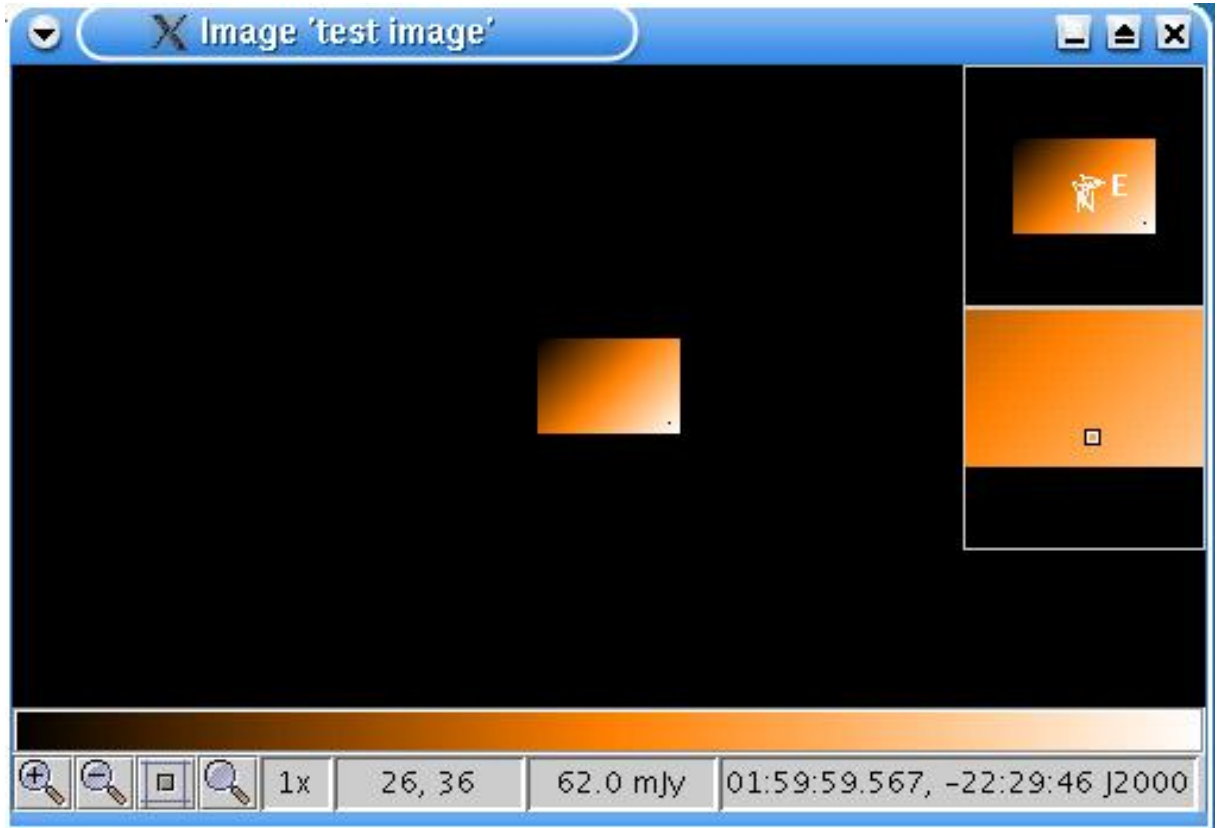


Figure 7.1. Display of an image created in Herschel DP.

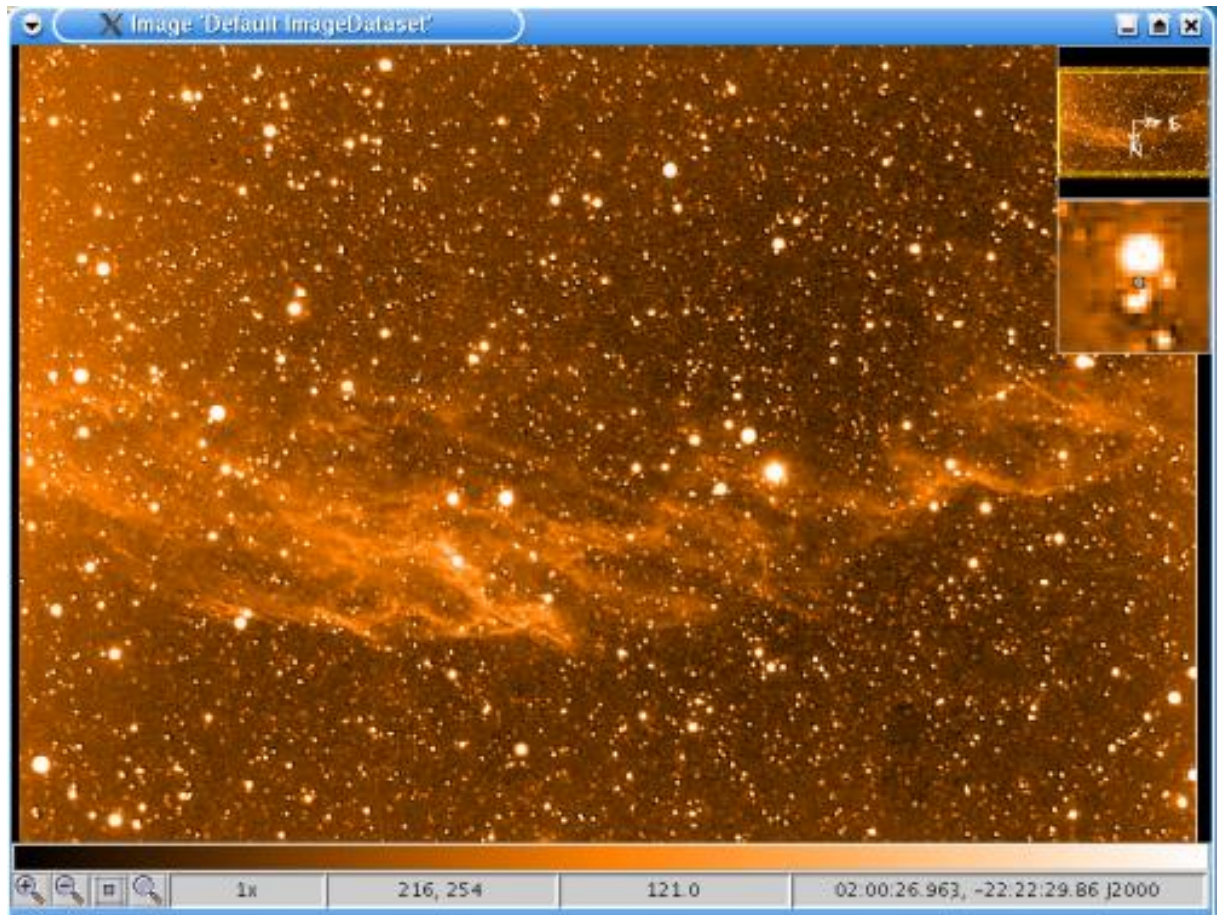


Figure 7.2. Display of an imported JPEG file.

The pixel position (x and y), the pixel value and the WCS coordinates of the mouse position in above figures are shown along the bottom of the image, together with zoom capabilities.



Warning

The x position increases from left to right, and the y position from top to bottom. Note that **the pixel position is currently displayed as y, x** .

Under the window where the overview is located, there is another frame which shows a zoomed in version of the *area located under your mouse*.

Directly under the image, you can see a color bar. It is possible to click on the *colorbar* and move your mouse to change the slope of it.

At the bottom of the window, you can see a *statusbar*. Using the four icons you can zoom in, zoom out, zoom to fit the window and return to the normal zoom (1x). Beside the icons, you can see the currently displayed magnification, the pixel coordinates at which the mouse is pointing, the pixel value and the concurrent sky coordinates (based on the WCS information provided).

When you click with the *right mouse button on the image*, you get a menu. Here, you can open a window that allows changes to the color table ('Edit colors'), open a window where you can edit the cut levels ('Edit cut levels'), zoom in on the place where the mouse is located ('Zoom in'), zoom out, create a screenshot or print the image.

7.5. Display in more Detail.

From here on, we will elaborate on `myDisplay2`. On a display object, you can do a lot of things. Not all of the possibilities are described here. For an overview of all methods, have a look in the Display javadoc.

Some of the more useful methods are listed below. To apply them on image display "myDisplay2" (for example) use

```
print myDisplay2.<method>
```

or

```
imWcs = myImage2.getWcs() # for a SimpleImage
```

```
imWcs = myDisplay2.getWcs() # for a displayed image
```

```
print imWcs
```

in order to get and print the WCS of an image.

and

```
print imWcs.getWorldCoordinates(100,200)
```

in order to get RA and Dec at the pixel coordinates 100, 200.

Table 7.1. Useful methods that can be used on Display

<code>getWcs()</code> As above -- can then get WCS values for a pixel or pixel coordinates for an input WCS position. For example in the latter case: <pre>print imWcs.getPixelCoordinates(30.1, -22.46)</pre>	Returns the image coordinates corresponding to the given sky coordinates, where the sky coordinates are presented in degrees.
<code>getIntensity(x ra, y dec)</code>	Returns the intensity of the pixel at the given (Sky or Image) coordinate position
<code>setColorTable(colortableName, intensityName, scaleName)</code>	Sets the <i>color table</i> of the image
<code>setBackground(Color.red)</code>	Sets the background of displayed image to red (or any other colour that is specified). Current default is white.
<code>setCutLevels(min, max)</code>	Set the cut levels between <i>min</i> and <i>max</i>
<code>setZoomFactor(zoomFactor)</code>	Zoom by the given <i>zoom factor</i>
<code>addAnnotation(annotation, x ra, y dec)</code>	Adds an annotation to the image on the given coordinates

The following example shows how some of the above can be used from the command line in JIDE.

```
# Sets the zoom factor on the second of our displays
myDisplay2.setZoomFactor(2)
# Prints output to the console of the sky coordinates at pixel (434,236)
print myDisplay2.getCurrentWcs().getWorldCoordinates(434, 236)
# Prints the intensity of the pixel at this position
print myDisplay2.getIntensity(434, 236)
# Sets min and max intensity levels for display
myDisplay2.setCutLevels(50, 250)
# Provides an annotation at coordinate (400,300)
myDisplay2.addAnnotation("Annotation", 400, 300)
```

Example 7.3. Illustration of some Display and SimpleImage methods.

The result can be seen in the figure below.

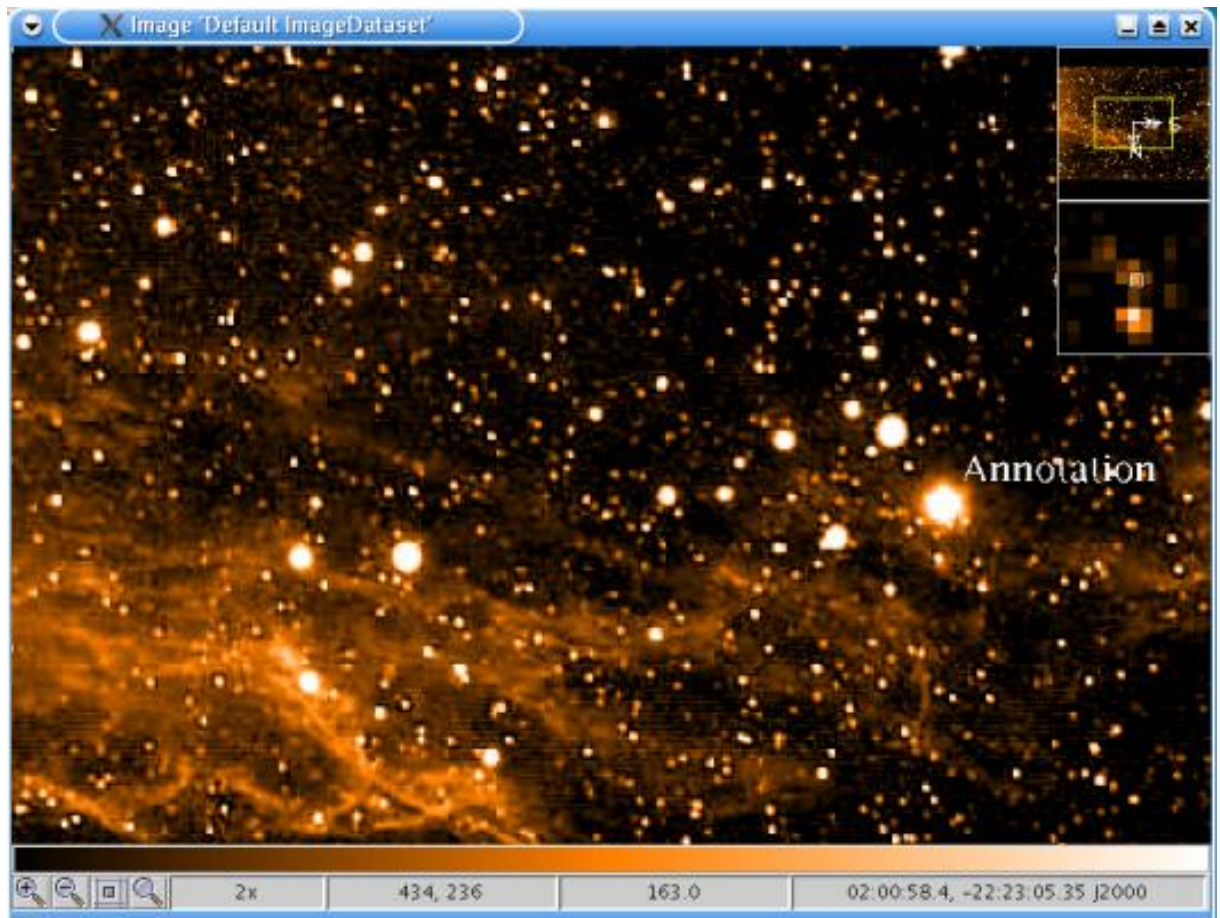


Figure 7.3. Results from above example showing the get sky coordinates, zoom and annotation features of Display.

7.6. Working with the World Coordinates System

The Wcs class enables the user to define a transformation between the pixel coordinates and world coordinates.

```
i = SimpleImage()
i.image=RESHAPE(Double1d.range(200*300), [200,300])
```

```

myWcs = Wcs()
myWcs.ctype1 = "LINEAR"
myWcs.cdelt1 = 5
myWcs.crvall = 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
print i.wcs #to see the WCS of the image
    
```

The above example will create a coordinate system, where the temperature and current are set for the axes. The x-axis is LINEAR (ctype1), has the central pixel in column 0 (crpix1), has a value of 200 in the central pixel (crvall), 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.

It is also possible to use the Wcs class to define transformations between pixel coordinates and sky coordinates. This can be done using the standard Wcs parameters. An example is given below :

```

wcs2 = Wcs() # ❶
wcs2.setCrpix1(128)
wcs2.setCrpix2(128) # ❷
wcs2.setCrvall(101.676612741936)
wcs2.setCrval2(0.829427624677429) # ❸
wcs2.setCtype1("RA--TAN")
wcs2.setCtype2("DEC--TAN") # ❹
wcs2.setRadesys("ICRS")
wcs2.setEquinox(2000.0) # ❺
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, "") #❻
    
```

- ❶ A Wcs is created.
- ❷ The central pixel is set. In this case, the central pixel is at (128, 128).
- ❸ The value of the central pixel is set. In this case, the first central pixel is located at 6h46'42.387" and the second pixel at 0 degrees 49'45.94".
- ❹ The type of the axes is set. The first axis defines the right ascension (in a gnomonic projection) and the second axis defines the declination (in a gnomonic projection).
- ❺ The coordinate system is set (here, we use the standard ICRS type). The equinox is also set.
- ❻ The linear transformation matrix is set. This defines the pixel size and the rotation of the images.

7.7. How can I use Operations on my Images?

At present, the following operations are available on (Simple)Images :

- clamping (clipping) the high and/or low values of an image (**ClampTask**)
- cropping an image to a smaller image (**CropTask**)
- making a histogram of a whole image or of a region of interest (**ImageHistogramTask**)
- scaling an image (**ScaleTask**)
- translating an image (**TranslateTask**)

- transposing an image (**TransposeTask**)
- performing aperture photometry (**AnnularSkyAperturePhotometryTask** and **RectangularSkyApertureTask**)
- contour plotting (**ManualContourTask** and **AutomaticContourTask**)
- 2D profile plotting **ProfileTask**
- smoothing/filtering an image (**AverageFilteringTask** and **MedianFilteringTask**)
- making noise images (**AverageNoiseTask** and **MedianNoiseTask**)
- getting the cut levels of the image (**CutLevelsTask**)
- adding a flag to an image for pixels for which the value is greater than a given value (the saturation value) : these pixels are flagged as SATURATED (**FlagSaturatedPixelsTask**)

7.7.1. Clamping (or clipping) an Image

We can set the minimum and maximum values for our display by the use of the clamping command. All pixels with values at or below the "low" parameter value are given the "low" parameter value. Similarly, all pixels with values at or above the "high" parameter value are allocated the "high" parameter value. An example of its use is given below. Here we create a clamped image in a SimpleImage called "im_clam" and then display it

```
im_clam = ClampTask()(image=myImage2, low=40.0, high=100.0)
Display(im_clam)
```

This could also have been done on one line without the necessity of creating the SimpleImage of the clamped image.

```
Display(ClampTask()(image=myImage2, low=40.0, high=100.0))
```

7.7.2. Cropping an Image

The size of an image can be reduced through cropping. The parameters, "row1", "column1" define the **top left** hand corner and "row2", "column2" define the **bottom right** hand corner of the cropped image. The following illustrates its usage. A cropped image is created, then it is displayed.

```
im_crop = CropTask()(image=myImage2, row1=50, column1=100, row2=250, column2=300)
Display(im_crop)
```

7.7.3. Histogram of an Image

Different tasks have been written that enable you to make a histogram of an image, or of a certain region of interest. The pixel values and the corresponding frequencies are placed in arrays, which can be plotted against each other using `PlotXY` (see Chapter 6). In all those tasks specifying the number of bins is mandatory and specifying the minimum and maximum pixel values is optional (the default extreme pixel values are the image cut levels). When making a histogram of a region of interest, one must specify the circle, ellipse, rectangle or polygon bounding this region.

To make a histogram of an image (`myImage`) using the minimum and maximum pixel values of your choice (200 and 1500) and 2000 bins, one must type

```
histogram=ImageHistogramTask()(image=myImage2, lowCut=0, highCut=250, bins=100)
```


If one wants to use the image cut levels, one should simply omit the second and third input parameter.

To make a histogram of a region bounded by a circle, one must specify the center (in pixel or sky coordinates) and the radius (in pixels) of that circle. The commands are

```
histogram=CircleHistogramTask()(image=myImage2, centerX=300, centerY=250, \  
    radiusPixels=100, bins=2000) # pixel coordinates  
histogram=CircleHistogramTask()(image=myImage2, centerRA="22.5", \  
    centerDec="30.0", radiusArcsec=150, bins=2000) # sky coordinates
```

To make a histogram of a region bounded by an ellipse, one must specify the center (in pixel or sky coordinates), the width and the height (in pixels) of the ellipse. The commands are

```
histogram=EllipseHistogramTask()(image=myImage2, centerX=200, centerY=100, \  
    widthPixels=100, heightPixels=75, bins=2000) # pixel coords  
histogram=EllipseHistogramTask()(image=myImage2, centerRA="22.5", \  
    centerDec="30.0", widthArcsec=100, heightArcsec=75, bins=2000) # sky coords
```

To make a histogram of a region bounded by a rectangle, one must specify the upper left corner (in pixel or sky coordinates), the width and the height (in pixels) of the rectangle. The commands are

```
histogram=RectangleHistogramTask()(image=myImage2, upperLeftX=200, upperLeftY=100, \  
    widthPixels=100, heightPixels=75, bins=2000) # pixel coords  
histogram=RectangleHistogramTask()(image=myImage, upperLeftRA="20.5", \  
    upperLeftDec="30.0", widthArcsec=100, heightArcsec=75, bins=2000)  
#sky coords
```

To make a histogram of a region bounded by a polygon, one must specify the edges (at least 3) of the polygon. This can be done by giving an `ArrayList` of `Doubles` with the pixel coordinates, or an `ArrayList` of `Strings` with the sky coordinates. The commands are

```
histogram=PolygonHistogramTask()(image=myImage, \  
    edgesPixel=arraylistWithPixelCoords, bins=2000)  
histogram=PolygonHistogramTask()(image=myImage, \  
    edgesSky=arraylistWithSkyCoords, bins=2000)
```

-->

To plot the resulting histogram, you must use the following

```
hist_dataset=histogram["Histogram"] #extract out the dataset of interest  
vals=hist_dataset[0].data # this gets the values data of the bins  
freq=hist_dataset[1].data # this gets the frequency data  
plot=PlotXY(vals, freq, titleText="Histogram")  
plot.xaxis.title.text="values"  
plot.xaxis.title.text="frequencies"
```

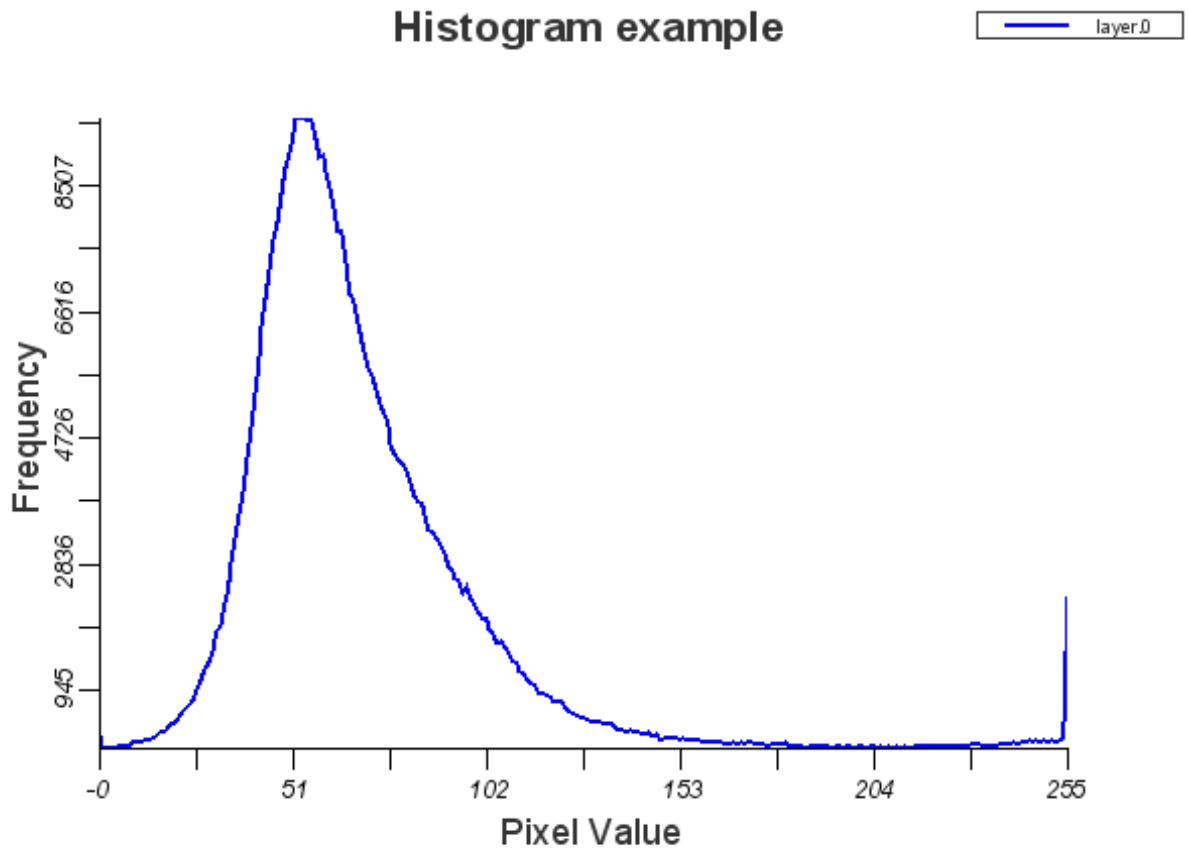


Figure 7.4. Histogram plot of the image of NGC6992 (Veil nebula) stored in the SimpleImage myImage.

It is also possible to produce a "histogram-type" plot by changing the linechart style. This can be done using the PlotXY properties window (type in p.props <return> after the example above) (see Figure 7.5). The "Chart Style" pulldown menu (see plot properties of the first layer -- Layer 0) has LINECHART and HISTOGRAM styles available. Default is LINECHART.

The HISTOGRAM chart style version of the plot is shown in Figure 7.6.

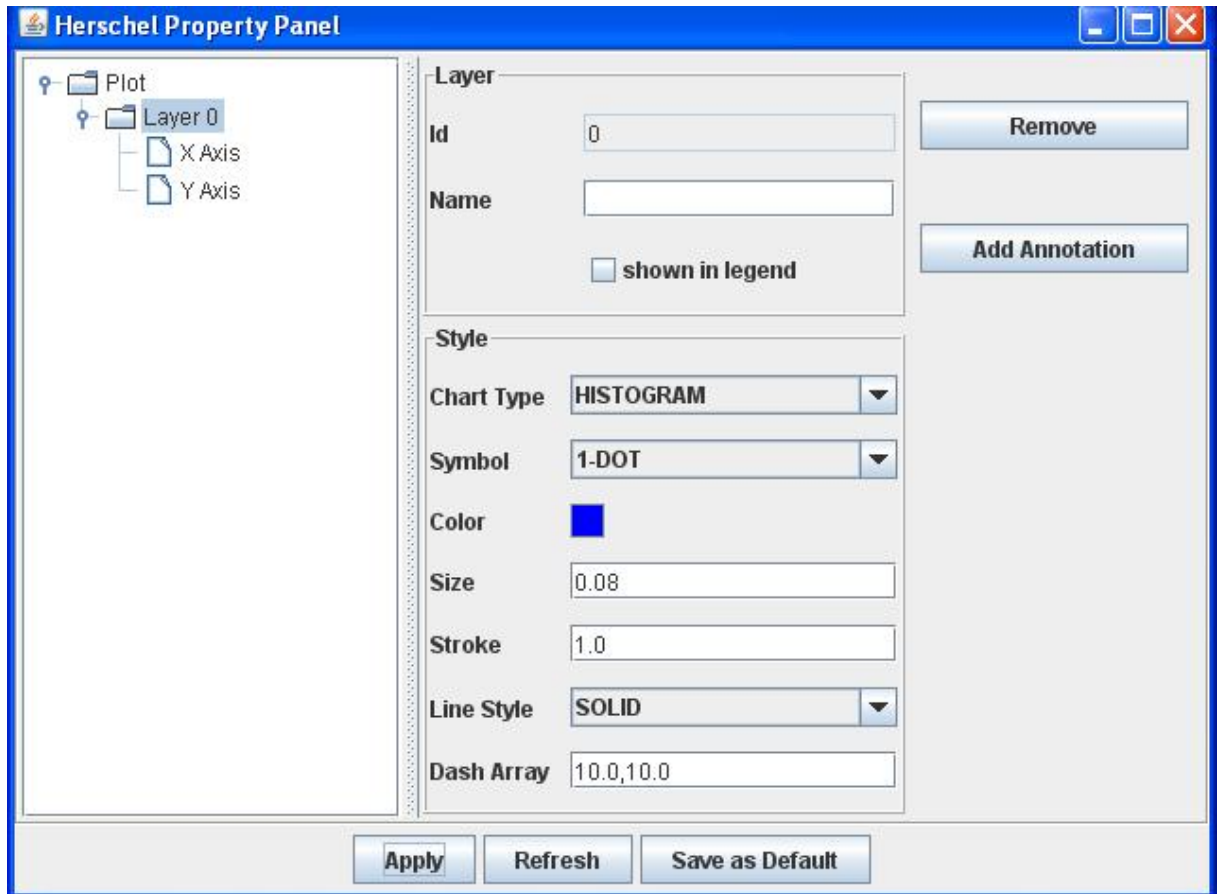


Figure 7.5. Properties for our plot allowing a HISTOGRAM chart style selection

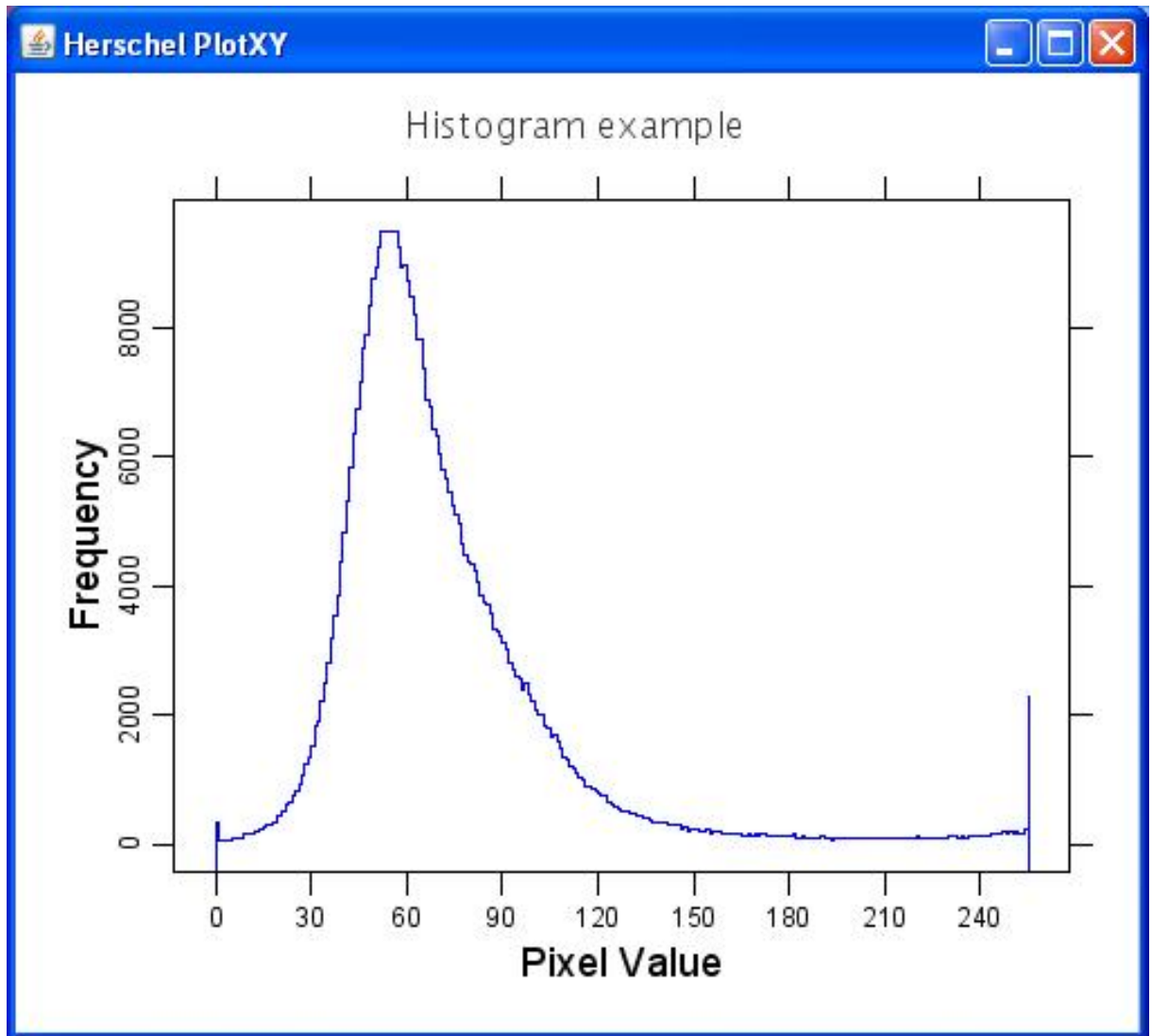


Figure 7.6. Properties for our plot allowing a HISTOGRAM chart style selection

7.7.4. Rotating an Image

The following input line provides an example of how an image may be rotated.

```
rot = RotateTask()(image=myImage2, angle=30.0)
Display(rot)
```

This example rotates the image over 30.0 degrees (clockwise) then displays the rotated image. The result is shown in the figure below.

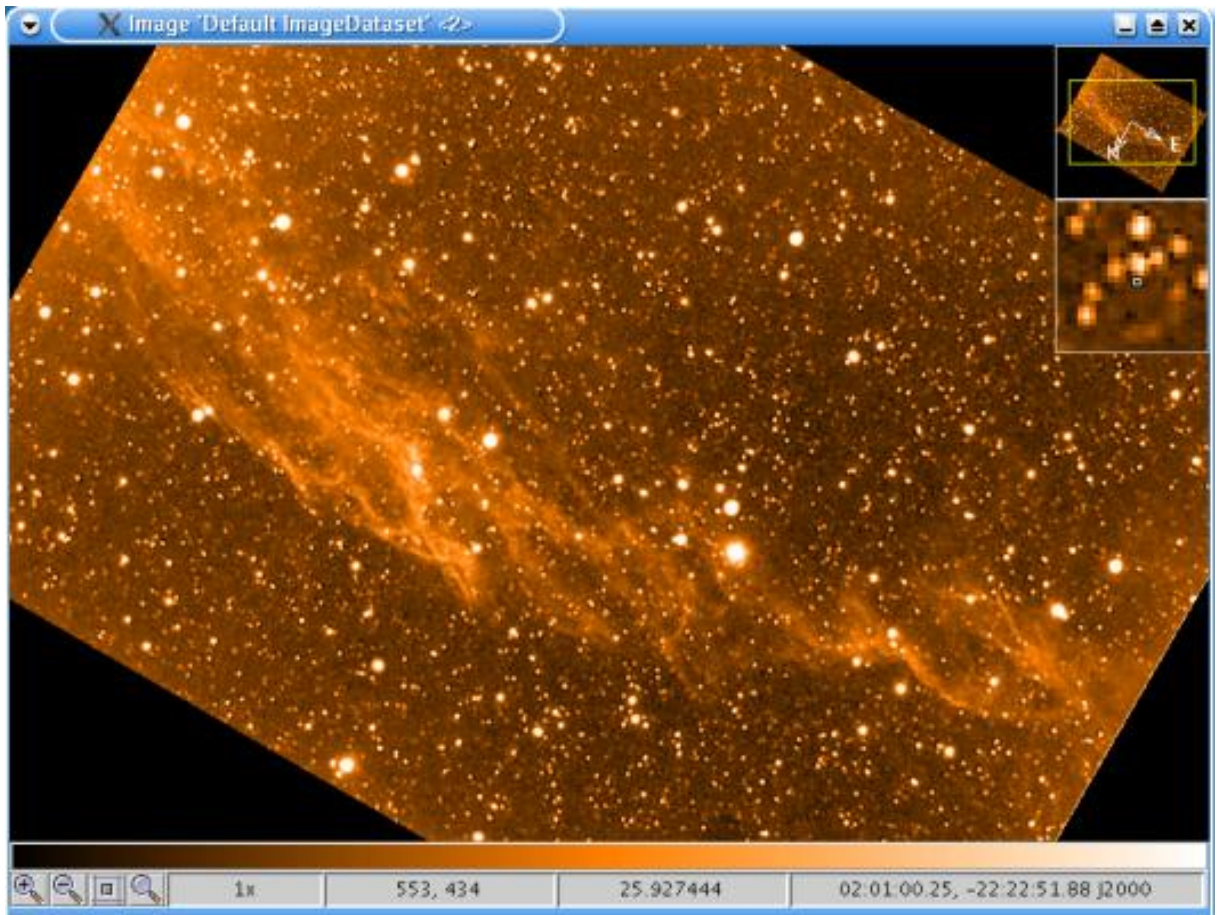


Figure 7.7. Illustration of how an image is rotated by +30 degrees (clockwise) using the DP display package.

When rotating the image, several types of interpolation are possible. By default linear interpolation is used. There are four types of interpolation possible.

- **RotateTask.INTERP_BILINEAR** (*the default* - interpolates one pixel to the right and one below)
- **RotateTask.INTERP_NEAREST** (direct pixel copying)
- **RotateTask.INTERP_BICUBIC** (uses interpolation via a piecewise cubic polynomial)
- **RotateTask.INTERP_BICUBIC_2** (variant of bicubic interpolation that can give sharper results than bicubic)

In order to use the different interpolation schemes we must define the "interpolation" parameter. The following illustrates this point.

```
b = RotateTask()(image=myImage2,angle=50.0,interpolation=RotateTask.INTERP_NEAREST)
Display(b)
```

Using the bicubic interpolations also requires a declaration of the number of subsample bits to be used. More bits gives more accuracy but at extra computational costs. The default bit subsampling is 16. An example of using bicubic interpolation with 32 bit subsampling is

```
b = RotateTask()(image=myImage2,angle=50.0,\
interpolation=RotateTask.INTERP_BICUBIC, subsampleBits=32)
Display(b)
```

7.7.5. Scaling an Image

The size of an image can be magnified in the x and y directions independently using the `ScaleTask()` command. It can be used in a similar way as the `RotateTask()` command (see section 8.5.4) and has the same set of interpolations available to it. The following magnifies in the x direction by a factor of 0.5 and in the y direction by a factor of 2. This is then displayed.

```
s = ScaleTask()(image=myImage2, x = 0.5, y = 2, \
    interpolation=ScaleTask.INTERP_BICUBIC, subsampleBits=32)
Display(s)
```

This provides a very elongated image as shown in the figure below.

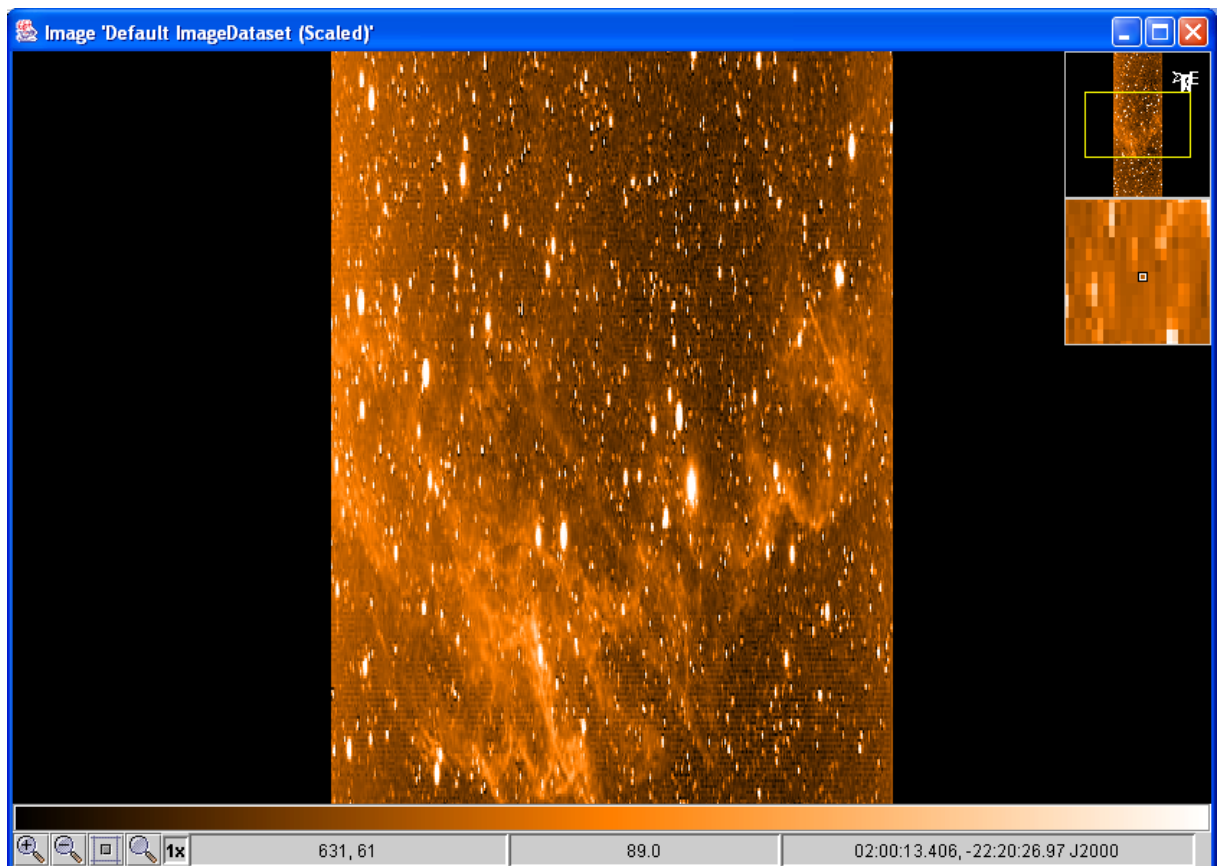


Figure 7.8. An example of independent x and y axis magnification of images using `ScaleTask()`.

7.7.6. Translating an Image

An image can be translated in x and y pixels (fractional pixels allowed) or by an angle (in degrees) in ra and dec (or sky coordinates).

Pixel translation:

```
from herchel.ia.toolbox.image import TranslateTask
im_trans_pix = TranslateTask()(image=myImage2, x= 50.4, y = -5.3)
```

Sky coordinate translation for images holding WCS information:

```
from herchel.ia.toolbox.image import TranslateTask
im_trans_sky = TranslateTask()(image=myImage2, ra=0.02, dec=-0.05)
```

This translates the specified number of degrees in RA and Dec.

7.7.7. Transposing an Image

Images can be transposed in a number of ways.

- FLIP_VERTICAL (flips top and bottom)
- FLIP_HORIZONTAL (flips from side to side)
- FLIP_DIAGONAL (bottom left to top right)
- FLIP_ANTIDIAGONAL (top left to bottom right)
- ROTATE_90 (clockwise rotation)
- ROTATE_180
- ROTATE_270

An example of how to use this command is

```
im_pose = TransposeTask()(image=myImage2, type=TransposeTask.FLIP_DIAGONAL)
```

This command flips the image so that pixels to bottom left appear at top right a flip around the image axis going from top left to bottom right of the image (*antidiagonal* works on the opposite diagonal).

7.7.8. AperturePhotometry

To perform aperture photometry on an SimpleImage we can use the AperturePhotometryTasks.

There are two asks available for aperture photometry: `AannularSkyAperturePhotometryTask` (if you want to use an annular sky aperture) and `rectangularSkyAperturePhotometryTask` (if you want to use a rectangular sky aperture). In both cases you have to specify the pixel coordinates (Doubles *centerX* and *centerY*) or the sky coordinates (Strings *centerRA* and *centerDEC*) of the target centre, the radius of the circular target aperture (Double *radius*) and the sky estimation algorithm (String *algorithm*).

There are 5 algorithms available, 0=Average, 1=Median, 2=Mean-median, 3=Synthetic mode, 4=DAOphot. It is also possible to have fractional pixels used if `fractional=1`.

When you want to use an annular sky aperture, you must specify the inner (Double *inner*) and outer (Double *outer*) radius of the annular sky aperture in pixels. An example is given below. Note that this aperture is centred around the target centre (i.e. the centre of the circular target aperture).

```
results1=annularSkyAperturePhotometryTask(image=myImage2,\
    centerX=30.0,radiusPixels=4.0,centerY=40.0,innerPixels=15, \
    outerPixels=20,fractional=1,algorithm=4)
# note that for sky centers use centerRA, centerDEC and Arcsec
# instead of Pixels in the above expression
```

The alternative is to use a rectangular sky aperture. Then you have to specify the pixel coordinates (Doubles *upperLeftX* and *upperLeftY*) or the sky coordinates (Strings *upperLeftRA* and *upperLeftDEC*) of the upper left corner of the rectangular sky aperture, the width (Double *width*) and the height (Double *height*) of the rectangular sky aperture in pixels, and the algorithm.

There are 5 algorithms available, 0=AVERAGE, 1=MEDIAN, 2=MEAN_MEDIAN, 3=SYNTHETIC_MODE, 4= DAOPHOT. It is also possible to have fractional pixels used if `fractional=1`.

```
results2 = rectangularSkyAperturePhotometryTask(image=myImage2,centerX=30.0,\
radiusPixels=4.0,centerY=40.0,upperLeftX=100.0,upperLeftY=200.0,\
widthPixels=5.0,heightPixels=4.0,fractional=1,algorithm=4)
# note that for sky centers use centerRA, centerDEC and Arcsec
# instead of Pixels in the above expression
```

Results are in the form of a dataset which can be viewed in the usual way HIPE/JIDE, e.g. click on the variable "results1" in HIPE -- then click on the dataset shown in the outline view. Results are shown in the Editor view. Or

```
print results1["Results table"]
```

```
-->
```

7.7.9. Contour plotting

For contour plotting on a given image (`SimpleImage`), two Tasks are available. The first Task, `manualContourTask`, takes a list of contour values as input parameter (`ArrayList<Double> values`). The second Task, `automaticContourTask`, takes the number of contour levels (`Integer levels`) and their distribution (`String distribution`) as input parameters. The extreme contour values (`double[] extreme`) are to be given optionally (default : the image cut levels). Examples are given below.

```
v = java.util.ArrayList()
v.add(140.0)
v.add(160.0)
v.add(210.0) # produced an array with contour levels in it called "v"
iml>manualContourTask(image=myImage, values=v) # apply manual contour levels
imageContour2=automaticContourTask(image=myImage, levels=3, min=140.0, \
max=200.0, distribution=1)
# possible distribution for contour levels between min and max values,
# 0=Linear, 1=Log, 2=LN
```

Both Tasks return an `ImageContour` format output. Such an `ImageContour` can easily be plotted over an image, using the method `addImageContour` from the `Display` class.

```
v = java.util.ArrayList()
v.add(40.0)
v.add(60.0)
v.add(80.0) # produced an array with contour levels in it called "v"
imageContour1>manualContourTask(image=myImage, values=v)

colors=java.util.ArrayList()
colors.add(java.awt.Color.red)
colors.add(java.awt.Color.green)
colors.add(java.awt.Color.blue)
s=Display(myImage)
s.addImageContour(imageContour1, colors,1) #for contour overlay
```

7.7.10. 2D Profile Plotting

`ProfileTask` allows determining the intensity of the pixels along a straight line on a given image, when you know the begin and end of that straight line. The only input parameters are the image (`ImageDataset image`) and the pixels coordinates (`Doubles beginX, beginY, endX` and `endY`) or sky coordinates (`Strings beginRA, beginDEC, endRA` and `endDEC`) of begin and end of the straight line. An example is given below.

```
profile=profileTask(image=myImage2, beginX=132, beginY=267, endX=500, endY=307)
p = PlotXY(profile["Profile"].data) # to display raw cut
# or with sky coordinates
profile = profileTask(image=myImage,beginRA="02:01:08.309",\
```



```
beginDec="-22:18:05.11",endRA="02:00:17.144",endDec="-22:15:21.95")
PlotXY(profile["Profile"].data)
```

Such a Task returns the `Double1d` *profile*, which represents the intensity of the pixels along the straight line.

7.7.11. Smoothing Images

Two Tasks have been developed to smooth images : `averageFilterTask` uses an average filter, `medianFilterTask` uses a median filter. The commands are

```
filteredImage1=averageFilteringTask(image=myImage)
Display(filteredImage1)

filteredImage2=medianFilteringTask(image=myImage)
Display(filteredImage2)
```

7.7.12. Making Noise Images

Two Tasks allow us to get an idea of the noise in an image : `averageNoiseTask` and `medianNoiseTask`. They use an average and a median filter respectively. The commands are

```
noiseImage1=averageNoiseTask(image=myImage)
Display(noiseImage1)

noiseImage2=medianNoiseTask(image=myImage)
Display(noiseImage2)
```

7.7.13. Flagging saturated Pixels

To flag (with the `SATURATED` flag) pixels for which the value is greater than a given (saturation) value the command is

```
flaggedImage=flagSaturatedPixelsTask(image=myImage, value=2000)
Display(flaggedImage)
```

7.8. How can I display my own numeric2d datatypes?

In many cases, you may have constructed your own 2D array, possibly with a datatype other than `Double2d` (e.g. `Int2d`) that you want to display. This can be done by simply feeding the datatype to `Display`. The example below provides an illustration of how to input a 2D array of `Int2d` (of 16 by 18 pixels) and displays it. A zoom factor of 20 is also used.

```
# Creating an image created from a 15x17 2D integer array
image = Int2d(15, 17)
# Loop to place "intensity" values into the array
for i in Int1d.range(15):
  for j in Int1d.range(17):
    image.set(i, j, i + j)

# Displaying the image with appropriate min/max levels and zooming
d = Display(image, zoomFactor = 20, cutLevels = (0, 30))
```

Example 7.4. Using Display with 2D integer arrays

The results from running above example are show below . .

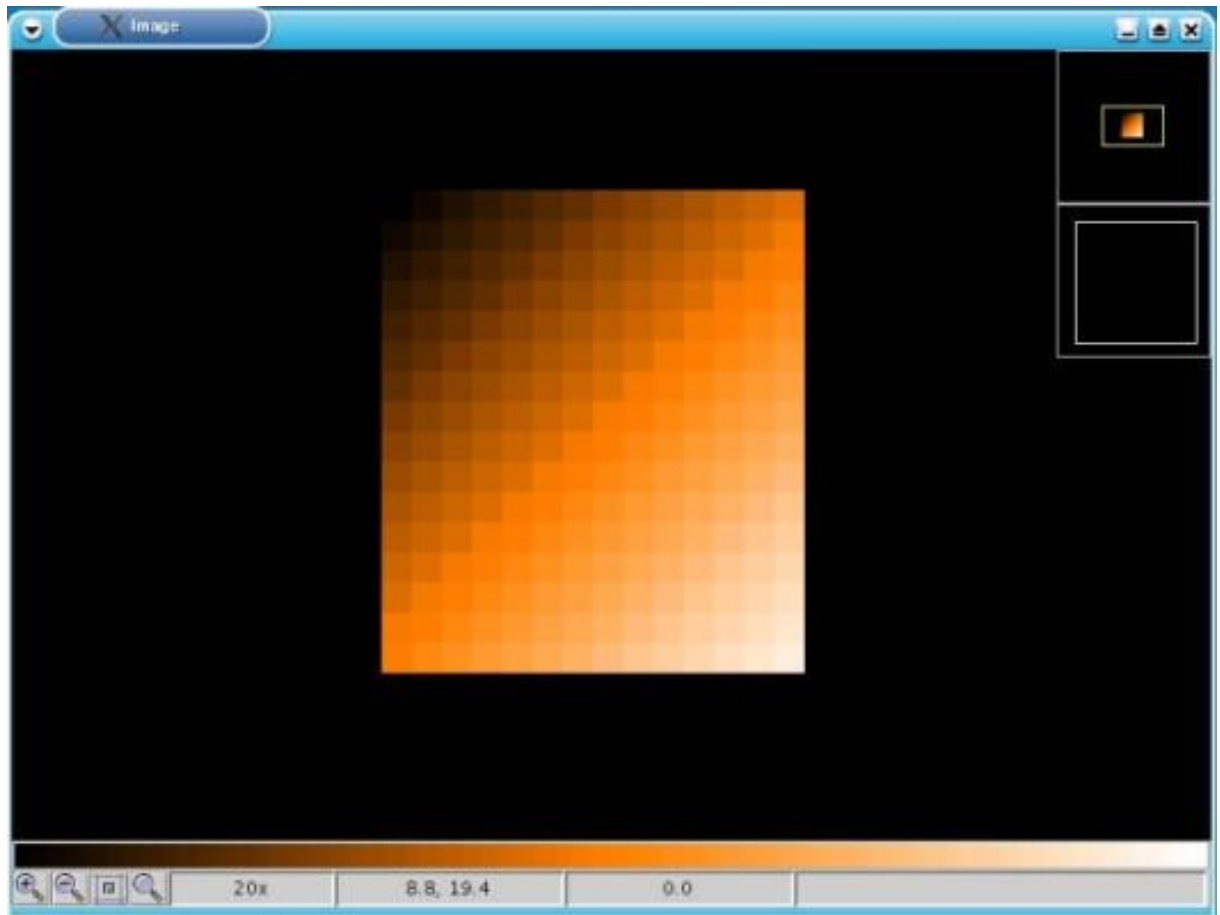


Figure 7.9. Display of an image created from a user-supplied numeric 2D array.

7.9. How to Use Different Layers?

It is possible to show different layers of an image. This can be done by adding a layer to the existing image, but it can also be done by displaying a cube of numeric3d datatype (like an `Int3d`, `Double3d`, ...).

The example shown in the figure below is an elaboration of the first example. It has been created using the command :

```
myDisplay.addLayer(myImage2)
```

We add `myImage2` to `myDisplay`. The screenshot shows that there appears a *slider in the statuspanel*, where you can switch between the different layers. The settings of the new layer will be the same as the settings of the old layers (cut levels, annotations, zoom factor,...).

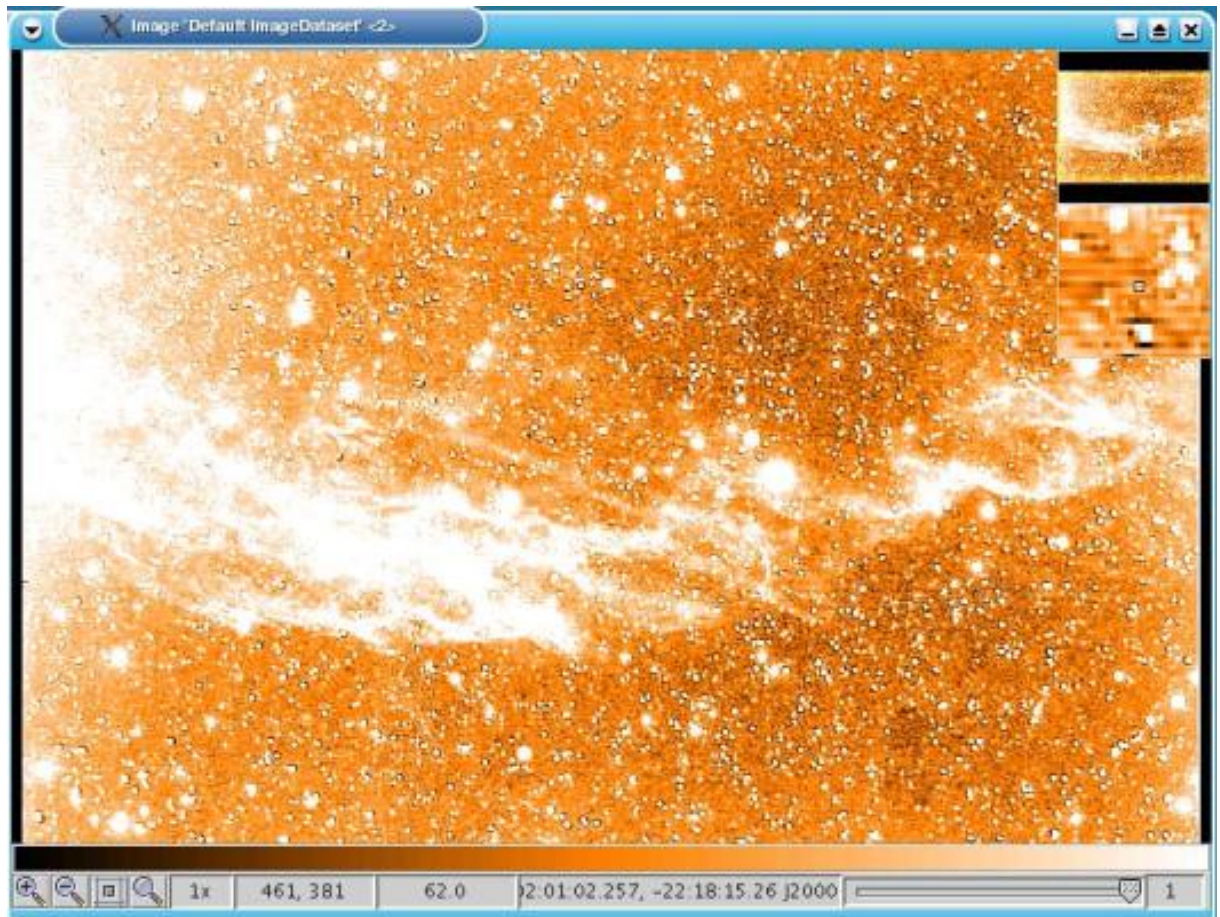


Figure 7.10. The use of layers in Display. Swapping between layers is performed using the slider at bottom right.

7.10. How to place annotations on an Image?

It is possible to add annotations to the image. This can be done in two different ways:

1. Using the command line from your DP session
2. Using the annotation toolbox

Once the annotations are on the image, it is possible to use the mouse to select an annotation, move it around and change its size.

7.10.1. Annotations from the Command Line in your DP session

It is possible to add annotations to the image from the DP prompt.

The following annotations are possible from the command line.

- *Text annotations:* Using `addAnnotation(...)`, `setAnnotationFont(...)` and `setAnnotationFontColor(...)` methods.
- *Greek text annotations:* Using `addGreekAnnotation(...)`, `setAnnotationFont(...)` and `setAnnotationFontColor(...)` methods. The `addGreekAnnotation` method translates the normal characters to greek characters ('a' becomes 'alpha', 'b' 'beta', 'c' 'gamma', ...).

- *Figures as annotations*: Using `addEllipse(...)`, `addLine(...)`, `addPolygon(...)`, `addPolyLine(...)` and `addRectangle(...)` methods. The `addPolygon` and `addPolyLine` methods need an array of doubles as parameter. In this array, the coordinates of the points should be added in this way : `polygon([x1, y1, x2, y2, ...], ...)`

The following example illustrates how to place annotations onto displays.

```

from java.awt import Font
from java.awt import Color

myDisplay2 = Display(myImage2)
# Placing annotation at position (321, 224) on image 'mydisplay2'
myDisplay2.addAnnotation("Veil nebula", 321, 224)
# Changing font type and size for the image 'mydisplay2'
myDisplay2.setAnnotationFont(321, 224, Font("Dialog", 0, 64))
# Changing annotation colour
myDisplay2.setAnnotationFontColor(321, 224, Color(0, 0, 255))
# Adding ellipse with centre at (500, 308.5) width=38, height=37
# linewidth = 3.0 and black color.
myDisplay2.addEllipse(500.0, 308.5, 38.0, 37.0, 3.0, Color(0, 0, 0))
# Adding a position label with greek letter notation at position (100,500)
myDisplay2.addGreekAnnotation("a = 12.34, d = +30.30", 100, 500)
# Changing font of annotation of at (100, 500)...
myDisplay2.setAnnotationFont(100, 500, Font("Dialog", 0, 64))
# ...and changing its color to black too...
myDisplay2.setAnnotationFontColor(100, 500, Color(0, 0, 0))
# ...but white is more visible.
myDisplay2.setAnnotationFontColor(100, 500, Color.white)

```

Example 7.5. Command line addition of annotations to images

The result is shown in the following figure.

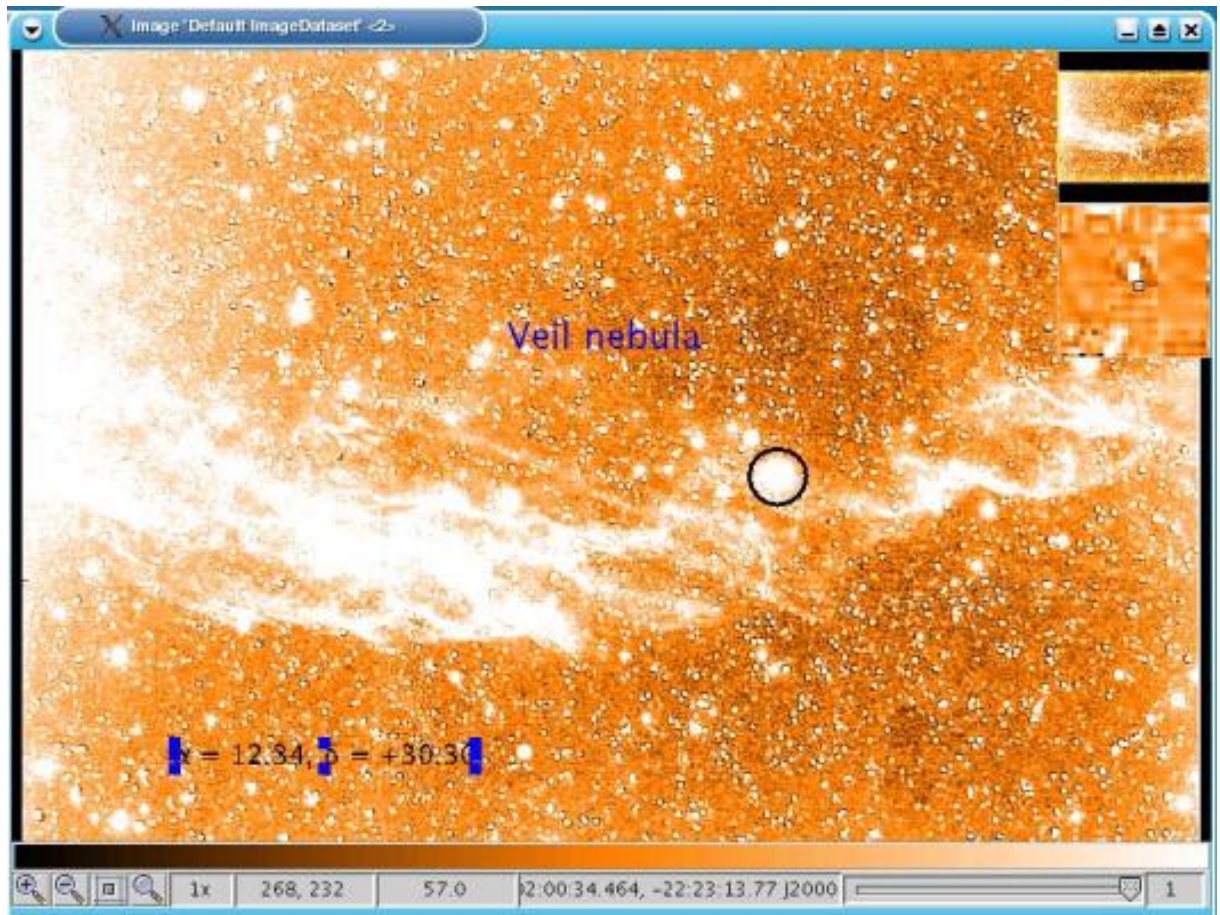


Figure 7.11. Illustrating how annotations can be added in various forms.

7.10.2. Annotations using the annotation toolbox

It is much easier to add the annotations using an annotation toolbox. The annotation toolbox can be shown using :

```
a = myDisplay2.annotationToolbox()
```

It is also possible to fire up the annotation toolbox by clicking with the right mouse button on the image. A popup will appear where you can select 'Annotation toolbox'. If you fire up the annotation toolbox using the popup menu, the jython code can not be generated.

The annotation toolbox is shown in the figure below.

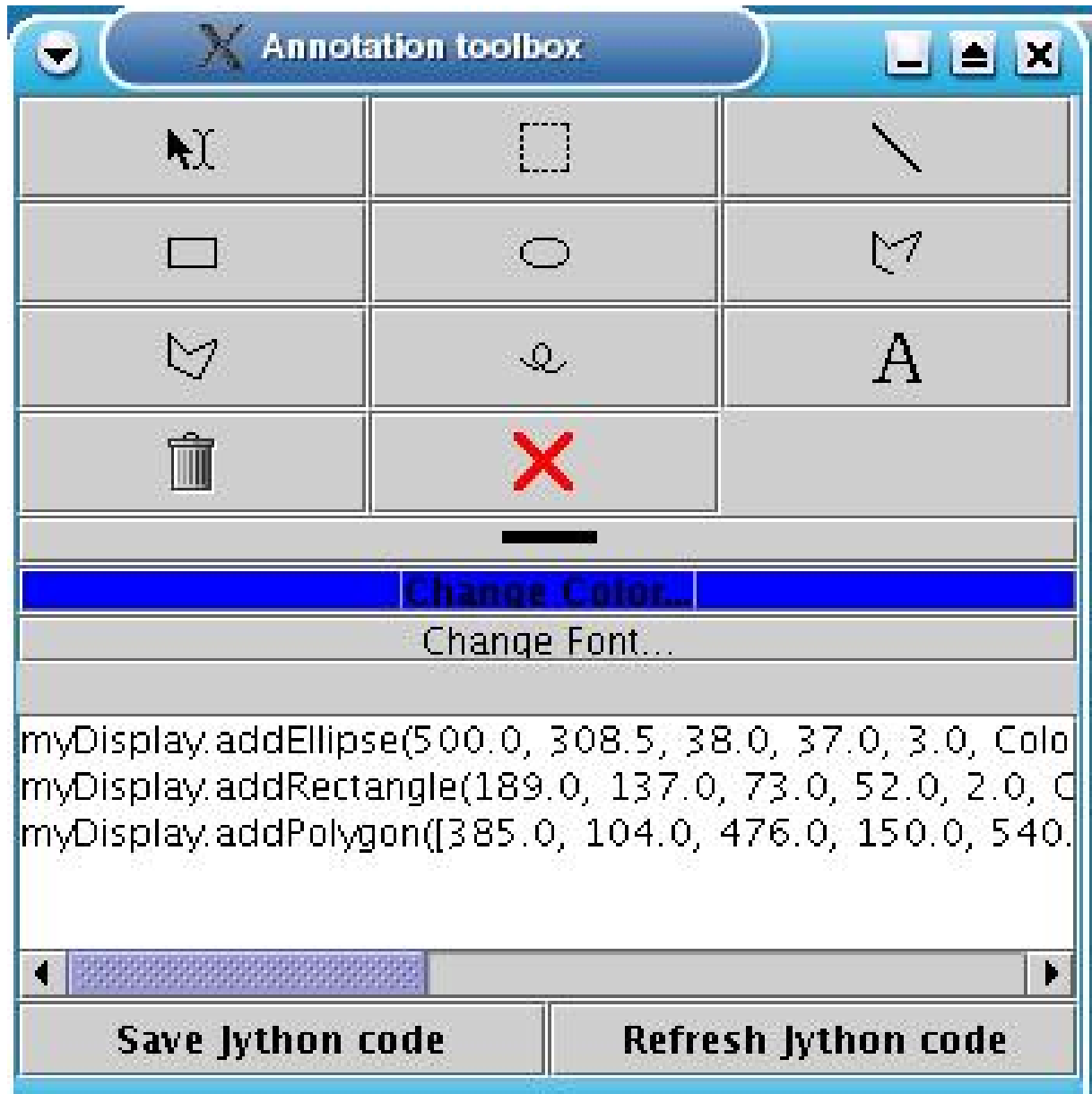


Figure 7.12. The annotation toolbox.

The icons in the annotation toolbox appearing in Figure 7.12 have the following usage (from left to right and from top to bottom):

- Select annotation
- Select all annotations in a region
- Draw a line
- Draw a rectangle
- Draw an ellipse
- Draw a polyline
- Draw a polygon
- Draw with the free hand on the image

- Add a text annotation
- Remove the selected annotation(s)
- Remove all annotations

Letting the mouse linger over an icon also displays its function.

The polygon and polyline methods will 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 will end the selection procedure.

The three buttons below the ones already described change the view of the annotation:

- Change the thickness of the line
- Change the color of the annotation
- Change the font of the text annotation

The jython code needed to regenerate all annotations is given in the lower part of the annotation toolbox. If you change the size of a text annotation, this will not be reflected in the jython code.

7.11. ImageAnalysisToolbox : Image Analysis with Herschel DP

The `ImageAnalysisToolbox` is an interactive, user-friendly toolbox which provides a set of functionalities for the analysis of images, including

- aperture photometry
- image/area histograms
- 2D profile plotting
- contour plotting

We start with a general description of the toolbox, followed by an elaboration on the different functionalities.

7.11.1. General Description of the Toolbox

In this section we describe the toolbox in general terms. We explain how a toolbox can be opened, how to load images into it and how to add layers to it. Also a brief overview of the different parts of the toolbox is given (image panner, image zoom, colorbar, statusbar, tabs,...) as well as a summary of the different functionalities (they will be explained in the next section).

7.11.1.1. Opening a Toolbox, loading Images, adding Images,...

One can open a toolbox and load an image into it, by typing the following commands :

Example 7.6. Opening an `ImageAnalysisToolbox` and loading an image (1)

```
myWcs=Wcs(crpix1=29, crpix2=29, crval1=30.0, crval2=-22.5)
myImage=SimpleImage(wcs=myWcs)
myImage.importFile("ngc6992.jpg")
iat=ImageAnalysisToolbox()
```

```
iat.setImage(myImage)
```

The last two lines can be replaced by

Example 7.7. Opening an ImageAnalysisToolbox and loading an image (2)

```
iat=ImageAnalysis(myImage)
```

When one has opened a toolbox (using the `iat=ImageAnalysisToolbox()` command), one can load an image file (*.jpeg, *.jpg, *.tiff, *.tif, *.png, *.fits, *.fts or *.fit) into it, using the `File > Open > Set image` menu. For fits files the WCS is retrieved from the header.

One can also load multiple images into the toolbox, using the command

```
iat.addImage(otherImage)
```

Example 7.8. Loading multiple images into an ImageAnalysisToolbox (i.e. adding layers)

One can also use the `File > Open > Add image` menu.

The toolbox with an image loaded into it, is shown in the figure below.

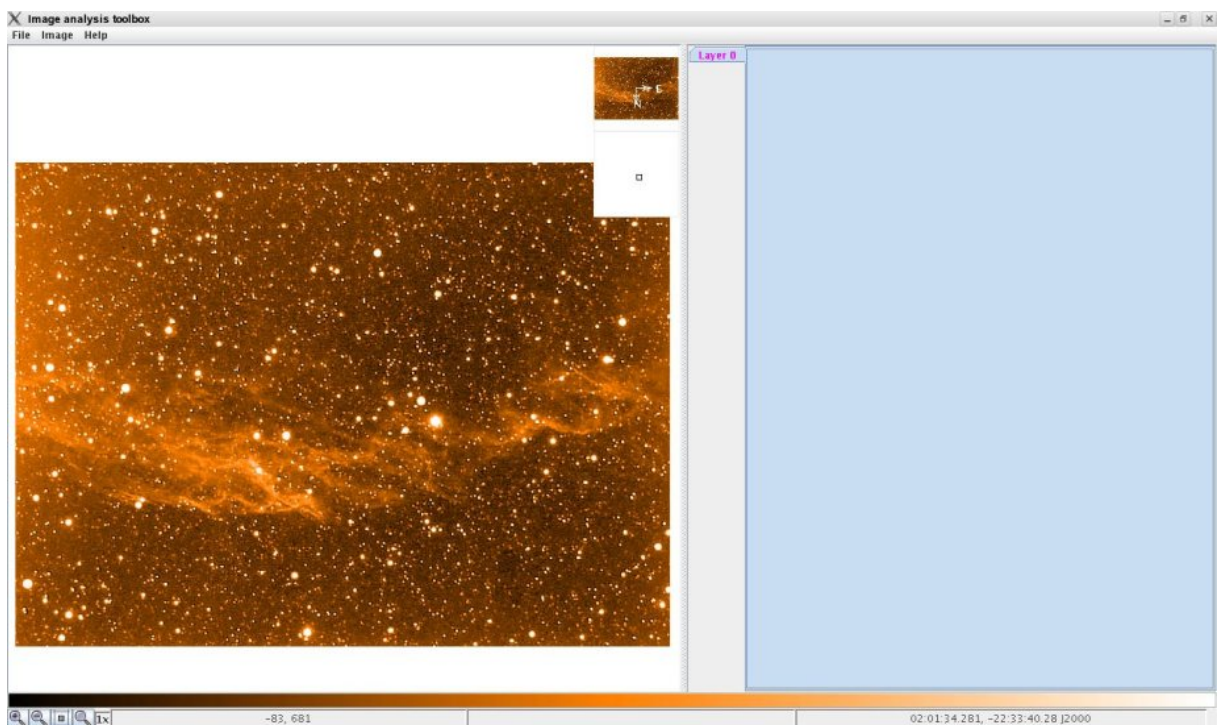


Figure 7.13. An ImageAnalysisToolbox with an image, created in Herschel DP

7.11.1.2. The ImageAnalysisToolbox in more Detail : Tabs, Image Panner, Image Zoom, Colorbar, Statusbar,...

The toolbox has two major parts :

- in the left half of the toolbox, the images are layered behind each other and the upper image (the active image) is shown to the user
- the right half of the toolbox is a tabbed pane for which each tab corresponds with a layer in the left half of the toolbox

By clicking on another tab, another image can be made active.

Image analysis can only be performed on the active image, but you can perform several types of image analysis on it. For each analysis on the active image, a tab is opened within the corresponding tab. You can switch between these tabs merely by clicking on them. Only the figures (straight lines, circles, ellipses, rectangles,...) of the current analysis are shown on the active image. Please do not use the slider, as this feature does not (yet) work properly.

The toolbox has a great resemblance to `Display`, since we integrated it into our toolbox. In the left half of the toolbox not only the active image is shown to the user, but also an *image panner* and an *image zoom*. The first gives an overview of the active image, while the latter zooms in on the region around the mouse position. At the bottom of the toolbox one finds a *colorbar*, on which one can slide the mouse, and a *statusbar*. The latter shows four icons, to zoom in, zoom out, zoom to fit and return to normal zoom (1x). Also the pixel coordinates, pixel value (intensity) and sky coordinates (if available) of the current mouse position are shown on the statusbar. As one moves the mouse over the active image, these data are updated.

When one clicks with the right mouse button on the active image, a menu will be opened. One can choose to change the color table (`Edit colors...`), edit the cut levels (`Edit cut levels...`), zoom in (`Zoom in`) on the current mouse position, zoom out (`Zoom out`), create a screenshot (`Create screenshot...`) or print the image (`Print the image...`). One can also choose the draw annotations (`Annotation toolbox...`).

Three menus can be selected : `File`, `Image` and `Help`.

- in the `File > Open` menu one can choose to set (`File > Open > Set image`) or add (`File > Open > Add image`) a layer
- in the `File > Close` menu one can choose to close one tab (`File > Close > Close active tab`) or all tabs for image analysis on the active image (`File > Close >`)
- clicking the `File > Exit` menu causes the toolbox to be closed
- in the `Image` menu one can choose the type of analysis one wants to perform on the active image

7.11.1.3. Functionalities

As `ImageAnalysisToolbox` is still **under development**, more functionalities will be added in the future, but in the current version of `ImageAnalysisToolbox` the following functionalities are available :

- *2D profile plotting*
- making a *histogram* of the whole image or of a certain region of interest, which is bounded by a circle, an ellipse, a rectangle or a polygon (the user should draw the bounding figure on the image)
- *aperture photometry* with a circular target aperture and an annular or a rectangular sky aperture
- *contour plotting*



Note

All these functionalities will be explained in the next few sections. Note that all these functionalities are also available as Tasks.

2D Profile Plotting

2D profile plotting allows the user to draw a straight line on an image and plot the intensity along that straight line.

One can start 2D profile plotting by choosing the Image > Profile Plotting > 2D Profile Plotting menu, or by typing the following command :

```
iat.plotProfile2D()
```

Example 7.9. Starting 2D profile plotting

Now one can start drawing the straight line on the active image. The beginning of the line can be fixed by clicking once on the image. As one moves the mouse over the active iamge, the straight line will be updated, until the end of the straight line is fixed by clicking a second time on the image. Simultaneously, the intensity plot along the straight line is updated in the right half of the toolbox. Under the intensity plot, one finds the pixel coordinates of begin and end of the straight line and the sky coordinates (if available). An example is given in the figure below.

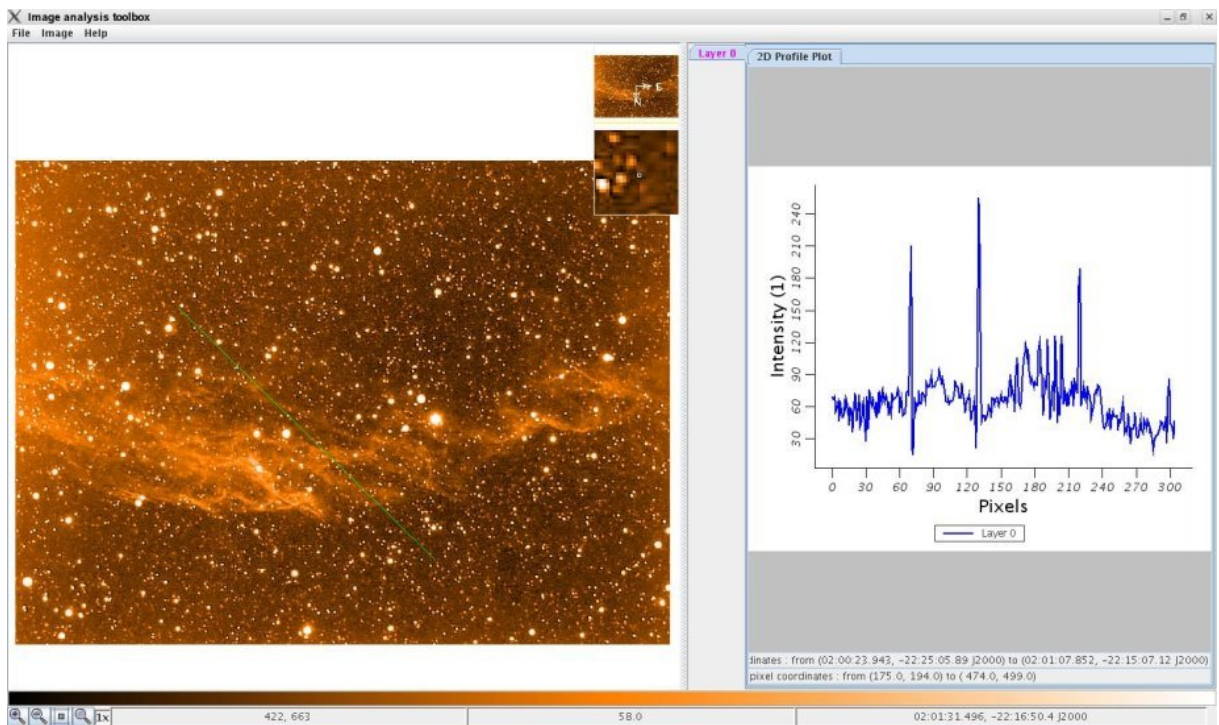


Figure 7.14. A 2D profile plot, created in Herschel DP

Area Histogram

One can make a histogram of an image as a whole, or of a certain region of interest which is specified by the user. This region can be bounded by a circle, an ellipse, a rectangle or a polgon, which one has to draw on the image.

One can start the procedure for making an area histogram by choosing the Image > Histogram menu, or by typing the following command

```
iat.histogram()
```

Example 7.10. Starting an area histogram

This command opens a new tab in the right half of the toolbox, where one can specify the parameters for making an area histogram. This is shown in the figure below.

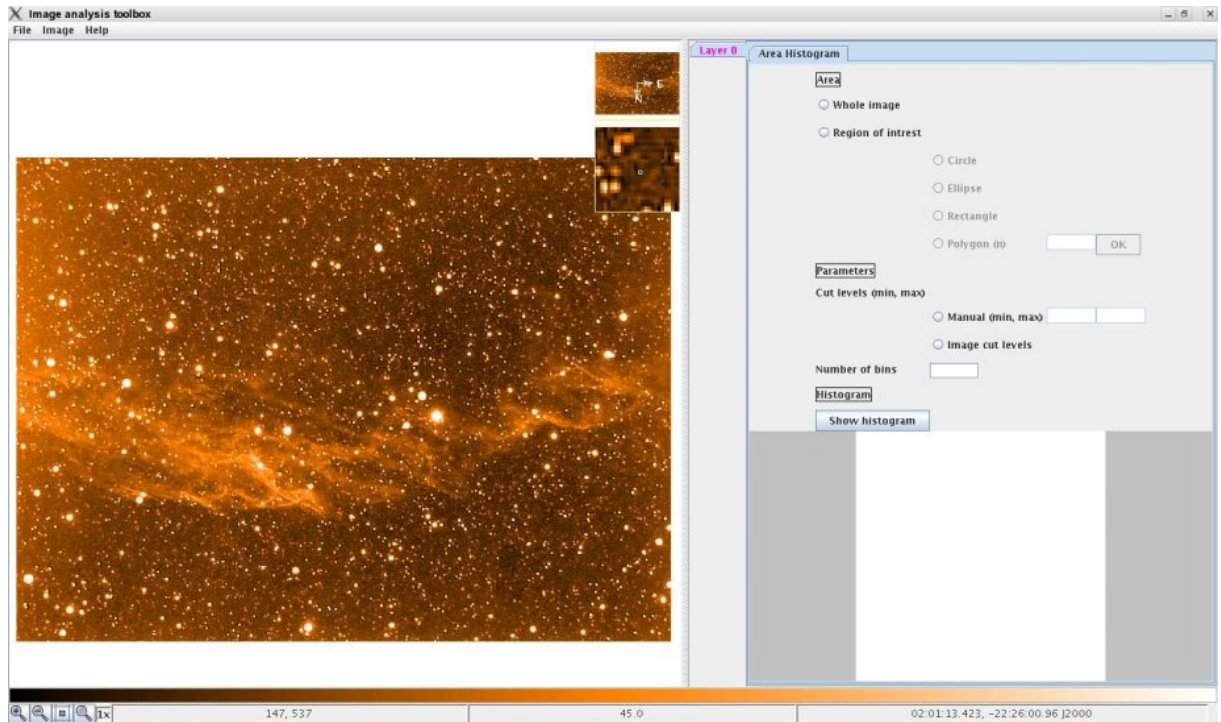


Figure 7.15. Starting an area histogram

As can be seen in this figure, one has to specify whether one wants to make a histogram of the whole (active) image, or merely of a region of interest. When making a histogram of such a region, one has to draw the figure that bounds it. Such a figure can be a circle, an ellipse, a rectangle or a polygon. For a circle, an ellipse and a rectangle one only has to specify two points : the first point by pressing the mouse, the second point by releasing it (the figure is updated as one drags the mouse). For a polygon one has to confirm the number of edges n and click n times on the image to fix all edges of the polygon.

One also has to specify the cut levels and the number of bins that should be used for the histogram. One can give the cut levels manually or decide to use the image cut levels. After hitting the Show histogram button, the histogram is shown in the right half of the toolbox.

An example of an area histogram of a region of interest which is bounded by a circle is given in the figure below.

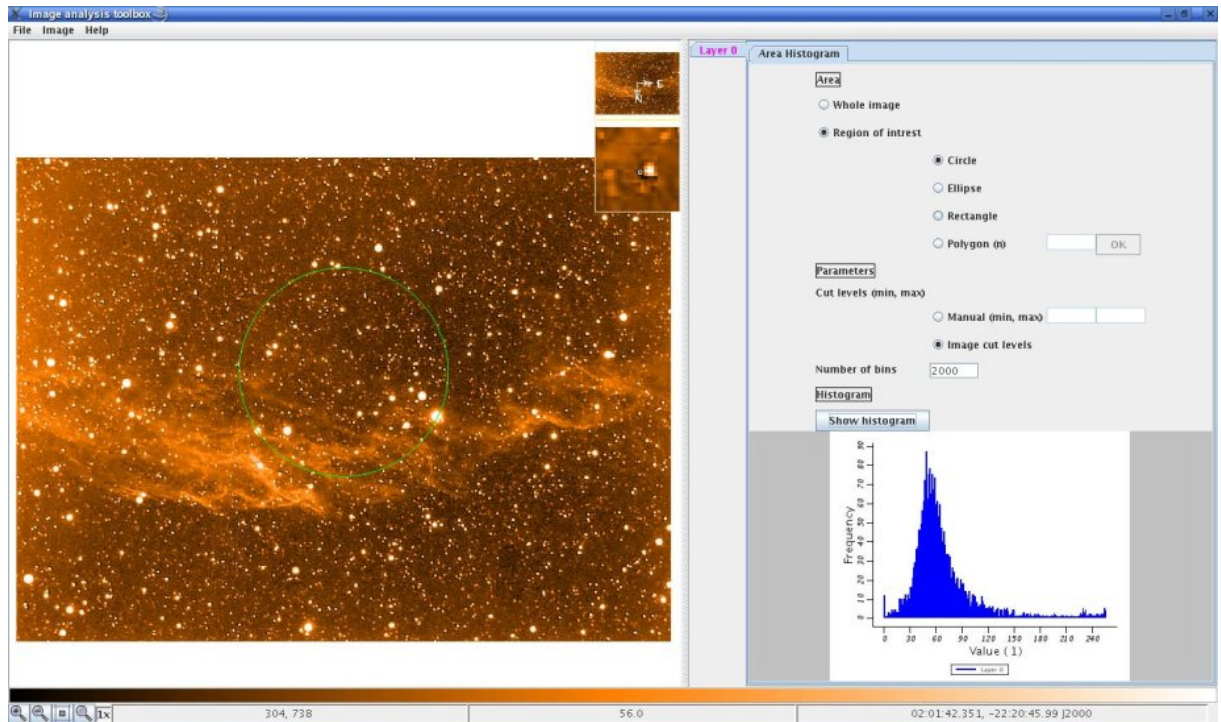


Figure 7.16. An area histogram, created in Herschel DP

Aperture Photometry

One can also perform aperture photometry on an image, using a circular target aperture and an annular or a rectangular sky aperture. There are five algorithms that can be used to estimate the sky : average, median, mean-median, kappa-sigma clipping and daophot. In the mean-median method all values further away from the median than a specified number of times the standard deviation (i.c. 1.5) are discarded and the remaining values are averaged. In the kappa-sigma clipping method this cycle is repeated until there is no change in the mean or the iteration limit (i.c. 10) has been reached. The daophot method is the translation of the algorithm used in the daophot package from IDL to Java.

One can start the aperture photometry by choosing the Image > Aperture Photometry menu or by typing

```
iat.aperturePhotometry()
```

Example 7.11.

Hereby a new tab is opened in the right half of the toolbox, where one must indicate the data that are needed for the aperture photometry. This is illustrated in the figure below.

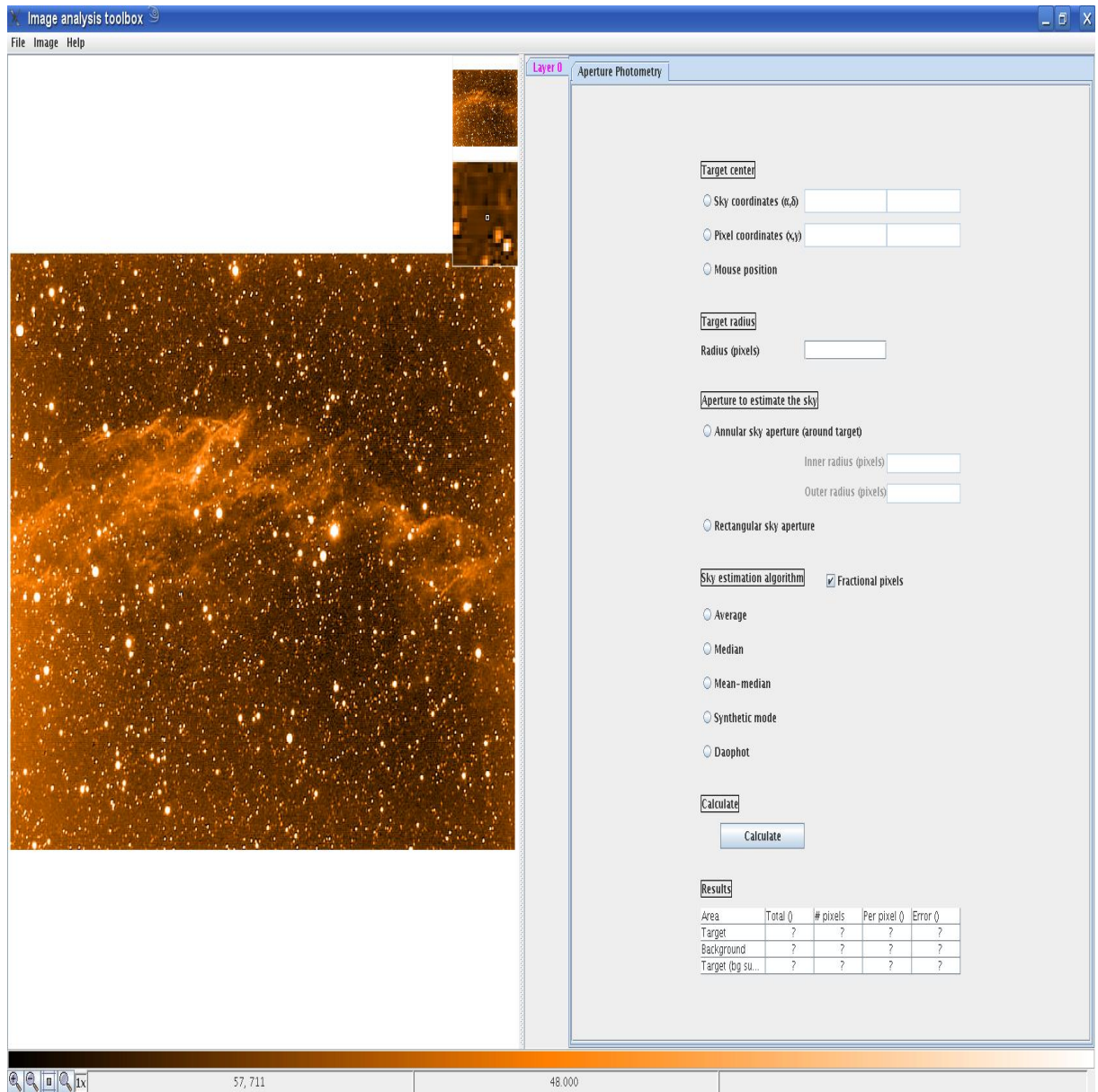


Figure 7.17.

First one must indicate how one wishes to denote the target center. This can be done by giving the sky coordinates (if available) or the pixel coordinates, or by a simple mouse click on the (active) image. For the circular target aperture one must give the radius in pixels. Further one can choose to use an annular sky aperture (which is centered around the target center) or a rectangular one. If one chooses to use an annular sky aperture, one must specify the inner and outer radii in pixels; the rectangular sky aperture can be drawn by pressing, dragging and releasing the mouse. Finally one also has to decide which algorithm to use to estimate the sky and whether to use fractional or entire pixels. After hitting the Calculate-button, the results are shown in a table in the right half of the toolbox.

In the figures below, an example is given for the two types of sky aperture.

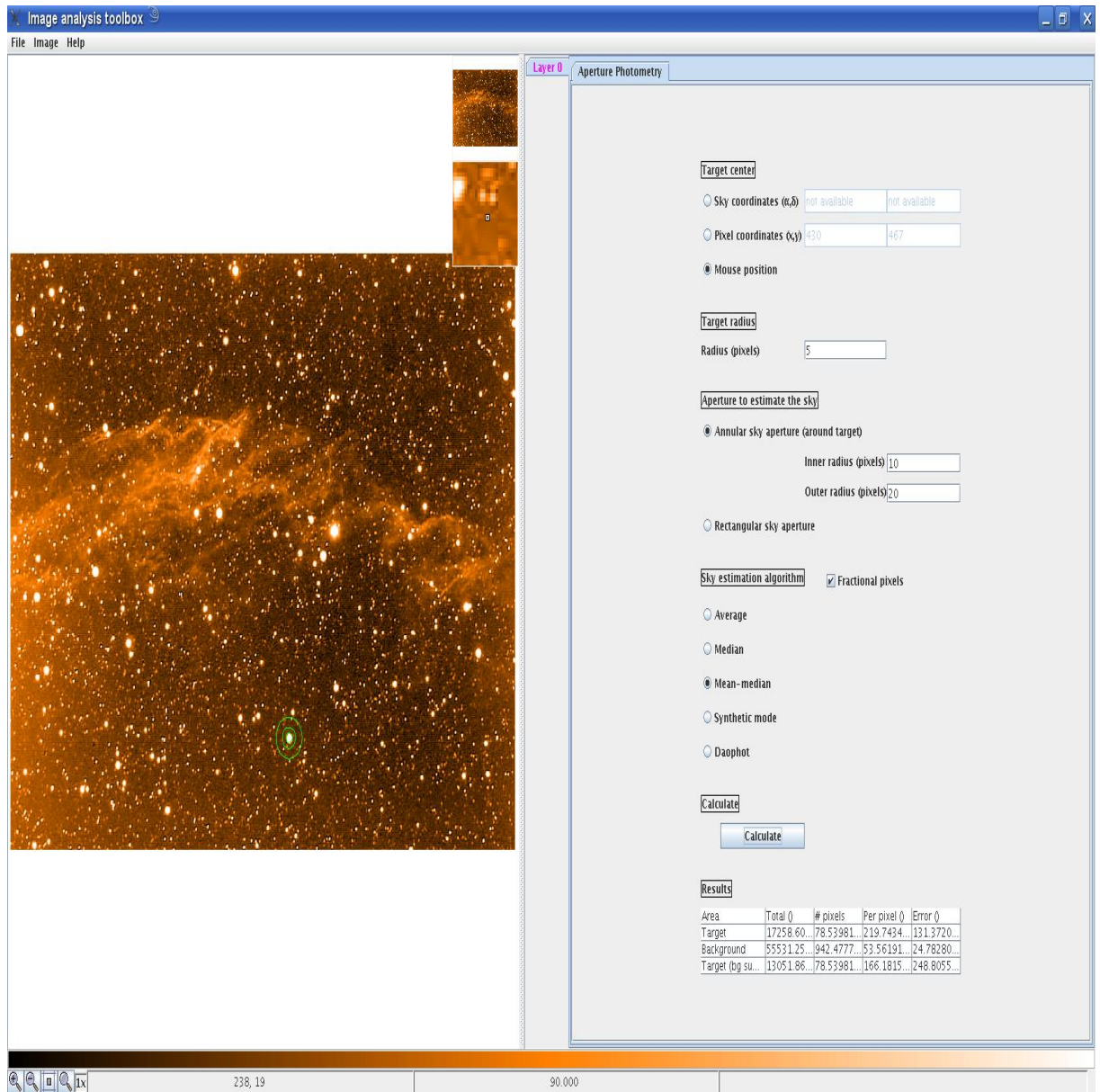


Figure 7.18. Aperture photometry with an annular sky aperture, performed in Herschel DP

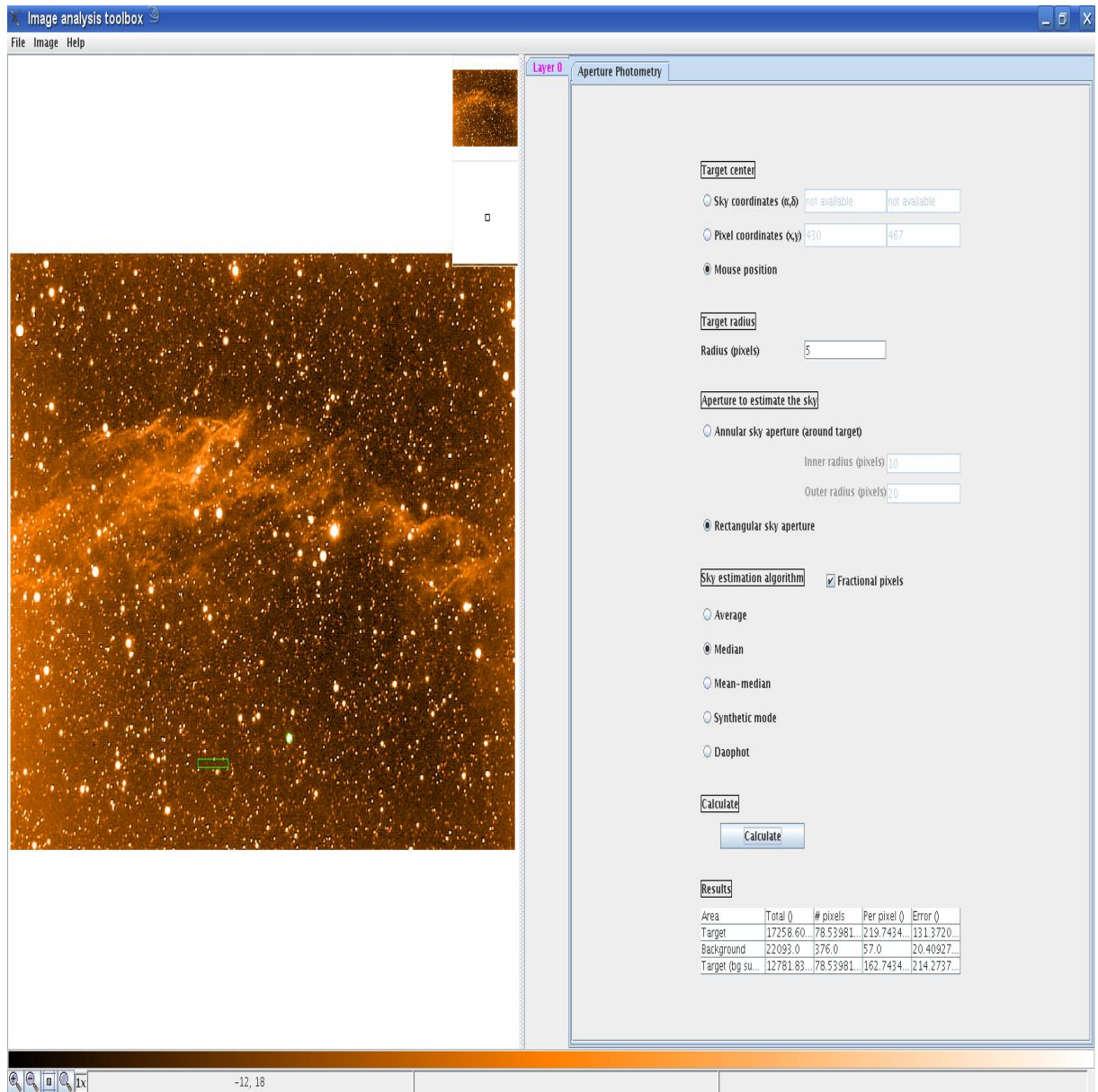


Figure 7.19. Aperture photometry with a rectangular sky aperture, performed in Herschel DP

In the columns of the results table, the total flux, the number of pixels, the flux per pixel and the root mean square (RMS) are given for the target plus background, the background and the target without the background respectively.

In both cases also the curve of growth is shown. This is a plot of the target flux as a function of the target radius. Such a plot can be used to see whether one has given a valid target radius. When one used an annular aperture to estimate the sky also a sky intensity plot is shown. This plot shows the intensity per sky pixel as a function of a varying inner radius (the outer radius is fixed).

Contour Plotting

Another functionality of the toolbox is contour plotting. A contour plot connects all points in the image with the same intensity, like isobars on a weather map.

One can start contour plotting by choosing the `Image > Contour Plotting` menu or by typing the following command

```
iat.contourPlotting()
```

Example 7.12. Starting contour plotting

This opens a new tab in the right half of the toolbox. In this tab, one must specify how to determine the contour values and the minimum length for contours to be drawn on the (active) image. An example is shown in the figure below.

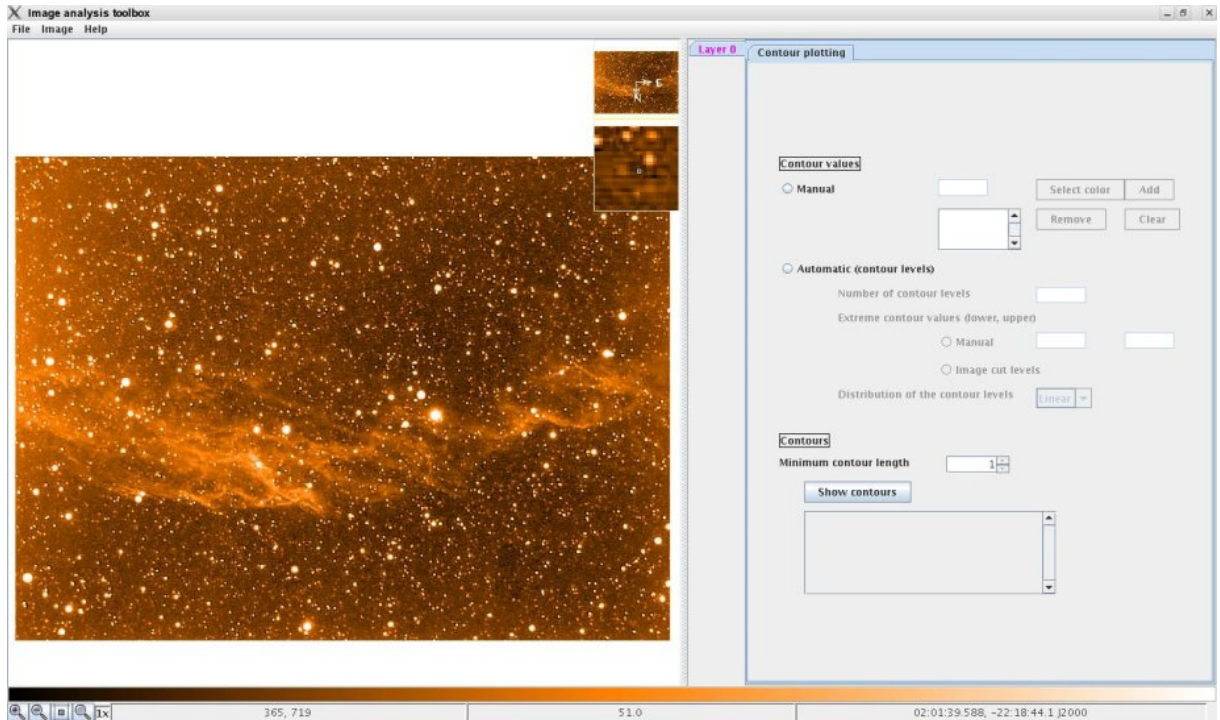


Figure 7.20. Starting contour plotting

When one decides to give the contour values one by one, one must enter a value, select a color and add the value to the list. This can be repeated as many times as one likes. One can clear this list at once, or delete one value at a time. The alternative is to let the contour values be determined automatically. In this case, one must specify the number of contour levels, the extreme contour values (one must give them manually or use the image cut levels) and the distribution of the contour levels (linear, logarithmic or ln). The contour colors will be generated automatically.

After hitting the Show contours-button, the contours will be drawn on the (active) image and a legend will be shown in the right half of the toolbox. This is illustrated in the figure below.

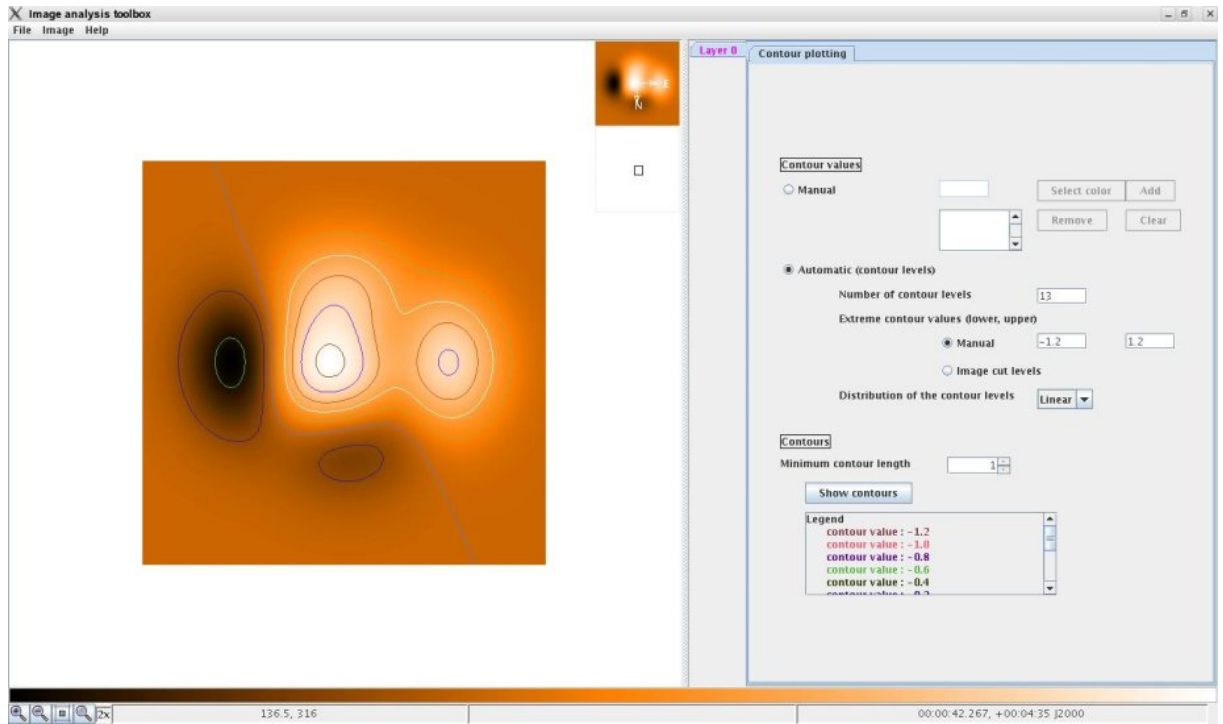


Figure 7.21. A contour plot, created in Herschel DP

Chapter 8. Introduction to Tasks

This chapter aims to be an introduction for users to the Task framework. Writing Tasks allows us to create modular and reusable code for data reduction and analysis, easier to distribute and to be used by people other than the author.



Warning

The Task behaviour has recently been changed. The old and new behaviours are still both available, and can be chosen by setting the `hcss.task.mode.newstyle` property to `true` (new behaviour) or `false` (old behaviour, the default). You can find it in the *Task* tab of the *progen* application, described as *Enable the new way of handling default values of TaskParameters*. Differences between the old and new behaviour will be highlighted throughout this chapter.

8.1. The Task framework

When we were talking about OOP in Chapter 3, we used as example a very real and tangible object like an airplane. However, we mentioned that objects can also represent more abstract concepts. Dealing with astronomical data presents us with such a situation. When reducing or otherwise treating our data we go through a succession of self-contained operations. Data enter each of these "boxes" in a certain state and exit in a modified state. We might want to have a general template to represent such boxes, with a way to specify input and output parameters and check for their consistency. It would also be great to have some form of history to track what we have been doing to a given set of data, without the need to write it in a separate place or try and squeeze the information in the file name. Another handy tool would be a command to get help on that particular "box", to know at a glance what it does and what kind of parameter it expects.

The `Task` framework provides it all. Here we can see many concepts of OOP in action: reusable code (that of the `Task` class) to create modular pieces of software (our tasks) easy to plug together into more complex structures. In the following sections we will learn how to write a Task in Jython.

8.2. My first Task

8.2.1. Before the Task

Before writing a Task we should have something to turn into a Task. Paste the following code into your JIDE upper panel and then load it into your session with the double arrow button at the top of JIDE.

```

#-----
# Average function
# Takes a TableDataset as input
# Returns a DoubleIcd (1D array of real numbers)
# in which each row is the average of the values
# in the input table columns
#-----
# Routine for calculating the average
def average(table):
    columns = table.columnCount
    divider = 1.0 / columns
    result = DoubleIcd(table.rowCount)
    for column in IntIcd.range(columns):
        result.add(table.getColumn(column).data)
    return result.multiply(divider)

# Routine for creating the initial table
def createTable():
    # Create array x (0.0, 1.0, 2.0, 3.0, 4.0)
    x = DoubleIcd.range(5)
    columns = 5
    # Create an empty table with a name
    table = TableDataset(description = "A test table")
    # Iterate for the the number of columns to fill up the table
    # Using " %i" % column " creates a string name for the
    # table-column which contains the integer value contained in
    # the variable name that appears after "%". In this case
    # column labels are just 0 1 2 3 4.
    for column in IntIcd.range(columns):
        table["%i" % column] = Column(x)
        x = x + 1
    # Return the result, a table called 'table'
    return table

# Routine for checking it out!!
def testAverage():
    # Create the table
    table = createTable()
    # Get the average and put it into an array called 'result'
    result = average(table)
    # Print the result (a 1D array)
    print 'Result:', result

```

Example 8.1. Before the Task

The above code has three functions in it. The important one is `average`, which does the "useful" bit of computation, giving the average of each column of a `TableDataset`. The `createTable` function simply creates the input `TableDataset` for `average`, while `testAverage` just calls the two functions above and prints out the result.

You can see how the above works by the following. The brackets indicate it is a function.

```
testAverage() # Result: [2.0,3.0,4.0,5.0,6.0]
```

8.2.2. What makes a Task?

In the current implementation, a task has two components:

- *Signature*. Someone's signature is something by which we can unambiguously identify that person (leaving forgery aside). In the same way a Task's signature, consisting of its name *and* the number and type of input parameters, is a way to identify the Task with no ambiguity.
- *Execution*. This component is made of three *methods*, i.e. object member functions. First we have the *preamble*, which checks the actual input parameter values. The *execute* method, as its name suggests, contains the algorithm performing the useful stuff. Finally, the *postamble* checks the output parameter values. The preamble, execute and postamble are empty by default (no input or

output parameters) and the developer usually writes only the execute method to perform a given algorithm.

Once parameters (input or output) receive a value, they are automatically reset to their default values after the Task has been executed. This is the **new style** behaviour. Note in particular that also *output* parameters are reset, so to keep a Task output for further inspection it has to be assigned to a variable upon execution, like this:

```
result = myTask()
```

When using the **old style** behaviour things are radically different. Once parameters have been given a value, this value remains available through successive executions (they become the defaults). In other words, input parameters need not to be given a value before each invocation of the execute method if that value is unchanged since the previous execution. If you want to optimise a script containing the execution method of a task within a loop, it is only useful to update those parameters where the value changes.

Something only available in the **new style** behaviour is the possibility to define new default values for Task parameters. If we have a `myInput` integer parameter for our `myTask` Task, we can set its new default value to 42 like this:

```
myTask.setAsDefault("myInput", 42)
```

Now equipped with this knowledge we can turn our average algorithm into a Task.

8.2.3. An Example of a Task: Average

To turn our average algorithm into a Task we need to wrap the algorithm into a suitable piece of code.

We will name the task itself `Average` (a Task is a class, it is callable from the command line, and generally class names are capitalised nouns). In our `Average` class we have no needs other than setting up a signature and calling the average function as part of its execution.

One change from our function to our class is that we will explicitly have two parameters in the class definition. One (in a similar way as the function) is our input table, but for the class we declare a second parameter to hold the result of computing the average. As a requirement, we would like to change our original average function as little as possible.

In the next paragraphs we explain (with code and comments) what packages are necessary to import, how to define the Task (creation code), the method to perform a function (execute) and how we use and test the Task (with different parameter access methods).

8.2.3.1. Importing definitions

For our given code we need to import definitions that are used by our task:

```
# Import task framework classes.
from herschel.ia.task.all import * # ❶
```

Some explanation about the import:

- ❶ Here we import all the task framework classes we need. Task and TaskParameter classes will be automatically imported with the `all` import statement.

Note that the preferred way to import the needed classes from the task framework is the so called 'all' import statement:

```
from herschel.ia.task.all import *
```

8.2.3.2. Creation

First the code for the creation method called `__init__` in python:

```
class Average(JTask): # ❶
# Creation method
def __init__(self,name = "averageTable"): # ❷
    p = TaskParameter("table",valueType = TableDataset, mandatory = 1) # ❸
    self.addTaskParameter(p) # ❹
    p = TaskParameter("result",valueType = DoubleIcd, type = OUT) # ❺
    self.addTaskParameter(p)
```

And some explanations about the code...

- ❶ Here we define a class `Average` which has `JTask` as a parent class. In other words, `Average` inherits from `JTask`. Note that `Jtask` is a python file and has no `JavaDoc` therefore.
- ❷ This line declares the creation method used by any instance of the `Average` class. `self` as the first argument represents the instance that we are currently working on. The `name` argument is the default value indicated (which the user can of course overwrite).

The rest of the code is the definition of the signature for the task `Average` and is as follows:

- ❸ This line creates a parameter whose name is `table`, data type is `TableDataset`. This is a mandatory parameter, i.e. an input parameter which must have a value before the algorithm is performed. The preamble will verify that the user has set a value for this parameter and will eventually warn the user that the execution of the task cannot take place.
- ❹ Here we add the parameter to the signature of this task.
- ❺ We proceed in a similar way for our second parameter (as mentioned above) which will hold the result of our computation. The only difference for the second parameter is the `type = OUT` statement which means that this parameter will hold an output value. As a side note the mode of parameters can be `IN`, `OUT` or `IO` (both input and output), the default being `IN`.

8.2.3.3. Execution

First we examine the code for the execution method called `execute` as predefined in the `JTask` base class. This simply follows on from the previous set of code that initiated the task and should be added to the end of it:

```
# Execute method itself
def execute(self): # ❶
    self.result = average(self.table) # ❷
```

- ❶ This is a declaration stating that we define the method `execute`. Actually we redefine the empty `execute` method of `JTask`. This method has a parameter `self` which refers to the task we are currently working with, rather than to any other parts of the current IA session.
- ❷ This line means 'take this instance `table` value, perform the average operation on it and deliver the result to this instance `result`'. So in one line we perform the whole operation using our own actual parameters.

Together with the signature defined in the previous section we have set up our Task. The complete script should look like the Task `Average` (below). We now load this into our session.

```
# File: Average.py
#=====
# Import task framework classes.
from herschel.ia.task.all import *
from herschel.ia.task.JTask import JTask

class Average(JTask):
    #Creation method
    def __init__(self,name = "averageTable"):
        #
        p = TaskParameter("table",valueType = TableDataset, mandatory = 1)
        self.addTaskParameter(p)
        p = TaskParameter("result",valueType = DoubleId, type = OUT)
        self.addTaskParameter(p)
        # Execute method itself does the running of 'average'
    def execute(self):
        self.result = average(self.table)
```

Example 8.2. The Average Task

8.2.3.4. Usage

Below is the command line code to input into JIDE bottom left panel for testing our Average task. First we *instantiate* the Average class creating an object called avg:

```
avg = Average()
```

We are using the default name of averageTable for our Task. To change the name we would have written for instance `avg = Average("Simple average of table data set")` or `avg = Average(name = "Mine")`.

We can now formulate a table using the createTable routine in the set of three functions we created at the outset.

```
table = createTable()
```

The interesting part comes when we use the following:

```
print avg(table)
```

We have executed the Task and printed its result. To make sure that it indeed executed successfully, we can look at the statusMessage:

```
print avg.statusMessage
```

A more direct way to execute our Task would be

```
print avg(createTable())
```

On the other hand, we could do everything in a long-hand fashion, doing one little step at a time:

```
avg.table = table
avg()
result = avg.result
print result
```

Here we tell our average task that its input is called 'table'. The second line runs the task itself and we assign the result from this to a variable called 'result' in the third line. Finally, this result is printed.

8.2.3.5. Getting help on Tasks

If you stumble upon a task you have never used before you will probably want some way of finding out about its parameters, whether they are mandatory or not, and so on. Taking our Average task as example, if you type

```
info('Average') # Note it's 'Average' with single quotes
```

you will be greeted by the following window:



Figure 8.1. Getting help on a Task.

It may appear fairly intimidating, but it provides a lot of useful information to users once they get past the initial shock. In particular, look at the sections called `Inputs:` and `Outputs:`. They list the input and output parameters, which are most of what is needed in order to use a Task. In particular, here we see that we have one input parameter called `table`, that it's a `TableDataset` and is mandatory (`Optional: false`). Similarly, we see that the Task will output a single `Double1d`. The information about `status`, `statusMessage`, `progress` and `views`, found in the lower part of the help window (not shown in the picture) is of limited interest to users.

What appears in the help window also depends on what developers originally put into the Task. For example, in our case we have the hardly reassuring `Task: null` and `Name: null` messages at the very top of the window. But if we give a name to our Task like this

```
avg.setName("My first Task")
```

we will see that after a short while the new information will appear in the help window.

8.2.3.6. Adaptations in the Preamble to a Script

The adaptation to the input of our Average script can be made in a preamble to the task, such as in the following script. Note that here we import the `task` classes one by one, just to show in detail what is needed.

```

# Importing JTask classes
from herschel.ia.task.all import *
# Other needed imports
from org.python.core import PyList
# And here is our AdaptAverage class
class AdaptAverage(JTask):
    # Creation method
    def __init__(self,name = "Running Average"):
        p = TaskParameter("vector1",valueType = PyList, mandatory = 1)
        self.addTaskParameter(p)
        p = TaskParameter("vector2",valueType = PyList, mandatory = 1)
        self.addTaskParameter(p)
        p = TaskParameter("result",valueType = DoubleId,type \
            = OUT)
        self.addTaskParameter(p)
        # Create an internal JTask variable 'table' which is our table data set
        self.__dict__['table'] = TableDataset()
    # In the preamble we do the adaptation from 2 vectors to one table
    def preamble(self):
        JTask.preamble(self)
        self.table["0"] = Column(DoubleId(self.vector1))
        self.table["1"] = Column(DoubleId(self.vector2))
    # Execute method itself
    def execute(self):
        self.result = average(self.table)

```

Example 8.3. The Adapt Average Task

In this example, the `from org.python.core import PyList` statement allows us to work with Python array lists (vectors). The task now takes two Python arrays and produces a table from the arrays with each array forming a column of the table. We then can run our average script on the table created in the preamble.

An internal instance variable is declared in the creation method with the statement: `self.__dict__['table'] = TableDataset()`.

Rewriting the preamble method. One should note that we first invoke the preamble from our parent task (JTask) to guarantee that our needed parameters do have a suitable value before putting them into the table.

The following short script can be used to test this adapted version of our averaging routine.

```

def test():
    sample1 = [1.0, 2.0, 3.0, 4.0]
    sample2 = [3.0, 4.0, 5.0, 6.0]
    avg = AdaptAverage()
    # Invocation using positional parameter
    print 'Result:', avg(sample1,sample2)

```

Input of the following command

```
test()
```

provides the following printed result

```
Result: [2.0,3.0,4.0,5.0]
```

8.2.3.7. Positional and Keyword Arguments in Tasks



Note

It should be noted that *positional* or *keyword* arguments can be used with tasks but NOT a mix of the two.

For example, the last line of our 'test' script effectively runs the following (try replacing the last line of the test() routine):


```
# Positional arguments
print 'Result:', AdaptAverage()(sample1, sample2)
# Keyword arguments
print 'Result:', AdaptAverage()(vector1=sample1, vector2=sample2)
# Since 'vector1' and 'vector2' are the two arguments for the
# AdaptAverage task.
```

Mixing of the two modes is ONLY allowed following all positional arguments. For example:

```
print 'Result:', AdaptAverage()(sample1, vector2=sample2)
```

But once keyword arguments start to be used then they must continue to be used. For example the following code snippet will result in a compiling error when added to the 'test' program and recompiled.

```
print 'Result:', AdaptAverage()(vector1 = sample1, sample2)
# If this is added to 'test' and "test" is then recompiled we get the
# following syntax error.
# SyntaxError: ('non-keyword argument following keyword',
# ('<string>', 6, 49, ''))
```

A similar syntax error occurs if the AdaptAverage() task was run on a single line outside of the 'test' routine.

8.2.3.8. The Transformer example

Yet another JTask example. This one takes an array and transforms it into the first column of a TableDataset. As before, the code comes with a testTran() function to check what the Task does.

```
from herschel.ia.task.all import *
from org.python.core import PyList

class Transformer(JTask):
    # Creation method
    def __init__(self, name = 'Vector Transformer'):
        p = TaskParameter(name = "input", valueType = array(Integer), mandatory = 1)
        self.addTaskParameter(p)
        p = TaskParameter( name = "result", valueType = TableDataset)
        p.type = OUT
        self.addTaskParameter(p)
    # Execute method
    def execute(self):
        self.result = TableDataset(description = 'Integrated vector as column zero')
        r = Double1d(len(self.input))
        index = 0
        for data in (self.input):
            r[index] = data
            index = index + 1
        self.result['0'] = Column(r)

def testTran():
    sample = [10, 20, 30, 40]
    # Turn it into a table data set
    transform = Transformer()
    table = transform(sample)
    print "Printing the table"
    print table
    print "Printing the first column of the table"
    print table['0']
    print "Printing just the data in the first column"
    print table['0'].data
```

Example 8.4. The Transformer Task

8.3. Guideline on How to Work With GUIs Within Tasks

This section describes how to handle GUI's and/or a dialog related to a task, how to check whether a certain task supports the use of a dialog and/or GUI, as well as describing how to apply them. *It should be emphasised that the developer of a task needs to implement a dialog or GUI in the task. This section simply provides guidance to the user for using tasks that have dialog or GUIs included within them.*

8.3.1. The use of task parameters handled via a dialog

In the case where a task includes a long or complex set of parameters a dedicated dialog can be provided by the original developer of the task. Such a component is handled by a boolean parameter called "dialog" which the user can invoke using

```
result = Task()(dialog=1)
```

Such a call results in a pop-up window which can be completed by selecting for example the "accept" button, which will close the GUI.

Note that all tasks in the future will include a boolean-parameter called "dialog". In cases where all the available input parameters are of the type String or Number (i.e. those the framework can handle for setting up a dialog) a dialog-popup will be provided, otherwise an exception is thrown.

8.3.2. The use of more enhanced GUIs

In case you have a more complex task or you want to re-execute a task several times using different inputs, a GUI might be introduced. Such a component is handled by a boolean parameter called "gui":

```
task = MyTask()
task.gui = 1 # gui interaction might include an task.execute()
result = task.result # another gui interaction
result2 = task.result
```

Such a command sequence is very useful as it increases transparency. For example, the GUI might show the state of the parameters by including a field for each parameter and a plot or image representing the quality of the resulting output.

To summarise: the user of a task applies its views by the use of related the booleans (task parameters). In case of a one-time user interaction such a boolean is called "dialog" and otherwise it is called "gui". Note that in case more GUI components are involved additional booleans could be introduced, the task specific documentation should include this info.

8.3.3. Example Task Handled by a Dialog

The following provides an example interaction between a user (USR) and the system (HCSS) for the use of a task "dialog".

USR: Asks to set up parameters of a task via dialog: result = MyTask()(dialog=1)

HCSS: looks for the default dialog provided by the task developer

a) dialog is found and displayed

b) dialog is not found in which case the framework (ia.task) tries to provide the user with an automatic dialog for the task signature

HCSS: display the dialog

USR: set/adjust parameter values AND approve those (for example, by selecting an "accept" button)

HCSS: close the dialog, run the task, return to the command line

Justification:

The user is given the possibility to setup the tasks signature via a GUI which is launched on his request.

Note: in case b) fails it will notify the user that a dialog cannot be provided by the framework and was not previously defined by the task developer

8.3.4. Example Task Controlled by a GUI

In this case we have a task that can be controlled via a GUI. The following shows a typical use case for a user (USR) interaction with the system (HCSS).

USR: Asks to run a tasks via a GUI:

```
mytask = MyTask()
```

```
mytask.gui = 1
```

HCSS: display the GUI interface provided by the task developer

USR: (possibly) insert parameter values

USR: execute the task (for example by selecting the "execute" button)

HCSS: run the task, update GUI to (possibly) show result in a plot of image or text field

USR: retrieve data within jide by calling:

```
result = task.result
```

USR: possible further analysis of result in jide session

USR: repeat steps 3 to 7 to compare results using diff. parameter settings, or close the GUI

Justification:

The GUI can provide more functionality: setup signature, allow task to execute, see results in a image/plot. The user is able to retrieve the task output -- for further analysis in DP -- as described above, i.e. the result can be fed back into jide by requesting "res1 = mytask.result". In this scenario the GUI lives next to JIDE.

Chapter 9. Other DP Packages: What is Available?

9.1. Introduction

To use the various packages within HCSS the user needs to import them into the HCSS session. This can be done automatically using the `import.py` file (see ???), editable by the user, for packages that are used frequently. Whether in the `import.py` or via a JIDE command line, all packages are imported via command lines of the type

```
from herschel.ia.numeric import *
```

There are several packages available within the HCSS. In this chapter we provide an overview of the main DP packages only. There are also a number of external library sets that are imported into DP when it is initiated (*these will be described in a later update to the manual.*) A full listing of classes (programs) available in the HCSS system is given in <ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/releases/doc/api/index.html>.

A number of DP packages have already been discussed in some detail. The *DP numeric* package was discussed in Chapter 4, the *DP plot* package in Chapter 6 and the *DP display* package is described in Chapter 7. Illustrations of how to use parts of several other HCSS packages are also shown in earlier chapters.

The contents of these sub-packages are also briefly described in this chapter.

9.2. Overview of JavaDocs Documentation for DP Packages

The javadoc is normally started up as three frames in a web browser as illustrated in Figure 9.1 The upper left frame contains the *packages index* which is a clickable list of all packages in the system. The title in that frame represents the HCSS build number for which this documentation is valid. The lower left frame contains the *classes index* which is a clickable list of all classes. The selection of classes shown in this frame depends on the package that was selected in the packages index frame. The *Main frame* contains overview information on the system and packages or shows the javadoc for a selected class.

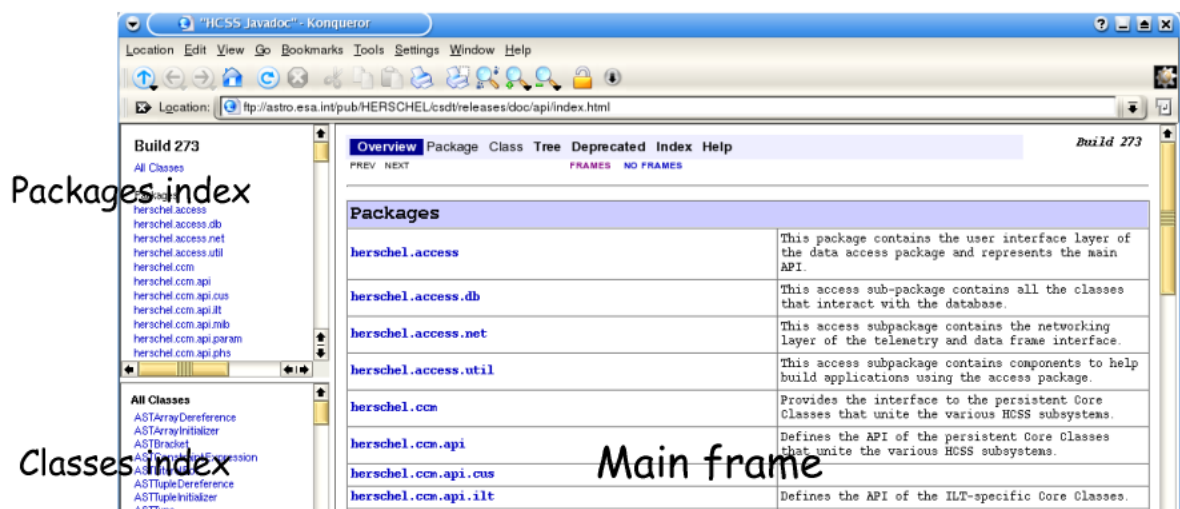


Figure 9.1. Web browser page of JavaDocs top level frame.

Click in the *Packages index* frame to select a package and update the *Classes index* frame to show those classes for the selected package. Click the *Classes index* frame to show the javadoc of a particular class in the *Main frame*.

The *Main frame* contains a kind of navigation bar at the top where the view in this frame can be selected. The figure above shows the overview of all the packages. Other views are: Package, Class, Tree, Deprecated, Index, and Help. These views will be explained in more detail below. In the overview the Package and Class views are disabled, they become available when a package or class is selected. Figure 9.2 shows the slightly expanded navigation bar for the Class view.



Figure 9.2. Navigation bar on the *class view* of JavaDocs.

Note that the navigation bar provides the possibility to browse through packages and classes with NEXT and PREVIOUS and provides direct access to the specific parts of the class documentation e.g. constructors (start class/program) or methods (which can be thought of as sub-routine components of programs that can be applied). It is also possible to switch between FRAMES and NO FRAMES. With NO FRAMES only the *Main frame* of the javadoc will be shown and index frames become unavailable.

9.3. Package view

Each package has a page that contains a list of its classes and interfaces, with a summary for each. This page can contain four categories: *Interfaces summary*, *Classes summary*, *Exceptions and Error summary*. Not all categories are always present. At the end there is the package description and possible links to specific and/or related documentation.

Figure 9.3 shows the `herschel.ia.dataset` package which contains a number of interface and classes e.g. *Dataset* and *TableDataset*. You can see that the *Classes index* frame provides a clear separation of interfaces and classes and the *Main frame* shows the interface and class summaries and provides a brief package description with links to package specific info at the bottom (The image of the *Main frame* has been manipulated to shows the categories available without too much cluttering the picture). You can navigate to the interface and class detailed documentation by clicking the names in the summary tables or in the *Classes index* frame.

Location Edit View Go Bookmarks Tools Settings Window Help

Location: http://localhost/~rik/hcss-current-docs/api/index.html

"HCSS Javadoc" How to read javadoc

herschel.ia.cal.api
herschel.ia.cal.impl
herschel.ia.classloader
herschel.ia.classloader.api
herschel.ia.classloader.impl
herschel.ia.dataset
herschel.ia.dataset.demo
herschel.ia.demo
herschel.ia.dfm
herschel.ia.doc
herschel.ia.framework
herschel.ia.framework.api
herschel.ia.framework.example
herschel.ia.framework.example

herschel.ia.dataset

Interfaces
Algorithm
Annotatable
Attributable
Composite
Dataset
DatasetVisitor
DataWrapper
History
NumericParameter
Parameter
ParameterVisitor
Quantifiable

Classes
AbstractDatasetAndDataVisitor
AbstractDatasetVisitor
ArrayDataset
BooleanParameter
Column
CompositeDataset
DatasetUtil
DateParameter
DoubleParameter
LongParameter
MetaData
Product
StringParameter
TableDataset

Overview Package Class Tree Deprecated Index Help *Build 272*

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package `herschel.ia.dataset`

This package provides a uniform approach for holding, annotating, quantifying and attributing data as is defined in the `herschel.ia.numeric` package.

See: [Description](#)

Interface Summary

Algorithm	Interface for backwards compatibility.
Annotatable	An Annotatable object is an object that can give a human readable description of itself.
Attributable	An Attributable object is an object that has the notion of meta data .
Composite	A container of named Datasets.
Dataset	Attributable and annotatable information container to which one can apply an Algorithm, and it can be part of a Product .

Class Summary

AbstractDatasetAndDataVisitor	Abstract implementation of DatasetVistor and DataVisitor interface.
AbstractDatasetVisitor	Abstract implementation of DatasetVistor interface.
ArrayDataset	Special dataset that contains a single Data object.
BooleanParameter	A Parameter with a Boolean value.
Column	A Column is a the vertical cut of a table for which all cells have the same signature.
CompositeDataset	An CompositeDataset is a Dataset that contains zero or more named Datasets.
DatasetUtil	Temporary placeholder for some utilities.
DateParameter	A Parameter with a Date value.
DoubleParameter	A Parameter with a Double value.
LongParameter	A Parameter with a Long value.
MetaData	A container of named Parameters.
Product	A Product is a generic result that can be passed on between (standalone) processes.

Package `herschel.ia.dataset` Description

This package provides a uniform approach for holding, annotating, quantifying and attributing data as is defined in the `herschel.ia.numeric` package. Actual storage is taken care of by the `herschel.ia.io` package.

Introduction

For a general discussion of this package, please read the [top documentation](#).

Note that this package is a library, not an application. One could argue that any user of this library is a code developer. This is partially true, as the code will be used by end-users as well. Most end-users however will access the various elements in `Jython` scripting language. As a result the design of this package is geared toward usage in `Jython` as well as in `Java`.

Basic elements

This package can be split into the following categories:

Products	
Product	Collection of datasets, meta-data and the history of the product
Datsets & Algorithm	

Figure 9.3. Package description page in JavaDocs.

9.4. Class view

Each class and interface has its own separate page in the *Main frame*. Each of these pages has three sections consisting of a class/interface description, summary tables for constructors and methods, and detailed descriptions of constructors, methods and attributes. The information shown in the class view is restricted to the public *API (Application Programming Interface)*.

Each summary entry contains the first sentence from the detailed description for that item. The summary entries are alphabetical, while the detailed descriptions are in the order they appear in the source code. This preserves the logical groupings established by the programmer.

Figure 9.4 is taken from the *Main frame* of the *TableDataset* class and shows the class description together with its hierarchy. You can see that the *TableDataset* implements a number of interfaces and also has one known sub-class i.e. *SpectrumDataset*. The second part of the figure shows a more detailed description of the class usage. This description is provided by the programmer in the source code.

herchel.ia.dataset

Class TableDataset

```
java.lang.Object
├─ herchel.ia.dataset.AbstractAnnotatable
│   └─ herchel.ia.dataset.TableDataset
```

All Implemented Interfaces:

[Annotatable](#), [Attributable](#), [Dataset](#)

Direct Known Subclasses:

[SpectrumDataset](#)

```
public class TableDataset
extends herchel.ia.dataset.AbstractAnnotatable
implements Dataset
```

A *TableDataset* is a tabular collection of [Columns](#). It is optimized to work on array *Data* as specified in the *herchel.ia.numeric* package.

This approach is convenient in many cases. For example, one has an event list, and each algorithm is adding a new field to the events (i.e. a new column).

The orthogonal approach (adding rows) is therefore expensive and therefore currently no mechanism is implemented to add rows to the table.

Jython usage:

creation:

```
creation:
$ x=TableDataset(description="This is my table")
$ x["Time"]=Column(data=time, quantity=SECONDS)
$ x["Energy"]=Column(data=energy, quantity=ELECTRON_VOLTS)
```

Figure 9.4. The class view of *TableDataset* showing a brief description and a short example of its usage.

Scrolling down in the *Main frame* brings you to the summary section which is shown in Figure 9.5. The constructor summary shows all public constructors for this class with their specific argument list. To see detailed information on the constructor click the name of the constructor that you need. Constructors are methods that create objects of a particular type. The code example in the description section above shows you how to create a *TableDataset* on the *jython* command line.

Constructor Summary	
<code>TableDataset()</code>	Constructs an empty table.
<code>TableDataset(java.lang.String description)</code>	Constructs a <code>TableDataset</code> with a description.
<code>TableDataset(TableDataset copy)</code>	Constructs a <code>TableDataset</code> that is a deep copy of specified argument.

Method Summary	
<code>Column</code>	<code>__getitem__(int index)</code> Jython only(!) wrapper for abbreviated access to a column by index.
<code>Column</code>	<code>__getitem__(java.lang.String key)</code> Jython only(!) wrapper for abbreviated access to a column by name.
<code>void</code>	<code>__setitem__(int index, Column value)</code> Jython only(!) wrapper for abbreviated replacement of a column by index.
<code>void</code>	<code>__setitem__(java.lang.String key, Column value)</code> Jython only(!) wrapper for abbreviated addition/replacement of a column by name.
<code>void</code>	<code>accept(DatasetVisitor visitor)</code> Accepts a visitor of this Dataset.
<code>void</code>	<code>add(Column column)</code> Deprecated. and replaced by <code>addColumn(herschel.ia.dataset.Column)</code> .
<code>void</code>	<code>add(java.lang.String name, Column column)</code> Deprecated. and replaced by <code>addColumn(herschel.ia.dataset.Column)</code> .
<code>void</code>	<code>addColumn(Column column)</code> Adds the specified column to this table, and creates a dummy name for this column, such that it can be accessed by <code>get(int)</code> .
<code>void</code>	<code>addColumn(java.lang.String name, Column column)</code> Adds the specified column to this table, and attaches a name to it.
<code>void</code>	<code>addRow(java.lang.Object[] array)</code> Adds the specified array as a new row to this table.
<code>Dataset</code>	<code>apply(Algorithm algorithm)</code> Applies the specified algorithm on a dataset.
<code>protected java.lang.String</code>	<code>contentsToString()</code>

Figure 9.5. Page showing the constructor mechanism (how to create a `TableDataset`) and the associated set of methods (what you can do with the `TableDataset` you created).

The method summary shows all public methods for this class in alphabetical order. For detailed information on a specific method, click its name. In this method summary there are a number of things to note. The return values of the methods are in the left column while the method signature and a summary line is in the right column. The summary line can be preceded with a **deprecation** note. Deprecation means that this method should not be used anymore because it is marked to be removed from future releases. The deprecation comment normally provides the alternate or new method to be used instead. An overview of all deprecated methods in the whole system is available from the navigation bar at the top of the *Main frame*.

Sometimes method names can start and end with two underscore characters like in `__getitem__` above. These methods are special constructs which allow you to use the specific jython syntax to access and manipulate objects from this class.

9.5. Tree view

There is a Class Hierarchy page for all packages, plus a hierarchy for each package. Each hierarchy page contains a list of classes and a list of interfaces. The classes are organised by inheritance structure starting with *java.lang.Object*. The interfaces do not inherit from *java.lang.Object*. When viewing the Overview page, clicking on "Tree" displays the hierarchy for all packages. When viewing a particular package, class or interface page, clicking "Tree" displays the hierarchy for only that package.

9.6. Deprecated view

The Deprecated API page lists all of the API that have been deprecated. A deprecated API is not recommended for use, generally due to improvements, and a replacement API is usually suggested.



Warning

Be warned that deprecated APIs may be removed in future implementations.

9.7. Index view

The Index contains an alphabetic list of all classes, interfaces, constructors, methods, and fields.

9.8. DP Packages And Documentation

The following short paragraphs outline the packages currently available within the Herschel DP system. A full listing of the classes (programs) available in these packages is provided in Javadoc form at <ftp://ftp.rssd.esa.int/pub/HERSCHEL/csd/releases/doc/api/index.html>.

9.8.1. *herschel.ia.dataflow*

herschel.ia.dataflow - a package for handling processing threads. Particularly useful for Quick Look Analysis (QLA) and Standard Product Generation (SPG). It can be used in interactive sessions too. Allows the user to connect scripts from process modules as is typically required for a set of data reduction steps. Dataflow also supports event-based processing as well as threads.

Sub-packages:

herschel.ia.dataflow.data.process ...classes for handling the processes used in a dataflow session.

herschel.ia.dataflow.example.indicator_control.monothread ...classes used to illustrate the control of a dataflow.

herschel.ia.dataflow.example.indicator_control.multithread ...ditto but for multiple threads.

herschel.ia.dataflow.template ...class to allow template dataflow to be created.

herschel.ia.dataflow.util ...contains a class for identifying dataflows.

9.8.2. *herschel.ia.dataset*

herschel.ia.dataset - a package for dealing with *TableDataset*, *ArrayDatasets* and *CompositeDatasets*. These datasets contain information to which an algorithm can be applied. The

package contains classes that deal with the set and handling of these datasets and also the handling of products (which can contain multiple datasets). An example product may be one that contains several tables plus metadata that describes the table contents which might have similarities to FITS header information.

Sub-packages:

herchel.ia.dataset.demo - contains classes that demonstrate the use of datasets and construct a user-defined *SpectrumDataset*.

9.8.3. herchel.ia.demo

herchel.ia.demo - package containing classes for use in a DP demo of end-to-end processing. See sub-package *herchel.ia.demo.endtoend* and demo script.

9.8.4. herchel.ia.doc

herchel.ia.doc - currently a place holder for a documentation package.

9.8.5. herchel.ia.document

herchel.ia.document - tools to generate documentation of dynamic as well as static DocBook documents in different formats: PDF, HTML, JHelp.

9.8.6. herchel.ia.help

This package contains the utilities and classes needed for providing the help facilities in an DP/JIDE session. Access to the on-line help is discussed in Chapter 3 of this manual.

9.8.7. herchel.ia.image

herchel.ia.image - package containing classes for handling images. The Display capabilities from this package were discussed in Chapter 8. The following classes exist in the package.

- *Display* - an image display implementation based on JSKY. User gets 800x600 window for image. Can handle, 1D, 2D and 3D image representations. Allows standard display capabilities such as annotation, rescaling, coordinate display.
- *Histogram* - currently a basic histogram capability. The histogram is based on the values taken from an *ImageDataset*.
- *ImageDataset* - a special form of a composite dataset that presents an image. Has layers which are image data, mask data, error data. World Coordinate System (WCS) information is held as metadata in the *ImageDataset*.
- *Layer* - constructs a layer of an *ImageDataset*.
- *Rotate* - allows rotation of an *ImageDataset*. Four different types of interpolation are possible. The WCS coordinates of the image are also rotated with the image.
- *Scale* - allows the scale of an image to be changed. Four different types of interpolation are possible.
- *Translate* - moves an *ImageDataset*. The WCS is also adapted.
- *WCS* - associates a World Coordinate System to an *ImageDataset*

Sub-package:

herschel.ia.image.gui - classes that handle GUIs. These should ONLY be called from within the Display program.

9.8.8. *herschel.ia.inspector*

This package contains the classes and utilities for providing the dataset and session inspectors available in JIDE (see Section 2.3.5).

9.8.9. *herschel.ia.io*

herschel.ia.io - This is a package that provides a means of accessing local archives where Products can be saved or loaded from. Products are combinations of data and information and can be likened to the contents of a single FITS file.

Sub-packages:

herschel.ia.io.fits - A FITS implementation that can write Products to a FITS file and read such FITS files back into the system. Allows the production of a FITS archive.

herschel.ia.io.ascii - Allows the input/output to and from ASCII files from within the DP environment.

herschel.ia.io.dbase - Allows data/products to be put into objects that can be stored in databases (Versant databases are currently available for use with the HCSS). See Chapter 12 for information about the setup and use of databases with DP.

9.8.10. *herschel.ia.jconsole*

herschel.ia.jconsole - Package containing the classes used in running JIDE, a GUI for running/editing of DP/Jython scripts. Allows control of the JIDE setup and access to classes that setup the components of the GUI interface (in *herschel.ia.jconsole.gui*).

9.8.11. *herschel.ia.numeric*

herschel.ia.numeric - This package is discussed in some detail in Chapter 4 and Chapter 5

Sub-packages:

herschel.ia.numeric.toolbox - Provides a large set of numeric classes available within Herschel DP. These include mathematical functions (trigonometric functions, polynomials), Fourier transforms, fitter functions, interpolation and matrix functions. **NOTE: toolbox classes are automatically loaded when starting DP.**

herschel.ia.numeric.toolbox.basic - Provides the classes that allow basic mathematical manipulation of numeric arrays, e.g., trigonometric functions, mathematical product, variance etc.

herschel.ia.numeric.toolbox.filter - Provides the classes `BoxCarFilter`, `Convolution` and `GaussianFilter`.

herschel.ia.numeric.toolbox.fit - Provides the classes that allow the fitting of data with numerous models (iterative fitters, sine model fitters, polynomial model fitters etc.).

herschel.ia.numeric.toolbox.interp - Provides the classes that allow the interpolation of data. These include `CubicSplineInterpolator`, `Interpolator` (a general interpolator), `LinearInterpolator` and `NearestNeighborInterpolator`.

herschel.ia.numeric.toolbox.matrix - Provides the classes that allow the manipulation of `Double2d` arrays holding matrices. It includes the classes `MatrixDeterminant`, `MatrixInverse`, `MatrixSolve`.

herschel.ia.numeric.toolbox.util - Provides the single class `MoreMath` which has methods for mathematical manipulation of single numerical elements (integers, doubles, bytes etc.).

herschel.ia.numeric.toolbox.xform - Provides the classes `FFT`, `Hamming` and `Hanning` for Fourier transforms and `Hanning/Hamming` smoothing of data.

9.8.12. **herschel.ia.plot**

herschel.ia.plot - This package provides access to the DP plotting utilities available with DP (callable from JIDE). This includes `PlotXY` and access to plot properties. The use of the plotting capabilities in Herschel DP is discussed in Chapter 6.

9.8.13. **herschel.ia.task**

herschel.ia.task - This package provides the tools needed to create a DP "task" which a user can then incorporate when constructing his/her own DP software package. This can be used by a user to set up a script which has an associated "signature" (parameter setup). In setting up a task, parameter checks can be performed and a history of the processing can be made.

This package is discussed in Chapter 8.

9.8.14. **herschel.ia.ui**

herschel.ia.ui - Provides the programs dealing with the GUI interfaces available within Herschel DP. The setup and use of GUIs and incorporation of Java.

Chapter 10. IO of DP Variables, Tabular ASCII and FITS Files

10.1. Introduction

This chapter describes how to save to disk and restore DP variables and how to read and write tabular ASCII and FITS data files within DP. Illustrations are provided that run in the jide environment.

10.2. Saving and Restoring DP Variables

Some or all of your DP variables can be saved to disk and restored later in the same session, or even a different session. DP variables types that can be saved are:

- simple scalar values, lists and strings (1, [1,2,3], "a string")
- numeric arrays (Int1d, ... Complex3d)
- datasets (TableDataset, ArrayDataset, CompositeDataset)
- products (Product) - which contain one or more datasets, a history of how they were created and meta-data describing their contents.

These can be saved from and brought back into a DP session using the `save` and `restore` commands. This is illustrated in Example 10.1.

```
a=1
b=[1,2,3]
c="Hello world"

x=Int1d.range(3)
y=Complex2d([ [1+2j,3+4j,5+6j], [0+1j,2+3j,4+5j] ] )
z=Double3d(4,2,3)
z[0,0,:]=x
z[3,1,:]=x+1

u=ArrayDataset(data=x.copy(),description="Demo array dataset")

# --- save some of the above variables
save("xyz.sav", "x,y,z")

# --- save all variables
save("all.sav")

# --- make all variables invalid
a=b=c=u=x=y=z=None
print a,b,c

# --- restore x,y,z
restore("xyz.sav")
print x,y,z
x=y=z=None
print x,y,z

# --- restore all
restore("all.sav")
print a,b,c
print x,y,z
print u
```

Example 10.1. Using save and restore

10.3. Getting Started with ASCII Import/Export

Assuming you have successfully started `jide`, then the following packages should already be available within the standard DP setup:

```
herschel.ia.io.ascii
herschel.ia.dataset
```

10.3.1. Basic ASCII Table Import/Export Tool Usage

The tool to read and write tabular ASCII files is called `AsciiTableTool`. In your session, you may have multiple instances of this tool - each with a different configuration to suit the format of the input/output tables being used.

In general, create the ASCII tool with default settings

```
ascii = AsciiTableTool()
```

`ascii` is now known in your session as a table import and export tool. You can apply methods on `ascii` to load and save tabular information from and to an ASCII file.

Let us set up a `TableDataset` to export. Input the following lines into JIDE bottom left panel:

```
table = TableDataset()
table["x"] = Column(Double1d([1.0,2.0,3.0]))
table["y"] = Column(Double1d([4.0,5.0,6.0]))
table["z"] = Column(Double1d([7.0,8.0,9.0]))
```

We can now export it to an ASCII file with the following command:

```
ascii.save("table.output",table)
```

If we now look inside the file, "table.output", we'll see something like this:

```
x,y,z
Double,Double,Double
''
''
1.0,4.0,7.0
2.0,5.0,8.0
3.0,6.0,9.0
```

The first two lines show the name and data type of each column. The third and fourth lines show the units and description of the columns. Here they are empty because we did not set any.

To load the data back into JIDE the command is

```
loadedTable = ascii.load("table.output")
```

You can look at the new `TableDataset` by typing `print loadedTable`, to see that it is the same as `table`, as expected.

You can change the behaviour of the tool to allow various formatting changes with the following attributes:

<code>parser=yourParser</code>	Changes the line parsing behaviour at import.
--------------------------------	---

formatter=yourFormatter	Changes the line formatting behaviour at export
template=yourTemplate	Specifies how to interpret raw cell data.

For example

```
ascii.parser = CsvParser()
```

indicates to use the `CsvParser`, while

```
ascii.formatter = CsvFormatter(delimiter = '&')
```

indicates that we want to use a non-standard delimiter (ampersand rather than a comma).

10.3.1.1. Import Parsers

A parser controls how to break-up a line into table cell data. All parsers share the following attributes:

ignore=expression	Lines containing expression are ignored. By default the expression skips lines starting with a hash, possibly preceded by one or more whitespaces:
skip=value	First number of lines can be skipped by specifying a value>0. Default is 0.
trim=0 1	Whether to strip lines from leading and trailing spaces, default is 0 (false).

The following lines make the parser skip the first twenty lines and removes leading and trailing blanks.

```
ascii.parser.skip = 20
ascii.parser.trim = 1
```

10.3.1.2. Comma-Separated-Variable Parser

The Comma(Character)-Separated-Variable Parser named `CsvParser` breaks up a line into cells using a delimiter symbol. The delimiter character can be part of one or more cell-data itself.

In addition to the common attributes of any parser, a `CsvParser` gives you control over the following extra attributes:

delimiter=character	The character used to distinguish cells within a line of data. Default is a comma character ','.
quote=character	The character used if cell-data contains a delimiter character. Default is a double quote character '"'.

This example skips 2 lines and makes the delimiter symbol a semi-colon. The * character is used to indicate cells containing the delimiter symbol.

```
ascii.parser = CsvParser(skip=2,delimiter=';',quote='*')
```

10.3.1.3. Fixed-Width Parser

The `FixedWidthParser` breaks up a line into cells by interpreting every cell to be of a fixed number of characters.

In addition to the common attributes of any parser, a `FixedWidthParser` gives you control over the following extra attributes:

<code>sizes=array</code>	An array <code>n</code> elements, where <code>n</code> is the number of columns, and each element specifies the width of that cell.
--------------------------	---

This example uses a `FixedWidth` parser that expects three columns in the table with widths 10, 20 and 10 characters respectively - and in that order.

```
ascii.parser = FixedWidthParser(sizes=[10,20,10])
```

10.3.1.4. Regular Expression Parser

The `RegexParser` breaks up a line into cells by interpreting every cell to be separated by a set of characters given by a standard regular expression.

The following short example uses a `RegexParser` that expects a vertical slash separator with one or more spaces either side.

```
ascii.parser=RegexParser(delimiter="\s*\| \s*")
```

10.3.1.5. Export Formatters

A formatter controls how to format a row of cells into a line of ASCII. All formatters share the following attributes:

<code>commented=0 1</code>	States whether comments will be allowed in the output or not, default=0 (false).
<code>commentPrefix=string</code>	Prefix used for all comments, default="# ".
<code>header=0 1</code>	Whether to precede the actual data with header information, default is 0 (false). This header may contain name, type, units and description of each column

In the following example, first indicate that a header is to be added to the output table, then allow comments in the output and finally indicate how comments are prefixed in the table.

```
ascii.formatter.header=1
ascii.formatter.commented=1
ascii.formatter.commentPrefix="$$$ "
```

10.3.1.6. Comma-Separated-Variable Formatter

Please read its counterpart `CsvParser` (see Section 10.3.1.2) for parameters and defaults.

The default comma(character) separated variable formatter has a ',' delimiter and a '#' quote character.

```
formatter = CsvFormatter()
```

The delimiter and quote characters can be changed, e.g. the `&` symbol is useful for creating latex tables


```
formatter = CsvFormatter(delimiter='&', quote='<')
```

10.3.1.7. Fixed-Width Formatter

Please read its counterpart `FixedWidthParser` (see Section 10.3.1.3) for parameters and defaults.

Take default width for table cells

```
formatter = FixedWidthFormatter()
```

Set the width of 3 columns of cells to specific sizes

```
formatter = FixedWidthFormatter(sizes=[5,12,3])
```

10.3.1.8. Table Template

Many tabular ASCII files contain only raw data. Though the human eye may interpret cell-data being a string or a rational number, the computer needs some more information.

The `TableTemplate` allows you to specify such information. The only mandatory argument for a table template is the number of columns that are expected. Its optional attributes are:

<code>names=array</code>	Specifies names that will be attached to the columns.
<code>types=array</code>	Specifies the types of all columns. If not specified, the template assumes that all columns are of type <code>String</code> . Allowed types are: <code>Boolean</code> , <code>Integer</code> , <code>Float</code> , <code>Double</code> and <code>String</code> .
<code>units=array</code>	Specifies the units of all columns. Uses SI units, and units that are accepted for use with SI.
<code>descriptions=array</code>	Specifies comments for all columns.

The following table template indicates a table with 4 columns with associated names character/number types and associated units

```
ascii.template=TableTemplate(4,\
names=["Frame", "Energy", "Foo", "Bar"], \
types=["Integer", "Double", "Double", "Double"], \
units=["s", "eV", "N m -1", "kg L-1"])
```

10.3.2. Examples of How to Import/Export ASCII Tables in DP

Section 10.3.1 introduced the various import and export capabilities of the `AsciiTableTool`. We can put these together to illustrate how a user can import and export ASCII tables of virtually any type. The example below provides an illustration of how to handle ASCII tables in the DP environment. A number of ASCII tables are created and reimported. These can be viewed by opening them with the `jide` window (or within any other text editor). In order to run the program the user will also require an input file, which is given below and can be downloaded from [here](#). Remember to rename the file to `ascii_demo_data.txt`, and to delete any blank lines at the end, otherwise you will get an error when reading its contents.

```
# Sample file, using default settings of AsciiTable object
# table=AsciiTableTool().load("ascii_demo_data.txt")
```

```

Frame,Counts,Valid,Comments
Integer,Double,Boolean,String
s,eV,,
'''
1,1.0,true,
2,5.0,true,
3,0.0,false,incomplete data
4,0.0,false,missing data
5,1.234567E-8,true,

```

```

# --- import a table that complies to default settings
ascii=AsciiTableTool()
table=ascii.load("ascii_demo_data.txt")
# --- export a table using defaults settings:
ascii.save("table.out1",table)
# --- export using Fixed Width format, with header info:
ascii.formatter=FixedWidthFormatter(sizes=[8,16,8,30])
ascii.save("table.out2",table)
# --- importing it back requires Fixed Width parser
ascii.parser=FixedWidthParser(sizes=[8,16,8,30])
table=ascii.load("table.out2")
# --- export using Fixed Width format, only raw data:
ascii.formatter.header=0
ascii.save("table.out3",table)
# --- importing a raw "fixed width" table that has only data. So we
# have to define the template ourselves:
ascii.template=TableTemplate(4,names=["Frame","Counts","Valid","\
Comments"], types=["Integer","Double","Boolean","String"])
table=ascii.load("table.out3")
# --- saving current state of AsciiTableTool:
ascii.save("table.template")
# --- quick save table with default settings, equivalent to
#"table.out1":
AsciiTableTool().save("table.out4",table)
# -- reloading state:
mine=AsciiTableTool("table.template")
table=mine.load("table.out3")
mine.save("table.out5",table)
# --- saving with comments
table.description="Sample description can be found here"
mine.formatter.header=1
mine.formatter.commented=1
mine.formatter.commentPrefix="; "
mine.save("table.out6",table)

```

Example 10.2. ASCII demo data

Finally, we also present an example of the use of the `RegexParser` for importing tables.

```

from herschel.ia.io.ascii import *

#instantiate the table tool
ascii = AsciiTableTool()
# regular expression looks for vertical slash between spaces
ascii.parser=RegexParser(delimiter="\s*\| \s*")
#6 columns will be read
ascii.template = TableTemplate(6)
# now load it
cat = ascii.load("test_ascii_space.dat")
#get the number of data elements in the first column
n = len(cat["Column0"].data)

#Now print out the columns we have read into "cat"
for i in range(n):
    print cat["Column0"].data[i],cat["Column1"].data[i],\
          cat["Column2"].data[i],cat["Column3"].data[i],\
          cat["Column4"].data[i],cat["Column5"].data[i]

#####

```

The data file for the above script is the following which should be called "test_ascii_space.dat":

```
#####
1 | 2 | 3 | 4
2 | 3 | 4 | 5
3 | 4 | 5 | 6
4 | 5 | 6 | 7 |

| 5 | 6 | 7 | 8 |
6 | 7 | 8 | 9 |
a | b | 8 | 9 |
#####
```

The result from above script should look like this:

```
#####
1 2 3 4 None None
2 3 4 5 None None
3 4 5 6 None None
4 5 6 7 None None
None None None None None None
None 5 6 7 8 None
6 7 8 9 None None
a b 8 9 None None
#####
```

10.4. Overview of FITS IO

In the next few sections we describe how to write and read Products (which contain one or more datasets, a history of how it was created and meta-data describing the contents - the latter two are typical FITS header components) to and from FITS files within the DP environment.



Note

FITS stands for Flexible Image Transport System, a format adopted by the astronomical community for data interchange and archival storage.

10.4.1. Getting Started With FITS IO

Assuming you have successfully started JIDE, the facilities needed to create products as well as to create FITS files should already be available in your session.

10.4.1.1. Basic FITS IO Tool

The tool to write and read Products to and from FITS files is `FitsArchive`. In your `jide` session, you may have multiple instances of this tool -each with a different configuration.

In general, we can set up a FITS file for archiving, export DP products to it and retrieve back a product from a FITS file.

A generic FITS reader is available. This generic reader can parse FITS files that were created by applications other than the HCSS software.

```
from herschel.ia.io.fits.FitsArchive import *

fits=FitsArchive(reader=STANDARD_READER)
product=fits.load("input.fits")
myDisplay3 = Display(Double2d(product["PrimaryImage"].data))
#which takes the data from the FITS file, puts it into a 2D array
#and displays it.
```

Example 10.3. Using FITS Archive

The HCSS `product` can be manipulated in the DP system in a similar way to DP-produced arrays. In the above example, a 2D FITS image is displayed after having been imported.

A DP product containing data and meta data can be saved into a FITS file using the following

```
fits.save("output.fits",product)
```

10.4.2. Parameter Name Conversion and FITS Header

The current implementation of the FITS archive converts long, mixed-case parameter name, defined in the meta data of your product, into a FITS compliant notation. The latter dictates that parameter names must be uppercase, with a maximum length of eight characters. Clearly, we do not want to force all our parameters to have names that fit within such a FITS specific restriction.

The FITS Archive uses lookup dictionaries that convert well known FITS parameter names into a convenient and human readable name. Currently the following dictionaries are in use:

Common keywords (<ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/releases/doc/ia/io/fits/dictionary/DictionaryHeasarc.map>) widely used within the astronomical community, which are taken from HEASARC (<http://heasarc.gsfc.nasa.gov/>), Standard (<ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/releases/doc/ia/io/fits/dictionary/DictionaryStandard.map>) FITS keywords, and HCSS keywords (<ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/releases/doc/ia/io/fits/dictionary/DictionaryHcss.map>) containing keywords that are not defined in the above dictionaries.

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

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

Providing the following FITS product header via direct translation using the lookup dictionaries.

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

```
PROGRAM = 'FooBar '
```

A full demonstration of FITS IO is available in the example below. 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. note this is only for demonstration
# purposes. For more information, please see the dataset and numeric
# chapters of the manual (chapters 5 and 6).
points=50
x=DoubleId.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 10.4. FITS IO from within Herschel DP

10.4.3. Caveats



Warning

A FITS header card is limited to 80 characters. Within those limitations the `FitsArchive` will try to store the abbreviated FITS keyword, parameter value, and in the comment area optionally a quantity and description. The latter two might be truncated due to these limitations. Also a `StringParameter` with a long value can be truncated.

For more information see the FITS IO general documentation (<ftp://ftp.rssd.esa.int/pub/HERSCHEL/csd/leases/doc/ia/io/fits/index.html>)

Chapter 11. Using Time in the DP Environment

11.1. Introduction

This note describes which and how Time is defined within HCSS and how to deal with it. Unfortunately, there are several ways in which time can be represented. The standard for the HCSS/DP is a `FineTime` - which is the number of microseconds since the beginning of 1 January 1958. This provides the kind of accuracy needed to represent time on a space mission.

However, there are several other time representations and it is often the case that conversions between times/dates is necessary. In particular, it is noted that the standard Java commands lead to date measurements with respect to 1 January 1970. This chapter indicates how to deal with times within DP and converting between the various times, particularly between dates and `FineTime`'s.

11.2. Time Definitions

11.2.1. System time in DP

There are many ways to access the system time within DP. See also the description of the Java class "Date" for a discussion of slight discrepancies that may arise between "computer time" and coordinated universal time (UTC).

The Java `Date` class is deprecated and is being replaced by a more flexible `SimpleDateFormat` capability within Java that allows the user to express dates more conveniently. A `Date` object is still obtained and can be turned into a `FineTime` (see below) once created.

Two possibilities for creating a "Date" object are:

```
# To get the current time in milliseconds:
# The difference, measured in milliseconds, between the current
# time and midnight, January 1, 1970 UTC.
print java.lang.System.currentTimeMillis()
# To get the number of milliseconds since
# January 1, 1970, 00:00:00 GMT represented by a Date object.
d = java.util.Date()
#printing this gives the current time and date at the location of the
#system on which the java is being run.
print d
#We can also get the number of milliseconds since Jan 1, 1970 using
#this Java Date
print d.getTime()
```

Example 11.1. Current Time

Note that while the unit of time of the return value is a millisecond, the granularity of the value depends on the underlying operating system and may be larger.

If we want to get the number of milliseconds since 1 January 1970 for any other date then we can use a non-default form of the Java `Date` capability where the year, month, day, hour, minute and second are provided.

- Year format -- year (A.D.) - 1900. So the year 2006 = 2006 - 1900 = 106
- Month format -- number of the month, beginning from January = 0. E.g. March = 2.
- Day -- just day number in the month.
- Hours, minutes, seconds -- on the 24-hour clock.

NOTE: This is the time on our computer system.

```
#Format of date is year (in units of true year - 1900), month (number 0...11),
#day, hour, minute, second. So the following gives us the number of milliseconds
#between the beginning of 1 January 1970 and 3:15:00 pm on 23 October 2004.
d = java.util.Date(104, 9, 23, 15, 15, 0)
print d # should indeed show we have 3:15pm on 23 October 2004
print d.getTime() # provides the number of milliseconds between this
#date and 1 Jan 1970.
```

The following sample code shows how to use `SimpleDateFormat` to create a "Date" object.

```
simpleDate = java.text.SimpleDateFormat("yyyy.MM.dd HH:mm:ss z")
#set up how you want to set up your input Date format. In this
#case we could input "2006.01.14 01:00:00 CST" for 1a.m. on 14
#January 2006. z -- indicates the time zone (default is the zone for the
#computer system being used).

simpleDate.applyPattern("dd/MM/yy HH:mm")
#change the pattern to a different format

startTime = simpleDate.parse("13/01/06 14:06")
#create the data object "startTime"

print startTime
#...and see what this looks like
```

Allowed choices for the data format are available from Java documentation of the `SimpleDateFormat` capability.

11.2.2. International Atomic Time (TAI) and `FineTime`

TAI is an international standard measurement of time based on the comparison of many atomic clocks. TAI is the basis for Coordinated Universal Time (UTC). `FineTime` is based on TAI as measured from 00:00:00 1 January 1958.

11.2.3. Coordinated Universal Time (UTC)

UTC, World Time, is the standard time common to every place in the world. UTC is derived from International Atomic Time (TAI) by the addition of a whole number of "leap seconds" to synchronise it with Universal Time 1 (UT1), thus allowing for the eccentricity of the Earth's orbit and the rotational axis tilt (23.5 degrees), but still showing the Earth's irregular rotation, on which UT1 is based.

11.2.4. DecMec Time [PACS only]

The commands `DPUSelectTime` and `DPUWriteTime` are selecting and setting a start time which is written to the `TMP1` and `TMP2` fields of the Dec/Mec headers. This is used in coordinating the activities of the mechanical devices on board PACS. It is possible to construct an absolute time by adding counters (CRDC) to the start time considering an offset between setting and writing the start time.

This offset is expected to be a number with an uncertainty depending on the system load. It might require a calibration file. Currently this offset is not considered.

In case the commands are not given the `TMP1` and `TMP2` fields are zero. To avoid software confusions the time will be related to a fixed date (1.Jan 1970, 0:00).

During construction of the `SpuBuffer` the time is computed from the `TMP1`, `TMP2` entries in the Dec/Mec header and the CRDC counter. This time is used during construction of the `DataFrameSequence` and the associated Tables holding the SPU science data.

Between the Dec/Mec time and the packet time (see `PusTmBinStruct`) we have an offset. Therefore the association between HK and science data will be within an accuracy of 2 seconds.

11.3. Time in Instrument House-Keeping (HK) Data

The most convenient method of obtaining time stamped HK information is the use of the "herschel.binstruct" package. The use of this is illustrated in Chapter 12.10 where HK data is obtained from a database and then read/converted for use within the DP environment.

When dealing with HK time information directly, it is important to know that telemetry packets contain the time as defined within the "PUS Data Field Header". The field represents the on-board reference time of the packet, referenced to TAI, expressed in spacecraft time units - CCSDS Unsegmented Time Code (CUC) units. CUC units are multiples of 1/65536 sec from 1 January 1958 in TAI time. CUC units cannot be expressed in whole microseconds but can be converted to the *FineTime* standard (see below).

CUC time is written for HK by the data processing unit (DPU).

Current `PusTmBinStruct` methods related to time:

long getTime()

Returns the packet time of the Pus telemetry packet.

boolean isTimeSynchronized()

Returns true if the telemetry packet is synchronized, false otherwise.

java.util.Date getTimeAsDate()

Returns the packet time as a Date object.

FineTime getTimeAsFineTime()

Returns the packet time of the Pus telemetry packet as FineTime.

11.4. Time conversion

11.4.1. Time conversion in HCSS

It can often be the case that we need to convert between FineTime (TAI) and Date (UTC). Coordinated Universal Time is expressed using a 24-hour clock and uses the Gregorian calendar. FineTime represents a TAI time (epoch 1958), whereas the Java Date class is used to represent UTC, by resetting the system clock whenever a leap second occurs and don't need to handle leap seconds. Converting between Java dates and the FineTime standard requires the use of the `DateConverter()` class. Long integers can also be directly converted to FineTimes and are interpreted as representing the number of microseconds since 00:00:00 1 January 1958. In Example 11.2 we illustrate how to create a FineTime from a long integer and convert back and forth between FineTime and Java Dates.

```
from herschel.ccm.util import *
from herschel.share.fltdyn.time import *

# FineTime to Date
# Enter a time in seconds (a long integer - put letter "l"
# at the end of the number)
c = FineTime(14360944497154001) # convert to a FineTime
# Prints corresponding date and time
print DateConverter.fineTimeToDate(c)
# Date to a FineTime
d = java.util.Date() # gets today's date and time
# Prints corresponding FineTime
print DateConverter.dateToFineTime(d)
```

Example 11.2. Time conversion between Date and FineTime

11.4.2. CucConverter

Converts between Spacecraft Elapsed Time, in CCSDS Unsegmented Time Code (CUC) format and FineTime (TAI). This implementation is for the Herschel CUC format, which is corrected on-board the spacecraft to TAI (epoch 1 Jan 1958). This representation uses 32-bits for seconds and 16 bits for fractional seconds. CUC times are multiples of 1/65536 sec and cannot be expressed as an exact multiple of 1 microsecond (the resolution of FineTime). However, the following relations hold for 'coarse' and 'fine' values in the allowed range:

long coarse(FineTime t)

Return the number of whole seconds since the epoch 1 Jan 1958.

long cucValue(FineTime t)

Return the number of 1/65536 fractional seconds since the epoch 1 Jan 1958.

int fine(FineTime t)

Return the fractional part of the number of 1/65536 seconds since the epoch 1 Jan 1958.

FineTime toFineTime(long cuc)

Return a new FineTime constructed from a 48-bit CUC time.

FineTime toFineTime(long coarse, int fine)

Return a new FineTime constructed from CUC coarse & fine fields.

```
from herschel.share.fltdyn.time import *

d=CucConverter.toFineTime(50000000000000L)
#Converts the long integer - representative of a CUC time -
#into a FineTime. The FineTime is stored in d.
e = CucConverter.coarse(d)
#provides the number of whole seconds since 1 Jan 1958
#and stores it in e.
print e
```

Chapter 12. Accessing and Retrieving Data

This chapter has two main sections: First the section Section 12.1 deals with the Product Access Layer (PAL) and Product Pools. A particularly useful concept is that of a local product pool (or local store) which is a product pool that you can create on your local (e. g. laptop) computer and allows you to work offline if need be (see ???). In addition it provides guidance on how to query these data pools with the Product Browser tool. The section Section 12.2 describes how to set up and use a database created with the Versant Database System. We note that products in a Versant database can also, in principal, be accessed via the PAL as a remote pool.

12.1. The Product Access Layer and Product Pools

The Product Access Layer (PAL) consists of several elements that allow a user to create, access or query *Product Pools*. Product Pools are data storage areas that could be on your local laptop, (a local store) or might be a remote pool. Examples of a remote pool are, i) the future Herschel Archive, ii) products accessed from a Versant database, or iii) a pool which you can share with others on a remote computer. Perhaps the most convenient component of the PAL is the *Product Browser*. This is a graphical visualisation tool, and will be covered in Section 12.1.10. We will show an example in a moment of how to launch this from a JIDE session. The browser is also easily launched within HIPE.

12.1.1. Available Product Pools

The implementation possibilities are unlimited (using an object-oriented database such as Versant, relational databases such as Oracle, MySQL), but that is beyond the scope of this project.

With the release of the Product Access Layer, two main pools are available (LocalStore and DbPool), plus some mechanisms for setting up or accessing remote pools:

- A *LocalStore* for storing and accessing Products in your local system (default is FITS format)
- A *DbPool* for accessing Products from a remote object database, such as a Versant database.
- A *SerialClientPool* allows you to read/write or access a remote pool. When used in conjunction with a *PoolDaemon* (which runs on the mechine of the remote pool) this can make the remote pool immediately available to your session.
- A *CachedPool* is a way to cache everything retrieved from a pool. It is useful if the pool you are working with is normally a remote on-line pool, and you want to work offline.
- *HsaReadPool*: This is a pool that allows access from an HCSS session to the Herschel Science Archive (HSA).

We may further expect in the near future:

- *HttpPool*: A networked pool similar to SerialClientPool.

In the next few sections we will discuss and provide examples of pools mainly in the context of Local pools, but most of these examples can be generalized to any kind of pool. In later sections we will describe these other kinds of pools and some other useful concepts that refer to them.

12.1.2. Local Pools

We will in this subsection discuss Local pools. However much of this information presented here is applicable generally to any kind of pool.

12.1.2.1. The Default Local Pool directory and how to change it

By default, data is stored in a directory with the user-supplied store name in the following directory

```
users_home_directory/.hcss/lstore/
```

This can be changed by changing the property `hcss.ia.pal.pool.lstore.dir`.

For example, in MS Windows we can do this using the following statement in our JIDE session:

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

Or in LINUX with:

```
hcss.ia.pal.pool.lstore.dir=~/.hcss/alternate_store/
```

12.1.2.2. Registering Local Pools

The storage location pointed to by `hcss.ia.pal.pool.lstore.dir` can contain several pools, which in the specific implementation of local store are subdirectories in that location. After importing the PAL classes with `from herschel.ia.pal import *`, we create a storage object with `storage=ProductStorage()`. We obtain a reference (`pool1`) to a pool from the pool manager using the statement `pool reference = PoolManager.getPool(poolname)`, where `poolname` is a string. Then the pool reference is registered by `storage.register(pool reference)`. With the command `print PoolManager.getPoolMap()` we can see which pools are currently registered.

A practical example where we open two pools would look like this:

```
from herschel.ia.pal import *
storage = ProductStorage()
pool1 = PoolManager.getPool('default')
pool2 = PoolManager.getPool('test')
storage.register(pool1)
storage.register(pool2)
print PoolManager.getPoolMap()
```

In case there is already a pool with that name in the default directory, it is registered and becomes accessible. If it doesn't exist, the pool is created as soon as we store a product there. This can be verified by inspecting the respective directory before and after.

At this point we have created a storage and opened two pools. Note that when writing to the storage, the data is written to the first pool that was registered. If you want to write to a different pool you can create and use another storage for writing, where you register the desired pool. The same pool can be registered with more than one storage at the same time. Here an example where we make the pool "test" accessible for saving products.

```
otherStorage = ProductStorage()
otherStorage.register(PoolManager.getPool('test'))
```

We should also note that storage can also be obtained with the `LocalStoreFactory`, however this is discouraged by developers who strongly recommend using the `PoolManager`.

12.1.2.3. Saving products in pools

Let's first create some products to play with. In this case we will create two products containing one table dataset each. First the table datasets are created from random numbers.

```
r = RandomGauss()
```

```
n = 1000
tbl1 = TableDataset(description='Test Dataset 1')
tbl1['time'] = Column(DoubleId.range(n))
tbl1['signal'] = Column(DoubleId(n).apply(r))
tbl1['error'] = Column(DoubleId(n).apply(r) * 0.3)
prod1 = Product(creator='That'sMe', description='Test Product 1')
prod1['Table1'] = tbl1
```

We'll do the same for a second product:

```
tbl2 = TableDataset(description='Test Dataset 2')
tbl2['time'] = Column(DoubleId.range(n))
tbl2['signal'] = Column(DoubleId(n).apply(r))
tbl2['error'] = Column(DoubleId(n).apply(r) * 0.5)
prod2 = Product(creator='That'sMe', description='Test Product 2')
prod2['Table1'] = tbl2
```

Now we have two products, `prod1` and `prod2`, at our disposal. Their contents can be verified by launching the dataset inspector. Any product can be saved in our storage using the following statement: `urn = Storage.save(product)`, where `product` is the product to be saved and `urn` is the resulting "Uniform Resource Name" that is a unique identifier of the product within the storage. This URN can be used directly to retrieve the product from the storage, however typically the URN is not remembered, but rather re-obtained by a query to the storage. This will be shown later.

Lets save our two products using:

```
urn1 = storage.save(prod1)
urn2 = storage.save(prod2)
```

To see how the URN looks just use:

```
print urn1, urn2
```

As they are written by default to the first registered pool of storage, they will end up in the pool named default. Lets store one of the products also in the pool named test using:

```
otherStorage.save(prod1)
```

As we will recover the URN of this product later by a query, we dont bother to store the URN right now.

12.1.2.4. Finding out what is in storage: Starting the *Product Browser*

If we have followed all previous examples, there should be now 3 new products in our storage that have listed as creator *That'sMe*. Two of the products should be in the first pool named default, while the third product should be found in test.

We will examine first the simpler way to examine the contents of the storage using a GUI tool called the *Product Browser*. It is launched with the statement: `uri = browseProduct(storage)`, where `storage` is the storage we want to access and `uri` contains a list of references that result from our query. In our example we would type:

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

which brings up the GUI.

In the field "Creator:" type *That'sMe* to restrict the selection to the files we created in our example and hit the "Submit" button. The Query result panel in the middle left should now show a table with 3

rows, one for each product. Clicking on one of the rows will highlight it and bring up a diagram of the product contents on the panel to the right, where we can verify that our products contain attributes, metadata and datasets. The string to the right of the P is the URN. Clicking subsequently on the 3 rows shows how the URN changes for each product. We can see that the pool names default and test are part of the string, which shows that indeed two products ended up in the first and one in the latter. The Product Browser can be used to bring the URN for a given product into the JIDE session, i.e. make it available on the command line. Lets click on the squares to the left of the result table so that they are marked and the corresponding entries appear in the Download panel below. Upon clicking Apply, a list of the selected URNs becomes available in the variable result.

The statement:

```
print result
```

will show the list of the URNs we have selected. Note that after changing the selection and hitting "Apply" again, the `print result` command will give a different result corresponding to your selection. The "OK" button will update "result" as well and close the GUI.

The object "result" contains now a list of references to our products. We can obtain the same result "GUI-free" by creating a query on the command line and applying this to our storage:

```
query1=Query("creator == 'ThatsMe'")
res = storage.select(query1)
print res
```

Now "res" contains the list of references. Printing "res" should give the same result as the previous first example with the Product Browser.

If we want to execute an unconditional query to find all products in our storage, we can use:

```
query2=Query("True")
res2 = storage.select(query2)
print res2
```

In case we have used the default storage before, there may be other products here that would now show up in the list.

12.1.2.5. More On Storage Queries: Other kinds of Querie and more examples of command line queries

The Product storage can handle three types of queries:

- Attribute query is a (fast) query on meta data that **all** Products contain: *creator*, *creationDate*, *startDate*, *endDate*, *instrument*, *modelName*. This is akin to querying a standard set of FITS header keywords.
- Meta data query is a (semi fast) query on meta data that can be different from Product to Product, depending on what was placed in the product by the person creating it in the first place. This is akin to doing a query on any FITS keywords (if present).
- Full query is a data mining query that allows querying on *all* data elements in Products, using the general methods provided for Products and datasets as well as the additional methods provided in specialisations of those datasets and Products.

All query types have the same syntax, but a different purpose as described above. Setting up a query is as follows:

```
#Simple query
query = Query(expression)
#More advanced queries
query = AttribQuery(product-class, variable, expression)
query = MetaQuery(product-class, variable, expression)
query = FullQuery(product-class, variable, expression)
```

where the parameters to the query are:

- `product-class`: restricts a family of products. All Product classes have `herschel.ia.dataset.Product` as the base class. You can restrict the query to a sub-family of Product. For example, if all HIFI Calibration Product classes stem from `HifiCalProduct`, you can limit your search by specifying that class.
- `variable`: is a string denoting the variable name of the product that will be used in the expression.
- `expression`: is a string holding the query expression, which is limited to the query type.

It is worthwhile mentioning that the syntax of the expression above uses the same syntax as you would usually use when inspecting the contents of numerical data in a JIDE session, (see eg Chapter 4) so there is no additional syntax to learn.

• **Query Example**

```
query = Query("instrument ==HIFI and band == 1a")
# a simple query should be the default form used by most users.
```

• **AttribQuery Example**

```
query = AttribQuery(Product, 'product', \
    'product.creator=="Me" and product.instrument="HIFI"')
```

• **MetaQuery Example**

This type of query allows to inspect any part of the meta data of the product specified in the first argument.

```
query = MetaQuery(HifiCalProduct, 'h', 'h.meta["key1"].value < 123 and \
    h.meta["key2"].value == "Hello world"')
```



Note

In order to obtain a numerical value (rather than, e.g., the string equivalent) it is necessary to stipulate that the meta key "value" is required, hence the need for the stipulation of query on 'h.meta["key1"].value' rather than 'h.meta["key1"]'

• **FullQuery Example**

A data mining query exploits the full interface of the product in question. Numeric functions defined in the basic toolbox are allowed:

```
query = FullQuery(Product, 'p', 'p.creator=="Me" and (ANY(p.spectrum.data < 2) \
    or ALL(p["myTable"]["myColumn"].data > 5)')
```



Note

Note that the ANY function used above is one of the standard numerical function provided for DP, and simply checks whether the expression provided in its argument is true for any of the elements in that argument. See the DP User's Reference Manual for more information.

12.1.2.6. Retrieving products from storage

The list of references obtained by our query with either the Product Browser or the command line allows to load the product back from the storage using `product = storage.load(res[index].urn).product`, where `index` is the index of the list entry to be retrieved. Following our example and assuming we still have the result `res` from our query1, we would retrieve and plot the second product in our list by:

```
p1 = storage.load(res[0].urn).product
```

The Table Dataset would be extracted and plotted with:

```
t1 = p1.get('Table1')
p1 = PlotXY( t1['time'].data, t1['signal'].data,\
style=Style(line=Style.MARKED, symbol=Style.TRIANGLE) )
p1.setErrorY(t1['error'].data,t1['error'].data)
```

12.1.2.7. Deleting Products from Storage

Now we want to clean up our storage again, as this was just an exercise. In theory we could go into the relevant directory, identify the products by their filename and delete the respective *FITS* files. After that we would need to re-build the index. This would work for the *Local Store*, we used in our example, but in other implementations like the *DbPool* that would not be an option.

To remove our test products within the *PAL* context, we first need to identify them again by obtaining their URNs and use the method `.remove()` on the storage. In our example we can remove the first two items in our list as follows:

```
query1 = Query(creator == That'sMe)

res = storage.select(query1) storage.remove(res[0].urn)

storage.remove(res[1].urn)
```

We can verify now with:

```
print storage.select(query1)
```

Trying to remove the third product in the previous list will result in an error, as we have no write permission to the pool test through this storage. We will need to access this pool through the other storage which was created by registering test as the first pool.

```
res1 = otherStorage.select(query1)
otherStorage.remove(res1[0].urn)
print storage.select(query1)
print otherStorage.select(query1)
```

The last two statements verify that the operation was successful and affected both storages because the pool test is registered in both. Both queries result in an empty list.

12.1.2.8. Updating/Repairing Storage

If the storage index becomes inconsistent, for example in the case of files being deleted or added in the directory, the index can be re-built using `pool.rebuildIndex()`, where `pool` is a pool reference obtained from the pool manager as shown above. For example the index of *Pool1* can be rebuilt with:


```
pool1.rebuildIndex()
```

There should be no attempt to access this pool during the operation, which can take a while depending on pool size.

12.1.3. DbPool

Used to access Products stored in a remote object (Versant) database. Here's an example:

```
pool = DbPool.getInstance()
# Access to Products from the default
# object database of logical name
# 'hcss.test.database'
pool = DbPool.getInstance("hifi.test.database") # access to Products from an
# object database of logical
# name 'hifi.test.database'
```

Note that this is an early implementation that needs to be tested thoroughly, so it is recommended to use DbPools only around test databases, or databases that are used for casual development purposes such that if data is lost, it is not a big problem.

It is recommended for performance purposes to cache products locally. To do this, wrap a CachedPool around a DbPool as follows:

```
pool = CachedPool(DbPool.getInstance())
```

12.1.4. HsaReadPool

The HSA read pool is an implementation that allows you to access and download observations held in the Herschel Science Archives. By default, the whole observation context is downloaded when using this pool (level 0, 0.5, 1 and 2, plus auxiliary products):

```
archive = HsaReadPool(urlHaio+'metadata.jsp',urlHaio+'product.jsp')
store = ProductStorage()
store.register(archive)
```

Where 'urlHaio' is the URL of the Herschel Science Archive. Metadata and products are obtained

12.1.5. CachedPool

The cached pool is an implementation that allows you to cache everything (including queries and their results!) retrieved from any remote pool. Any remote pool, regardless of whether it is an Oracle, Versant or whatever implementation, can therefore be cached as follows:

```
pool = CachedPool(remotePool)
```

Registering a cached remote pool allows you to work offline.

12.1.6. Setting up and Accessing Remote Pools

12.1.6.1. PoolDaemon

If you have a pool that you wish to share with someone then you can start a PoolDaemon that allows a person access and indicates whether they have read/write/query access. The PoolDaemon can be started from a command line in your system.

```
java herschel.ia.pal.pool.serial.PoolDaemon [<hostPort>(=4444)
```

```
[<poolname>(<=${hcss.ia.pal.defaultpool}>=stdprod)
[<loadAccess>(<=true) [<saveAccess>(<=true)]]]
Examples:
    java herschel.ia.pal.pool.serial.PoolDaemon
    java herschel.ia.pal.pool.serial.PoolDaemon 4567
    java herschel.ia.pal.pool.serial.PoolDaemon 4567 stdprod
    java herschel.ia.pal.pool.serial.PoolDaemon 4567 stdprod true true
```

This makes the pool available on port number 4567.

12.1.6.2. Accessing Remote Pools Using the SerialClientPool

SerialClientPool (prototype) and *PoolDaemon* can be used to access remote pools.

SerialClientPool can be used for accessing a remote product pool. Usage:

```
# a PoolDaemon is running at
#   host=the.host.domain
#   port=4567
#   pool.id=foo
# create a store and register the pool:
store=ProductStorage()
store.register(SerialClientPool("the.host.domain",4567,"foo"))
```

A simple mechanism to allow read/write/query access to remote pools. This remote pool can be a Versant one (making happy all those who cannot run a Versant client such as the MacOS X fellows, or those who do not have a Versant licence), or a local store of a colleague.

Note that wrapping it up in a *CachedPool* ensures that you do not have to download a product twice.

12.1.7. Special Imports into Pools

We can import/store files of various types in pools. Here, we give some specific examples.

12.1.7.1. Putting a Directory of FITS Files Into a Pool

It is possible to take any set of FITS files (e.g. from the Herschel Science Archive) and place these into a pool. We can iteratively place all FITS files from a directory into a pool which can be accessed via a browser and queried using the mechanisms described in this chapter.

```
from java.io import File

lstore=LocalStoreFactory.getStore("newdir") # or any local store name
storage=ProductStorage()
storage.register(lstore)

lstore.ingest(File("C:/testdata/"), 0)

#or any directory name
# to look at what you have use the Product Browser

a=browseProduct(storage)
```

In the above example a local store is placed in the default area (.hcss directory under the user's home directory) of the user's computer. It is directly accessible in the same way as other pools from there. This method does, however, not reproduce any hierarchy to the pool. It is a "flat" pool.

12.1.7.2. Placing Image (PNG) Files in a Pool and/or FITS File

Image data can be stored in a pool by placement in a Product with a suitable name, and saving this product in pool or in a FITS file:

```
# Obtain bytes from PNG image
bytes = ...

# Create a product with PNG data
p = Product()
p["png"] = ArrayDataset(bytes)

# Save it in a PAL pool
pool = PoolManager.getPool("myPool")
storage = ProductStorage()
storage.register(pool)
storage.save(p)

# Save it directly in a FITS file
fits = FitsArchive()
fits.save("myPng.fits", p)
```

The image can be placed in a byte array for storage in a dataset that can be placed in the pool.

```
# Obtain bytes from PNG image
# (it depends on how you generate the PNG image of a plot)
from java.awt.image import BufferedImage
from java.io import ByteArrayOutputStream
from javax.imageio import ImageIO

image = BufferedImage(<image name>) # implementing java.awt.image.RenderedImage
stream = ByteArrayOutputStream()
ImageIO.write(image, "png", stream)
bytes = ByteId(stream.toByteArray())
```

12.1.8. Common Problems

Why do I keep getting 'IndexError' or 'IllegalArgumentException: <query> could not be evaluated correctly' messages when I run my query on my PAL Product Storage?

You could get these message for one of the following reasons:

1. Your query string (the third argument of a query, eg 'p.creator==..') is simply not consistent with the jython syntax and could not be correctly interpreted by the internal jython interpreter the PAL uses. Check your query string by evaluating it on the jython command line. If your query uses a 'handle' to a product (eg the 'p' in a query 'p.meta[..]' is a handle), then create a dummy product of the type you want to query on the command line to test the query against.
2. It could be possible that the query references some data that does not exist in **any** of the products that match the product type you have passed in that query. If you see in the details of the error message something along the lines of '<something> does not exist', then this may be the case for you.

For example, consider the following MetaQuery:

```
query =MetaQuery(Product, 'p', 'p.meta["temperature"].value==10)
resultset=storage.select(query)
```

The query first starts creating a shortlist of all products in the storage matching type 'Product'. It then runs the query string on each product in that shortlist. If any of those products don't contain the information referenced in the query string, an error is raised.

There are two ways to avoid this:

- Be as specific as you can when it comes to specifying the product type in a query. If you know the product type you want to query is of type 'CalHrsQDCFull', then specify that. Running queries using the most general product type of 'Product' is *not* recommended.

- Run a two-stage query, using the `containsKey()` operator to check whether a component exists first, e.g.

```
# Get a sub-set of products that contain the metadata 'temperature'
queryOne= MetaQuery(Product, 'p', 'p.meta.containsKey("temperature")')
resultsetOne = storage.select(queryOne)
# Run the original query on this subset
queryTwo =MetaQuery(Product, 'p', 'p.meta["temperature"].value==10)
resultsetTwo = storage.select(queryTwo, resultSetOne)
```

Accessing the Results of a Query

The results set can be accessed in the following way

```
a = resultsetTwo.toArray()[0].product
b = resultsetTwo.toArray()[1].product
```

Why is my PAL query so slow?

One of the possible reasons is that you are executing a `FullQuery`, and full queries by their very nature are the most intense of queries and are therefore the slowest.

`FullQuery` executions should be run as the last stage of a multi-stage query operation. Below is an example of how to search a storage for products of type 'MyProduct' that are created by a developer called 'timo', but contain a specific value in the product data itself.

```
# Stage one: Find all products of type MyProduct with creator 'timo'
attquery = AttQuery(MyProduct, 'p', p.creator=='timo')
resultset = storage.select(attquery)
# Final stage: Find all products in selection generated from previous queries,
# that has a value 10 in the column 'mycolumn' in dataset 'mydataset'
fullquery = FullQuery(Product, 'p', 'p["mydataset"]["mycolumn"].data[5]==10')
storage.select(fullquery, resultset)
```

There can be as many intermediate queries between the first stage and final stage involving `AttribQuery` or `MetaQuery`, but `FullQuery`'s should be left to last.

12.1.9. Storage Product Versioning

12.1.9.1. Versioning

To save a set of versions of a particular edition of a Product:

```
edition = Product()
storage.save(edition) # version 0 of Product saved
# Modify edition
storage.save(edition) # version 1 of Product saved
```

To get the latest version of the Product edition, or the list of versions for that edition, you need to have available at least one, arbitrary, version. With this, you can recover the latest version of that Product, and the list of all versions of the Product in the storage. For example:

```
latest=storage.getHead(productRefOfAnyVersionOfEdition)
versions=storage.getVersions(productRefOfAnyVersionOfEdition)
```

You can get information on the current version of each product, as well as tag information, as follows:

```
print storage.versioningInfo
```

12.1.9.2. Querying Product Versions

The default query is to search for just the latest version of a Product edition:

```
query=AttribQuery(Product, "p", "1")
storage.select(query) # Just the latest versions
```

If you want to get all versions of editions that match a query, use the extended query constructors, setting the fourth argument to true (or 1):

```
query=AttribQuery(Product, "p", "1", 1)
storage.select(query) # All versions of Product editions that match
```

(Note that with this extended query, the special products containing versioning information, `VersionTrackProduct` and `TagsProduct`, are also returned if they match the query.)

Warning: make sure that you use the `meta.containsKey()` checks when performing Full or Meta-data queries, as the presence of versioned products may affect those queries, or worse, result in an exception if the metadata being queried for is not present in any product version.

12.1.9.3. Tagging Products in a Store

To save a product with a given tag:

```
storage.saveAs(myproduct, "mytag")
# saves myproduct to URN=product:123, and links tag 'mytag' to that URN
storage.load("mytag")
# returns a ProductRef to product at URN=product:123
```

To assign a tag to an existing product in the storage:

```
storage.setTag("mytag", urn)
```

You can assign multiple tags to the same product:

```
storage.setTag("mytag1", urn)
storage.setTag("mytag2", urn)
storage.setTag("mytag3", urn)
```

You can re-assign tags from one product to another:

```
storage.setTag("mytag", urn1)
storage.setTag("mytag", urn2)
```

Note that the above steps removes the tag `mytag` from `urn1`, and re-assigns it to `urn2`. A given tag maps to only one URN.

You can also remove tags from the system:

```
storage.removeTag("mytag")
```

And check if a given tag exists:

```
print storage.tagExists("mytag")
```

12.1.9.4. Turning Off Product Versioning

If Product versioning is not wanted or required, you can turn off the use of versioning within your session by using.

```
hcss.ia.pal.version = none
```

12.1.9.5. Using the New Versioning Mechanism Against Existing Pools

You can use the new versioning mechanism against pools with previously existing data. Although it is highly recommended to use the mechanism against new pools with no data.

If you wish to use the mechanism against pools with existing data be aware that existing products in your pool do not have versioning information. So if you modify such products, and then save them:

```
p = oldstorage.load("myurn").product
// modify p
oldstorage.save(p)
```

The PAL does not know what version the modified product belongs to, and therefore saves the modified version of the product as the first version of a whole new version track.

It is therefore recommended to use the new versioning mechanism against a clean ProductStorage, devoid of any products, or as the next best thing, migrate your products to a fresh pool as follows:

```
storage.register(newpool)
storage.register(oldpool)
p = storage.load("urn:123").product
storage.save(p) # saves the product with versioning information, to newpool
```

And then use the newpool for future sessions (archive or remove oldpool).

Note also that a tool for copying pools, which reads all products and saves them back again, by preserving their hierarchy, will be placed in the HCSS at a later date. This will allow migration from old to new pools to be done more easily.

12.1.10. The Product Browser

After all the theory it is now time to entertain ourselves with a graphical tool, the *Product Browser*. We will start by describing how to start the browser from JIDE, before moving on to a short description of the current browser features.

To start the browser, open a JIDE session and execute the following script:

```
from herschel.ia.pal import ProductStorage
from herschel.ia.pal.pool.simple import SimplePool
from herschel.ia.pal.browser.ProductBrowser import browseProduct
storage=ProductStorage()
storage.register(SimplePool.getInstance())
result = browseProduct(storage)
# Use the popped up GUI to explore and select products.
# The result variable will not be populated until you push
# either 'Ok' or 'Apply' in the Product Browser.
print result
```



Note

Alternatively, execute the script `herschel/ia/pal/browser/browserStart.py`

12.1.10.1. A visual tour of the browser

The following image shows a screenshot of the product browser user interface. The screen is divided into four areas:

1. Query area: enter query parameters.
2. Result area: view the result.
3. Result inspection area: inspect a selected product.
4. JIDE basket area: collect the products to be returned to JIDE.

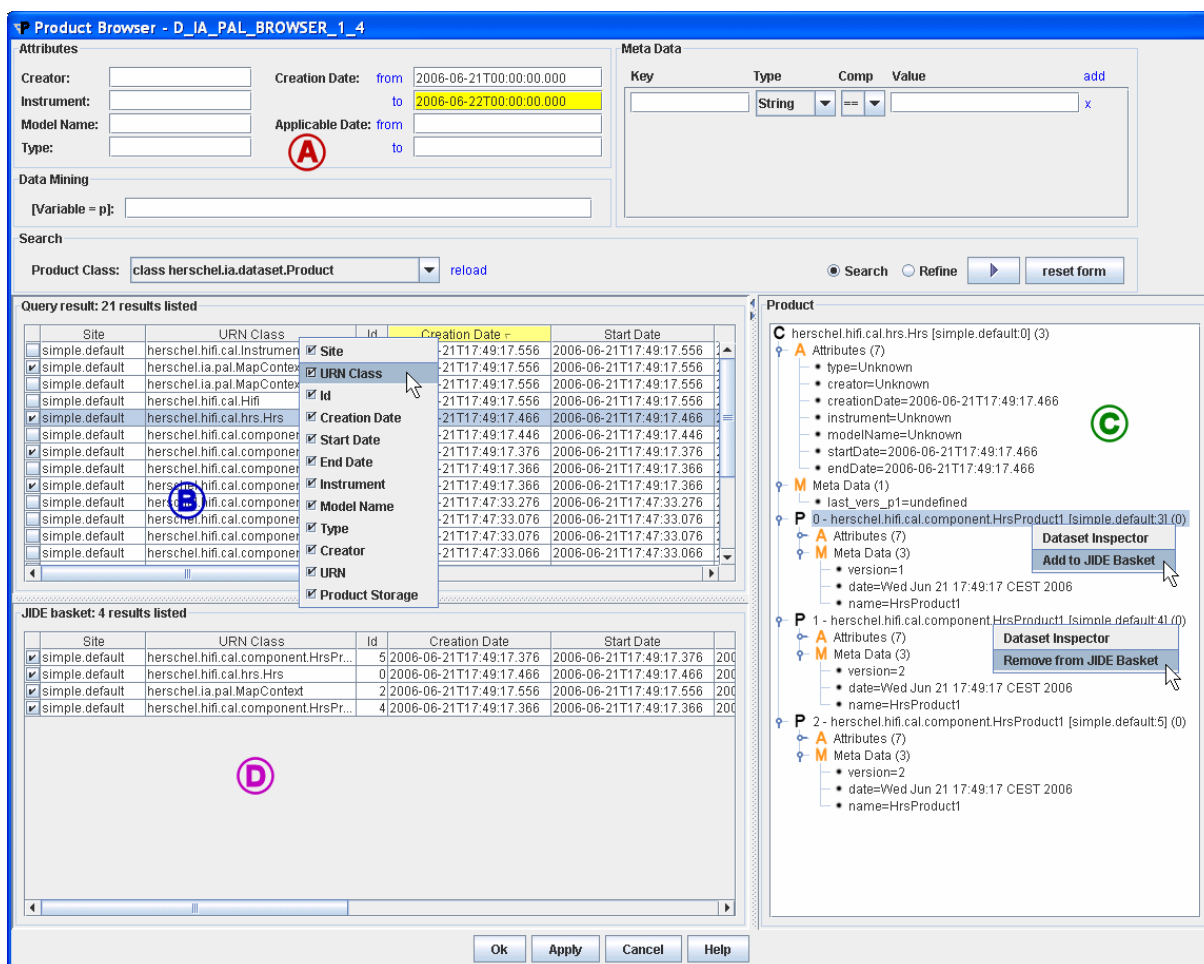


Figure 12.1. The Product Browser

The following sections describe first a typical use case and then each area in more details.

12.1.10.2. Simple use case

1. Specify attributes of a product in the query area (A)
2. Click on the "play" button to execute the query

3. Review the results in the result area (B)
4. Optional: if there are too many results, refine the query by specifying meta data and/or data mining queries, and press the "Refine" button.
5. Inspect selected results in the result inspection area (C)
6. Transfer the results of interest from the area (B) and (C) to the JIDE basket (D)
7. Click "ok" or apply and process the selected results in JIDE. The results are available in the return variable of the browseProduct() method (in the browser start example above it is called "result").

12.1.10.3. A: Query area

The query area is divided into three input areas: Attributes, Meta Data, and Data Mining.

1. Attributes queries search commonly defined attributes only.
2. Meta data queries search on additional meta data specific to a product. The user needs detailed knowledge about a product to specify meta data queries. However, the result inspection area (C) may be used to see available meta data for a product.
3. Data mining queries allow to specify free form queries in the Jython query language. Refer to the documentation of the ProductStorage for further information on this topic.

Note that all attributes and meta data parameters are joined by the AND operator.

Note for power users: for simple OR-Operations you can use the JIDE basket (D). First, do a query for the first term (e.g. Creator="André") and add the results to the JIDE basket. Then, do a query for the second term (e.g. Creator="Marc") and add the results to the JIDE basket.

For more complex OR-queries you can use Data Mining queries, although they might become very slow. Complex OR-Queries on meta data level are currently not supported.

12.1.10.4. B: Result area

This table displays all products that match a specific query.

Check/Uncheck a product to move it to or remove it from the JIDE basket.

You have several possibilities to rearrange the products:

- Click on a table header to sort ascending or descending.
- Right click on a table header to pop up a context menu where you can hide/unhide a column.
- Drag and drop a column header to rearrange the column order.
- Click between two column headers to resize a column.

Please note that the current version of the browser does not store your settings between two sessions. This is one of the high priority features for the next version.

Currently the browser supports the following columns:

- Site (URN): the data store of the result.
- URN Class (URN): the class of the product as encoded in the URN.
- Id (URN): the unique id within the data store.

- Create Date (Attribute): self-explaining.
- Start Date (Attribute): self-explaining.
- End Date (Attribute): self-explaining.
- Instrument (Attribute)self-explaining.
- Model Name (Attribute)self-explaining.
- Type (Attribute)self-explaining.
- Creator (Attribute)self-explaining.
- URN (URN)convenience column for copy & paste. If you triple click into a cell of this column you can select and copy the urn to your operating system clipboard. This is one way to use the browser independently from JIDE.
- Product Storage: experimental only. Might be of use if the browser support multiple storages.

12.1.10.5. C: Result inspection area

Select any entry in the query result area (B) or in the JIDE basket (D) to inspect its attributes, meta data and children in the result inspection area C. The selected product or context will be displayed in a hierarchical tree structure.

There are currently five types of nodes:

- P: a Product contains the real data and can be examined with the data set inspector. To open the data set inspector you can either double or right click on the 'Product...' node.
- C: a Context contains other Contexts or Products.
- A: a predefined set of Attributes common to all products and contexts.
- M: Meta data that is specific to each type of products.
- V: old Versions of a product or context.

To add/remove products and contexts to or from the JIDE basket you can right click and select the appropriate action: Add to/Remove from JIDE Basket.

First note for power users: The current implementation of the tree supports only contexts that are inherited from ListContext or MapContext. This is due to missing generic meta information about the children of an ordinary context.

Second note for power users: The current implementation of the tree does not support the description attribute of a product. This is due to a missing getter-method in ProductRefs.

12.1.10.6. D: JIDE basket area

The JIDE basket collects the products and contexts of interest. Clicking on "Ok" or "Apply" will make the content of the basket available within JIDE. "Ok" will close the browser, "Apply" will keep it open for further usage. Note that the results are sorted the same way as in the JIDE basket.

Now you can further analyse the results in JIDE. Note that the ProductBrowser will return a list of ProductRefs rather than a list of Products. A ProductRef is a small object that stores a pointer to a Product, without loading the Product into memory.

```

result = browseProduct(storage)
# This will print the list of ProductRefs
print result
# This will print the first ProductRef in the list.
print result[0]
# This will print the first Product in the list.
print result[0].getProduct()

```

12.2. Databases

12.2.1. Introduction

If you want to work with databases, which is one of the main ways in which test and (later) observational data are to be stored within the HCSS, then you will need to have a Versant Database System available to you. For most large sites your system manager will have installed a Versant license which allows the setup and use of databases at your home institution. You can also install a database capability on your own computer/laptop. Unix and Windows versions are available.

Most users will not need to set up a database but rather just access for reading stored data. In this case, Section 12.2.2 may be skipped.



Note

Versant is commercial software and procurement has been done centrally for Herschel. Please contact the Herschel software administrator at your institute for more details on how to proceed.

Alternatively you can contact the following people:

HIFI:

- Albrecht de Jonge
- Peer Zaal

PACS:

- Ekkehard Wieprecht for PACS/MPE
- Wim de Meester for PACS/KUL

SPIRE:

- Steve Guest

Some notes on Versant database setup are available in Section 1.4. For further information please also consult the Known issues with Versant Databases document.

12.2.2. Starting Up A Database:

The following command is the only one required to set up a database within the HCSS and make it available for use.

```
> db_admin -i <dbname>@<host> #initializes directory
```

Database names should be given in the format: `tony_hcss@lin-sron-02.sron.rug.nl`.

The database now being used should be in the properties file (use "progen" to check this out - Just put "progen" on the command line and hit the "General" tab at the top. The database currently in use is on the second line down. Change if needed).

Now we can fill the database.

12.2.3. Schema Evolution

On occasions, new database formats need to be created for the HCSS. In such cases, it is necessary to perform a *schema evolution* on old databases to update for use in current DP environment. For development purposes, it may of course be acceptable to simply create a new database, if there is no data to be preserved.

Schema Evolution is supported for databases created by versions of the HCSS back to HCSS-v0.1.3 (build number 168) although, in principle, it should be possible to go back to HCSS build number 162.

Schema evolution is necessary when a new version of the HCSS is installed that has a higher schema version than the database. The schema version of the CCM can be found by examining the file '%HCSS_DIR%/doc/SCHEMA_VERSION' in the HCSS distribution and is displayed against releases in the HCSS download web page.

If it is determined that a schema evolution is needed then the user is referred to the manager of the system and document DBA procedures.

12.2.4. Providing Database Access for a DP Session

Database access can be changed during a DP session without the need to exit JIDE. After editing properties or saving changes made using the properties tool (progen), the user can use the new settings immediately within an ongoing DP session.

There are two methods for changing properties to allow database access.

12.2.4.1. Properties File Setup for Database Access

There are two ways of setting up your properties to allow access to a particular database during a DP session. First, the file *hcss.props* (on Windows) or the file *myconfig* (on Unix) can be edited.

On Windows, the *hcss.props* file is usually in the top directory of the user (e.g., C:\Documents and Settings\

On Unix, the *myconfig* file is in the directory ~/ .hcss.

The following three lines should be placed in the file being edited if they are not already there.

```
var.database.server = servername
var.database.devel = dbname@${var.database.server}
hcss.access.database = dbname@${var.database.server}
```

where *servername* is the name of the server where the database is located (e.g. lin-sron-02.sron.rug.nl) and *dbname* is the name of the given database to be used in the DP session.

12.2.4.2. Using the Progen Tool

Alternately, the progen tool can be used to indicate the server and database to be used. The progen tool can be started from a terminal prompt assuming the HCSS system has been installed and it has been setup to run on the system.

At a terminal prompt, the command

```
> propgen
```

will start up the propgen tool (see Figure 12.2). Using the tabs at the top of the propgen screen, the user should click on the tab marked "General".

Now edit the variables **var.database.server** (input *servername*) and **var.database.devel** (input *dbname@\${var.database.server}*) at the top of the tabbed page.

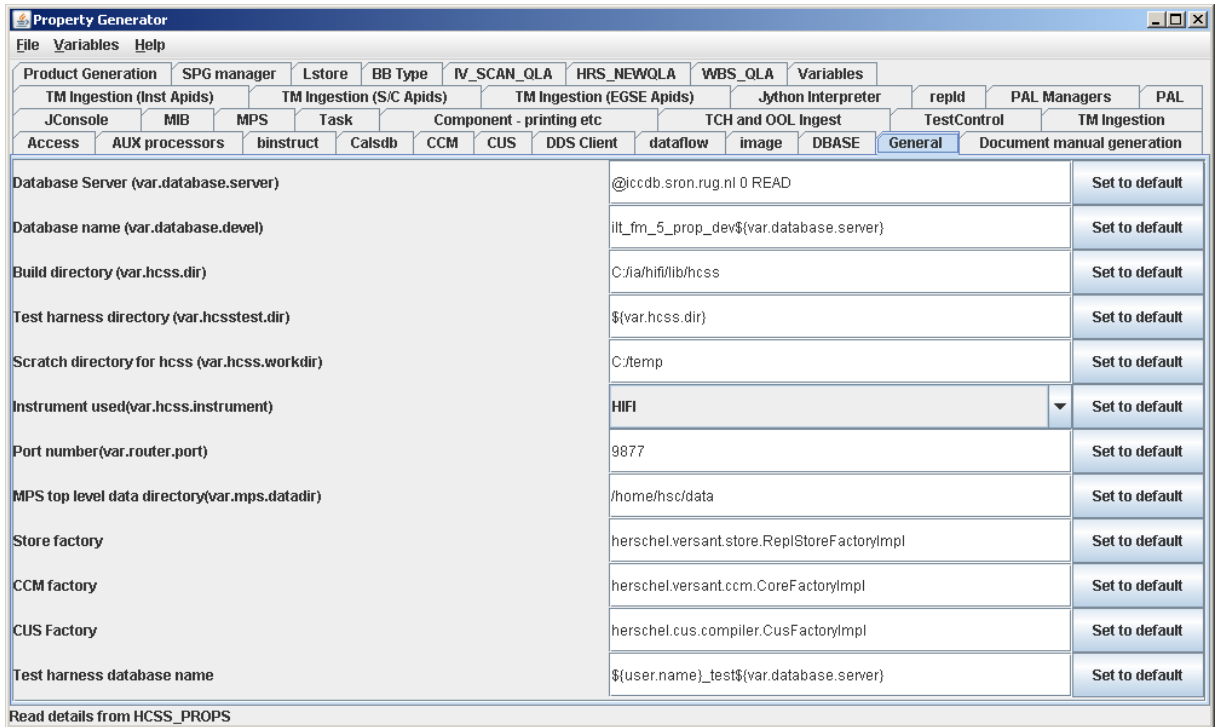


Figure 12.2. The propgen window.

Most properties that can be changed using the property generator are created automatically in the DP environment. The first screen 'General' allows the identification of a default database and server name to be set up.

12.2.5. Changing the Database to be Accessed

The database to be accessed for information can be changed by changing a configuration property called 'hcss.access.database'.

```
Configuration.setProperty("hcss.access.database", "<database name>")
```

12.2.6. Browsing a Database

In order to know what you might want out of a database, you need to be able to browse through the database contents. The `TestExecutionBrowser` task allows the user to do just that. Input of the following short example allows the user to view the database connected to. In dealing with databases we use the `herschel.access` package and the sub-package `herschel.access.util`. The first two lines of the small script shown below import these packages. These need to be imported into our session before using the browser to display the contents of a Test Execution database.



Warning

For large databases this can take up considerable amount of memory (and time) and is therefore only recommended for relatively small databases at present.

```
import herschel.ia.task.Task
from herschel.access import *
from herschel.access.util import *
# Used in this mode, the browser is not set up to allow mouse selection
TestExecutionBrowser.display()
```

Successful execution of this command will bring up a separate window displaying information on the data contained in the database. This includes information on the script used to create data, the observation ID (scroll to the far right of the window) and the time (local) for when the data was placed in the database.

At a future date, a filtered display of a database is expected to be possible (see bottom left of current *TestExecutionBrowser* window). It is also expected that selection and download following a mouse click will become available.

12.2.7. Getting Data Frames From a Database

Once connected to a database and knowing the date or observation id of your data (see previous section), we can retrieve both data frames and housekeeping data from the database. In this section we discuss the basic means for obtaining dataframes from a database. Here we are handling RAW dataframes for which there is no directly associated meta-data, although housekeeping data is available from the period of time during which the data was taken (see next section).

There are two main methods for obtaining dataframes.

- Command line access
- Through a DataSelector GUI

When accessing dataframes it is particularly useful to use the navigation property available in DP. This speeds up the accessing of dataframes in a database. In order to do this, start the progen tool (see Section 12.2.4.2) and then go to the "Access" tab. Near the bottom of the window, change navigation value from false to true (click on the cell containing the word "false" and pulldown to "true").

12.2.7.1. Command Line Access to Data Frames

The basic idea for command line access is to

- Create a means to access data frames
- Indicate which data you want to get (e.g., by observation identification, obsid)
- Go search for it in the database
- Actually get the frames and put them into an array (or table).

The following example illustrates how the above is done within a DP environment. In this example, an observation made up of several frames is placed in a table with each column of the table being a single 1D spectrum. Something similar could also be set up for multi-dimensional data. In these cases, each "column" of a table would have an N-dimensional object.

```

from herschel.access.util import *
from herschel.access import *
# Create a tabledataset for the data frames to go into
table=TableDataset()
# Start means by which we will access the dataframes
# in the database
dfaccess=DataFrameAccess()
# Provide an id for the frames we are looking for
# In this case the observation has an identification number of 1844
dfaccess.setObsid(1844)
# Find the data in the database (navigate/query).
# This just provides a set of references
# to where data frames fitting the criteria reside in the database
data=HcssConnection.get(dfaccess)
# If there is something found, length of the references > 0
if len(data) > 0:
    # Then loop around and get all the frames associated with the obsid
    for j in range(len(data)):
        df = data[j]
        # Now actually get each frames and put them in a real 1D array
        datad = Double1d(df.getFrame())
        # And we make each frame into a column in a table
        # so that table[0] is the first column and contains
        # the first 1D spectrum of the observation, value for each channel
        # The column label is the string value of j, i.e., 0, 1, 2, 3...
        table[str(j)]=Column(datad)

```

Example 12.1. Basic command line method for getting data frames from a database



Warning

Using `HcssConnection.get()` means that all the dataframes in the observation are passed into the user's JIDE session at one time. Care should be taken since this could lead to large amounts of data being requested and the JIDE session running out of memory. This will usually then require the user to quit the current session and unsaved work is lost.

Example 12.1 brings in a set of spectra as a table. To see what is in the table we can

```

# Get general overview
print table
# See what is in the first column
print table[0]
# See just the data for the first column. No quantities, column headings etc.
print table[0].data

```

A plot of `table[0].data` will show a channel versus value 1D spectrum.

12.2.7.2. From Database to ASCII File

Following on from the previous section. If we want to have the spectra be placed in an ASCII table output file, then we can add the following code to our example:

```

# Set up an output table
mine=AsciiTableTool()
# Add a description to our table
table.description="Sample spectra"
# Make sure there is a header on the output - see AsciiTableTool help
mine.formatter.header=1
# Make sure that comments are allowed
mine.formatter.commented=1
# Indicate the prefix for comments in the file
mine.formatter.commentPrefix="; "
# Provide a name for the ascii output file and save the data
mine.save("sample_spectra",table)

```

Being a little more sophisticated, we can add in a prompt and also iterate around to obtain several observations from a database and place them in ASCII files. The next example provides a basic Java Swing component to prompt the user for a starting and ending obsid. The data is then passed onto appropriately named ASCII table files.

```
# Import Java swing for GUI components
import javax.swing as sshwing
from herschel.access.util import *
from herschel.access import *
# The data will be placed in comma-delimited tables.
# Prompt the user for first obsid using a JAVA Swing component
input_obsid = sshwing.JOptionPane.showInputDialog\
("Enter first obsid in list: ")
start_obsid = int(input_obsid)
# Prompt again for last obsid
input_obsid = sshwing.JOptionPane.showInputDialog\
("Enter last obsid in list: ")
end_obsid = int(input_obsid)
for i in range(start_obsid,end_obsid+1):
    table=TableDataset()
    dfaccess=DataFrameAccess()
    dfaccess.setObsid(i)
    data=HcssConnection.get(dfaccess)
    if len(data) > 0:
        for j in range(len(data)):
            df = data[j]
            datad = DoubleIcd(df.getFrame())
            table[str(j)]=Column(datad)
            mine=AsciiTableTool()
            table.description="Sample spectra"
            mine.formatter.header=1
            mine.formatter.commented=1
            mine.formatter.commentPrefix="; "
            mine.save("sample_spectra_"+str(i),table)
```

Example 12.2. Database to ASCII tables for multiple spectra

The inner loop in the above example allows us to get each frame in an observation in turn and place it into a table "column". The outer loop takes the tables formed for each observation id and places them in an ASCII file called `sample_spectra_<obsid>.txt`. These are comma-delimited ASCII tables viewable in any text editor.

12.2.7.3. Downloading Dataframes from a Database Using a GUI

A somewhat more sophisticated method of accessing a database from within a DP session involves the use of a GUI interface such as a *DataSelector* tool. This is available via the *ProcessConnect* command. The next example provides a downloadable script that uses just such an interface for obtaining HIFI dataframes and is given as an example of how to include GUI components to download dataframes from a database.

```


from herschel.hifi.generic import *
from herschel.ccm.api import *
import java.lang.reflect
import javax.swing as swing
# The following defines a class we can then run in a DP session
class Hifids:
    def __init__(self):
        # Connect the processor so that we get data output to 'a'.
        self.pc = ProcessConnect("pc")
        # Now set up place for output of dataframe
        self.out = self.pc.getConnector("df-output")
        # Create an array which will hold HIFI data frames - up to 1000 of them
        self.a = java.lang.reflect.Array.newInstance(HifiDataFrame,1000)
        # Provide passage for the dataframes into 'a'.
        self.out.pass(self.a)
        # Now setup a user GUI for the process connector and a completion button
        self.win = swing.JFrame()
        self.win.contentPane.layout=java.awt.FlowLayout()
        self.win.contentPane.add(self.pc.getJComponent())
        choose = swing.JButton("Finished", size=(65,70), \
            actionPerformed=self.dataChoice)
        self.win.contentPane.add(choose)
        self.win.pack()
        self.win.show()

    def dataChoice(self, event):
        # This subroutine creates a table when the GUI's "Finished" button is clicked.
        table=TableDataset()
        table.description=("Data output")
        # Allow the table (output) to be seen within the session, not just the class.
        global table
        for j in range(1000):
            if (self.a[j] != None):
                datad = Double1d(self.a[j].getFrame())
                table[str(j)] = Column(datad)
        # ..and gets rid of the pop-up window to finish.
        self.win.dispose()

```

Example 12.3. An example GUI interface to a database



To use the program, download it into your JIDE session and hit the  button. Now, whenever you want to run the program during the rest of your DP session, type the following (e.g., at the IA>> prompt)

```
Hifids()
```



Warning

The above example script handles HIFI dataframes only for now, but is used as an illustration.

This brings up a window similar to that shown in Figure 12.3 - showing the "play" tab screen. You can browse the database with the button (bottom left), choose between dataframes or source packets under the "data" tab and get the data under the "play" tab. Dataframes associated with particular APID, building block ID or observation ID can be chosen (see Figure 12.4). A timeframe can also be indicated.



Figure 12.3. The Datasector tool **Play** tab.

Once the dataframes have been identified, they can be obtained by hitting the play button under the "play" tab. This is the single arrowed button to the left. The buttons on under this tab have similar functions to those on a DVD player! Once play is complete, hitting the Finished button exits the GUI and places the dataframes in a table available to the DP session of the user.



Figure 12.4. The Datasector tool **Data** tab.

Output for this program is placed in a TableDataset, called *table*, where one column holds a single 1D spectrum. This table is then available for use in the user's DP session.

12.2.8. Accessing Housekeeping (HK) Data

Assuming you have access to a database whose schema is compatible with the version of the software you are running (see above for information regarding schema evolution) then the HCSS package *binstruct* can be used to access housekeeping information. Housekeeping packets are dealt with in a somewhat different way to dataframes, but there are some similarities in structure.

12.2.8.1. Accessing HK Information For a Given Obsid

The following example illustrates basic housekeeping packet access for an observation with an obsid of 1400. The end product is a table with two columns, time in the first column and the housekeeping parameter value (raw) in the second column. Although an example relative to HIFI is given, this can be adapted to dealing with HK data from the SPIRE and PACS too.

```

# Import packages needed
from herschel.access import *
from herschel.access.util import *
from herschel.binstruct import *
from herschel.pus import *
# Look to access HK packets associated with obsid = 1400
pk = PacketAccess(1400)
# Connect to the default database to find the packets
hk_set = HcssConnection.get(pk)
# Create an empty Java array list - needed for the
# PacketSequence routine below.
arrList = java.util.ArrayList()
# Loop around adding the housekeeping dataset into our array
for x in range(len(hk_set)):
    arrList.add(hk_set[x])

# We can look at our array
print arrList
# ...but to get something sensible we need packets in a time order.
pseq = PacketSequence(arrList)
# Get a listing of the parameter types contained
print pseq
# Find packets in the sequence which contain information on
# temperatures within the focal plane unit
seq_FPU_Temp = pseq.select(TypeEquals("FPU_Temperatures"))
# Find out what parameters are contained in the selected packets
# by obtaining the housekeeping parameter names from the first
# selected packet in the sequence
par_FPU_Temp = seq_FPU_Temp[0].getParametersContained()
# Print out to the DP session the names of all the parameters contained
print par_FPU_Temp

# Choose the FPU Temperature parameter you want to get info on
# ...and get a time ordered set of housekeeping data for it
# The output file plot_fpu_hk is a TableDataset with one column for time
# (a Finetime of microseconds since 1 January 1958)
# and one for the value of the parameter (RAW rather than engineering
# value). Here we choose the parameter FPU_b_body_top for the table
# output and get the converted values (in degrees K)
plot_fpu_hk = seq_FPU_Temp.getConvertedMeasures(["FPU_b_body_top"])
time = DoubleId(plot_fpu_hk[0].data/1000000.0) # puts time into seconds
data = DoubleId(plot_fpu_hk[1].data)

# Now we can plot the timeline of the HK data over the
# time period of the observation (obsid=1400) by plotting the table
p = PlotXY(time, data, style=Style(line=8, color=Color.black))
# Resize the window
p.height = 400
p.width=600
# Give a layer/legend name...
p[0].name="time plot"
# ...and add a title
p.title.text="FPU temperature"

```

Example 12.4. Basic HK packet access

12.2.8.2. Accessing HK Data For a Given Time Period

We may be interested in looking at HK data for longer periods of time, e.g., over an extended period covering several observations within the same database. This is particularly useful when looking for trends in instrument data.

In the next example we show how HK data can be obtained for a set of parameters over a given time period entered as Java Dates. The example is specific to use with HIFI databases but provides a general illustration how HK data can be obtained from a HCSS database.



Note

Care needs to be taken that time periods being sampled are not too long since the HK data is held in memory and days of HK data can lead to an "OutOfMemory" error.

```
# Import needed packages for handling databases and HK data
from herschel.access import *
from herschel.access.util import *
from herschel.binstruct import *
from herschel.pus import *
# And this allows us to deal with times.
from herschel.share.fltdyn.time import *
# First we enter a start and stop time for HK information.
# We enter Java Dates, given as year (-1900), Month (-1),
# day, hour, minute, second.
# Our start_time is therefore 01:10:00 on 25 October 2004
start_time = java.util.Date(104, 9, 25, 1, 10, 0)
# stop_time is 01:15:00 on the same day
stop_time = java.util.Date(104, 9, 25, 1, 15, 0)
# Need to convert final numbers into a FineTime used in database.
start_1 = DateConverter.dateToFineTime(start_time)
# Date/time of start for plotted data
prod_date = DateConverter.fineTimeToDate(start_1)
# Ditto for stop time
stop_1 = DateConverter.dateToFineTime(stop_time)
end_date = DateConverter.fineTimeToDate(stop_1)

# Initialize some parameters
pk=0
hk_set = 0
# Get object ready for sorting packets.
pseq = PacketSequence()
# Set up the query for accessing packets of HK data
# Here we ask for packets with an APID of 1026, which carries
# HIFI HK data. The database identified by the user's
# properties is accessed for packets of this type
# between the given start and stop FineTimes.
pk = PacketAccess(1026,start_1,stop_1)

# Now we know where to look, we can get the packets!
# First we create an array with the packets in
hk_set = HcssConnection.get(pk)

# ...then we loop over the array to get the contents and
# put packets into our packet sequence
for x in range(len(hk_set)):
    pseq.add(PusTmSourcePacket(hk_set[x].getContents()))

# Now we get the parameters in the packets that we can plot.
seq_HIFI_HK = pseq.select(TypeEquals("HIFI_HK_rev_3"))
# Let's pick out some of them
mnemonics = ["HF_AH1_MXMG_V", "HF_AV1_MXMG_V"]

# ...and get their converted (physical unit) measurements.
# "plot_HIFI_HK" is a TableDataset with a first column measuring time

# and the next 2 columns holding the HK parameter values
# at those times. We can now plot any of the parameters versus
# time, or against each other, by picking out the appropriate
# column of the table.
plot_HIFI_HK = seq_HIFI_HK.getConvertedMeasures(mnemonics)

# This is what to do to set up the plot. Since time
# is in microseconds we convert it to
# seconds first.

# Get the first column and divide by 1 million
time = plot_HIFI_HK[0].data/1000000.0

# Let's measure time on the plot from the beginning of the observation....
# We subtract the initial time value
```

```

plot_time = time - time[0]

# We will plot the two voltages contained in columns 2 and 4
h_voltage = plot_HIFI_HK[1].data
v_voltage = plot_HIFI_HK[2].data
# Now plot the data
p = PlotXY(plot_time, h_voltage, style=Style(line=8, color=Color.black))

# Resize the window
p.height = 400
p.width=600

# Change the legend
p[0].name = "H Mixer Plot"
# Change the axis labels...
p.xaxis.title.text="Time (hours)"
p.yaxis.title.text="Mixer voltage [V]"

# ...and add a title
p.title.text="Plot of Mixer Voltages. Start: "+str(prod_date)+"
            "End: "+str(end_date)

# Now we can also overlay the second voltage trend in blue.
p[1]=LayerXY(plot_time, v_voltage, name= "V Mixer Plot", \
            style=Style(color=Color.blue))
    
```

The kind of output one can expect from this example is shown below.

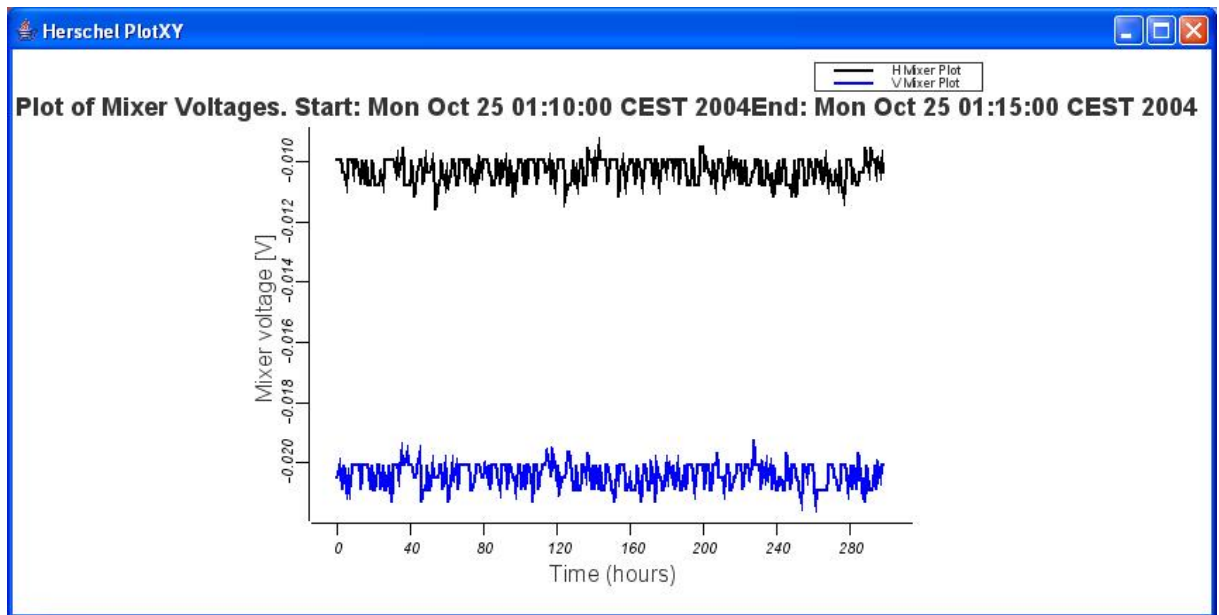


Figure 12.5. Sample timeline plot of HK data.

12.2.9. Removing a Database

Removal of a database that you have created can be done *at a terminal prompt* (not within the jide session).

```
>> removedb -rmdir mydatabase
```

Appendix A. Data Reduction Tutorial

-- contributed by Russ Shipman

A.1. Introduction

This is a quick start tutorial for getting a taste for the Data Processing. The purpose of this tutorial is to relay the flavour of the Herschel Data Processing software. This tutorial is written for Calibration Scientists and Instrument Engineers who require deeper access to the HCSS software. This tutorial should also provide a reasonable starting place for astronomers who simply want to have more control over the processing of their data.

The tutorial will focus on a number of general tasks listed below. Each topic will progress from the basic functionality to more complicated constructions. The tutorial will not (cannot?) shy away from the fundamental nature of the software and therefore avoids using any "Helper" functions which may hinder a deeper understanding of how the software works.

One last point. I am not a software developer. I am sure there are many subtleties which I have failed to appreciate. The only comfort I may give to the reader is that I am using JIDE at the same time as writing this tutorial so each line of code presented here actually runs.

This tutorial has the following outline:

- Reading a FITS file stored locally on disk. Section A.2.
- Understanding and interpreting data types. Section A.3.
- Numerical operations on data within . Section A.4
- Displaying the results. Section A.5
- Fitting models to data. Section A.7
- Writing scripts and procedures. Section A.6
- Saving the work and exporting the data to a FITS file Section A.8

A.2. Getting Data into Your Session

This section walks through an example of reading a FITS image or spectrum from a file on a local disk. More information of accessing FITS data structures may be found in User Manual Chapter 10.

The first step in retrieving a FITS data structure is to set up the access the FITS archive class.

```
from herschel.ia.io.fits.FitsArchive import *
from herschel.share.util import Configuration

#You will need to have access to the test data. For this we must set up the
#directory where the
#data are stored on you system.
dir=Configuration.getProperty('var.hcss.dir') + '/doc/ia/demo/data/'
#The first part of this is where HCSS is installed on your system, the second part
#is the path to the test data.
#
#note the unix systax for directories. If you are working on a Windows machine the
#actual directory name
#will look very strange, but the HCSS system will take care of that.
```

```
#
#Now create an instance of the class FitsArchive
fits = FitsArchive()
#Not all FITS files are created equal. The most general FITS structure can be
#found in the STANDARD_READER. Apply the STANDARD_READER to our FitsArchive.
fits.reader = fits.STANDARD_READER
#now the instance fits is able to read a generic fits file. This will return
#a "product".
#You will need to have access to the test
fitsproduct = fits.load(dir+"test.fits")
```

First, for more information the concept of PRODUCTS is described in Section A.3 below. From the example above you can see the flavour of Herschel Data Processing at a low level. We have accessed a general *FITS* utility for reading and writing FITS files. The utility has a method *load* specifically for reading FITS data.

OK, now it is your turn. Choose your favourite FITS file (image, spectrum, cube, table, etc) load it into your session. Be sure to specify the entire path to the file. The next section shows how to look into the product within your session and see its structure.

A.3. Products and Data Wrappers

Herschel data are carried about in structures called **products**. Products are software onions; they are made up of layers and each layer has its own description of the contents (e.g. labels of columns of tables, etc.), specific actions which can be taken with that layer and a history describing how the product was created. The software onion has the great benefit of being able to do all that its inner layers can do as well as the new items which are provided by the outer layer. This is called **inheritance**. Peeling away all the layers, will give a data structure containing actual data, be it an array or a single number.

To see what I'm talking about, let's go through the steps to create a product from double floating point sequence of numbers ranging from 0 to 9.

```
#Create our data:
#   range(10) gives the sequence of numbers from 0 to 9
#   DoubleId puts these numbers into a 1 dimensional double precision floating
#   point array
x = DoubleId.range(10)
#Wrap the Data x in an Array
array=ArrayDataset()
array.setData(x)
array.setDescription("Data-Onion")
#Now wrap the Array in a Product
prod=Product()
prod.setDescription("Product-Onion")
prod["Dataset-Onion"]=array
# To see what Dataset Inspector shows instantiate the Inspector
ds=DatasetInspector()
# Now tell the inspector what it should look at..
ds.register(prod,"prod")
#
```

Registering items one by one is not very useful, and also not the intention of the `DatasetInspector` which should show all the datasets and products within your Jide session. For this you should make use of the special `DatasetInspector` button provided to you by Jide itself (or alt-D).

This will bring up a window which shows two fields and initially two tags: Datasets and Products. Open the Products tag by double clicking on the word Products. The Products branch will be expanded to show all the products currently available in your session.

Double clicking on the name of a product within the `DatasetInspector` will give you details about it. Specifically, the tree will be expanded with two more branches on *Meta data* and another *Dataset*. The *Meta data* for the `fitsproduct` is the FITS header and has a clear connection to the data within the

product. The *PrimaryImage* contains the actual data array. By clicking on *PrimaryImage*, you can see the values of the elements and the dimension of the array itself.

Significantly more information about datasets and products can be found in User Manual Chapter 4.

Since a *Product* is a high level all-encompassing object, the data within the product still must be extracted. Both the data and the header are extracted in the following steps:

```
#Retrieve the 1st data field of a product using "default" and put it into a
#variable named fitsdata.
fitsdata=fitsproduct.default
#The FITS header is contained in the MetaData of the product. Put it into a
#variable called fitsheader
fitsheader=fitsproduct.getMeta()
```

The Session Inspector (Alt-I) gives a slightly different view into the session. It will show all the variables which are currently defined within your Jide session whereas the DatasetInspector only shows Datasets and Products. There will be a number of tags within the session inspector: Variables, Functions, Classes, and Packages. Open the "Variables" tag and look for the variables named "fitsdata" and "fitsheader". The variable "fitsdata" is a 4 dimensional array. Not what I expected, but the FITS header does say that it should be that way. I want a simple Double1d array which is the proper length of just my data.

```
#Now let's deal with those extra 3 annoyance dimensions.
from herschel.ia.numeric.toolbox.basic import Reshape
#
# Reshape with no parameters takes an "any" dimensioned array and
# turns it into a 1 dimensional array.
to1D=Reshape()
# Now apply it
spectrum=to1D(fitsdata.data)
# By the way the same results are possible by:
spectrum=Reshape()(fitsdata.data)
```

An extremely useful feature is the `.__class__` method. This method works on every object within Jide. The result of this method is the name of the class of the object. I wanted "spectrum" to be a one dimensional array. Let's find out what it is:

```
print spectrum.__class__
```

This will give the type of value we now have which should be a "herschel.ia.numeric.Int1d". This is saying that our spectrum at this point is an integer array. When examining the FITS header, this is exactly what the FITS file contains. In the next section, we'll go through the steps to apply the calibration information contained in the FITS header in order to create a floating point array, frequency scale and velocity scale.

A.4. Numerical Calculations

We now have a Double1d array of spectral data (That is what is present in the test.fits file provided with this tutorial). However, the values themselves are not so interesting without the frequency scale. This still has to be constructed from the fits header. So let's gather all the fits header information we need and put it into variables which we have some experience with. Please note that at this point, I heading back into a more traditional realm of processing and increasing the complexity since *ALL* parameters and values are contained in fitsheader.

```
#Extract FITS header information into variables
#
bscale = fitsheader.get('BSCALE').value
bzero = fitsheader.get('BZERO').value
crpix = fitsheader.get('CRPIX1').value
crval = fitsheader.get('CRVAL1').value
```

```
restfr = fitsheader.get('RESTFREQ').value
cdelt1 = fitsheader.get('cdelt1').value
altrpix= fitsheader.get('ALTRPIX').value
altrval= fitsheader.get('ALTRVAL').value
deltav = fitsheader.get('DELTAV').value
#We will also need the length of the data vector
naxis1 =len(spectrum)
#The len command is a built in Python command,
#Now create the frequency and velocity vectors
frequency=(Double1d.range(naxis1)-crpix)*cdelt1 + crval + restfr
# The velocity is recorded in the header as m/s, I want this in km/s
velocity=((Double1d.range(naxis1)-altrpix)*deltav + altrval)/1000.00
#Now convert the integers values of the spectrum to doubles.
spectrum = spectrum*bscale + bzero
#
```

I do believe that numerical operations are about the simplest part of HCSS. That is, however, my personal opinion.

It is now time to view the result of our efforts. That is the topic of the next section.

A.5. Plotting

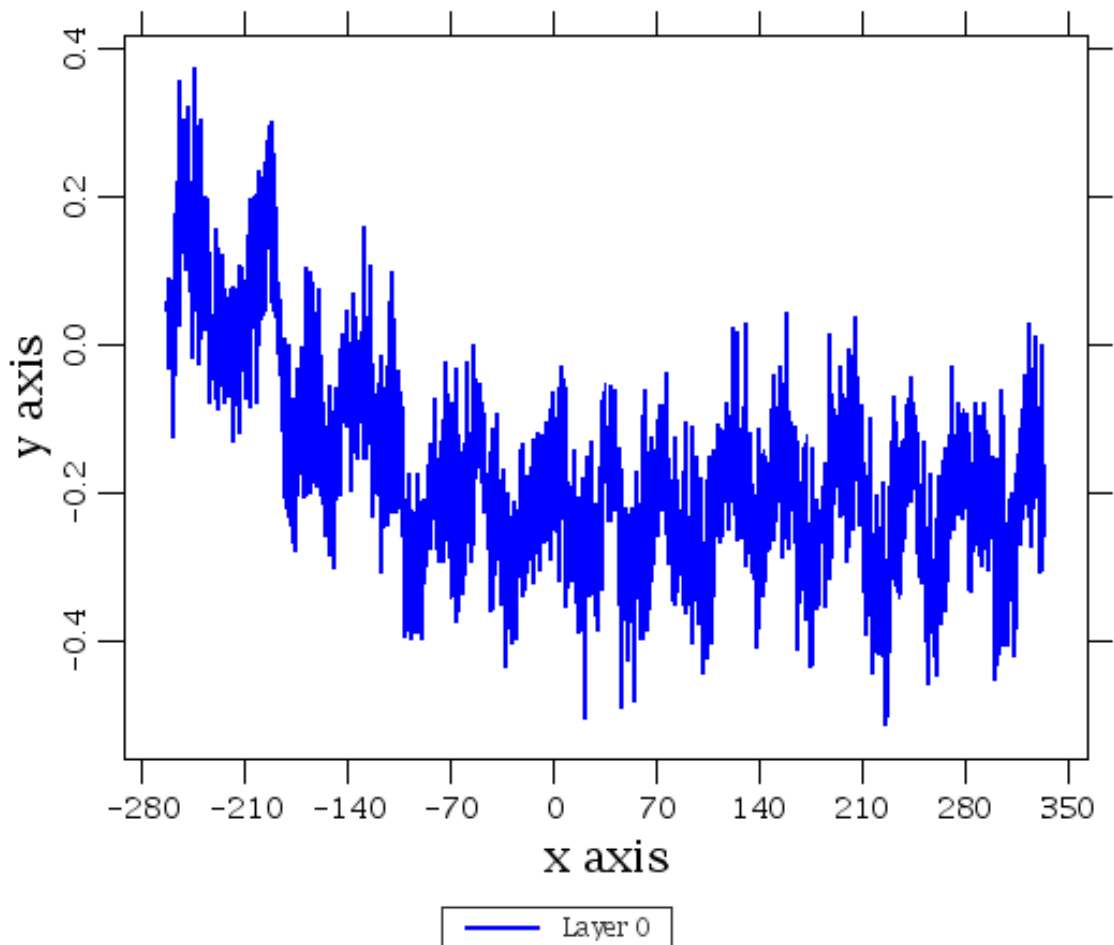
Plotting is easy and, although the system is still under development, is highly advanced. The simplest way to create a plot is the following:

```
from herschel.ia.gui.plot import *
# Simplest way to plot:
PlotXY(velocity,spectrum)
#
```

On this plot many items can be set or changed, via the properties window which activated by a right button click on the mouse. However, if you want to have multiple plots in your JIDE session, you should rather work on an instance of PlotXY instead of the main CLASS itself.

```
myplot = PlotXY(velocity,spectrum)
```

The above line produces the plot shown.



Simple plotting example.

Figure A.1.

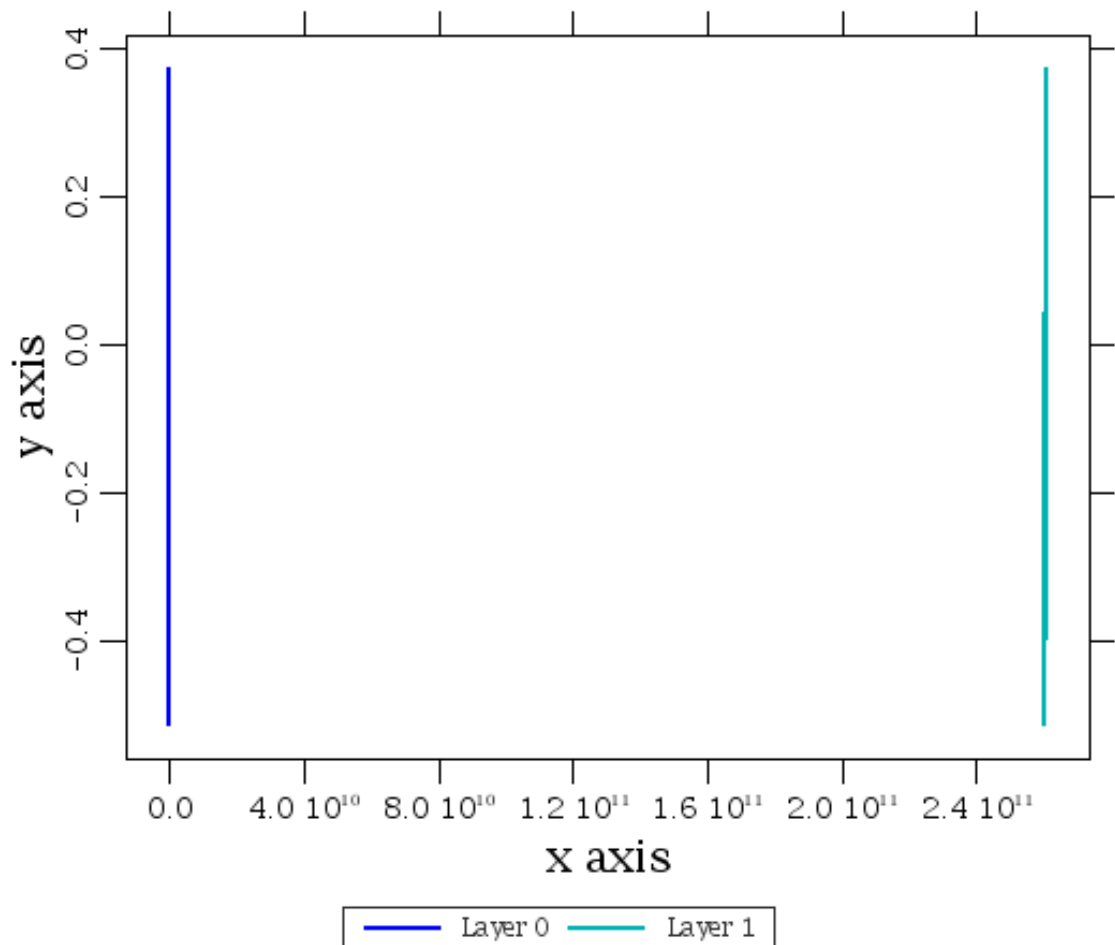
If you have multiple data sets to display on the same plot, these are added as layers. Layers give you significant control over your plot since they can be added and removed. Say for example you want to overlay the same data but offset by a small amount. This is done with:

```
myplot[1] = LayerXY(velocity,spectrum+0.1)
#but adding 0.1 was not enough to see both data sets. So remove the Layer and
#add it again.
myplot.removeLayerXY(1)
myplot[1] = LayerXY(velocity,spectrum + 0.4)
```

With the properties dialog box, it is possible to fully annotate the spectrum and axes as well as change line styles and plot symbols. The end result of all the work can be saved and restored for reuse.

I have a plot of my data as a function of velocity. I would also like to add an axis for the frequency itself. This is possible as another layer. One important point to keep in mind, layers can always be changed. If the first rendering of the layer is wrong the plot does not need to be thrown away and restarted. So let's build this other layer in steps.

```
#Clear out the layer 1 again. We could move on to other layers, but there
#is no real need yet.
myplot.removeLayerXY(1)
#Create another layer of the spectrum as a function of frequency.
myplot[1]=LayerXY(frequency,spectrum)
```



Not what I expected example.

Figure A.2.

This is a rather disturbing plot. Believe it or not, but this is what I had asked for, to plot on the same scale another data set with numbers both around 2.6 E11 and 200. I don't have to throw anything away. I only need to tell the plot that I want this layer (1) to have its own axis and not to have the two axes locked together.

```
#Give layer 1 a dummy axis.  
myplot[1].xaxis=Axis()  
#Currently the scales of the axes (0 and 1) are locked together.  
#To unlock Axis 1  
myplot[1].xaxis.lock=0  
#Still the scales are off, so rescale  
myplot[1].xaxis.autoRange=1  
#The range for layer 1 is OK but not layer 0  
myplot[0].xaxis.autoRange=1
```

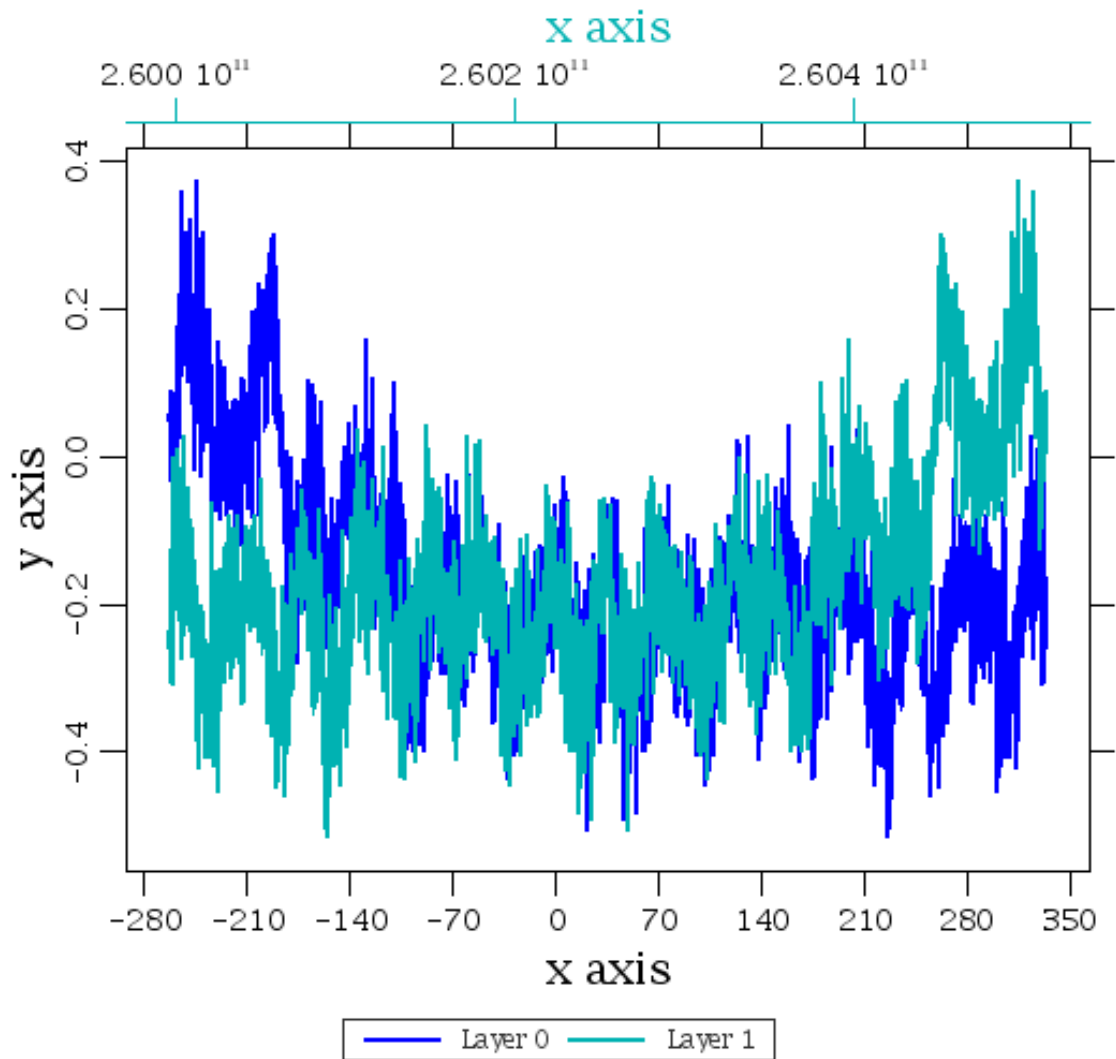


Figure A.3.

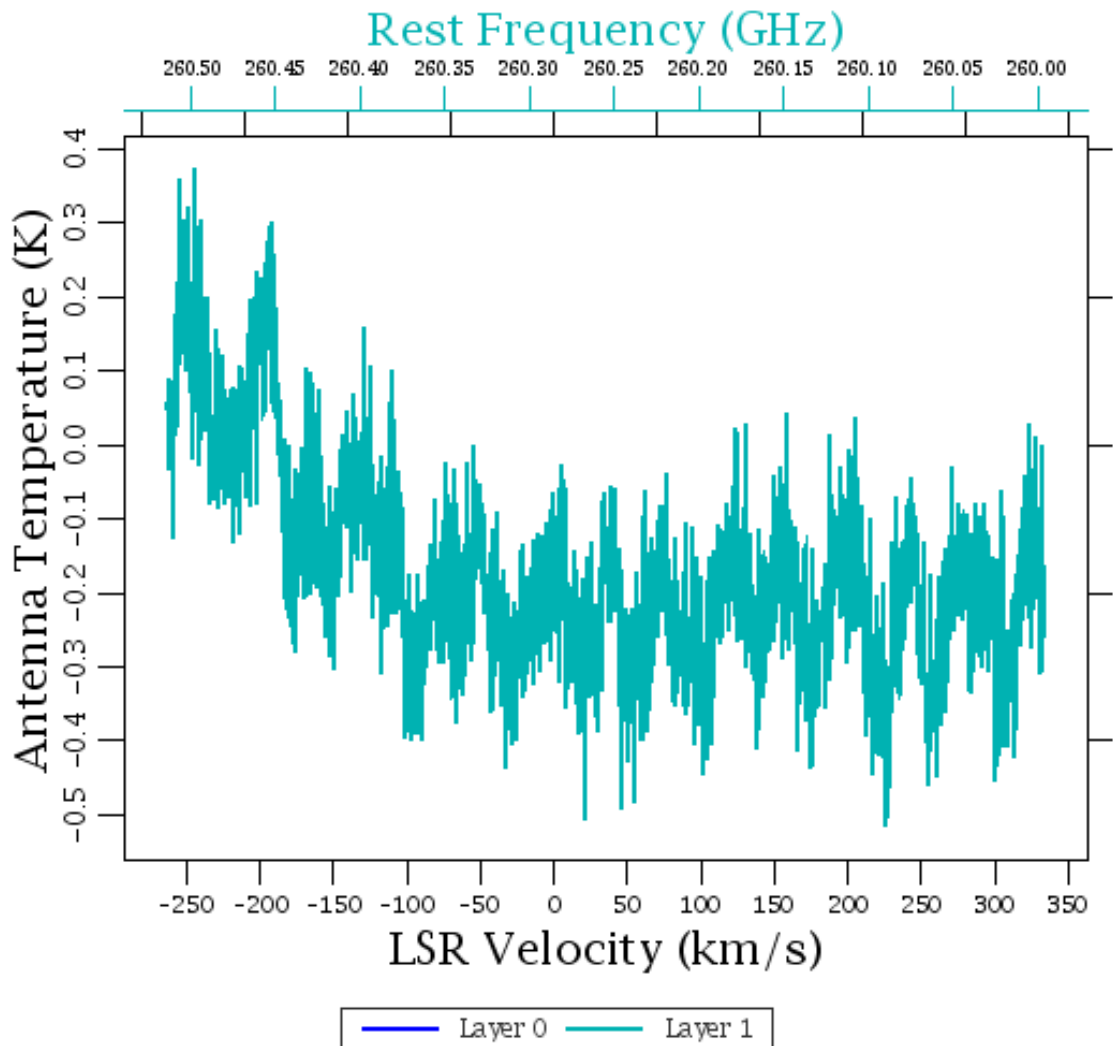
Things are looking better. They are not yet perfect, but better. As can be seen in the last plot, both plots of the same spectrum at least show up in the same window, but one is the inverse of the other. Velocity to frequency is a flip. Also, the scale on the frequency axis does not have enough precision in the display to show anything other than $2.6e+011$. The final problem is that there are no meaningful labels or annotations, just place markers. These can all be fixed as follows:

```
# The axis for layer 1 can be easily flipped.  
myplot[1].xaxis.inverted=1
```

The fix for the scale on layer 1 requires a change to the data i.e., the frequency scale itself. But this means a re-making of Layer 1. OK, I'll remove layer 1 (again) and add in a new layer 1.

```
#First change the units to GHz.  
freqGHz = frequency/1e9  
#Now remove layer 1  
myplot.removeLayerXY(1)  
#Now add the new layer 1  
myplot[1]=LayerXY(freqGHz,spectrum)  
#give the layer its own axis, otherwise the following steps will always use the same  
axis  
myplot[1].xaxis=Axis()  
myplot[1].xaxis.lock=0
```

```
myplot[1].xaxis.inverted=1
myplot[1].xaxis.autoRange=1
#Add real titles to Axes
myplot[0].xtitle='LSR Velocity (km/s)'
myplot[1].xtitle='Rest Frequency (GHz)'
myplot[0].ytitle='Antenna Temperature (K)'
#And a few items about the tick marks.
myplot[0].xaxis.getTick().setInterval(50.0)
myplot[1].xaxis.getTick().setInterval(0.05)
myplot[0].yaxis.getTick().setInterval(0.1)
```



Not what I expected example.

Figure A.4.

With the dialog box, you can change fonts and placement of all the titles and labels. For now I will just write the results to a PNG file (as I've been doing for all the figures so far) and move on.

A.6. Writing a Task

If the calculations are general or you do the same steps again and again, you will likely want to put make your steps available for later re-use. This can be done either by saving as a script, or by writing a Task which will perform your script but allow different parameters.

In the previous section, we saw that currently there are quite a few steps needed to plot two axes on a single plot. Let's try to make that part into a Task.

As I think about making a task, there are multiple ways a task can behave with respect to plotting multiple axes. The most straight forward in my mind, is passing a plotting task two x-axis vectors and on y-axis vector. The plot task then puts all three together in one plot. Another approach would be to pass a plot of an x-y pair (x axis vector and y axis vector already in a plot, and simply add the second x-axis. The second approach, if it can be done, is using the fact that plots are just objects themselves, they can be passed around and modified. I'll stick with passing a plotting task the three vectors I need.

First import the necessary libraries.

```
# Import task framework classes.
from herschel.ia.task.JTask import JTask
from herschel.ia.task import TaskParameter
from herschel.ia.task.api import SignatureEntry
from herschel.ia.gui.plot import *
```

Tasks are just CLASS definitions, but using a particular set of methods to define the input and output parameters which has some user support built in. We'll see these later.

Tasks consist of two parts, a preamble and an execute. Remember that Jython definitions are highly sensitive to spacings, so be sure to indent consistently within a definition (or loop). For ease of understanding, I show the entire Task below.

```
class Plot2XY(JTask):
#Creation method
#
#   def __init__(self, name="Plot2XY"):
#
#This is the preamble. Here I am defining the input parameters
#and what type they are. I define:
# xaxis1 as DoubleIeld, xaxis1 is the name of the parameter to be used below in
# the execute part.
# y as DoubleIeld (for the y-axis)
# xaxis1 also as DoubleIeld, this is also indicated as mandatory.
#
#   p=TaskParameter("xaxis1",valueType=DoubleIeld,mandatory=1)
#   self.addTaskParameter(p)
#   p=TaskParameter("y",valueType=DoubleIeld,mandatory=1)
#   self.addTaskParameter(p)
#   p=TaskParameter("xaxis2",valueType=DoubleIeld,mandatory=1)
#   self.addTaskParameter(p)
#
#assume 2nd axis is not inverted, but allow that to be changed
#invert becomes another parameter with initial value of 0 (False)
#
#   p=TaskParameter("invert",False,mandatory=0)
#   self.addTaskParameter(p)
#
# This task will return a modified PlotXY object.
# for the task, the parameter name is "plot" which will be the output.
#
#   p=TaskParameter("plot",valueType=PlotXY)
# This should be made the output of the task
#   p.setType(p.OUT)
#   self.addTaskParameter(p)
#
#Now define the execute part
#
#   def execute(self):
#
# Go through all steps needed to make two axis on a single plot.
# Note the notation self.name is the way to use parameters initiated above in
# the task.
#
#   self.plot = PlotXY(self.xaxis1,self.y)
#   self.plot[1] = LayerXY(self.xaxis2,self.y)
```

```
# Create the 2nd axis as separate from the first
self.plot[1].xaxis=Axis()
self.plot[1].xaxis.lock=0
self.plot[1].xaxis.autoRange=1
self.plot[0].xaxis.autoRange=1
#
# Now check if invert = 1, invert the 2nd axis
#
    if self.invert:
        self.plot[1].xaxis.inverted=1
```

Here is what this task produces.

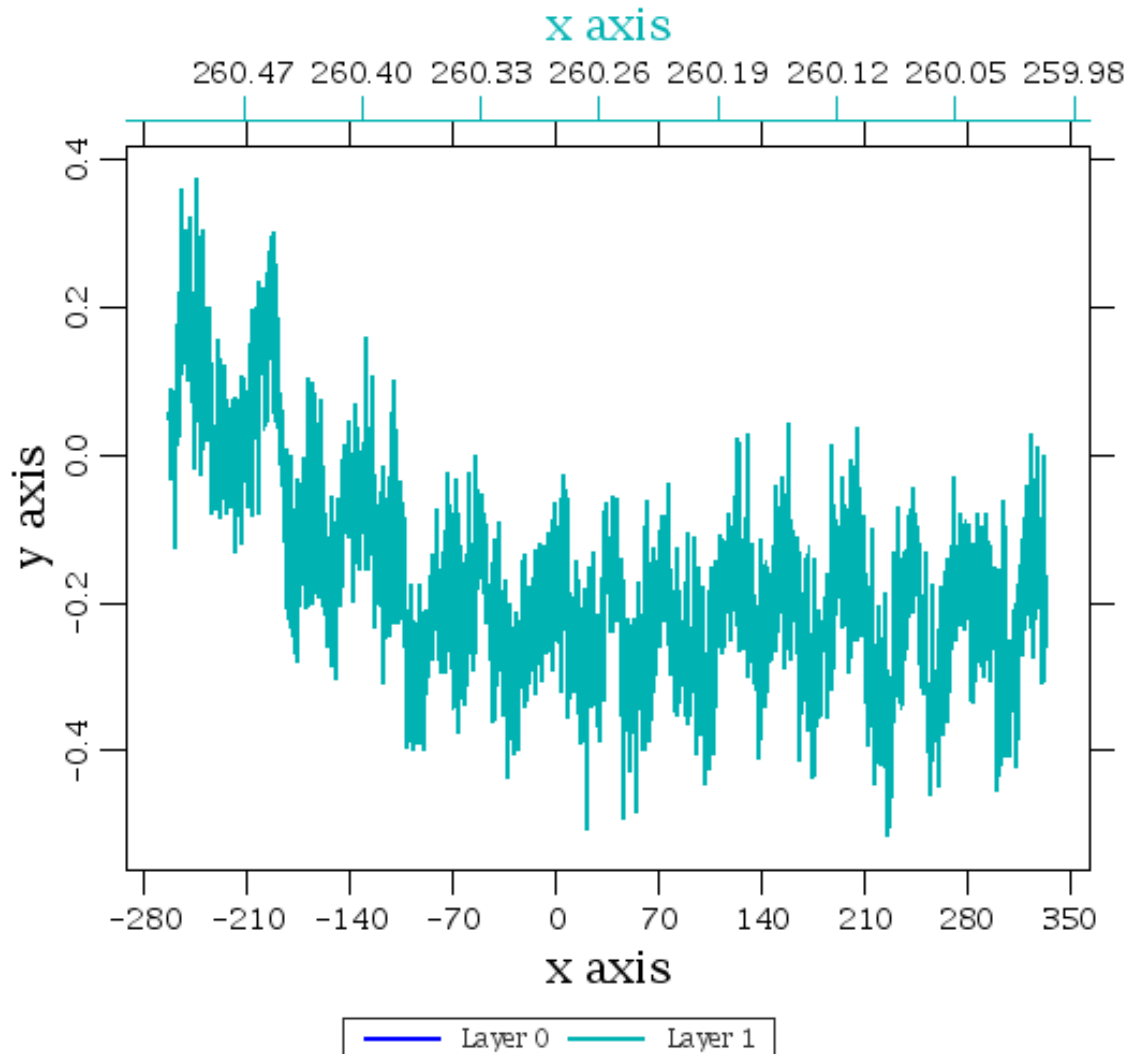


Figure A.5.

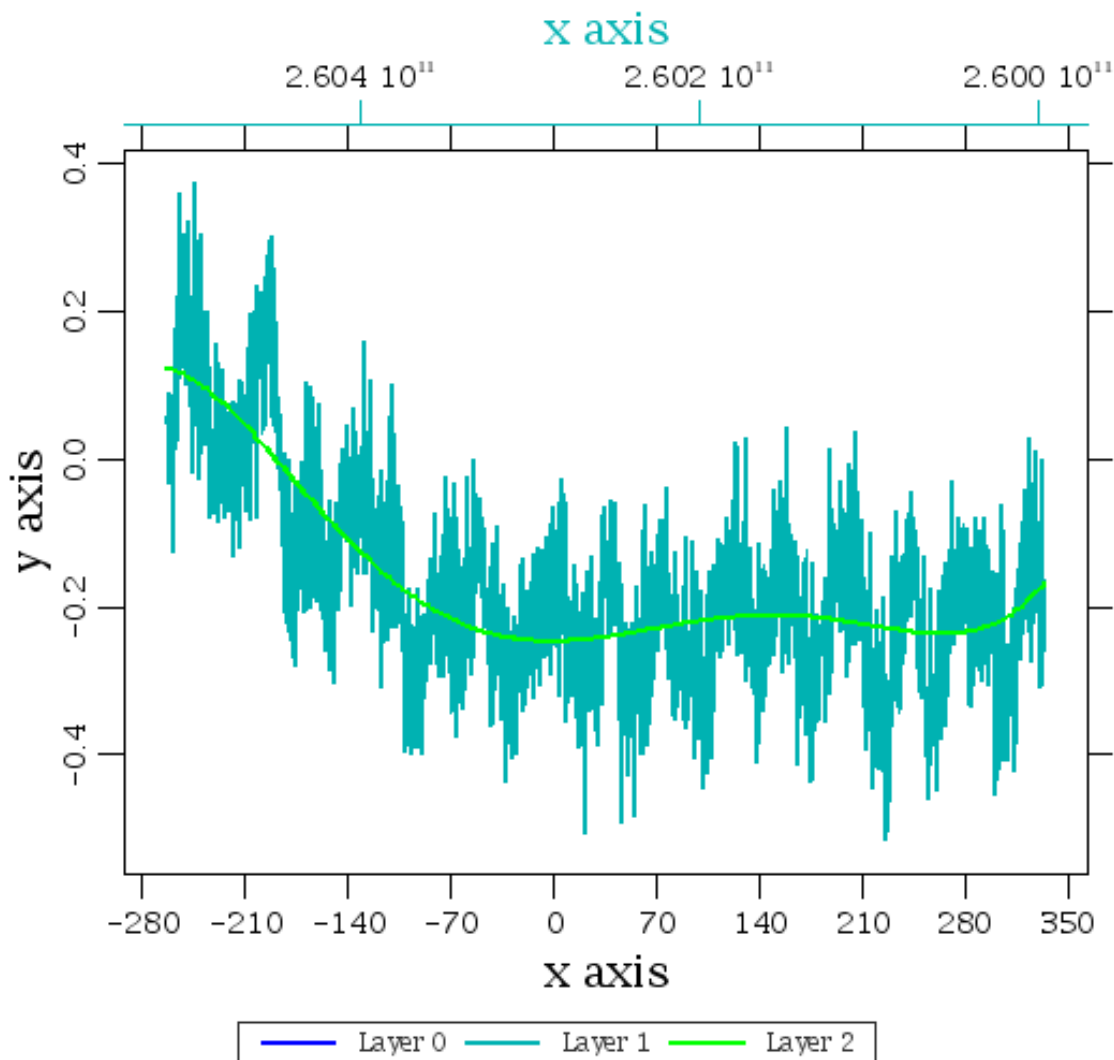
A.7. Fitting a Model

Fitting of models is of general interest, whether it is fitting a straight line or fitting a complex model. As can be seen, the spectrum has some serious problems: an unruly baseline to say the least. If I have reason to believe that the data are still salvageable, I could try to clean them up.

The first that I will try is to fit a polynomial to remove the drop from negative velocities and zero the spectrum in general. Fitting involves two conceptually different steps. The first is the model to be fit,

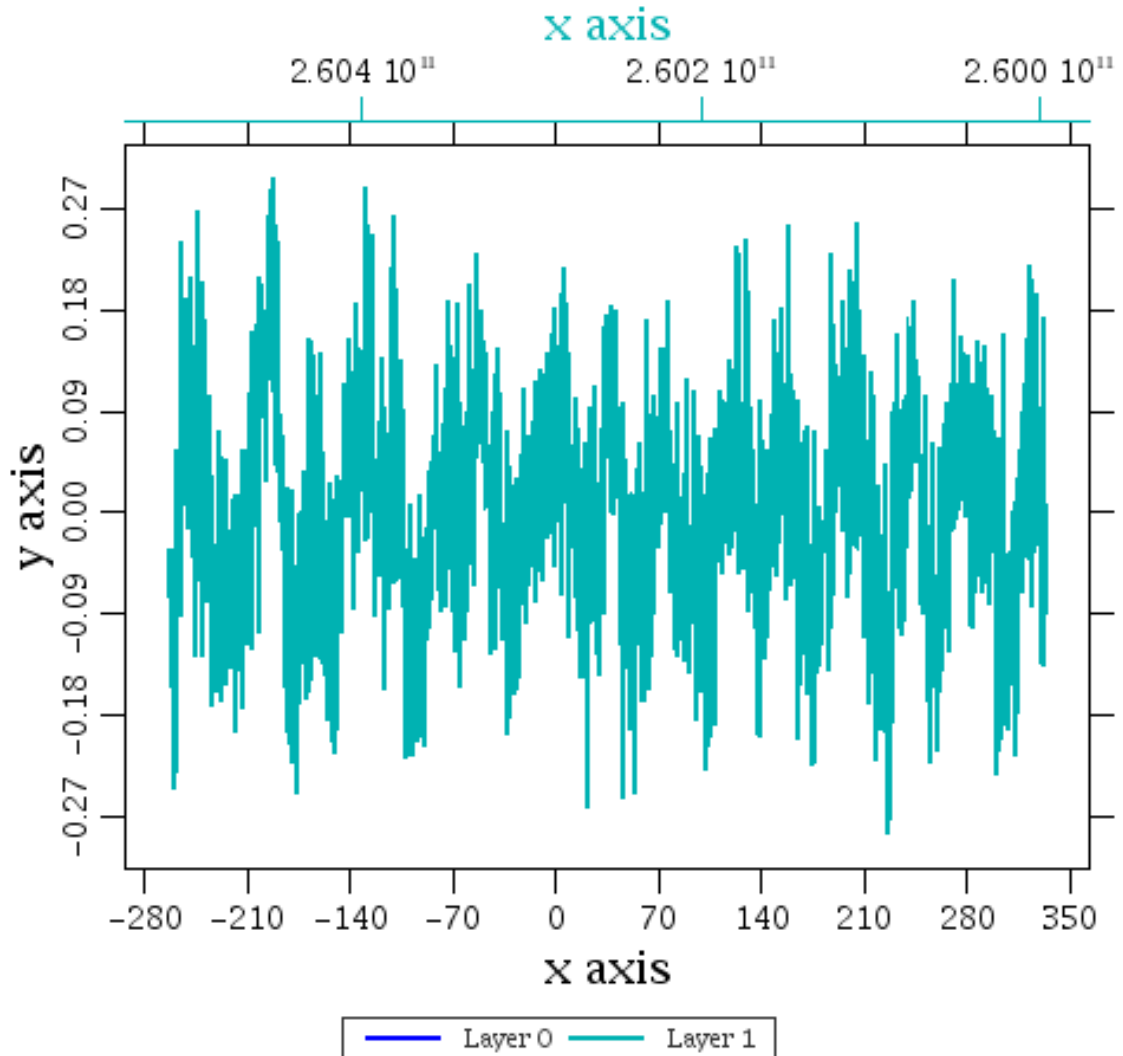
polynomial, sinusoid, whatever. The second concept is *how* to fit this model. Some models are linear and the fit is a straight forward matrix inversion. Others have to be done iteratively.

```
from herschel.ia.numeric.toolbox.fit import *
#
#I'll choose a 5th order polynomial model to smooth out
# the general trend in the spectrum
polymodel=PolynomialModel(5)
#now set up the fitter to use the model we've just defined.
linfit=Fitter(velocity,polymodel)
#just apply the fitter to the spectral data.
#Note that linfit knows that it is fitting a polynomial of 5th order
#to an array of velocity values. So at this stage the independent
#variable is not necessary.
params=linfit.fit(spectrum)
# lets take a look
print params
# And create a "data" array with the fit.
baseline=polymodel.result(velocity,params)
#let's plot things and have a look
plot=Plot2XY()(velocity,spectrum,frequency,1)
plot[2]=LayerXY(velocity,baseline)
```



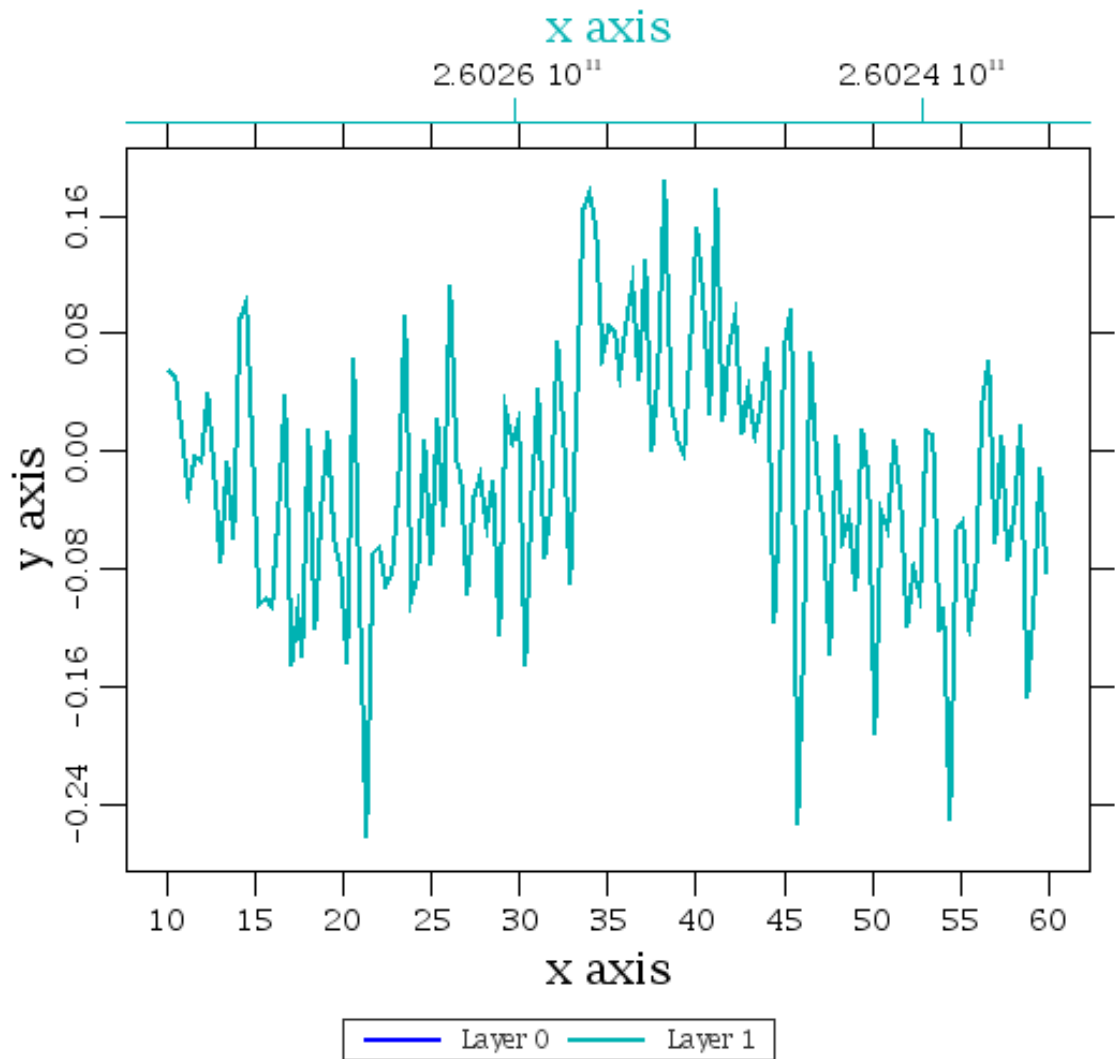
A 5th order polynomial baseline fitted to the data.

```
#I can even correct the spectrum now.  
spectrum1=spectrum-baseline  
plotc=Plot2XY()(velocity,spectrum1,frequency,1)  
#
```



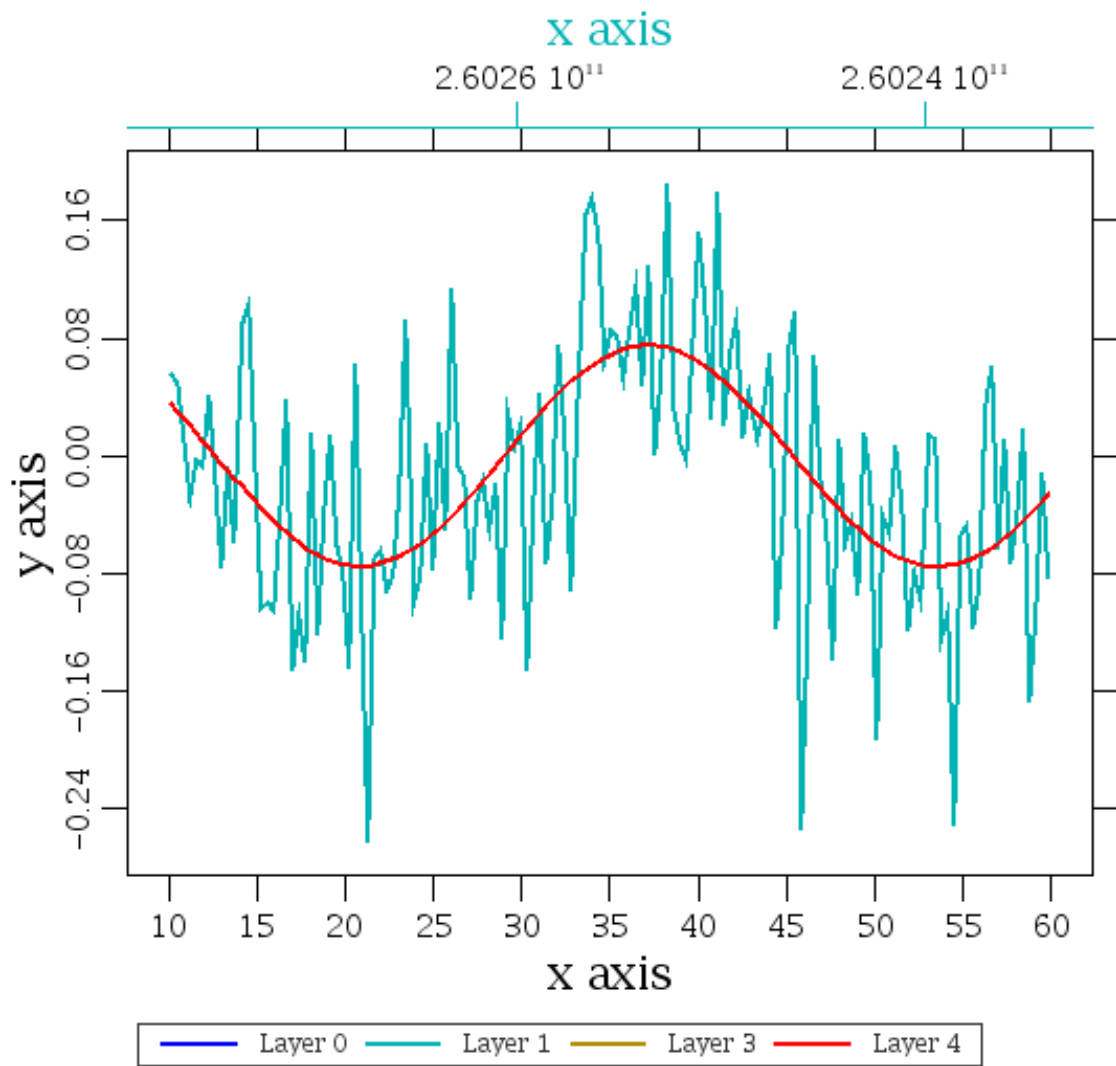
Spectrum after removal of polynomial baseline.

```
#Now we should try to remove a sinus to make the spectrum  
#flat.  
#I've already done this once and know that I cannot  
#get a good fit of a single frequency over a wide range  
#so I need to select out part of the spectrum. The  
#line I am interested in should be around 30 km/s. So I  
#can select the spectrum from say -40 to 80 km/s and still have  
#significant ripples to fit a sinus.  
#  
#The following >and < make boolean arrays  
q1=velocity > 10  
q2=velocity < 60  
#now identify the indices which correspond to the boolean  
q=velocity.where(q1.and(q2))  
ps=Plot2XY()(velocity[q],spectrum1[q],frequency[q],1)
```

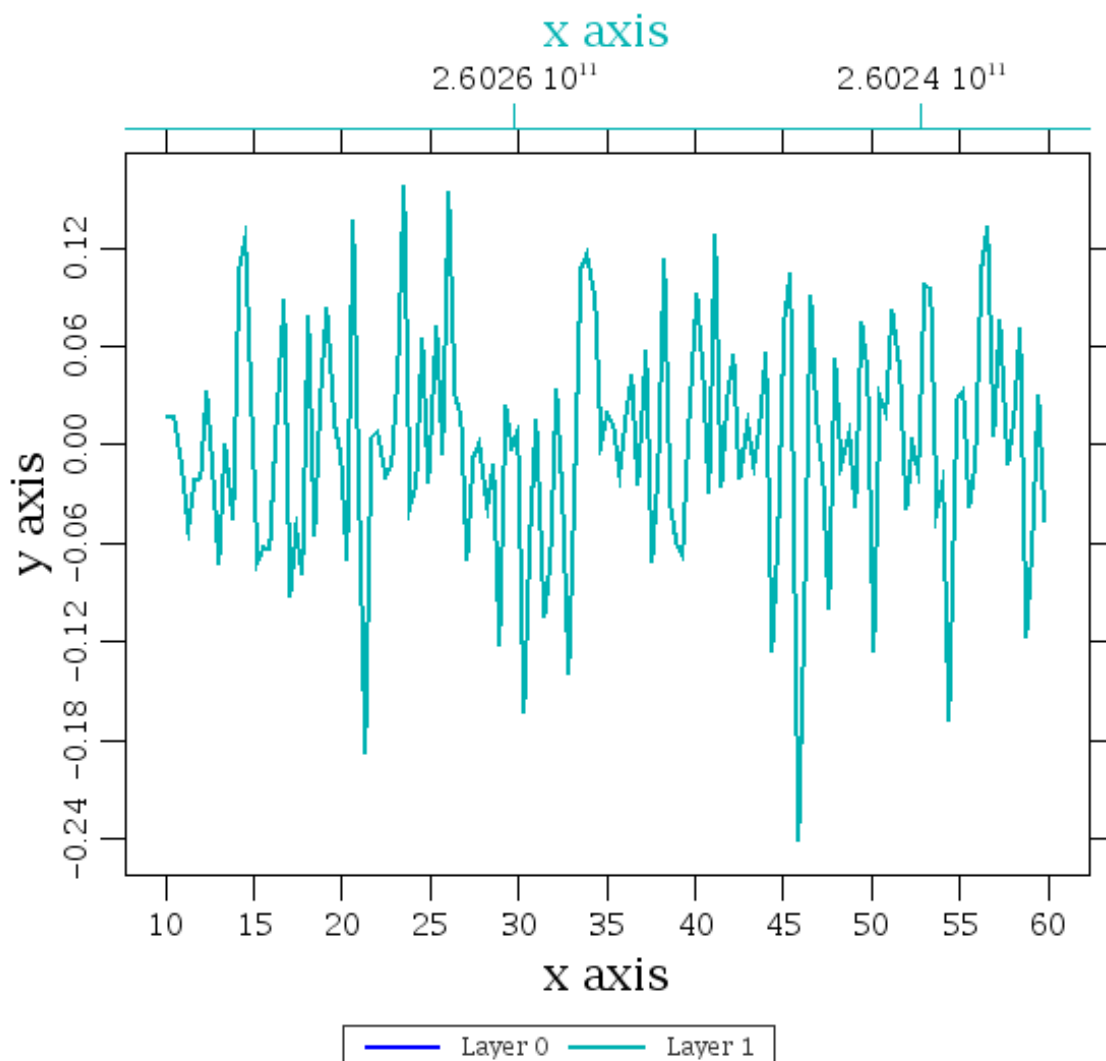
Closeup of spectrum.

```
#now we are ready for the fitting
#First identify the model which should be a sine model.
sine0=SineModel()
#and the fitter
lev0=LevenbergMarquardtFitter(velocity[q],sine0)
amoeba=AmoebaFitter(velocity[q],sine0)
#for our sine model we need an initial guess at the parameters
#Our sine wave goes through 1 period in about 30 km/s, so
# the initial guess at the frequency is 1./30.0,the amplitudes
# are around 0.1
#
param0=Double1d([1./30.0,0.1,0.1])
lev0.setParameters(param0)
lev0.setTolerance(0.000001)
#
amoeba.setSimplex(param0,Double1d([0.01,0.1,0.1]))
#
#
param2=lev0.fit(spectrum1[q])
param3=amoeba.fit(spectrum1[q])
ps[3]=LayerXY(velocity[q],sine0.result(velocity[q],param2))
ps[4]=LayerXY(velocity[q],sine0.result(velocity[q],param3))
```



Sine fit to the spectrum.

```
print param2 , param3
spectrumfixed=spectrum[q]-sine0.result(velocity[q],param3)
#
#
off = SUM(spectrumfixed)/len(spectrumfixed)
spectrumfixed=spectrumfixed-off
pgood=Plot2XY()(velocity[q],spectrumfixed,frequency[q],1)
```



Sine removed from the spectrum.

A.8. Saving Data and Session

Of course is also possible to save the work you have done. To save all the variables defined in the entire session just use the save command.

```
#Saving all variables, datasets and products in a file: mysave.save
save('mysave.save')
#OR save only the velocity and spectrum
save('mysave.save','velocity,spectrum')
#
```

We can also save the corrected spectrum as a fits file. First we should make the onion again. At the core of the onion was our array of data.

```
#Create a corrected array of fluxes
spectrum_corr = spectrum - baseline - sine0.result(velocity,param3)
```

Let's put this result into a HCSS defined spectrum with a column for fluxes, frequencies (or wavelengths) and velocities.

```
myspec=Spectrum1d()
```

```
myspec.setFlux(spectrum_corr)
myspec.set('Wave',frequency)
myspec.set('Velocity',velocity)
```

Now I want to update the header information to reflect the changes I've made i.e. all the fit parameters.

```
# Add fit information to header key words.
# Make a copy of the original header
newheader=fitshead.copy
#First the polynomial fit
for i in range(6):
    newheader.set("poly"+ i.toString(),DoubleParameter(params[i]))
#The the sine fit
for i in range(param3.length()):
    newheader.set("sine"+i.toString(),DoubleParameter(param3[i]))
# Add the new header to the metadata of of the spectrum dataset
myspec.setMeta(newheader)
```

Now, this dataset can be wrapped into a product.

```
#Create a product from the spectrum dataset
myspectrum_prod=Product()
myspectrum_prod("Spectrum Dataset",myspec)
```

And saved as a FITS file.

```
#now save is as a FITS file
fits.save('corr_spectrum.fits',myspectrum_prod)
```

That's it. A later version of his Tutorial will demonstrate how to save into a Pool from the Product Access Layer.

Appendix B. Example User's Property File

An example properties file to be placed in the file `${HOME}/.hcsm/myconfig` for UNIX users or `C:\Documents and Settings\\hcsm.props` for Windows users.



Note

`${HCSS_DIR}/config/devel.props` is the file which contains the system default properties, whereas `${HOME}/.hcsm/myconfig` is the file which contains your properties.

For most users, the first few lines are the most important ones.

```
var.hcsm.workdir=C://temp
hcsm.access.ccm = herschel.versant.ccm
hcsm.access.query.navigate=true
hcsm.access.database = ilt_qm_9${var.database.server}
dbname = ilt_qm_9
dbfactory = herschel.versant.store.StoreFactoryImpl
hcsm.pg.useList = true
hcsm.pg.xml.listLocation = ${var.hcsm.dir}/config/defns,
    ${var.hcsm.dir}/../config/defns
****add old myconfig
# HCSS Properties File - location SRON
#
# Author: Craig Porrett
#
#To show queries submitted by access
#hcsm.store.verbosity = 1
# General
var.database.server = @lin-sron-02.sron.rug.nl
var.database.devel=tony_hcsm${var.database.server}
dbfactory = herschel.versant.store.StoreFactoryImpl
dbname = tony_hcsm
hcsm.cus.database=tony_hcsm@${var.database.server}
hcsm.cus.instrument = HIFI
hcsm.cus.tabledir = ${user.dir}/CUS/custables
var.hcsm.dir=C:/ia/hifi/lib/hcsm
#var.hcsm.workdir = ${user.home}/
#changed by Peer on 27-09-2002
#var.hcsm.dir = ${user.home}/hcsm_builds/latest_build
#var.hcsm.dir = /Users/users/hcsm_bld/hcsm_builds/latest_build
# Access
hcsm.access.database = ${var.database.devel}
hcsm.access.test.database = ${var.database.devel}
hcsm.access.connection = herschel.access.db.LocalConnection
hcsm.access.network = socket
hcsm.access.socket.host = localhost
hcsm.access.socket.port = 8050
hcsm.access.url = http://lin-sron-02.sron.rug.nl:5019/servlets/
hcsm.access.packetprocessor = HIFI
hcsm.access.instrumentmodel = Engineering
hcsm.access.factory.query = herschel.access.db.VersantQueryFactory
hcsm.access.router.host = localhost
hcsm.access.router.port = 9877
hcsm.access.query.allpks = select selfoid from
    herschel.versant.ccm.TmSourcePacketImpl
hcsm.access.query.alldfs = select selfoid from herschel.versant.ccm.DataFrameImpl
# CCM
hcsm.ccm.test.database = ${var.database.devel}
# following from Kevin's email on 29th Jan. 2004 siteid = 1 for hifi-icc
# this following from the ICD
hcsm.ccm.siteid = 1
hcsm.ccm.mission.config = democonfig
hcsm.ccm.mission.database = ${var.database.server}
# Formatter
```

```

# formatter package needs to be changed to use the var.hcss.dir system
hcss.formatter.directory.root = ${var.source.dir}
# MIB
var.mib.defns = ${var.hcss.dir}/data/mib/defns
var.mib.data = ${var.hcss.dir}/data/mib/example-mibs/example-1
var.mib.aux = ${var.mib.data}/auxil
var.mib.raw = ${var.mib.data}/ascii-tables
hcss.mib.database = ${var.database.devel}
#hcss.mib.database = hcssbld_hcss
hcss.mib.datadir = ${var.mib.raw}
hcss.mib.tablelist = ${var.mib.aux}/tablelist
hcss.mib.tc_command_durns = ${var.mib.aux}/tc-durns
hcss.mib.tm_param_list = ${var.mib.aux}/tmparams
hcss.mib.test_tc_command_list = ${var.mib.aux}/tcmds
hcss.mib.test_tm_param_list = ${hcss.mib.tm_param_list}
hcss.mib.tabledefs = ${var.mib.defns}/table-defns/
hcss.mib.dbroot = hcss_mib_root
hcss.mib.uplink_id = 1
hcss.mib.test_uplink_id = 1
hcss.mib.downlink_id = 1
hcss.mib.test_downlink_id = 1
hcss.mib.erroronly = false
hcss.mib.logfile = mibchecker.log
hcss.mib.readallcmds = true
hcss.mib.tc_command_list = xxx
# TM Ingest
hcss.tmingest.database = ${var.database.devel}
hcss.tmingest.port = 9877
# TM Proc
# Store
hcss.store.test.database = ${var.database.devel}
#ia dataflow
herschel.ia.dataflow.maxbuffersize = 50
# pcss needed for ia demo 28th January
hcss.mib.cus_file = gencus_scripts.out
hcss.mib.instrument = HIFI
#hcss.ccm.mission.config = democonfig
#hcss.ccm.mission.database = hcssbld_hcss@lin-sron-02.sron.rug.nl
# binstruct
hcss.binstruct.ip_filename = instr_props.ip
hcss.binstruct.tm_version_map=TmVersions.tbl
hcss.binstruct.mib=C:/ia/binstruct
hcss.binstruct.services = herschel.binstruct.mib.MibAsciiServices
hcss.binstruct.mib_source = ascii
# JConsole
hcss.jython.user.import=${user.home}/iltscripts_qm_reports.py
hcss.jconsole.buffer.size=320000
hcss.jconsole.prompt = "Tony's IA>>"
hcss.jconsole.width = 900
hcss.jconsole.height = 600

```

Appendix C. Jython Operators

The following tables shows all the various operators you can use in Jython. For completeness we have also listed one operator introduced in the latest development version of Jython (2.2 alpha) but absent from the stable version (2.1).

This list and the associated operator descriptions have been largely taken from the Python Reference Manual, which you can find online at <http://docs.python.org/ref/>.

Table C.1. Jython unary arithmetic operators

Operator	Operator description	Example
+	Unary plus: yields its numeric argument unchanged.	print +5 # 5
-	Unary minus: yields the negation of its numeric argument.	print -5 # -5
~	Invert: yields the bitwise invert of its plain or long integer argument.	print ~5 # -6

Table C.2. Jython binary arithmetic operators

Operator	Operator description	Example
+	Sum: yields the sum of its arguments.	print 2 + 2 # 4
-	Subtraction: yields the difference of its arguments.	print 2 - 3 # -1
*	Multiplication: yields the product of its arguments.	print 3 * 2 # 6
/	Division: yields the quotient of its arguments.	print 5 / 2 # 2 print 5.0 / 2 # 2.5
//	Floor division (Jython 2.2 alpha only): yields the result of the <code>floor()</code> function applied to the quotient of its arguments.	print 5 // 2 # 2 print 5.0 // 2 # 2.0
%	Modulo: yields the remainder from the division of its arguments.	print 5 % 2 # 1
**	Power: yields its left argument raised to the power of its right argument.	print 5**2 # 25

Table C.3. Jython shifting operators

Operator	Operator description	Example
<<	Left shift: <code>a << b</code> shifts plain or long integer <code>a</code> by <code>b</code> bits.	print 5 << 1 # 10
>>	Right shift: <code>a >> b</code> shifts plain or long integer <code>a</code> by <code>b</code> bits.	print 5 >> 1 # 2

Table C.4. Jython binary bitwise operators

Operator	Operator description	Example
&	Bitwise AND: yields the bitwise AND of its plain or long integer arguments.	print 5 & 6 # 4
^	Bitwise XOR: yields the bitwise exclusive OR of its plain or long integer arguments.	print 5 ^ 6 # 3
	Bitwise OR: yields the bitwise inclusive OR of its plain or long integer arguments.	print 5 6 # 7

Table C.5. Jython comparison operators

Operator	Operator description	Example
<	Less than: a < b yields true if a is less than b.	print 5 < 6 # 1
>	Greater than: a > b yields true if a is greater than b.	print 5 > 6 # 0
==	Equal to: a == b yields true if a and b are equal.	print 5 == 6 # 0
>=	Greater or equal to: a >= b yields true if a is greater than or equal to b.	print 5 >= 6 # 0
<=	Less or equal to: a <= b yields true if a is less than or equal to b.	print 5 <= 6 # 1
!= (preferred) or <>	Not equal to: a != b yields true if a is not equal to b.	print 5 != 6 # 1 print 5 <> 5 # 0

Table C.6. Jython boolean operators

Operator	Operator description	Example
and	Boolean AND: yields True if both arguments are true, False otherwise.	print 1 and 0 # 0
or	Boolean OR: yields True if at least one argument is true, False otherwise.	print 1 or 0 # 1
not	Boolean NOT: yields True if the argument is false, False otherwise.	print not 1 # 0

Appendix D. Demo script

D.1. Introduction

This is a collection of many (but not all) of the available scripts all over the system. The collection is organized by package.

D.2. Demonstrations illustrating specific functionality

Demo Files

<code>simple.py</code>	Overview of Jython capabilities
<code>help_demo.py</code>	Demonstration of help facility
<code>session_inspector.py</code>	Demonstration of session inspector facility.
<code>logging_demo.py</code>	Demonstration of jconsole's message logging facility.
<code>save_restore_demo.py</code>	Demonstration of the save and restore of data feature in Jconsole.
<code>numeric_whatisNew.py</code>	Demonstration of how to use the new functionalities of the numeric library from Jython.
<code>numeric_demo.py</code>	Demonstration of how to use the 1D functionality of the numeric library from Jython
<code>numeric_2D_demo.py</code>	Demonstration of how to use the 2D functionality of the numeric library from Jython
<code>numeric_reshaping.py</code>	Demonstration of how to use the reshaping functionality of the numeric library from Jython
<code>numeric_shifting.py</code>	Demonstration of how to use the shifting functionality of the numeric library from Jython
<code>numeric_slicing.py</code>	Demonstration of how to use the slicing functionality of the numeric library from Jython
<code>convolution_demo.py</code>	Demonstration of how to use the convolution functions in the numeric library
<code>fit_demo1.py</code>	1) Demonstration of how to perform fitting from the numeric library
<code>fit_demo2.py</code>	2) Demonstration of how to perform fitting from the numeric library
<code>fit_demo3.py</code>	3) Demonstration of how to perform fitting from the numeric library
<code>fit_demo4.py</code>	4) Demonstration of how to perform fitting from the numeric library
<code>fft_demo.py</code>	Demo of FFT functionality

<code>boxcar_demo.py</code>	Boxcar filtering demo
<code>gaussian_filter_demo.py</code>	Gaussian filtering demo
<code>interpolate_demo.py</code>	Demonstrates interpolation
<code>matrix_demo.py</code>	Demo of matrix functions
<code>dataset_demo.py</code>	Demonstration of how to use datasets and create products
<code>ascii_demo.py</code>	Demonstration of Import/Exporting of ASCII tables, the data file <code>ascii_demo_data.txt</code> is also required to run this demo
<code>fits_demo.py</code>	Demonstration of Import/Exporting of FITS data
<code>imageExample1.py</code>	Demonstration of general image functionality, the image file <code>ngc6992.jpg</code> is also required in your home directory to run this demo
<code>imageExample2.py</code>	Shows how to create an image from a simple numeric 2d array.
<code>task_example.py</code>	Demonstration of how to write a task
<code>task_array.py</code>	Demonstration on how to pass an array to a task.
<code>task_stop.py</code>	Demonstration on how to stop a task.
<code>TestPlotXY.py</code>	Demonstration of the new PlotXY capabilities
<code>TestAxis.py</code>	Demonstration of how to use PlotXY Axis
<code>TestLayerXY.py</code>	Demonstration of how to use PlotXY Layers
<code>TestAnnotation.py</code>	Demonstration of how to use PlotXY Annotations
<code>TestStyle.py</code>	Demonstration of how to use PlotXY Styles
<code>TestCompositePlot.py</code>	Demonstration of how to compose Plots(XY)
<code>TestMemory.py</code>	Demonstration of how PlotXY use memory efficiently

Appendix E. Naming Conventions

for Java and Jython users and developers. Version 0.3, 6th December 2006

Element	Description	Naming convention
Class <i>UM section 3.14.1</i>	Defines the state and behaviour of something. Classes are defined as declaring variables (fields) and functions (methods) associated with the objects of that class.	Names should be nouns and written in mixed case starting with an upper case letter. Do not use underscores to separate words. DataFrameGenerator, FitsArchive
Interface <i>UM section 3.14.2.1</i>	Defines a collection of method definitions and constant values. It can later be implemented by classes that define this interface with the implements keyword.	Names have the same convention as class names but are preferably adjectives. Try to end the names with -able or -ible: Sortable, Accessible, Savable
Variable	An item of data named by an identifier. Each variable has a type, such as int or Frame, and a scope.	Names should be mixed case starting with a lower case letter. Do not use underscores to separate words. frameBufferCounter, nSamples, line, detectorNo
Instance Variable <i>UM section 3.14.1</i>	A variable that is part of an object. For the rationale of this naming convention see HSCDT/TN009 on ESA Livelink	Names should start with an underscore, otherwise follow the general conventions for variables (see above). _packetType, _isVisible
Local Variable	A variable that is part of a function or method.	Names follow the naming convention of normal variables. counter, length, pixelName
Constant	A variable whose value that can not be changed during execution.	Names should be all uppercase using an underscore to separate words: MAX_ITERATIONS
Boolean variable and method	A logical type/function that can only have or return the values 'true' or 'false'. Methods have parentheses () while variable haven't.	Names should start with is-, has-, can-, or should-. isVisible, hasChanged(), canHandle(), shouldAbort
Parameter		

Element	Description	Naming convention
	An argument to a function or a method.	Names follow the naming convention of normal variables. name, packet
Property <i>UM section 1.5</i>	A platform independent implementation of environment variables and settings.	Names should be all lower case and start with 'hcss'. The hierarchical parts should be separated with a dot. hcss.binstruct.services
Method <i>UM section 3.14.1</i>	A function defined in a class.	Names should be verbs and written in mixed case starting with a lower case letter. Do not use underscores to separate words. getName(), load()
Function <i>UM section 3.12</i>	A jython function is a collection of code lines to perform a specific task under one name. Functions take arguments and provide one output. They are like methods, except they are not inside a class. A function can also be an instance of the Task class.	Names follow the same convention as method names in classes. resample(), readTm()
Numeric function <i>UM section 5.4</i>	Parameterless Java functions provided by the herchel.ia.numeric toolboxes. For these function only one instance is needed. Other numeric functions follow the same convention as classes.	Names are in all uppercase with an underscore to separate words. UNIQ, MEDIAN, IS_FINITE
Task <i>UM chapter 8</i>	A Task is a class which can be called as a function. Tasks do input and output parameter type checking and provide history to Products.	Names follow the same conventions as for classes. Task names should end with the word 'Task'. DisplayDataFrameTask, ResampleTask
Package <i>UM section 3.14.4</i>	Defines a collection of related classes and interfaces in Java. Packages provide the namespace in Java and Jython.	Names should be in lower-case letters and digits, don't use underscores. herchel.ia.numeric Package names should be short so that the fully qualified package name doesn't become excessively long.

Abbreviations and acronyms should **not** be all uppercase when used as a name:

GOOD	BAD
<code>exportAsHtml()</code>	<code>exportAsHTML()</code>
<code>saveAsJpeg()</code>	<code>saveAsJPEG()</code>
<code>OolPacket</code>	<code>OOLPacket</code>

Using all uppercase for the abbreviations in base names will give conflicts with the naming conventions given above. A variable of this type would have to be named `hTML`, `jPEG` etc. which obviously is not very readable. Another problem is illustrated in the examples above: when the name is connected to another, the readability is seriously reduced, since the word following the acronym does not stand out as it should.

The term *compute* can be used in methods where something is computed and might take considerable time to execute.

```
computeAverage(), matrix.computeInverse()
```

Give the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.

The 'n' prefix should be used for variables representing a number of objects, note that the names are plural.

```
nPoints, nLines, nSamples
```

The notation is taken from mathematics where it is an established convention for indicating a number of objects. Note that Sun uses the `num` prefix in the core Java packages for such variables. This is probably meant as an abbreviation of number of, but as it looks more like number it makes the variable name strange and misleading. If "number of" is the preferred phrase, `numberOf` prefix can be used instead of just `n`. The `num` prefix must not be used.

The 'No' suffix should be used for variables representing an entity number.

```
tableNo, employeeNo
```

The notation is taken from mathematics where it is an established convention for indicating an entity number.

Reserved words: the following words are reserved by Java as language keywords and can not be used for variables, methods or class names in Java.

```
abstract, continue, for, new, synchronized, assert, default, goto,
package, this, boolean, double, if, private, throws, break, do,
implements, protected, throw, byte, else, import, public, transient,
case, enum, instanceof, return, try, catch, extends, interface,
short, void, char, finally, int, static, volatile, class, final,
long, super, while, const, float, native, switch.
```

Java code example

```
package herschel.ia.numeric; // herschel.ia.numeric: PACKAGE
public final class Complex1d // Complex1d: CLASS
    implements Serializable // Serializable: INTERFACE
{
    private transient double[][] _internal; // _internal: INSTANCE VARIABLE
    // writeObject: METHOD
    private void writeObject(ObjectOutputStream os) { // os = METHOD PARAMETER
        os.defaultWriteObject();
        os.writeInt(length());
        if (length()==0) return;
    }
}
```

```
    for (int i=0,n=length();i<n;i++) { // i = LOCAL VARIABLE
        os.writeDouble(_re[i]); os.writeDouble(_im[i]);
    }
}
```

Jython code example

```
# herschel.ia.dataset.gui = PACKAGE; DatasetInspector = CLASS
from herschel.ia.dataset.gui import DatasetInspector
# PI = CONSTANT
from java.lang.Math import PI
# testName = VARIABLE
testName = "chop_freq_test_2909_1832_1902_"
# load = METHOD
t2 = fits.load(myDir+testname+"PHOTF.fits").default
# MAX = NUMERIC FUNCTION
maxStep = MAX(step[step.where(step < 0xffff)])
# startEndTimes = FUNCTION; step, maxStep, time... = FUNCTION PARAMETERS
def startEndTimes(step, maxStep, time, startTime, endTime):
    for i in range(0, maxStep): # i = LOCAL VARIABLE
        temp=(step.where(step == i+1))
        endTime[i] = time[MAX(temp.toIntld())]
    return endTime
# len = FUNCTION
upper = len(startarr)
```